

Détection et analyse de l'impact des défauts de code dans les applications mobiles

Geoffrey Hecht

▶ **To cite this version:**

Geoffrey Hecht. Détection et analyse de l'impact des défauts de code dans les applications mobiles. Génie logiciel [cs.SE]. Université Lille 1 : Sciences et Technologies; Université du Québec à Montréal, 2016. Français. NNT : . tel-01418158

HAL Id: tel-01418158

https://theses.hal.science/tel-01418158

Submitted on 16 Dec 2016

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.

DÉTECTION ET ANALYSE DE L'IMPACT DES DÉFAUTS DE CODE DANS LES APPLICATIONS MOBILES

GEOFFREY HECHT
Université Lille 1 - Université du Québec à Montréal - Inria

Présentée et soutenue publiquement le 30 Novembre 2016

Devant le jury composé de :

Bram Adams (Polytechnique Montréal) - Rapporteur
Jean-Rémy Falleri (Enseirb-Matmeca) - Rapporteur
Sébastien Gambs (UQAM) - Examinateur
Pierre-Étienne Moreau (Loria - Mines Nancy) - Examinateur
Naouel Moha (UQAM) - Directrice
Romain Rouvoy (INRIA - Université Lille) - Directeur

Décembre 2016 – version 1.13

Geoffrey Hecht : *Détection et analyse de l'impact des défauts de code dans les applications mobiles,* , © Décembre 2016

“En essayant continuellement on finit par réussir. Donc : plus ça rate, plus on a de chance que ça marche.” — Les Shadoks

“They say a little knowledge is a dangerous thing, but it’s not one half so bad as a lot of ignorance.” — Terry Pratchett

“I may not have gone where I intended to go, but I think I have ended up where I needed to be.” — Douglas Adams

ABSTRACT

Mobile applications are becoming complex software systems that must be developed quickly and evolve continuously to fit new user requirements and execution contexts. However, addressing these constraints may result in poor low-level design choices, known as code smells. The presence of code smells within software systems may incidentally degrade their quality and performance, and hinder their maintenance and evolution. Thus, it is important to know this smells but also to detect and correct them. While code smells are well-known in object-oriented applications, their study in mobile applications is still in their infancy. Moreover there is a lack of tools to detect and correct them. That is why we present a classification of 17 code smells that may appear in Android applications, as well as a tool to detect and correct code smells on Android. We apply and validate our approach on large amounts of applications (over 3000) in two studies evaluating the presence and evolution of the number of code smells in popular applications. In addition, we also present two approaches to assess the impact of the correction of code smells on performance and energy consumption. These approaches have allowed us to observe that the correction of code smells is beneficial in most cases.

RÉSUMÉ

Les applications mobiles deviennent des logiciels complexes qui doivent être développés rapidement tout en évoluant de manière continue afin de répondre aux nouveaux besoins des utilisateurs ainsi qu'aux mises à jour régulières du contexte d'exécution, à savoir le système d'exploitation des périphériques mobiles. S'adapter à ces contraintes peut provoquer la présence de mauvais choix d'implémentation ou de conception que nous appelons défauts de code. La présence de défauts de code au sein d'une application peut dégrader la qualité et les performances d'une application en compliquant les tâches de maintenance et d'évolution. Il est alors important de connaître ces défauts mais aussi de pouvoir les détecter et les corriger, afin de permettre aux développeurs d'améliorer la qualité et les performances de leur application.

Les défauts de code sont bien connus pour les applications orientés objets et de nombreux outils permettent leurs détections, mais ce n'est pas le cas pour les applications mobiles. Les connaissances concernant les défauts de code dans les applications mobiles sont lacunaires, notamment au niveau de l'impact de la correction de ces défauts sur l'application, de plus les outils permettant la détection et la correction des défauts sont inexistants ou peu matures.

Nous présentons donc ici une classification de 17 défauts de code pouvant apparaître dans les applications Android, ainsi qu'un outil permettant la détection et la correction des défauts de code sur Android. Nous appliquons et validons notre méthode sur de grandes quantités d'applications (plus de 3000) dans deux études qui évaluent la présence et l'évolution du nombre des défauts de code dans des applications populaires. De plus, nous présentons aussi deux approches destinées à évaluer l'impact de la correction des défauts de code sur les performances et la consommation d'énergie des applications. Nous avons appliqué ces approches sur des applications libres ce qui nous a permis d'observer que la correction des défauts de code est bénéfique dans la plupart des cas.

PUBLICATIONS

Certaines des recherches présentées dans cette thèse sont déjà disponibles dans les publications suivantes :

Article de conférence

— Geoffrey Hecht, Benomar Omar, Romain Rouvoy, Naouel Moha et Laurence Duchien **Tracking the Software Quality of Android Applications Along Their Evolution** [36]. – *30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE. 2015*

— Geoffrey Hecht, Benjamin Jose-Scheidt, Clément De Figueiredo, Naouel Moha et Foutse Khomh **An empirical study of the impact of cloud patterns on quality of service** [33]. – *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference*

— Geoffrey Hecht, Naouel Moha et Romain Rouvoy **An Empirical Study of the Performance Impacts of Android Code Smells** [32]. – *IEEE/ACM International Conference on Mobile Software Engineering and Systems 2016*

— Antonin Carette,Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha et Romain Rouvoy **Investigating the Energy Impact of Android Smells** [11]. – *IEEE 24rd International Conference on Software Analysis, Evolution, and Reengineering 2017 (SANER)*

Article de journal

— Geoffrey Hecht, Romain Rouvoy, Naouel Moha et Javier Gonzalez-Huerta, **An Empirical Analysis of Android Code Smells** [37]. – *En cours de révision pour le journal Transactions on Software Engineering (TSE) - sous réserve d'acceptation*

Autres (rapport de recherche et posters)

— Geoffrey Hecht, Romain Rouvoy, Naouel Moha et Laurence Duchien **Detecting Antipatterns in Android Apps** [35]. – *Rapport de recherche RR-8693. INRIA Lille ; INRIA, mar. 2015*

— Geoffrey Hecht **An approach to detect Android antipatterns** [31]. – *Poster IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE) 2015*

— Geoffrey Hecht, Laurence Duchien, Naouel Moha et Romain Rouvoy **Detection of Anti-patterns in Mobile Applications** [34]. – *Poster COMPARCH 2014*

TABLE DES MATIÈRES

I INTRODUCTION 1

1 INTRODUCTION 3

1.1 Contexte du sujet . . . 3

1.1.1 Défauts de code . . . 3

1.1.2 Applications mobiles . . . 4

1.1.3 Android . . . 4

1.2 Motivation : améliorer la qualité des applications mobiles 5

1.3 Problématiques de la thèse . . . 6

1.3.1 Problématique 1 : Manque de spécifications des défauts de code spécifiques à Android . . . 7

1.3.2 Problématique 2 : Absence d'approche de détection de défauts de code pour les applications Android . . . 7

1.3.3 Problématique 3 : Impact de la présence des défauts de code méconnus dans les applications mobiles . . . 8

1.4 Contributions . . . 8

1.4.1 Une classification des défauts de code . . . 8

1.4.2 PAPRIKA : Détection des défauts de code dans les applications Android . . . 8

1.4.3 CURRY : Analyse de l'impact des défauts de code sur les performances . . . 9

1.4.4 HOT-PEPPER : Analyse de l'impact des défauts de code sur la consommation d'énergie . . . 9

1.4.5 Autre contribution : Impact des patrons infonuagiques sur la Qualité de Service (QdS) . . . 9

1.5 Plan de la thèse . . . 10

II ÉTAT DE L'ART 11

2 INFORMATIONS CONTEXTUELLES SUR LES MÉTRIQUES 13

2.1 Mesure de la performance . . . 13

2.2 Mesure de la consommation d'énergie . . . 15

3 ÉTAT DE L'ART 17

3.1 Définitions de défauts de code spécifiques aux applications mobiles . . . 17

3.2 Détection et correction des défauts de code spécifiques aux applications mobiles . . . 17

3.3 Étude sur la présence des défauts de code dans les applications mobiles . . . 18

3.4 Analyse de l'impact des défauts de code sur la performance et la consommation . . . 20

3.5 Conclusion . . . 22

III CONTRIBUTIONS 23
4 CLASSIFICATION DES DÉFAUTS DE CODE 25
4.1 Catégories et définitions des défauts de code 25
4.1.1 Défauts de code affectant la maintenance 25
4.1.2 Défauts de code affectant la performance : Micro-optimisations 26
4.1.3 Défauts de code affectant la gestion de la mémoire 28
4.1.4 Affichage de l'IHM 29
4.1.5 Blocage de processus 30
4.2 Classification des défauts de code 31
5 DÉTECTION DES DÉFAUTS DE CODE : PAPRIKA 37
5.1 Étape 1 : Génération du modèle de l'application 39
5.1.1 Entrées 39
5.1.2 Sortie 39
5.1.3 Description 39
5.1.4 Implémentation 41
5.2 Étape 2 : Conversion du modèle de PAPRIKA en graphe . 43
5.2.1 Entrée 43
5.2.2 Sortie 44
5.2.3 Description 44
5.2.4 Implémentation 44
5.3 Étape 3 : Détection des défauts de code 46
5.3.1 Entrées 46
5.3.2 Sortie 46
5.3.3 Description 46
5.3.4 Implémentation 47
5.4 Étape optionnelle 1 : Correction des défauts de code . . . 56
5.4.1 Entrées 56
5.4.2 Sortie 56
5.4.3 Description 56
5.4.4 Implémentation 57
5.4.5 Exemple de processeur IGS 58
5.5 Étape optionnelle 2 : Calcul de la qualité de l'application 59
5.5.1 Entrée 59
5.5.2 Sortie 59
5.5.3 Description 59
5.5.4 Implémentation 59
6 MESURE DE L'IMPACT DES DÉFAUTS DE CODE SUR LES PERFORMANCES 63
6.1 Étape 1 : Génération des versions 65
6.1.1 Entrées 65
6.1.2 Sortie 65
6.1.3 Description 65
6.1.4 Implémentation 65
6.2 Étape 2 : Instrumentation 66
6.2.1 Entrées 66

6.2.2 Sortie . . . 66
6.2.3 Description . . . 66
6.2.4 Implémentation . . . 66
6.3 Étape 3 : Traitement des métriques . . . 67
6.3.1 Entrées . . . 67
6.3.2 Sortie . . . 67
6.3.3 Description . . . 67
6.3.4 Implémentation . . . 69
7 MESURE DE L'IMPACT DES DÉFAUTS DE CODE SUR LA CONSOMMATION D'ÉNERGIE 71
7.1 Étape 1 : Génération des versions . . . 71
7.1.1 Entrées . . . 71
7.1.2 Sortie . . . 71
7.1.3 Description . . . 71
7.2 Étape 2 : Instrumentation . . . 71
7.2.1 Entrées . . . 71
7.2.2 Sortie . . . 73
7.2.3 Description . . . 73
7.2.4 Implémentation . . . 73
7.3 Étape 3 : Traitement des métriques . . . 75
7.3.1 Entrées . . . 75
7.3.2 Sortie . . . 75
7.3.3 Description . . . 75
7.3.4 Implémentation . . . 76

IV VALIDATIONS ET EXPÉRIMENTATIONS 77
8 VALIDATION DE PAPRIKA 79
8.1 Validation sur une application témoin . . . 79
8.1.1 Sujets . . . 79
8.1.2 Objet . . . 79
8.1.3 Conception expérimentale . . . 80
8.1.4 Analyse de sensibilité . . . 80
8.1.5 Comparaison avec d'autres outils . . . 81
8.1.6 Impact de l'obfuscation sur la détection des défauts de code . . . 82
8.1.7 Menaces à la validité . . . 83
8.2 Validation empirique par des experts . . . 84
8.2.1 Hypothèses . . . 84
8.2.2 Sujets . . . 84
8.2.3 Objets . . . 84
8.2.4 Conception expérimentale . . . 85
8.2.5 Processus de validation . . . 86
8.2.6 Résultats de la validation . . . 86
8.2.7 Menaces à la validité . . . 87
8.3 Validation empirique de la logique floue . . . 90

8.3.1 Analyse de sensibilité pour les défauts de code subjectifs . . . 90
9 DÉTECTION DES DÉFAUTS DE CODE SUR UNE FOULE D'APPLICATIONS MOBILES 93
9.1 Ensemble de données . . . 93
9.1.1 Catégorie . . . 94
9.1.2 Notes . . . 95
9.1.3 Taille . . . 95
9.1.4 Nombre de téléchargements . . . 96
9.2 Résultats . . . 96
9.2.1 Pouvons-nous observer une tendance dans la distribution des défauts de code au sein des applications Android ? (RQ1) . . . 97
9.2.2 Pouvons-nous utiliser la proportion de défauts de code d'une application pour évaluer la qualité d'une application Android ? (RQ2) . . . 101
9.2.3 Pouvons-nous observer une relation entre la présence des défauts de code et la popularité d'une application ? (RQ3) . . . 103
9.2.4 Est-ce que les classes qui héritent du cadre d'application Android (AFIC) ont tendance à contenir plus certains défauts de code que les autres classes ? (RQ4) . . . 104
9.2.5 Existe-t-il des relations entre les différentes occurrences de défauts de code ? (RQ5) . . . 105
9.3 Menaces à la validité . . . 107
9.4 Conclusion . . . 109
10 ÉVOLUTION DE DÉFAUTS DE CODE DANS LE TEMPS 111
10.1 Ensemble de données . . . 111
10.1.1 Catégorie . . . 112
10.1.2 Notes . . . 112
10.1.3 Nombre de téléchargements . . . 113
10.2 Résultats . . . 115
10.2.1 Pouvons-nous trouver des tendances d'évolution des défauts de code sur plusieurs versions d'une application ? (RQ1) . . . 115
10.2.2 Existe-t-il des relations entre les tendances d'évolution des défauts de code ? (RQ2) . . . 118
10.2.3 Pouvons-nous utiliser PAPRIKA pour suivre la qualité d'une application à travers le temps ? (RQ3) . . . 121
10.3 Menaces à la validité . . . 123
10.4 Conclusion . . . 124
11 IMPACT DES DÉFAUTS DE CODE SUR LA PERFORMANCE 125
11.1 Conception de l'étude . . . 126
11.1.1 Objets . . . 126
11.1.2 Sujets . . . 126

11.1.3 Conception . . . 127
11.1.4 Procédure . . . 127
11.1.5 Variables et hypothèses . . . 129
11.2 Résultats . . . 130
11.2.1 Aperçu des résultats . . . 130
11.2.2 Est-ce que la correction de *Internal Getter/Setter*, *Member Ignoring Method* ou *HashMap Usage* améliore les performances au niveau de l'affichage ? (RQ1) . . . 134
11.2.3 Est-ce que la correction de *Internal Getter/Setter*, *Member Ignoring Method* ou *HashMap Usage* améliore les performances au niveau de l'utilisation de la mémoire ? (RQ2) . . . 135
11.2.4 Est-ce que la correction des trois défauts de code améliore significativement les performances au niveau de l'affichage en comparaison de la correction d'un seul défaut de code ? (RQ3) . . . 136
11.2.5 Est-ce que la correction des trois défauts de code améliore significativement les performances au niveau de l'utilisation de la mémoire en comparaison de la correction d'un seul défaut de code ? (RQ4) 137
11.2.6 Est-ce que la correction des défauts de code a toujours un impact en utilisant le moteur ART plutôt que Dalvik ? (RQ5) . . . 137
11.3 Menaces à la validité . . . 137
11.4 Conclusion . . . 138
12 IMPACT DES DÉFAUTS DE CODE SUR LA CONSOMMATION D'ÉNERGIE 139
12.1 Conception de l'étude . . . 139
12.1.1 Objets . . . 140
12.1.2 Sujets . . . 141
12.1.3 Conception . . . 141
12.1.4 Procédure . . . 141
12.1.5 Variables et hypothèses . . . 142
12.2 Résultats . . . 143
12.2.1 Aperçu des résultats . . . 143
12.2.2 Est-ce que la correction de *Internal Getter/Setter* réduit la consommation d'énergie de l'application ? (RQ1) . . . 143
12.2.3 Est-ce que la correction de *Member Ignoring Method* réduit la consommation d'énergie de l'application ? (RQ2) . . . 144
12.2.4 Est-ce que la correction de *HashMap Usage* réduit la consommation d'énergie de l'application ? (RQ3) 144

12.2.5 Est-ce que la correction des trois défauts de code améliore significativement la consommation d'énergie en comparaison de la correction d'un seul défaut de code ? (RQ4) . . . 145
12.3 Menaces à la validité . . . 145
12.4 Conclusion . . . 146

V CONCLUSION ET PERSPECTIVES 147
13 CONCLUSION 149
14 PERSPECTIVES 153
14.1 Perspectives à court terme . . . 153
14.1.1 Amélioration de la classification . . . 153
14.1.2 Amélioration de PAPRIKA . . . 153
14.1.3 Impact des défauts de code . . . 154
14.2 Perspectives à long terme . . . 154
14.2.1 Correction de défauts de code complexe . . . 154
14.2.2 Détection et correction de défauts sur d'autres plate-formes . . . 155
14.2.3 Exploitation des bases de données . . . 155
14.2.4 Anti-patrons d'interaction avec des services distants 155
14.2.5 Collaboration avec des professionnels . . . 156

VI ANNEXE 157
A IMPACT DES PATRONS INFONUAGIQUES SUR LA QUALITÉ DE SERVICE (QDS) 159
A.1 Patrons choisis . . . 159
A.2 Application analysée . . . 159
A.3 Expérimentations . . . 161
A.4 Résultats et conclusions . . . 161

BIBLIOGRAPHIE 165

TABLE DES FIGURES

FIGURE 1 Architecture du système Android (HAL : Hardware Abstraction Layer). Source : https://source.android.com/source/index.html 6
FIGURE 2 Aperçu de l'approche PAPRIKA 38
FIGURE 3 Capture d'écran de la représentation graphe d'une partie de l'application Firefox 45
FIGURE 4 Exemple de boîtes de Tukey utilisées pour la détection de BLOB 47
FIGURE 5 Fonctions d'appartenances pour le BLOB 50
FIGURE 6 Exemple de fonction résultat pour BLOB 51
FIGURE 7 Valeurs résiduelles issues d'une régression linéaire 62
FIGURE 8 Aperçu de l'approche de mesure des performances 64
FIGURE 9 Processus de collecte des données de CURRY . . 67
FIGURE 10 Aperçu de l'approche de mesure de la consommation d'énergie 72
FIGURE 11 Schéma du montage nécessaire afin de mesurer la consommation d'énergie 74
FIGURE 12 Photo du montage nécessaire pour se connecter entre la batterie et le téléphone 75
FIGURE 13 Boîte de Tukey représentant la différence absolue entre les valeurs floues de PAPRIKA et le ratio oui/non 91
FIGURE 14 Distribution des applications selon leur catégorie 94
FIGURE 15 Distribution des applications selon les notes moyennes 95
FIGURE 16 Nombre moyen de classes en fonction du nombre de téléchargements 97
FIGURE 17 Régression linéaire entre le nombre total de défauts de code et le nombre de classes. Chaque point représente est une application. 100
FIGURE 18 Distribution du score de qualité dans notre ensemble de données 101
FIGURE 19 Boîte de Tukey des scores de chaque catégorie . . 102
FIGURE 20 Boîte de Tukey des scores pour chaque note . . . 103
FIGURE 21 Boîte de Tukey des scores pour chaque nombre de téléchargements 104
FIGURE 22 Distribution des applications Android en fonction de la catégorie 112
FIGURE 23 Distribution des applications Android en fonction de la note 113
FIGURE 24 Nombre moyen de classes d'une application en fonction du nombre de téléchargements 114

FIGURE 25 Nombre de versions en fonction du nombre de téléchargements 114
FIGURE 26 Évolution du score de l'application PICSART pour *Long Method* 116
FIGURE 27 Évolution du score de l'application FLIPBOARD pour *Complex Class* 117
FIGURE 28 Évolution du score de l'application FIREFOX pour *Leaking Inner Class* 117
FIGURE 29 Évolution du score de l'application de EVERNOTE pour *BLOB* 118
FIGURE 30 Évolution du score de l'application SKYPE pour *Leaking Inner Class* 119
FIGURE 31 Évolution du score de l'application IMO pour *BLOB*, *CC*, *LM* et *MIM* 120
FIGURE 32 Évolution du score de l'application RETRO CAMERA pour *BLOB* et *Long Method* 120
FIGURE 33 Évolution de la taille de TWITTER 122
FIGURE 34 Évolution du score de l'application TWITTER pour *Complex Class* 122
FIGURE 35 Utilisation mémoire moyenne de toutes les versions de *SoundWaves* sur les 120 dernières secondes 130
FIGURE 36 Sélection aléatoire entre un film et inventaire d'un client 162
FIGURE 37 Insertion d'un film avec Local Sharding-Based Router et Priority Queue 163

LISTE DES TABLEAUX

Tableau 1 Classification des défauts de code 34
Tableau 2 Liste des propriétés des entités de PAPRIKA . . . 40
Tableau 3 Liste des relations entre entités de PAPRIKA . . . 41
Tableau 4 Liste de métriques des entités de PAPRIKA . . . 42
Tableau 5 Relation entre défauts de code et nombre d'entités 60
Tableau 6 Correspondance entre Cohen's d to Cliff's δ. . . 68
Tableau 7 Applications utilisées pour le calcul des seuils pour la validation sur l'application témoin . . . 80
Tableau 8 Analyse de sensibilité sur l'application témoin . 81
Tableau 9 Impact de l'obfuscation sur la détection des défauts de code 82
Tableau 10 Liste des applications libres utilisées pour la validation 85
Tableau 11 Exactitude, Précision, Rappel, et F1-mesure de PAPRIKA pour tous les défauts de code 88
Tableau 12 Exactitude, Précision, Rappel et F1-mesure de PAPRIKA pour les défauts de code subjectifs et objectifs 91
Tableau 13 Analyse de sensibilité pour les défauts de code subjectifs 92
Tableau 14 Proportion de défauts de code dans toutes les applications 98
Tableau 15 Nombre et ratio de AFIC affectés par les défauts de code 106
Tableau 16 Relations entre les défauts de code 108
Tableau 17 Versions expérimentales 127
Tableau 18 Nombre de défauts de code corrigés, nombre d'entités concernées et nombre moyen d'invocations des défauts de code durant le scénario 128
Tableau 19 Valeurs moyennes des métriques sur 60 expérimentations 131
Tableau 20 Pourcentages de différence des métriques entre les versions de SoundWaves 132
Tableau 21 Mann-Whitney U et Cliff δ (f pour Faible, M pour Moyen, F pour Fort) pour toutes les versions de SoundWaves 132
Tableau 22 Pourcentages de différence des métriques entre les versions de Terminal Emulator 133
Tableau 23 Mann-Whitney U et Cliff δ (f pour Faible) pour toutes les versions de Terminal Emulator 134
Tableau 24 Nombre de classes, de méthodes et de défauts de code corrigés dans chaque application 140

Tableau 25 Nombre moyen d'étapes, temps d'attente et d'éxécution moyen pour chaque scénarios 142
Tableau 26 Nombre moyen de défauts invoquées durant l'exécution du scénario 142
Tableau 27 Consommation moyenne d'énergie (en Joules) de chaque application 143
Tableau 28 Pourcentage de différence de la consommation moyenne d'énergie (en J) de chaque version par rapport à V_0 144
Tableau 29 Cliff δ pour toutes les versions,V_m représente la meilleure version parmi l'ensemble que nous comparons à V_4 (f pour Faible, M pour Moyen, F pour Fort) 144
Tableau 30 Différentes versions 160

LISTE DES FRAGMENTS

Fragment 1 Fichier FCL pour la détection de BLOB 48
Fragment 2 Requête Cypher pour la détection de BLOB avec un seuil à HIGH 51
Fragment 3 Requête Cypher pour la détection de SAK avec un seuil à HIGH 51
Fragment 4 Requête Cypher pour la détection de LM avec un seuil à HIGH 51
Fragment 5 Requête Cypher pour la détection de CC avec un seuil à HIGH 52
Fragment 6 Requête Cypher pour la détection de IGS . . . 52
Fragment 7 Requête Cypher optimisée pour la détection de IGS sur un ensemble d'applications 52
Fragment 8 Requête Cypher pour la détection de MIM . . . 52
Fragment 9 Requête Cypher pour la détection de IOD . . . 53
Fragment 10 Requête Cypher pour la détection de NLMR . . 53
Fragment 11 Requête Cypher pour la détection de LIC . . . 53
Fragment 12 Requête Cypher pour la détection de UCS . . . 54
Fragment 13 Requête Cypher pour la détection de HMU . . 54
Fragment 14 Requête Cypher pour la détection de UIO . . . 54
Fragment 15 Requête Cypher pour la détection de IWR . . . 54
Fragment 16 Requête Cypher pour la détection de UHA . . . 55
Fragment 17 Requête Cypher pour la détection de HAS avec un seuil à HIGH 55
Fragment 18 Requête Cypher pour la détection de HSS avec un seuil à HIGH 55
Fragment 19 Requête Cypher pour la détection de HBR avec un seuil à HIGH 56
Fragment 20 Extrait du processeur corrigeant les IGS 58

ACRONYME

OO Orienté Objet

IHM Interface Homme Machine

AFIC Android Framework Inherited Classes

QdS Qualité de Service

BLOB Blob Class

SAK Swiss Army Knife

LM Long Method

CC Complex Class

IGS Internal Getter/Setter

MIM Member Ignoring Method

IOD Init OnDraw

NLMR No Low Memory Resolver

LIC Leaking Inner Class

UCS Unsuited LRU Cache Size

HMU Hashmap Usage

UIO UI Overdraw

IWR Invalidate Without Rect

UHA Unsupported Hardware Acceleration

HAS Heavy AsyncTask

HSS Heavy Service Start

HBR Heavy BroadcastReceiver

AST Abstract Syntax Tree

HR Hypothèse de Recherche

Première partie

INTRODUCTION

1 INTRODUCTION

Dans ce chapitre, nous introduisons le contexte nécessaire à la compréhension du sujet ainsi que les motivations et problématiques qui ont justifiées l'élaboration de cette thèse. Nous finissons par présenter les principales contributions de ces trois années.

1.1 CONTEXTE DU SUJET

Dans cette section, nous présentons le contexte nécessaire pour aborder ce mémoire, en particulier les expressions-clefs qui composent le titre de cette thèse, à savoir *défauts de code* et *application mobile*. Nous présentons aussi brièvement le système mobile sur lequel nous avons travaillé tout au long de cette thèse, c'est-à-dire le système Android.

1.1.1 *Défauts de code*

Les défauts de code [22](*Code smells* en Anglais) sont des problèmes d'implémentation qui proviennent de mauvais choix conceptuels. Dans leur définition d'origine, ils sont qualifiés de symptômes de problèmes profonds qui peuvent rendre plus complexes la maintenance et l'évolution d'un logiciel. Ils sont des violations de principes fondamentaux de conception qui ont un impact négatif sur la qualité du logiciel [76] et tendent à contribuer à la dette technique des logiciels et à générer des coûts de développement supplémentaires. Par exemple, un défaut de code peut être une méthode trop longue [22] qui, par conséquent, sera complexe à comprendre et à maintenir. Par la suite, d'autres défauts de code ont aussi été reconnus comme pouvant affecter d'autres aspects comme la performance [10, 12] ou la consommation d'énergie [25, 85]. Ils peuvent donc générer des usages inutiles de certaines ressources (comme le processeur, la mémoire, la batterie, etc.) [10] et, par conséquent, empêcher le déploiement de solution efficace et durable. On retrouvera ici des défauts de code comme l'utilisation de condition mutuellement exclusive [85] dans un SI, qui retournera donc toujours VRAI et donnera donc lieu à l'exécution d'instructions inutiles pour évaluer la condition. Notons ici que les défauts de code ne sont généralement pas des bogues, c'est-à-dire qu'ils n'entraînent pas forcément la terminaison inattendue de l'application. Toutefois, ils participent à dégrader la qualité des applications. Ajoutons aussi, dans un souci de précision, que dans le cadre de cette thèse, nous considérons aussi quelques défauts de conception [9] (ou anti-patrons) qui sont des mauvaises pratiques de plus haut niveau. Les défauts de conception sont d'ailleurs souvent composés de défauts

de code [61]. On y retrouve par exemple, le *BLOB* [9] qui est une classe complexe monopolisant plusieurs processus importants de l'application et qui est, par conséquent, difficile à maintenir et faire évoluer. Toutefois, dans la suite de cette thèse, pour des raisons de lisibilité, nous ne ferons plus la distinction entre ces deux types de défauts et utiliserons défauts ou défauts de code pour les qualifier. Cette non-distinction est aussi courante dans la littérature, les auteurs utilisant l'un ou l'autre des termes [45, 49, 80, 84].

1.1.2 *Applications mobiles*

Par applications mobiles, nous désignons les logiciels spécifiquement développés pour être exécutés sur périphérique mobile comme un téléphone ou une tablette. Ces périphériques ont le plus souvent un système d'exploitation spécifiquement conçu pour les appareils mobiles, comme par exemple Android, iOS ou Windows Phone. Ces applications sont généralement développées en utilisant des langages orientés objets multi plates-formes et courants (Java, C#, Objective-C), toutefois elles doivent être adaptées aux systèmes sur lesquelles elles tournent, notamment en utilisant le cadre d'applications du système, mais aussi prendre en compte les contraintes des périphériques sur lesquelles elles tournent (quantité de mémoire, puissance de calcul et autonomie limitée, disponibilité et qualité du réseau non garantie...). Par conséquent, les applications mobiles ont tendance à être plus légères et à utiliser plus massivement le cadre d'applications, des librairies externes ou même la réutilisation de classes par rapport aux applications non-mobiles [60, 71, 89].

Ces applications sont généralement distribuées via des plates-formes de téléchargement proposant des interfaces intégrées aux systèmes d'exploitation des périphériques. Ces plates-formes distribuent le code binaire des applications et ne permettent généralement pas l'accès au code source des applications.

Les applications mobiles connaissent un succès retentissant ces dernières années, notamment avec la démocratisation des téléphones intelligents, à tel point qu'on estime que plus de 168 milliards d'applications seront téléchargées en 2017 [1].

1.1.3 *Android*

Pour aborder les applications mobiles, nous avons choisi de nous baser sur les applications du système Android (toutefois notons que certains des travaux présentés ici sont en cours de réplication sur iOS pour un autre projet). Android est un système d'exploitation, développé par Google, pour les appareils mobiles. Il est basé sur un noyau Linux et les

1. Number of mobile app downloads worldwide from 2009 to 2017 : https://www.statista.com/statistics/266488/forecast-of-mobile-app-downloads

applications Android sont habituellement développées en Java. Nous avons choisi de travailler sur Android car c'est un système d'exploitation libre. C'est aussi le système mobile le plus populaire avec environ plus de 85% de parts de marché dans le monde pour les téléphones intelligents [2]. De plus, ce système nous offrait l'opportunité d'accéder à un grand nombre d'applications plus simplement que ses concurrents. On peut aussi constater au travers de notre état de l'art (chapitre 3) que la grande majorité de la communauté scientifique utilise aussi ce système, sans doute pour les mêmes raisons. L'architecture Android est résumée dans la figure 1. Dans le cadre de cette thèse, deux couches nous intéressent tout particulièrement. Tout d'abord, le cadre d'application Android (en vert) sur lequel repose l'ensemble des applications. Ainsi la plupart des défauts de code que nous abordons dans cette thèse sont liés à ce cadre d'application, notamment parce qu'ils apparaissent dans des classes qui héritent de ce cadre. De plus, comme nous l'avons évoqué précédemment, les applications sont fortement dépendantes de ce cadre [60, 71, 89]. Il est donc possible que ce cadre d'application ait une influence sur la présence et la répartition des défauts de code des applications. La deuxième couche qui nous intéresse est la couche du moteur d'exécution (en jaune) qui représente la machine virtuelle qui va exécuter le code. Bien que la plupart des applications Android utilisent le langage Java, ce système n'utilise pas la machine virtuelle Java classique mais des machines virtuelles personnalisées, à savoir Dalvik et ART [3]. Ces machines virtuelles sont optimisées pour répondre aux besoins des périphériques mobiles, en particulier elles se basent sur l'utilisation de registres plutôt que de piles afin de diminuer la quantité de mémoire utilisée. Les premières versions d'Android utilisent Dalvik alors que depuis la version 5.0 (Lollipop), ART est l'environnement par défaut. La différence principale entre les deux étant que Dalvik utilise de la compilation à la volée alors qu'ART utilise de la compilation anticipée. ART améliore les performances en terme de temps d'exécution, diminue la consommation d'énergie et possède un ramasse-miettes plus efficace. De plus, ART fournit aussi un environnement de débogage amélioré. Ces différences entre les machines virtuelles ont aussi un impact sur la présence de certains défauts de code liés à la performance comme nous le verrons dans le chapitre 4.

1.2 MOTIVATION : AMÉLIORER LA QUALITÉ DES APPLICATIONS MOBILES

Comme nous l'avons vu précédemment, les applications mobiles connaissent un succès grandissant avec des milliards d'applications téléchargées chaque

2. Flat Smartphone Growth Projected for 2016 as Mature Markets Veer into Declines, According to IDC : http://www.idc.com/getdoc.jsp?containerId=prUS41699616

3. ART and Dalvik : https://source.android.com/devices/tech/dalvik

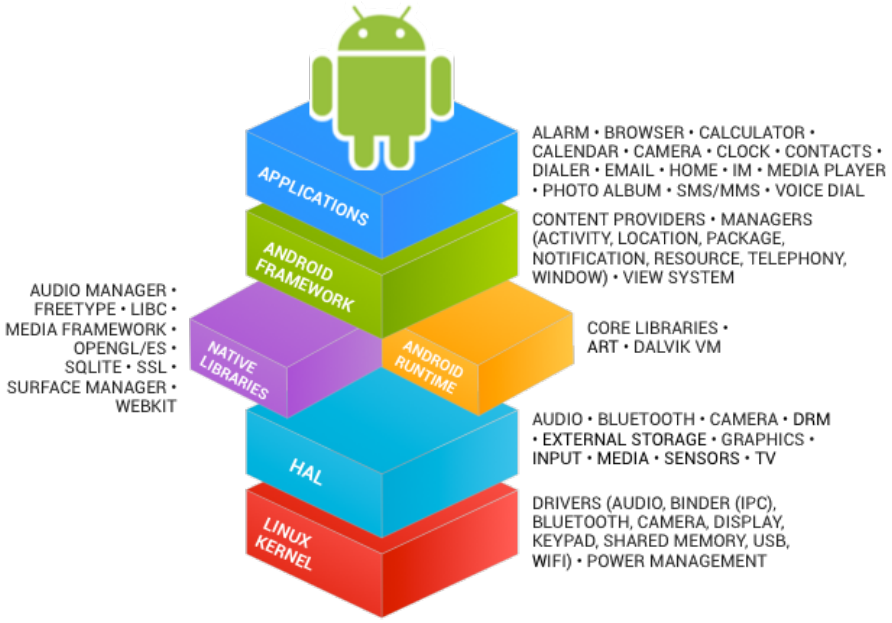


FIGURE 1 – Architecture du système Android (HAL : Hardware Abstraction Layer). Source : https://source.android.com/source/index.html

année. Les applications mobiles sont aussi mises à jour plus régulièrement que les applications de bureau pour répondre aux demandes du marché [59]. Or, la fréquence d'apparitions des défauts de code tend à augmenter lorsque les applications connaissent des changements de code fréquents sur de courtes périodes de temps [80]. Une étude de Gartner [42] a aussi démontré que la qualité de l'application a une importance majeure dans son succès. Ainsi, on y apprend que 51% des utilisateurs d'applications mobiles déclarent avoir abandonné un achat en ligne car l'application était trop lente, mais aussi que 79% des utilisateurs n'essayeront pas plus de deux fois une application ne fonctionnant pas correctement avant de passer à la concurrence. Dès lors, il devient essentiel pour les développeurs d'améliorer la qualité de leurs applications en utilisant tous les moyens à leurs dispositions. Puisque que la présence de défauts de code a tendance à réduire la qualité des applications, il est alors particulièrement important de les connaître et de pouvoir les détecter pour les applications mobiles qui sont mises à jour régulièrement.

1.3 PROBLÉMATIQUES DE LA THÈSE

Dans cette section, nous abordons les trois problématiques qui nous ont permis structurer l'ensemble de cette thèse. À savoir le manque de définitions des défauts de code spécifiques à Android, l'absence d'approche de détection de défauts code pour les applications Android, et la

méconnaissance de l'impact de la présence des défauts de code dans les applications.

1.3.1 *Problématique 1 : Manque de spécifications des défauts de code spécifiques à Android*

La présence des défauts de code orientés objets dans les applications mobiles a fait l'objet de plusieurs études [45, 49, 84], toutefois ce n'est pas le cas pour les défauts de code spécifiques à Android. À notre connaissance, seule une publication par Reimann *et al.* [67, 68] s'est penchée sur le problème en fournissant un catalogue de 30 *quality smells* reportant des défauts de code mais aussi des mauvaises pratiques diverses dans la gestion des ressources, du réseau ou encore l'interface graphique. Ce catalogue fournit aussi une classification en fonction de la qualité affectée (accessibilité, efficacité, stabilité, sécurité...) et du contexte (réseau, interface graphique, base de données...). Ce fut une ressource précieuse pour nous mais toutefois insuffisante pour pouvoir prétendre à une réelle connaissance des défauts de code spécifiques à Android. On peut d'ailleurs étendre cette observation aux autres systèmes puisqu'ils ne sont pas considérés dans la littérature concernant les défauts de code.

1.3.2 *Problématique 2 : Absence d'approche de détection de défauts de code pour les applications Android*

De nombreux outils permettent la détection de code orientés objets sur Android [3, 61, 79], mais ils ne permettent malheureusement pas la détection de défauts de code spécifiques aux application mobiles. En plus des raisons techniques (la plupart de ces approches existaient avant l'avènement des applications mobiles), cela est sans doute lié à la problématique 1, puisqu'il est compliqué de détecter ce qui n'est pas clairement défini. Le catalogue précédemment cité est bien supposé être accompagné d'un outil permettant la détection et la correction des mauvaises pratiques [68] toutefois, dans la pratique, nous n'avons pas réussi à le faire fonctionner sur une application Android. De plus, puisqu'aucune étude ou validation n'a été menée avec cet outil, il n'existe pas de preuve que la totalité des mauvaises pratiques présentées soit détectable avec l'approche proposée. Toutefois, une inspection du code source de l'application semble montrer que 6 des 30 mauvaises pratiques sont disponibles dans l'outil [4]. Au début de cette thèse, il ne semblait donc pas exister d'approche mature permettant la détection des défauts de code spécifiques à Android.

4. SVN de l'outil Refactory : goo.gl/2IL2KA

1.3.3 *Problématique 3 : Impact de la présence des défauts de code méconnus dans les applications mobiles*

Cette problématique est elle aussi probablement la conséquence des deux problématiques précédentes. Il existe toutefois certains travaux évaluant l'impact de mauvaises pratiques semblables à des défauts de code sur la consommation d'énergie ou le temps d'exécution des méthodes [43, 62, 66, 78]. Toutefois, ces études se limitent généralement à exécuter la mauvaise pratique en boucle (plusieurs millions de fois) dans une application jouet et à comparer les résultats obtenus lorsque la mauvaise pratique n'est pas présente. Bien que très utile pour prouver l'existence d'un impact, nous estimons que cela est insuffisant pour motiver les développeurs à corriger ces mauvaises pratiques. En effet, cela ne prouve pas que les effets seront toujours observables dans des applications réelles et lors d'une utilisation classique.

1.4 CONTRIBUTIONS

Dans cette section, nous présentons brièvement les contributions que nous avons apportées afin de résoudre nos problématiques. Afin de répondre à la problématique 1, nous avons tout d'abord proposé une classification de 17 défauts de code. Tandis que vous avons ensuite proposé l'approche outillée PAPRIKA pour détecter et corriger les défauts de code et ainsi résoudre la problématique 2. Nous avons aussi proposé deux outils, CURRY et HOT-PEPPER, pour mesurer les performances et la consommation d'énergie afin de répondre à la problématique 3. Enfin, nous présentons aussi une contribution annexe de la thèse sur les patrons infonuagiques à la fin de cette section.

1.4.1 *Une classification des défauts de code*

Afin de répondre à notre première problématique, nous avons créé une classification regroupant 17 types de défauts de codes différents pouvant apparaître dans les applications Android. Quatre de ces défauts de code sont orientés objets et 13 d'entre eux sont spécifiques à Android. À notre connaissance, huit des ces défauts de code n'apparaissaient pas dans la littérature scientifique avant nos travaux. Cette classification repose aussi sur cinq critères orthogonaux permettant de mieux comprendre et d'ouvrir la voie à la détection des différents défauts. Elle est présentée plus en détails dans le chapitre 4.

1.4.2 PAPRIKA *: Détection des défauts de code dans les applications Android*

Afin de répondre à la deuxième problématique, nous avons conçu PAPRIKA, qui est une approche outillée permettant la détection de défauts

de code orientés objets et spécifiques à Android. Elle est actuellement capable de détecter tous les défauts de code de la classification. C'est une approche flexible pour laquelle il est facile d'écrire des requêtes afin de détecter de nouveaux défauts de code. De plus, cette approche profite des connaissances accumulées lors des précédentes analyses pour calibrer les seuils de détection lorsque des métriques sont utilisées. L'approche propose aussi la correction automatique de trois des défauts de code. Cette approche est présentée plus en détails dans le chapitre 5. Elle nous a permis d'obtenir les résultats d'analyse de larges volumes d'applications qui sont présentées dans le chapitre 9 et le chapitre 10.

1.4.3 CURRY : *Analyse de l'impact des défauts de code sur les performances*

Afin de répondre à la troisième problématique, nous nous sommes penchés sur l'impact des défauts de code sur les performances. Nous proposons donc une approche nommée CURRY, qui permet de comparer les performances de plusieurs versions d'une même application. Ces versions diffèrent par la présence ou non de défauts de code, il devient alors possible de déterminer l'impact de la présence des défauts de code. Cette approche est particulière car elle fournit des résultats au niveau global de l'application en s'intéressant à des métriques liées à l'expérience utilisateur et non pas seulement à des métriques locales. À l'aide de tests statistiques, CURRY permet aux développeurs de choisir la version la plus performante de ce point de vue. Cette approche est présentée plus en détails dans le chapitre 6 alors que les résultats d'une étude l'utilisant sont présentés dans le chapitre 11.

1.4.4 HOT-PEPPER : *Analyse de l'impact des défauts de code sur la consommation d'énergie*

HOT-PEPPER est similaire à l'approche précédente mais pour évaluer la consommation d'énergie d'une application cette fois. Bien que reposant sur les mêmes principes et utilisant les mêmes versions, cette approche nécessite en plus l'utilisation d'un ampèremètre qui se branche entre le téléphone et la batterie afin de mesurer précisément la consommation d'énergie. Cette approche est présentée plus en détails dans le chapitre 7 alors que les résultats d'une étude l'utilisant apparaissaient dans le chapitre 12.

1.4.5 *Autre contribution : Impact des patrons infonuagiques sur la Qualité de Service (QdS)*

Au début de la thèse, nous avons travaillé sur les patrons et antipatrons infonuagiques. Cela a donné lieu à une publication [33] et une

présentation à la conférence internationale CloudCom'14 [5]. Nous avons ensuite réorienté la thèse vers les applications mobiles. Nous avons effectué ce choix car le code source des applications infonuagiques est difficile à obtenir, et il nous semblait donc plus intéressant d'apporter des contributions dans le domaine des applications mobiles. Il aurait été plus intéressant pour nous de concentrer nos efforts sur des anti-patrons ou défauts de code comme pour les applications mobiles mais malheureusement ceux-ci ne sont pas bien définis dans la littérature. Bien que le domaine soit différent, il est à noter que cette publication fut utile pour la poursuite de la thèse. En effet, la méthode d'analyse des résultats utilisée fut réutilisée pour les publications sur l'impact des défauts de code Android sur les performances et sur la consommation d'énergie. Nous présentons cette contribution dans l'annexe A.

1.5 PLAN DE LA THÈSE

La suite de cette thèse est divisée en quatre parties. La partie ii regroupe des informations contextuelles et l'état de l'art. La partie iii présente plus en détails les contributions de cette thèse. La partie iv montre les validations et résultats que nous avons obtenus grâce à nos contributions. Enfin, dans la partie v, nous présentons nos conclusions ainsi que les perspectives de recherche.

5. http://2014.cloudcom.org

Deuxième partie

ÉTAT DE L'ART

2

INFORMATIONS CONTEXTUELLES SUR LES MÉTRIQUES

Ce chapitre fournit des informations contextuelles nécessaires à la compréhension des approches que nous proposons. Les informations fournies ici complètent donc le contexte du chapitre 1 tout en étant plus spécifique aux travaux de cette thèse. En particulier, nous présentons ici les métriques que nous utilisons pour mesurer les performances et la consommation d'énergie.

2.1 MESURE DE LA PERFORMANCE

Afin de mesurer l'impact de la présence des défauts de code, nous avons choisi d'utiliser des métriques liées à l'expérience utilisateur. Des ressources sont déjà disponibles pour évaluer l'impact de cette présence à un niveau local [4, 5], c'est-à-dire au niveau du temps d'exécution d'une méthode. Ces tests sont aussi exécutés sur des applications jouets qui appellent la même méthode en boucle. Bien qu'étant utile pour prouver l'impact de la correction des défauts de code, cela reste insuffisant pour prouver que cet impact sera visible sur une application en condition d'utilisation réelle. Nous avons donc conçu l'approche CURRY afin de pouvoir déterminer l'impact de la correction sur l'expérience utilisateur. Cette approche décrite au chapitre 6 utilise les métriques suivantes :

Temps d'affichage des trames : Le temps d'affichage d'une trame est le temps pris par la système pour dessiner une trame sur l'écran. Par défaut, les systèmes Android fournissent 60 trames par seconde, toutefois cette valeur peut baisser si le calcul des trames est trop long. Dans ce cas, des trames ne sont pas affichées, ce qui peut avoir pour conséquence une sensation de lenteur et de manque de réactivité de l'application pour l'utilisateur. C'est d'ailleurs un ressenti qui apparaît régulièrement dans les commentaires utilisateurs [46]. Afin d'atteindre 60 trames par seconde, toutes les entrées, calculs, opérations réseaux et opérations d'affichage doivent s'effectuer en moins de 16 ms par trames [29, 77]. Par conséquent, le temps d'affichage des trames est un indicateur global de la performance d'une application, qui peut affecter l'expérience utilisateur. De nombreux types d'optimisations peuvent potentiellement améliorer ce temps, en particulier celles qui concernent la correction des défauts de code comme nous le montrons dans le chapitre 11.

Nombre de trames différées : Dans le cas où une trame prend plus de 16 ms à être affichée, l'affichage de la trame est alors différée. En effet, le tampon de trame est envoyé toutes les 16 ms, ce qui signifie

que la trame différée ne sera pas affichée (on conserve l'ancienne trame) avant 32 ms même si elle ne prend que 17 ms à être totalement calculée. Une application dont toutes les trames pendraient 17 ms à être calculées, n'afficherait alors que 30 trames par seconde plutôt que les 60 recommandées [29, 77]. Une trame peut être différée plusieurs fois de suite si, par exemple, elle prend plus de 32ms ou 48ms à être affichée. Cette métrique est directement dérivée du temps de trame, elle permet de fournir des informations supplémentaires sur les lenteurs que pourraient ressentir un utilisateur.

Quantité de mémoire utilisée : La quantité de mémoire utilisée (en Ko) par une application peut affecter le système au complet. En effet, la mémoire est souvent limitée sur les appareils mobiles (512 Mo est encore une configuration courante) alors qu'ils doivent faire fonctionner plusieurs applications à la fois. Quand le système ne dispose plus de la mémoire nécessaire pour faire fonctionner les applications actives ou les services internes, il doit alors libérer de la mémoire en déclenchant le ramasse-miettes. Si le ramasse-miettes n'est pas capable de libérer assez de mémoire, alors le système commence à tuer les processus des applications les moins récemment utilisées. Les applications concernées nécessitent alors un redémarrage à froid lors de la prochaine utilisation, qui est beaucoup plus long qu'un démarrage à chaud et affecte donc l'expérience utilisateur. Des quantités excessives de mémoire utilisées sont d'ailleurs régulièrement reportées dans les commentaires des utilisateurs [46]. Il est donc recommandé de limiter la quantité de mémoire utilisée de chaque application afin de permettre à l'utilisateur de changer rapidement d'applications actives [29, 40].

