UNIVERSITÉ DU QUÉBEC À MONTRÉAL

FORAGE D'APPLICATIONS MOBILES
À LA DÉCOUVERTE DE DÉFAUTS DE CODE

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

RUBIN JÉHAN

AVRIL 2019

REMERCIEMENTS

Je souhaite tout d'abord remercier ma directrice de recherche, Naouel Moha, pour son temps, son aide ainsi que sa patience tout au long de ma maîtrise. Mes remerciements vont aussi à mon co-directeur, Mohamed Bouguessa, pour son aide et les conseils qu'il m'a prodigués durant mon travail.

Je remercie tout particulièrement Ellen qui m'a soutenu durant ces deux dernières années. Elle a su me conseiller et me pousser lors des moments difficiles. Sa cuisine m'a permis de manger autre chose que des pâtes, et son dégoût pour la vaisselle m'a rendu plus performant qu'un lave-vaisselle industriel.

Je remercie aussi les membres du Therland : Titi, Justine, Adriz, Hélène, Dodo, Damien, Chachoo, Tone, Pépé, Fixlard, Clément, Sebou et Julien pour votre présence et votre amitié depuis le lycée qui m'ont permis d'avancer lors des moments difficiles (notamment en fin de soirée).

Je remercie aussi mes copains de Montréal : Vincent, Lam, Mathilde, Agathe, Val, Florian, Quentin, Mélanie, Simon, Andrea, Karim, Sara et Dylan. Merci pour ces moments de détente, pour ces bières, ces barbecues, ces couscous et ces bonnes tranches de rigolade durant ces deux dernières années.

Je tiens à remercier Alexandre qui m'a beaucoup aidé pour l'implémentation technique de la méthode durant son stage.

Enfin, je remercie mes parents, à qui je dédie ce travail pour leur soutien sans faille et leurs conseils avisés. Sans eux je ne serais pas arrivé ou j'en suis aujourd'hui.

Je remercie aussi ma grande soeur, mon petit frère et ma petite soeur sans qui je ne serais pas ce que je suis aujourd'hui.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

L'intérêt pour les périphériques mobiles a considérablement augmenté ces dernières années. Les applications fonctionnant sur ces périphériques sont devenues une part importante du marché du développement de logiciels. La demande de nouvelles applications et de nouvelles fonctionnalités entraîne une complexité accrue des applications mobiles ainsi qu'un temps de développement en constante diminution. Ces contraintes peuvent amener les développeurs à faire de mauvais choix en matière de conception et de mise en œuvre, appelés défauts de code. Les défauts de code dans les applications mobiles entraînent des problèmes de performance tel qu'une surconsommation de ressources matérielles (processeur, mémoire vive, batterie). Certains outils ont été proposés pour la détection des défauts de code dans les applications Android tels que PAPRIKA ou ADOCTOR. Ces outils reposent sur des règles de détection utilisant des métriques. Ces règles sont définies manuellement en fonction de la compréhension des définitions des défauts de code. Cependant, les règles définies manuellement peuvent être inexactes et subjectives, car elles sont basées sur l'interprétation du concepteur. Dans ce mémoire, nous proposons une approche outillée, appelée FAKIE, permettant l'inférence automatique des règles de détection en analysant les données relatives aux défauts de code. Pour ce faire, nous utilisons un algorithme d'extraction de règles d'association : FP-GROWTH. Une fois les règles extraites nous appliquons des filtres afin de sélectionner les règles les plus pertinentes dans le cadre d'une détection de défauts de code. Nous avons validé FAKIE en l'appliquant à un ensemble de données de validation de 30 applications mobiles à code source ouvert. Nous avons pu générer des règles de détection pour une douzaine de défauts de code avec une F-mesure moyenne de 0,95. Après cela, nous avons effectué une étude empirique en appliquant FAKIE sur 2,993 applications téléchargées sur ANDROZOO, un dépôt d'applications mobiles. Notre étude a montré que les règles d'associations permettent d'analyser les défauts de code et d'en extraire des informations intéressantes. Par exemple, l'une des règles extraites indique que le défaut de code *BLOB* est principalement contenu dans les activités Android. Nous observons également que 58% des défauts de code *HMU* contiennent un grand nombre d'instructions, ce qui signifie que les défauts de code *LM* et *HMU* co-occurrent régulièrement ensemble.

MOTS CLES : Android, défauts de code, détection, règles d'association, applications mobiles.

INTRODUCTION

Le périphérique mobile est devenu ces dernières années un des périphériques les plus utilisés. Avec plus de 1,500 millions de ventes en 2017 [1], la vente de téléphone mobiles a subi une augmentation de 500% [2]. Malgré cela, cette technologie reste une technologie très récente. Le premier téléphone mobile nouvelle génération est présenté par la firme Apple en 2007. Apple dévoile alors un périphérique mobile nouvelle génération intégrant un GPS, un appareil photo, de multiples connexions réseaux et un écran tactile : l'iPhone.

La même année, l'entreprise Google met en avant son système d'exploitation Android racheté en 2005. Android est un système d'exploitation destiné aux périphériques mobiles. Il est basé sur un noyau Linux et autorise ainsi une certaine liberté que le système d'exploitation d'Apple ne permet pas. Android est devenu de plus en plus populaire, au point qu'il domine aujourd'hui le marché de la téléphonie mobile avec une part de marché de plus de 85%. Notre travail porte sur le système d'exploitation Android puisque celui-ci est le système le plus utilisé.

La popularité des périphériques mobiles augmentant, les applications mobiles gagnent aussi considérablement en importance dans le marché logiciel. Dû à une forte augmentation de la demande, le temps de développement accordé à chaque

---

1. (En ligne; Accès Janvier-2018). Statista. `https://fr.statista.com/statistiques/565015`.

2. (En ligne; Accès Janvier-2018). Statista. `https://fr.statista.com/statistiques/565015`.

application mobile est en baisse malgré l'augmentation de leur complexité. Ces facteurs peuvent provoquer de mauvais choix de conception lors du développement. Ces mauvais choix sont connus sous le nom de défaut de code (Fowler *et al.*, 1999). Ils entraînent une augmentation de la complexité d'un système (Xie *et al.*, 2009) et affectent la maintenance logicielle (Yamashita et Moonen, 2012; Yamashita et Moonen, 2013). Ainsi, ils rendent difficile l'évolution d'un système. Plus spécifiquement pour les périphériques mobiles, les défauts de code dans les applications mobiles causent des problèmes de performance tels que la surconsommation de ressources (processeur, mémoire vive, batterie) ou encore des blocages de l'application. Les périphériques mobiles possèdent des composants beaucoup moins performant qu'un ordinateur classique ainsi qu'une batterie limité. Il est important d'optimiser la longévité de la batterie du périphérique. Éviter la surconsommation des ressources est donc impératif.