Nombre d'appels au ramasse-miettes : Le ramasse-miettes permet le nettoyage et la gestion automatique de la mémoire, il permet de libérer la mémoire des objets qui ne sont plus utilisés par une application. Avec la machine virtuelle Dalvik, un appel au ramasse-miettes peut prendre jusqu'à 20 ms sur des périphériques considérés comme rapides [29]. Ainsi, des appels répétés au ramasse-miettes dus à une mauvaise gestion de la mémoire par une application peuvent dégrader les performances du système. En particulier, cela peut mener à une augmentation du nombre de trames différées et par conséquent affecter l'expérience utilisateur [29, 40].

Lien avec les défauts de code : Comme nous le verrons au chapitre 4, nous disposons de défauts de code qui peuvent affecter le temps d'exécution d'une méthode (*Internal Getter/Setter* ou *Member Ignoring Method* par exemple). Ce type de défauts peut donc avoir un impact sur le temps d'affichage d'une trame, et par conséquent le nombre de trames différées, puisque le temps d'exécution des méthodes est inclus dans ce temps d'affichage (il inclut le temps de toutes les méthodes appelées sur le processus principal).

D'autres défauts comme *Hashmap Usage* sont réputés pour directement augmenter la quantité de mémoire utilisée ou le nombre d'appels

au ramasse-miettes. De plus, des effets de bord sur l'une ou l'autre des métriques sont théoriquement possibles pour l'ensemble des défauts de code.

2.2 MESURE DE LA CONSOMMATION D'ÉNERGIE

Pour mesurer la consommation d'énergie, nous utilisons un ampèremètre qui nous fournit des valeurs instantanées d'intensité (I) de courant ainsi que les spécifications de la batterie sur le voltage (V). Le voltage est supposé constant en sortie de la batterie, toutefois l'intensité varie au cours du temps. Ainsi pour obtenir la consommation d'énergie globale sur une période de temps donnée, nous utilisons donc l'équation suivante :

$$\mathsf{E}_{\mathsf{globale}} = \sum (\mathsf{V} * \Delta\mathsf{t} * \mathsf{I}_{\mathsf{moyenne}}) \tag{1}$$

$\Delta\mathsf{t}$ représente la durée entre deux mesures de l'ampèremètre. Le résultat est une quantité d'énergie en Joules.

3

ÉTAT DE L'ART

Dans ce chapitre, nous présentons un état de l'art sur tous les sujets relatifs à cette thèse, en particulier sur les définitions de défauts de code spécifiques aux applications mobiles, la détection et la correction des défauts de code ainsi que l'analyse de l'impact des défauts de code sur les performances et la consommation d'énergie. Notons que nous nous intéressons ici majoritairement aux défauts de code spécifiques aux applications mobiles mais que nous aborderons aussi les défauts de code orientés objets. En effet, les publications dédiées uniquement aux applications mobiles sont assez rares. Mannan *et al.* ont d'ailleurs étudié les publications concernant ces défauts dans les conférences les plus réputées de la communauté du génie logiciel (ICSE, FSE, OOPSLA/SPLASH, ASE, ICSM/ICSME, MSR, et ESEM) entre 2008 (début d'Android) et 2015 [49]. Ils n'ont trouvé uniquement que cinq papiers sur les défauts de code qui évoquent la plate-forme Android contre 52 avec des défauts de code sur les applications de bureau uniquement. Il n'existe aucune publication concernant les défauts de code sur iOS ou Windows Phone dans ces conférences. Cela montre bien que les connaissances que nous possédons sur les défauts de code dans les applications mobiles sont pour le moment lacunaires.

3.1 DÉFINITIONS DE DÉFAUTS DE CODE SPÉCIFIQUES AUX APPLICATIONS MOBILES

Comme vu précédemment, en dehors de notre approche, nous n'avons trouvé que les travaux de Reimann *et al.* [68] qui ont pour objectif la définition de nouveaux défauts de code sur Android. En revanche, il existe de nombreuses ressources en ligne [5, 30, 47, 48, 51–53, 55, 57, 63], comme des blogues de développeurs ou la documentation Android, qui listent des mauvaises pratiques permettant d'inférer des défauts de code. Ce sont d'ailleurs ces ressources qui ont permis d'alimenter le catalogue de ces travaux ainsi que notre classification. Nous décrivons plus en détails 13 défauts de code spécifiques à Android dans le chapitre 4.

3.2 DÉTECTION ET CORRECTION DES DÉFAUTS DE CODE SPÉCIFIQUES AUX APPLICATIONS MOBILES

Concernant la littérature scientifique, une fois de plus seul Reimann *et al.* [68] semblent proposer une approche de détection et de correction des défauts de code spécifiques à Android. Nous n'avons pas réussi à tester l'outil, toutefois théoriquement il fonctionne en analysant et

modifiant le modèle *Eclipse Modeling Framework* (EMF) d'une application. Il est nécessaire d'avoir accès au code source de l'application et d'avoir généré son modèle EMF. La correction des défauts de code nécessite alors l'implémentation en Java de la détection et de la correction des défauts en parcourant ce modèle. Pour le moment, 6 des 30 mauvaises pratiques proposées semblent être disponibles dans l'outil. Cette détection ne semble pas reposer sur des métriques sujettes à interprétation (comme la longueur ou la complexité) mais sur la présence ou non de certains éléments dans le modèle. En comparaison, notre approche PAPRIKA ne nécessite pas le code source pour la détection (mais tout de même pour la correction). Nous stockons le modèle de l'application dans une base de données, ce qui nous permet de proposer des mécanismes statistiques et de la logique floue pour la détection lorsqu'elle se base sur des métriques. De plus, nous proposons déjà la détection de 17 défauts de code et l'ajout de nouveaux défauts se fait très facilement en utilisant le langage de requête de la base de données qui fait office de langage spécifique au domaine. PAPRIKA permet aussi l'analyse à large échelle de milliers d'applications à la fois plutôt qu'une analyse par projet.

L'environnement de développement Android Studio intègre par défaut l'outil LINT [3]. C'est un outil permettant de détecter des mauvaises pratiques en analysant des projets. Par exemple, il est ainsi capable de détecter des ressources erronées ou mal optimisées, des erreurs de syntaxe dans les fichiers XML ou encore des défauts de code. LINT dispose d'un analyseur statique de code ce qui lui permet donc de détecter des défauts de code orientés objets et spécifiques à Android. On retrouve ainsi une détection de variantes des défauts de code comme *Leaking Inner Class*, *Internal Getter/Setter*, *Init OnDraw*, et *OverDraw* que nous présentons dans le chapitre 4. Une fois de plus contrairement à PAPRIKA, l'ajout de nouvelles règles de détection nécessitent d'implémenter des algorithmes en Java. De plus, les règles de détection utilisent toutes des seuils prédéfinis (mais modifiables par l'utilisateur) qui peuvent sembler arbitraire (par exemple, une méthode est considérée comme longue si elle a plus de 80 lignes de code). Ici aussi, LINT [3] propose une analyse par projet unique, et ne permet donc pas facilement d'analyse à plus large échelle.

3.3 ÉTUDE SUR LA PRÉSENCE DES DÉFAUTS DE CODE DANS LES APPLICATIONS MOBILES

Les études sur la présence des défauts de code dans les applications mobiles portent essentiellement sur les défauts de code orientés objets puisque les définitions et outils de détection de code spécifiques à Android sont encore dans leurs débuts. Ainsi, Linares-Vásquez *et al.* [45] ont utilisé la méthode DECOR pour détecter 18 défauts de code dans des applications mobiles basées sur le langage *Java Mobile Edition* (J2ME) [41]. Cette étude à large échelle sur 1,343 applications montre que la présence

des défauts de code affecte négativement les métriques de qualité logiciel, en particulier les métriques qui sont réputées pour être liées à la présence de bogues, comme une complexité excessive. Ils ont aussi découvert que certains défauts sont plus courants dans certaines catégories d'applications Java mobile.

Concernant Android, Verloop [84] a utilisé des outils de réusinage populaires pour Java comme PMD [64] ou JDEODORANT [79] pour détecter des défauts comme *Large Class* ou *Long Method* dans des applications libres. Il a découvert que les défauts de code apparaissent à des fréquences différentes dans les classes spécifiques à Android (appelées *core classes*) en comparaison des autres classes. Par exemple, *Long Method* est détecté deux fois plus souvent dans ces classes en terme de ratio.

Mannan *et al.* [49] ont comparé la présence de défauts classiques orientés objets dans 500 applications Android et 750 applications de bureau développées en Java. Ils observent qu'il n'y pas de différences majeures entre ces types d'applications dans le type et la densité de présence des défauts de code. Toutefois, ils observent aussi que la distribution des défauts de code est plus variée dans les applications Android alors que sur les applications de bureau, elle est dominée par des défauts de duplications internes et externes. Enfin, ils appellent à la publication de plus de travaux concernant les défauts de code dans les applications mobiles.

Sağlam [72] a étudié la corrélation entre la présence de défauts dans les applications et les notes moyennes de ces applications sur les plates-formes de téléchargement. En particulier, il a observé que les applications contenant des *Member Ignoring Method* ont tendance à avoir plus de mauvaises notes dans des proportions significatives. Il suppose donc que la suppression des défauts de code pourrait augmenter les notes utilisateurs sur le long terme. C'est donc une étude qui établit un lien entre présence des défauts de code et expérience utilisateur dans les applications Android.

Notons aussi qu'il existe des études qui, bien que n'utilisant pas directement les défauts, fournissent des détails intéressants sur les métriques et les structures de code pertinentes pour la détection de défauts de code.

Ainsi l'outil SAMOA [60] permet aux développeurs d'analyser leurs applications Android avec le code source. Cet outil collecte des métriques comme le nombre de paquetages, de lignes de code ou encore la complexité cyclomatique. Il permet aussi de visualiser les appels à des méthodes d'interfaces de programmation externes, ainsi que d'observer l'évolution des métriques à travers le temps et de les comparer avec d'autres applications analysées. Ils ont effectué une étude sur 20 applications et ont découvert que ces applications étaient significativement différentes des applications de bureau. En effet, les applications Android tendent à contenir moins de classes, mais utilisent intensive-

ment des libraires externes. Le code ainsi produit serait donc plus complexe à comprendre durant les phases de maintenance et d'évolution.

Gjoshevski and Schweighofer [24] ont utilisé 140 règles LINT dans une étude afin de savoir si la taille d'une application (en nombre de lignes de code) est liée à la dette technique mais aussi pour savoir quelles sont les règles les plus courantes dans les applications Android. Ils ont donc analysé 30 applications Android libres et ont conclu que la taille en nombre de lignes de code n'a pas d'impact sur la dette technique des applications. De plus, les règles les plus courantes dans les applications analysées sont *Visibility modifier*, *Avoid commented-out lines of code*, et *Magic number*.

Ruiz *et al.* [71] ont analysé des applications Android pour comprendre la réutilisation de classes dans ce type d'applications. À cette fin, ils extraient le code binaire et analysent les signatures des méthodes. Ils ont découvert que la réutilisation de classes à travers l'héritage interne, les librairies externes et le le cadre d'application Android est plus fréquente que dans les autres types d'applications. Xu [89] a aussi examiné le code binaire de 122,570 Android applications. Dans cette étude, l'auteur détermine les erreurs et les pièges courants qui sont fréquents pour les développeurs dans le manifeste et les permissions du système Android. Il a aussi observé que la réflection et l'obfuscation sont couramment utilisés dans ces applications, ce qui rend donc la rétro-ingénierie plus complexe. Ici aussi il constate que les librairies externes sont massivement utilisées par les applications Android.

Les études que nous présentons dans cette thèse viennent compléter ces travaux en se basant surtout sur les défauts de code spécifiques à Android avec de larges volumes d'applications.

3.4 ANALYSE DE L'IMPACT DES DÉFAUTS DE CODE SUR LA PERFORMANCE ET LA CONSOMMATION

Il y a relativement peu d'études dans la littérature sur l'impact des défauts de code sur la performance ou la consommation d'énergie. Même en se concentrant uniquement sur les mauvaises pratiques, on retrouve peu d'études sur les performances, toutefois des tests de comparaisons sont disponibles en ligne, notamment sur la documentation Android [4, 5, 30, 57]. Mais ces tests sont généralement effectués sur des applications jouets et se contentent d'exprimer des différences en terme de temps d'exécution ou de consommation de mémoire d'une seule méthode. Concernant la consommation d'énergie, on trouve plus de ressources sur les mauvaises pratiques de haut niveau comme le *wakelocks* (lorsque qu'une ressource matérielle comme l'écran reste allumée pour de mauvaises raisons) [7, 26], l'utilisation massive de publicités [28] ou la mauvaise utilisation du réseau de données [6, 18, 44]. La suite de cette section aborde les études relatives aux défauts de code qui traitent sou-

vent à la fois les aspects de performance et de consommation d'énergie dans une seule publication.

Li and Halfond [43] ont étudié l'impact de pratiques de programmation sous Android permettant de réduire la consommation d'énergie. Ils ont comparé dans une application témoin, répétant la même opération en boucle, l'impact de la présence de *Internal Getter/Setter* et *Member Ignoring Method*. Ils ont constaté une diminution de la consommation d'énergie de 33% pour *Internal Getter/Setter* et 15% pour *Member Ignoring Method* lorsque ces défauts de code sont corrigés. Ils ont aussi découvert que lorsqu'une application utilise plus de mémoire elle consomme aussi légèrement plus d'énergie.

Tonini *et al.* [78] ont étudié le temps d'exécution et la consommation d'énergie de *Internal Getter/Setter* et différentes syntaxes pour les boucles `for` dans les applications Android. Cette comparaison a été effectuée en exécutant 30 fois une portion de code contenant les différentes pratiques en matière de boucles et de présence de *Internal Getter/Setter*. Ils ont confirmé que la correction de *Internal Getter/Setter* améliorait le temps d'appel de la méthode de 30% et réduit la consommation d'énergie de 27%.

Mundody et K [62] ont fait un travail similaire avec le temps d'exécution et la consommation d'énergie pour *Internal Getter/Setter* et différentes syntaxes de boucles. En utilisant le test-t de Student [75], ils ont constaté que la correction de *Internal Getter/Setter* avait un impact significatif et positif sur le temps d'exécution et la consommation d'énergie d'une application en corrigeant ce défaut dans deux applications. Toutefois, il n'y a pas de détail sur le nombre de corrections et le processus utilisé pour instrumenter les applications.

Ricardo Pérez-Castillo et Mario Piattini [66] ont analysé l'impact de la correction de *BLOB* sur la consommation d'énergie des applications Android. Ils ont découvert que la correction de ce défaut de code augmente la consommation d'énergie, en ajoutant des méthodes, des classes et en augmentant le nombre de lignes de code. Il peut donc y avoir des effets de bord négatifs à la correction de certains défauts de code.

Rodriguez *et al.* [69] ont analysé l'impact que la présence de certains défauts de code peut avoir sur la consommation d'énergie d'application Android. Ils se concentrent sur quatre défauts de code orientés objet, à savoir *God Class*, *Brain Method*, *No Encapsulated Field*, et *No Self-Encapsulated Field*. Ils ont comparé les versions de trois applications contenant ou ne contenant pas les défauts de code. Ils ont ainsi découvert que la correction de ces défauts de code qui affectent la maintenance, peut augmenter la consommation d'énergie dans les cas où de nombreux objets sont créés mais la réduire dans le cas contraire.

Gottschalk *et al.* [25] proposent une approche de réusinage capable de détecter et corriger automatiquement certaines mauvaises pratiques liées à la consommation d'énergie, comme l'utilisation excessive du GPS. Bien que l'approche proposée ne soit pas spécifique aux plates-formes

mobiles, ils l'illustrent avec un exemple sur la plate-forme Android. La détection des mauvaises pratiques reste toutefois assez simpliste et ne spécifie pas comment la correction pourrait être effectuée. Ils n'ont pas évalué l'impact de la correction de ces mauvaises pratiques.

Bien que tous ces travaux soient pertinents et représentent des contributions intéressantes, ils se focalisent souvent sur l'impact de la correction à un niveau local, contrairement à nos travaux qui utilisent des métriques de plus haut niveau liées à l'expérience utilisateur mais aussi des scénarios complets d'utilisation sur des applications libres. De plus, certains des résultats de ces études sont à prendre avec précaution puisque les protocoles de mesures de la consommation d'énergie sont rarement décrits en détails.

3.5 CONCLUSION

En parcourant cet état de l'art, nous pouvons faire le constat que les connaissances et les approches concernant l'étude des défauts de code sur Android sont encore très lacunaires, en particulier concernant les défauts de code spécifiques à Android. Nous nous sommes donc fixés pour but de fournir non seulement des résultats qui complètent les études déjà réalisées mais aussi des approches complètes permettant la mise en œuvre facile d'études futures dans ce domaine.

Troisième partie

CONTRIBUTIONS

4 CLASSIFICATION DES DÉFAUTS DE CODE

Dans ce chapitre, nous décrivons et classifions les défauts de code OO et spécifiques à Android détectés par l'approche Paprika. À ce jour, Paprika est capable de détecter 17 types de défauts de codes différents, dont 4 sont des défauts de codes classiques de l'OO et 13 défauts de code spécifique à Android. Afin de faciliter la compréhension de ces défauts de code, nous les avons tout d'abord catégorisés selon les propriétés de qualité affectées par la présence de ces défauts de code. On retrouve ainsi deux grandes catégories : *les défauts de code affectant la maintenance* et *les défauts de code affectant la performance*. Puisque la majorité des défauts de codes détectés par notre approche sont liés aux performances, nous subdivisons cette catégorie en 4 sous-catégories : *micro-optimisations*, *gestion de la mémoire*, *affichage de l'interface utilisateur (UI)* et *blocage de processus léger*. Ces sous-catégories sont décrites dans la suite de ce chapitre. Il est à noter que notre volonté première a été d'étendre la liste de défauts de code liés à la maintenance en y ajoutant certains spécifiques à Android. Toutefois, ces défauts de code sont inexistants dans la littérature et les retours de développeurs nous portent à croire que les défauts de code OO sont toujours d'actualité et ne nécessitent donc pas de mise à jour concernant la maintenance des applications Android [84]. Nous avons donc concentré nos efforts sur la recherche de défauts de code liés aux performances. À notre connaissance, seul cinq de ces défauts sur les 13 de cette catégorie avaient déjà été évoqués dans la littérature scientifique avant nos recherches et aucun outil ne proposait leur détection.

4.1 CATÉGORIES ET DÉFINITIONS DES DÉFAUTS DE CODE

4.1.1 *Défauts de code affectant la maintenance*

Les défauts de code affectant la maintenance sont des défauts de codes connus dans le monde de l'OO pour compliquer les tâches de maintenance et d'évolution d'une application. En rendant le code source plus complexe qu'il ne devrait l'être, ils rendent plus difficile sa compréhension et augmentent les efforts à fournir pour modifier ce code. Les quatres défauts de code suivants sont issus de la littérature scientifique.

4.1.1.1 *Blob Class (BLOB) - OO*

Un *Blob class*, parfois aussi appelé *God class*, est une classe avec un large nombre d'attributs et/ou de méthodes [9]. C'est une classe qui centralise les traitements et a donc beaucoup de responsabilités en com-

paraison des autres classes qui ne contiennent que des données ou se charge de l'éxécution de processus simples. La cohésion est très faible au sein d'un Blob puisque que ses méthodes et ses attributs concernent différents concepts et processus de l'application. Ce type de classes est souvent associé à des nombreuses classes dites de données, ce qui augmentent le couplage avec ces classes. Par conséquent, les blobs sont difficiles à réutiliser, modifier ou à tester.

4.1.1.2 *Swiss Army Knife (SAK) - OO*

Un *Swiss Army Knife* est une interface de classe très complexe qui regroupe de nombreuses méthodes. Cette interface est conçue de manière à couvrir différentes abstractions et de nombreux besoins pour une seule classe l'implémentant. Ce type d'interfaces et les classes qui en héritent sont difficiles à comprendre et à maintenir à cause de la complexité qui découle du grand nombre de méthodes. De plus, seule une partie des méthodes héritées se révèle utile aux classes qui héritent d'un *Swiss Army Knife* [9].

4.1.1.3 *Long Method (LM) - OO*

Une *Long Method* est comme son nom l'indique, une méthode qui contient de nombreuses lignes de code. Ces méthodes sont souvent très complexes, et par conséquent difficiles à comprendre et à maintenir. De plus, souvent ces méthodes effectuent plus de traitement que ce que leurs noms suggèrent rendant encore plus difficile la maintenance de la méthode. Ces longues méthodes peuvent généralement être divisées en méthodes plus petites et spécialisées afin de résoudre le problème [21].

4.1.1.4 *Complex Class (CC) - OO*

Une *Complex Class* est une classe qui contient plusieurs méthodes complexes. Cette fois aussi, ces classes sont difficiles à comprendre et à maintenir. Elles doivent donc être réusinées afin d'obtenir plusieurs classes aux méthodes moins complexes [21]. Afin de calculer la complexité d'une classe, on somme la complexité interne de chacune des méthodes qui la composent. Pour calculer la complexité d'une méthode, on peut par exemple utiliser la Complexité Cyclomatique de McCabe (*McCabe's Cyclomatic Complexity* en anglais) [58].

4.1.2 *Défauts de code affectant la performance : Micro-optimisations*

Les défauts de code qui affectent légèrement les performances d'une méthode et qui peuvent être corrigés de manière systématique sont classés dans la catégorie micro-optimisations. Généralement, une micro-optimisation peut s'obtenir en modifiant une seule ligne de code au sein de la méthode. Deux défauts de code parmi les trois suivants sont issus

de la littérature scientifique et de la documentation Android [4, 10], tandis que nous avons défini le dernier à partir d'une vidéo présentant les recommandations des développeurs de la plate-forme Android [56].

4.1.2.1 *Internal Getter/Setter (IGS) - Android*

Un *Internal Getter/Setter* est un défaut de code spécifique à Android, qui se produit lorsqu'un attribut est utilisé au sein même de la classe qui le déclare au travers d'un accesseur (*getter* en Anglais, par exemple `var = getField()`) et/ou d'un mutateur (*setter* en Anglais, par exemple `setField(var)`). Bien que courante, cette pratique peut dégrader la performance de l'application. L'utilisation d'*Internal Getter/Setter* est habituelle dans les langages OO comme C++, C# ou Java et n'a pas d'impact significatif puisque les compilateurs ou les machines virtuelles peuvent optimiser automatiquement ces cas en remplaçant l'appel d'une fonction par le code de cette fonction (*inlining* en anglais) dans ces circonstances. Mais, ce n'est pas le cas pour la machine virtuelle Dalvik sous Android [87] qui est seulement capable d'effectuer cette optimisation dans des cas très simples. Par conséquent, les appels à des accesseurs ou des mutateurs sont, dans la majorité des cas, convertis en appels virtuels qui sont au moins trois fois plus lents qu'un accès direct à la variable. Ce défaut de code peut donc être corrigé simplement en accédant directement à l'attribut de la classe à l'intérieur de celle-ci (`var = this.myField`, `this.myField = var`) tout en conservant les accesseurs et mutateurs pour les classes externes.

4.1.2.2 *Member Ignoring Method (MIM) - Android*

Sur Android, pour des raisons de performances, il est recommandé d'utiliser des méthodes statiques dès que possible sans se soucier de la sémantique [4, 10]. Plus précisément, une méthode devrait être statique si elle n'est pas un constructeur ou si elle n'utilise aucun attribut ou aucune méthode non statique. L'usage de méthodes statiques est aussi considéré comme facilitant la compréhension du code, puisque cela signifie que la méthode appelée ne modifiera pas l'état de l'objet.

4.1.2.3 *Init OnDraw (IOD) - Android*

La méthode `OnDraw` des classes héritant de `View` est responsable de la mise à jour de l'interface graphique. Lorsque la vue est active, cette méthode est appelée à chaque rafraîchissement (jusqu'à 60 fois par seconde). Par conséquent, il est important de minimiser le temps d'exécution de cette méthode afin de maximiser les performances et la fluidité de l'application. Sachant qu'un grand nombre d'allocations mémoire va augmenter la consommation de mémoire et le nombre d'appels au ramasse-miettes (coûteux en temps d'exécution), il est important d'éviter d'effectuer des allocations mémoire (via l'usage de `new` ou d'une fabrique) dans les méthodes `OnDraw` [56].

4.1.3 *Défauts de code affectant la gestion de la mémoire*

Les quatre défauts de code qui ont un effet direct sur l'utilisation ou la gestion de la mémoire entrent dans la catégorie *gestion de la mémoire*. Les deux premiers défauts sont issus de la littérature scientifique [10] alors que nous avons défini les deux derniers à partir des recommandations des développeurs de la plate-forme Android [5, 30, 57].

4.1.3.1 *No Low Memory Resolver (NLMR) - Android*

Dès lors que le système Android commence à manquer de mémoire vive pour fonctionner correctement, il appelle automatiquement la méthode `onLowMemory()` pour chaque activité en cours d'exécution. Cette méthode, qui doit être implémentée par le développeur de l'application, a pour responsabilité de libérer de la mémoire pour chaque activité en libérant les ressources superflues. Si cette méthode n'est pas implémentée, alors le système commence à tuer automatiquement des processus peu actifs pour libérer la mémoire nécessaire. Cela produit des fins de programme non souhaitées par l'utilisateur et oblige les applications à effectuer un coûteux démarrage à froid lors de la prochaine utilisation [10]. En effet, l'utilisateur peut naviguer presque instantanément entre les applications tant que toutes les ressources nécessaires à leur fonctionnement sont en mémoire. Mais si certains processus ont été tués alors il devient nécessaire de redémarrer l'application totalement.

4.1.3.2 *Leaking Inner Class (LIC) - Android*

En Java, une classe interne (*Inner Class* en anglais) est une classe qui est définie et contenue au sein d'une autre classe, appelée la classe externe. Les classes internes non statiques et les classes anonymes possèdent automatiquement une référence vers la classe externe. Ainsi, il est nécessaire que la classe externe soit instanciée pour avoir une instanciation de la classe interne. Ce comportement peut facilement générer des fuites de mémoire (*Memory Leak* en anglais) sur les systèmes Android [10, 47]. En effet, si la classe interne est référencée par une instance d'objet autre que sa classe externe, il devient nécessaire de garder en mémoire la classe externe aussi même si elle n'est plus utile au programme. Le ramasse-miettes ne peut alors pas libérer la mémoire occupée par cette classe externe. Cela peut rapidement saturer la mémoire si ces instances sont nombreuses ou si la classe externe référence elle aussi de nombreux objets, comme c'est souvent le cas pour des Activités, par exemple.

4.1.3.3 *Unsuited LRU Cache Size (UCS) - Android*

Sur Android, il est possible d'utiliser facilement un système de cache de type LRU (*Least Recently Used*) afin d'optimiser l'utilisation de la mémoire. Pour cela, l'API d'Android met à disposition l'objet `LruCache`

[48] qui est souvent utilisé pour stocker des données volumineuses comme des tableaux de bits. Toutefois, la mémoire disponible d'un type de périphérique à un autre peut grandement varier sous Android, c'est pourquoi il est recommandé d'ajuster la taille du cache en fonction de la mémoire disponible sur le périphérique [57]. Ainsi, l'initialisation d'un cache sans vérifier la mémoire disponible du périphérique, via la méthode `getMemoryClass` disponible sur l'API Android, est considérée comme une mauvaise pratique.

4.1.3.4 *Hashmap Usage (HMU) - Android*

Afin de remplacer la traditionnelle `HashMap` Java, l'API d'Android fournit les classes `ArrayMap` et `SimpleArrayMap`. Ces deux `Map` sont adaptés aux besoins des périphériques mobiles, et sont donc optimisés afin de consommer moins de mémoire tout en déclenchant moins souvent le ramasse-miettes [5]. De plus, les opérations d'ajouts ou de suppressions sont aussi performantes que pour une `HashMap`. Ces optimisations sont efficaces pour une taille de `Map` allant jusqu'à plusieurs centaines de valeurs, la `HashMap` restant plus adapté en terme de performance pour les plus grandes quantités. Par conséquent, on considère l'utilisation de petites et moyennes `HashMaps` comme un défaut de code [5, 30].

4.1.4 *Affichage de l'IHM*

Les trois défauts de code qui affectent le processus d'affichage sur Android sont catégorisés dans *Affichage de l'IHM*. Une fois de plus nous avons définis ces défauts de code à partir des recommandations des développeurs de la plate-forme Android [55, 63].

4.1.4.1 *UI Overdraw (UIO) - Android*

L'*UI Overdraw* apparaît lorsque le système utilise des cycles du processeur pour générer l'affichage de pixels qui n'apparaîtront finalement pas lors de l'affichage final de l'image sur l'écran. Cela peut être provoqué par la superposition de plusieurs couches d'interfaces graphiques ou quand le calcul est effectué pour redessiner des pixels qui ne nécessitent pas d'être redessiner. En effet, s'il n'y a pas d'effets de transparence alors seule la dernière couche dessinée sera visible par l'utilisateur. Pour éviter ces situations, l'API Android propose la méthode `canvas.cliprect()` qui permet de définir précisément quelle zone doit être dessinée en excluant les autres parties qui ne seront alors pas dessinées ou redessinées. Il existe aussi une méthode `canvas.quickreject()` qui, à l'inverse, permet d'exclure des zones de l'interface graphique qui ne seront pas considérées lors du prochain calcul de l'affichage. Ainsi, la non utilisation de ces méthodes peut réduire les performances de l'application [55].

4.1.4.2 *Invalidate Without Rect (IWR) - Android*

La plupart du temps, le système Android va automatiquement détecter si un changement affecte une vue et nécessite donc le rafraîchissement de celle-ci. Toutefois, pour certaines animations ou pour des vues personnalisées, il devient nécessaire de forcer le rafraîchissement de la vue. Sur Android, cette action se réalise simplement en appelant la méthode `invalidate()` sur les vues concernées. Cette phase de rafraîchissement s'exécute sur le processus léger principal de l'application. Ce processus est sollicité par toutes les vues et un rafraîchissement est une opération coûteuse. Il est donc nécessaire de réduire au maximum le temps de monopolisation de ce processus afin de ne pas nuire à l'expérience utilisateur en bloquant l'affichage, surtout si la méthode `invalidate()` est appelée fréquemment. Afin de réduire le temps d'affichage, il est recommandé de préciser exactement quelles sont les parties de la vue qui doivent être redessinées en utilisant un `Rect` comme argument pour chaque appel à `invalidate()` [63]. Ainsi l'utilisation de `invalidate()` sans cet argument `Rect` est considérée comme une mauvaise pratique pouvant affecter les performances.

4.1.4.3 *Unsupported Hardware Acceleration (UHA)*

Lorsque c'est possible, le système Android utilise l'accélération graphique matérielle pour les opérations d'affichages. Cela signifie que les calculs pour gérer l'affichage sont effectués sur le processeur graphique plutôt que sur le processeur traditionnel. En effet, le processeur graphique est bien plus performant pour ce type d'opérations et cela permet de laisser le processeur disponible pour les autres processus du système. Toutefois, certaines opérations graphiques (*e.g.*, la méthode `drawPath` de la classe `android.graphics.Canvas`) ne peuvent pas être effectués sur le processeur graphique. Il est donc recommandé d'éviter ces opérations autant que possible et d'utiliser les alternatives qui peuvent s'exécuter sur le processeur graphique [63]. Par exemple, la méthode `drawPath` peut souvent être remplacé par plusieurs appels à `drawLine`.

4.1.5 *Blocage de processus*

Les trois défauts de code pouvant mener à un message "L'application ne répond plus" (*Application Not Responding* (ANR) en anglais) en bloquant le processus léger principal de l'application sont catégorisés dans la catégorie *blocage de processus*. Nous avons définis ces trois défauts de code à partir du retour d'expérience d'un développeur sur son blogue et de la documentation Android [51–53].

4.1.5.1 *Heavy AsyncTask (HAS) - Android*

Sous Android, une `AsyncTask` est une tâche asynchrone disponible dans l'API qui est destinée au lancement de courtes tâches de fond pour l'application. C'est une classe pratique pour les développeurs qui n'ont alors pas besoin de lancer un processus manuellement afin d'effectuer certaines tâches. Toutefois, trois des quatre étapes d'exécution d'une `AsyncTask` sont exécutées sur le processus principal et non en tâche de fond. Par conséquent, il est nécessaire d'éviter de bloquer ou d'effectuer de longs traitements sur ces trois étapes qui sont représentées par les méthodes `onPostExecute`, `onPreExecute`, et `onProgressUpdate` [51]. Dans le cas contraire cela peut provoquer : *i)* un manque de réactivité de l'interface utilisateur, dégradant l'expérience de ce dernier *ii)* une fenêtre de dialogue "L'application ne répond plus" qui propose de forcer la fermeture de l'application.

4.1.5.2 *Heavy Service Start (HSS) - Android*

Heavy Service Start est similaire à HAS. Sur Android, les services sont destinés à effectuer des opérations lourdes en tâche de fond. Toutefois, lors du démarrage d'un service la méthode `OnStartCommand` est exécutée sur le processus principal et peut donc bloquer celui-ci. Le fonctionnement de cette méthode est parfois mal compris par les développeurs qui pensent que l'exécution est effectuée sur un autre processus, alors que ce processus doit être créé explicitement par le développeur [53]. Dans le contraire, cela peut mener à un message "L'application ne répond plus".

4.1.5.3 *Heavy BroadcastReceiver (HBR) - Android*

Android permet la communication d'une application avec le système ou avec d'autres applications via l'utilisation de `BroadcastReceiver`. Cette opération peut être assimilée à une tâche de fond mais, comme dans les cas vus précédemment, une mauvaise implémentation peut déclencher un message "L'application ne répond plus". En effet, la méthode `onReceive` d'un `BroadCastReceiver` s'exécute sur le processus principal. Il est donc aussi recommandé d'éviter les opérations bloquantes ou longues à exécuter dans cette méthode [52].

4.2 CLASSIFICATION DES DÉFAUTS DE CODE

Il existe déjà de nombreux catalogues de défauts de code OO dans la littérature, le plus célèbre étant celui de Beck [22]. Toutefois, la plupart de ces catalogues ne proposent aucune classification. Des classifications de défauts de code existants ont donc été proposés [50, 86], mais celles-ci se limitent le plus souvent aux défauts de code OO et ne sont donc pas applicables aux défauts de code présentés précédemment ou à Android de manière générale. C'est pour cette raison que nous avons choisis de

créer notre propre classification, applicable à tous les défauts de code détectés par PAPRIKA. Cette classification facilite la compréhension des défauts de code tout en prenant en compte les spécificités des systèmes mobiles et en particulier d'Android.

Nous proposons donc une taxonomie hybride qui regroupe à la fois les défauts de code OO et Android. Cette taxonomie repose sur cinq critères orthogonaux qui sont décrits dans les paragraphes suivants.

OO ET SPÉCIFIQUE À ANDROID Naturellement, nous différencions d'abord les défauts de code spécifiques à Android de ceux qui peuvent s'appliquer à n'importe quel langage OO. Les défauts de code spécifique à Android ne concernent donc que les applications codées en Java en utilisant l'API Android. Ces défauts de code sont soit dus à des spécificités de l'API soit dus à la machine virtuelle qui exécute le code (Dalvik ou ART selon les versions d'Android).

DÉTECTION OBJECTIVE ET SUBJECTIVE Nous classifions aussi les défauts de code en fonction du degré d'objectivité de la métrique sur laquelle repose la détection. En effet, la détection de certains défauts de code sont basés sur des métriques totalement objectives qui ne peuvent prendre que deux valeurs (VRAI ou FAUX) tandis que la détection des autres reposent sur des métriques qui varient dans un intervalle (*e.g.*, la complexité cyclomatique ou le nombre de lignes de code). Concrètement, la détection de défauts de code subjectifs nécessite de définir des seuils qui vont guider la détection, par exemple pour déterminer à partir de combien de lignes de codes peut-on considérer une méthode comme étant effectivement longue. Bien que ces seuils puissent être calculés avec rigueur à partir de méthodes statistiques, leur efficacité reste arbitraire puisque la validité de la décision dépendra finalement du point de vue de l'utilisateur. Un développeur pouvant estimer qu'une méthode est longue à partir d'environ 30 lignes tandis que son collègue estimera que 60 lignes est plus adapté comme seuil. Au contraire, lorsque la détection est objective cela repose sur une métrique qui ne dépend pas du point de vue de l'utilisateur. Par exemple, l'usage d'un accesseur interne à une classe ne peut pas être contesté.

NIVEAUX DE SÉVÉRITÉ Nous utilisons aussi trois niveaux de sévérité (suspect, problématique et critique) pour classifier les défauts de code. *Les défauts de code critiques* devraient absolument être évités en toute circonstance parce qu'ils ne font que dégrader la qualité ou les performances de l'application sans apporter de bénéfice. *Les défauts de code problématiques* affectent aussi négativement la qualité ou les performances de l'application et il donc préférable de les éviter. Toutefois, les développeurs de l'application peuvent avoir des raisons pour les maintenir dans l'application, par exemple : *i)* Ils ne sont pas faciles à corriger, et donc la correction serait coûteuse en terme de temps ; *ii)* Leurs cor-

rections pourraient générer d'autres problèmes au sein de l'application, que ce soit en terme de conception ou de performance ; *iii)* Ils ont été prévus lors de la conception de l'application et leur présence est donc volontaire. Enfin, *les défauts de code suspects* nécessitent un examen approfondi et un jugement subjectif de la part des développeurs afin de déterminer si les occurrences détectés sont à considérer comme des cas réels de défauts de code ou non. Il est ainsi préférable de les éviter dans les applications Android, mais il arrive que leur présence soit totalement justifiée ou que l'impact sur la qualité ou les performances de l'application ne soit pas assez significatif pour justifier une correction du défaut.

CLASSES AFFECTÉES Finalement, nous classifions les défauts de code selon le type de classes qui peuvent être affectées par le défaut de code en question. En effet, certains défauts peuvent se retrouver dans n'importe quelle classe tandis que d'autres sont spécifiques aux classes qui héritent du cadre d'application Android (AFIC pour *Android Framework Inherited Classes* en anglais). La plupart du temps, ces défauts apparaîssent uniquement dans les méthodes surchargées des AFICs. Par exemple, Init OnDraw n'apparaît que dans les méthodes `OnDraw` des activités Android.

COMMENTAIRES La classification des défauts de code utilisant tous ces critères apparaît dans la table 1.

La plupart des défauts de code que nous détectons sont objectifs (10 sur 17) et utilisent donc des métriques booléennes qui nous permettent de savoir si les conditions sont réunies pour la présence ou non du défaut. Les défauts de code concernant la maintenance et le blocage de processus sont au contraire tous subjectifs. En effet, on peut constater que des notions de longueurs ou de complexités apparaissent dans la définition de ces défauts.

On peut remarquer que, concernant le niveau de sévérité, il y seulement deux défauts de code critiques alors que la majorité sont considérés comme suspects. De plus, les défauts de code subjectifs ne sont jamais catégorisés comme critiques puisque les seuils qui sont utilisés pour leur détection peuvent être débattus. D'un autre côté, les défauts de code objectifs ne sont pas tous critiques. Ici, la présence du défaut n'est pas remise en question, toutefois il peut exister des raisons valables pour les conserver dans l'application.

Sept défauts de code concernent uniquement des AFICs tandis que les dix autres (dont quatre sont OO) peuvent apparaître dans n'importe quel type de classes.

Enfin remarquons que IOD ne se situe pas dans la catégorie Affichage de l'IHM malgré ce que pourrait laisser penser son nom qui le relie directement aux méthodes `OnDraw()`. Nous avons choisi de le catégoriser dans les micro-optimisations puisque ce défaut de code n'est pas direc-

Tableau 1 – Classification des défauts de code

	Orienté Objet				Android												
	Maintenance				Micro-optimisations			Gestion Mémoire				Affichage IHM			Blocage processus		
	BLOB	CC	LM	SAK	IGS	MIM	IOD	LIC	UCS	HMU	NLMR	UHA	IWR	UIO	HAS	HSS	HBR
Objectif/Subjectif																	
Objectif					✗	✗	✗	✗	✗	✗	✗	✗	✗	✗			
Subjectif	✗	✗	✗	✗											✗	✗	✗
Sévérité																	
Critique					✗	✗											
Problématique	✗	✗	✗				✗	✗	✗	✗	✗	✗					
Suspect				✗									✗	✗	✗	✗	✗
Classes affectées																	
AFIC							✗				✗		✗	✗	✗	✗	✗
Toutes classes	✗	✗	✗	✗	✗	✗		✗	✗	✗		✗					

tement relié à la manière dont l'écran est dessiné mais plutôt à un problème de performance due à la création d'instances d'objets à chaque rafraîchissement.

5

DÉTECTION DES DÉFAUTS DE CODE : PAPRIKA

Dans ce chapitre, nous présentons l'approche outillée qui est au coeur de cette thèse : Paprika. Le but premier de cette approche est de fournir permettre la détection des défauts de code du chapitre précédent. Toutefois, Paprika propose aussi la correction de certains défauts de code ainsi que la possibilité de calculer un score de qualité pour chaque application. C'est une approche qui se veut flexible et novatrice. En effet, à notre connaissance c'est la seule approche fonctionnelle capable de détecter des défauts spécifiques à Android. De plus, elle utilise les connaissances acquises sur l'ensemble des applications analysées pour proposer les seuils de détection de manière fiable et non arbitraire. Cette approche qui permet aussi bien l'analyse d'application libres que propriétaires puisqu'elle ne nécessite pas l'accès au code source de l'application. Elle consiste en cinq étapes (dont deux optionnelles), comme on peut l'apercevoir dans la figure 2, que nous allons décrire en détails dans ce chapitre.

La première étape consiste en l'analyse de l'APK de l'application à inspecter pour en extraire le modèle de l'application et les métriques associées. Cette étape utilise aussi des métadonnées en argument (par exemple, nombre de téléchargements, note moyenne des utilisateurs) indisponible dans l'APK mais que l'on peut par exemple trouver sur le Google Play Store. Dans la deuxième étape, la représentation obtenue est automatiquement visitée afin de convertir le modèle obtenu précédemment (incluant classes, méthodes, attributs, etc.) en un graphe qui possède les métriques de qualité sur ses noeuds. Le graphe résultant est stocké dans une base de données. La troisième étape permet l'exécution des requêtes destinées à la détection des défauts de code dans les applications analysées.

Concernant les étapes optionnelles, la première permet la correction de certains défauts de code alors que la seconde fournit à l'utilisateur le moyen d'évaluer la qualité de son application par rapport au reste de la base de données. De plus, cette étape peut aussi permettre de suivre l'évolution de la qualité d'une application sur plusieurs versions. Ces étapes sont optionnelles car elles ne sont pas nécessairement activés lors de l'utilisation de Paprika mais peuvent être activées par l'utilisateur. Chacune de ces étapes sera décrites en fonction de ses entrées et sorties accompagnées d'une description et de détails sur l'implémentation.

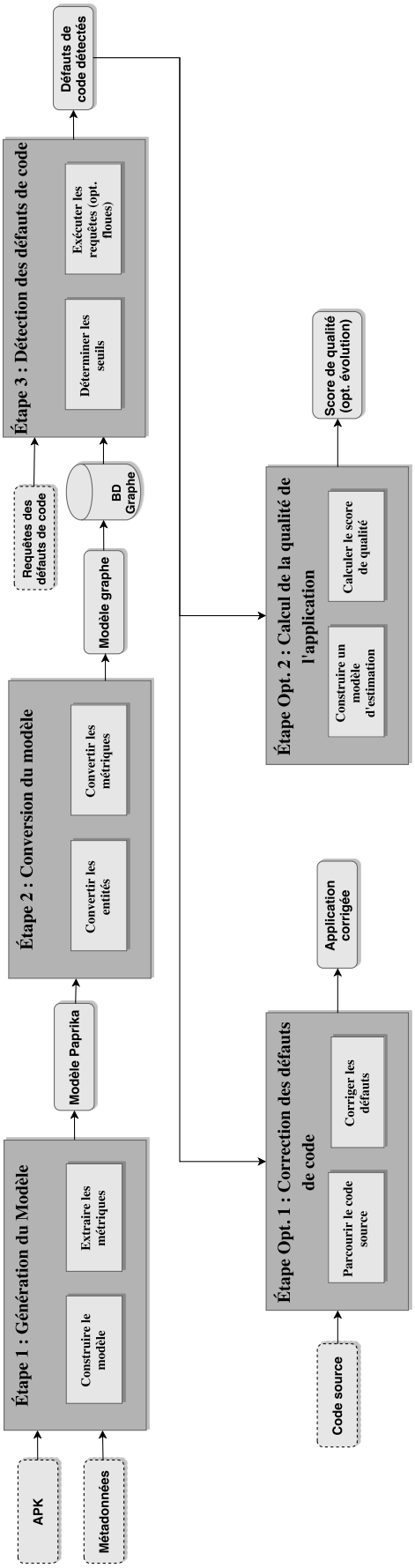


FIGURE 2 – Aperçu de l'approche PAPRIKA

5.1 ÉTAPE 1 : GÉNÉRATION DU MODÈLE DE L'APPLICATION

5.1.1 *Entrées*

Un fichier APK de l'application à analyser et les métadonnées correspondantes.

5.1.2 *Sortie*

Un modèle PAPRIKA contenant entités, métriques et propriétés.

5.1.3 *Description*

Cette étape consiste à générer un modèle représentant l'application tout en extrayant des métriques de qualité brutes depuis le paquet contenant l'application.

Ce modèle est construit de manière incrémentale pendant l'analyse du code binaire afin de définir des entités auxquelles sont associées des propriétés annexes, extraites du Google Play Store. Le modèle ainsi construit par PAPRIKA est composé de huit entités :

1. App qui représente l'application analysée,
2. Class qui représente une classe (possiblement interne) appartenant à l'application,
3. Method qui représente une méthode appartenant à une classe,
4. Argument qui représente un argument de méthode,
5. Variable qui représente une variable de classe,
6. ExternalClass qui représente une classe utilisée par l'application mais appartenant à l'API Java, Android ou une libraire externe,
7. ExternalMethod qui représente une méthode utilisée par l'application mais appartenant à une ExternalClass,
8. ExternalArgument qui représente un argument appartenant à une ExternalMethod.

Les trois dernières entités ont généralement les mêmes propriétés et relations que les entités internes correspondantes. Toutefois, certaines propriétés sont moins détaillées et les métriques (par exemple, couplage ou complexité) ne sont pas calculées pour ces entités puisqu'elle sortent du cadre de l'application et ne sont donc pas sous la responsabilité du développeur.

Des propriétés sont attachées à chacune de ces entités sous la forme d'attributs qui sont répertoriés dans le tableau 2. L'identifiant unique de l'application est généralement le hachage SHA-256 effectué sur l'APK de l'application, les autres propriétés qui ne concernent que l'application sont à passer en argument lors de l'utilisation de PAPRIKA et proviennent généralement du Google Play Store. Les autres propriétés servent

à compléter le modèle de l'application. Les entités sont aussi liées entre elles par les relations décrites dans le tableau 3. On retrouve ici les relations d'appartenances,d'usage,d'héritage et d'implémentation ainsi qu'une relation pour modéliser le graphe d'appel des méthodes.