En corrélation avec le succès du système, l'intérêt concernant les défauts de code spécifiques à Android a augmenté. Certaines tentatives pour la détection de défauts de code se sont faites à l'aide d'outils conçus à la base pour les systèmes orientés objet classiques (Linares-Vásquez *et al.*, 2013; Verloop, 2013). Les applications mobiles sont développées généralement à l'aide de langages orientés objet classiques telles que Java, C# ou encore Objective-C. Cependant, le développement mobile est différent du développement traditionnel (Wasserman, 2010). La plate-forme Android possède des limitations et des contraintes supplémentaires comparée aux systèmes orientés objet classiques fonctionnant sur ordinateur. Ces contraintes peuvent concerner la mémoire, la puissance du processeur, la batterie ou encore la taille de l'écran. De ce fait, les applications Android possèdent des défauts de code spécifiques en plus des défauts de code orientés objet classiques.

(Reimann *et al.*, 2014) ont élaboré un catalogue présentant 30 défauts de code spécifiques à Android. À ce jour, il n'y a que peu d'outils de détection spécialisés

dans la détection de défauts de code spécifiques à la plate-forme Android tels que PAPRIKA (Hecht *et al.*, 2015) ou ADOCTOR (Palomba *et al.*, 2017a). Ces méthodes de détection sont des approches basées sur l'utilisation de règles de détection définies manuellement. Elles se basent sur les définitions des différents défauts de code (Reimann *et al.*, 2014) afin d'élaborer leurs règles de détection. Cependant, l'utilisation de règles manuelles engendre potentiellement plusieurs limitations.

La première limitation est en rapport avec le choix des composants de la règle de détection, en particulier dans le choix des métriques impliquées dans la présence d'un défaut de code. L'interprétation subjective peut entraîner l'oubli d'un élément dans la règle de détection. La deuxième limitation est liée à la difficulté de faire la bonne combinaison des métriques et de leurs valeurs qui, encore une fois, est déterminée par l'interprétation de la personne créant la règle de détection. Par exemple, à partir de quelle valeur une métrique doit elle être considérée comme élevée ? Cette information dépend totalement de la personne qui le juge. Enfin, la dernière limitation est la sélection des relations entre entités d'un programme à inclure dans la règle. Pour conclure, les règles de détections devraient être les plus objectives possibles afin d'obtenir une détection la plus efficace et précise.

Le but de notre travail vise à atteindre les objectifs suivants :

— Proposer une approche automatique pour la création de règle de détection pour les défauts de code dans les applications mobiles, et en diminuer la subjectivité ;
— Appliquer les règles de détection générées afin d'identifier et d'étudier les occurrences de défauts de code dans les applications mobiles ;

À travers ce mémoire, nous présentons les différentes études et méthodes utilisées pour la détection de défauts de code dans les applications mobiles. Le but final étant de proposer une approche permettant de générer des règles de détection de manière automatique. Les contributions apportées par notre travail dans ce mémoire sont les suivantes :

— L'approche outillée FAKIE permettant de générer automatiquement des règles de détection en les inférant à partir d'exemples de défauts de code ;
— Une étude empirique utilisant notre approche FAKIE portant sur 2,993 applications Android afin d'analyser des occurrences de défauts de code ;

Ce mémoire est structuré en quatre chapitres. Le chapitre I présente les concepts préliminaires permettant une bonne compréhension du travail. Le chapitre II met en avant la revue de littérature réalisée pour effectuer cette recherche. Le chapitre III introduit FAKIE, notre méthode permettant de générer des règles de détection. Pour finir, le chapitre IV décrit les études réalisées durant ce travail ainsi qu'une discussion concernant les résultats obtenus. Une synthèse du travail ainsi que la présentation des futurs travaux concluent ce mémoire.

CHAPITRE I

CONCEPTS PRÉLIMINAIRES

Ce chapitre explique les différents concepts essentiels à la bonne compréhension du travail présenté dans ce mémoire. Ce travail couvre différents domaines tels que le développement d'applications dans les systèmes Android, la qualité logicielle ainsi que l'apprentissage automatique.

1.1 Le système d'exploitation Android

Android est un système d'exploitation basé sur un noyau Linux pour les appareils mobiles. Il fonctionne aussi bien pour les téléphones intelligents que pour les tablettes. Ce système d'exploitation a été créé en 2003 par Andy Rubin, Rich Miner, Nick Sears et Chris White. Il est détenu par Google et une version bêta du système sort le 12 novembre 2007. Le kit de développement Android (SDK) est entièrement en Java.

L'architecture Android est constituée d'une pile de cinq composants logiciels principaux. La figure 1.1 représente les six différents composants logiciels répartis en cinq couches. Elle se lit du bas vers le haut, le noyau Linux étant le composant le plus proche du matériel.

1. **Noyau Linux :** c'est la fondation de l'architecture Android.

2. **La couche d'abstraction matérielle :** composant qui fournit différentes interfaces. Ces interfaces permettent de gérer les traitements entre l'interface de programmation (API) Java et les composants matériels.
3. **Le moteur d'exécution d'Android :** chaque application fonctionnant sur le système Android possède sa propre instance du moteur d'exécution. Tout comme en Java, les applications Android fonctionnent sur une machine virtuelle, à la différence que sur le système Android chaque application possède sa propre instance de machine virtuelle.
4. **Bibliothèques logicielles :** certains composants logiciels sont développés en C et C++, ce qui permet de développer des applications ou des librairies en utilisant ces mêmes langages de programmation. Ces bibliothèques permettent d'utiliser des technologies telles qu'OpenGL, une bibliothèque permettant de faire des graphismes 2D et 3D. Il existe de ce fait une interface de programmation d'applications Java OpenGL permettant d'utiliser cette bibliothèque dans le code natif d'une application.
5. **L'interface de programmation d'applications en Java :** toutes les fonctionnalités du système d'exploitation Android sont disponibles grâce à l'API écrite en Java. Cette API permet de développer une application Android. Celle-ci est composée de plusieurs parties, à savoir :
   - Un système de vue permettant de créer une interface graphique ;
   - Un gestionnaire de ressources ;
   - Un gestionnaire de notifications ;
   - Un gestionnaire d'activités qui gère le cycle de vie d'une application ;
   - Un fournisseur de contenu qui permet aux applications d'accéder à des données fournies par d'autres applications ;
6. **Les applications système :** cette couche contient les applications système qui sont les applications natives de la plate-forme.