Tableau 2 – Liste des propriétés des entités de PAPRIKA

Nom	Entités	Commentaires
`name`	All	Nom de l'entité
`app_key`	All	Identifiant unique de l'application
`rating`	App	Note de l'application
`date_download`	App	Date de téléchargement de l'APK
`date_analysis`	App	Date de l'analyse
`package`	App	Nom du paquet principal
`size`	App	Taille de l'APK (MB)
`developer`	App	Nom du développeur
`category`	App	Catégorie de l'application
`price`	App	Prix de l'application
`nb_download`	App	Nombre de téléchargements
`sdk`	App	Version du SDK lors de la compilation
`target_sdk`	App	Version du SDK visée par l'application
`version_code`	App	Numéro de version interne
`version_name`	App	Numéro de version publique
`parent_name`	Class	Héritage de classe
`modifier`	Class Variable Method	`public`, `protected` ou `private`
`type`	Variable	Type (objet) de la variable
`full_name`	Method	`method_name#class_name`
`return_type`	Method	Type de retour de la méthode
`position`	Argument	Position de l'argument dans la signature

Pendant l'extraction de ces entités et relations, PAPRIKA analyse ou calcule des métriques qui seront associées aux entités. Les 42 métriques actuellement disponible dans PAPRIKA sont recensées dans le tableau 4. PAPRIKA peut contenir à la fois des métriques *OO* classiques mais aussi des métriques *spécifiques à Android*. Les métriques booléennes servent à distinguer les différents types d'entités tandis que les métriques numériques servent à compter certaines entités (par exemple, nombre de service) ou sont des métriques calculées (par exemple, couplage). Contrairement aux propriétés, les métriques requièrent souvent des calculs ou des manipulations pour être extraites du code binaire. Par exemple, il est nécessaire de parcourir l'arbre d'héritage afin de déterminer si une

Tableau 3 – Liste des relations entre entités de PAPRIKA

Nom	Entités	Commentaires
APP_OWNS_CLASS	App – Class	Application possède Classe
CLASS_OWNS_METHOD	Class – Method	Classe possède Méthode
CLASS_OWNS_VARIABLE	Class – Attribute	Classe possède Variable
METHOD_OWNS_ARGUMENT	Method – Argument	Méthode possède Argument
EXTENDS	Class – Class	Classe hérite de Classe
IMPLEMENTS	Class – Class	Classe implémente Interface
CALLS	Method – Method	Méthode appelle Méthode
USES	Method – Variable	Méthode utilise Variable

classe hérite d'une classe importante du cadre d'application Android tel que :

— *Activity*, qui représente un écran unique sur l'interface utilisateur. Une activité peut démarrer d'autres activités de l'application ou même d'autres applications.

— *View*, qui est un composant de l'interface utilisateur. Une vue représente une zone rectangulaire sur l'écran et gère l'affichage et les événements au sein de cette zone.

— *Service*, qui est une tâche de fond destinée aux longs traitements ou aux traitements distants.

— *AsyncTask*, qui est destinée aux tâches de fond courtes qui nécessitent un affichage des résultats sur l'interface utilisateur.

— *Content provider*, qui permet la gestion des données à partager avec d'autres applications et le système.

— *Broadcast receiver*, qui permet d'écouter et de répondre aux annonces de diffusion générale du système ou d'autres applications.

— *Application*, qui est utilisée pour maintenir un état global de l'application.

Certaines métriques sont dites dérivées [23], comme par exemple `ClassComplexity` ou `LackofCohesionInMethods`. Elles nécessitent généralement plus de calculs et sont basées sur d'autres métriques brutes ; elles sont calculées à la fin du processus lorsque toutes les métriques brutes sont disponibles.

On peut constater que les entités, propriétés et métriques permettent d'avoir un modèle élaboré et presque complet de l'application. En effet, seules les instructions sont manquantes afin de pouvoir reconstituer l'application depuis notre modèle.

5.1.4 *Implémentation*

Nous utilisons le cadre d'application SOOT [83] et son module DEXPLER [8] afin d'analyser les APKs Android. SOOT convertit le code binaire de Dalvik spécifique à Android en une représentation interne très

Tableau 4 – Liste de métriques des entités de PAPRIKA

Nom	Type	Entité	Commentaires
NumberOfClasses	OO	App	Nombre de classes
NumberOfInnerClasses	OO	App	Nombre de classes internes
NumberOfInterfaces	OO	App	Nombre d'interfaces
NumberOfAbstractClasses	OO	App	Nombre de classes abstraites
NumberOfVariables	OO	App	Nombre de variables
NumberOfMethods	OO	App, Class	Nombre de méthodes
DepthOfInheritance	OO	Class	Profondeur d'héritage
NumberOfImplementedInterfaces	OO	Class	Nombre d'interfaces implémentées
NumberOfAttributes	OO	Class	Nombre d'attributs
NumberOfChildren	OO	Class	Nombre d'enfants de la classe
ClassComplexity	OO	Class	Somme des complexités des méthodes
CouplingBetweenObjects	OO	Class	Couplage, Chidamber et Kemerer [13]
LackofCohesionInMethods	OO	Class	LCOM2 [13]
IsAbstract	OO	Class, Method	Booléen pour méthode abstraite
IsFinal	OO	Class, Variable, Method	Booléen pour méthode finale
IsStatic	OO	Class, Variable, Method	Booléen pour méthode statique
IsInnerClass	OO	Class	Booléen pour classe interne
IsInterface	OO	Class	Booléen pour interface
NumberOfParameters	OO	Method	Nombre de paramètres
NumberOfDeclaredLocals	OO	Method	Nombre de variables locales
NumberOfInstructions	OO	Method	Nombre d'instructions
NumberOfDirectCalls	OO	Method	Nombre d'appels effectués par la méthode
NumberOfCallers	OO	Method	Nombre d'appels par d'autres méthodes
CyclomaticComplexity	OO	Method	Compléxité de McCabe [58]
IsGetter	OO	Method	Booléen pour accesseur
IsSetter	OO	Method	Booléen pour mutateur
IsInit	OO	Method	Booléen pour constructeur
IsSynchronized	OO	Method	Booléen pour méthode synchronisée
IsOverride	OO	Method	Booléen pour méthode surchargée
NumberOfActivities	Android	App	Nombre d'Activités
NumberOfAsyncTasks	Android	App	Nombre d'Asynctasks
NumberOfBroadcastReceivers	Android	App	Nombre de BroadcastReceivers
NumberOfContentProviders	Android	App	Nombre de ContentProviders
NumberOfServices	Android	App	Nombre de Services
NumberOfViews	Android	App	Nombre de Vues
IsActivity	Android	Class	Booléen pour Activité
IsApplication	Android	Class	Booléen pour Application
IsAsyncTask	Android	Class	Booléen pour AsyncTask
IsBroadcastReceiver	Android	Class	Booléen pour BroadcastReceiver
IsContentProvider	Android	Class	Booléen pour ContentProvider
IsService	Android	Class	Booléen pour Service
IsView	Android	Class	Booléen pour Vues

proche du langage Java. SOOT peut aussi générer le graphe d'appel des fonctions du code ainsi analysé. Il est nécessaire de fournir à SOOT le code du SDK Android utilisé, afin de permettre la génération de ce graphe d'appel et d'avoir un modèle complet. Le modèle PAPRIKA se construit de manière incrémentale en visitant la représentation interne de SOOT. Les propriétés collectées depuis le Google Play Store sont ajoutées à ce modèle et servent aussi à préciser l'analyse en déterminant le paquet principal de l'application. En parallèle et de manière similaire, PAPRIKA procède à l'extraction des métriques brutes en se basant sur le modèle SOOT. La plupart des métriques doivent être calculées à la fin de l'exploration d'une ou plusieurs entités et nécessitent donc des traitements supplémentaires de la part de PAPRIKA. Il est important de noter que ces étapes ne sont pas exécutées séquentiellement mais plutôt de manière opportuniste lors de la visite du modèle SOOT, afin d'optimiser les performances et afin de réduire le temps d'exécution.

Traditionnellement, les approches de détection de défaut de code s'effectuent sur le code source [84] plutôt que sur le code binaire. En effet, bien que l'utilisation du code binaire permette l'analyse d'applications propriétaires cela soulève quelques problèmes techniques. Par exemple, il nous est impossible d'utiliser des métriques courantes comme le nombre de lignes de code d'une méthode ou le nombre de variables locales déclarées. Nous utilisons donc des métriques abstraites, qui sont des approximations des métriques manquantes. Par exemple, nous nous basons sur le nombre d'instructions afin d'estimer le nombre de lignes de code.

De plus, comme mentionné précédemment, de nombreuses applications disponibles sur les magasins d'applications en ligne comme le Google Play Store sont obfusquées. Cette obfuscation a pour but de réduire la taille de l'application mais aussi de rendre la rétro-ingénierie de l'application plus compliquée. Ainsi la plupart des méthodes, attributs et classes sont renommés avec des lettres sans signification. Notre analyse ne peut donc pas reposer sur ces données lexicales manquantes. Nous devons donc utiliser quelques stratégies détournées afin d'obtenir certaines informations. Par exemple, alors qu'il est habituel de recenser les accesseurs et mutateurs en se basant sur le nom, nous utilisons plutôt le nombre et le type d'instructions exécutées ainsi que les informations sur les variables utilisées à l'intérieur d'une méthode.

Précisons aussi que cela signifie qu'un code qui serait ajouté durant la phase d'obfuscation mais ne ferait pas partie du code source sera tout de même analysé par notre approche.

5.2 ÉTAPE 2 : CONVERSION DU MODÈLE DE PAPRIKA EN GRAPHE

5.2.1 *Entrée*

Un modèle PAPRIKA contenant entités, propriétés, métriques et relations.

5.2.2 *Sortie*

Un modèle sous forme de graphe stocké dans une base de données.

5.2.3 *Description*

Contrairement aux approches habituelles de détection de défauts de code qui consiste à analyser les applications indépendamment, PAPRIKA a été conçu dans l'optique de pouvoir analyser des applications à large échelle afin de profiter de l'ensemble des données pour la détection. Par conséquent, nous avons choisi de stocker les modèles de chaque application analysée dans une base de données. Notre choix s'est porté sur l'utilisation d'une base de données graphe car c'est une solution flexible et efficace, qui correspond bien au modèle annotée de métriques qui est générée par PAPRIKA. En effet, les bases de données graphes ne reposent pas sur un schéma rigide et il nous est possible de représenter chaque entité par un noeud tandis que les métriques et les propriétés sont des attributs de chaque noeud. Les relations entre entités peuvent quant à elles, être représentées par des relations unidirectionnelles entre noeuds du graphe.

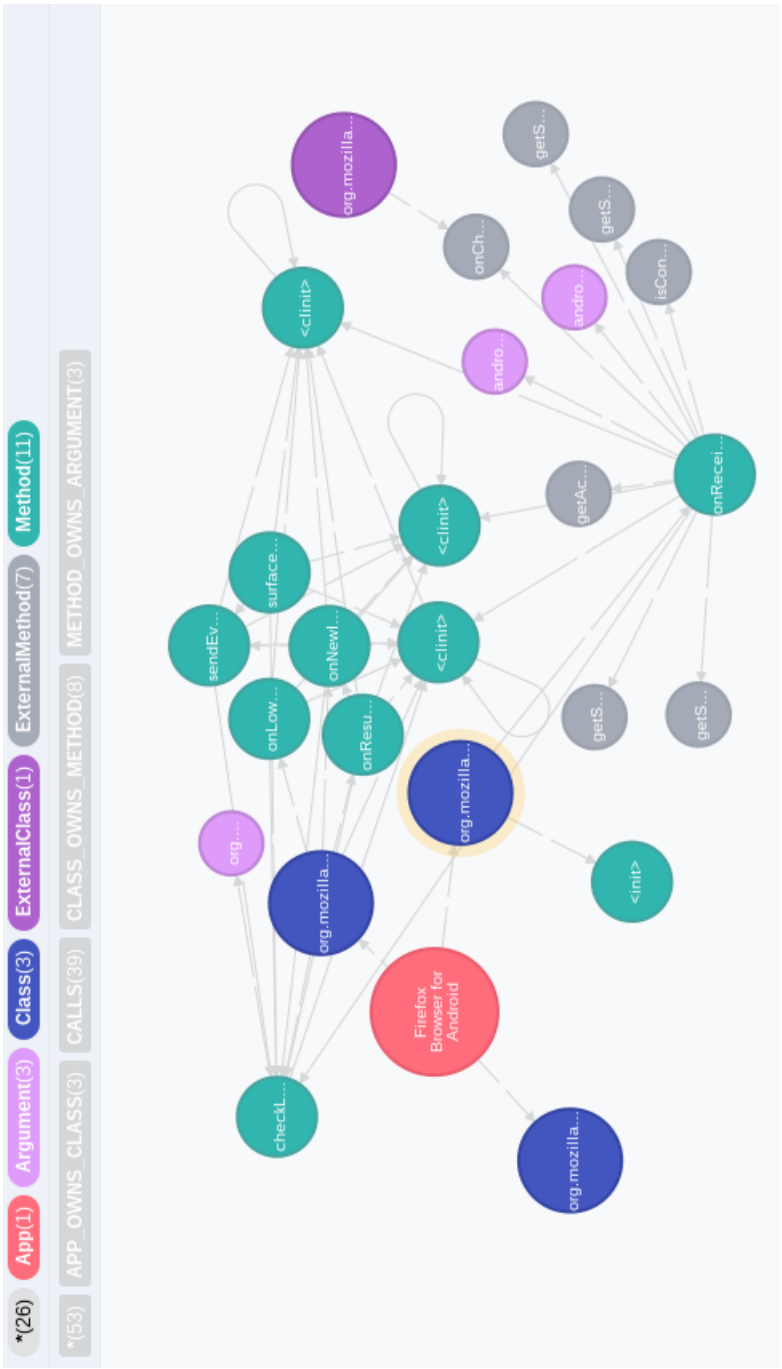
5.2.4 *Implémentation*

Nous avons choisi d'utiliser la base de données graphe NEO4J [91]. Nous utilisons la version embarquée Java de NEO4J, ainsi tous les traitements sont directement intégrés dans PAPRIKA. Notre choix s'est porté sur la technologie NEO4J pour plusieurs raisons :

- Elle offre de très bonnes performances même sur les grand jeux de données, spécialement pour la version embarquée Java [38] ;
- Elle propose le langage de requête CYPHER [90] qui est à la fois simple, flexible et puissant ;
- Au besoin, la base de données propose une interface graphique simple et lisible pour représenter un graphe comme on peut le constater sur la figure 3 ;
- La base peut contenir jusqu'à 2^{35} noeuds et relations, ce qui est largement suffisant pour nos besoins ;
- C'est une des bases de données graphes les plus matures, réputées et utilisées ;
- L'API proposée permet une conversion presque directe du modèle PAPRIKA en graphe.

Afin de diminuer le temps d'exécution sur les requêtes qui concernent des grands jeux de données, nous utilisons plusieurs index. En particulier, nous créons automatiquement un index sur l'identifiant unique qui est présent sur chaque noeud d'une application.

FIGURE 3 – Capture d'écran de la représentation graphe d'une partie de l'application Firefox



5.3 ÉTAPE 3 : DÉTECTION DES DÉFAUTS DE CODE

5.3.1 *Entrées*

Une base de données contenant le modèle des applications à analyser, la requête associée à un défaut de code et un fichier *Fuzzy Control Language* (FCL) optionnel.

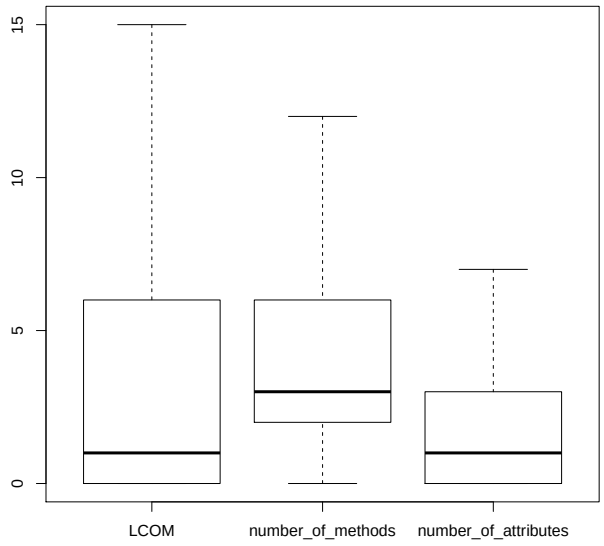
5.3.2 *Sortie*

Un fichier listant les défauts de code détectés dans les applications.

5.3.3 *Description*

Une fois que le modèle de l'application est enregistré et indexé dans la base de données, il est possible d'utiliser le langage de requête de la base pour détecter les défauts de code. Pour les défauts de code subjectifs, PAPRIKA propose un système de logique floue qui est détaillé dans l'implémentation. La requête peut être adapté pour retourner des informations différentes sur le résultats, mais par défaut PAPRIKA propose deux niveaux de détails. Il est ainsi possible de faire des requêtes qui vont agréger les résultats pour chaque application dans la base, par exemple pour recueillir le nombre de MIM détectés dans chaque application. Ce mode de fonctionnement est plutôt destiné à l'analyse de gros volumes d'applications, il permet ainsi de comparer les applications entre elles et d'effectuer des observations statistiques sur les résultats. Le second mode de fonctionnement est plus détaillé, il fournit les informations nécessaires afin de pouvoir identifier précisément la position de chaque instance de défauts de code. Il permet ainsi par exemple de déterminer quelles sont les méthodes qui sont des MIM dans chaque application. On retrouvera aussi les métriques associées aux défauts de code ainsi que les éventuels résultats en cas d'utilisation de logique floue. Ces résultats sont retournés sous forme de fichier CSV. Pour les défauts de code subjectifs qui sont détectés en utilisant des ratios ou des intervalles, nous avons besoin d'un seuil qui va déterminer les valeurs considérées comme anormalement élevées des valeurs qui peuvent être considérées comme normales [14]. Toutefois, l'approche basique qui consiste à considérer qu'un défaut de code est valide dès lors que les seuils sont dépassés est limité. En effet, les résultats obtenus sont alors uniquement booléens. Or, comme évoqué dans le chapitre 4, les avis des développeurs sont rarement aussi tranchés. Pour se rapprocher du raisonnement humain, nous avons donc introduit l'usage de la logique floue dans PAPRIKA. L'utilisation de la logique floue nous permet de donner un indice de confiance lors de la détection des défauts de code subjectifs.

FIGURE 4 – Exemple de boîtes de Tukey utilisées pour la détection de BLOB



5.3.4 *Implémentation*

Que la logique floue soit utilisée ou pas, il est nécessaire de calculer les seuils qui serviront à la détection. Pour cela, nous utilisons une boîte de Tukey [81] qui nous permet d'identifier les valeurs anormalement hautes pour chacune des métriques. En effectuant une seule requête dans la base de données, nous calculons donc le *premier quartile* (Q1), le *troisième quartile* (Q3), ainsi que la distance entre Q1 et Q3, c'est-à-dire *l'étendue interquartile* (IQR). Ce calcul statistique utilise l'ensemble des applications analysées et stockées dans la base de données. Toutes les valeurs supérieures à $Q3 + 1.5 \times IQR$ sont des données aberrantes que nous considérons comme hautes (HIGH), alors que toutes les valeurs supérieures à $Q3 + 3 \times IQR$ sont des données extrêmement aberrantes que nous considérons comme extrêmement hautes (VERY_HIGH). PAPRIKA peut fonctionner avec l'un ou l'autre des seuils pour détecter les défauts de code comme on peut l'apercevoir en sous-sous-section 5.3.4.1. En prenant pour exemple la requête du BLOB Fragment 2 avec des seuils à HIGH, on obtient les boîtes de Tukey sur les trois métriques (LCOM, nombres de méthodes et nombres d'attributs) présenté en figure 4. Ici `lack_of_cohesion_in_methods`=25 tandis que `number_of_methods`=15 et `numbers_of_attributes`=9. Sans logique floue, toutes les entités qui dépassent tous les seuils à la fois sont alors considérées comme des défauts de code.

Le processus est plus raffiné lorsque l'utilisation de la logique floue a été choisie. Dans ce cas, nous utilisons jFuzzyLogic [14] afin de calculer l'indice de confiance du défaut de code. Pour chaque défaut de code, il

est alors nécessaire d'utiliser un fichier écrit en *Fuzzy Control Language* (FCL) qui détermine les paramètres qui seront utilisés pour calculer l'indice de confiance. Fragment 1 présente le fichier FCL utilisé pour la détection de BLOB.

Fragment 1 – Fichier FCL pour la détection de BLOB

```
1
FUNCTION_BLOCK BLOB

VAR_INPUT
        lack_of_cohesion_in_methods : REAL;
6       number_of_methods : REAL;
        number_of_attributes : REAL;
END_VAR

VAR_OUTPUT
11      res : REAL;
END_VAR

FUZZIFY lack_of_cohesion_in_methods
  TERM high := (25, 0) (40, 1);
16 END_FUZZIFY

FUZZIFY number_of_methods
  TERM high := (14.5, 0) (22, 1);
END_FUZZIFY
21
FUZZIFY number_of_attributes
  TERM high := (8.5, 0) (13, 1);
END_FUZZIFY


26
DEFUZZIFY res
        METHOD : LM;
        TERM blob := (0, 0) (1, 1);
        DEFAULT := 0;
31 END_DEFUZZIFY

RULEBLOCK No1
        AND : PROD;
        ACT : MIN;
36      ACCU : MAX;
        RULE 1 : IF lack_of_cohesion_in_methods IS
           high AND number_of_methods IS high AND
           number_of_attributes IS high THEN res IS
            blob;
```

```
END_RULEBLOCK

END_FUNCTION_BLOCK
```

Les noms des variables doivent correspondre entre la requête Cypher et le fichier FCL. On définit tout d'abord les métriques qui seront utilisées ainsi que les seuils. Ici, on définit la fonction d'appartenance à chacune des métriques avec les valeurs numériques des seuils HIGH et VERY_HIGH. Les valeurs inférieures ou égales à HIGH donneront une valeur d'appartenance égale à 0 tandis que les valeurs supérieures à VERY_HIGH auront une valeur d'appartenance égale à 1. On exécute donc une requête Cypher classique avec le seuil le plus bas et les résultats subissent ensuite un traitement pour obtenir l'indice de confiance. Toutes les valeurs comprises entre ces bornes prennent une valeur d'appartenance entre 0 et 1 de manière linéaire (mais il serait possible d'utiliser d'autres modèles comme une courbe de Gauss, par exemple) comme on peut l'apercevoir sur la figure 5. Ensuite, ces valeurs d'appartenance sont combinées selon les règles définies dans le RULEBLOCK, dans ce cas les valeurs sont multipliées les unes avec les autres. De plus, nous considérons qu'une classe est BLOB lorsqu'elle a un manque de cohésion élevée entre ses méthodes ainsi qu'un nombre élevé d'attributs et de méthodes. Enfin, la valeur finale est retournée selon les règles données dans DEFUZZIFY, ici on utilise la règle du sommet le plus à gauche (LM pour *LeftMost*) car pour des valeurs données notre modèle de calcul de logique floue reste assez simple comme on peut l'apercevoir sur la figure 6 (avec pour valeurs `lack_of_cohesion_in_methods`=60, `number_of_methods`=17, `numbers_of_attributes`=17). Tous les fichiers FCL que nous utilisons actuellement incluent des règles similaires. En logique floue, il n'existe pas de règle absolue lors du choix de la logique. En effet, ces choix doivent souvent être définis par un expert du domaine afin de donner un résultat s'approchant du raisonnement humain. Dans notre cas, nous avons choisis ces règles après avoir essayé toutes les combinaisons de paramètres possibles. Cette combinaison nous semble la plus proche du raisonnement que nous effectuons en tant que développeur. En particulier, les résultats fournis avec ces règles utilisent bien tout l'intervalle 0–1 et donnent le même poids à toutes les métriques. Évidemment, il est tout à fait possible de modifier ces règles, notamment pour les adapter à d'autres défauts de code.

En plus des requêtes de détection de défauts de code déjà intégrées, notre outil propose aussi de nombreuses requêtes permettant d'obtenir des informations sur les applications analysées, en particulier des statistiques sur les métriques. De plus, Paprika est facilement extensible puisqu'il permet à l'utilisateur d'exécuter ses propres requêtes écrites avec Cypher.

FIGURE 5 – Fonctions d'appartenances pour le BLOB

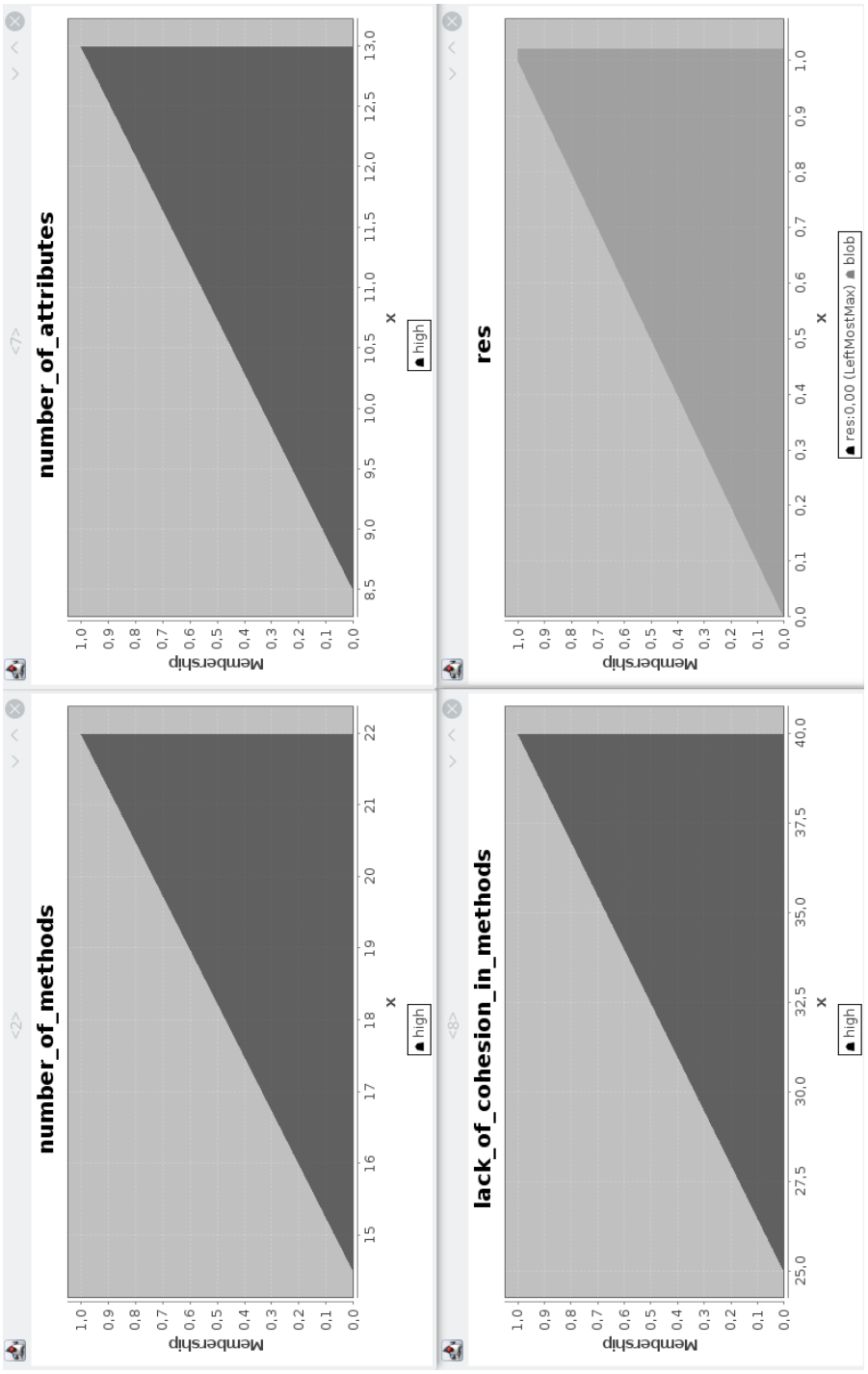
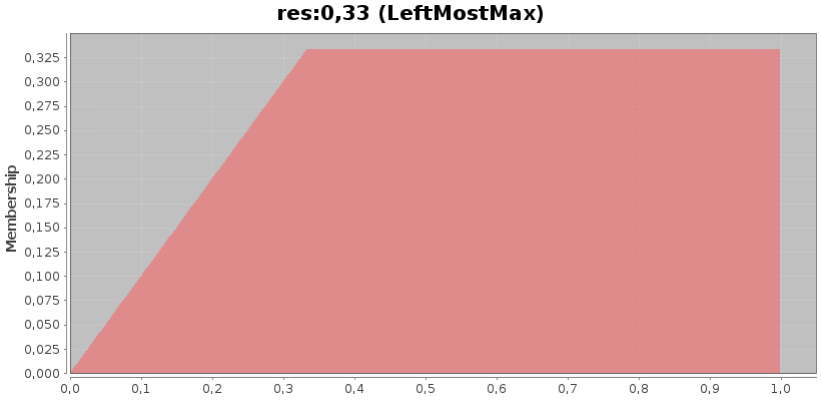


FIGURE 6 – Exemple de fonction résultat pour BLOB



5.3.4.1 *Requêtes*

Dans la présentation des requêtes qui suivent nous utilisons toujours le seuil HIGH, mais il est possible d'utiliser VERY_HIGH ou une valeur numérique à la place de ses seuils. De plus, le retour des requêtes est simplifié afin de faciliter la lecture mais à l'utilisation plus de détails sont fournis ou une agrégation est effectuée afin de compter les instances de défauts de code pour chaque application. L'utilisation de la logique floue repose sur ces requêtes mais une étape supplémentaire est effectuée par Paprika comme présenté précédemment.

Fragment 2 – Requête CYPHER pour la détection de BLOB avec un seuil à HIGH

```
MATCH (cl:Class)
WHERE cl.lack_of_cohesion_in_methods > HIGH
  AND cl.number_of_methods > HIGH
  AND cl.number_of_attributes > HIGH
RETURN cl
```

Pour la détection du *BLOB*, nous retournons les instances de classes qui ont à la fois un nombre élevé de méthodes, d'attributs et un manque de cohésion considéré comme élevées entre les méthodes.

Fragment 3 – Requête CYPHER pour la détection de SAK avec un seuil à HIGH

```
MATCH (cl:Class)
WHERE HAS(cl.is_interface)
  AND cl.number_of_methods > HIGH
RETURN cl
```

Un *Swiss Army Knife* est une interface avec un nombre anormalement élevé de méthodes.

Fragment 4 – Requête CYPHER pour la détection de LM avec un seuil à HIGH

```
MATCH (m:Method)
```

```
WHERE m.number_of_instructions > HIGH
RETURN m
```

Une *Long Method* est une méthode avec un nombre d'instructions élevé.

Fragment 5 – Requête Cypher pour la détection de CC avec un seuil à HIGH

```
MATCH (cl:Class)
WHERE cl.class_complexity > HIGH
RETURN cl
```

De la même manière, une *Complex Class* est une classe avec une haute complexité.

Fragment 6 – Requête Cypher pour la détection de IGS

```
MATCH (cl:Class)-[:CLASS_OWNS_METHOD]->(m1:Method)
   -[:CALLS]->(m2:Method)
WHERE (m2.is_setter OR m2.is_getter)
  AND cl-[:CLASS_OWNS_METHOD]->m2
RETURN m1
```

La requête la plus simple pour détecter les *Internal Getter/Setter* consiste à trouver des méthodes qui appellent une autre méthode qui est un accesseur ou un mutateur tout en s'assurant que ces deux méthodes appartiennent à la même classe.

Fragment 7 – Requête Cypher optimisée pour la détection de IGS sur un ensemble d'applications

```
MATCH (a:App)
WITH a.app_key as key
  MATCH (cl:Class {app_key: key})-[:CLASS_OWNS_
     METHOD]->(m1:Method {app_key: key})-[:CALLS
     ]->(m2:Method {app_key: key})
  WHERE (m2.is_setter OR m2.is_getter)
    AND cl-[:CLASS_OWNS_METHOD]->m2
RETURN m1
```

Bien que la requête précédente fonctionne correctement pour *Internal Getter/Setter*, elle n'est pas très performante sur de gros volumes d'applications. Le graphe d'appel peut être très complexe pour chaque application et les possibilités sont nombreuses. Par défaut, la base de données va aussi possiblement explorer les éventuelles relations entre méthodes qui n'appartiennent pas à une application, ce qui n'est évidemment pas nécessaire. On s'assure donc dans cette requête de n'explorer le graphe d'appel que d'une seule application à la fois. Cette optimisation permet de diminuer le temps d'exécution de la requête pas un facteur 30.

Fragment 8 – Requête Cypher pour la détection de MIM

```
MATCH (m1:Method)
```

```
WHERE m1.number_of_callers>0
  AND NOT HAS(m1.is_static)
  AND NOT HAS(m1.is_override)
  AND NOT m1-[:USES]->(:Variable)
  AND NOT HAS(m1.is_init)
  AND NOT (m1)-[:CALLS]->(:ExternalMethod)
  AND NOT (m1)-[:CALLS]->(:Method)
RETURN m1
```

Bien que comportant beaucoup de conditions, la requête pour *Member Ignoring Method* est assez simple. On se contente de regarder les méthodes appelées qui ne sont pas statiques, non surchargées, qui n'utilisent aucune variable d'instance, ne sont pas des constructeurs, et n'appellent aucune autre méthode externe ou interne à l'application. On exclut ici les appels à d'autres méthodes statiques pour des besoins de simplification de la requête.

Fragment 9 – Requête CYPHER pour la détection de IOD

```
MATCH (:Class{isView:true})-[:CLASS_OWNS_METHOD]->(
   m:Method{name:'onDraw'})-[:CALLS]->({name:'<init
   >'})
RETURN m
```

Pour *Init OnDraw*, il suffit de regarder les méthodes `onDraw` des Vues qui font appel à des constructeurs.

Fragment 10 – Requête CYPHER pour la détection de NLMR

```
MATCH (cl:Class)
WHERE HAS(cl.is_activity)
  AND NOT (cl:Class)-[:CLASS_OWNS_METHOD]->(:Method
      { name: 'onLowMemory' })
  AND NOT cl-[:EXTENDS]->(:Class)
RETURN cl
```

Pour *No Low Memory Resolver*, on détecte les activités qui n'implémentent pas la méthode `onLowMemory`. De plus, on exclut les activités qui héritent d'une autre activité interne de l'application. En effet, celle-ci pourrait implémenter la méthode pour l'ensemble de la hiérarchie et on laisse donc au développeur la responsabilité de savoir quelle classe est la plus pertinente pour l'implémentation de cette méthode.

Fragment 11 – Requête CYPHER pour la détection de LIC

```
MATCH (cl:Class)
WHERE HAS(cl.is_inner_class)
  AND NOT HAS(cl.is_static)
RETURN m
```

Une *Leaking Inner Class* est simplement une classe interne qui n'est pas statique.

Fragment 12 – Requête Cypher pour la détection de UCS

```
Match (m:Method)-[:CALLS]->(e:ExternalMethod {full_
    name:'<init>#android.util.LruCache'})
WHERE NOT (m)-[:CALLS]->(:ExternalMethod {full_name
    :'getMemoryClass#android.app.ActivityManager'})
RETURN m
```

Pour *Unsuited LRU Cache Size*, on recherche les méthodes qui initialisent un `LruCache` depuis l'API Android mais qui ne font jamais appel à `getMemoryClass`. Évidemment, il est nécessaire pour le développeur de s'assurer, qu'à l'exécution, l'appel à `getMemoryClass` est en effet intéressant puisque nous ne disposons pas de la taille au niveau statique.

Fragment 13 – Requête Cypher pour la détection de HMU

```
MATCH (m:Method)-[:CALLS]->(e:ExternalMethod{full_
    name:'<init>#java.util.HashMap'})
RETURN m
```

Pour *Hashmap Usage*, on recherche les méthodes qui utilisent une HashMap de l'API Java. Ici aussi, nous laissons la responsabilité au développeur de vérifier si la taille à l'exécution mérite une correction du défaut de code.

Fragment 14 – Requête Cypher pour la détection de UIO

```
MATCH (:Class{is_view:true})-[:CLASS_OWNS_METHOD
    ]->(m:Method{name:"onDraw"})-[:METHOD_OWNS_
    ARGUMENT]->(:Argument{position:1, name:"android.
    graphics.Canvas"})
WHERE NOT n-[:CALLS]->(:ExternalMethod{full_name:\"
    clipRect#android.graphics.Canvas\"})
  AND NOT n-[:CALLS]->(:ExternalMethod{full_name:\"
      quickReject#android.graphics.Canvas\"})
RETURN m
```

Tout comme pour *Init OnDraw*, nous travaillons ici avec les méthodes `onDraw` des vues. Mais cette fois, pour *UI Overdraw*, on s'intéresse à l'utilisation qui est faite du `Canvas` (la classe qui représente la zone effective qui sera dessinée). En particulier, on obtient une détection dès lors que la méthode `onDraw` ne fait jamais appel à `clipRect` or `quickReject`.

Fragment 15 – Requête Cypher pour la détection de IWR

```
MATCH (:Class{is_view:true})-[:CLASS_OWNS_METHOD
    ]->(m:Method{name:'onDraw'})-[:CALLS]->(e:
    ExternalMethod{name:'invalidate'})
WHERE NOT e-[:METHOD_OWNS_ARGUMENT]->(:
    ExternalArgument)
RETURN m
```

Pour *Invalidate Without Rect*, on s'intéresse aussi aux méthodes `onDraw` des vues. La détection s'effectue dès lors qu'il y a un appel à la méthode externe `invalidate` sans aucun argument. Ce qui signifie qu'aucune zone n'a été définie pour la phase de rafraîchissement et que donc toute la vue sera dessinée.

Fragment 16 – Requête CYPHER pour la détection de UHA

```
MATCH (m:Method)-[:CALLS]->(e:ExternalMethod)
WHERE e.full_name = "drawPicture#android.graphics.
    Canvas"
OR e.full_name = "drawPosText#android.graphics.
    Canvas"
OR e.full_name = "drawTextOnPath#android.graphics.
    Canvas"
OR e.full_name = "drawPath#android.graphics.Canvas"
OR e.full_name = "setLinearText#android.graphics.
    Paint"
OR e.full_name = "setMaskFilter#android.graphics.
    Paint"
OR e.full_name = "setPathEffect#android.graphics.
    Paint"
OR e.full_name = "setRasterizer#android.graphics.
    Paint"
OR e.full_name = "setSubpixelText#android.graphics.
    Paint"
RETURN m
```

Pour *Unsupported Hardware Acceleration*, il suffit de vérifier qu'aucune méthode de l'application ne fait appel à une méthode externe qui n'est pas supportée par le processeur graphique.

Fragment 17 – Requête CYPHER pour la détection de HAS avec un seuil à HIGH

```
MATCH (c:Class{is_async_task:true})-[:CLASS_OWNS_
    METHOD]->(m:Method)
WHERE (m.name='onPreExecute' OR m.name='
    onProgressUpdate' OR m.name='onPostExecute')
  AND  m.number_of_instructions > HIGH
  AND m.cyclomatic_complexity > HIGH
RETURN m
```

Pour *Heavy AsyncTask* on détecte les méthodes spécifiques aux `AsyncTask` qui peuvent bloquer le processus principal. Notre heuristique consiste à retourner les méthodes qui sont complexes et ont des instructions nombreuses. Une fois de plus, le développeur doit vérifier la pertinence de la détection puisque le blocage ne peut se produire qu'à l'exécution.

Fragment 18 – Requête CYPHER pour la détection de HSS avec un seuil à HIGH

```
MATCH (c:Class{is_service:true})-[:CLASS_OWNS_
   METHOD]->(m:Method{name:'onStartCommand'})
  AND  m.number_of_instructions > HIGH
  AND m.cyclomatic_complexity > HIGH
RETURN m
```

Même principe que précédemment pour *Heavy Service Start* sauf qu'ici ça ne peut pas s'appliquer sur une méthode.

Fragment 19 – Requête CYPHER pour la détection de HBR avec un seuil à HIGH

```
1 MATCH (c:Class{is_broadcast_receiver:true})-[:CLASS
    _OWNS_METHOD]->(m:Method{name:'onReceive'}
  AND  m.number_of_instructions > HIGH
  AND m.cyclomatic_complexity > HIGH
RETURN m
```

Même principe que précédemment pour *Heavy BroadcastReceiver*.

5.4 ÉTAPE OPTIONNELLE 1 : CORRECTION DES DÉFAUTS DE CODE

5.4.1 *Entrées*

Code source de l'application, liste de défauts de code détectés par PAPRIKA.

5.4.2 *Sortie*

Code source de l'application sans les défauts de code.

5.4.3 *Description*

PAPRIKA propose actuellement la correction automatique de trois défauts de code, à savoir *Internal Getter/Setter*, *Member Ignoring Method* et *Hashmap Usage*. Le choix de ces défauts de code n'est pas un hasard. En effet, il existe une manière de corriger systématiquement chacun de ces défauts sans affecter le comportement ou la conception de l'application. De plus, nous avons choisi les deux premiers car ils font partie de la catégorie critique, et donc devraient être toujours corrigés dans les applications. Nous avons aussi choisi *Hashmap Usage* car, bien qu'il ne soit que problématique, nous avons remarqué qu'il est bénéfique de le corriger dans la majorité des cas lors de nos expérimentations. Cette correction optionnelle est quelque peu externe au processus de PAPRIKA car, contrairement à l'approche originale, il est nécessaire de travailler sur le code source afin d'effectuer les corrections.

5.4.4 *Implémentation*

La correction s'effectue à l'aide de SPOON [65]. SPOON est une librairie qui permet l'analyse statique et la transformation de code source Java. Pour la transformation, SPOON utilise un modèle sous forme d'arbre abstrait de syntaxe (AST) qu'il génère à partir du code source. Cette AST est accessible de manière simplifiée au travers d'une API au développeur qui peut manipuler le modèle à sa guise. À la manière de ce qui est proposé par SOOT, SPOON permet de naviguer et d'accéder aux éléments du programme comme les classes, les méthodes ou la ligne de code au travers de son modèle interne. L'utilisateur doit implémenter les transformations qu'il souhaite effectuer sur l'application à l'aide d'un processeur SPOON. Ce processeur est en fait un programme indépendant qui va parcourir l'application avant sa compilation, et effectuer les transformations sur le code source selon les filtres utilisés par le développeur. Dans notre cas, c'est la détection effectuée par PAPRIKA qui sert de filtre. La transformation va donc s'effectuer uniquement sur les instances de méthode détectée par notre approche.

Il est donc nécessaire d'écrire un processeur SPOON pour chaque type de défaut de code. Chaque processeur est ensuite exécuté indépendamment afin d'effectuer la correction.

L'intégration à un projet est très aisée, il suffit d'installer SPOON, son plugin GRADLE et les processeurs fournis par PAPRIKA dans le dépôt MAVEN de la machine qui va compiler le projet. Ensuite, il suffit d'ajouter la dépendance au module d'extension SPOON-ANDROID et la liste des processeurs que l'on souhaite utiliser pour le projet. À chaque compilation du projet, la correction s'effectuera automatiquement sur le code source.

Notons ici que nous avons choisi SPOON plutôt que SOOT car il est plus aisé pour transformer le code. De plus, nous souhaitions pouvoir effectuer les transformations sur le code source afin que les développeurs puissent en profiter lors des futures évolutions de l'application. A l'inverse, il aurait été possible de d'utiliser SPOON plutôt que SOOT lors de la phase d'analyse de PAPRIKA. Nous avons d'ailleurs envisagé plusieurs fois d'utiliser un moteur d'analyse basée sur SPOON. Toutefois, au début du développement de PAPRIKA, SPOON n'était pas totalement compatible avec Android et surtout cela nous empêcherait de pouvoir analyser les applications sans disposer du code source. Or les applications les plus populaires sur Android sont pour la plupart propriétaires comme on peut l'observer dans notre ensemble de données du chapitre 9. Nous avons aussi constaté que les applications libres disponibles sont généralement de plus petite taille et ne sont maintenues que par quelques développeurs. Nous avons donc conservé SOOT pour l'analyse afin de ne pas brider les possibilités de PAPRIKA, ce qui nous a permis de faire des études sur des ensembles de données représentatifs du Google Play Store comme nous le verrons dans les chapitres suivants.

5.4.5 *Exemple de processeur IGS*

Le Fragment 20 est un exemple d'utilisation de processeur SPOON pour corriger *Internal Getter/Setter*. On a ici les lignes essentielles de la correction. Tout d'abord, la méthode surchargée `isToBeProcessed` va déterminer quelles sont les méthodes à transformer. Notre implémentation consiste à transformer uniquement les méthodes qui apparaissent dans le fichier de résultats de PAPRIKA en comparant le nom de la méthode, la classe d'appartenance et les types d'arguments.

Enfin, la méthode `process` va effectuer la transformation du code. Après un parcours de la méthode pour déterminer à quel endroit s'effectue l'appel du mutateur ou de l'accesseur, on le remplace simplement par un accès direct au champ de classe grâce aux lignes apparaissant dans le bloc try.

Fragment 20 – Extrait du processeur corrigeant les IGS

```
public class IGSProcessor extends AbstractProcessor
    <CtInvocation> {
        ...

        @Override
     public boolean isToBeProcessed(CtMethod invok)
        {
          ...
        }

        @Override
   public void process(CtMethod invok) {
                ...
                try{
                        CtStatement igsSetter =
                           getFactory().Code().
                           createCodeSnippetStatement
                           (
                              getField + " = " +
                                 invok.getArguments
                                 ().get(0));
        invok.replace(igsSetter);
    }
                ...
        }
```

5.5 ÉTAPE OPTIONNELLE 2 : CALCUL DE LA QUALITÉ DE L'APPLICATION

5.5.1 *Entrée*

Liste des défauts de code détectés pour chaque application.

5.5.2 *Sortie*

Score de qualité, éventuellement sur plusieurs versions de l'application.

5.5.3 *Description*

En plus de son fonctionnement classique qui fournit les informations sur le nombre de défauts de code détectés et leurs localisations, PAPRIKA permet aussi à l'utilisateur de comparer la qualité des différentes versions de son application par rapport à l'ensemble des applications analysées. À cet effet, PAPRIKA fournit un score de qualité pour chaque type de défauts de code d'une application. Ce score est basé sur la relation qui existe entre le nombre de défauts de code d'une application et le nombre d'entités qui constituent l'application. Les relations sont présentées dans le tableau 5. Par exemple, il existe une relation entre le nombre de *Heavy BroadcastReceiver* et le nombre de BroadcastReceiver. Dans sa dernière version, PAPRIKA propose aussi un score global en fonction du nombre total de défauts de code et de la taille de l'application analysée, c'est-à-dire son nombre de classes. De plus, lorsque plusieurs versions d'une même application sont analysées, PAPRIKA propose un suivi des scores dans le temps qui permet de déterminer si une application a tendance à augmenter ou diminuer sa qualité en terme de défauts de code.

5.5.4 *Implémentation*

Pour calculer nos scores de qualité, nous bâtissons un modèle d'estimation du nombre de défauts de code d'une application en utilisant une régression linéaire. Le modèle issu de la régression linéaire représente donc la relation entre le nombre de défauts de code et le nombre d'entités d'une application. Cette régression est estimée par la méthode des moindres carrés que l'on applique sur l'ensemble des applications analysées par PAPRIKA. Comme nous le verrons dans le chapitre 9 et le chapitre 10, les régressions que nous exploitons ainsi ont un coefficient de corrélation supérieur à 0.80 et sont donc significatives. En particulier, dans le cas de la régression entre ce nombre total de défauts de code et nombre de classes, nous obtenons un coefficient de corrélation de 0.96 sur notre base de données contenant plus de 3000 applications

Tableau 5 – Relation entre défauts de code et nombre d'entités

Défauts de code	Entités associées
BLOB	Classe
Swiss Army Knife	Interface
Long Method	Méthode
Complex Class	Classe
Internal Getter/Setter	Méthode
Member Ignoring Method	Méthode
No Low Memory Resolver	Activité
Leaking Inner Class	Classe interne
Unsuited LRU Cache Size	Classe
Hashmap Usage	Classe
Init OnDraw	Vue
Invalidate Without Rect	Vue
Unsupported Hardware Acceleration	Classe
Heavy AsyncTask	Asyntask
Heavy Service Start	Service
Heavy BroadcastReceiver	BroadcastReceiver
Tous	**Classe**

populaires du Google Play Store. Cette régression nous sert donc de référence pour estimer le nombre de défauts de code qu'une application possède en fonction de sa taille. Évidemment, bien que cette régression soit globalement vraie pour l'ensemble de la base de données, il existe des différences par rapport à cette référence pour chaque application. C'est cette différence que nous exploitons pour calculer notre score de qualité. Nous mesurons cette différence en calculant la valeur résiduelle pour chaque application (c'est-à-dire, la différence entre le résultat de l'application et la valeur que nous devrions obtenir avec notre modèle), comme illustré sur la figure 7. Par exemple, la seconde application de ce graphique possède une valeur résiduelle de -10. Pour calculer le score, nous prenons l'opposé de la valeur résiduelle. Ainsi plus la valeur résiduelle est grande positivement, c'est-à-dire plus il y a de défauts de code par rapport au nombre d'entités, et plus le score de qualité de l'application est faible. À l'inverse, une application qui aura moins de défauts de code en comparaison de notre référence aura un score de qualité élevé. Ainsi, il est possible de comparer chacune des applications avec l'ensemble des autres applications.