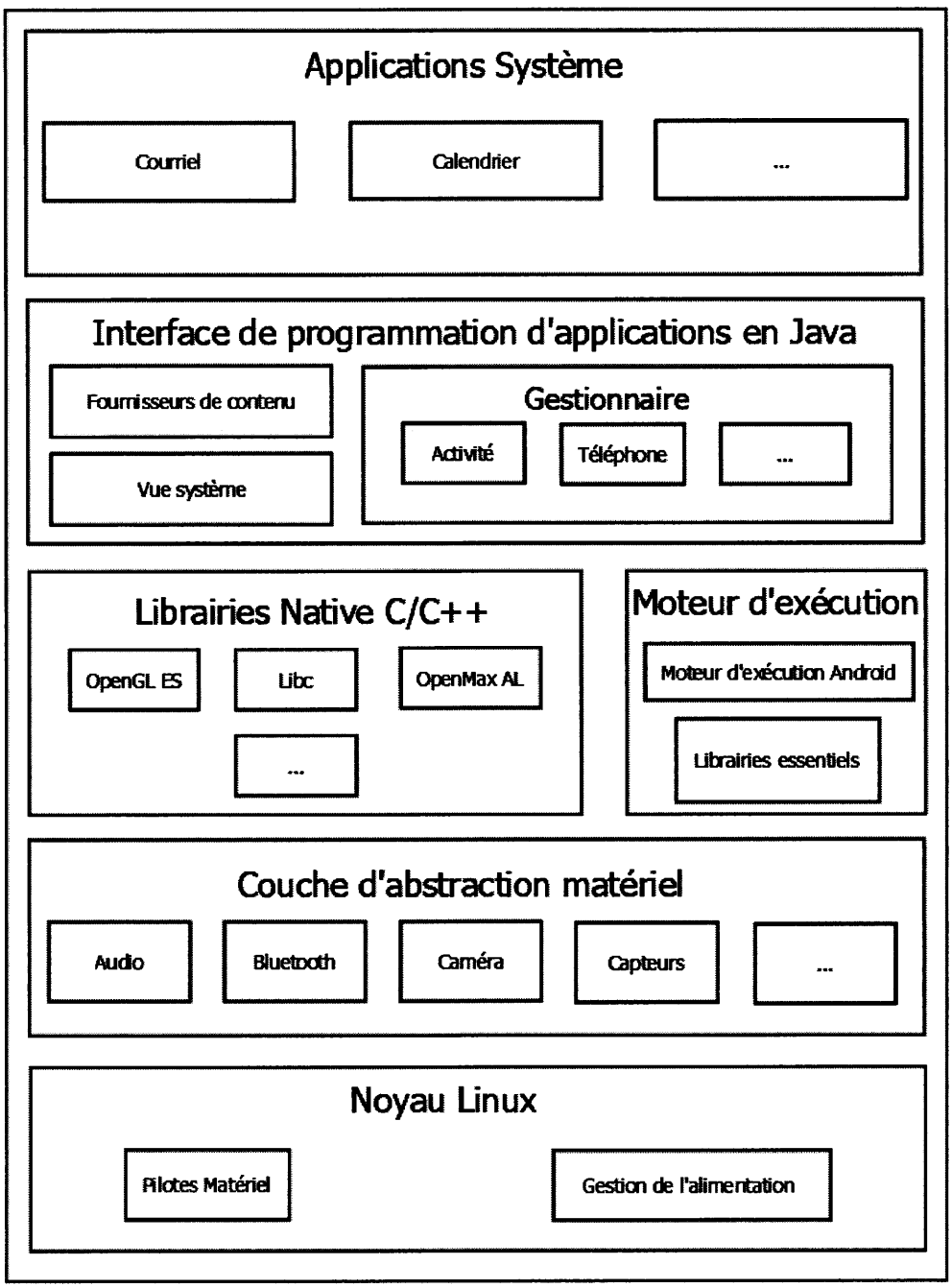

Figure 1.1 Architecture du système d'exploitation Android. Source : `https://developer.android.com/guide/platform/`

1.2 Les applications Android

Une application Android est représentée, une fois compilée, sous la forme d'une archive : c'est l'APK (Android Package Kit) d'une application. Cette archive est composée de plusieurs fichiers, à savoir :

- **Le manifeste :** ce fichier décrit les informations essentielles de l'application. Il est possible d'y trouver le nom des paquets, les composants de l'application (activité, services, etc.) ou encore les permissions requises pour certaines fonctionnalités comme l'accès à l'appareil photo.
- **Les librairies :** nous retrouvons dans l'APK les librairies utilisées pour le développement et le bon fonctionnement de l'application.
- **Le code binaire de l'application :** c'est le code source de l'application une fois que celui-ci est compilé.
- **Les ressources :** ce sont les images et les informations contenues dans des fichiers XML. Concrètement, les ressources sont tous les fichiers qui ne sont pas du code source.

Une application Android est constituée de différents composants : les activités, les services, les récepteurs de diffusion ou encore les fournisseurs de contenu. Une activité est concrètement une vue de l'application. Par exemple, pour une application de gestion de courriel, une activité servira à la liste des courriels reçus et une autre servira à la lecture du contenu. Lorsqu'il y a plus d'une activité, une d'entre elles doit être définie comme l'activité principale, autrement dit celle qui sera affichée lors du lancement de l'application. Pour finir, Android utilise des vues qui correspondent à l'aspect graphique de l'application. Une vue est un fichier XML faisant référence à différents *layout* (mise en page) permettant de définir l'aspect graphique.

Les activités gèrent le cycle de vie de l'application. Chaque activité possède son propre cycle de vie. La figure 1.2 montre les états que peuvent prendre les activités Android. Lorsqu'une nouvelle activité commence, elle est placée en haut de la pile d'exécution de toutes les activités présentes au même moment sur le système. Elle devient l'activité principale est sera celle présentement exécutée. Une activité possède plusieurs fonctions permettant de gérer son cycle de vie :

- **onCreate()** : cette fonction est appelée lorsqu'une activité est créée.
- **onStart()** : l'appel de cette fonction est fait lorsque l'activité en question peut être visible pour l'utilisateur sur l'écran.
- **onRestart()** : cette fonction est appelée si l'activité a été stoppée et doit reprendre son exécution.
- **onResume()** : l'utilisation de cette fonction est lorsque l'activité commence à interagir avec l'utilisateur courant.
- **onPause()** : lorsque le système d'exploitation désire pivoter sur l'exécution d'une autre activité, la fonction onPause() est appelée sur l'activité courante. Elle permet de réduire la consommation en ressources de l'activité courante et de sauvegarder les données courantes.
- **onStop()** : lorsqu'une activité n'est plus visible à l'écran (réutilisation d'une autre activité, ou création d'une nouvelle), cette fonction est appelée.
- **onDestroy()** : une fois qu'une activité est en "onStop()", elle peut être détruite par cette fonction. Ceci peut être fait automatiquement par le système d'exploitation ou indiqué par l'utilisateur afin de récupérer des ressources.

Comme cité ci-dessus, une application est aussi composée de services. Ces composants s'exécutent en arrière-plan de l'application et leur but est d'effectuer des opérations sur le long terme. Les récepteurs de diffusion permettent de répondre

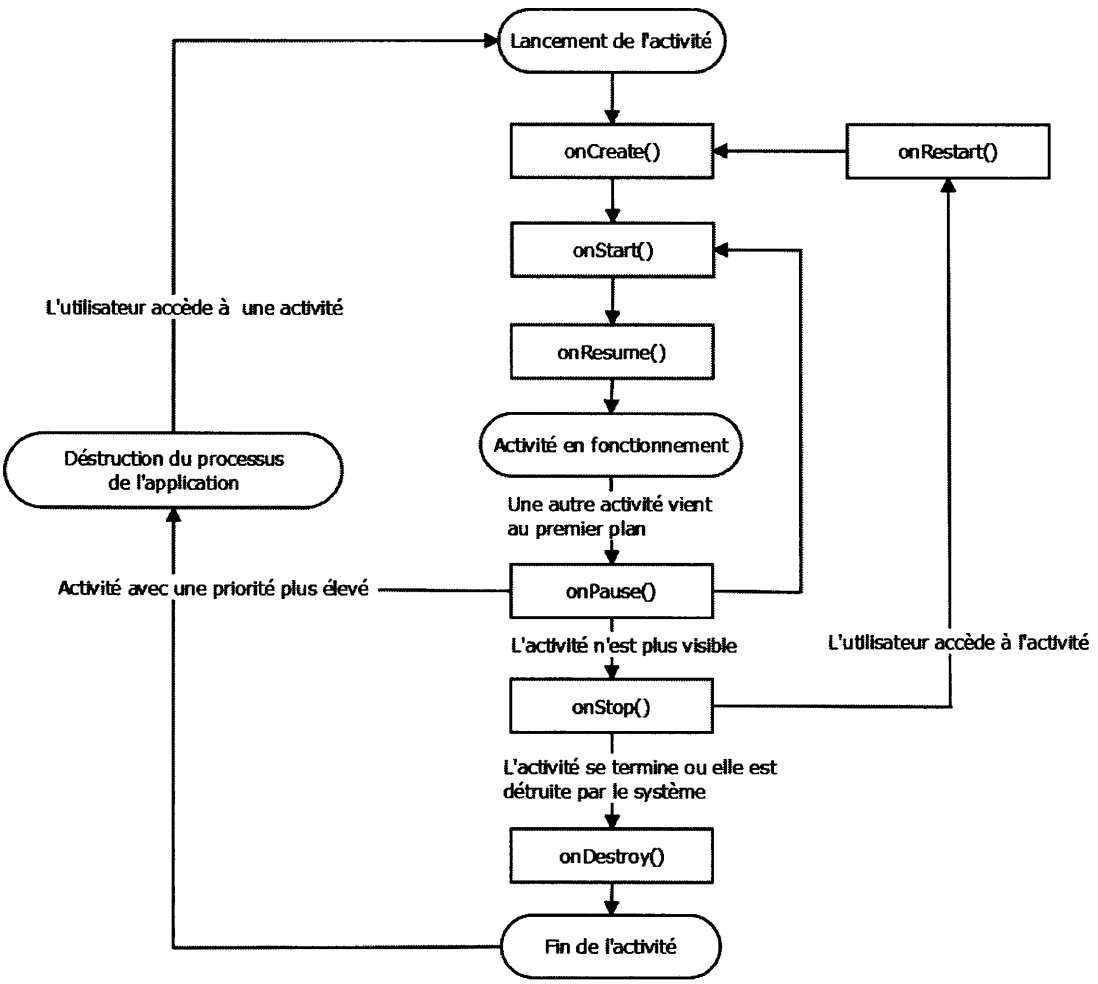

Figure 1.2 États possibles d'une activité. Source : `https://developer.android.com/topic/libraries/architecture/lifecycle`

aux messages diffusés provenant d'éléments extérieurs à l'application courante, tels que des messages provenant du système d'exploitation ou alors d'une autre application. Ensuite, un composant important est le fournisseur de contenu. Celui-ci permet de faire transiter des données entre applications. Il existe d'autres composants tels que les fragments. Les fragments se comportent comme une activité : ils sont liés à une interface fonctionnent selon un cycle de vie précis. La différence est qu'un fragment n'est pas lié à un écran complet comme une activité,

mais seulement à une partie de l'écran. L'intérêt principal des fragments est de scinder les activités d'une application en composant réutilisable. Ils sont contenus dans une activité Android. Les fragments sont utilisés pour développer des interfaces graphique évolutive en fonction de la taille de l'écran ou de l'orientation du périphérique. De plus, ils permettent de faciliter la maintenance du code et la réutilisation grâce à l'encapsulation de celui-ci. La figure 1.3 illustre un exemple d'utilisation des fragments dans une application Android.

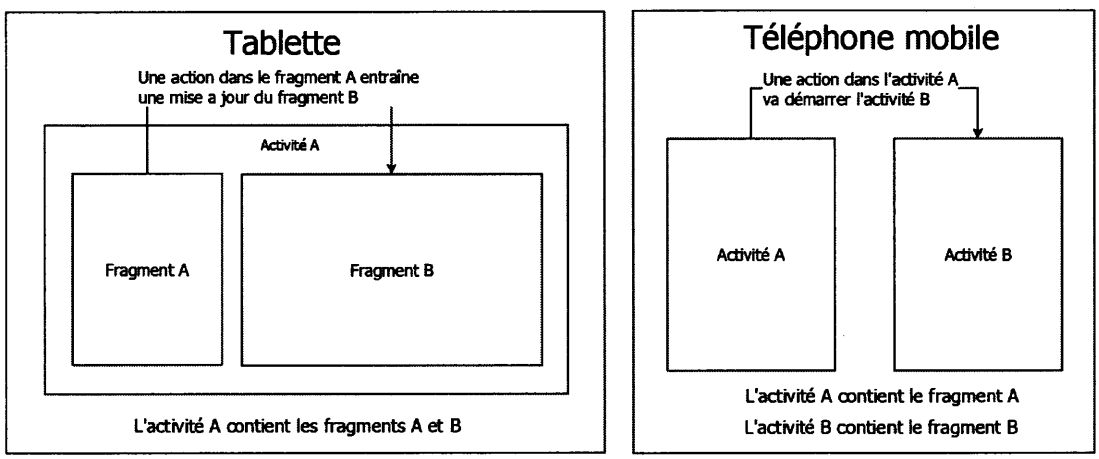

Figure 1.3 Exemple d'utilisation de fragments. Source : `https://developer.android.com/guide/components/fragments`

1.3 Les défauts de code

Les défauts de code (Fowler *et al.*, 1999) (ou code *smell* en anglais) sont des erreurs d'implémentation pouvant entraîner une dégradation de la qualité d'un logiciel. La présence de défauts de code dans un logiciel provoque une maintenance difficile et une évolution du logiciel très complexe. De manière générale, les défauts de code ont un impact négatif sur les systèmes et sur la qualité globale d'un logiciel (Suryanarayana *et al.*, 2014). Il est important de préciser le fait qu'un défaut de code n'est pas spécialement un bogue qui entraîne des erreurs de compilation ou

des erreurs durant le fonctionnement. Ils sont définit comme des indicateurs de potentiels désagréments. Plus précisément, les défauts de code sont considérés comme des violations des principes fondamentaux de conception (Suryanarayana *et al.*, 2014). Par exemple, une *God Class* est une classe possédant beaucoup de responsabilités et contrôlant un très grand nombre d'autres classes (Riel, 1996). Cette erreur de conception entraîne une très grande complexité au niveau de la maintenance du système. Un défaut de code peut aussi être une méthode trop longue ou beaucoup trop complexe (Fowler *et al.*, 1999). Une telle méthode est difficile à comprendre et à maintenir. Il est possible de prendre également comme exemple un code qui n'est jamais exécuté dans une application, celui-ci entraînera une difficulté à la compréhension du code dû à sa non-utilité dans l'application (Mäntylä et Lassenius, 2006).

1.4 Fouille de données

Dans cette partie nous décrivons les concepts de base de la fouille de données, puis nous présentons un peu plus en profondeur le type d'algorithme utilisé dans notre travail.