De plus, dans le cas des différentes versions d'une application, nous procédons à une agrégation du score à travers le temps. Pour cela, pour une version X, nous calculons la moyenne des scores de toutes les ver-

sions précédentes et de la version X. Ainsi, le score de qualité donné pour une version X prend en compte le score des versions précédentes. Il est alors possible de donner une tendance de l'évolution de la qualité de l'application dans le temps comme nous le verrons dans le chapitre 10.

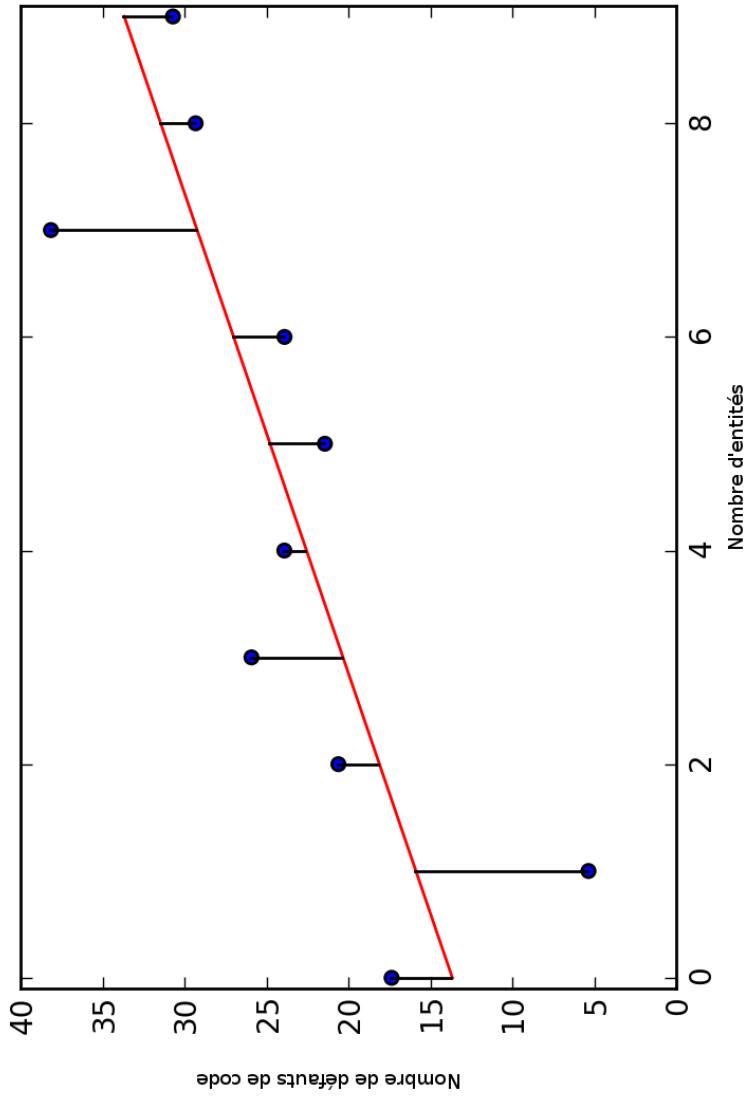


FIGURE 7 – Valeurs résiduelles issues d'une régression linéaire

6

MESURE DE L'IMPACT DES DÉFAUTS DE CODE SUR LES PERFORMANCES

Dans le cadre de cette thèse, nous l'avons utilisé de manière complémentaire à Paprika afin de mesurer l'impact de la présence des défauts de code sur les performances d'une application et nous la décrirons donc dans ce contexte. Toutefois, les phases de mesures et de comparaisons restent génériques et pourraient être utilisées aisément à d'autres fins.

Dans notre cas, les différentes versions d'une application sont des versions où uniquement certains types ou tous les défauts de code ont été corrigés. En exécutant de manière répétée un même scénario pour toutes ces applications, nous sommes capables d'observer des différences sur des métriques liées à la performance et ensuite de déterminer les versions les plus performantes. Les métriques de performance que nous utilisons sont liées à l'expérience de l'utilisateur. En effet, nous utilisons des métriques qui concernent l'affichage (temps d'affichage d'une trame et nombre de trames différées) qui peuvent affecter la fluidité de l'application en ajoutant des ralentissements visibles par l'utilisateur, ainsi que des métriques sur l'utilisation de la mémoire (quantité de mémoire utilisée et nombre d'appels au ramasse-miettes) qui peuvent ralentir le chargement et diminuer la réactivité de l'application. Notre processus de mesure peut être divisé en trois étapes, comme on peut l'apercevoir dans la figure 8.

La première étape fait appel à Paprika et à ses fonctionnalités de détection et de correction de code. Le but ici étant de produire des versions de l'application qui ne contiendront pas un seul type ou l'ensemble des types de défauts de code afin de pouvoir les comparer. Notons que Curry a été développé avant l'automatisation de la correction des défauts dans Paprika, et que par conséquent nous effectuons ici une correction manuelle. Bien que cela soit toujours possible, il est maintenant possible de profiter des fonctionnalités de correction automatique de Paprika pour produire ces versions.

La deuxième étape est une phase qui consiste à exécuter n fois un scénario d'utilisation de l'application fourni par l'utilisateur tout en collectant les métriques de performances précédemment évoquées.

Enfin, dans la troisième étape, nous agrégeons les données brutes de ces expérimentations afin d'en extraire des métriques raffinées, ces nouvelles métriques subissent ensuite des tests statistiques afin de fournir les résultats qui permettront de déterminer l'application la plus performante sur chacune des métriques.

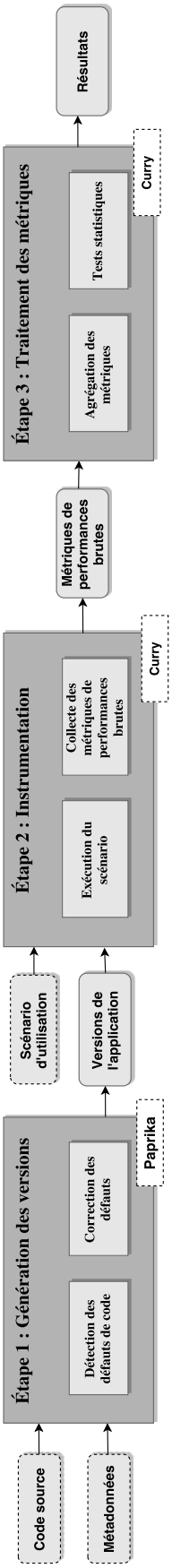


FIGURE 8 – Aperçu de l'approche de mesure des performances

6.1 ÉTAPE 1 : GÉNÉRATION DES VERSIONS

6.1.1 *Entrées*

Le code source de l'application et les métadonnées correspondantes.

6.1.2 *Sortie*

Plusieurs versions de l'application dont un seul type ou tous les défauts de code ont été corrigés.

6.1.3 *Description*

La première étape consiste à produire les différentes versions qui seront utilisées lors de la phase d'instrumentation. Ces versions doivent être similaires à l'application d'origine en conservant notamment le même comportement lors de l'exécution, toutefois les défauts de code sont corrigés dans chacune des versions. Nous produisons une version pour chacun des types de défauts de code corrigés, ainsi qu'une version où tous les défauts de code sont corrigés.

Nous produisons aussi une version qui va consigner chaque invocation d'un défaut de code lors de l'exécution de l'application. Cette version un peu particulière, servira à interpréter plus finement les résultats finaux en déterminant quels sont les défauts de code les plus appelés durant le scénario.

Nous conservons aussi la version qui ne contient aucune correction, qui sera elle aussi instrumentée et servira de référence lors des comparaisons.

6.1.4 *Implémentation*

Pour l'implémentation nous utilisons PAPRIKA. La première étape consiste à détecter les défauts de code dont nous souhaitons évaluer l'impact. Puis, dans un second temps, à produire les différentes versions en supprimant les défauts de code. Dans les cas où PAPRIKA ne fournit pas une correction automatique, il est alors possible de produire les versions manuellement en se basant sur les résultats de PAPRIKA. Nous avons d'ailleurs procédé de cette manière pour l'étude présentée dans le chapitre 11, toutefois notons que la correction manuelle est une étape longue, pénible et sujette à erreurs.

6.2 ÉTAPE 2 : INSTRUMENTATION

6.2.1 *Entrées*

Les différentes versions de l'application ainsi qu'un scénario d'utilisation de l'application.

6.2.2 *Sortie*

Un ensemble de fichier contenant les métriques de performances brutes.

6.2.3 *Description*

Cette phase consiste à exécuter un scénario d'utilisation de l'application n fois pour chaque version afin d'obtenir les métriques de performances brutes, c'est-à-dire le temps d'affichage de chaque trame, la quantité de mémoire utilisée à intervalle de temps régulier ainsi que le registre d'utilisation du ramasse-miettes. Nous obtenons donc chacune de ces données dans un fichier différent pour chaque exécution.

Un scénario d'utilisation unique est à fournir pour l'application. C'est un scénario de tests automatiques de l'application qui va simuler le comportement d'un utilisateur utilisant l'application. Il est possible de le produire avec différents outils d'automatisation de tests comme ROBOTIUM [1] ou CALABASH [2]. Nous recommandons un scénario le plus complet possible qui explore le maximum de fonctionnalités de l'application et qui simule les enchaînements que ferait un utilisateur de l'application. Ainsi les résultats obtenus reflètent au plus près l'expérience de l'utilisateur. Ce scénario s'exécute sur un périphérique Android connecté à l'ordinateur utilisant CURRY. CURRY se charge de le lancer n fois pour chaque version tout en collectant les métriques.

6.2.4 *Implémentation*

Ce processus est implémenté grâce à un script Python qui exécute des commandes de débogage sur le téléphone via ADB (*Android Debug Bridge*), comme nous pouvons l'apercevoir dans la figure 9. Chaque expérimentation consiste à lancer en parallèle le scénario d'utilisation et les commandes nécessaires à la collecte des données.

La commande `Logcat` est lancée au début de chaque expérimentation, elle tourne en continu et nous obtenons un fichier qui liste les différentes types d'appels au ramasse-miettes.

La commande `Gfxinfo` est lancée à chaque seconde, elle permet de collecter toutes les informations sur le temps d'affichage des trames.

1. http://www.robotium.org
2. http://calaba.sh

Cette commande fonctionne comme un tampon qui affiche les informations sur les 128 dernières trames dessinées, ce tampon est vidé à chaque appel de la commande. En utilisant une seconde comme délai, on s'assure donc de collecter les informations sur toutes les trames avec une certaine marge puisqu'il n'y a jamais plus de 60 trames par seconde.

Enfin, nous lançons la commande `Meminfo` pour obtenir la quantité de mémoire utilisée toutes les deux secondes, il est possible de configurer ce paramètre dans le script.

Le script se charge aussi de relancer l'expérimentation autant de fois que demandé en fermant l'application et en nettoyant la mémoire entre deux expérimentations. Notons que si une expérimentation se termine de manière involontaire (par exemple à cause d'une erreur de l'application), le script se relance pour obtenir n expérimentations valides.

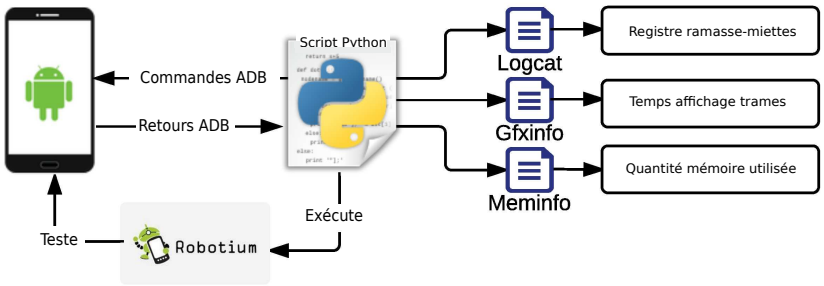


FIGURE 9 – Processus de collecte des données de CURRY

6.3 ÉTAPE 3 : TRAITEMENT DES MÉTRIQUES

6.3.1 *Entrées*

Fichiers contenant les métriques de performances brutes.

6.3.2 *Sortie*

Résultats sur les performances de chaque version d'une application.

6.3.3 *Description*

Les informations obtenues par les commandes de débogage ne sont toutefois pas exploitables en l'état et nous devons donc les mettre en forme afin de pouvoir effectuer nos tests statistiques. De plus, il est nécessaire d'agréger les résultats des n expérimentations. Pour cela, nous les combinons dans un seul fichier CSV après traitement. Enfin, c'est dans cette phase que nous effectuons des tests statistiques qui nous permettront de déterminer la version plus performante sur chacune des métriques.

Pour chacune des métriques, nous effectuons tout d'abord une moyenne sur les n expérimentations ce qui nous permet d'obtenir aussi les différences en pourcentage entre les versions, en particulier entre la version non corrigée et les versions contenant des corrections.

Ensuite, concernant le temps d'affichage des trames ainsi que la quantité de mémoire utilisée, nous utilisons le test statistique de Mann-Whitney U [73]. C'est un test non-paramétrique qui permet de déterminer si deux ensembles de valeurs indépendants sont similaires ou si l'une des deux distributions tend à avoir des valeurs supérieures à l'autre. Nous avons choisi ce test car il est adapté au grand nombre de valeurs que nous possédons pour ces métriques à chaque expérimentations et aussi parce qu'il ne fait aucune hypothèse sur le type de distributions des deux ensembles. Ce test retourne une valeur entre 0 et 1, une valeur proche de 0 signifie que les ensembles sont différents tandis qu'une valeur proche de 1 qu'ils sont similaires. Nous exécutons généralement nos tests avec un niveau de confiance à 95%, c'est-à-dire avec une p-value < 0.05.

Concernant le nombre d'appels au ramasse-miettes et le nombre de trames différées, nous ne disposons que d'une valeur par expérimentation. Nous avons donc choisi un test non-paramétrique, qui lui aussi ne fait aucune hypothèse sur la distribution, mais qui est plus adapté pour de plus petits ensembles de valeurs (à partir de dix valeurs), à savoir le test de taille d'effet Cliff's δ [70]. Les résultats de ce test nous permettent de savoir si deux ensembles sont disjoints. En particulier, il retourne une valeur qui va de -1 (si toutes les valeurs du premier groupe sont inférieures à celle du second groupe) à $+1$ (si toutes les valeurs du premier groupe sont supérieures aux valeurs du second groupe). La valeur est égale à 0 si les deux groupes sont identiques [15].

Il est plus courant d'utiliser le test de Cohen's d pour mesurer une taille d'effet, toutefois Cliff's δ est réputé plus robuste et fiable [70]. Il existe toutefois une correspondance entre les résultats de ces deux tests (en regardant le pourcentage de valeurs des ensembles qui ne se chevauchent pas) qui facilite l'interprétation des résultats comme on peut l'apercevoir dans le tableau 6. Cohen [16] stipule que un effet de taille moyen représente une différence qui sera visible pour un observateur attentif, alors qu'un effet de taille fort sera beaucoup plus visible. Un effet de faible taille est existant mais pas forcément visible pour l'observateur.

Tableau 6 – Correspondance entre Cohen's d to Cliff's δ.

Norme de Cohen	Cohen's d	% de non-chevauchement	Cliff's δ
Faible	0.20	14.7%	0.147
Moyen	0.50	33.0%	0.330
Fort	0.80	47.4%	0.474

6.3.4 *Implémentation*

C'est ici aussi un script en Python qui se charge du traitement des données et de l'exécution des tests statistiques. Concernant le traitement des données, il est nécessaire de compter et de filtrer les appels au ramasse-miettes, nous excluons par exemple les appels volontaires au ramasse-miettes puisqu'ils sont souhaités par le développeur et ne sont pas affectés par la présence de défauts de code. Le temps d'affichage des trames doit être calculé pour chaque trame en additionnant les valeurs de sous-processus de l'affichage (temps de calcul, temps d'exécution, temps de dessin) et ces valeurs doivent être extraites des résultats de `Gfxinfo` qui contient beaucoup de bruit. De même pour `Meminfo`, même si cette fois il est possible d'extraire directement la quantité de mémoire utilisée. Nous calculons le nombre de trames différées en observant le nombre de trames qui mettent plus de 16.67 ms à s'afficher. Il est possible que plus d'une trame soit différée à la fois et donc que l'effet s'accumule sur plusieurs trames. Ensuite, ces résultats sont sauvegardés sous forme de fichier CSV et les tests statistiques sont lancés sur ses fichiers. Les résultats sont alors disponibles pour l'utilisateur à qui nous laissons le soin de les interpréter, comme nous le faisons dans le chapitre 11.

7

MESURE DE L'IMPACT DES DÉFAUTS DE CODE SUR LA CONSOMMATION D'ÉNERGIE

Au cours de cette thèse, nous avons aussi choisi de mesurer l'impact des défauts de code sur la consommation d'énergie. Pour cela, nous proposons une approche similaire à celle du chapitre précédent comme le montre la figure 10. Les différences se trouvent dans les étapes 2 et 3 qui font appel à l'approche Hot-Pepper afin de pouvoir mesurer la consommation d'énergie. En particulier, bien que la deuxième étape utilise aussi un scénario d'utilisation, il est ici nécessaire d'utiliser un support physique (un ampèremètre) pour mesurer l'intensité du courant à intervalle régulier. Les mesures fournies par cet équipement nous permettent de calculer la consommation d'énergie de chaque version durant la troisième étape, et ainsi de comparer les résultats d'une manière similaire à ce qui a été vu précédemment.

7.1 ÉTAPE 1 : GÉNÉRATION DES VERSIONS

7.1.1 *Entrées*

Le code source de l'application et les métadonnées correspondantes.

7.1.2 *Sortie*

Plusieurs versions de l'application dont un seul type ou tous les défauts de code ont été corrigés.

7.1.3 *Description*

Cette étape est totalement similaire à la première étape du chapitre précédent (section 6.1), de même pour l'implémentation.

7.2 ÉTAPE 2 : INSTRUMENTATION

7.2.1 *Entrées*

Les différentes versions de l'application ainsi qu'un scénario d'utilisation de l'application.

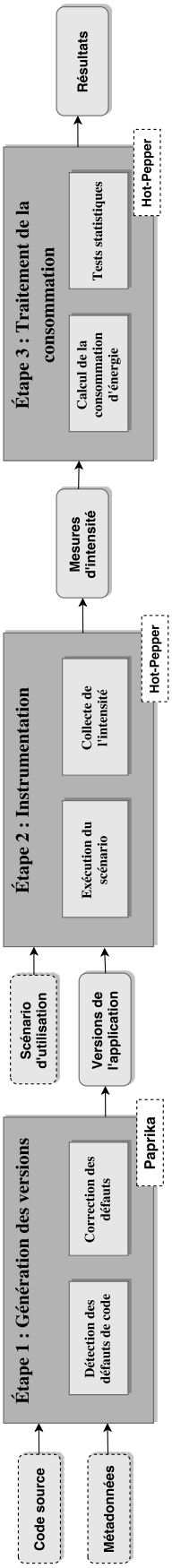


FIGURE 10 – Aperçu de l'approche de mesure de la consommation d'énergie

7.2.2 *Sortie*

Un ensemble de fichiers contenant les métriques de performances brutes.

7.2.3 *Description*

La phase d'instrumentation diffère de celle de CURRY par l'utilisation nécessaire d'un ampèremètre afin de mesurer l'intensité du courant fourni par la batterie. Il est nécessaire de brancher ce périphérique en série entre la batterie et le téléphone, ce qui nécessite d'effectuer un montage comme sur la figure 11 et sur la photo de la figure 12. Nous utilisons l'ampèremètre Yocto-Amp [1] mais il serait possible d'utiliser d'autres modèles. C'est un montage assez intrusif et qui nécessite un peu de matériel, mais c'est un moyen fiable d'obtenir la consommation d'énergie. Nous avons aussi envisagé l'utilisation de solutions logicielles telles que BATTERY HISTORIAN [2] de Google mais ces outils ne sont pas assez précis pour mesurer une différence lors de nos expérimentations. La solution physique est d'ailleurs utilisée par les ingénieurs de Mozilla qui fournissent les plans des circuits et des supports afin de faciliter le montage [3]. HOT-PEPPER propose un serveur qui va communiquer avec l'ampèremètre et récupérer les valeurs à la fréquence choisie par l'utilisateur (75 mesures par seconde par défaut). Ce serveur se charge, tout comme CURRY, de gérer l'exécution du scénario en boucle et il collecte aussi le temps d'exécution pour chaque expérimentation.

7.2.4 *Implémentation*

Le serveur est codé en Python et utilise l'interface de programmation du Yocto-Amp pour communiquer avec ce dernier. Nous récupérons la valeur de l'ampèremètre à intervalle régulier tout au long du déroulement de l'expérimentation. La batterie ne peut pas être chargée durant les expérimentations, par conséquent elle va se décharger et il est généralement impossible de faire les n expérimentations en une fois. Nous effectuons donc cela en plusieurs phases de x expérimentations. Afin d'obtenir les meilleurs résultats possibles, nous prenons aussi les précautions suivantes que nous recommandons à toute personne souhaitant utiliser une approche similaire :

1. Désactiver le contrôle automatique de la luminosité sur le téléphone et la mettre au minimum,
2. Désactiver ou supprimer les applications qui se lancent en tâche de fond,

1. Yocto-Amp : http://www.yoctopuce.com/EN/products/usb-electrical-sensors/yocto-amp
2. BATTERY HISTORIAN : https://github.com/google/battery-historian
3. Supports aux montages : https://github.com/JonHylands/fxos-battery-harness

3. Désactiver le Bluetooth, le GPS et les autres composants qui pourraient consommer de l'énergie mais ne sont pas nécessaires à l'application,
4. Mettre le son au minimum, sans le couper,
5. Mettre un fond d'écran noir et non dynamique,
6. Utiliser une version d'Android la plus épurée possible, sans surcouche lourde,
7. Utiliser un réseau Wi-Fi stable et non pas le réseau de la carte SIM (couper le Wi-Fi si non nécessaire),
8. Charger la batterie au maximum avant les expérimentations, et recharger au maximum à intervalle régulier toutes les x expérimentations,
9. Éteindre le téléphone pendant 5 minutes après toutes les x expérimentations, afin de laisser les composants refroidir,
10. Utiliser une connexion pour USB plutôt que Wi-Fi pour lancer les scénarios.

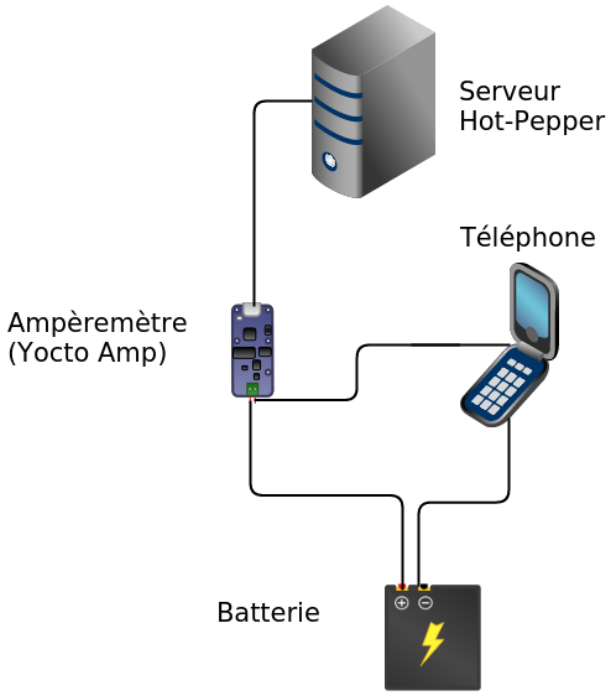


FIGURE 11 – Schéma du montage nécessaire afin de mesurer la consommation d'énergie

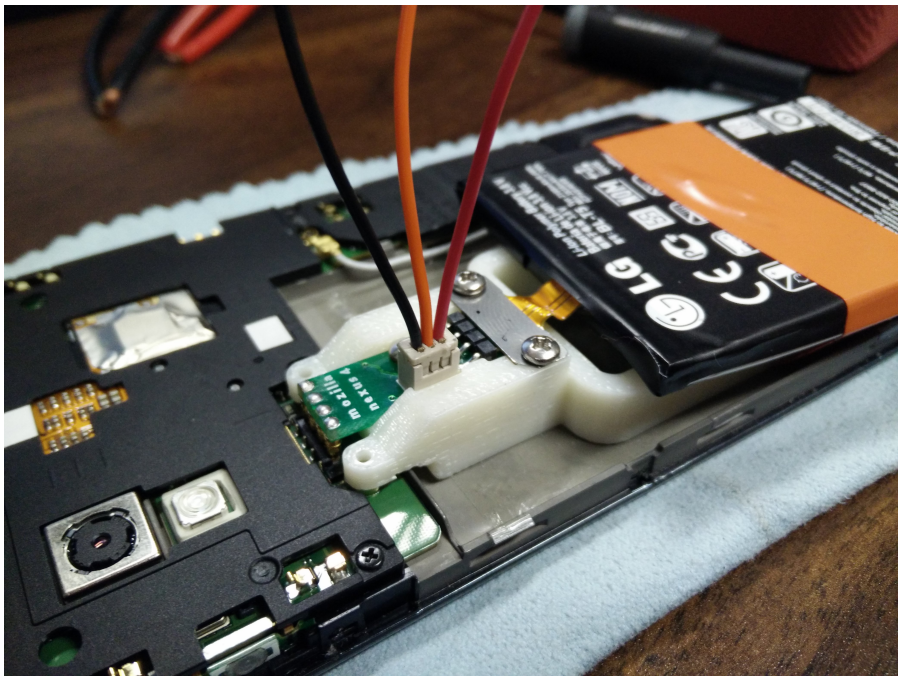


FIGURE 12 – Photo du montage nécessaire pour se connecter entre la batterie et le téléphone

7.3 ÉTAPE 3 : TRAITEMENT DES MÉTRIQUES

7.3.1 *Entrées*

Fichiers contenant les métriques de performances brutes.

7.3.2 *Sortie*

Résultats sur les performances de chaque version d'une application.

7.3.3 *Description*

Cette étape est similaire à ce qui est effectué sur CURRY. Toutefois il existe quelques différences, tout d'abord il s'agit ici de calculer une quantité d'énergie en utilisant les mesures d'intensité. Pour cela, nous utilisons l'équation 1 du chapitre 2 en prenant la somme des mesures d'intensité multipliée par la durée de l'intervalle de mesure et par le voltage (constant et dépendant des spécifications de la batterie). Nous obtenons ainsi une quantité d'énergie consommée pour chaque expérimentation, nous utilisons donc la moyenne et Cohen's $\mathbf{d}$ qui sont adaptés pour ce nombre de mesures.

7.3.4 *Implémentation*

Ces calculs sont directement gérés par HOT-PEPPER, l'interprétation des résultats est laissée à l'utilisateur, toutefois nous suggérons tout de même automatiquement la version la plus efficace en fournissant le code code source corrigé ainsi que l'APK. On peut retrouver un exemple d'utilisation dans le chapitre 12.

Quatrième partie

VALIDATIONS ET EXPÉRIMENTATIONS

8 VALIDATION DE PAPRIKA

Dans ce chapitre, nous présentons les diverses validations effectuées afin de prouver l'efficacité de notre approche de détection des défauts de code. Dans un premier temps, la validation a été effectuée sur une application témoin puis, par la suite, nous avons effectué une validation empirique sur différentes applications en demandant l'avis d'experts sur les résultats de Paprika. Le but de la première validation est de s'assurer que les défauts de code renvoyés par Paprika correspondent bien à l'intention des requêtes présentées dans le chapitre 5, tandis que les deux validations suivantes valident aussi cet aspect technique mais aussi la validité des définitions et la pertinence de l'utilisation de la logique floue en demandant l'opinion d'experts.

8.1 VALIDATION SUR UNE APPLICATION TÉMOIN

Dans cette section, nous présentons les résultats d'une validation empirique que nous effectuons sur une application témoin afin de garantir l'efficacité de Paprika. Il est à noter que cette étude détaillée a été publiée en milieu de thèse et ne concerne que 8 des 17 défauts de code présentés précédemment avec le seuil HIGH et sans logique floue. Toutefois nous utilisons toujours l'application témoin actuellement afin de tester Paprika techniquement, en nous assurant que la précision et le rappel soient toujours à 1 pour l'ensemble des défauts de code et en émettant l'hypothèse que le seuil HIGH soit valide pour la détection. On ne pose donc pas ici la question de la subjectivité des seuils choisis qui est abordée dans les validations suivantes.

8.1.1 *Sujets*

Nous appliquons Paprika sur 8 défauts de code, à savoir *BLOB*, *Swiss Army Knife*, *Complex Class*, *Leaking Inner Class*, *Long Method*, *Internal Getter/Setter*, *Member Ignoring Method* et *No Low memory Resolver*.

8.1.2 *Objet*

L'application témoin est l'objet de cette étude. Nous avons développé cette application afin qu'elle contienne tous les types de défauts de code, en veillant à introduire des difficultés pour la détection (par exemple en jouant avec l'héritage ou les surcharges de méthodes). Lors de cette étude, l'application contenait 62 instances de défauts de code parmi les

8 listés précédemment. Le code source de cette application est disponible sur GitHub [1]. La version étudiée possède 51 classes, dont 6 activités ainsi que 205 méthodes. Les classes générées à la compilation comme `BuildConfig` et `R` sont inclus dans ce compte. Notons qu'elles sont désormais ignorées par PAPRIKA lors de la détection puisque nous considérons que le développeur n'a pas le contrôle sur le code généré.

8.1.3 *Conception expérimentale*

Nous avons effectué une analyse de sensibilité des résultats avec divers seuils, ainsi qu'une comparaison rapide avec d'autres outils et une évaluation de l'impact de l'obfuscation sur notre application témoin. Les seuils statistiques ont été calculés sur les 15 applications présentées dans le tableau 7. Elles ont été téléchargées depuis le Google Play Store en Juin 2013. Il est à noter que les seuils que nous utilisons actuellement sont légèrement différents car calculés sur des ensembles de données contenant plus de 3000 applications.

Tableau 7 – Applications utilisées pour le calcul des seuils pour la validation sur l'application témoin

Nom	**Téléchargements**	**Note**	**Classes**	**Méthodes**
Adobe Reader	100,000,000+	4.3	902	5,214
Android Temperature	500,000+	4	373	2,238
Facebook	500,000,000+	3.9	9,117	46,867
Fitnet Apps	50+	4.9	13	74
FLV HD MP4 Video	1,000,000+	4.3	364	2,332
Free 5000 Movies	1,000,000+	3	3	5
Opera Mini browser	100,000,000+	4.4	182	1,830
Savoir Maigrir	50,000+	2.9	449	2,190
Simulator Laser	5,000,000+	2.1	225	1,205
Skype	100,000,000+	4.1	2,364	12,901
Superbuzzer Trivia	500,000+	3.9	858	5,265
Tcheck'it	5,000+	2.2	1,306	9,268
Twitter	100,000,000+	4.1	4,335	29,309
Video Chat Rooms	100,000+	4.1	7	94
Zoom Camera Free	5,000,000+	4	415	2,131

8.1.4 *Analyse de sensibilité*

Cette analyse de sensibilité utilise plusieurs types de seuils, à savoir la médiane, la moyenne, Q3 et $Q3 + 1.5 \times IQR$ qui sont utilisées pour les

1. Code source de l'application témoin : https://github.com/GeoffreyHecht/paprika-witness-app

défauts de code subjectifs. Les résultats apparaissent dans le tableau 8. Comme nous pouvons nous y attendre, l'utilisation du seuil $Q3 + 1.5 \times IQR$ fournit des résultats parfaits au regard ce que nous attendons de cette application témoin.

Tableau 8 – Analyse de sensibilité sur l'application témoin

Seuil	**Précision**	**Rappel**	**F1-mesure**
Médiane	0.348	1	0.5161
Moyenne	0.8	1	0.8889
Q3	0.7619	1	0.865
Q3+1.5*IQR	1	1	1

Globalement, nous nous attendions à la présence de 62 défauts de code soigneusement ajoutés manuellement dans l'application. L'utilisation du seuil à $Q3 + 1.5 \times IQR$ nous permet de tous les détecter et de n'avoir aucun faux positif, contrairement aux autres seuils qui ont tendance à avoir un rappel qui baisse avec la valeur du seuil.

8.1.5 *Comparaison avec d'autres outils*

Bien que PAPRIKA soit le seul outil capable de détecter des défauts de code spécifiques à Android, nous l'avons comparé à d'autres outils capables de détecter des défauts de code OO, afin de valider nos choix techniques. Les méthodes de détection et les métriques utilisées sont différentes, il n'est donc pas pertinent de comparer les résultats bruts. Nous les comparons donc d'un point de vue pratique. Tout d'abord nous l'avons comparé à PMD [2] qui est un outil populaire pour les applications Java. Il est capable de détecter des défauts de code en fonction de règles prédéfinies. Il contient actuellement 347 règles, dont *ExcessiveMethodLength* qui est similaire à *Long Method*. Cette règle utilise principalement un seuil arbitraire statique qui peut être modifié dans les préférences de PMD. Par défaut, une méthode est classée comme *ExcessiveMethodLength* si elle contient plus de 100 lignes de code. Toutefois, cette valeur semble arbitraire et peu appropriée pour les applications Android puisqu'il s'avère que 75% des méthodes ont moins de 15 instructions [35]. En particulier, seuls trois cas sur les huit que nous détectons apparaissent avec ce seuil. Évidemment, il est toujours possible de modifier ce seuil mais PMD ne fournit aucun mécanisme d'aide avec de déterminer une valeur objective.

Pour résoudre ce problème, d'autres approches comme JDEODORANT [3] ou PTIDEJ [4] font usage d'une analyse statistique sur l'application en cours d'analyse pour déterminer les seuils de détection. Avec ce moyen,

2. PMD : http://pmd.sourceforge.net
3. JDeodorant : http://www.jdeodorant.com
4. Ptidej : http://www.ptidej.net

JDeodorant est capable de détecter une des deux instances de *BLOB* que nous détectons. Pour cela, il utilise la moyenne du ratio de la cohésion sur le couplage pour chaque classe [20]. Toutefois notons qu'avec cette technique, si nous supprimons toutes les classes sauf les *BLOB* (ou si nous fusionnons toutes les classes pour n'avoir que deux *BLOB* de cohésions équivalentes), alors JDeodorant ne détectera plus aucune instance de *BLOB* puisque statistiquement les deux classes ne sont plus des données aberrantes. On voit ici l'avantage de l'utilisation de la base de données de Paprika qui détectera quand même ces deux classes puisqu'elles apparaîtront comme des données aberrantes par rapport à l'ensemble des autres classes de toutes les applications analysées.

8.1.6 *Impact de l'obfuscation sur la détection des défauts de code*

Nous avons aussi évalué l'impact de l'obfuscation sur ces résultats. Nous avons donc analysé les APKs des applications après avoir été obfusquées et optimisées avec le très populaire Proguard 5.1 [5] et Stringer Java Obfuscator 1.6.11 [6]. Proguard est un obfuscateur Java libre compatible pour Android. Il est disponible par défaut dans une version embarquée dans la plupart des environnements de développement comme Netbeans, Eclipse et surtout Android Studio. Stringer Java Obfuscator est un outil commercial qui permet entre autre de crypter les chaînes de caractères, il est aussi compatible avec la plupart des environnements de développement. Les deux outils supportent le renommage des classes, méthodes et variables ainsi que certaines optimisations de code comme la suppression du code mort. Comme on peut l'apercevoir dans la documentation de Proguard, ces outils ne pratiquent pas d'obfuscation du flux d'exécution afin de ne pas affecter négativement les performances et la taille de l'application. Toutefois la phase d'optimisation peut restructurer le code source, notamment en supprimant le code mort. Comme nous pouvons l'apercevoir dans le tableau 9, Stringer n'a aucun impact sur la détection alors que Proguard n'affecte que légèrement le rappel de notre détection.

Tableau 9 – Impact de l'obfuscation sur la détection des défauts de code

Outil	**Précision**	**Rappel**	**F1-mesure**
Obfuscation Stringer	1	1	1
Obfuscation/Optimisation Stringer	1	1	1
Optimisation Proguard	1	0.9838	0.9918
Obfuscation/Optimisation Proguard	1	0.9032	0.9491

L'impact de Proguard sur le rappel peut être expliqué par deux facteurs. Tout d'abord, le processus d'optimisation de Proguard supprime

5. Proguard : http://proguard.sourceforge.net
6. Stringer Java Obfuscator : https://jfxstore.com/stringer

une instance de *Long Method* en optimisant les ressources de style de l'application dans la classe `R` générée par Android. Notons que cet aspect n'est plus d'actualité avec la nouvelle version de PAPRIKA. Ensuite, le processus d'obfuscation nous empêche de détecter les classes internes et par conséquent toutes les instances de *Leaking Inner Class*. L'impact reste donc assez limité mais est à prendre en compte dans les résultats de PAPRIKA sur les applications obfusquées, il est à noter que de manière générale nous sommes capables de détecter les classes internes dans la plupart des applications que nous analysons.

8.1.7 *Menaces à la validité*

Dans cette sous-section, nous discutons les problèmes principaux qui peuvent menacer la validité de cette validation et qui doivent donc être pris en compte afin de mitiger les conclusions. Pour cela, nous présentons les menaces en suivant les recommandations proposées dans [17].

Validité interne : Les menaces à la validité interne sont pertinentes pour les études qui cherchent à effectuer des relations de causes à effets. Dans ce cas, cela pourrait être dû aux seuils que nous utilisons. Toutefois, cette validation est plus à caractère technique et par conséquent elle est n'est pas menacée ici puisque nous ne cherchons pas à interpréter la pertinence des résultats présentés.

Validité externe : Les menaces à la validité externe sont les approximations qui pourraient émerger de nos conclusions surtout lorsqu'elles impliquent des généralisations sur des contextes différents. Nous avons fait attention à ne pas effectuer des conclusions qui généralisent la fiabilité des résultats de PAPRIKA. Il est évident que l'application témoin n'est pas représentative de l'ensemble des applications, toutefois cette étude permet de prouver que PAPRIKA fonctionne en accord avec ces spécifications.

Fiabilité : Les menaces à la fiabilité concernent la possibilité de reproduire cette étude et les analyses qui en découlent. Nous fournissons tout le matériel nécessaire afin d'aider à la reproduction de cette étude, de plus les outils statistiques utilisés sont clairement décrits.

Validité conceptuelle : La validité conceptuelle de cette validation aurait pu être influencée par les mesures et tests statistiques effectués durant nos analyses. Nous avons tenté de minimiser ce facteur en utilisant les indicateurs classiques (précision, rappel et F1-mesure) du domaine de l'extraction d'informations [27].

Validité des conclusions : La menace principale aux conclusions de cette étude pourrait provenir des tests statistiques que nous avons utilisés, nous avons réduit ce risque en utilisant seulement des tests statistiques couramment employés dans la communauté du génie logiciel [54].

8.2 VALIDATION EMPIRIQUE PAR DES EXPERTS

Dans cette section, nous présentons les résultats d'une étude empirique destinée à valider l'efficacité de PAPRIKA en terme d'exactitude, précision, rappel et F1-mesure. Pour cette analyse, nous avons comparé les résultats obtenus par PAPRIKA avec les opinions de développeurs Android sur différentes instances présentant ou non des défauts de code. Une fois de plus, nous suivons les recommandations de Wohlin *et al.* [88] pour présenter les résultats de cette validation.

8.2.1 *Hypothèses*

Afin de pouvoir évaluer l'efficacité de notre approche, nous avons formulé les hypothèses suivantes :
H_1. Exactitude : La détection de PAPRIKA a au moins une exactitude de 75%, c'est-à-dire que plus de trois quarts des occurrences de défauts de code sont correctement classifiées.
H_2. Précision : La détection de PAPRIKA a au moins une précision de 75%, c'est-à-dire que plus de trois quarts des occurrences de défauts de code sont des vrais positifs.
H_3. Rappel : La détection de PAPRIKA a au moins un rappel de 75%, c'est-à-dire que plus de trois quarts des défauts de code existants dans le code source sont détectés.
H_4. F1 : La détection de PAPRIKA a au moins une F1-mesure de 75%. La F1-mesure regroupe la précision et le rappel et c'est donc une bonne estimation de la pertinence d'un classifieur [27]. Une haute valeur de F1 signifie que la solution analysée combine une haute précision et un haut rappel.

8.2.2 *Sujets*

Les sujets de notre validation sont les 14 développeurs Android qui ont évalué la présence des défauts de code dans des portions de code provenant d'applications Android existantes. Aucun des sujets n'a été impliqué dans le développement de PAPRIKA et ils n'étaient d'ailleurs souvent pas au courant de la nature des défauts de code Android qu'ils ont dû analyser malgré leurs expériences dans le développement de telles applications.

8.2.3 *Objets*

Les objets de cette étude sont les 17 types de défauts de code détectés par PAPRIKA, que nous avons déjà présenté dans le chapitre 4.

8.2.4 *Conception expérimentale*

Pour valider notre approche, nous avons donc demandé à 14 experts en développement Android d'inspecter un total de 76 occurrences potentielles de défauts de code qui ont été extraites de sept applications au code source libre.

L'analyse manuelle du code source complet d'une application demande beaucoup de temps et peut être source d'erreurs d'inattention, nous avons décidé de faire évaluer des potentielles instances de défauts de code qui étaient marqués comme défauts de code ou non par PAPRIKA. Tous les instances ont été sélectionnées de manière aléatoire. Nous avons différencié les instances positives qui ont été détectées par PAPRIKA, des instances que nous appelons négatives qui sont en fait des portions de code qui pourraient théoriquement contenir un défaut de code mais que PAPRIKA n'a pas identifié comme en contenant.

Afin de sélectionner les instances, nous avons tout d'abord analysé avec PAPRIKA 45 applications téléchargées depuis le dépôt d'application libre F-Droid (`https://f-droid.org`), afin d'obtenir tout type de défauts de code. Les seuils utilisés par cette validation ont été calculés avec l'ensemble de données présentées dans le chapitre 9. Toutefois, nous n'avons malheureusement pas trouvé d'occurrences d'*Invalidate Without Rect* puisque ce défaut est assez rare. Il ne fait donc pas partie de cette validation, mais les résultats sur l'application témoin laisse à penser que PAPRIKA détecte ce défaut de code de manière efficace.

Nous avons ensuite réduit ce nombre d'applications libres à sept afin d'obtenir la détection de chaque type de défauts de code dans au moins deux applications. Les applications ainsi sélectionnées apparaissent dans le tableau 10. Pour chaque défaut de code, nous avons choisi aléatoirement des instances positives et négatives dans ces applications en se basant sur les résultats donnés par PAPRIKA. Une fois sélectionné, nous avons extraits la portion de code intéressante—c'est-à-dire, la portion de code permettant l'identification par un humain du défaut de code—dans le code source de chaque application.

Tableau 10 – Liste des applications libres utilisées pour la validation

Nom	**Paquet**	# **Version**
SoundWaves	org.bottiger.podcast	38
Terminal Emulator	jackpal.androidterm	71
Amaze	com.amaze.filemanager	29
WordPress	org.wordpress.android	103
Sky Map	com.google.android.stardroid	1113
Vanilla Music	ch.blinkenlights.android.vanilla	1031
ADW.Launcher	org.adw.launcher	34

Nous avons ensuite séparé les instances sélectionnées en deux ensembles expérimentaux arbitraires. Le premier ensemble regroupe les défauts de code LIC, LM, MIM, NLMR, SAK, UCS, UHA et UIO tandis que le second ensemble est constitué de BLOB, CC, HAS, HBR, HMU, HSS, IGS et IOD. Nous avons ensuite créé un formulaire en ligne pour chaque ensemble afin de collecter les réponses des participants. Chaque question du formulaire était composée d'une portion de code et de la définition d'un défaut de code. Le participant devait alors donner son opinion sur la présence du défaut de code dans la portion de code. Ces questionnaires de forme oui/non sont toujours disponibles en ligne pour consultation [7].

Nous avons formé deux groupes d'experts, chaque groupe devant se prononcer sur un des ensembles expérimentaux (un groupe de défauts de code). La séparation en deux ensembles permet de réduire le temps nécessaire pour chaque participant et réduit aussi le nombre de définitions de défauts de code à assimiler.

8.2.5 *Processus de validation*

La validation s'est donc effectuée en utilisant les formulaires précédemment mentionnés. Les experts se sont prononcés sur la présence d'un éventuel défaut de code pour chaque question. Les participants ne possédaient aucune information sur les résultats données par l'analyse de Paprika. Ils n'étaient donc pas au courant du nombre de cas positifs, négatifs ou faisant appel à la logique floue.

Nous avons effectué la validation dans une pièce prévue à cet effet en ne permettant pas d'interactions entre les différents participants. Chaque expert a été interrogé indépendamment et nous avons supervisé individuellement les réponses apportées pour chaque portion de code. En effet, nous voulions être disponibles pour clarifier toutes les questions qui pourraient concerner les définitions des défauts de code. De plus, nous voulions observer le raisonnement suivi par les experts, notamment lorsque la logique floue était utilisée par Paprika. Les experts ont pris entre 30 et 50 minutes pour effectuer la validation.

8.2.6 *Résultats de la validation*

Afin de calculer l'exactitude, la précision, le rappel et la F1-mesure de notre approche, nous avons comparé les opinions des développeurs avec les résultats obtenus par Paprika. Nous excluons ici les cas avec logique floue qui seront abordés dans la section suivante. Dans les autres cas, pour calculer l'exactitude, la précision, le rappel et la F1-mesure pour chacun des défauts de codes, nous avons appliqué les formules suivantes (avec *vp* qui correspond au nombre de vrais positifs, *vn* au nombre de

7. Questionnaires de la validation : http://sofa.uqam.ca/paprika/journal16.php#Forms

vrais négatifs, *fp* au nombre de faux positifs, et enfin, *fn* le nombre de faux négatifs) :

$$\text{Exactitude} = \frac{vp + vn}{vp + vn + fp + fn} \tag{2}$$

$$\text{Précision} = \frac{vp}{vp + fp} \tag{3}$$

$$\text{Rappel} = \frac{vp}{vp + fn} \tag{4}$$

$$\text{F1} - \text{mesure} = 2 \times \frac{\text{Précision} \times \text{Rappel}}{\text{Précision} + \text{Rappel}} \tag{5}$$

Le tableau 11 montre les résultats pour chacune de ces variables.

Nous pouvons observer que globalement PAPRIKA a une excellente exactitude (0.9), précision (0.88), rappel (0.93), et F1-mesure (0.9). Par conséquent, PAPRIKA donne la majorité du temps des vrais positifs et des faux négatifs pour tous les défauts de code. Toutes nos hypothèses sont donc validées et nous estimons donc que PAPRIKA est une approche fiable et efficace pour les développeurs.

Toutefois, on peut observer quelques différences entre les défauts de code. Par exemple, on obtient une exactitude de 100% pour le BLOB ainsi qu'une F1-mesure de 1 alors que le F1-mesure de Member Ignoring Method est seulement de 0.76.

Nous nous attendions à une exactitude moindre pour les défauts de code que nous considérons comme problématiques ou suspicieux, car comme évoqué précédemment bien que le mécanisme de détection de PAPRIKA soit fiable, il est possible que chaque participant utilise un seuil subjectif et personnel pour déterminer la présence ou non de ces défauts de code.

À l'inverse, nous nous attendions à de meilleurs résultats pour *Leaking Inner Classes* et surtout *Member Ignoring Method* qui sont deux défauts de code qui n'utilisent pas de notion de seuils. Après inspection des résultats, il s'avère que les résultats présentés aux experts sont effectivement des vrais positifs ou des faux négatifs mais qu'ils sont parfois difficiles à détecter visuellement au sein du code d'une classe. Il est donc fort probable, qu'il y ait donc erreur humaine plus qu'une opinion différente des résultats de PAPRIKA pour les quelques réponses contradictoires.

8.2.7 *Menaces à la validité*

Dans cette sous-section, nous discutons les problèmes principaux qui peuvent menacer la validité de cette validation et qui doivent donc être

Tableau 11 – Exactitude, Précision, Rappel, et F1-mesure de PAPRIKA pour tous les défauts de code

	BLOB	CC	HAS	HBR	HMU	HSS	IGS	IOD	LIC	LM	MIM	NLMR	SAK	UCS	UHA	UIO	**TOUS**
Exactitude	1.00	0.89	0.86	0.86	1.00	0.79	1.00	1.00	0.96	0.93	0.75	0.86	0.86	0.89	0.96	0.79	0.90
Précision	1.00	0.79	0.93	0.71	1.00	0.79	1.00	1.00	0.93	0.86	0.79	0.93	0.71	0.79	0.93	0.86	0.88
Rappel	1.00	1.00	0.81	1.00	1.00	0.79	1.00	1.00	1.00	1.00	0.73	0.81	1.00	1.00	1.00	0.75	0.93
F1-mesure	1.00	0.88	0.87	0.83	1.00	0.79	1.00	1.00	0.96	0.92	0.76	0.87	0.83	0.88	0.96	0.80	0.90

pris en compte afin de mitiger les conclusions. Pour cela, nous présentons les menaces en suivant les recommandations proposées dans [17].

Validité interne : Dans notre cas, les menaces principales sont : l'expérience des participants dans le domaine, l'échange d'information entre participants, le manque d'objectivité vis-à-vis des auteurs de l'étude, ainsi que la difficulté de compréhension des informations contenues dans les formulaires en ligne. Concernant l'expérience des participants, nous avons choisi des développeurs habitués au développement sur la plateforme Android. Il est à noter que la majorité des participants ne connaissaient pas la plupart des défauts de code, nous avons donc veillé à fournir des définitions et informations détaillées dans chacun des formulaires. L'échange d'information a été réduit en interrogeant les participants de manière indépendante mais aussi en utilisant différents formulaires et en surveillant les participants pendant leurs participations à l'étude. Le manque d'objectivité a été réduit en ne mentionnant pas aux participants que les défauts de code qu'ils examinaient étaient les résultats de notre approche mais simplement en leur demandant de se prononcer sur la présence ou non du défaut. Nous avons aussi veillé à réduire notre influence sur l'étude en sélectionnant les instances de défaut de code de manière aléatoire, sans prendre en compte les différentes informations annexes que Paprika peut fournir (comme les valeurs des métriques) et qui auraient pues guider nos choix. Enfin, concernant l'incompréhension, nous avons essayé d'atténuer au maximum les doutes qu'auraient pu avoir les participants en clarifiant toutes les incompréhensions qui ont pues intervenir durant les expérimentions, notamment concernant les définitions des défauts de code.

Validité externe : La menace principale concerne ici la représentativité des résultats présentés. Pour minimiser ce risque, nous avons analysé 45 applications libres et différentes avant de réduire la sélection des cas à 7 applications qui pouvaient contenir la majorité des différents de défauts de code que Paprika est capable de détecter. Ce jeu d'applications réduit peut donc être considéré comme représentatif des différents types de défauts de code qui pourraient apparaître ou non dans une application. L'expérience des participants dans le développement Android leur a permis de comprendre facilement les portions de code contenant les défauts de code de ces 7 applications. Cette expérience a aussi été mise à contribution lorsque les développeurs ont eu à donner leurs avis, ce qui nous offre une latitude de généralisation plus importante.

Validité conceptuelle : La validité conceptuelle de cette validation aurait pu être influencée par les mesures et tests statistiques effectués durant nos analyses. Nous avons tenté de minimiser ce facteur en utilisant les indicateurs classiques (exactitude, précision, rappel et F1-mesure) du domaine de l'extraction d'informations [27].

Fiabilité : Ici aussi, nous fournissons tout le matériel nécessaire afin d'aider à la reproduction de cette étude, de plus les outils statistiques utilisés sont clairement décrits.

Validité des conclusions : La menace principale aux conclusions de cette étude pourrait provenir des tests statistiques que nous avons utilisés, nous avons réduit ce risque en utilisant seulement des tests statistiques couramment employés dans la communauté du génie logiciel [54].

8.3 VALIDATION EMPIRIQUE DE LA LOGIQUE FLOUE

La validation précédente fut aussi l'occasion pour nous de valider la pertinence de l'utilisation de la logique floue pour nos résultats. La conception et les menaces à la validité de la section précédente sont donc aussi valables pour la section présente.

En plus des cas précédemment mentionnés de forme oui/non, nous avons aussi inclus des cas flous provenant de deux applications pour chaque défaut de code subjectif. Toutefois, nous n'avons trouvé aucun cas utilisant la logique floue pour *Heavy Service Start*, il a donc été exclu des résultats qui suivent. Ces résultats ont aussi été l'occasion pour nous d'effectuer une analyse de sensibilité afin d'observer les bénéfices de l'usage de la logique floue par rapport aux seuils traditionnels.

Pour analyser les résultats, nous comparons simplement le ratio de oui/non de réponses des participants à la valeur floue donnée par PAPRIKA. On observe alors une différence de seulement 0.173 en moyenne entre ces deux résultats. La boite de Tukey de la figure 13, qui a été tracée avec les différences de chaque type de défauts de code subjectifs, nous confirme que la différence se situe la plupart du temps entre 0.08 et 0.26, avec toutefois un maximum à 0.32. Par conséquent, nous estimons que les résultats fournis par PAPRIKA représentent bien les différentes opinions des experts. De plus, il s'avère que nous avons pu observer que les cas de logique floue étaient ceux sur lesquels nos experts ont eu le plus de mal à se prononcer sachant qu'ils ne pouvaient répondre que par oui ou non. Cette observation nous donne donc encore plus confiance dans la pertinence de l'utilisation de la logique floue.

Nous avons aussi comparé les résultats entre les défauts de code subjectifs et objectifs dans le tableau 12. Nous nous attendions à une différence en faveur des défauts de code subjectifs mais finalement la différence observée pour cette validation est négligeable. En effet, les très bons résultats que nous pouvons observer pour les défauts de code utilisant des seuils sont liés à l'utilisation de la logique floue, comme on peut le voir dans l'analyse de sensibilité qui suit.

8.3.1 *Analyse de sensibilité pour les défauts de code subjectifs*

Nous avons aussi effectué une analyse de sensibilité qui confirme que l'utilisation de la logique floue pour les défauts de code subjectifs est pertinente. Pour cela, nous avons comparé l'exactitude, la précision, le rappel et la F1-mesure des résultats de notre approche avec la logique floue avec une approche utilisant un seuil unique à Q3 + 1.5

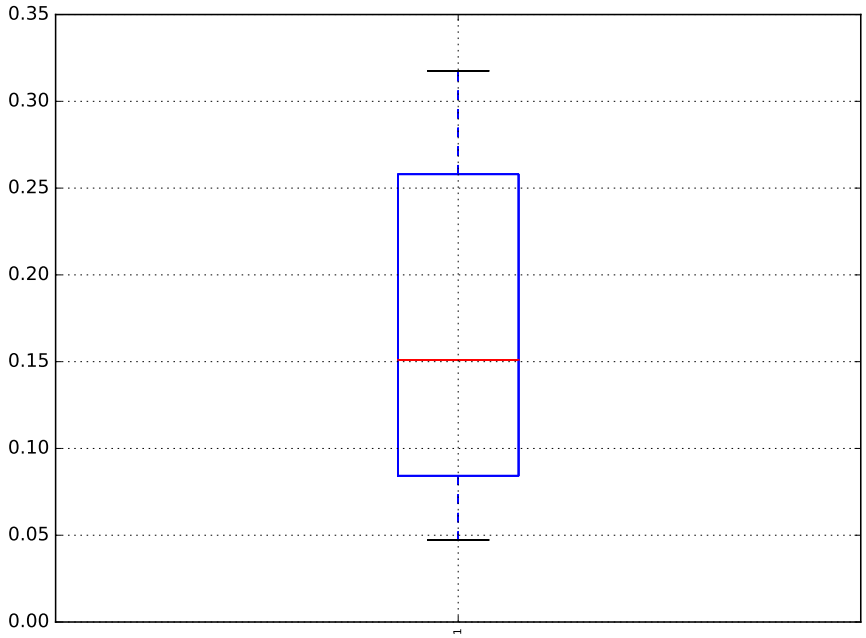


FIGURE 13 – Boîte de Tukey représentant la différence absolue entre les valeurs floues de PAPRIKA et le ratio oui/non

Tableau 12 – Exactitude, Précision, Rappel et F1-mesure de PAPRIKA pour les défauts de code subjectifs et objectifs

	Subjectif	Objectif
Exactitude	0.89	0.90
Précision	0.83	0.90
rappel	0.96	0.90
F1-mesure	0.89	0.90

IQR (c'est-à-dire, l'étendue interquartile) appelée HIGH et avec une approche VERY_HIGH qui utilise cette fois Q3 + 3 IQR. Les résultats de cette comparaison apparaissent dans le tableau 13.

De manière prévisible, le seuil HIGH présente le meilleur rappel mais la pire des précisions puisqu'il inclut plus de cas que la logique floue ou l'approche VERY_HIGH. De manière contraire, l'approche avec le seuil VERY_HIGH augmente la précision mais réduit le rappel en excluant les cas entre Q3 + 1.5 IQR et Q3 + 3 IQR. Mais surtout, on observe que c'est l'approche utilisant la logique floue qui présente les meilleurs résultats en terme d'exactitude et de F1-mesure. Cela prouve que les instances traitées de manière floue par Paprika sont en effet les cas les plus discutés par nos experts, et que donc l'utilisation de la logique floue est justifiée. Notre approche est donc le meilleur compromis, et cela permet au développeur de donner la priorité à la correction des défauts de code en fonction de valeur de logique floue que Paprika lui donne.

Tableau 13 – Analyse de sensibilité pour les défauts de code subjectifs

.	FLOUE	HIGH	VERY_HIGH
Exactitude	**0.90**	0.81	0.86
Précision	0.83	0.71	**1.00**
Rappel	0.97	**1.00**	0.70
F1-mesure	**0.90**	0.83	0.82

9

DÉTECTION DES DÉFAUTS DE CODE SUR UNE FOULE D'APPLICATIONS MOBILES

Ce chapitre décrit les résultats d'une étude empirique que nous avons effectuée sur plus de 3000 applications afin d'y étudier la présence de défauts de code Android et OO.

Cette étude a pour but d'analyser la présence de défauts de code dans des applications populaires de la plate-forme Android en prenant en compte les tendances de répartition des défauts parmi les applications, mais aussi l'impact de cette présence sur la popularité d'une application ainsi que les relations entre les défauts de code et les classes qui héritent du cadre d'application Android. Cette étude a été réalisée sur les 17 défauts de code présentés dans le chapitre 4.

Plus précisément, nous avons répondu aux cinq questions de recherches suivantes :

RQ$_1$: *Pouvons-nous observer une tendance dans la distribution des défauts de code au sein des applications Android ?*

RQ$_2$: *Pouvons nous utiliser la proportion de défauts de code d'une application pour évaluer la qualité d'une application Android ?*

RQ$_3$: *Pouvons-nous observer une relation entre la présence des défauts de code et la popularité d'une application ?*

RQ$_4$: *Est-ce que les classes qui héritent du cadre d'application Android (AFIC), comme les vues ou les activités, ont tendance à contenir plus de défauts de code que les autres classes ?*

RQ$_5$: *Existe-t-il des relations entre les différentes occurrences de défauts de code ?*

9.1 ENSEMBLE DE DONNÉES

Nous avons conçu cette étude autour d'un grand ensemble d'applications Android disponibles sur le Google Play Store. Cet ensemble est constitué de 3553 applications qui sont différentes tant en terme d'attributs internes comme leur taille qu'en terme d'attributs externes du point de vue de l'utilisateur comme la note de l'application ou le nombre de téléchargements sur le magasin en ligne. Ces applications proviennent de l'ensemble de données AndroZoo [2] qui archive les APK disponibles sur le Google Play Store. Ces applications ont été téléchargées entre Juin 2013 et Juin 2014. Toutes ces applications proviennent des tops 150 des 26 catégories disponibles. Ce top est directement géré par Google qui catégorise toutes les applications en utilisant plusieurs critères (par exemple, le nombre de téléchargements, la qualité de l'appli-

cation, la fréquence des interactions entre l'utilisateur et l'application, le nombre de désinstallations ou encore les notes et commentaires donnés par les utilisateurs). La liste des applications utilisées pour cette étude est disponible en ligne [1], elle fournit aussi des informations sur les attributs internes et externes de ces applications. Précisons que le champ taille (*size* en anglais) est ici la taille de l'application en kilo-octets, et que cela inclut l'ensemble des ressources de l'application comme les images, fichiers de données ou librairies tierces. Il est à noter que toutes les applications des tops 150 du Google Play Store ne sont pas disponibles sur AndroZoo ce qui explique le nombre de 3553 applications analysées.

Nous avons classé les applications analysées en utilisant les critères suivants :

9.1.1 *Catégorie*

Nous classons les applications selon leur catégorie d'appartenance sur le magasin en ligne (par exemple, jeux vidéo, finance, loisirs, réseaux sociaux). Chacune des 26 catégories de l'ensemble de données est représentée par au moins 80 applications. La figure 14 représente la distribution des applications pour chaque catégorie.

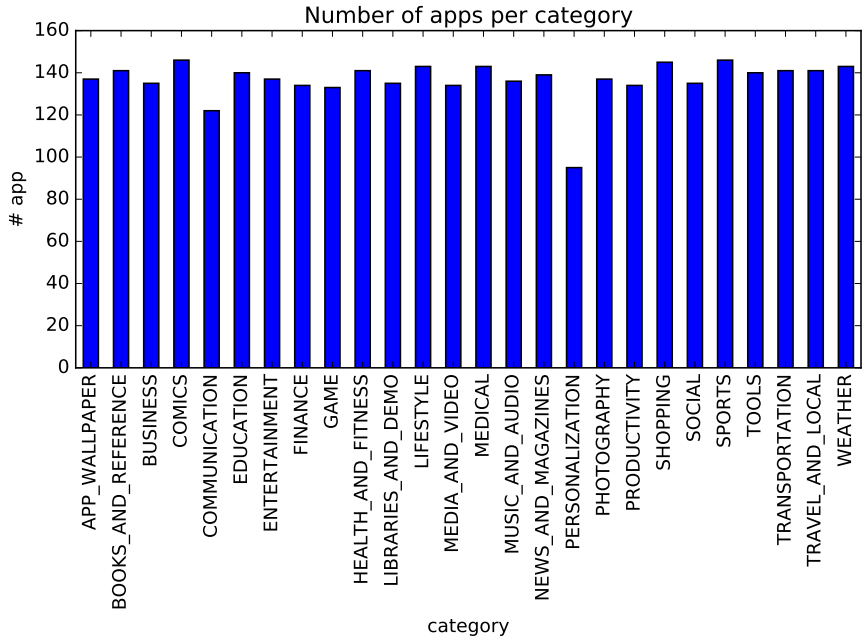


FIGURE 14 – Distribution des applications selon leur catégorie

1. Information sur l'ensemble de données : http://sofa.uqam.ca/paprika/journal16.php#Apps

9.1.2 *Notes*

Nous caractérisons aussi les applications de cette étude en fonction de la note globale calculée à partir des notes individuelles des utilisateurs. La figure 15 représente donc la distribution des applications en fonction de la note des utilisateurs. On peut observer que la majorité des applications sont notées au dessus de 3,5 sur 5 (environ 90% de l'ensemble). C'est ici la conséquence de notre choix d'analyser les applications les plus populaires de chaque catégorie, qui sont par conséquent des applications généralement bien notées, puisque le classement prend en compte cette note. On peut toutefois observer sur la figure 15 que les notes de nos applications ont tendance à suivre une distribution normale, comme on pourrait l'attendre d'un tel ensemble de données. La note donnée aux applications est basée sur échelle de Likert et, par conséquent, la note globale est calculée comme la moyenne des valeurs ordinales individuelles (entre 0 et 5). Bien que cela ne soit pas souhaitable d'un point de vue théorique sur les mesures [19], cette note donne toutefois un aperçu du sentiment des utilisateurs finaux par rapport aux applications analysées.

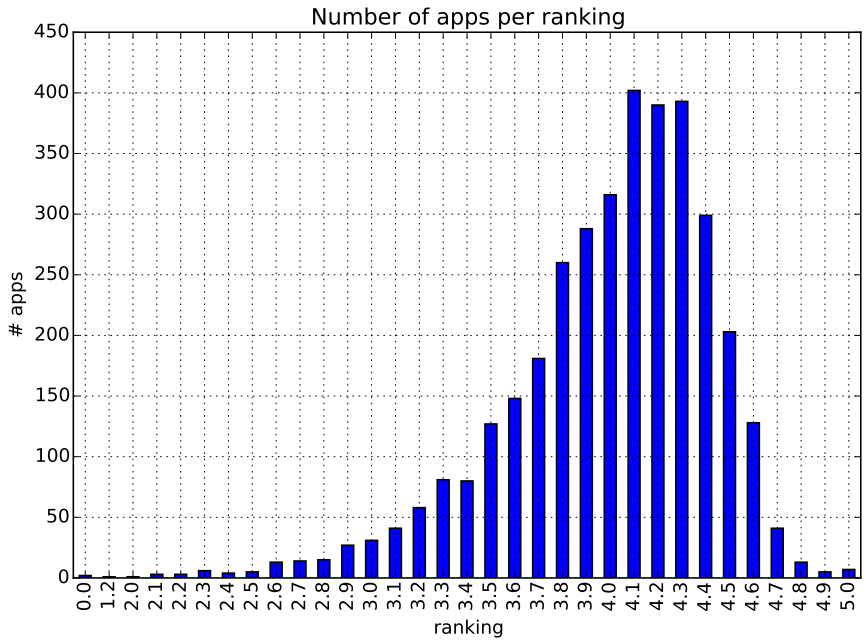


FIGURE 15 – Distribution des applications selon les notes moyennes

9.1.3 *Taille*

Afin d'analyser la relation qui pourrait exister entre la complexité d'une application et la présence de défauts de code, nous caractérisons aussi nos applications en prenant en compte la taille de l'application en

terme de nombre de classes. L'utilisation du nombre de classes est plus pertinente que la taille en octets, car comme évoqué précédemment, la taille en octets prend en compte les ressources de l'application. Notre ensemble de données est très variable en terme de nombre de classes. On retrouve donc des applications qui vont de 1 à plus de 9000 classes. Nous disposons aussi d'autres informations sur la taille, notamment le nombre de AFIC pour chaque application. Ces informations sont disponibles dans la liste des applications.

9.1.4 *Nombre de téléchargements*

Le nombre de téléchargements est également un bon indicateur de la popularité d'une application auprès des utilisateurs finaux, c'est pourquoi nous l'utilisons aussi pour catégoriser nos données. Ce nombre de téléchargements est une des métriques utilisées sur le Google Play Store pour gérer les tops de chaque catégorie. Tout comme la note globale, ce nombre est mis en avant pour caractériser le succès d'une application. Il est à noter que ce nombre n'est disponible qu'en terme d'intervalles (plus de 10/100/1000/... téléchargements).

Une première observation intéressante est la corrélation existante entre le nombre de classes d'une application et le nombre de téléchargements. On peut l'apercevoir dans la figure 16 qui représente le nombre moyen de classes pour chaque intervalle en terme de nombres de téléchargements. On peut donc observer que plus une application est grande, plus elle est populaire. Les applications populaires proposent sans doute des fonctionnalités plus nombreuses et complexes que les petites applications. Cette observation nous conforte dans notre opinion concernant la complexité grandissante des applications mobiles.

9.2 RÉSULTATS

Nous avons donc appliqué le processus d'analyse de PAPRIKA sur les 3553 applications de notre ensemble de données. Ensuite, nous avons généré plus de 120 graphiques afin d'analyser les résultats et de répondre à nos questions de recherche. Sans surprise, la proportion de défauts de code peut grandement varier d'une application à une autre et d'un défaut de code à un autre. Par exemple, l'application GOOGLE EARTH a, en moyenne, 0.04 *Member Ignoring Method* par méthode, alors que TWITTER à une moyenne de 0.14 par méthode. Nous avons toutefois pu déterminer des tendances globales (qui représentent le cas moyen) sur l'ensemble de nos applications pour chaque défaut de code. Notons que la logique floue est prise en compte dans les résultats suivants. Nous avons simplement considéré la valeur floue donnée par PAPRIKA(entre 0 et 1) dans tous les calculs, alors que les instances non floues sont considérées comme des 1.

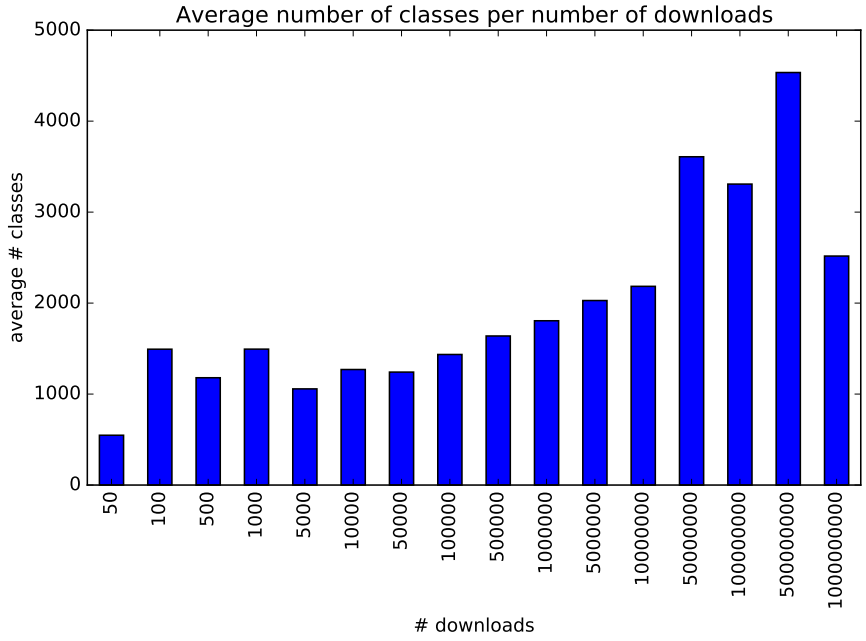


FIGURE 16 – Nombre moyen de classes en fonction du nombre de téléchargements

9.2.1 *Pouvons-nous observer une tendance dans la distribution des défauts de code au sein des applications Android ? (RQ1)*

Le tableau 14 montre le total et la proportion de chaque défaut de code détecté pour l'ensemble des applications. Pour calculer la proportion, nous considérons la somme des valeurs (floues ou non) de chaque occurrence de type de défaut de code que nous divisons par le nombre d'entités concernées par le défaut de code. Les entités concernées sont celles déjà présentées dans le tableau 5 du chapitre 5. On a donc, par exemple, la classe qui est utilisée comme unité d'analyse pour *BLOB*, *Complex Class*, *Unsuited LRU Cache Size*, *HashMap Usage* et *UnsupportedHardwareAcceleration*. Nous divisons donc le nombre total de chacun de ces défauts de code par le nombre de classes de l'application. Pour d'autres comme *No Low Memory Resolver*, c'est un nombre plus spécifique au défaut qui est utilisé, ici le nombre d'activités.

Nous pouvons observer que *No Low Memory Resolver* est présent dans la plupart des activités. En effet, la méthode *onLowMemory()* n'est pas implémentée dans 68% des cas. Bien que l'absence de cette méthode puisse être justifiée dans les cas où absolument aucune ressource ne peut être libérée par l'application, il est peu probable qu'une si grande proportion soit justifiée. Nous émettons donc l'hypothèse que les développeurs ne sont pas au courant des bénéfices de l'implémentation de la méthode *onLowMemory()* ou alors ils estiment que les gains ne seront pas assez importants. De plus, nous avons pu faire la même observation lors de notre validation sur des applications libres. Il y avait aussi pré-

Tableau 14 – Proportion de défauts de code dans toutes les applications

Entité	Class					Method			AsyncTask
Défaut	BLOB	CC	UCS	HMU	UHA	LM	MIM	IGS	HAS
# entités	5767115					36709213			40170
# défauts	190360,492	476783,904	326	125263	2931	2557416,541	4186916	477697	2934.222
Ratio	3.49%	8.24%	0.00%	2.46%	0.09%	7.87%	10.01%	1.04%	6.96%
Entité	View			Interface	Activity	Inner Class	Service	BroadcastReceiver	
Défaut	UIO	IOD	IWR	SAK	NLMR	LIC	HSS	HBR	
# entités	120366			690326	76972	2189148	11822	35597	
# défauts	7073	1645	271	58175,292	37875	555222	453,836	5990,272	
Ratio	6.85%	2.82%	0.26%	6.52%	68.45%	32.47%	3.55%	13.72%	

sence une régulière de *No Low Memory Resolver* bien que des ressources auraient pu être libérées au niveau de l'activité.

Leaking Inner Class est aussi un défaut de code très courant avec une présence dans 32% des classes internes concernées qui pourraient facilement être statiques ou utiliser une référence faible pour corriger le défaut.

Member Ignoring Method concerne environ 10% de toutes les méthodes des applications. Dans les 90% de méthodes restantes, 21.06% sont déjà des méthodes statiques et les autres ne peuvent pas devenir des méthodes statiques car elles font appel à une autre méthode non statique ou utilise une variable d'instance de la classe. Cela est à mettre en parallèle avec les méthodes statiques qui représentent 17.4% de l'ensemble des méthodes. On pourrait donc obtenir 27.4% de méthodes statiques en corrigeant les défauts de code. Les développeurs oublient donc régulièrement de mettre leurs méthodes statiques alors qu'ils pourraient améliorer les performances en le faisant dès que possible.

Concernant les défauts de code OO, on retrouve *Swiss Army Knife*, *Long Method* et *Complex Class* dans des proportions similaires, entre 6% et 9%, alors que *BLOB* apparaît pour environ 3.5% des classes. Ces proportions restent non négligeables, mais sans surprise pour nous puisque que nous avons déjà eu des proportions similaires lors des premières analyses de PAPRIKA [35, 36].

Les défauts de code pouvant entraîner un blocage de processus sont aussi assez courants. On retrouve 13.72% pour *Heavy BroadcastReceiver*, 6.96% pour *Heavy AsyncTask* et 3.55% *Heavy Service Start*. Bien que ces défauts soient détectés avec les mêmes métriques, on observe des différences entre ces trois types. Cela peut s'expliquer par les différents buts et utilisations des classes concernées. En effet, les BroadcastReceivers doivent souvent traiter les différents et potentiellement nombreux Context et Intent dans la méthode *onReceive* ce qui pourrait expliquer pourquoi cette méthode est souvent longue et complexe. D'un autre côté, *Heavy AsyncTask* et *Heavy Service Start* ne peuvent apparaître que dans des méthodes de rappels (*callback method* en anglais) qui sont supposées rester courtes. *Heavy AsyncTask* peut apparaître dans trois méthodes différentes alors que *Heavy Service Start* ne concerne qu'une méthode, ce qui peut expliquer la différence de pourcentage observée.

Les trois défauts de code qui concernent l'affichage de l'IHM sont assez peu communs. *UI Overdraw* apparaît tout de même dans 6.85% des vues, mais *Invalidate Without Rect* et *UnsupportedHardwareAcceleration* apparaissent seulement dans moins de 0.5% des vues. En effet, l'usage de la méthode *invalidate* et des méthodes n'utilisant pas l'accélération graphique est peu fréquent dans l'ensemble de nos applications.

Init OnDraw, qui est aussi lié aux vues, apparaît lui dans seulement 2.82%. Les développeurs semblent donc suivre les recommandations de Google dans ce cas.

Nous pouvons faire la même observation pour *Internal Getter/Setter* qui est présent dans seulement 1.04% de toutes les méthodes.

HashMap Usage est présent pour 2.46% des classes car les HashMaps ne sont pas toujours utilisées. Toutefois, l'utilisation de l'alternative recommandée qu'est l'ArrayMap est totalement négligeable puisque nous avons trouvé seulement 12 appels à son constructeur dans toutes nos applications. En proportion, HashMap reste très populaire.

Enfin, le défaut de code le moins courant est *Unsuited LRU Cache Size* avec quelques dizaines de cas car les caches sont assez peu utilisés.

Au niveau de l'ensemble des défauts de code et du nombre de classes des applications, on peut obtenir une régression linéaire significative (valeur-r : 0.96, valeur-p : 0) comme on peut l'observer sur la figure 17. Cette régression que nous avons obtenue par la méthode des moindres carrés montre que les développeurs ont tendance à introduire des défauts de code dans leurs applications proportionnellement aux nombres de classes qui la composent. Notons que bien cette observation n'est valable que lorsque l'on utilise les 17 défauts de code, ce n'est pas toujours le cas lorsque que l'on se concentre sur un seul type de défauts. Par exemple, il n'y a pas de relation linéaire entre le nombre de classes et le nombre de *Heavy AsyncTask*. Même en prenant, le nombre d'entités spécifiques il n'est pas toujours possible d'observer une relation. Ainsi, il ne semble pas y avoir de relations entre le nombre d'AsyncTasks et de *Heavy AsyncTask*. En fait, *Leaking Inner Class*, *Member Ignoring Method*, *Swiss Army Knife*, *Complex Class* et *Long Method* semblent être les seuls à être concernés pas une telle régression linéaire.

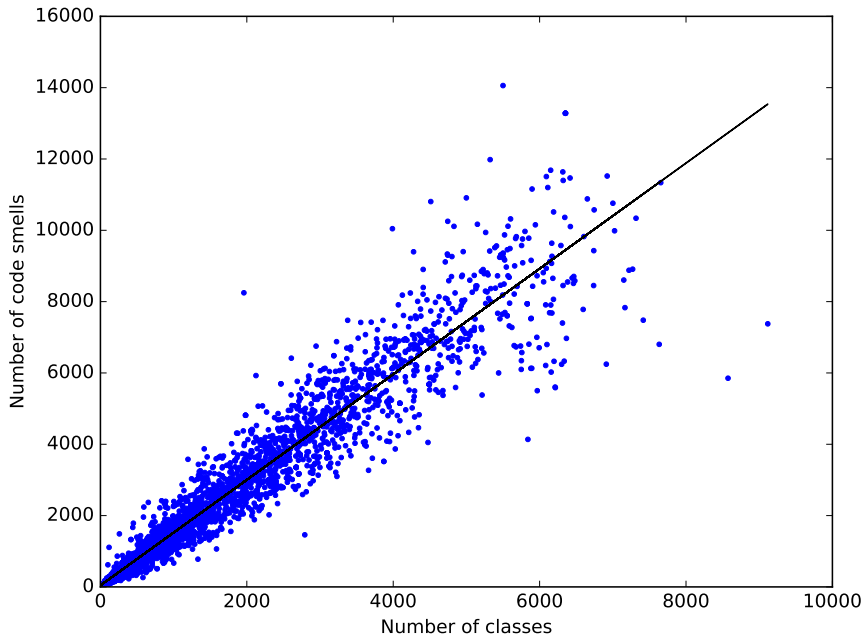


FIGURE 17 – Régression linéaire entre le nombre total de défauts de code et le nombre de classes. Chaque point représente est une application.

9.2.2 *Pouvons-nous utiliser la proportion de défauts de code d'une application pour évaluer la qualité d'une application Android ? (RQ2)*

La régression linéaire précédente nous permet donc de prédire le nombre de défauts de code d'une application et d'évaluer la qualité d'une application comme nous l'avons présenté dans la section 5.5. Comme nous l'avons décrit, il nous est alors possible d'utiliser la valeur résiduelle pour établir un score de qualité pour chaque application.

La distribution du score de qualité que nous obtenons est présentée dans la figure 18. Comme prévu, une très large proportion d'applications possède un score proche de zéro puisque notre régression linéaire est très significative. On observe toutefois que nous avons plus d'applications avec un score positif (2,126 exactement) qu'avec un score négatif (1,427). Cela signifie que la valeur absolue du score à tendance à être plus grande pour les applications avec un mauvais score que pour les applications avec un bon score. On a donc des applications qui ont tendance à être de très mauvaises qualités mais un peu moins nombreuses que les applications de bonnes qualités.

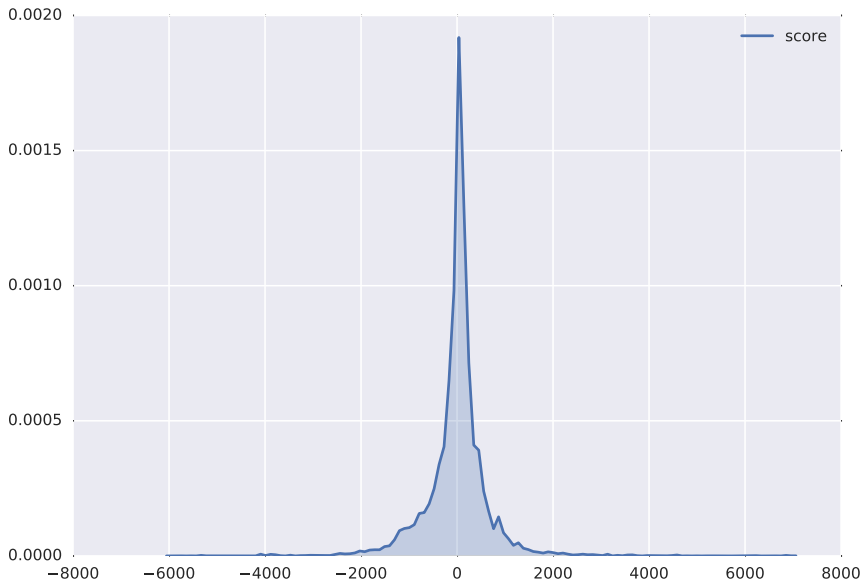


FIGURE 18 – Distribution du score de qualité dans notre ensemble de données

En regardant dans le détail les applications, on observe que les deux meilleurs applications de notre jeu de données ont été développées par Facebook avec un score de 6882 pour Messenger et de 6160 pour l'application Facebook. L'entreprise Facebook a donc probablement des processus de qualités robustes destinés à éviter les défauts de code dans ses applications. Notons que c'est sans doute dans cette optique que Facebook propose INFER [2] qui permet de faire des analyses statiques

2. Facebook INFER : https://github.com/facebook/infer

de code pour les applications mobiles. Les applications développées par Google ont souvent aussi un excellent score avec, par exemple, Youtube et Google Books parmi les dix meilleures applications avec des scores de 3134 et 3374, respectivement. De manière similaire, plusieurs applications avec un mauvais score proviennent des mêmes développeurs. Ainsi, parmi les dix pires applications, deux d'entre elles ont été développées par mtt et quatre par eduPad. L'application avec le pire score de qualité est Bandsintown Concerts avec −5875.

Le score moyen est aussi différent d'une catégorie à une autre comme on peut l'apercevoir dans la figure 19. La catégorie communication possède en moyenne un bon score de 236 et par conséquent moins de défauts de code en proportion, peu importe la taille de l'application. Les applications de sports et éducation ont en revanche le pire des scores moyens avec −210. Ces catégories contiennent des différences majeures d'une application à l'autre, contrairement aux catégories comme librairies, démos ou comics qui tendent à avoir un score similaire d'une application à l'autre. Nous n'avons pas pu identifier les causes de ces différences puisque nous n'avons pas accès au code source des applications, mais il serait intéressant de creuser ces découvertes avec d'autres études pour comprendre et expliquer ces différences. Nous pensons tout de même qu'en l'état, le score est une information pertinente pour les développeurs qui peuvent ainsi comparer leur application aux autres, au sein de l'ensemble ou d'une catégorie.

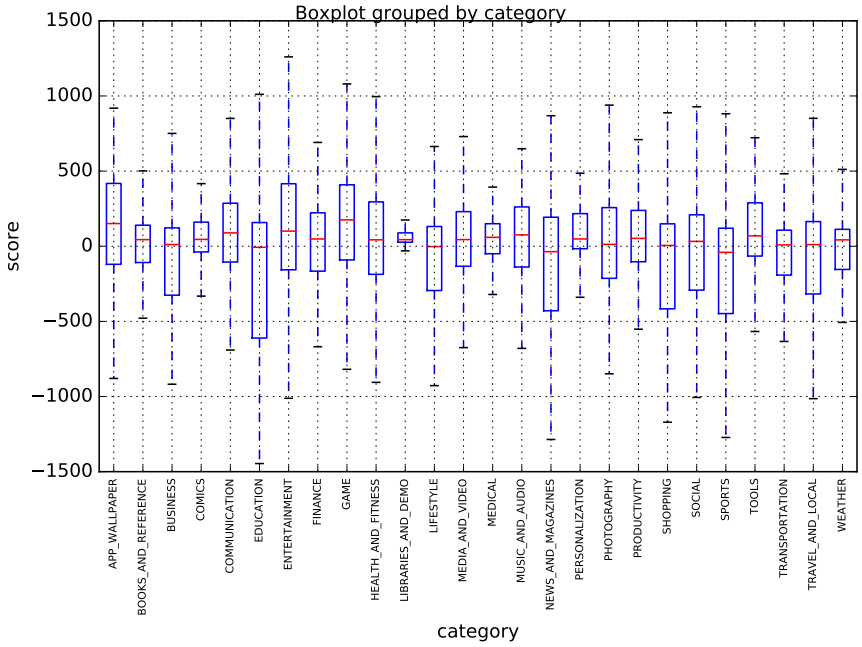


FIGURE 19 – Boîte de Tukey des scores de chaque catégorie

9.2.3 *Pouvons-nous observer une relation entre la présence des défauts de code et la popularité d'une application ? (RQ3)*

Nous avons aussi étudié la relation entre notre score et la popularité des applications, avec les indicateurs disponibles à notre disposition, c'est-à-dire la note moyenne et le nombre de téléchargements.

Concernant la note moyenne, la figure 20 montre qu'il ne semble pas exister de relation entre les scores et les notes. Nous obtenons un résultat similaire en utilisant simplement le nombre de défauts de code à la place du score. Bien que tous les défauts de code détectés ont normalement un impact sur les performances et la maintenance de l'application, il semblerait que cela n'affecte pas l'avis des utilisateurs. Nous supposons ici que les utilisateurs ont plutôt tendance à noter l'application en fonction d'autres critères comme les fonctionnalités, la facilité d'utilisation, la stabilité ou encore sur des critères esthétiques.

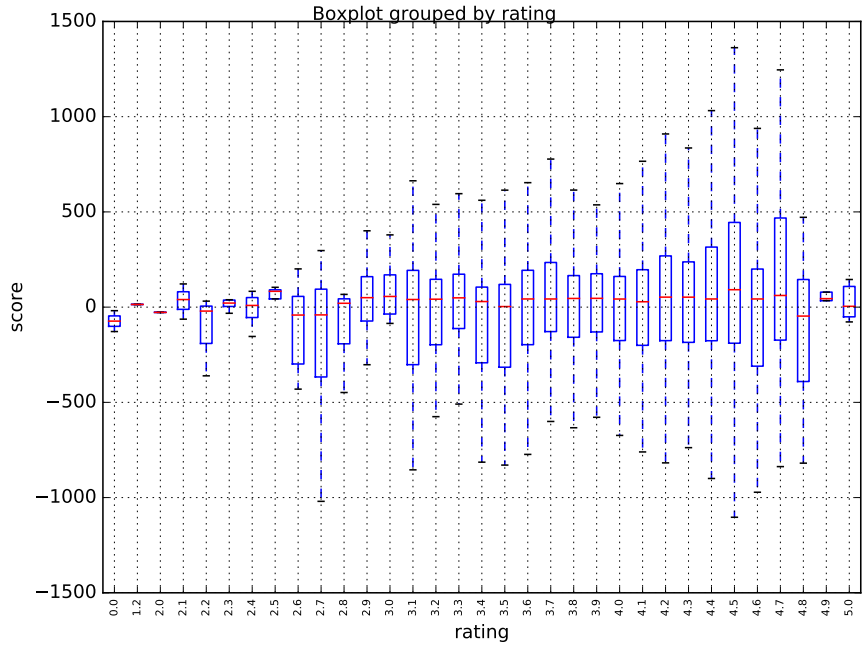


FIGURE 20 – Boîte de Tukey des scores pour chaque note

Au contraire, on observe une relation entre notre score et le nombre de téléchargements d'une application. Comme on peut l'observer dans figure 21, il y a une tendance nette à une augmentation du score avec le nombre de téléchargements (on exclut de ce raisonnement les cas 50+ and 1000000000+ nombres de téléchargements qui ne sont représentés que par une seule application dans notre ensemble d'applications). Nous ne prétendons pas qu'avoir moins de défauts de code augmente le nombre de téléchargements d'une application, mais il est plus probable que les équipes de développement travaillent plus la qualité des applications dès lors qu'elles deviennent populaires. Cela rejoint les observations que nous avons pu faire précédemment sur les applications

de Facebook et Google. Notons que cette relation est nette avec l'utilisation du score mais qu'elle ne fonctionne plus en utilisant le nombre de défauts de code brutes pour chaque application.

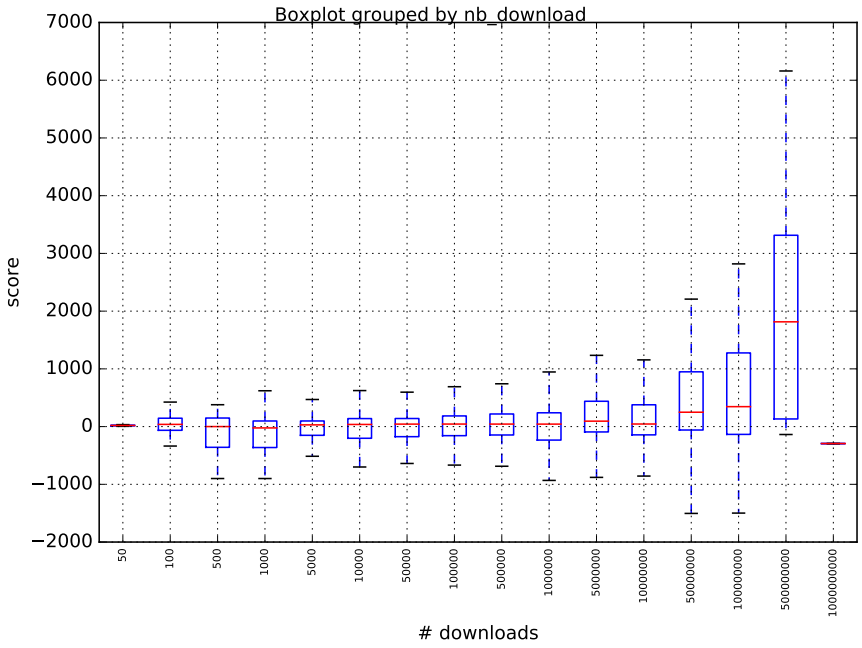


FIGURE 21 – Boîte de Tukey des scores pour chaque nombre de téléchargements

9.2.4 *Est-ce que les classes qui héritent du cadre d'application Android (AFIC) ont tendance à contenir plus certains défauts de code que les autres classes ? (RQ4)*

Afin de déterminer si les AFIC tendent à contenir plus de défauts de code, nous avons calculé le ratio de chaque type de défauts de code avec l'AFIC et nous l'avons comparé au ratio que l'on obtient sur l'ensemble des classes (AFIC et non-AFIC donc). Évidemment dans la réponse à cette question, nous n'évoquons que les défauts de code qui peuvent théoriquement affecter tous types de classes comme présentés dans le chapitre 4 puisque les autres sont spécifiques à un AFIC. Ces ratios ainsi que les chiffres bruts servant à les calculer apparaissent dans le tableau 15. Etant donné que certains défauts de code peuvent apparaître plus d'une fois par classe (lorsqu'ils concernent des méthodes), il est possible d'obtenir un taux de présence supérieur à 100%. De manière générale, il y a toujours une différence de ratio entre les classes AFIC et l'ensemble des classes. On observe aussi également d'une AFIC à une autre. En particulier, on peut remarquer que les vues ont tendance à être plus souvent affectées par *BLOB* (14.61% des vues), *Long Method* (106.18%), *Member Ignoring Method* (23.54%), *Complex Class* (26.03%) en comparaison de l'ensemble des classes. C'est aussi le cas pour les ser-

vices et les activités et *BLOB* (7.28% des services et 12.95% des activités), *Long Method* (95.20% and 110.34%), *Member Ignoring Method* (22.27% and 38.14%) et *Complex Class* (14.84% and 23.51%). On peut expliquer cette différence par le fait que les vues, services et activités sont des blocs de construction essentiels de toutes les applications Android. Ces classes ont donc tendance à contenir des méthodes complexes et nombreuses afin de gérer les fonctionnalités principales de l'application. On remarque que c'est aussi le cas de ContentProvider, sauf pour *BLOB* qui n'apparaît que dans 1.61% des cas. On peut expliquer ce faible pourcentage par le fait que les classes héritant de ContentProvider n'implémentent que rarement autre chose que les cinq méthodes à surcharger de cette classe. En revanche, les implémentations de ces méthodes sont souvent assez longues et complexes. BroadcastReceiver et AsyncTask ont tendance à contenir *Leaking Inner Class* (18.85% et 37.43%), ce qui est peu surprenant puisque ce sont des classes qui sont souvent utilisées pour démarrer des processus légers à l'aide de classes internes. On remarquera en revanche que les classes Non-AFIC sont moins affectées par *Leaking Inner Class* (9.63% de toutes les classes), tout simplement parce qu'elles ont généralement moins de classes internes. *Internal Getter Setter* est très courant dans les vues (14.06%) parce que ces classes liées à l'interface graphique contiennent beaucoup d'attributs afin de gérer cette interface. Au contraire, les autres AFIC ont tendance à posséder moins de *Internal Getter Setter* parce que les développeurs ajoutent rarement des attributs en plus de ceux obtenus lors de l'héritage. Sans surprise, *Unsupported Hardware Acceleration* apparaît presque uniquement dans les vues (1.02%) parce que ce sont ces classes qui sont dédiées au rendu de l'interface graphique. On en retrouve toutefois parfois dans les activités (0.04%) mais presque jamais dans les autres AFIC. *Unsuited LRU Cache Size* est vraiment rare, peu importe le type de classes, ce qui rejoint nos observations précédentes. Enfin, on peut constater que *HashMap Usage* est légèrement plus courant avec ContentProvider (4.26%) ce qui est logique puisque ces classes permettent de gérer l'accès à un ensemble structuré de données, qui se trouve être souvent une HashMap.

9.2.5 *Existe-t-il des relations entre les différentes occurrences de défauts de code ? (RQ5)*

Nous avons aussi exploré les relations potentielles qui pourraient exister entre les différents types de défauts de code. Pour cela, nous avons regardé quelles sont les instances qui sont affectées par plusieurs défauts de code (par exemple, une classe qui contiendrait deux types de défauts de code ou alors qui contiendrait aussi une méthode avec un défaut de code différent). Les résultats sont disponibles dans le tableau 16. Une fois de plus, il est possible de dépasser les 100% car une classe peut contenir plus d'une méthode avec un défaut de code. Comme nous pou-

Tableau 15 – Nombre et ratio de AFIC affectés par les défauts de code

	BLOB	LM	MIM	CC	LIC	IGS	UCS	UHA	HMU
# AFIC + Non-AFIC	122041	1798420	399903	365093	555222	477697	326	2931	125263
# BroadcastReceiver	76	7712	1993	456	6709	95	0	0	583
# AsynTask	45	20280	1112	980	15035	1011	6	4	790
# ContentProvider	40	2760	392	357	0	20	0	0	106
# View	17590	127803	28329	31333	4879	16926	1	1228	3074
# Service	861	11254	2633	1754	0	386	0	0	432
# Activity	9965	84929	29357	18098	6	3533	0	33	2434
Ratio AFIC + Non-AFIC	2.12%	31.18%	6.93%	6.33%	9.63%	8.28%	0.01%	0.05%	2.17%
Ratio BroadcastReceiver	0.21%	21.66%	5.60%	1.28%	18.85%	0.27%	0.00%	0.00%	1.64%
Ratio AsyncTask	0.11%	50.49%	2.77%	2.44%	37.43%	2.52%	0.01%	0.01%	1.97%
Ratio ContentProvider	1.61%	110.80%	15.74%	14.33%	0.00%	0.80%	0.00%	0.00%	4.26%
Ratio View	14.61%	106.18%	23.54%	26.03%	4.05%	14.06%	0.00%	1.02%	2.55%
Ratio Service	7.28%	95.20%	22.27%	14.84%	0.00%	3.27%	0.00%	0.00%	3.65%
Ratio Activity	12.95%	110.34%	38.14%	23.51%	0.01%	4.59%	0.00%	0.04%	3.16%

vions l'attendre, les défauts de code qui utilisent des métriques (comme la complexité ou le nombre d'instructions) ont tendance à apparaître dans les mêmes classes et méthodes. On a ainsi par exemple, 88.97% de *BLOB* qui sont aussi *Complex Class*, et inversement 36.15% de *Complex Class* qui sont *BLOB*. En effet, les métriques que nous utilisons sont liées à une notion de complexité, avec par exemple les méthodes qui ont de nombreuses instructions qui ont aussi tendance à avoir une complexité élevée. On peut aussi remarquer que certains défauts de code sont de simples spécialisations d'autres, c'est le cas pour *Heavy BroadcastReceiver*, *Heavy Service Start* et *Heavy AsyncTask* qui sont toujours des *Long Method*. Comme on a pu le voir dans le chapitre 5, ce sont effectivement des méthodes longues avec un critère de complexité et un filtre sur le nom et la classe de la méthode. Même si *Internal Getter Setter* n'utilise pas de métriques de complexité, on le retrouve souvent lié à *BLOB* et *Complex Class* car les classes complexes ont souvent de nombreux attributs ce qui augmente la probabilité d'utilisation d'accesseurs et de mutateurs. C'est aussi le cas pour *HashMap Usage* qui apparaît aussi souvent dans des classes complexes. *Heavy Broadcast Receiver* et *Heavy AsyncTask* apparaissent souvent avec *Leaking Inner Class* car comme évoqué pour la question de recherche précédente, les classes internes sont très courantes dans BroadcastReceiver et AsyncTask. *Init OnDraw*, *UI Overdraw* et *Invalidate Without Rect* apparaissent souvent dans les mêmes vues. Il y a donc des instances de vues qui sont très affectées par des défauts de code affectant les performances, ce qui semble signifier que les développeurs qui ont tendance à ignorer un des défauts de code ignorent les autres aussi.