La fouille de données (ou exploration de données) est une discipline ayant pour but d'extraire du savoir en analysant une grande quantité de données. Ces analyses sont rendues possibles grâce à l'utilisation de différentes disciplines, telles que la statistique, l'informatique ou encore l'intelligence artificielle. Cette discipline est appliquée dans des domaines variés tels que la détection de fraude ou la gestion des relations client. Le but premier est d'explorer une très grande quantité de données, de créer des modèles (structures de données, motifs redondants) et d'en extraire des connaissances intéressantes. Il existe deux catégories d'exploration de

données :

— Les méthodes descriptives ;
— Les méthodes prédictives ;

1.4.1 Les méthodes descriptives

La première catégorie permet de dresser le portrait d'un ensemble de données. Les données n'ont pas de rapport particulier entre elles, et ne possèdent aucune indication concernant un groupe d'appartenance particulier. Le but ici est d'effectuer un rassemblement entre les données dites *proches* et donc homogènes. Cela permet en finalité d'extraire des comportements ou des tendances par rapport à une population. Le but final étant de créer des groupes (clusters) dans cette population. Ces différents groupes doivent vérifier deux propriétés :

— Les données d'un groupe doivent être relativement similaires
— Les données de différents groupe doivent être les plus distincts possible.

Cette catégorie regroupe des méthodes de différents domaines tels que la statistique (analyse factorielle) ou encore l'apprentissage non supervisé : K-means, DBSCAN, etc. Les techniques d'extraction de règles d'associations sont aussi présentes dans cette catégorie. Les différents groupes

1.4.2 Les méthodes prédictives

La deuxième catégorie, à savoir les algorithmes prédicatifs, permet de faire des prédictions suite à un apprentissage. Nous parlons particulièrement des techniques d'apprentissage automatique supervisé. Les classes (ou étiquettes) de chaque données sont connues au préalable. Les données sont analysées par l'algorithme et un

modèle de classification est construit. Celui-ci permet par la suite de prédire la classe d'une nouvelle instance de données en fonction de ce que l'algorithme a appris durant la construction du modèle. Il existe différents algorithmes permettant de faire de l'apprentissage machine supervisée, à savoir : Les Arbres de décisions, KNN, SVM, classification Bayésienne, réseau neuronal, etc.

1.4.3 Mesure d'efficacité d'un classifieur

Dans le but d'évaluer l'efficacité d'un modèle d'apprentissage, la métrique la plus utilisée est la F-mesure. La F-mesure est une moyenne harmonique entre la précision et le rappel, et c'est un bon estimateur de l'efficacité d'un détecteur (Goutte et Gaussier, 2005).

Le rappel est le nombre d'instances correctement classées comme classe A par rapport au nombre réel d'instances étant définies comme étant de la classe A. Il se calcule de cette manière :

$$rappel = \frac{ICC}{IR}$$

Où ICC représente le nombre d'instances correctement classées et IR est le nombre réel d'instances du jeu de données.

La précision permet d'avoir une autre indication concernant le classifieur. Celle-ci est exprimée par le rapport entre le nombre d'instances correctement attribuées à une certaine classe et le nombre total d'instances attribuées à cette classe. Si la précision est faible, cela signifie que le détecteur attribue des instances non pertinentes à une certaine classe. La précision se calcule de cette manière :

$$precision = \frac{ICC}{ICT}$$

Où ICC est le nombre d'instances correctement classées et ICT est le nombre total d'instances classées. Enfin, la F-mesure est la moyenne harmonique de ces deux premières mesures. Elle s'exprime de cette manière :

$$Fmesure = 2 \times \frac{precision \times rappel}{precision + rappel}$$

Les valeurs de ces trois mesures sont entre 0 et 1, 1 étant la valeur indiquant la meilleure performance. De ce fait, une F-mesure proche de 1 dénote une très bonne performance lors d'une classification ou une détection. Cette métrique permet aussi de mesurer l'efficacité d'une détection. De ce fais, nous utilisons cette métrique pour l'évaluation de notre approche.

1.4.4 Règles d'association

Les techniques d'extraction de règles d'association sont des techniques consistants à détecter des associations entre des valeurs d'attributs d'un jeu de données. Ce sont des techniques exploratoires mises en avant par (Agrawal *et al.*, 1993) permettant d'extraire des motifs récurrents dans un jeu de données. Agrawal *et al.* présentent dans un article l'algorithme APRIORI (Agrawal *et al.*, 1994). Cet algorithme permet d'extraire des motifs fréquents en calculant la fréquence d'apparition des items entre eux.

Ces techniques vont permettre d'analyser des jeux de données transactionnels et d'en analyser les relations en fonction des valeurs de chaque attribut. Les données transactionnelles sont un ensemble de transactions (enregistrement) représenté par des données binaires. Le but de ces techniques est d'extraire des règles d'association sous la forme : $X, Y \Rightarrow Z$, ou X, Y et Z sont des attributs (item). Les éléments de part et d'autre de la flèche sont appelés jeux d'item (*itemset* en

anglais). Le jeu d'item à gauche de la flèche est appelé le prémisse. Le jeu d'item à droite de la flèche est appelé la conséquence. Dans le but d'illustrer le fonctionnement et le type d'application des règles d'association, nous allons présenter un exemple se basant sur le panier d'un client de supermarché. Un supermarché enregistre les achats d'un client à chacun de ses passages. Au fur et à mesure des visites du client, un jeu de transaction est constitué. Le jeu de transaction sera sous la forme :

Tableau 1.1 Exemple de jeu de données représentant le panier d'un client de supermarché.

Pizza	**Couche**	**Pain**	**Lait**
true	true	false	false
true	false	true	true
false	true	false	false
true	false	false	false
true	true	false	true
false	true	true	true

L'extraction de règles d'association va permettre d'analyser les relations entre les différents items du jeu de données. Le but est de mettre en avant des habitudes d'achat : si le client achète du pain, alors il y a des très grandes chances qu'il achète également du lait. Il existe deux métriques importantes pour mesurer la pertinence d'une règle d'association, à savoir : le support et la confiance (Agrawal *et al.*, 1993). Le support est la fréquence d'apparition du motif. La confiance représente la proportion d'apparitions de l'item Y lorsque l'item X est présent lui aussi, dans le cas de la règle $X \Rightarrow Y$. La confiance est mesurée par la proportion de transactions (itemset) avec l'item X et dans lequel l'item Y est présent aussi.

Le support est calculé de cette manière :

$$Support\{X \Rightarrow Y\} = \frac{(X \cup Y).count}{T}$$

Où T est le nombre total de transactions.

La confiance est calculée de cette manière :

$$Confiance\{X \Rightarrow Y\} = \frac{Support\{X \Rightarrow Y\}}{Support\{X\}}$$

Ces deux métriques possèdent une valeur comprise entre 0 et 1. Le but est donc de générer des règles possédant une confiance et un support élevé (proche de 1).

1.5 Conclusion

Ce chapitre nous a permis de définir et de décrire les différents concepts en rapport avec notre recherche, à savoir : les défauts de code, le système d'exploitation Android ainsi que son architecture, les applications et les différents composants puis pour finir la fouille de données. Le chapitre 2 présentera l'état de l'art concernant les défauts de code ainsi que les méthodes de détection.