9.3 MENACES À LA VALIDITÉ

Dans cette sous-section, nous discutons les problèmes principaux qui peuvent menacer la validité de cette étude et qui doivent donc être pris en compte afin de mitiger les conclusions. Pour cela nous présentons les menaces en suivant les recommandations proposées dans [17] comme précédemment.

Validité interne : Dans le cas présent, la validité interne pourrait être menacée par la stratégie de détection choisie par PAPRIKA. Nous avons donc essayé de n'utiliser que des métriques robustes pour évaluer la présence des défauts dans toutes les applications analysées. L'utilisation de la base de données complète pour déterminer les seuils réduit aussi l'influence que nous pourrions avoir sur les résultats. Toutefois la validité des résultats de PAPRIKA reste soumise à la fiabilité des validations précédemment effectuées.

Validité externe : Dans cette étude, nous avons analysé un grand nombre d'applications hétérogènes qui proviennent toutes du Google Play Store. Évidemment le critère de sélection des applications (c'est-à-dire ne prendre que les plus populaires) a un effet sur les résultats, mais

Tableau 16 – Relations entre les défauts de code

	BLOB	LM	MIM	SAK	NLMR	CC	LIC	UIO	HSS	HBR	HAS	IGS	UCS	IOD	UHA	HMU	IWR
BLOB		10.70%	16.05%	4.35%	13.26%	36.15%	0.03%	20.95%	36.64%	1.95%	0.54%	20.77%	6.12%	20.39%	20.03%	14.46%	26.95%
LM	86.71%		7.22%	13.15%	51.35%	87.63%	11.86%	38.08%	100.00%	100.00%	100.00%	20.91%	6.12%	64.54%	53.97%	35.85%	64.54%
MIM	28.16%	1.78%		9.55%	19.93%	21.95%	0.13%	0.00%	0.00%	0.00%	0.00%	2.18%	0.00%	0.00%	6.15%	3.39%	0.00%
SAK	0.45%	0.17%	0.56%		0.01%	0.46%	0.00%	0.03%	0.00%	0.37%	0.07%	0.63%	0.00%	0.00%	0.40	1.10%	0.00%
NLMR	7.42%	3.54%	6.35%	0.05%		4.40%	0.00%	0.07%	0.00%	0.08%	0.00%	1.19%	0.00%	0.00%	0.95	2.31%	0.71%
CC	88.97%	26.62%	30.78%	10.89%	19.36%		0.43%	34.32%	62.33%	10.78%	10.32%	35.75%	14.29%	37.59%	41.89%	33.60%	53.90%
LIC	0.25%	11.30%	0.55%	0.00%	0.02%	1.36%		4.01%	0.00%	21.82%	67.03%	4.41%	14.29%	6.21%	4.13%	4.93%	7.80%
UIO	1.77%	0.22%	0.00%	0.03%	0.01%	1.18%	0.04%		0.00%	0.00%	0.00%	0.07%	0.00%	96.28%	12.50%	0.00%	95.04%
HSS	0.29%	0.05%	0.00%	0.00%	0.00%	0.20%	0.00%	0.00%		0.00%	0.00%	0.02%	0.00%	0.00%	0.00	0.02%	0.00%
HBR	0.13%	0.45%	0.00%	0.24%	0.01%	0.29%	0.19%	0.00%	0.00%		0.00%	0.02%	0.00%	0.00%	0.00	0.10%	0.00%
HAS	0.02%	0.28%	0.00%	0.03%	0.00%	0.17%	0.35%	0.00%	0.00%	0.00%		0.00%	0.00%	0.00%	0.00%	0.05%	0.00%
IGS	15.71%	1.97%	0.84%	4.64%	1.62%	10.99%	0.43%	1.17%	3.42%	0.46%	0.00%		2.04%	3.19%	5.77	8.04%	2.84%
UCS	0.01%	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%		0.00%	0.00	0.02%	0.00%
IOD	0.32%	0.07%	0.00%	0.00%	0.00%	0.24%	0.01%	17.72%	0.00%	0.00%	0.00%	0.04%	0.00%		6.92	0.00%	46.81%
UHA	0.69%	0.16%	0.07%	0.13%	0.06%	0.59%	0.02%	6.36%	0.00%	0.00%	0.00%	0.18%	0.00%	19.15%		0.10%	17.02%
HMU	10.02%	2.10%	0.81%	7.36%	2.86%	9.47%	0.44%	0.03%	1.71%	1.29%	1.13%	5.00%	10.20%	0.18%	2.12%		0.00%
IWR	0.10%	0.02%	0.00%	0.00%	0.00%	0.09%	0.00%	4.37%	0.00%	0.00%	0.00%	0.01%	0.00%	11.70%	1.54%	0.00%	

il nous fournit aussi un moyen neutre de sélectionner les applications tout en apportant de la diversité tant sur les attributs internes que les attributs externes. Nous avons aussi considéré un très large ensemble de défauts de code Android et OO, ce qui nous laisse croire que les résultats sont significatifs.

Validité conceptuelle : Ici la validité conceptuelle pourrait être menacée par notre définition de la taille d'une application lorsque nous prenons en compte le nombre de défauts de code. Toutefois, l'utilisation du nombre de classes pour estimer la taille et la complexité d'une application est une pratique courante dans le monde du génie logiciel, c'est par exemple utilisé dans les approches qui font de l'analyse de points de fonction (IFPUG). L'utilisation de la valeur résiduelle pourrait menacer la validité conceptuelle, toutefois la fiabilité de notre régression nous fait penser que c'est un indicateur fiable pour estimer la qualité d'une application.

Fiabilité : Ici aussi, nous fournissons tout le matériel nécessaire afin de permettre la reproduction de cette étude, de plus les outils statistiques utilisés sont clairement décrits.

Validité des conclusions : La menace principale aux conclusions de cette étude pourrait provenir des tests statistiques que nous avons utilisés, nous avons réduit ce risque en utilisant seulement des tests statistiques couramment employés dans la communauté du génie logiciel [54].

9.4 CONCLUSION

Cette étude a été l'occasion de prouver l'efficacité de PAPRIKA et son fonctionnement possible à large échelle. En analysant 3553 applications populaires du Google Play Store, nous avons découvert plusieurs résultats significatifs. Tout d'abord, la fréquence d'apparition des défauts de code est différente d'un type de défaut à un autre, mais il y a une tendance à ce que le nombre de défauts de code augmente linéairement avec le nombre de classes. Cette observation, nous a permis de comparer la qualité des différentes applications de notre ensemble de données. Nous avons alors découvert qu'il y a une relation entre la qualité d'une application et le nombre de téléchargements de celle-ci. Finalement, nous avons aussi découvert que les classes qui héritent du cadre d'application Android ont tendance à être plus souvent affectées par certains types de défauts de code, mais aussi que certains défauts de code ont tendance à apparaître dans les mêmes instances de classes ou de méthodes. Nous croyons que les résultats de notre approche et les résultats que nous avons présentés ici peuvent être aussi utiles aux développeurs qu'au magasin en ligne d'applications afin d'améliorer la qualité des applications.

10

ÉVOLUTION DE DÉFAUTS DE CODE DANS LE TEMPS

Ce chapitre décrit les résultats d'une étude empirique que nous avons effectuée sur 3568 versions de 106 applications Android afin d'y étudier l'évolution de la présence des défauts de code dans le temps. Il est à noter que cette étude est postérieure à l'étude précédente. Et, à cette date, nous n'avons étudié que 7 défauts de code (*BLOB*, *Long Method*, *Complex Class*, *Member Ignoring Method*, *Leaking Inner Class*, *UI Overdraw* et *Heavy Broadcast Receiver*) sans utilisation de logique floue et avec un seuil à Q3 + 1.5 IQR.

Cette étude a pour but d'analyser l'évolution de la présence des défaut de code dans des applications populaires de la plate-forme Android au fil des versions. Nous souhaitions aussi voir apparaître des tendances d'évolution communes à plusieurs applications, ainsi que de démontrer que les résultats donnés par PAPRIKA sont utiles pour suivre l'évolution de la qualité d'une application. Pour cela, nous avons répondu aux trois questions de recherche suivantes :

RQ_1 : *Pouvons-nous trouver des tendances d'évolution des défauts de code sur plusieurs versions d'une application ?*

RQ_2 : *Existe-t-il des relations entre les tendances d'évolution des défauts de code ?*

RQ_3 : *Pouvons-nous utiliser* PAPRIKA *pour suivre la qualité d'une application à travers le temps ?*

10.1 ENSEMBLE DE DONNÉES

Cette étude aussi a été conçue autour d'un grand ensemble de versions d'applications populaires du Google Play Store. Nous profitons donc de la capacité de PAPRIKA à analyser directement le code binaire des applications. La difficulté principale lors de la construction de cet ensemble de données fut de trouver des versions de mêmes applications en quantité suffisante.

Nous considérons donc 106 applications qui sont différentes tant en terme d'attributs internes que d'attributs externes. Nous avons pu récolter un total de 3568 versions de ces applications qui proviennent une fois de plus d'AndroZoo et furent collectées entre Juin 2013 et Juin 2014. Ce sont aussi les applications des tops 150 de chaque catégorie du Google Play Store, mais cette fois nous avons filtré la liste d'applications pour ne maintenir que les applications dont nous possédons plus

de 20 versions afin que l'étude dans le temps soit pertinente. La liste des applications étudiées et leurs versions est disponible en ligne [1].

Afin d'illustrer la diversité de cet ensemble de données, nous le présentons sous différents points de vue.

10.1.1 *Catégorie*

Tout d'abord, nous avons classé les applications selon la catégorie associée sur le Google Play Store parmi les 24 existantes. La figure 22 montre le nombre d'applications pour chaque catégorie. Les applications ne changent pas de catégorie à travers le temps. Par exemple, *Twitter* sera toujours dans la catégorie *Social*. Les proportions que l'on retrouve aussi nous semble représentatives des applications les plus populaires et les plus mises à jour du magasin en ligne. En particulier, on remarque que la majorité des applications appartiennent à quatre catégories : *Social*, *Communication*, *Productivity* and *Photography*.

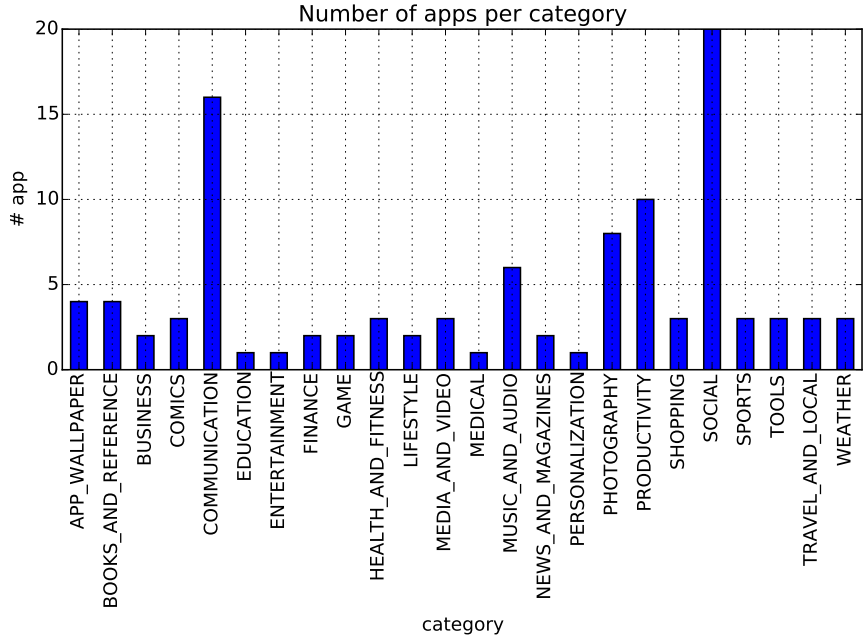


FIGURE 22 – Distribution des applications Android en fonction de la catégorie

10.1.2 *Notes*

Nous décrivons aussi notre ensemble de données en fonction de la moyenne des notes attribuées par les utilisateurs. La distribution des applications en fonction des notes apparaît sur la figure 23. Ici, la plupart de nos applications ont une note supérieure à 4.0 ce qui est sans doute dû à notre filtrage sur le nombre de versions.

1. Liste des applications et des versions : https ://goo.gl/MktCYM

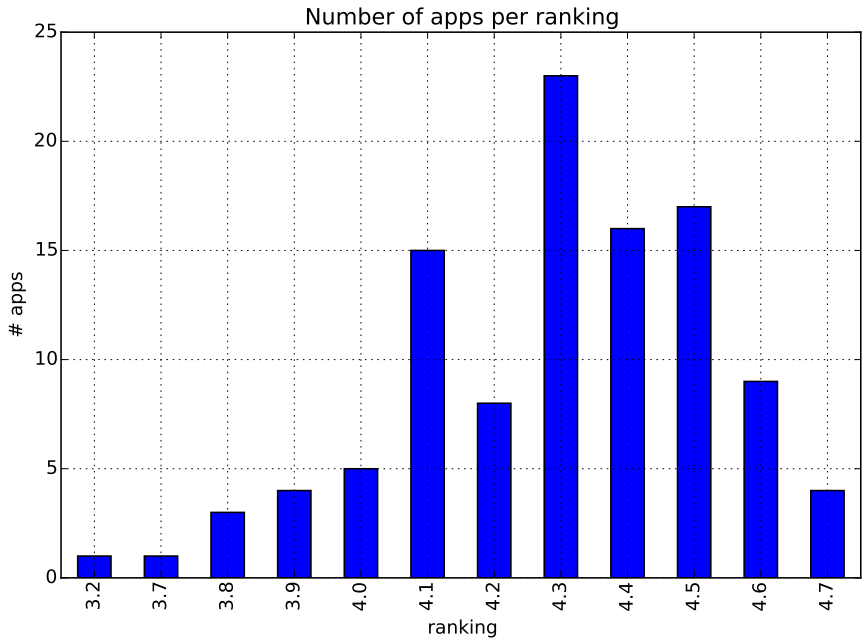


FIGURE 23 – Distribution des applications Android en fonction de la note

10.1.3 *Nombre de téléchargements*

Le nombre de téléchargements est aussi un bon indicateur de la popularité d'une application. Une observation intéressante sur notre ensemble de données est qu'il existe une corrélation entre le nombre de téléchargements et le nombre de classes de l'application. Les applications les plus populaires sont donc les plus complexes et fournissent certainement le plus de fonctionnalités.

10.1.3.1 *Versions*

Le nombre de versions d'une application peut indiquer la maturité de celle-ci et refléter le succès de l'application auprès des utilisateurs finaux. Toutefois, nous ne disposons que d'informations de temps relatives à propos des versions (l'ordre dans le temps) mais pas de mesure de temps absolue ou de l'intervalle entre deux versions. Nous ne pouvons donc pas spéculer sur la relation entre la fréquence de sortie des versions ou l'évolution du score dans le temps. La figure 25 présente le nombre de versions en fonction du nombre de téléchargements. En moyenne, chaque application possède un peu moins de 34 versions.

10.1.3.2 *Taille*

Comme nous l'avons vu précédemment, en prenant le nombre de classes de l'application comme taille, nous avons une grande variété de cas. Notre ensemble de données varie de 8 à plus de 9000 en fonction des versions. De manière générale, les applications ayant le plus de classes

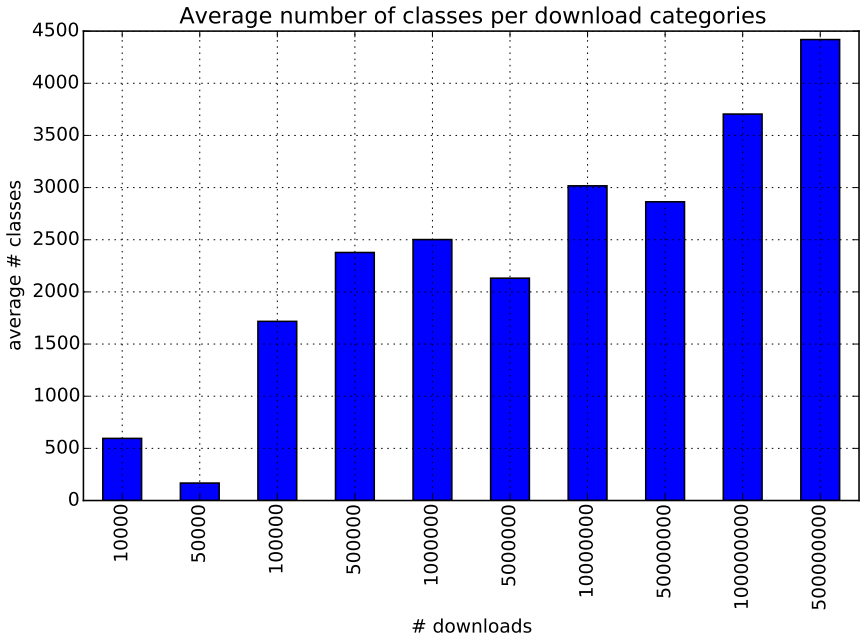


FIGURE 24 – Nombre moyen de classes d'une application en fonction du nombre de téléchargements

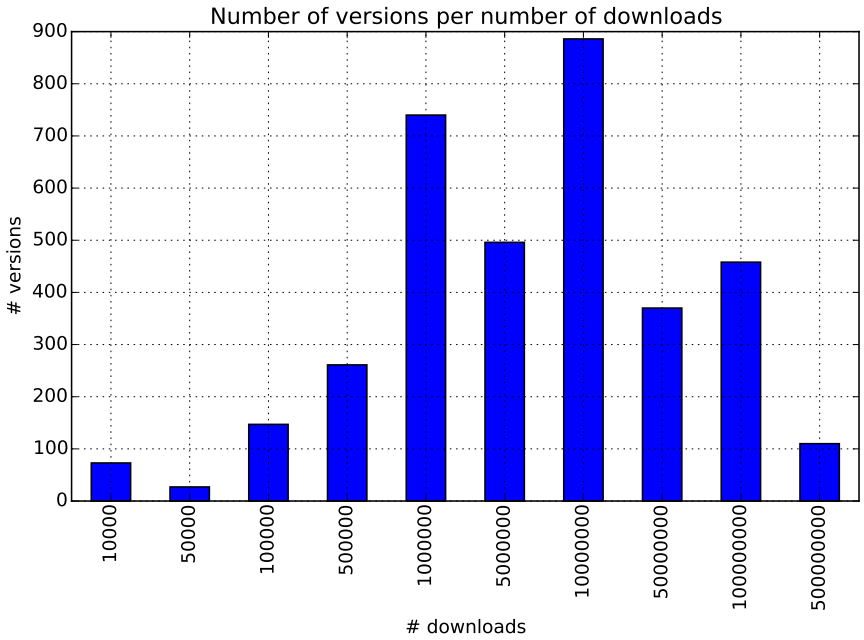


FIGURE 25 – Nombre de versions en fonction du nombre de téléchargements

ont aussi les meilleurs notes, le plus de téléchargements et plus grand nombre de versions.

10.2 RÉSULTATS

Pour obtenir les résultats, nous avons donc appliqué PAPRIKA sur les 3568 versions et généré les graphes concernant l'évolution du nombre de défauts de code pour chaque application. Au final, nous avons obtenu plus de 700 graphiques à analyser et qui sont résumés dans cette étude. Comme nous le supposions, l'évolution du nombre de défauts de code varie d'une application à l'autre et il n'est pas possible de dégager une tendance généralisable à l'ensemble des applications. Nous obtenons une conclusion similaire même en prenant en compte le ratio entre le nombre de défauts de code et les entités concernées. Par exemple, l'application CAMERA360 a 0.75 *Member Ignoring Method* en moyenne par classe tandis que l'application TO DO CALENDAR PLANNER en a en moyenne 2.5 par classe. Comme pour l'étude précédente, nous avons calculé le score de chaque version. La différence étant qu'ici nous avons suivi son évolution à travers le temps et que nous nous sommes concentrés sur un score spécifique à chaque type de défauts de code et non sur un score global.

10.2.1 *Pouvons-nous trouver des tendances d'évolution des défauts de code sur plusieurs versions d'une application ? (RQ1)*

En analysant les graphiques d'évolution du score pour chaque application et chaque type de défauts de code, nous avons découvert cinq tendances d'évolution que l'on retrouve couramment sur les applications.

A) Baisse constante : Généralement, la taille de l'application augmente au fil des mises à jour et des nouveaux défauts de code sont constamment introduits. De plus, ce type d'évolution implique qu'aucune forme de maintenance n'a été mise en place pour supprimer les défauts de code des versions précédentes. Par conséquent, le score de l'application diminue parce que les nouveaux défauts de code introduits viennent s'ajouter aux anciens. C'est la tendance que nous avons rencontré le plus souvent dans notre ensemble de données, elle peut apparaître tout au long de la vie de l'application ou seulement pour quelques versions. La figure 26 montre une telle baisse constante de la qualité pour le défaut de code *Long Method*.

B) Augmentation constante : Malgré le fait que la taille de l'application évolue avec le temps, il arrive que l'évolution semble plus contrôlée et moins arbitraire. Cette situation peut se produire lorsque l'équipe de développement suit des standards de développement rigoureux et prend en compte l'existence des défauts de code. Dans cette situation, nous obtenons une augmentation constante du score, que la taille de l'application varie ou pas à travers le temps. Par exemple,



FIGURE 26 – Évolution du score de l'application PICSART pour *Long Method*

comme on peut l'apercevoir sur la figure 27, le score de l'application FLIPBOARD ne cesse d'augmenter. Pour autant, l'application n'a changé que deux fois de taille dans les 26 versions considérées (aux versions 19 et 24). Il y a donc sans doute eu des phases de maintenance qui ont supprimé des défauts de code sans pour autant changer le nombre de classes de l'application.

C) Stabilité : Dans ce cas, le score de l'application reste constant sur plusieurs versions. La ratio de défauts de code introduits et de la taille de l'application reste stable sur plusieurs versions. Cela peut aussi être dû au fait que les versions ne comportent que des changements mineurs qui n'introduisent ni nouvelles classes ni nouveaux défauts de code. Cette tendance n'est généralement que temporaire, et ne concerne donc que quelques versions. On peut l'observer pour les 10 premières versions de FIREFOX sur *Leaking Inner Class* sur la figure 28.

D) Baisse soudaine : Dans certains cas, le score de l'application chute abruptement sur une version. C'est un point de variation majeure dans l'évolution de l'application et cela s'accompagne d'une variation importante du nombre de classes de l'application. On assiste donc souvent à une mise à jour majeure de l'application. Cette chute du score se propage sur les versions suivantes jusqu'à l'apparition d'un autre profil d'évolution. On peut observer cette forme d'évolution pour l'application EVERNOTE sur la figure 29. Ici la baisse soudaine se produit entre la version 5 et 6, ensuite le score se stabilise

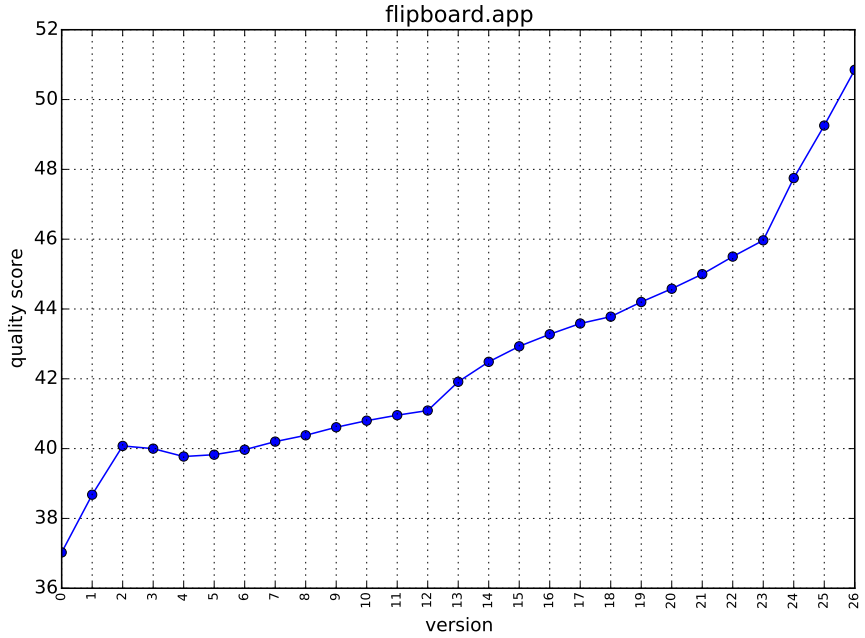


FIGURE 27 – Évolution du score de l'application FLIPBOARD pour *Complex Class*

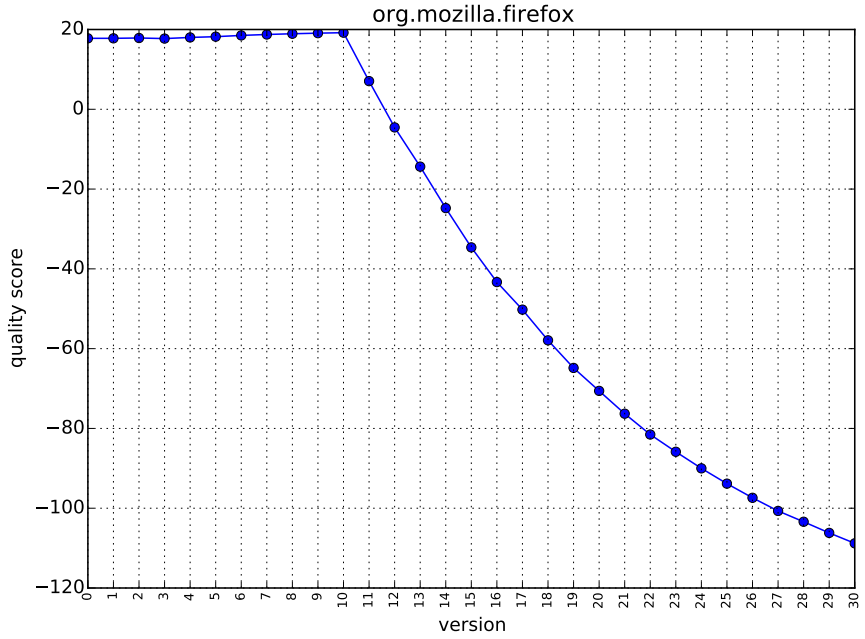


FIGURE 28 – Évolution du score de l'application FIREFOX pour *Leaking Inner Class*

mais reste faible jusqu'à la version 10 qui est suivie d'une augmentation constante.

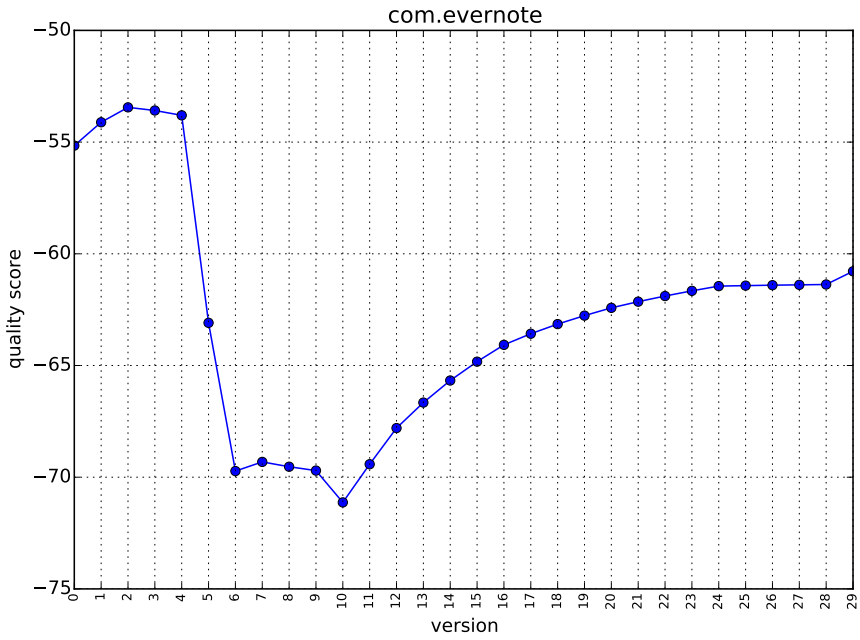


FIGURE 29 – Évolution du score de l'application de EVERNOTE pour *BLOB*

E) Augmentation soudaine : C'est l'inverse de la tendance précédente (D) et cette tendance se caractérise donc par une augmentation brutale du score sur une version. On peut l'observer sur la figure 30 avec une augmentation soudaine pour l'application SKYPE sur *Leaking Inner Class* à la version 5. On va aussi avoir cette augmentation de qualité qui va se propager sur les versions suivantes. Ici aussi on observe ce changement de score avec un changement de taille important entre les versions. Mais ce n'est pas toujours le cas, et cela peut être dû à un réusinage de l'application afin de supprimer les défauts de code, ce qui se répercute sur le score et dont l'effet persiste sur plusieurs versions.

10.2.2 *Existe-t-il des relations entre les tendances d'évolution des défauts de code ? (RQ2)*

Notre estimation de la qualité d'une application est basée sur la corrélation entre le nombre de défauts de code et l'entité concernée. Alors que certains défauts de code ont une meilleure corrélation avec le nombre de classes comme le *BLOB*, d'autres sont liés à d'autres entités comme le nombre de vues (*UIO*) ou encore le nombre de classes internes (*LIC*), par exemple.

Nous pouvons observer que lorsque la même entité est concernée par le défaut de code, alors les tendances d'évolutions des différents défauts

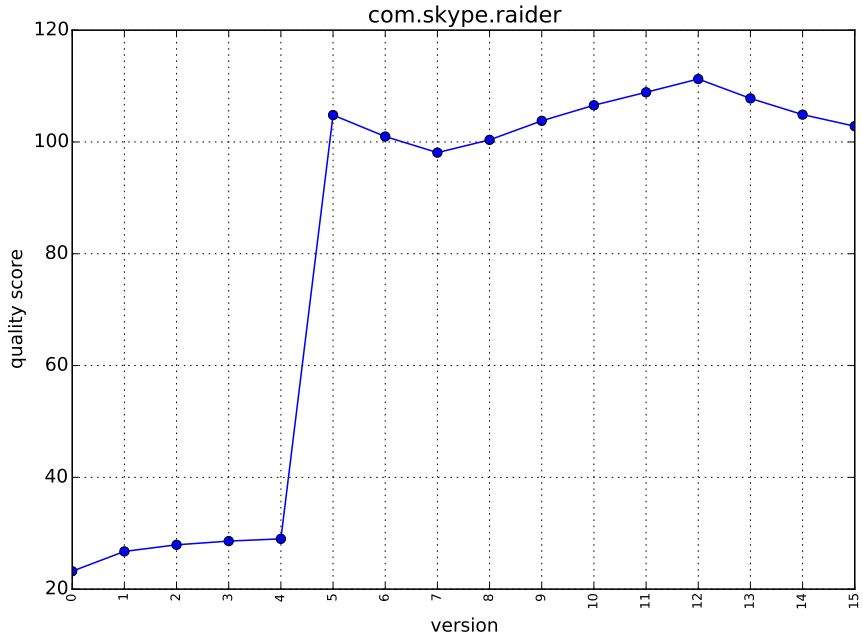


FIGURE 30 – Évolution du score de l'application SKYPE pour *Leaking Inner Class*

de code sont souvent similaires. Dans le cas présent, nous considérons que *Long Method* et *Member Ignoring Method* sont corrélées au nombre de classes, bien que dans la réalité ce soit plutôt le nombre de méthodes. Toutefois, cette approximation reste valable en vue du coefficient de corrélation que nous obtenons pour nos régressions (0.8 dans chaque cas). La figure 31 montre le genre de similarité que l'on peut observer dans l'évolution du score, ici avec *BLOB*, *Complex Class*, *Long Method* et *Member Ignoring Method* pour l'application IMO. On remarque donc que les scores baissent et augmentent de la même manière et que les changements de tendance se produisent aux mêmes versions. Trois de ces défauts de code sont liées à la notion complexité, il n'est donc pas étonnant de les retrouver corréler au travers de toute l'application. Toutefois, la présence de *Member Ignoring Method* est plus surprenante. Il est probable que les développeurs ne se soucient d'aucun de ces défauts de code en particulier, et qu'ils soient donc introduits ou supprimés dans des proportions similaires.

Ce type de relations n'est pas le seul que nous avons observé plusieurs fois. En effet, dans certains cas, le score calculé pour *BLOB* va évoluer de manière inverse du score de *Long Method* et/ou *Complex Class*. La figure 32 montre une telle évolution pour *BLOB* et *Long Method* sur l'application RETRO CAMERA. Nous avons découvert que cette situation se produit lorsqu'un *BLOB* "absorbe" une ou plusieurs *Long Method* et/ou *Complex Class*. En effet, un *BLOB* est souvent une *Complex Class* et contient plusieurs *Long Method*. Par conséquent, sa présence

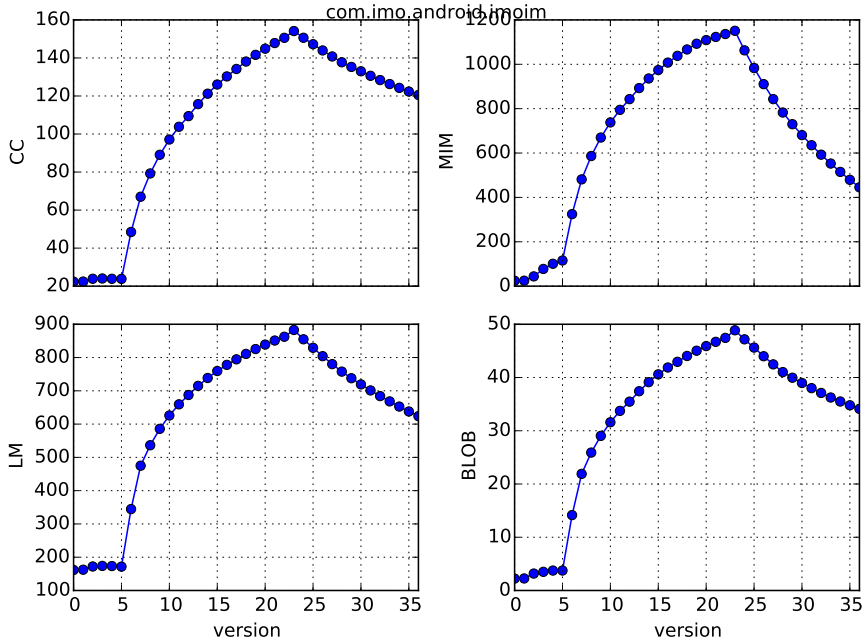


FIGURE 31 – Évolution du score de l'application IMO pour *BLOB*, *CC*, *LM* et *MIM*

tend à réduire l'apparition de ces deux défauts de code dans les autres classes et méthodes en les agrégeant en un seul endroit.

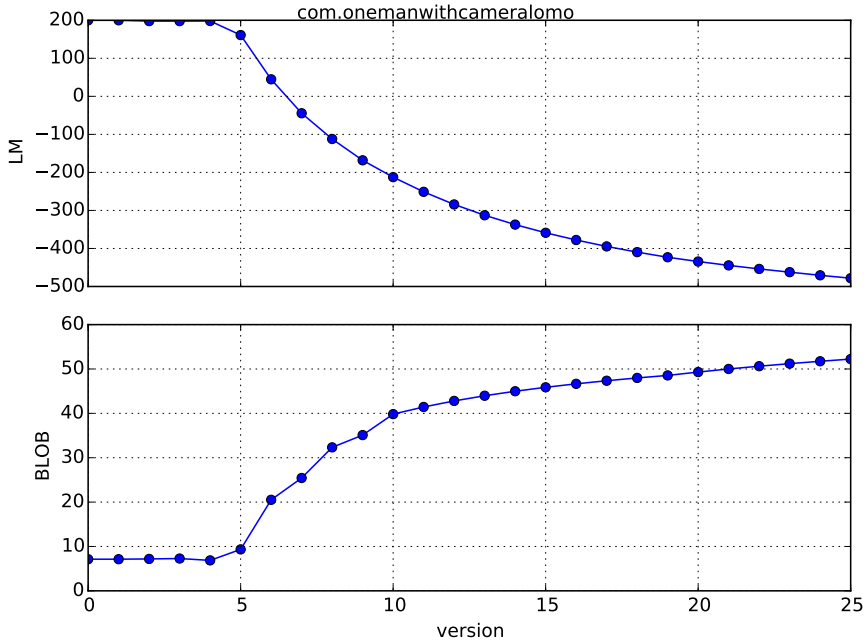


FIGURE 32 – Évolution du score de l'application RETRO CAMERA pour *BLOB* et *Long Method*

Notons que nous n'avons pas observé de relations similaires entre *Leaking Inner Class*, *UI Overdraw* et *Heavy Broadcast Receiver*. Ces défauts de code sont liés aux classes internes, vues et BroadcastReceivers et dépendent donc énormément de la manière dont l'application mobile est développée. En particulier, les nombres de vues et de BroadCastReceivers ne changent pas souvent au cours de l'évolution d'une application et les changements ne se produiront pas forcément à la même version, ce qui explique l'absence de relations observées.

10.2.3 *Pouvons-nous utiliser* PAPRIKA *pour suivre la qualité d'une application à travers le temps ? (RQ3)*

Ici, nous montrons comment il est possible d'utiliser notre approche outillée pour suivre l'évolution de la qualité d'une application, au travers d'une étude de cas sur l'application TWITTER.

À la date de cette analyse, TWITTER possédait une note moyenne de 4.1 et avait été téléchargée plus de 100 millions de fois, ce qui démontre sa grande popularité. La question que nous nous posons est la suivante : Comment la qualité de TWITTER en terme de défauts de code évolue à travers du temps ? À cette fin, nous avons analysé 75 versions de l'application avec PAPRIKA.

Pour expliquer les changements observés, nous analysons aussi les changements en terme de nombre de classes entre les différentes versions de TWITTER. La figure 33 montre l'évolution du nombre de classes pour chaque version. Depuis cette information de taille, nous pouvons observer plusieurs changements intéressants, que nous allons utiliser tout au long de cette étude de cas. Tout d'abord, il y a de nombreuses classes ajoutées entre les versions 9 et 12 (d'environ 530 à 2800 classes) Ensuite, il y a augmentation régulière de la taille entre les versions 19 et 47 ainsi que des versions 48 à 74. Toutefois, dans ces deux cas il y a une baisse ponctuelle de la taille durant cette période (version 31 et 59) qui suggère un réusinage mineur. Enfin, il y a une baisse soudaine la taille de TWITTER à la version 48 (d'environ 4700 à moins de 3000 classes). Il y a sans doute donc eu un réusinage majeur pour cette version.

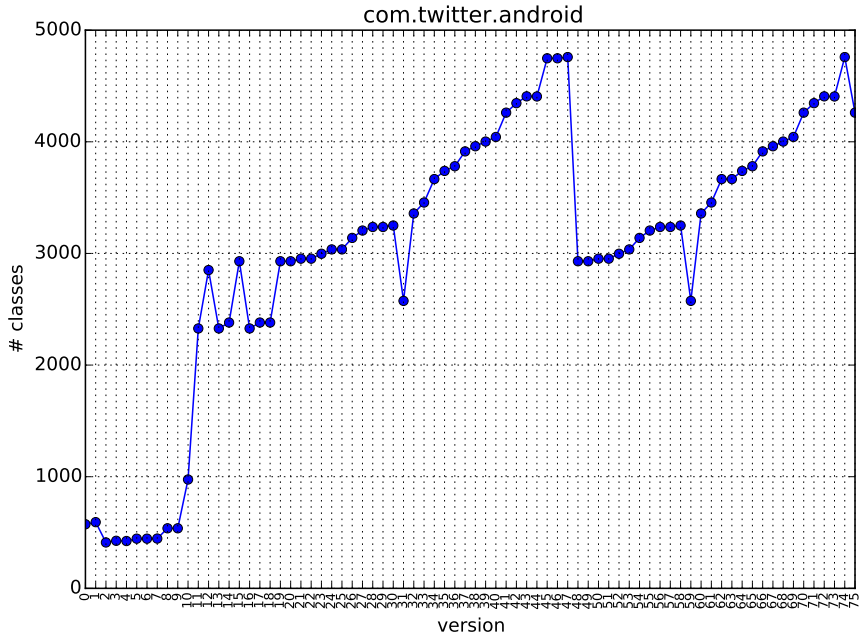


FIGURE 33 – Évolution de la taille de TWITTER

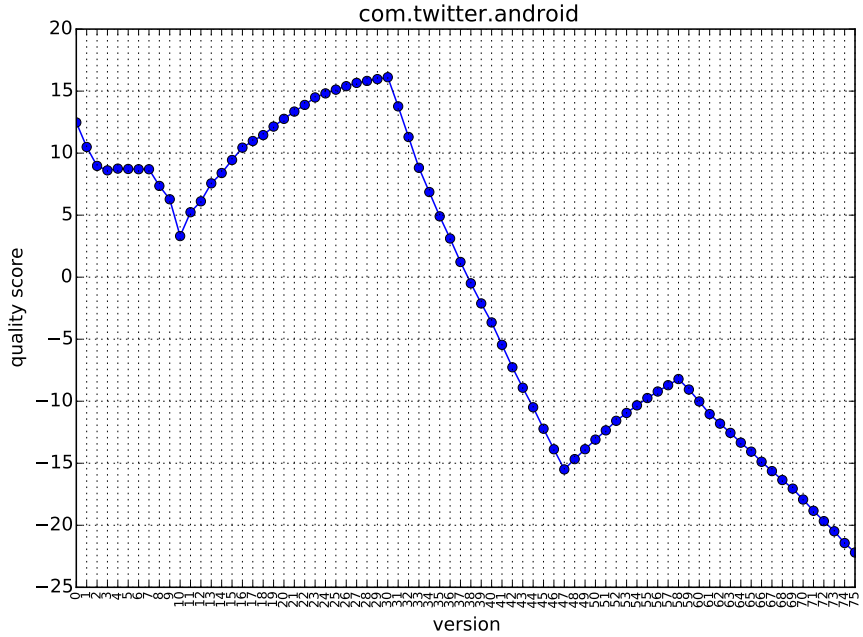


FIGURE 34 – Évolution du score de l'application TWITTER pour *Complex Class*

Avec ces informations, il devient possible d'interpréter les changements de score des défauts de code. Par exemple, la figure 34 montre l'évolution du score pour *Complex Class* pour chaque version. Nous nous concentrons ici sur les changements de situation liés aux évolutions de taille citées précédemment. Tout d'abord, l'ajout de nombreuses classes

après la version 9 correspond à une augmentation du score, ce qui signifie que les classes ajoutées contenaient très peu de défauts de code. Cette phase d'amélioration continue jusqu'à la version 31 qui correspond à une phase de baisse constante du score. Cela correspond au réusinage mineur qui a eu lieu pour cette version comme on peut l'apercevoir sur la figure 33. On retrouve le même schéma à la version 59, ce qui est cohérent avec l'évolution de la taille de l'application. Enfin, la baisse soudaine de la taille à la version 48 s'est vu accompagnée d'une augmentation du score. Cela indique que de nombreux défauts de code ont été supprimés durant ce réusinage majeur. La qualité augmente ensuite jusqu'à la version 59 que nous avons précédemment mentionnée.

Il est donc possible d'utiliser PAPRIKA pour faire de telles analyses, les développeurs peuvent ainsi améliorer le processus de développement afin de l'optimiser pour obtenir la meilleure qualité possible de l'application.

10.3 MENACES À LA VALIDITÉ

Dans cette sous-section, nous discutons les problèmes principaux qui peuvent menacer la validité de cette étude et qui doivent donc être pris en compte afin de mitiger les conclusions. Cette discussion est similaire à celle que nous avons effectuée pour l'étude précédente.

Validité interne : Cette fois aussi, la validité interne pourrait être menacée par la stratégie de détection choisie par PAPRIKA. Nous avons donc essayé de n'utiliser que des métriques robustes pour évaluer la présence des défauts dans toutes les applications analysées. L'utilisation de la base de données complète pour déterminer les seuils réduit aussi l'influence que nous pourrions avoir sur les résultats. Toutefois, la validité des résultats de PAPRIKA reste soumise à la fiabilité des validations précédemment effectuées.

Validité externe : Dans cette étude aussi, nous avons analysé un grand nombre d'applications hétérogènes qui proviennent toutes du Google Play Store. Évidemment, le critère de sélection des applications (c'est-à-dire ne prendre que les plus populaires) a un effet sur les résultats, mais il nous fournit aussi un moyen neutre de sélectionner les applications tout en apportant de la diversité tant sur les attributs internes que les attributs externes. Le filtre que nous avons appliqué quant au nombre de versions était aussi nécessaire afin d'obtenir des résultats intéressants dans le temps.

Validité conceptuelle : Ici aussi la validité conceptuelle pourrait être menacée par notre définition de la taille d'une application lorsque nous prenons en compte le nombre de défauts de code. Toutefois, l'utilisation du nombre de classes pour estimer la taille et la complexité d'une application est une pratique courante dans le monde du génie logiciel comme évoqué précédemment. L'utilisation du nombre d'entités concernées suit la même logique. L'utilisation de la valeur résiduelle pourrait

aussi menacer la validité conceptuelle, toutefois la fiabilité des régressions utilisées nous laisse penser que ce sont des indicateurs fiables.
Fiabilité : Ici aussi, nous fournissons tous les détails nécessaires afin de permettre la reproduction de cette étude, de plus les outils statistiques utilisés sont clairement décrits.
Validité des conclusions : La menace principale aux conclusions de cette étude pourrait provenir de la manière dont nous avons interprété nos graphiques, toutefois nous estimons que les observations que nous avons faites sont clairement visibles sur chacun des graphiques présentés.

10.4 CONCLUSION

Cette étude a été l'occasion de prouver l'utilité de PAPRIKA pour suivre l'évolution des défauts de code au fil des versions. L'analyse de plus de 3000 versions nous a permis d'obtenir plusieurs résultats. Tout d'abord, nous avons découvert qu'il existe des relations entre l'évolution des différents défauts de code. Ensuite, nous avons découvert qu'il existe cinq tendances d'évolutions majeures des défauts de code en utilisant notre score qui se base sur la régression linéaire. Enfin, nous avons montré comment il est possible d'utiliser PAPRIKA pour analyser une application en détails et expliquer les changements observés. Nous croyons que les résultats présentés ici peuvent susciter un intérêt auprès des développeurs qui seront plus à même d'interpréter les résultats en disposant du code source et des informations de leurs gestionnaires de versions.

11

IMPACT DES DÉFAUTS DE CODE SUR LA PERFORMANCE

Ce chapitre décrit les résultats d'une étude empirique que nous avons effectuée sur deux applications afin d'évaluer l'impact de la correction de trois types de défauts de code sur la performance à l'aide de l'approche présentée dans le chapitre 6. Les trois défauts de code étudiés sont *Internal Getter/Setter*, *Member Ignoring Method* et *HashMap Usage*. Nous les avons choisis pour les mêmes raisons qui font qu'aujourd'hui PAPRIKA propose leur correction. Car ils sont simples à corriger sans changer le comportement de l'application et que la majorité du temps leur présence n'est pas du tout souhaitable dans les applications. De plus, bien qu'il soit recommandé de les éviter pour des raisons de performance, ces évaluations n'ont été effectuées qu'avec des tests de performances qui mesurent le temps d'exécution ou l'usage mémoire d'une méthode comme nous l'avons abordé dans l'état de l'art. Ces informations sont intéressantes mais insuffisantes pour conclure que la correction de ces défauts de code aura un intérêt pour l'utilisateur final. C'est pourquoi nous utilisons donc CURRY et ses métriques de plus haut niveau qui sont liées à l'expérience utilisateur. Il nous est ainsi possible de savoir si la correction des défauts de code n'a qu'un impact local ou alors si l'amélioration des performances va se répercuter au niveau global et être ainsi plus intéressantes pour le développeur. Pour cela, nous avons cherché à répondre aux cinq questions de recherche suivantes :

RQ_1 : *Est-ce que la correction de* Internal Getter/Setter, Member Ignoring Method *ou* HashMap Usage *améliore les performances au niveau de l'affichage ?*

RQ_2 : *Est-ce que la correction de* Internal Getter/Setter, Member Ignoring Method *ou* HashMap Usage *améliore les performances au niveau de l'utilisation de la mémoire ?*

RQ_3 : *Est-ce que la correction des trois défauts de code améliore significativement les performances au niveau de l'affichage en comparaison de la correction d'un seul défaut de code ?*

RQ_4 : *Est-ce que la correction des trois défauts de code améliore significativement les performances au niveau de l'utilisation de la mémoire en comparaison de la correction d'un seul défaut de code ?*

RQ_5 : *Est-ce que la correction des défauts de code a toujours un impact en utilisant le moteur ART plutôt que Dalvik ?*

11.1 CONCEPTION DE L'ÉTUDE

Dans cette section, nous détaillons la manière dont nous avons conçu l'étude en détails afin de permettre sa réplication.

11.1.1 *Objets*

Nous avons étudié deux applications libres dans cette étude. La première application est *SoundWaves Podcast*[1], c'est un client de baladodiffusion (*podcast* en anglais) qui permet à l'utilisateur de rechercher, télécharger et écouter des fichiers audio sur son appareil Android. Il utilise les moteurs de recherche iTunes et gPodder afin de trouver une grande variété de contenus. Cette étude a été effectuée sur la version 0.112 disponible sur GitHub[2] qui était la dernière version au moment de cette étude. Cette application contient environ 520 classes (classes internes comprises) et 2,672 méthodes.

La seconde application de cette étude est *Terminal Emulator for Android*[3], une application qui permet à l'utilisateur d'interagir avec l'interface de ligne de commande Linux disponible sur Android. Nous avons utilisé la dernière version disponible sur GitHub[4]. *Terminal Emulator* contient 141 classes et 978 méthodes.

Ces applications ont été choisies après l'analyse de 50 applications libres disponibles sur le dépôt F-Droid[5]. Nous avons choisi ces applications car les trois types de défauts de code sont présents dans des quantités qui permettent la correction manuelle puisque c'est une tâche qui prend beaucoup de temps, et qu'à cette époque PAPRIKA ne proposait pas la correction automatique.

Cette étude ne concerne que le paquetage principal de ces deux applications, à savoir `org.bottiger.podcast` et `jackpal.androidterm`. Toutes les bibliothèques externes, comme Picasso[6], sont donc exclues de cette étude puisqu'elles sont importées en utilisant Gradle, et que par conséquent les développeurs n'ont aucun contrôle sur le code.

11.1.2 *Sujets*

Les sujets de cette étude sont les défauts de code *Internal Getter/Setter*, *Member Ignoring Method* et *HashMap Usage* décrits dans le chapitre 4. Nous avons détecté 60 instances de défauts de code dans

1. Page Google Play Store de Soundwaves : https://play.google.com/store/apps/details?id=org.bottiger.podcast
2. Github de Soundwaves : https://github.com/bottiger/SoundWaves
3. Page Google Play Store de Terminal Emulator : https://play.google.com/store/apps/details?id=jackpal.androidterm
4. Github de Terminal Emulator : https://github.com/jackpal/Android-Terminal-Emulator
5. F-Droid : https://f-droid.org/
6. Picasso : http://square.github.io/picasso

SoundWaves : 24 *Internal Getter/Setter*, 29 *Member Ignoring Method* et 7 *HashMap Usage*. Par ailleurs, nous avons détecté 20 instances de défauts de code dans *Terminal Emulator* : 6 *Internal Getter/Setter*, 10 *Member Ignoring Method* et 4 *HashMap Usage*.

11.1.3 *Conception*

Pour déterminer l'impact de la correction des défauts de code, nous avons utilisé l'approche CURRY. Nous avons utilisé les scénarios décrits dans la sous-section 11.1.4. Les métriques ont été collectées pendant 60 exécutions du scénario. Au total, nous avons donc produit cinq versions pour chaque application comme on peut le voir sur le tableau 17. V_0 est la version de GitHub sans aucune modification. V_1, V_2, et V_3 sont des versions dérivées de V_0 avec la correction d'un seul des défauts de code. Sur la version V_4, tous les défauts de code sont corrigés. Nous n'utilisons que deux applications car la production des versions et les expérimentations demandent beaucoup de temps. Par exemple, l'exécution des 60 scénarios peut prendre plus de cinq heures pour une seule version. Afin de répondre à la RQ_5, nous avons instrumenté *SoundWaves Podcast* sous ART et Dalvik.

Nous avons effectué toutes les expérimentations sur un Motorola Moto G XT1032 8GB avec la version 4.4.4 (KitKat) d'Android. Ce téléphone possède un processeur Qualcomm Snapdragon 400 Quad core à 1.2 GHz, 1 Go de RAM et un écran de 4.5 pouces avec une résolution de 720 x 1280 pixels. C'est un téléphone de milieu de gamme pour l'année 2015.

Tableau 17 – Versions expérimentales

Version	Défauts corrigés
V_0	Aucun
V_1	*Internal Getter/Setter*
V_2	*Member Ignoring Method*
V_3	*HashMap Usage*
V_4	Tous (IGS + MIM + HMU)

11.1.4 *Procédure*

11.1.4.1 *Détection et correction des défauts de code :*

Nous avons détecté les défauts de code en utilisant PAPRIKA. La correction a été faite manuellement pour produire les différentes versions de chaque application. Notons que *HashMap Usage* est corrigé en utilisant `ArrayMap` du paquetage `android.support.v4.util` pour s'assurer que V_3 et V_4 gardent la même compatibilité de versions que V_0. L'implémentation dans `android.util` est seulement disponible pour l'API 19 et les

versions supérieures. La liste des défaut de code détectés et corrigés est disponible en ligne [7].

Tableau 18 – Nombre de défauts de code corrigés, nombre d'entités concernées et nombre moyen d'invocations des défauts de code durant le scénario

		# corrigés	# méthodes	# classes	moy. # invoc.
SoundWaves	IGS	24	21	11	3145
	MIM	29	29	21	4361
	HMU	7	4	4	21
	Total	60	53	33	7527
Terminal	IGS	6	2	4	2832
	MIM	10	10	8	29
	HMU	4	2	2	12
	Total	20	8	16	2873

Comme on peut l'apercevoir dans le tableau 18, nous avons détecté et corrigé un total de 80 défauts de code dans 49 classes pour les deux applications. Par conséquent, ces classes contiennent parfois plus d'une instance de défauts de code et certaines méthodes peuvent aussi contenir plus d'un défaut. Dans *SoundWaves*, les `HashMaps` sont déclarées deux fois dans des méthodes et 5 fois dans des classes tandis que 3 des 4 `HashMaps` de *Terminal emulator* sont déclarées dans des classes, mais elles sont toujours utilisées dans au moins une méthode. Évidemment, il ne peut y avoir qu'un seul *Member Ignoring Method* par méthode. Puisque que *Terminal Emulator* contient beaucoup moins de classes, l'application contient aussi moins de défauts de code que *Soundwaves*.

Nous avons instrumenté la version d'origine des applications pour compter le nombre d'invocations de chaque défaut de code, comme on peut l'apercevoir dans le tableau 18. Ces versions sont légèrement différentes de V_0 à cause du code integré pour obtenir cette information, par conséquent le nombre d'invocations peut être légèrement différent des autres versions. Les valeurs présentées ici permettent toutefois de donner un ordre de grandeur.

On peut observer que *Internal Getter/Setter* et *Member Ignoring Method* sont fréquemment invoqués dans *Soundwaves*, respectivement 3145 et 4361 fois. Seul *Internal Getter/Setter* est fréquent pour *Terminal Emulator* avec 2832 invocations contre 29 pour *Member Ignoring Method*. Ces valeurs élevées apparaissent car certaines des méthodes concernées sont directement ou indirectement appelées depuis les méthodes `onDraw()` des vues et activités. Elles peuvent donc être appelées

7. Liste des défaut de code corrigés : http://sofa.uqam.ca/paprika/mobilesoft16.php#CodeSmells

jusqu'à 60 fois par seconde. *HashMap Usage* est moins fréquent, toutefois la structure va rester en mémoire pendant une durée indéterminée. L'effet sur l'utilisation de la mémoire et le nombre d'appels au ramasse-miettes va donc lui aussi persister dans le temps.

11.1.4.2 *Scénarios d'utilisation :*

Pour les scénarios nous avons utilisé ROBOTIUM [8]. Pour *SoundWaves*, nous avons un scénario composé de 185 étapes (avec 90 opérations d'attente) qui utilise la plupart des fonctionnalités de l'application et explore la plupart des vues et menus. Par exemple, le scénario inclut la recherche d'un fichier audio avec l'aide de mots-clés et l'abonnement à un fournisseur de contenu audio. Ce scénario dure environ 235 secondes. Le scénario de *Terminal Emulator* contient 201 étapes (avec 91 opérations d'attente) et il dure environ 167 secondes. On y retrouve par exemple, l'utilisation de la ligne de commande *ls* et l'utilisation de plusieurs fenêtres de commandes. Le téléchargement et la lecture d'un fichier audio ne sont pas utilisés dans *SoundWaves*, car trop dépendants de la qualité du réseau et car cela utilise le lecteur par défaut d'Android pour lire la musique, ce qui sort du contexte de notre application. Les opérations d'attente sont utilisées pour s'assurer que les vues et les images sont totalement chargées avant d'utiliser une autre fonctionnalité, ainsi c'est toujours exactement le même scénario qui s'exécute. Nous avons aussi simulé les gestes de l'utilisateur, comme le défilement ou le glissement, afin d'être au plus proche d'une vrai expérience utilisateur. Les deux scénarios sont disponibles en ligne [9].

11.1.5 *Variables et hypothèses*

Variables indépendantes : Le nombre de *Internal Getter/Setter*, *Member Ignoring Method* ou *HashMap Usage* corrigés dans chaque version, ainsi que l'environnement ART ou Dalvik, sont les variables indépendantes de cette étude.

Variables dépendantes : Les variables dépendantes sont les métriques liées aux performances des applications présentées dans le chapitre 2.

Hypothèses : Pour répondre à nos cinq questions de recherche, nous formulons les hypothèses nulles suivantes, que nous avons utilisés sur les deux applications. Ici V_0, V_x ($x \in \{1 \dots 3\}$) et V_4 sont les versions décrites dans le tableau 30.

— $HR^{FT}_{V0\,Vx}$: Il n'y pas de différence entre le *temps d'affichage d'une trame* (FT) des versions V_0 et V_x ;

8. http://www.robotium.org
9. Scénarios d'utilisation : http://sofa.uqam.ca/paprika/mobilesoft16.php#Scenarios

— $HR^{DF}_{V0\,Vx}$: Il n'y pas de différence entre le *nombre de trames différées* (DF) des versions V_0 et V_x ;

— $HR^{MU}_{V0\,Vx}$: Il n'y pas de différence entre la *quantité de mémoire utilisée* (MU) des versions V_0 et V_x ;

— $HR^{GC}_{V0\,Vx}$: Il n'y pas de différence entre le *nombre d'appels au ramasse-miettes* (GC) des versions V_0 et V_x ;

— $HR^{FT}_{V4\,Vx}$: Il n'y pas de différence entre le *temps d'affichage d'une trame* (FT) des versions V_4 et V_x ;

— $HR^{DF}_{V4\,Vx}$: Il n'y pas de différence entre le *nombre de trames différées* (DF) des versions V_4 et V_x ;

— $HR^{MU}_{V4\,Vx}$: Il n'y pas de différence entre la *quantité de mémoire utilisée* (MU) des versions V_4 et V_x ;

— $HR^{GC}_{V4\,Vx}$: Il n'y pas de différence entre le *nombre d'appels au ramasse-miettes* (GC) des versions V_4 et V_x.

11.2 RÉSULTATS

Dans cette section, nous détaillons les résultats et répondons à nos cinq questions de recherche.

11.2.1 *Aperçu des résultats*

La figure 35 montre l'utilisation mémoire moyenne de toutes les versions de *SoundWaves* en fonction du temps, en reportant les 120 dernières secondes du scénario. Nous pouvons déjà constater que l'utilisation de la mémoire suit une courbe similaire pour l'ensemble des versions. Les performances sont d'ailleurs assez similaires sur cette métrique, avec toutefois un léger avantage pour les versions V_3 et V_4. Nous obtenons des résultats similaires pour *Terminal Emulator*.

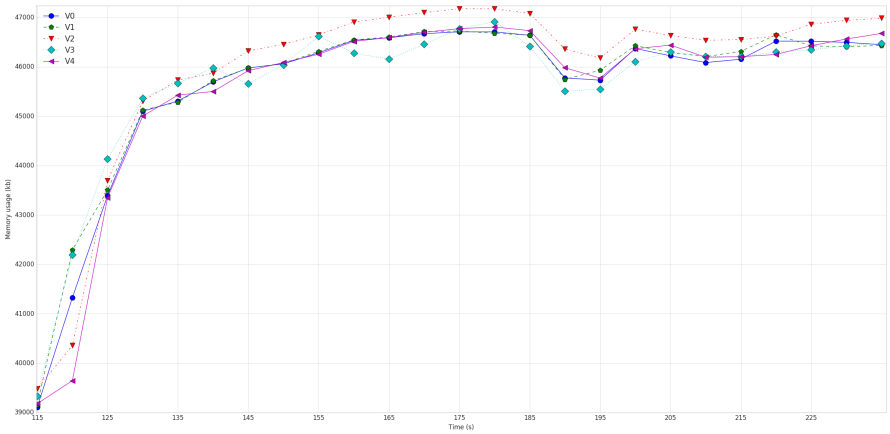


FIGURE 35 – Utilisation mémoire moyenne de toutes les versions de *SoundWaves* sur les 120 dernières secondes

Les valeurs moyennes de toutes les métriques sur 60 expérimentations pour les deux applications sont présentées dans tableau 19. Tout d'abord, on peut remarquer que dans la plupart des cas, la correction des défauts de code améliore les valeurs moyennes (en les faisant baisser), mais il existe tout de même des cas discordants qui peuvent surprendre. Par exemple, la correction de *Internal Getter/Setter* (V_1) augmente légèrement la quantité de mémoire utilisée pour les deux applications. Pour ces deux applications, c'est V_4 qui est la meilleure version concernant l'utilisation de la mémoire. D'ailleurs de manière générale, V_4 est une excellente version pour toutes les métriques car elle a tendance à accumuler les effets de l'ensemble des corrections. Pour les autres versions, les résultats ne sont pas similaires dans les deux applications. V_2 est la meilleure version pour *SoundWaves* concernant les métriques liées à l'affichage. Mais sur *Terminal Emulator*, V_2 est la meilleure version seulement sur le temps d'affichage d'une trame alors que V_1 est la meilleure sur le nombre de trame différées. V_4 est la meilleure version pour le ramasse-miettes sur *Terminal Emulator*, tandis que c'est V_3 pour *SoundWaves*. Nous nous attentions à de telles différences, étant donné les différences dans le scénario, le nombre et les emplacements des défauts de code corrigées pour les deux applications. Dans le reste de cette étude, nous nous concentrons sur les métriques et applications où l'on observe une différence significative entre les versions. Même si l'on peut déjà observer un impact positif en faveur des versions corrigées, la moyenne est insuffisante pour se prononcer sur la significativité des résultats. C'est pourquoi, nous avons aussi utilisé les pourcentages (tableau 20 et tableau 22) et des tests statistiques (tableau 21 et tableau 23) fournis par CURRY.

Tableau 19 – Valeurs moyennes des métriques sur 60 expérimentations

App	Version	Temps trame	Mémoire (KB)	Trame différées	Ramasse-miettes
SoundWaves	V_0	5.36	39,071.78	54.08	485.68
	V_1	5.32	39,119.85	50.27	494.07
	V_2	**5.30**	39,152.48	**47.40**	473.70
	V_3	5.33	38,920.65	51.73	**467.90**
	V_4	5.31	**38,887.32**	48.60	468.13
Terminal	V_0	3.83	13,223.11	39.78	78.22
	V_1	3.81	13,227.36	**39.18**	76.83
	V_2	**3.80**	13,217.89	40.07	77.43
	V_3	3.82	13,227.78	40.67	77.12
	V_4	3.82	**13,141.58**	40.27	**76.25**

Nous pouvons déjà constater en regardant le tableau 21 et le tableau 23 que les différences entre versions ne sont pas toujours significatives, en particulier concernant le temps d'affichage d'une trame et l'utilisation de la mémoire. Pour les trames différées et le ramasse-miettes, on constate dans le tableau 20 et le tableau 22, que les différences sont plus importantes que pour les autres métriques et bien souvent en faveur

Tableau 20 – Pourcentages de différence des métriques entre les versions de SoundWaves

	Temps trame	Mémoire	Trame différées	Ramasse-miettes
V_0,V_1	-0.61%	0.12%	-7.06%	1.73%
V_0,V_2	**-1.12%**	0.21%	**-12.36%**	-2.47%
V_0,V_3	-0.50%	-0.39%	-4.35%	**-3.66%**
V_0,V_4	-0.83%	**-0.47%**	-10.14%	-3.61%
V_1,V_4	-0.22%	-0.59%	-3.32%	-5.25%
V_2,V_4	0.30%	-0.68%	2.53%	-1.18%
V_3,V_4	-0.33%	-0.09%	-6.06%	0.05%

Tableau 21 – Mann-Whitney U et Cliff δ (f pour Faible, M pour Moyen, F pour Fort) pour toutes les versions de SoundWaves

	Mann-Whitney U test		Cliff's delta	
	Temps trame	Mémoire	Trame différées	Ramasse-miettes
V_0,V_1	0.065	0.786	**-0.314 (f)**	**0.708 (F)**
V_0,V_2	0.744	$<$**0.05**	**-0.285 (F)**	**0.729 (F)**
V_0,V_3	0.633	0.303	**-0.190 (f)**	**0.729 (F)**
V_0,V_4	0.618	0.123	**-0.364 (M)**	**0.728 (F)**
V_1,V_4	0.183	0.312	0.0002	**0.539 (F)**
V_2,V_4	0.475	$<$**0.05**	-0.019	**0.520 (F)**
V_3,V_4	0.953	0.250	**-0.149 (M)**	0.140

des nouvelles versions des applications. Toutefois, nous pouvons voir que pour le ramasse-miettes dans *SoundWaves*, Cliff δ (tableau 21) nous signale un fort impact négatif alors que les pourcentages (tableau 20) ont tendance à donner un impact positif pour les versions corrigées. En effet, si on prend les valeurs individuellement, le nombre d'appels au ramasse-miettes pour V_0 est souvent légèrement inférieur à ce qu'on obtient pour les autres versions, ce qui explique les résultats de Cliff δ. Mais cette version V_0 possède aussi quelques valeurs extrêmes qui dépassent de loin ce que l'on retrouve sur les autres versions, ce qui explique les différences que nous observons sur les pourcentages.

Dans le cas de *SoundWaves*, nous avons aussi collecté les résultats en utilisant le moteur ART. Le ramasse-miettes est exclu de cette comparaison car les améliorations apportées par ART [1] le rendent non pertinent pour notre analyse. En effet, nous ne relevons que deux appels partiels au ramasse-miettes durant le scénario sous ART, et cela pour toutes les versions. Nous ne présentons pas en détails les résultats pour ART mais ils sont disponibles en ligne [10]. Mais, pour résumé, nous n'observons pas de différence significative sur le nombre de trames différées et l'utilisation de la mémoire pour toutes les versions. Toutefois, il existe un impact significatif, mais très léger (aux alentours de 1%), pour toutes les versions. En comparaison de Dalvik, l'utilisation de la mémoire augmente d'environ 20% mais il y a un gain de 8% sur le temps d'affichage des trames et 18% de trames différées en moins.

Les résultats qui n'ont pas été abordés dans cette aperçu sont discutés plus en détails dans les réponses aux questions de recherche.

Tableau 22 – Pourcentages de différence des métriques entre les versions de Terminal Emulator

	Temps trame	Mémoire	Trame différées	Ramasse-miettes
V_0,V_1	-0.54%	0.03%	**-1.51%**	-1.77%
V_0,V_2	**-0.58%**	-0.04%	0.71%	-1.00%
V_0,V_3	-0.24%	0.04%	2.22%	-1.41%
V_0,V_4	-0.20%	**-0.62%**	1.21%	-**2.51**%
V_1,V_4	0.34%	-0.65%	2.76%	-0.76%
V_2,V_4	0.39%	-0.58%	0.50%	-1.53%
V_3,V_4	0.04%	-0.65%	-0.98%	-1.68%

10. Résultats de l'étude sous ART : http://sofa.uqam.ca/paprika/mobilesoft16.php#Results

Tableau 23 – Mann-Whitney U et Cliff δ (f pour Faible) pour toutes les versions de Terminal Emulator

	Mann-Whitney U test		Cliff's delta	
	Temps trame	Mémoire	Trame différées	Ramasse-miettes
V_0,V1	<**0.05**	0.934	-0.031	**-0.282 (f)**
V_0,V2	<**0.05**	0.797	0.060	-0.059
V_0,V3	0.122	0.985	0.102	**-0.151 (f)**
V_0,V_4	<**0.05**	<**0.05**	0.070	**-0.147 (f)**
V_1,V_4	0.487	<**0.05**	0.113	0.075
V_2,V_4	0.841	<**0.05**	0.045	-0.121
V_3,V_4	0.406	<**0.05**	-0.018	-0.05

11.2.2 *Est-ce que la correction de* Internal Getter/Setter*,* Member Ignoring Method *ou* HashMap Usage *améliore les performances au niveau de l'affichage ? (RQ1)*

11.2.2.1 Internal Getter/Setter

Les résultats des tableaux précédents tendent à montrer un impact très léger ou non significatif sur le temps d'affichage des trames pour les deux applications, nous acceptons donc $HR^{FT}_{V0\ V1}$. Concernant le nombre de trames différées, nous rejetons $HR^{DF}_{V0\ V1}$ et confirmons que la correction de *Internal Getter/Setter* réduit bien le nombre de trames différées comme nous pouvons l'observer pour *SoundWaves* dans le tableau 20 et le tableau 21. De plus, bien que l'amélioration ne soit pas significative pour *Terminal Emulator* (tableau 22 et tableau 23), on peut tout de même constater que V_1 reste la meilleure version pour cette métrique avec -1.51%. Les résultats observés montrent que la plupart des trames ne sont pas concernées par la correction des défauts de code, ce qui explique le manque d'impact sur le temps des trames, en revanche les trames concernées sont moins souvent différées. De plus, pour ces trames, les accesseurs et mutateurs sont appelés directement ou indirectement par la méthode `onDraw()` de certaines vues. Par conséquent, ils peuvent être appelés jusqu'à 60 fois par secondes.

11.2.2.2 Member Ignoring Method

Pour les mêmes raisons, nous acceptons $HR^{FT}_{V0\ V2}$ mais rejetons $HR^{DF}_{V0\ V2}$. Nous observons qu'il y un a impact positif sur le nombre de trames différées pour la correction de *Member Ignoring Method*. Les résultats ne sont pas significatifs pour *Terminal Emulator*, mais ce n'est pas surprenant vu le faible nombre d'invocations des méthodes concernées. Une fois de plus, dans *SoundWaves* les appels depuis `onDraw()` sont les responsables des effets observés.

11.2.2.3 HashMap Usage

Nous acceptons $HR^{FT}_{V0\,V3}$ mais rejetons $HR^{DF}_{V0\,V3}$ au vu des résultats significatifs de *SoundWaves*. Cet effet peut être expliqué par l'effet de bord de la correction sur le nombre d'appels au ramasse-miettes, qui est connu pour augmenter le nombre de trames différées, comme expliqué dans le chapitre 2.

11.2.2.4 *Résumé*

Nos résultats montrent que la correction de chacun des trois défauts de code n'a pas d'impact sur le temps d'affichage des trames mais réduit de manière significative le nombre de trames différées. Par conséquent, cela contribue à améliorer les performances en matière d'affichage.

11.2.3 *Est-ce que la correction de* Internal Getter/Setter*,* Member Ignoring Method *ou* HashMap Usage *améliore les performances au niveau de l'utilisation de la mémoire ? (RQ2)*

11.2.3.1 Internal Getter/Setter

Les résultats des deux applications ne montrent aucun impact de la correction de *Internal Getter/Setter* sur l'utilisation de la mémoire, par conséquent nous acceptons $HR^{Mu}_{V0\,V1}$. En revanche, les résultats sont partagés entre nos deux applications pour le nombre d'appels au ramasse-miettes. Sur *SoundWaves*, il y a une augmentation significative de 1.73% alors qu'on assiste à une réduction significative de 1.77% sur *Terminal Emulator*. Par conséquent, nous rejetons l'hypothèse nulle $HR^{GC}_{V0\,V1}$. En revanche, nous ne nous prononçons par sur le signe de l'effet constaté, qui peut être positif ou négatif. Cet effet pourrait apparaître surprenant puisque la définition de *Internal Getter/Setter* ne fait jamais mention d'un effet sur la mémoire. Toutefois, cela peut être expliqué par le fait que la machine virtuelle Dalvik utilise un cache pour les emplacements d'appels virtuels de méthodes, afin de procéder à des optimisations [87]. Puisque la correction supprime certains appels virtuels, cela peut affecter le comportement du ramasse-miettes. Plus de recherches seraient nécessaires afin de comprendre dans quels cas l'effet est positif et dans quel cas il devient négatif.

11.2.3.2 Member Ignoring Method

Ici aussi les résultats de *Terminal Emulator* sont peu significatifs à cause du peu de méthodes invoquées. Pour *SoundWaves*, la différence est notée comme significative par Mann-Whitney U, mais elle est seulement de 0.21% (tableau 20), par conséquent nous acceptons $HR^{Mu}_{V0\,V2}$. Nous rejetons $HR^{GC}_{V0\,V2}$ mais nous ne prononçons pas sur le signe de l'impact, comme expliqué dans l'aperçu des résultats (11.2.1), Cliff δ et les valeurs moyennes ne permettent pas le même constat. Nous ne

connaissons pas l'existence de spécificités de la machine virtuelle Dalvik qui pourraient expliquer cet effet sur le ramasse-miettes. Toutefois, nous émettons l'hypothèse d'un lien dû à l'usage d'un paramètre objet implicite vers la classe dès lors qu'une méthode non-statique est appelée.

11.2.3.3 HashMap Usage

Ici aussi les différences dans la quantité de mémoire utilisée sont vraiment légères pour les deux applications (-0.39% et 0.05%) et non significatives. Par conséquent, nous acceptons $HR^{MU}_{V0\,V3}$. Les *HashMap Usage* corrigés ne concernent que des structures contenant moins de 50 objets, ce que qui peut expliquer que la différence ne soit pas observable du point de vue global. Nous avons aussi des résultats conflictuels avec le ramasse-miettes pour *SoundWaves*, mais V_3 est la meilleure version avec une réduction moyenne de 3.66%. De plus, nous observons aussi un effet significatif et positif pour *Terminal Emulator*. Nous rejetons donc $HR^{GC}_{V0\,V3}$ pour *HashMap Usage*. On observe un effet sur le ramasse-miettes, même s'il y a très peu de ces défauts de code.

11.2.3.4 *Résumé*

Globalement, nos résultats montrent que la correction de *HashMap Usage* réduit le nombre d'appels au ramasse-miettes, et par conséquent améliore les performances au niveau de l'utilisation de la mémoire. Pour autant ce n'est pas le cas pour *Internal Getter/Setter* et *Member Ignoring Method*.

11.2.4 *Est-ce que la correction des trois défauts de code améliore significativement les performances au niveau de l'affichage en comparaison de la correction d'un seul défaut de code ? (RQ3)*

Concernant le temps d'affichage des trames, nous conservons les hypothèses nulles $HR^{FT}_{V4\,Vx}$ puisque la correction des trois défauts de code n'a pas d'impact significatif. Cela s'explique par le fait que la correction individuelle n'a pas non plus d'impact.

En nous basant sur les résultats du tableau 20 et du tableau 21, nous rejetons $HR^{DF}_{V4\,V0}$ et $HR^{DF}_{V4\,V3}$ mais acceptons $HR^{DF}_{V4\,V1}$ et $HR^{DF}_{V4\,V2}$. Dans les deux applications, les effets des défauts de code ont tendance à s'accumuler et par conséquent même si ce n'est pas la meilleure des versions, la version V_4 reste excellente quand on la compare à l'ensemble des versions. C'est donc un choix raisonnable.

11.2.5 *Est-ce que la correction des trois défauts de code améliore significativement les performances au niveau de l'utilisation de la mémoire en comparaison de la correction d'un seul défaut de code ? (RQ4)*

Concernant l'utilisation de la mémoire, les très bons résultats de V_4 sur *Terminal Emulator*, nous permettent de rejeter toutes les hypothèses $HR^{Mu}_{V4\,Vx}$. V_4 est aussi la meilleure version de *SoundWaves* avec -0.62% pour cette, même si comme on peut le voir dans la figure 35 c'est une amélioration mineure. Ces résultats sont intéressants car nous avions accepté toutes les hypothèses concernant l'utilisation de la mémoire pour la RQ2. C'est probablement dû aux effets très légers des corrections qui se sont accumulés et sont finalement devenus significatifs.

Concernant le ramasse-miettes, nous rejetons toutes les hypothèses nulles $HR^{GC}_{V4\,Vx}$ sauf $HR^{GC}_{V4\,V3}$. Toutefois, les performances sur cette métrique sont égales ou un peu meilleures pour V_3 que pour V_4. Pour *SoundWaves*, nous faisons la même observation que dans la sous-section 11.2.1 : prises individuellement, la plupart des valeurs pour V_0,V_1,V_2 et V_3 sont légèrement inférieures, mais en moyenne les autres valeurs sont largement plus grandes. Par conséquent, nous ne pouvons pas confirmer ou rejeter un effet positif pour cette application, mais l'effet est toujours positif sur *Terminal Emulator*.

En résumé, la correction des trois défauts a tendance à améliorer les performances concernant la mémoire en comparaison des autres versions.

11.2.6 *Est-ce que la correction des défauts de code a toujours un impact en utilisant le moteur ART plutôt que Dalvik ? (RQ5)*

Nos résultats sur *SoundWaves* tendent à montrer qu'il n'y aucun impact de la correction sur le nombre de trames différées et l'utilisation de la mémoire. Nous acceptons donc toutes les hypothèses $HR^{DF}_{V0\,Vx}$ et $HR^{Mu}_{V0\,Vx}$ pour ART. Concernant le temps d'affichage des trames, Mann-Whitney U nous donne un impact significatif. Nous rejetons donc toutes les hypothèses $HR^{FT}_{V0\,Vx}$. Mais l'impact est très léger car il n'y a une amélioration que d'environ 1% pour tous les défauts de code, cela explique que l'on observe aucun effet sur le nombre de trames différées.

11.3 MENACES À LA VALIDITÉ

Dans cette sous-section, nous discutons les problèmes principaux qui peuvent menacer la validité de cette étude.

Validité interne : Durant cette étude, nous avons été très prudents en interprétant les résultats. Nous avons aussi cherché à expliquer les causes de nos observations en parcourant le code source et le processus

des machines virtuelles. Nous sommes conscients que la correction de plusieurs défauts de code peut mener à des interactions non prévues, nous avons donc été très prudents lors de nos interprétations.

Validité externe : D'autres études sur d'autres applications et avec des défauts de code différents sont nécessaires afin de pouvoir prétendre avoir une connaissance étendue de l'impact des défauts de code sur la performance. D'ailleurs, les valeurs que nous avons trouvées sont spécifiques à nos applications, aux scénarios et au téléphone choisi, et ne sont donc pas généralisables. Nous n'affirmons donc pas que nos résultats permettent la prédiction des effets de la correction. Toutefois, nous avons prouvé que dans certains contextes, l'impact de la correction est existant sur les métriques choisies. Il y a donc un potentiel effet global plausible pour la plupart des applications.

Validité conceptuelle : Dans cette étude, les menaces à la validité conceptuelle peuvent être liées aux erreurs de mesures. C'est pourquoi nous avons répété les expérimentations 60 fois plutôt que de prendre des valeurs uniques. De plus, nous avons cherché à réduire au maximum les risques d'erreurs en utilisant seulement les services nécessaires à l'expérimentation. Nous avons aussi utilisé un vrai périphérique plutôt qu'un émulateur puisque l'émulation des processeurs est encore au stade expérimental [82].

Fiabilité : Nous avons essayé de fournir toutes les informations nécessaires afin de permettre la réplication de cette étude. De plus, nous fournissons en ligne les scénarios, l'approche et l'ensemble des données nécessaires à la réplication [11].

Validité des conclusions : Nous avons fait attention de respecter toutes les hypothèses des tests statistiques. C'est pourquoi CURRY utilise seulement des tests non-paramétriques qui ne font pas d'hypothèse sur la distributionze des métriques. Nous avons aussi été prudents de ne pas généraliser nos conclusions, surtout quand les résultats de nos applications n'étaient pas similaires.

11.4 CONCLUSION

Cette étude prouve que la correction des défauts de code *Internal Getter/Setter*, *Member Ignoring Method* ou *HashMap Usage* peut avoir un impact sur l'expérience de l'utilisateur en améliorant les performances. Nous complétons donc les connaissances qui étaient disponibles sur l'impact des performances au niveau local. Évidemment, l'impact ne sera pas significatif sur toutes les applications et dépend fortement du scénario utilisateur. Toutefois, nous recommandons la correction systématique de ces défauts de code, d'autant plus qu'elle est maintenant automatique avec PAPRIKA.

11. Matériaux de l'étude : http://sofa.uqam.ca/paprika/mobilesoft16.php

12

IMPACT DES DÉFAUTS DE CODE SUR LA CONSOMMATION D'ÉNERGIE

Ce chapitre est similaire au chapitre précédent mais cette fois nous avons considéré l'impact de la présence des défauts de code sur la consommation d'énergie plutôt que sur les performances. Nous avons donc utilisé l'approche HOT-PEPPER du chapitre 7 pour évaluer cet impact sur les trois mêmes défauts de code, à savoir *Internal Getter/Setter*, *Member Ignoring Method* et *HashMap Usage*. Nous avons profité de la correction automatique de PAPRIKA pour effectuer ces expérimentations sur cinq applications. Toutefois, précisons que les processus de sélection des applications, de création des scénarios et d'expérimentations restent longs, ce qui explique le nombre d'applications. Notons aussi que les résultats présentés ici n'ont pas encore été publiés et qu'ils restent donc préliminaires. Toutefois, nous les estimons assez fiables et la rédaction d'un article les exploitant est d'ailleurs en cours. Comme nous l'avons montré dans notre état de l'art, l'impact de la correction des défauts de code dans de vrais applications reste encore largement méconnu, c'est pourquoi nous avons cherché à répondre aux quatre questions de recherche suivantes :

RQ$_1$: *Est-ce que la correction de* Internal Getter/Setter *réduit la consommation d'énergie de l'application ?*

RQ$_2$: *Est-ce que la correction de* Member Ignoring Method *réduit la consommation d'énergie de l'application ?*

RQ$_3$: *Est-ce que la correction de* HashMap Usage *réduit la consommation d'énergie de l'application ?*

RQ$_4$: *Est-ce que la correction des trois défauts de code améliore significativement la consommation d'énergie en comparaison de la correction d'un seul défaut de code ?*

12.1 CONCEPTION DE L'ÉTUDE

Dans cette section, nous détaillons la manière dont nous avons conçu l'étude en détails afin de permettre sa réplication.

12.1.1 *Objets*

Nous avons étudié cinq applications libres dans cette étude. Ces applications sont *SoundWaves Podcast*[1], *Aizoban*[2], *Opacclient*[3], *Calculator*[4] and *Todo*[5]. Elles ont été choisies après l'analyse de plus de 1900 applications du dépôt F-droid, en tentant de suivre les critères suivants :

— Des applications qui contiennent au minimum deux types de défauts de code parmi les trois étudiés,

— Des applications appartenant à des catégories et ayant des types d'utilisations diversifiés,

— Des applications qui soient populaires en terme de nombre de téléchargements et de notes,

— Des applications stables, facilement compilables et permettant la production facile d'un scénario d'utilisation.

SoundWaves Podcast est la même application que celle du chapitre précédent, nous utilisons la version 0.130. *Aizoban* est un catalogue de mangas qui permet à l'utilisateur de télécharger et lire des mangas sur Android, nous utilisons la version 1.2.5. *Opacclient* qui permet d'effectuer des réservations dans plus de 500 bibliothèques publiques de 30 pays, nous utilisons la version 4.5.9. *Calculator* est la calculatrice par défaut du système Android CyanogenMod[6], nous utilisons la version 5.1.1. *ToDo* est une application permettant de gérer une liste de tâches à effectuer, nous utilisons la version 1.0. La taille en terme de nombre de classes et le nombre de méthodes de chaque application sont donnés dans le tableau 24.

Tableau 24 – Nombre de classes, de méthodes et de défauts de code corrigés dans chaque application

App	Classes	Méthodes	HMU	IGS	MIM	Total
Aizoban	524	2773	39	190	110	339
Calculator	147	830	0	10	8	18
Web Opac	367	2176	48	77	43	168
SoundWaves	520	2,672	5	47	14	66
Todo	161	610	9	3	0	12

1. https://play.google.com/store/apps/details?id=org.bottiger.podcast
2. https://f-droid.org/repository/browse/?fdfilter=manga&fdid=com.jparkie.aizoban
3. https://play.google.com/store/apps/details?id=de.geeksfactory.opacclient&hl=en
4. https://play.google.com/store/apps/details?id=com.android2.calculator3&hl=en
5. https://f-droid.org/repository/browse/?fdfilter=todo&fdid=com.xmission.trevin.android.todo
6. http://www.cyanogenmod.org

12.1.2 *Sujets*

Les sujets de cette étude sont les défauts de code *Internal Getter/-Setter*, *Member Ignoring Method* et *HashMap Usage* décrits dans le chapitre 4. Le nombre d'instances détectées dans chaque application est donnée dans le tableau 24. Notons que certaines applications ne contiennent parfois pas un certain type de défaut de code, mais toujours au moins deux types.

12.1.3 *Conception*

Pour déterminer l'impact sur la consommation d'énergie, nous avons utilisé l'approche HOT-PEPPER. Nous avons fait chaque expérimentation 20 fois pour chacune des versions des applications en respectant toutes les précautions énoncées dans le chapitre 7. Nous avons ici aussi produit cinq versions pour chaque application comme pour le tableau 17 du chapitre précédent. V_0 est la version de GitHub sans aucune modification. V_1, V_2, et V_3 sont des dérivées de V_0 avec la correction d'un seul des défauts de code. Sur la version V_4, tous les défauts de code sont corrigés.

Ici les expérimentations ne concernent que le moteur Dalvik. Nous les avons effectuées sur un téléphone Google Nexus 4 avec le système CyanogenMod 11 qui repose sur Android 4.4.4 (KitKat). Le Nexus 4 possède un processeur Qualcomm Snapdragon S4 (1.5 GHz quad-core), un écran de 4,7 pouces avec une résolution de 768x1280 pixels ainsi que 2 Go de RAM.

12.1.4 *Procédure*

12.1.4.1 *Détection et correction des défauts de code :*

Comme détaillé dans le chapitre 7, nous avons utilisé PAPRIKA pour détecter et corriger l'ensemble des défauts de code des applications. Toutes les versions générées sont issues de la version V_0, nous avons aussi vérifié manuellement les corrections pour s'assurer que PAPRIKA fonctionne correctement.

12.1.4.2 *Scénarios d'utilisation :*

Les scénarios d'utilisation ont été créés manuellement en utilisant CALABASH [7] qui est similaire à ROBOTIUM utilisé précédemment. Ici aussi nous avons essayé de produire des scénarios complets simulant le comportement d'un utilisateur. Le tableau 25 rassemble le nombre d'étapes moyen de chaque scénario ainsi que le temps d'exécution et d'attente moyen pour chaque application. Le tableau 26 montre le nombre moyen

7. http://calaba.sh

de défauts de code invoqués durant l'exécution des scénarios. Comme on peut le voir, nous avons des scénarios variés tant en terme de temps d'exécution que de défauts de code invoqués, ce qui s'explique par la diversité de nos applications.

Tableau 25 – Nombre moyen d'étapes, temps d'attente et d'éxécution moyen pour chaque scénarios

App	Étapes	Attente	Exécution
Aizoban	169	17s	3 min 54 s
Calculator	325	0s	14 min 15 s
OpacClient	136	79s	3 min 03 s
SoundWaves	172	53s	9 min 43 s
Todo	248	1s	4 min 38 s

Tableau 26 – Nombre moyen de défauts invoquées durant l'exécution du scénario

App	#HMU	#IGS	#MIM
Aizoban	10	1300	0
Calculator	0	6122	1350
SoundWaves	420	8053	6560
Todo	40	20	0
Web Opac	6	133	40

12.1.5 *Variables et hypothèses*

Variables indépendantes : Le nombre de *Internal Getter/Setter*, *Member Ignoring Method* ou *HashMap Usage* corrigés dans chaque version.

Variables dépendantes : Les variables dépendantes sont les métriques liées à la consommation que nous récupérerons, c'est-à-dire le temps d'exécution du scénario, les valeurs d'intensité ainsi que le voltage de la batterie. Nous utilisons ensuite la formule présentée dans le chapitre 2 afin de calculer la consommation d'énergie à partir de ces valeurs.

Hypothèses : Pour répondre à nos cinq questions de recherche, nous formulons les hypothèses nulles suivantes, que nous avons utilisées sur l'ensemble des applications. Ici V_0, V_1, V_2, V_3 et V_4 sont les versions décrites dans le tableau 30.

— $HR^{CE}_{V0\,Vx}$: Il n'y pas de différence de consommation d'énergie entre les versions V_0 et V_X ;

— $HR^{CE}_{V4\,Vx}$: Il n'y pas de différence de consommation d'énergie entre les versions V_4 et V_X ;

12.2 RÉSULTATS

Dans cette section, nous détaillons les résultats et répondons à nos quatre questions de recherche.

12.2.1 *Aperçu des résultats*

Le tableau 27 montre la consommation moyenne d'énergie en Joules pour chaque application. Certaines versions n'ont pas de résultat car le défaut de code n'étant tout simplement pas présent dans l'application. On peut déjà remarquer que les versions corrigées consomment généralement moins que la version de base. La version sans aucun défaut de code est d'ailleurs la meilleure pour deux des cinq applications *Calculator* et *Todo*). Le tableau 28 avec les pourcentages précise ces résultats. On remarque donc que la différence peut atteindre 4,83% pour *Todo*. Dans le cas où la différence n'est pas à l'avantage de la correction, c'est-à-dire pour V_2 avec *Aizoban* et *SoundWaves*, le pourcentage de différence reste faible (0.08% et 0.29%). En observant le tableau 29, on remarque que dans la plupart des cas, les résultats sont significatifs et aussi en faveur des applications corrigées. Ce tableau présente aussi la comparaison entre V_4 et la meilleure des versions. Nous expliquons ces résultats plus en détails en répondant aux questions de recherche.

Tableau 27 – Consommation moyenne d'énergie (en Joules) de chaque application

App	V_0	V_1	V_2	V_3	V_4
Aizoban	423.87	419.25	424.61	**415.37**	418.04
Calculator	300.16	299.63	298.80	-	**295.10**
SoundWaves	859.24	**846.96**	866.71	855.90	848.17
Todo	369.60	362.05	-	360.73	**351.75**
Web Opac	391.93	383.78	**376.78**	383.86	378.21

12.2.2 *Est-ce que la correction de* Internal Getter/Setter *réduit la consommation d'énergie de l'application ? (RQ1)*

On peut constater dans le tableau 28 et le tableau 29 que les résultats sont significatifs et en faveur des applications sans *Internal Getter/Setter* sauf dans le cas de l'application *Calculator*, malgré les 6122 invocations du défaut de code. C'est aussi la meilleure version pour *SoundWaves*. Toutefois, nous rejetons quand même $HR^{CE}_{V0\,V1}$ au vu des résul-

Tableau 28 – Pourcentage de différence de la consommation moyenne d'énergie (en J) de chaque version par rapport à V_0

App	V_0, V_1	V_0 ,V_2	V_0 ,V_3	V_0 ,V_4
Aizoban	-1.09%	+0.08%	**-2.00%**	-1.38%
Calculator	-0.18%	-0.45%	-	**-1.69%**
SoundWaves	**-1.43%**	+0.29%	-0.38%	-1.29%
Todo	-2.04%	-	-2.40%	**-4.83%**
Web Opac	-2.08%	**-3.86%**	-2.06%	-3.50%

Tableau 29 – Cliff δ pour toutes les versions,V_m représente la meilleure version parmi l'ensemble que nous comparons à V_4 (f pour Faible, M pour Moyen, F pour Fort)

Apps	V_0, V_1	V_0 ,V_2	V_0 ,V_3	V_0 ,V_4	V_m ,V_4
Aizoban	0.46 (M)	-0.06	0.58 (F)	0.58 (F)	-0.01
Calculator	0.11	0.18 (f)	-	0.42 (M)	-
SoundWaves	0.26 (f)	-0.24(f)	0.08	0.26 (f)	0
Todo	0.62 (F)	-	0.66 (F)	0.92 (F)	-
Web Opac	0.46 (M)	0.69 (F)	0.43 (M)	0.60 (F)	-0.11

tats sur les autres applications, et confirmons donc que la correction de *Internal Getter/Setter* peut réduire la consommation d'énergie.

12.2.3 *Est-ce que la correction de* Member Ignoring Method *réduit la consommation d'énergie de l'application ? (RQ2)*

Les résultats pour *Member Ignoring Method* sont plus mitigés avec des résultats positifs pour *Calculator* (faible) et *Web Opac* (fort) tandis qu'il y avoir une faible augmentation de la consommation sur *SoundWaves*. Toutefois, sur *SoundWaves*, cela se traduit par une augmentation de seulement 0,29% alors que l'on obtient une baisse de 3.86% pour *Web Opac* qui est la meilleure version de l'application. Nous considérons donc que la correction de *Member Ignoring Method* peut réduire la consommation d'énergie et rejetons $HR^{CE}_{V0\,V2}$. Toutefois des études supplémentaires sont nécessaires pour confirmer ces résultats et comprendre dans quel contexte les résultats sont positifs ou négatifs.

12.2.4 *Est-ce que la correction de* HashMap Usage *réduit la consommation d'énergie de l'application ? (RQ3)*

Les résultats pour *HashMap Usage* sont tous en faveur des versions corrigées et significatifs pour trois des quatre applications concernées (*Aizoban*, *Todo* et *Web Opac*). C'est d'ailleurs la meilleure version pour

Aizoban. Nous rejetons donc $HR^{CE}_{V0\ V3}$ et considérons que la correction de *HashMap Usage* peut réduire la consommation d'énergie d'une application.