CHAPITRE II

REVUE DE LA LITTÉRATURE

Ce chapitre présente les travaux en rapport avec les défauts de code sur des systèmes orientés objet et Android. Nous commençons par présenter les défauts de code de manière générale, puis nous allons plus en détail sur les outils permettant de les détecter. Ensuite, nous abordons le cas des systèmes Android. Pour finir, une section présente les travaux couplant la détection de défaut de code et les technologies d'exploration de données. Étant donné qu'il n'y a que peu de travaux combinant ces deux domaines, notre revue de littérature porte autant sur les systèmes Android que sur les systèmes orientés objet classiques.

2.1 Défauts de code Android

Comme nous avons pu le voir dans la chapitre précédent, un défaut de code est une mauvaise pratique portant sur l'implémentation ou sur la conception d'un système. Ces défauts de code sont indépendants de l'environnement utilisé. Malgré cela, chaque environnement peut présenter ses propres défauts de code liés directement à ses spécificités. Les spécificités peuvent être matérielles ou tout simplement liées à l'environnement logiciel dans lequel l'application est exécutée. Concernant les systèmes Android, le langage utilisé pour développer des applications est Java. De ce fait, nous retrouvons dans les applications Android des

défauts de code et des mauvaises pratiques liées aux langages orientés objet. Mais la plate-forme étant différente, il existe des défauts de code bien plus spécifiques aux systèmes Android. Les travaux de (Reimann *et al.*, 2014) ont permis d'établir un catalogue de 30 défauts de code spécifiques à Android. Il y a cinq catégories différentes permettant de séparer les défauts de code, à savoir : interface utilisateur, entrée/sortie, implémentation, réseau et base de données. Une grande majorité est liée à l'implémentation. Ce catalogue a permis de servir de base à plusieurs travaux en rapport avec les défauts de code Android (Hecht *et al.*, 2015; Kessentini et Ouni, 2017; Palomba *et al.*, 2017a). De plus, la documentation Android met en avant une liste de mauvaises pratiques [1] et permet d'éviter d'affecter les performances d'une application. Cette liste a permis d'inférer des défauts de code Android dans différents travaux (Hecht *et al.*, 2015; Carette *et al.*, 2017). Notre recherche porte sur les défauts de code présents dans les systèmes Android. De ce fait, cela englobe à la fois les défauts de code orienté objet (OO) et les défauts de code spécifiques à Android. Dans la suite de cette section, nous présentons les divers défauts de code en rapport avec notre recherche.

2.1.1 *BLOB Class (BLOB)*

Un *BLOB* (Brown *et al.*, 1998) ou une *God Class* est une classe contenant un très grand nombre d'attributs et de méthodes. Elle possède aussi une très faible cohésion entre ses méthodes. La grande taille de cette entité par rapport aux autres lui confère un très grand nombre de responsabilités dans une application. Ce type de classes est très difficile à maintenir et rend l'évolution de l'application difficile.

---

1. (En ligne; Accès Septembre-2018) Android Performance tips. `https://developer.android.com/training/articles/perf-tips.html`.

2.1.2 *No Low Memory Resolver (NLMR)*

Dans le système Android, la méthode onLowMemory() est appelée par le système lorsque celui-ci est en manque de mémoire. Cette méthode permet de libérer de l'espace mémoire et doit donc être implémentée par chaque activité Android. En l'absence de cette méthode, le système peut mettre fin au processus entraînant l'arrêt non désiré d'une application. On considère donc son absence comme un défaut de code.

2.1.3 *Swiss Army Knife (SAK)*

Un *Swiss Army Knife* est une interface complexe. C'est-à-dire qu'elle regroupe un très grand nombre de méthodes. Le problème étant que ce type d'interface entraîne une complexité supplémentaire lors de la compréhension du code des classes enfants.

2.1.4 *Long Method (LM)*

Le défaut de code *Long Method* est présent lorsqu'une méthode comporte beaucoup plus d'instructions que les autres méthodes. Si le défaut de code *Long Method* est présent, c'est qu'il est possible de diviser la méthode en plusieurs petites méthodes. Ce type de méthode entraîne une complexité dans la compréhension, la maintenance et l'évolution du code (Fowler *et al.*, 1999).

2.1.5 *Complex Class (CC)*

Les *Complex Class*, comme leur nom l'indique, sont des classes avec une complexité très élevée. Elles contiennent souvent des méthodes complexes. Elles sont de ce

fait très difficiles à comprendre et à maintenir (Fowler *et al.*, 1999). La complexité est calculée en effectuant la somme des complexités internes de chaque méthode. Pour définir la complexité d'une méthode, on utilise la complexité cyclomatique (McCabe, 1976).

2.1.6 *Leaking Inner Class (LIC)*

Dans un système Android, une classe interne (*inner class*) est une classe qui est définie au sein d'une autre classe. Les classes internes non statiques possèdent une référence vers les classes externes, de ce fait il est obligatoire que la classe externe (classe contenant la classe interne) soit instanciée pour avoir une instanciation de la classe interne. Ce type d'implémentation de classe interne entraîne des fuites mémoires (Lockwood, 2013; Reimann *et al.*, 2014). Par exemple, si une autre entité a besoin de faire référence à la classe interne, alors la classe externe est gardée en mémoire même si elle est inutile au système. La classe interne doit donc être statique afin qu'elle demeure accessible sans l'utilisation d'une instance de la classe externe.

2.1.7 *Heavy AsyncTask (HAS)*

Une tâche asynchrone (AsyncTask) est une tâche s'effectuant en parallèle de l'exécution principale du programme. Elle est considérée comme une tâche de fond. Trois méthodes sont tout de même exécutées sur le processus principal, à savoir : onPostExecute, onPreExecute, onProgessUpdate. Ces trois méthodes ne doivent pas effectuer de traitement lourd ou même bloquant, car cela peut entraîner un manque de réactivité de l'interface ou encore un blocage de l'application.

2.1.8 *Heavy Broadcast Receiver (HBR)*

Les récepteurs de diffusion (*Broadcast Receiver* en anglais) permettent aux applications de communiquer avec des entités extérieures telles que le système ou encore d'autres applications. Ils interceptent des messages, que l'on appelle *Intents*, permettant aux différents composants de communiquer entre eux. Ce type d'action peut être effectué de manière asynchrone et donc en tâche de fond. Cependant, les récepteurs de diffusion implémentent une méthode, *onReceive*, qui s'exécute sur le processus principal. Cette méthode ne doit pas effectuer de traitement lourd voir bloquant sous peine de ralentir l'application et de provoquer des blocages.

2.1.9 *Member Ignoring Method (MIM)*

Lorsqu'une méthode n'accède pas à un attribut ou n'est pas un constructeur, il est conseillé d'implémenter cette méthode en statique afin d'augmenter les performances. Effectivement, invoquer une méthode statique est environ 15% plus rapide qu'une invocation dynamique[2]. L'utilisation des méthodes statiques est considérée comme une bonne pratique, car celle-ci rend la compréhension du code bien plus aisée et garantit que l'état de l'objet ne sera pas modifié (Reimann *et al.*, 2014).

2.1.10 *HashMap Usage (HMU)*

L'utilisation des traditionnels HashMap est déconseillée s'il y a une petite quantité de données. L'API Android fournit les ArrayMap et SimpleArrayMap afin de remplacer le HashMap pour des Map contenant un petit nombre de valeurs. Ces

---

2. (En ligne; Accès Septembre-2018) Android Performance tips. `https://developer.android.com/training/articles/perf-tips.html`.

deux Map sont adaptées aux périphériques mobiles et permettent de consommer moins de mémoire. Par conséquent, la création de HashMap est considérée comme une mauvaise pratique (Haase, 2015) si celles-ci sont utilisées pour des tailles de Map de quelques centaines de valeurs. Cependant, si le jeu de données à stocker dans le Map est très élevé, l'utilisation des HashMap est préconisée.

2.2 Détection des défauts de code

Les défauts de code étant néfastes pour les systèmes et applications, beaucoup de travaux se sont portés sur la détection de ces défauts de code. Nos travaux portent spécifiquement sur les systèmes Android. Cependant les applications Android étant développées en Java, nous estimons important de présenter les cas plus généraux portant sur les systèmes orientés objet. De plus, il n'y a à ce jour que peu de travaux portant spécifiquement sur l'environnement Android.

Des métriques peuvent être extraites d'une application ou d'un système en analysant les composants de ceux-ci. Ces métriques de qualité permettent de mesurer la qualité d'un logiciel (Aggarwal *et al.*, 2009; Basili *et al.*, 1996; Li et Shatnawi, 2007; Singh *et al.*, 2010; Chidamber et Kemerer, 1994). L'utilisation de ces métriques a permis de développer des techniques d'identification et de détection de défauts de code dans des systèmes orientés objet tels que DECOR (Moha *et al.*, 2010).

Concernant la détection dans les systèmes orientés objet, une liste de différents défauts de code a été proposée par Fowler(Fowler *et al.*, 1999). Chaque défaut de code possède une liste de ses symptômes et la manière de le corriger (réusinage) . (Ciupke, 1999) propose une approche basée sur les violations des règles de conception. Il analyse le code légataire, précise les problèmes de conception fréquents sous forme de requêtes et va localiser manuellement les occurrences de ces

défauts de code dans un modèle extrait du code source. La plupart des outils de détection utilisent les métriques de qualité extraites par l'analyse de code source.

Par la suite (Kothari *et al.*, 2004) proposent une approche permettant de visualiser la présence des défauts de code en colorant les parties de code en fonction de la valeur des métriques de qualité. (Marinescu, 2004) effectue une détection en créant des règles de détection utilisant les métriques et leurs valeurs. Marinescu parvient à détecter neuf défauts de code spécifiques aux systèmes orientés objet avec une précision supérieure à 50%, tous défauts de code confondus. (Van Emden et Moonen, 2002) proposent un outil totalement automatique afin de détecter des défauts de code dans du code Java. (Moha *et al.*, 2010) présentent DECOR, une approche pour mettre en avant les spécificités des défauts de code et permettant de les détecter. Ils utilisent un langage spécifique au domaine (DSL) afin de définir les défauts de code à un haut niveau d'abstraction. Ils identifient quatre défauts de code : *BLOB*, *Swiss Army Knife*, *Functional Decomposition*, *Spaghetti Code.*

Concernant les systèmes Android, il y a peu de travaux à ce jour. (Linares-Vásquez *et al.*, 2013) utilisent des techniques spécifiques aux systèmes orientés objet appliqués aux systèmes Android. Ils détectent 18 défauts de code dans 1,343 applications Android. Verloop *et al.* metent en avant le fait que les différents défauts de code présents dans les systèmes Android héritent la plupart du temps des classes de l'API Android (Verloop, 2013). De plus, selon Verloop *et al.* les applications Android possèdent des métriques de qualité spécifiques. Pour la détection de défauts de code spécifiques à Android, (Hecht *et al.*, 2015) présentent PAPRIKA, un outil utilisant des règles pour la détection. Ils découvrent que quatre défauts de code spécifiques à Android apparaissent plus fréquemment que les défauts des systèmes orientés objet. (Palomba *et al.*, 2017a) développent un outil, ADOCTOR, permettant de détecter 14 défauts de code spécifiques à Android.

2.3 Exploration de données et défauts de code

Il y a peu de travaux couplant l'exploration de données et la détection de défaut de code dans les systèmes Android. De ce fait, nous élargissons notre revuer de littérature pour inclure les travaux utilisant l'exploration de données en rapport avec les défauts de code. D'une manière général, nous restons quand même focalisés sur les systèmes orientés objet Java. Pour commencer, (Khomh *et al.*, 2011) développent un outil couplant les réseaux bayésiens et une approche par les buts nommés BD-TEX permettant de détecter des défauts de code. Ils parviennent à effectuer une détection pour trois défauts de code différents. Par la suite (Maiga *et al.*, 2012b) utilisent un algorithme d'apprentissage automatique, SVM, pour détecter des défauts de code. Grâce à cette approche, ils parviennent à obtenir une classification efficace pour quatre défauts de code. Les données sont générées à l'aide de POM[3], un outil analysant le code source Java et pouvant extraire jusqu'à soixante métriques. Ces métriques constituent le jeu de données permettant d'entraîner l'algorithme. (Maiga *et al.*, 2012a) améliorent leur approche en proposant SMURF, une méthode incrémentale utilisant aussi l'algorithme SVM. À chaque classification, SMURF ajoute les instances de données classifiées à son jeu de données d'entraînement afin d'augmenter ses scores de classification et donc d'améliorer l'apprentissage de l'algorithme.

Fontana *et al.* mettent en avant une étude comparative d'algorithmes de classification pour détecter la présence de défauts de code dans une application (Fontana *et al.*, 2013). Leurs jeux de données comportent quatre classes différentes pour la classification : (1) pas de défauts de code, (2) présence de défauts de code légers, (3) présence de défauts de code, (4) présence de défauts de code sévères. Ils effectuent une classification avec six algorithmes d'apprentissage machine dif-

3. (En ligne ; Accès Janvier 2018). Ptidej. `https://wiki.ptidej.net/doku.php?id=pom`.

férents, à savoir : Arbre de décision, forêt aléatoire, bayésien naïf, SMO, SVM. La plupart des algorithmes obtiennent une F-mesure supérieure à 90%. Les jeux de données sont composés de métriques de qualité extraites en analysant le code source de chaque application. Les mêmes auteurs présentent plus tard une longue étude portant sur une comparaison de différents algorithmes de classification pour la détection de défauts de code (Fontana *et al.*, 2016). Ils procèdent à une comparaison de 16 algorithmes d'apprentissage machine et avancent le fait que les arbres de décision et les forêts aléatoires sont les techniques les plus efficaces. De plus, ils montrent qu'un faible jeu de données (une centaine d'occurrences) suffit à entraîner un algorithme afin d'obtenir une précision supérieure à 95%.