12.2.5 *Est-ce que la correction des trois défauts de code améliore significativement la consommation d'énergie en comparaison de la correction d'un seul défaut de code ? (RQ4)*

Le tableau 29 compare aussi la version V_4 à la meilleure des versions pour chaque application. On remarque ici qu'il n'y a pas de différence significative, et donc la version avec tous les défauts de code corrigées n'est jamais pire que les autres versions. C'est aussi la meilleure des versions pour *Calculator* et *Todo*. Toutefois, nous ne rejetons pas $HR^{CE}_{V4\ Vx}$ car la différence avec les autres versions reste minime. Nous avons donc la même conclusion que pour le chapitre précédent, bien que V_4 ne soit pas significativement meilleure que les autres versions, cela reste une excellente version qui proposera une réduction de la consommation similaire à la meilleure des versions.

12.3 MENACES À LA VALIDITÉ

Dans cette sous-section, nous discutons les problèmes principaux qui peuvent menacer la validité de cette étude. Les menaces présentées ici sont similaires à celles du chapitre précédent, toutefois viennent s'ajouter les menaces liées à l'aspect physique des mesures.

Validité interne : Durant cette étude, nous avons été très prudents en interprétant les résultats. Puisque nous n'avons modifié que les défauts de code dans le code source, il nous est possible de faire le lien entre la correction et les observations effectuées. Notons que nous avons vérifié manuellement les modifications apportées par PAPRIKA pour s'assurer de l'exactitude de ce lien.

Validité externe : Ici aussi d'autres études sur d'autres applications et avec des défauts de code différents sont nécessaires afin de pouvoir prétendre avoir une connaissance étendue de l'impact des défauts de code sur la consommation. Les valeurs que nous avons trouvées sont spécifiques à nos applications, aux scénarios et au téléphone choisi, et ne sont donc pas généralisables. Nous n'assumons donc pas que nos résultats permettent la prédiction des effets de la correction. Toutefois, nous avons prouvé que dans certains contextes, l'impact de la correction est existant sur la consommation d'énergie. Il y a donc un potentiel effet global plausible pour la plupart des applications.

Validité conceptuelle : Dans cette étude, les menaces à la validité conceptuelle peuvent être liées aux erreurs de mesures. C'est pourquoi nous avons répété les expérimentations 20 fois plutôt que de prendre des valeurs uniques. De plus, nous avons cherché à réduire au maximum les risques d'erreurs en minimisant de manière rigoureuses les facteurs ex-

ternes qui pourraient interférer dans nos mesures. Notons aussi que les mesures de notre ampèremètre pourraient ne pas être exactes puisque ce sont des mesures physiques, toutefois nous avons utilisé deux ampèremètres différents qui donnaient des mesures similaires lors des tests de HOT-PEPPER. De plus, nous utilisons majoritairement des valeurs relatives qui ne fausseront pas les conclusions, d'autant plus que les valeurs obtenues lors des 20 expérimentations étaient souvent similaires.
Fiabilité : Nous avons essayé de fournir toutes les informations nécessaires afin de permettre la réplication de cette étude. Elles sont disponibles en ligne [8].
Validité des conclusions : Nous avons fait attention de respecter toutes les hypothèses des tests statistiques. C'est pourquoi HOT-PEPPER utilise seulement des tests non-paramétriques qui ne font pas d'hypothèses sur la distribution des métriques. Nous avons aussi été prudent de ne pas généraliser nos conclusions.

12.4 CONCLUSION

Cette étude qui utilise HOT-PEPPER prouve que la correction des défauts de code *Internal Getter/Setter*, *Member Ignoring Method* ou *HashMap Usage* peut réduire la consommation d'énergie d'une application de manière significative. Ici aussi, nous complétons donc les connaissances qui étaient disponibles sur l'impact sur la consommation au niveau local. Évidemment, ici aussi l'impact ne sera pas significatif sur toutes les applications et dépend fortement du scénario utilisateur. Mais dans tous les cas, nous recommandons la correction systématique de ces défauts de code avec PAPRIKA puisqu'il ne semble y avoir aucun effet négatif.

8. Matériaux de l'étude : https://github.com/SOMCA/hot-pepper-data

Cinquième partie

CONCLUSION ET PERSPECTIVES

13

CONCLUSION

Les applications mobiles sont relativement nouvelles dans l'univers du génie logiciel, elles diffèrent des applications de bureaux traditionnelles par leurs compositions et leurs besoins en terme de développement. De plus, elles reposent souvent sur des cycles d'évolution très courts afin de satisfaire des utilisateurs exigeants et afin de s'adapter à leurs environnements qui sont eux aussi en constante évolution. Il est donc nécessaire de fournir aux développeurs de ces applications les connaissances et les outils adaptés afin de leur permettre de fournir des applications de grande qualité. Les défauts de code sont des indicateurs de qualité bien connus pour les applications classiques et leur correction est souvent un bon moyen d'améliorer et d'assurer la pérennité d'un logiciel facilement. Évidemment des processus de développement logiciel rigoureux et des tests réguliers doivent être mis en œuvre en parallèle car la correction des défauts de code n'est qu'un facteur de qualité parmi d'autres. Malheureusement dans le cadre des applications mobiles, les connaissances sur les défauts de code sont encore lacunaires et les outils pour les détecter et les corriger peu matures ou inexistants. Les ressources en ligne traitant des défauts de code et autres mauvaises pratiques similaires existent et montrent l'intérêt de la communauté des développeurs d'applications mobiles pour le sujet. Mais elles sont éparpillées et elles reposent souvent sur des assertions qui restent à prouver. Cela rend donc la prise en compte des recommandations qui y sont faites difficiles et incertaines pour les développeurs. Nos études tendent à prouver que les développeurs ne suivent pas fréquemment ces recommandations d'ailleurs, même quand elles proviennent de sources sûres comme la documentation d'Android. Il y a donc un travail de collecte et de structuration des connaissances nécessaires à réaliser. Ce rôle incombe sans doute à la communauté scientifique qui l'a déjà effectué pour les défauts de code dans d'autres domaines, ce qui a permis la prise en compte des défauts de code dans des outils courants tels que les environnements de développement. Notons que nous ne sommes pas les seuls à avoir cet avis, et que plusieurs des chercheurs que nous citons dans notre état de l'art font le même constat.

Nous avons donc tenté au travers de cette thèse, de faire avancer à la fois la connaissance et les outils afin de compléter les savoirs actuels sur les défauts de code dans les applications mobiles. Pour commencer, nous avons donc proposé une **classification de 17 défauts de code pour la plate-forme Android**, en particulier cette classification comporte 13 défauts de code spécifiques à Android dont la plupart n'apparaissaient pas dans la littérature scientifique avant nos travaux. Nous estimons que cette classification est destinée à évoluer et à être complé-

tée, toutefois elle permet de commencer à structurer les connaissances éparses sur les défauts de code de la plate-forme Android.

Ensuite, nous fournissons **une approche outillée nommée PAPRIKA capable de détecter ces défauts de code** de manière efficace en analysant le code binaire des applications. C'est une approche flexible qui permet la détection de nombreux défauts de code et qui pourrait être étendue facilement en écrivant des requêtes de juste quelques lignes. De plus, elle innove dans son traitement des métriques, en utilisant une base de données pour calculer les seuils de détection et en proposant l'utilisation de la logique floue pour proposer des résultats similaires au raisonnement humain. Nous avons validé notre approche de diverses manières, notamment en faisant appel à des experts, et nous l'avons exploité afin d'analyser de grandes quantités d'applications populaires. En effet, notre approche permet d'analyser n'importe quelle application puisqu'elle se base sur le code binaire et non le code source. Cela facilite donc la production d'études et nous espérons d'ailleurs que notre outil sera utilisé par la communauté scientifique ou des développeurs dans le futur. Cette approche propose aussi depuis peu **la correction de trois des défauts de code** directement sur le code de l'application, évidemment l'implémentation de la correction d'autres défauts de code est possible mais elle dépend de la complexité de la correction des défauts en question. Bien que perfectible, ces premières corrections sont un premier pas dans la prise en charge des défauts de code par des outils automatisés. Cette approche nous a permis de mener deux études à large échelle, et ainsi de découvrir entre autre qu'il existe des tendances d'évolution de la présence des défauts de code au fil des versions, que certains défauts de code ont tendance à être plus présents dans les classes héritées du cadre d'application Android ou encore que les applications les plus téléchargées ont tendance à contenir moins de défauts de code.

Nous proposons aussi **l'approche CURRY qui permet de mesurer l'impact des défauts de code sur la performance**. En comparant plusieurs versions d'une même application, CURRY permet de détecter la version la plus efficace sur des métriques qui sont liées à l'expérience utilisateur. En particulier, cette approche prend en compte des métriques liées à l'affichage et à l'utilisation de la mémoire, ce qui diffère des résultats des études précédentes qui ne présentent généralement que des métriques bas-niveau comme le temps d'exécution ou la quantité de mémoire utilisée par une méthode. Cette approche repose aussi sur l'utilisation d'un scénario utilisateur permettant de tester des applications réelles d'une manière proche de ce que ferait l'utilisateur de l'application. Nous avons donc étudié l'impact de la correction de trois défauts de code dans deux applications Android, et en avons conclu que de manière générale la correction des défauts de code est recommandable. Toutefois, notons que plus d'études seraient nécessaires afin de confirmer nos résultats.

De la même manière, nous proposons l'outil **Hot-Pepper qui permet de mesurer l'impact des défauts de code sur la consommation d'énergie**. À l'aide d'un ampèremètre et d'un scénario utilisateur, cette approche nous permet de déterminer si une version d'une application consomme moins qu'une autre version. Une fois de plus, nous avons étudié l'impact de la correction sur trois défauts de code mais cette fois sur cinq applications. Ici, nous concluons qu'il est préférable de corriger les défauts de code afin de diminuer la consommation d'énergie, toutefois d'autres études seraient nécessaires pour confirmer ces résultats.

Ces deux dernières approches sont donc destinées à motiver les développeurs à corriger les défauts dans leurs applications ou à les éviter durant le développement, en confirmant que leur absence sont bénéfiques pour l'utilisateur et pourront donc améliorer les avis des utilisateurs sur les applications.

Le travail présenté dans cette thèse, représente donc un avancement sur les connaissances et les outils qui pourront servir de bases aux futures recherches sur les défauts de code dans les applications mobiles.

14 PERSPECTIVES

Les perspectives de recherche pour compléter nos travaux sont nombreuses. Évidemment cela passe par une amélioration des outils mais aussi par la production d'études complémentaires. Nous organisons ce chapitre en abordant tout d'abord les perspectives à court terme puis celles à long terme.

14.1 PERSPECTIVES À COURT TERME

14.1.1 *Amélioration de la classification*

La classification pourrait être complétée avec de nouveaux défauts de code et si nécessaire de nouvelles catégories. Une première étape serait d'y ajouter des défauts de code qui ne sont pas liés à la performance ou à la maintenance, avec par exemple des défauts de code qui affectent l'expérience utilisateur ou la sécurité de l'application. Il est aussi possible de compléter la liste des défauts de code orientés objets qui sont toujours pertinents pour les applications mobiles.

14.1.2 *Amélioration de* PAPRIKA

Tout d'abord, il serait nécessaire de compléter et d'améliorer le processus de correction de défauts de code de PAPRIKA. Afin de le rendre plus fluide, il serait pratique de permettre à PAPRIKA d'analyser le code source pour la détection des défauts, afin de permettre la détection et la correction en une seule étape plutôt que deux.

Concernant l'analyse pure de code binaire, nous estimons que PAPRIKA est assez mature, toutefois il serait sans doute possible d'optimiser certaines requêtes. Une évolution souhaitable serait l'intégration de PAPRIKA dans les environnements de développement, toutefois l'utilisation de la base de données rend ce processus plus complexe que pour les outils d'analyse classiques puisqu'il faut théoriquement de nombreuses applications dans la même base pour rendre l'approche intéressante. Il convient donc de réfléchir à cette intégration en proposant par exemple un processus d'analyse à distance. En effet, PAPRIKA pourrait proposer des analyses en ligne et gérer ainsi la base de données complète des applications sur un serveur tout en ne donnant que des résultats locaux aux utilisateurs. Cela serait aussi un bon moyen d'augmenter la taille et la diversité des données par rapport à ce dont nous disposons actuellement. Cette base pourrait aussi tout simplement envoyer les informa-

tions concernant les seuils et la logique floue de manière périodique aux clients tout en permettant une analyse hors-ligne.

14.1.3 *Impact des défauts de code*

Concernant l'impact des défauts de code, il est évident que plus d'études seraient nécessaires. Il faudrait notamment étudier plus d'applications, plus de défauts et obtenir plus de résultats pour ART. Cela demande des moyens et du temps, mais il serait aussi intéressant de répliquer nos expérimentations sur d'autres périphériques mais aussi en utilisant d'autres versions d'Android. Nous avons tenté de fournir des approches très génériques qui puissent être adaptées à différents contextes, ce qui devrait faciliter ces travaux. Il serait aussi intéressant de pouvoir analyser de gros volumes d'applications, toutefois cela reste complexe à cause de la nécessité du scénario (mais il est sans doute possible d'automatiser cette étape dans une certaine mesure) mais aussi à cause des manipulations physiques dans le cas de la consommation d'énergie.

14.2 PERSPECTIVES À LONG TERME

14.2.1 *Correction de défauts de code complexe*

Les corrections proposées par PAPRIKA sont encore assez simples, et il serait intéressant de s'attaquer à des défauts de code qui demandent une correction plus complexe. En effet, actuellement les trois défauts peuvent se corriger d'une manière systématique et comme nous l'avons vu la correction est souhaitable dans la majorité de cas. Toutefois, ce n'est pas le cas pour tous les défauts qui peuvent dans certains cas se corriger de plusieurs façons (par exemple pour savoir comment un *Heavy Service Start* doit être réusiner) ou alors ne pas nécessiter une correction systématique (lorsque l'on va estimer que la correction peut produire des effets de bord plus néfastes que la présence du défaut). Il convient donc d'apporter plus d'intelligence dans la correction mais aussi de donner le choix aux développeurs dans certains cas. Il sera alors sans doute nécessaire de rendre l'approche plus flexible, avec par exemple de l'intelligence artificielle pour apprendre des corrections ultérieures effectuées manuellement. Il sera sans doute aussi nécessaire de proposer des choix multiples de corrections aux développeurs qui sont souvent les plus à même de juger de la pertinence d'une correction. Il faudra alors intégrer un processus de suivi à long terme des corrections avec un retour vers PAPRIKA une fois que celles-ci sont effectuées, ainsi les corrections pourront s'adapter et s'améliorer au fil des versions.

14.2.2 *Détection et correction de défauts sur d'autres plate-formes*

Notons aussi qu'il serait intéressant de proposer PAPRIKA pour d'autres systèmes mobiles comme iOS ou Windows Phone. Sarra Habchi a d'ailleurs commencé des travaux dans ce sens à l'INRIA de Lille en permettant l'analyse de code Objective-C et Swift et la détection de plusieurs défauts de code spécifiques à iOS en modifiant notre approche. Bien que l'approche puisse être facilement adaptée aux autres plate-formes en principe (il suffit normalement de remplacer le moteur d'analyse de code et d'écrire de nouvelles requêtes), les défis ne sont pas à négliger. Tout d'abord, il n'existe pas forcément d'équivalent à SOOT ou SPOON pour les autres langages. Il devient alors nécessaire de créer ces équivalents. Cela peut déjà s'avérer complexe pour l'analyse de code source (ou du code binaire lorsque compilé) mais ça l'est encore plus pour la correction qui doit s'assurer que le code produit soit correct. Enfin, il y a aussi beaucoup de travail préliminaire à effectuer pour pouvoir écrire les requêtes. En effet, les ressources sont peu nombreuses pour les autres plateformes et la recherche de défauts de code spécifique à la plateforme sera donc encore plus compliquée que pour Android.

14.2.3 *Exploitation des bases de données*

En plus de fournir d'autres études sur les défauts de code sur Android, il serait aussi intéressant d'utiliser les modèles des applications que nous analysons pour obtenir d'autres connaissances. En effet, ces modèles sont très complets et pourraient ainsi servir à d'autres études. En particulier, nous avons déjà envisagé de faire émerger des ensembles de bonnes et de mauvaises pratiques en utilisant de l'intelligence artificielle sur l'ensemble des données analysées. Il serait alors possible de découvrir de nouveaux défauts de code mais aussi des patrons et des anti-patrons. Toutefois, nos premiers travaux dans ce sens n'ont pas été concluants car notre modèle est assez complexe et nous manquons des connaissances nécessaires en intelligence artificielle. Nous sommes donc ouverts aux collaborations futures afin d'exploiter toutes ces données. Évidemment, nos données pourraient être exploitées à d'autres fins que la recherche de défauts de code. Comme nous l'avons déjà évoqué, le modèle de chaque application stockée dans la base de données est très complet. On pourrait imaginer l'exploitation pour obtenir des informations sur l'utilisation de librairies externes, l'évolution des architectures ou des métriques ou encore pour faire des analyses sémantiques sur les applications non obfusquées.

14.2.4 *Anti-patrons d'interaction avec des services distants*

Dans le cadre de cette thèse, nous avons aussi effectué des recherches sur les anti-patrons d'interaction entre applications mobiles et services

distants, en particulier avec des services infonuagiques afin de faire le lien avec nos travaux présentés en annexe A. Malheureusement, nous n'avons pas eu le temps de proposer une approche permettant la détection de ces derniers. Il s'avère que les applications mobiles interagissent souvent avec de tels services et que par conséquent il existe des bonnes et des mauvaises pratiques qui régissent ces interactions. Il serait donc nécessaire d'effectuer le même travail que ce que nous avons fait pour les défauts de code dans les applications mobiles, c'est-à-dire définir clairement les anti-patrons d'interactions mais aussi de permettre leur détection, leur correction et évaluer l'impact de la présence des anti-patrons sur les applications. Un nouvel étudiant de maîtrise à l'UQAM, Abdelkarim Belkhir devrait travailler sur ce sujet à partir de janvier 2016.

14.2.5 *Collaboration avec des professionnels*

Enfin d'une manière plus globale pour l'ensemble de nos recherches, il serait nécessaire de collaborer avec les professionnels pour avoir des retours sur la pertinence de nos résultats et l'intérêt de nos outils. Nous pourrions aussi ainsi comprendre les raisons de la présence des défauts de code et interpréter plus finement nos résultats, ce qui fait défaut pour le moment puisque nous ne disposons généralement pas du code source ni des avis des développeurs. De la même manière, ces retours pourraient servir à améliorer le processus de correction en collectant les retours des développeurs sur cet aspect. Cela pourrait aussi permettre d'améliorer les outils d'analyses de l'impact, en effet peut-être que d'autres métriques seraient plus pertinentes que celle que nous utilisons. L'utilisation d'un ampèremètre est aussi contraignante, d'autant que toutes les batteries de téléphone ne sont pas amovibles. Pour être moins intrusif et plus simple à utiliser par les professionnels, Hot-Pepper pourrait intégrer un modèle d'estimation de la consommation d'énergie à la place de mesures physiques. La mise en place de telles collaborations demande du temps afin de s'assurer que les professionnels soient vraiment impliqués dans le processus et il sera nécessaire de les convaincre que cela leur sera bénéfique sur le long terme.

Sixième partie

ANNEXE

A

IMPACT DES PATRONS INFONUAGIQUES SUR LA QUALITÉ DE SERVICE (QDS)

Dans le cadre de cette étude sur les patrons infonuagiques nous avons choisi d'étudier l'impact sur la qualité de service de trois patrons à la fois recommandés par la littérature scientifique et l'industrie. Nous présentons donc brièvement cette publication dans cette annexe.

A.1 PATRONS CHOISIS

Deux des patrons sont spécifiques aux base de données relationnelles. Le premier patron choisi est le *Local Database Proxy* [74] qui utilise la réplication et une gestion maître/esclave. Les requêtes en lecture sont orientées vers les bases de données esclaves tandis que les requêtes en écriture sont dirigées vers le maître. C'est un composant de l'application, le *proxy*, qui détermine le routage de la requête. Ce patron est conseillé lorsque l'application effectue un grand nombre de lectures et peu d'écritures dans la base de données. L'élasticité peut être gérée en augmentant le nombre de bases de données esclaves.

Le deuxième patron est le *Local Sharding-Based Router* [74] qui consiste à séparer les tables de la base de données en *shards* indépendants et à les répartir sur différentes instances de base de données. Ce *sharding* est habituellement effectué sur l'identifiant dans la table, par exemple les entrées 1 à 10 000 seront dans une base et les entrées 10 001 à 20 000 dans la base suivante. Ici un routeur local à l'application se charge d'effectuer le routage de la requête vers la base, que ce soit une lecture ou une écriture. Ce patron est conseillé lorsque beaucoup d'écritures sont nécessaires et qu'il est possible de séparer les données indépendamment en *shards*. En revanche, il n'est pas du tout adapté pour les requêtes qui feraient de l'agrégation sur toute une table. Il est possible de rajouter ou de supprimer des instances pour gérer l'élasticité mais cela demande des traitements supplémentaires afin d'équilibrer les *shards*.

Enfin, le dernier patron de cette étude est une file d'attente de messages avec priorité [39]. Comme vu dans l'état de l'art, ce patron est recommandé pour réduire le couplage entre composants et améliorer la flexibilité de l'application.

A.2 APPLICATION ANALYSÉE

Nous avons bâti notre propre application pour l'étude afin de pouvoir implémenter et combiner facilement les différents patrons. C'est une application distribuée de type client-serveur centrée sur la base de

données. Elle est écrite en Java et hébergée sur un serveur d'application Glassfish [1] et utilise des appels REST pour assurer la communication. Le système de gestion de base de données utilisé est MySQL [2] ce qui nous a permis d'utiliser la base de données exemple Sakila [3] qui contient 16 tables, 7 vues et environ 50 000 entrées. L'application et la base de données ont été déployées sur 10 machines virtuelles réparties sur trois serveurs, ceci afin de simuler les instances d'un réseau infonuagique. Il est à noter que notre application n'est pas vraiment déployée sur une infrastructure infonuagique, nous avons fait ce choix afin d'être certain de n'évaluer que les patrons et non pas une partie de l'infrastructure.

Enfin, l'application utilise plusieurs algorithmes pour les deux premiers patrons. Ces algorithmes et les différentes versions de l'application sont récapitulés dans le tableau 30, ces versions peuvent être combinées. Pour le *Local Database Proxy*, le choix de l'esclave où faire la lecture s'effectue soit aléatoirement, soit en *Round-Robin* (instance 1 puis 2 puis 3 en boucle) soit avec un algorithme plus réactif qui sélectionne la base de données ayant le temps de réponse le moins élevé et le moins de connexions actives.

Pour le *Local Sharding-Based Router*, c'est l'algorithme de *sharding* qui diffère [4]. La première méthode consiste à utiliser un simple modulo sur l'identifiant. Le *lookup* utilise un tableau (plus grand que le nombre d'instances) remplit aléatoirement avec un numéro d'instance de base de données et un modulo pour déterminer le choix de la case du tableau (et donc de l'instance). Enfin, le *consistent hashing* utilise les algorithmes de hachage MD5 ou SHA-1 ; pour chaque requête, on génère une clef de hachage utilisant l'identifiant de la requête et l'identifiant du *shard*, c'est le *shard* qui a la plus longue clef qui prend en charge la requête.

Tableau 30 – Différentes versions

Patron	Algorithme	Version
Basic Version		E0
Local Database Proxy	Random Allocation	E1
	Round-Robin	E2
	Custom Load Balancing	E3
Local Sharding-Based Router	Modulo Algorithm	E4
	Lookup Algorithm	E5
	Consistent Hashing	E6
Priority Message Queue		E7

1. https://glassfish.java.net/fr
2. http://www.mysql.fr
3. http://dev.mysql.com/doc/sakila/en
4. Plus de détails sur les algorithmes de sharding : http://kennethxu.blogspot.fr/2012/11/sharding-algorithm.html

A.3 EXPÉRIMENTATIONS

L'application a subi des tests de charge avec différents types de requêtes (lecture, écriture, agrégation) et une intensité variant de 1 à 10 000 requêtes. Nous avons utilisé le temps de réponse ainsi que le nombre moyen et maximum de requêtes traitées par minute comme métriques pour évaluer la QdS. Chaque test a été effectué 5 fois et les moyennes ont été utilisées comme résultats.

Pour déterminer si les patrons ont un impact réel ou non, nous avons utilisé des tests statistiques, à savoir le test non-paramétrique de *Mann-Whitney U* [73] et mesuré la taille d'effet avec *Cliff's* δ [70].

A.4 RÉSULTATS ET CONCLUSIONS

La figure 36 et la figure 37 sont des exemples de résultats collectés [5]. Ils illustrent parfaitement les conclusions de la publication. À savoir, les patrons améliorent généralement la QdS dans le cas où le contexte (lecture ou écriture) du patron est respecté mais la QdS peut être dégradée dans le cas contraire. Le choix du patron est plus important que le choix de l'algorithme et le choix de l'algorithme a un effet mineur sur la QdS. La combinaison de plusieurs patrons (file d'attente de messages et un patron de base de données par exemple) a aussi un effet non-négligeable et généralement positif sans qu'il soit possible de généraliser. Ces conclusions démontrent que les patrons ont une influence importante sur la QdS et cet aspect devrait donc être pris en compte lors de la conception d'une application.

5. L'ensemble des résultats collectés sont disponibles à l'adresse : http ://-goo.gl/B9upx8

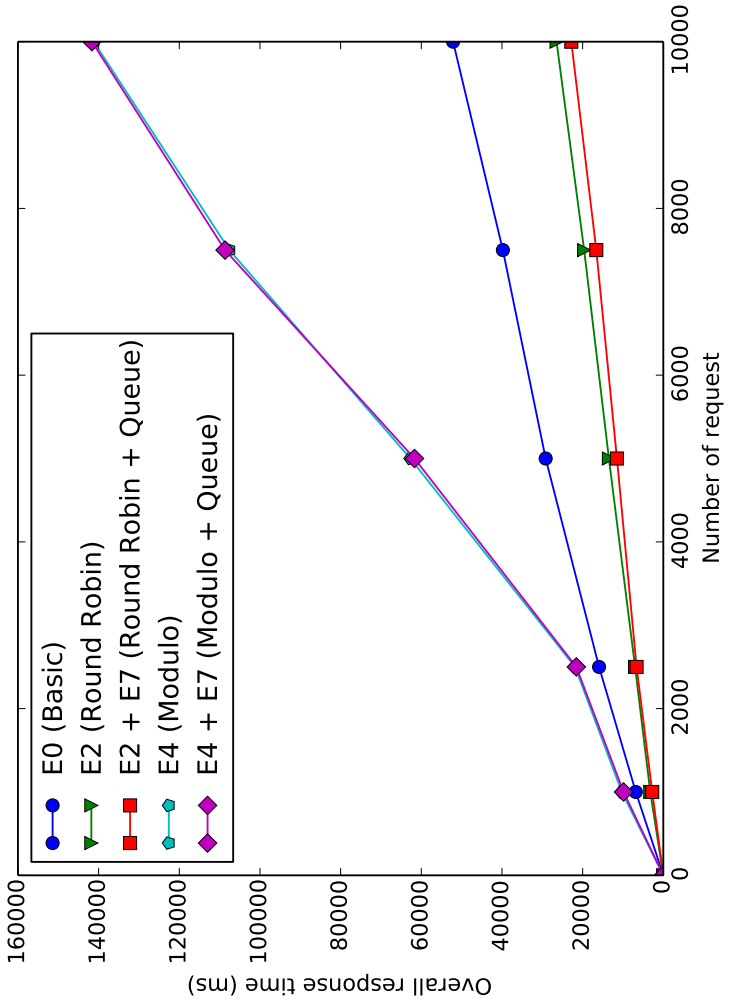


FIGURE 36 – Sélection aléatoire entre un film et inventaire d'un client

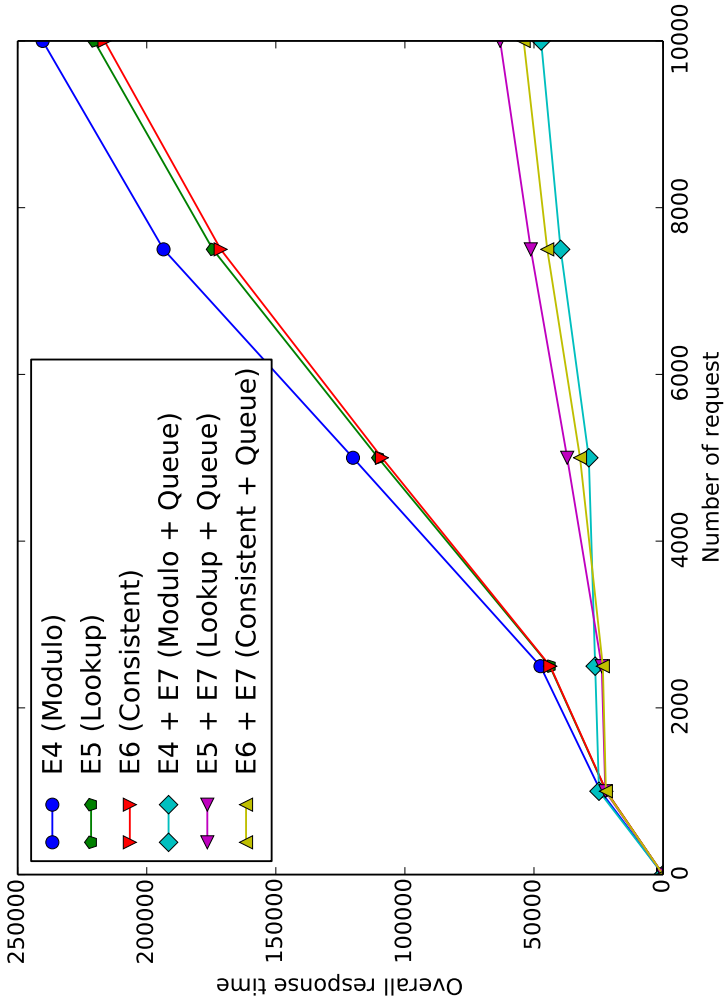


FIGURE 37 – Insertion d'un film avec Local Sharding-Based Router et Priority Queue

BIBLIOGRAPHIE

[1] *ART and Dalvik*. https://source.android.com/devices/tech/dalvik/. [En ligne, accès Septembre 2016]. 2015.

[2] Kevin ALLIX, Tegawendé F BISSYANDÉ, Jacques KLEIN et Yves LE TRAON. « AndroZoo : collecting millions of Android apps for the research community ». In : *Proceedings of the 13th International Workshop on Mining Software Repositories*. ACM. 2016, p. 468–471.

[3] *Android Lint - Android Tools Project Site*. [En ligne, accès Septembre 2016]. 2015. URL : http://tools.android.com/tips/lint.

[4] *Android Performance Tips*. http://developer.android.com/training/articles/perf-tips.html. [En ligne, accès Septembre 2016]. 2015.

[5] *ArrayMap*. http://developer.android.com/reference/android/support/v4/util/ArrayMap.html. [En ligne, accès Septembre 2016]. 2015.

[6] Niranjan BALASUBRAMANIAN, Aruna BALASUBRAMANIAN et Arun VENKATARAMANI. « Energy consumption in mobile phones : a measurement study and implications for network applications ». In : *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. 2009, p. 280–293.

[7] Abhijeet BANERJEE, Hai-Feng GUO et Abhik ROYCHOUDHURY. « Debugging Energy-efficiency Related Field Failures in Mobile Apps ». In : MobileSoft, 2016.

[8] Alexandre BARTEL, Jacques KLEIN, Yves LE TRAON et Martin MONPERRUS. « Dexpler : converting android dalvik bytecode to jimple for static analysis with soot ». In : *Proc. of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. ACM. 2012, p. 27–38.

[9] William J BROWN, Hays W MCCORMICK, Thomas J MOWBRAY et Raphael C MALVEAU. *AntiPatterns : refactoring software, architectures, and projects in crisis*. 1. Auflage. Wiley New York, 1998.

[10] Martin BRYLSKI. *Android Smells Catalogue*. http://www.modelrefactoring.org/smell_catalog. [En ligne, accès Septembre 2016]. 2013.

[11] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha et Romain Rouvoy. « Investigating the Energy Impact of Android Smells ». In : *IEEE 24rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2017.

[12] Christopher Chambers et Christopher Scaffidi. « Smell-driven performance analysis for end-user programmers ». In : *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE. 2013, p. 159–166.

[13] Shyam R Chidamber et Chris F Kemerer. « A metrics suite for object oriented design ». In : *IEEE Transactions on Software Engineering* 20.6 (1994), p. 476–493.

[14] Pablo Cingolani et Jesús Alcalá-Fdez. « jFuzzyLogic : a Java Library to Design Fuzzy Logic Controllers According to the Standard for Fuzzy Control Programming ». In : (2013), p. 61–75.

[15] Norman Cliff. « Dominance statistics : Ordinal analyses to answer ordinal questions. » In : *Psychological Bulletin* 114.3 (1993), p. 494.

[16] Jacob Cohen. « A power primer. » In : *Psychological bulletin* 112.1 (1992), p. 155.

[17] TD Cook et DT Campbell. *Quasi-experimentation : Design & analysis issues for field settings*. Houghton Mifflin Company, 1979. URL : http://dickyh.staff.ugm.ac.id/wp/wp-content/uploads/2009/ringkasanbukuquasi-experimentakhir.pdf.

[18] Julien Duribreux, Romain Rouvoy et Martin Monperrus. *An Energy-efficient Location Provider for Daily Trips*. Rapp. tech. 2014.

[19] Norman E. Fenton et Shari Lawrence Pfleeger. *Software Metrics : A Rigorous and Practical Approach*. 2nd. Boston, MA, USA : PWS Publishing Co., 1998. ISBN : 0534954251.

[20] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia et Alexander Chatzigeorgiou. « JDeodorant : identification and application of extract class refactorings ». In : *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, p. 1037–1039.

[21] Martin Fowler. *Refactoring : improving the design of existing code*. Pearson Education India, 1999.

[22] Martin Fowler, Kent Beck, John Brant, William Opdyke et Don Roberts. *Refactoring : Improving the Design of Existing Code*. Addison Wesley, 1999, p. 1–431. ISBN : 9780201485677. DOI : 10.1007/s10071-009-0219-y.

[23] Félix García, Manuel F. Bertoa, Coral Calero, Antonio Vallecillo, Francisco Ruíz, Mario Piattini et Marcela Genero. « Towards a Consistent Terminology for Software Measurement ». In : *Information and Software Technology* 48 (2006), p. 631–644.

[24] M Gjoshevski et T Schweighofer. « Small Scale Analysis of Source Code Quality with regard to Native Android Mobile Applications ». In : *4th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications*. Maribor, Slovenia : CEUR Vol. 1375, 2015, p. 2–10. URL : http://ceur-ws.org/Vol-1375/SQAMIA2015_Paper2.pdf.

[25] Marion Gottschalk, Mirco Josefiok, Jan Jelschen et Andreas Winter. « Removing Energy Code Smells with Reengineering Services ». In : *42nd Annual Meeting of the Society for computer science (GI)*. T. 208. Bonn, Germany, 2012, p. 441–455.

[26] Marion Gottschalk, Mirco Josefiok, Jan Jelschen et Andreas Winter. « Removing Energy Code Smells with Reengineering Services ». In : *GI-Jahrestagung*. Sous la dir. d'Ursula Goltz, Marcus A. Magnor, Hans-Jürgen Appelrath, Herbert K. Matthies, Wolf-Tilo Balke et Lars C. Wolf. T. 208. 2012, p. 441–455.

[27] C Goutte et E Gaussier. « A probabilistic interpretation of precision, recall and F-score, with implication for evaluation ». In : *Advances in information retrieval* (2005). URL : http://link.springer.com/chapter/10.1007/978-3-540-31865-1{_}25.

[28] Jiaping Gui, Ding Li et William G.J. Wan Mianand Halfond. « Lightweight Measurement and Estimation of Mobile Ad Energy Consumption ». In : *Proceedings of the 5th International Workshop on Green and Sustainable Software – GREENS*. 2016.

[29] Chet Haase. *Developing for Android, I : Understanding the Mobile Context*. https://goo.gl/KUN6XC. [En ligne, accès Septembre 2016]. 2015.

[30] Chet Haase. *Developing for Android, II The Rules : Memory*. https://medium.com/google-developers/developing-for-android-ii-bb9a51f8c8b9. [En ligne, accès Septembre 2016]. 2015.

[31] Geoffrey Hecht. « An approach to detect Android antipatterns ». In : *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. T. 2. IEEE. 2015, p. 766–768.

[32] Geoffrey Hecht, Naouel Moha et Romain Rouvoy. « An Empirical Study of the Performance Impacts of Android Code Smells ». In : *IEEE/ACM International Conference on Mobile Software Engineering and Systems*. T. 1. 1. IEEE. 2016.

[33] Geoffrey Hecht, Benjamin Jose-Scheidt, Clément De Figueiredo, Naouel Moha et Foutse Khomh. « An empirical study of the impact of cloud patterns on quality of service (qos) ». In : *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*. IEEE. 2014, p. 278–283.

[34] Geoffrey Hecht, Laurence Duchien, Naouel Moha et Romain Rouvoy. « Detection of Anti-patterns in Mobile Applications ». In : *COMPARCH 2014*. 2014.

[35] Geoffrey Hecht, Romain Rouvoy, Naouel Moha et Laurence Duchien. *Detecting Antipatterns in Android Apps*. Research Report RR-8693. INRIA Lille ; INRIA, mar. 2015. URL : https://hal.inria.fr/hal-01122754.

[36] Geoffrey Hecht, Benomar Omar, Romain Rouvoy, Naouel Moha et Laurence Duchien. « Tracking the Software Quality of Android Applications along their Evolution ». In : *30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2015, p. 12.

[37] Geoffrey Hecht, Naouel Moha, Romain Rouvoy et Javier Gonzalez-Huerta. « An Empirical Analysis of Android Code Smells ». In : *En cours de révision pour le journal Transactions on Software Engineering (TSE) - sous réserve d'acceptation*. 2016.

[38] Florian Holzschuher et René Peinl. « Performance of graph query languages : comparison of cypher, gremlin and native access in neo4j ». In : *Proc. of the Joint EDBT/ICDT 2013 Workshops*. ACM. 2013, p. 195–204.

[39] Alex Homer, John Sharp, Larry Brader, Masashi Narumoto Narumoto et Trent Swanson. *Cloud Design Patterns Prescriptive Architecture Guidance for Cloud Applications (Microsoft patterns practices)*. Microsoft patterns practices, 2014, p. 109–115.

[40] *Investigating Your RAM Usage*. http://developer.android.com/tools/debugging/debugging-memory.html. [En ligne, accès Septembre 2016]. 2015.

[41] *Java platform, Micro Edition (Java ME)*. http://www.oracle.com/technetwork/java/embedded/javame/index.html. [En ligne, accès Septembre 2016]. 2015.

[42] Jonah Kowall. *Improve Quality Mobile Application Delivery With Mobile Application Performance Monitoring*. 2013.

[43] Ding Li et William GJ Halfond. « An investigation into energy-saving programming practices for android smartphone app development ». In : *Proceedings of the 3rd International Workshop on Green and Sustainable Software*. ACM. 2014, p. 46–53.

[44] Ding Li, Yingjun Lyu, Jiaping Gui et William G.J. Halfond. « Automated Energy Optimization of HTTP Requests for Mobile Applications ». In : *Proceedings of the 38th International Conference on Software Engineering – ICSE*. A paraitre. 2016.

[45] Mario Linares-Vásquez, Sam Klock, Collin McMillan, Aminata Sabané, Denys Poshyvanyk et Yann-Gaël Guéhéneuc. « Domain matters : bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps ». In : *Proc. of the 22nd International Conference on Program Comprehension*. ACM. 2014, p. 232–243.

[46] Yepang Liu, Chang Xu et Shing-Chi Cheung. « Characterizing and detecting performance bugs for smartphone applications ». In : *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, p. 1013–1024.

[47] Alex Lockwood. *How to Leak a Context : Handlers and Inner Classes*. http://www.androiddesignpatterns.com/2013/01/inner-class-handler-memory-leak.html. [En ligne, accès Septembre 2016]. 2013.

[48] *LruCache*. https://developer.android.com/reference/android/util/LruCache.html. [En ligne, accès Septembre 2016]. 2015.

[49] Umme Ayda Mannan, Iftekhar Ahmed, Rana Abdullah M Almurshed, Danny Dig et Carlos Jensen. « Understanding code smells in Android applications ». In : *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. ACM. 2016, p. 225–234.

[50] M. Mantyla, J. Vanhanen et C. Lassenius. « A taxonomy and an initial empirical study of bad smells in code ». In : *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings*. IEEE Comput. Soc, 2003, p. 381–384. isbn : 0-7695-1905-9. doi : 10.1109/ICSM.2003.1235447. url : http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1235447.

[51] Gabriele Mariotti. *AntiPattern : freezing a UI with Broadcast Receiver*. http://gmariotti.blogspot.ca/2013/02/antipattern-freezing-ui-with-asynctask.html. [En ligne, accès Septembre 2016]. 2013.

[52] Gabriele Mariotti. *AntiPattern : freezing a UI with Broadcast Receiver*. http://gmariotti.blogspot.ca/2013/02/antipattern-freezing-ui-with-broadcast.html. [En ligne, accès Septembre 2016]. 2013.

[53] Gabriele Mariotti. *Antipattern : freezing the UI with a Service and an IntentService*. http://gmariotti.blogspot.ca/2013/03/antipattern-freezing-ui-with-service.html. [En ligne, accès Septembre 2016]. 2013.

[54] K Maxwell. *Applied statistics for software managers*. Prentice Hall, 2002. URL : http://scholar.google.es/scholar?q=applied+statistics+for+softwaremanagers{\&}btnG={\&}hl=en{\&}as{_}sdt=0,5{\#}1.

[55] Colt McAnlis. *Android Performance Patterns : Overdraw, Cliprect, QuickReject*. https://youtu.be/vkTn3Ule4Ps. [En ligne, accès Septembre 2016]. Google Developers, 2015.

[56] Colt McAnlis. *Avoiding Allocations in onDraw() (100 Days of Google Dev)*. https://youtu.be/HAK5acHQ53E. [En ligne, accès Septembre 2016]. Google Developers, 2015.

[57] Colt McAnlis. *The Magic of LRU Cache (100 Days of Google Dev)*. https://youtu.be/R5ON3iwx78M. [En ligne, accès Septembre 2016]. Google Developers, 2015.

[58] Thomas J McCabe. « A complexity measure ». In : *IEEE Transactions on Software Engineering* 4 (1976), p. 308–320.

[59] Stuart McIlroy, Nasir Ali et Ahmed E Hassan. « Fresh apps : an empirical study of frequently-updated mobile apps in the Google play store ». In : *Empirical Software Engineering* 21.3 (2016), p. 1346–1370.

[60] Roberto Minelli et Michele Lanza. « Software Analytics for Mobile Applications–Insights and Lessons Learned ». In : *17th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE. 2013, p. 144–153.

[61] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien et A Le Meur. « DECOR : A method for the specification and detection of code and design smells ». In : *Software Engineering, IEEE Transactions on* 36.1 (2010), p. 20–36.

[62] Sona Mundody et Sudarshan. K. « Evaluating the Impact of Android Best Practices on Energy Consumption ». In : *IJCA Proceedings on International Conference on Information and Communication Technologies* ICICT.8 (2014). Full text available, p. 1–4.

[63] Ian Ni-Lewis. *Custom Views and Performance (100 Days of Google Dev)*. https://youtu.be/zK2i7ivzK7M. [En ligne, accès Septembre 2016]. Google Developers, 2015.

[64] *PMD*. http://pmd.sourceforge.net/. [En ligne, accès Septembre 2016]. 2015.

[65] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera et Lionel Seinturier. « Spoon : A Library for Implementing Analyses and Transformations of Java Source Code ». In : *Software : Practice and Experience* (2015), na. doi : 10.1002/spe.2346. url : https://hal.archives-ouvertes.fr/hal-01078532/document.

[66] Ricardo Pérez-Castillo et Mario Piattini. « Analysing the Harmfull Effect of God Class Refactoring on Power Consumption ». In : *IEEE Software*. T. 31. IEEE, 2014, p. 48–54.

[67] Jan Reimann, Martin Brylski et Uwe Assmann. « A Tool-Supported Quality Smell Catalogue For Android Developers ». In : *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*. T. 2014. 2014.

[68] Jan Reimann, Martin Brylski et Uwe Assmann. « A Tool-Supported Quality Smell Catalogue For Android Developers ». In : *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung – MMSM 2014*. 2014.

[69] Ana Rodriguez, Mathias Longo et Alejandro Zunino. « Using bad smell-driven code refactorings in mobile applications to reduce battery usage ». In : *16º Simposio Argentino de Ingeniería de Software*. Rosario, Argentina, 2015, p. 56–68.

[70] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio et Jeff Skowronek. « Appropriate statistics for ordinal level data : Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys ». In : *annual meeting of the Florida Association of Institutional Research*. 2006, p. 1–33.

[71] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams et Ahmed E Hassan. « Understanding reuse in the android market ». In : *20th International Conference on Program Comprehension (ICPC)*. IEEE. 2012, p. 113–122.

[72] Ġsmaġl Alper Sağlam. « Measuring And Assesment Of Well Known Bad Pratices In Android Application Developments ». Thèse de doct. Middle East Technical University, 2014.

[73] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

[74] Steve Strauch, Vasilios Andrikopoulos, Uwe Breitenbuecher, Oliver Kopp et Frank Leyrnann. « Non-functional data layer patterns for Cloud applications ». In : *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. IEEE. 2012, p. 601–605.

[75] Student. « The probable error of a mean ». In : *Biometrika* (1908), p. 1–25.

[76] Girish Suryanarayana, Ganesh Samarthyam et Tushar Sharma. *Refactoring for Software Design Smells : Managing Technical Debt*. T. 11. Elsevier Science, 2014, p. 258. isbn : 0128016469. url : https://books.google.com/books?id=1SaOAwAAQBAJ\&pgis=1.

[77] *Testing Display Performance*. http://developer.android.com/training/testing/performance.html. [En ligne, accès Septembre 2016]. 2015.

[78] Aline Rodrigues Tonini, Leonardo Matthis Fischer, Julio Carlos Balzano de Mattos et Lisane Brisolara de Brisolara. « Analysis and evaluation of the Android best practices impact on the efficiency of mobile applications ». In : *2013 III Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE. 2013, p. 157–158.

[79] Nikolaos Tsantalis, Theodoros Chaikalis et Alexander Chatzigeorgiou. « JDeodorant : Identification and removal of type-checking bad smells ». In : *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE. 2008, p. 329–331.

[80] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia et Denys Poshyvanyk. « When and why your code starts to smell bad ». In : *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, p. 403–414.

[81] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[82] *Using the Emulator*. http://developer.android.com/tools/devices/emulator.html. [En ligne, accès Septembre 2016]. 2015.

[83] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam et Vijay Sundaresan. « Soot-a Java bytecode optimization framework ». In : *Proc. of the conference of the Centre for Advanced Studies on Collaborative research*. IBM Press. 1999, p. 13.

[84] D Verloop. « Code Smells in the Mobile Applications Domain ». Thèse de doct. TU Delft, Delft University of Technology, 2013.

[85] Antonio Vetro, Luca Ardito, Giuseppe Procaccianti et Maurizio Morisio. « Definition, implementation and validation of energy code smells : an exploratory study on an embedded system ». In : (2013).

[86] WC Wake. *Refactoring workbook*. Addison Wesley Longman Publishing Co., Inc., 2003. url : https://books.google.es/books?hl=en{\&}lr={\&}id=KE-zdnHQehYC{\&}oi=fnd{\&}pg=PR13{\&}dq=Refactoring+Workbook{\&}ots=nDwpOuHVOE{\&}sig=EdsSaf{_}nL3WpHnXeQNmyPyg3-Lc.

[87] *What optimizations can I expect from Dalvik and the Android toolchain ?* http://stackoverflow.com/a/4930538. [En ligne, accès Septembre 2016]. 2011.

[88] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell et Anders Wesslén. *Experimentation in software engineering*. Springer, 2012.

[89] Liang Xu. « Techniques and Tools for Analyzing and Understanding Android Applications ». Thèse de doct. University of California Davis, 2013.

[90] Cypher. http://neo4j.com/developer/cypher-query-language. [En ligne, accès Septembre 2016]. 2015.

[91] Neo4j. http://neo4j.com. [En ligne, accès Septembre 2016]. 2015.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX :

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here :

http://postcards.miede.de/

Final Version as of 16 décembre 2016 (`classicthesis` version 1.13).