(Di Nucci *et al.*, 2018) répliquent les études faites par Fontana *et al.* et mettent en avant le fait que les résultats obtenus sont très corrélés à la nature de leurs données. Ils trouvent plusieurs biais à la détection effectuée dans l'étude de Fontana *et al.* Premièrement, la répartition des valeurs des métriques du jeu de données d'entraînement est très disparate. Effectivement, les instances du jeu de données possédant un défaut de code ont des valeurs très marquées. Ce type de données facilite grandement la classification pour n'importe quel algorithme d'apprentissage machine. Deuxièmement, suite aux premières observations, Di Nucci *et al.* utilisent leurs propres jeux de données afin de répliquer les expérimentations. Les différences sont cette fois moins évidentes. Les scores de classification possèdent une précision inférieure à 90% et aucune F-mesure ne dépasse un score de 50%.

(Jiang *et al.*, 2014) utilisent l'algorithme K plus proche voisin adapté pour l'apprentissage machine non supervisé. Ils calculent la distance entre deux versions d'une entité[4] afin de détecter le défaut de code *Divergent change*. Le but est de corriger le défaut de code en extrayant le schéma de réusinage du défaut de code

4. Classe ou méthode

en analysant les résultats de l'algorithme. Il existe des travaux utilisant les réseaux de neurones pour la détection de défauts de code. (Kim, 2017) utilise des réseaux de neurones dans le but de détecter sept défauts de code. Les résultats sont encourageants avec une précision supérieure à 90% pour l'ensemble des détections.

Des travaux utilisent d'autres types de techniques d'exploration de données, notamment de l'extraction de règles d'association. (Palomba *et al.*, 2015) utilisent l'algorithme APRIORI afin d'analyser l'historique des versions d'un système. Le but final est de détecter la présence de défauts de code. L'outil proposé, HIST permet de détecter 5 défauts de code. Par la suite, (Palomba *et al.*, 2017b) étudient les cooccurrences des différents défauts de code. Ils utilisent cette fois encore l'algorithme APRIORI(Agrawal *et al.*, 1993) et mettent en avant le fait que six défauts de code apparaissent fréquemment dans les mêmes entités. Pour finir, (Kessentini et Ouni, 2017) proposent une approche utilisant la programmation génétique multiobjectifs dans le but de générer des règles de détection de défauts de code pour les systèmes Android. Les règles de détection obtiennent une efficacité supérieure à 82%.

2.3.1 Conclusion

Cette revue de littérature sur les défauts de code nous donne une vue d'ensemble de l'avancement du domaine autant dans les systèmes orientés objet que dans les systèmes Android, bien que dans ces derniers systèmes il n'y ait pas encore suffisamment de travaux de recherche. De manière générale, les outils de détection se servent de métriques de qualité comme représentation des entités, et la plupart utilisent des règles de détection. Les règles de détection sont définies manuellement par rapport à la définition des défauts de code. Pour le système Android, il semble y avoir encore beaucoup de travail concernant la détection de défauts de code ou

même la découverte de défauts spécifiques à Android. Concernant l'exploration de données et les défauts de code, tous systèmes confondus, il en ressort des difficultés notamment pour la classification. Certaines entités comportant des défauts de code possèdent des valeurs de métriques très proches de celles ne comportant pas de défauts. La séparation n'étant pas toujours triviale, il est très difficile pour les algorithmes de détecter efficacement les entités comportant des défauts de code dû à la proximité avec celles n'en possédant pas.

Pour conclure, l'utilisation de règles semble le plus efficace pour la détection de défauts de code. Mais elle entraîne une subjectivité liée à l'écriture manuelle de celles-ci. Le but des travaux utilisant l'exploration de données est de réduire un maximum la subjectivité pour la détection, mais il semble qu'aucune méthode à ce jour ne soit vraiment efficace pour la détection de défauts de code Android.

CHAPITRE III

FAKIE : GÉNÉRATION AUTOMATIQUE DE RÈGLES DE DÉTECTION DE DÉFAUTS DE CODE ANDROID

Dans ce chapitre nous présentons l'approche outillée FAKIE permettant de générer des règles de détection de défauts de code. L'approche utilisée dans FAKIE est présentée de manière succincte, puis les différentes étapes du processus sont détaillées par la suite.

3.1 Présentation de l'approche

Dans l'optique de générer des règles de détection, nous avons utilisé un algorithme de fouille d'association sur des données relatives à différents défauts de code. Le but est d'analyser les motifs fréquents dans les données et de générer des règles d'association pouvant potentiellement servir de règles de détection de défauts de code. Notre méthode se déroule en 4 étapes allant de l'extraction des données à la sélection et le filtrage des règles générés par FP-GROWTH. La figure 3.1 montre les différentes étapes de l'approche.La première étape correspond à l'extraction de données à partir d'applications Android. Puis la deuxième étape va permettre de générer un jeu de données composé de métriques de qualité et de relations. Les deux dernière étapes consiste en l'extraction de règles d'associations à partir des jeux de données puis le filtrage des résultats.

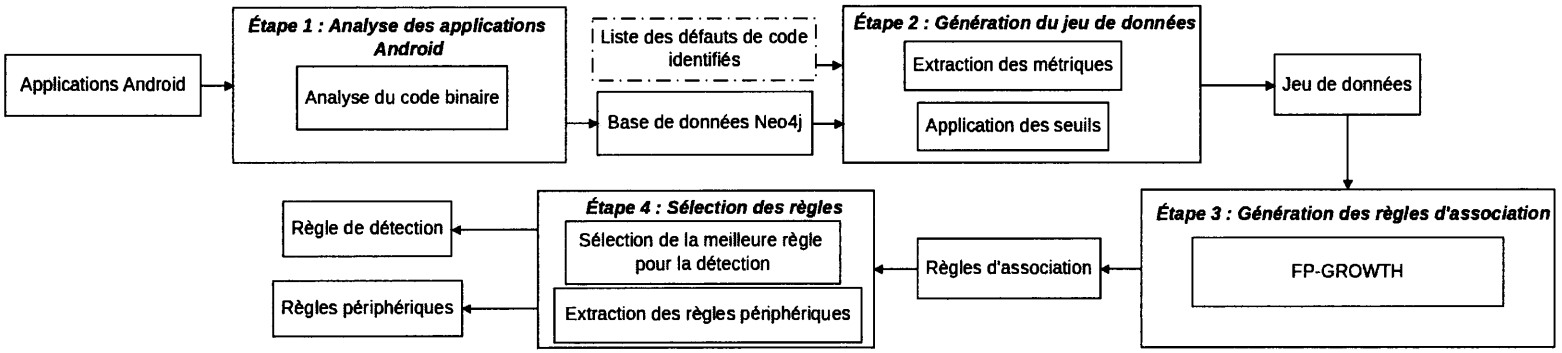

Figure 3.1 Vue d'ensemble de l'approche FAKIE.

3.2 Étape 1 : analyse des applications Android

Entrée : applications Android

Sortie : base de données Neo4j

Description : la toute première étape est l'analyse des applications Android. Afin d'extraire les métriques de qualités ainsi que les relations entre classes et méthodes, nous avons fait le choix d'utiliser la partie analyse de l'outil PAPRIKA. Cet outil permet d'extraire des informations relative aux applications analysées puis de calculer les métriques de qualités. PAPRIKA utilise le cadre d'application SOOT (Vallée-Rai *et al.*, 2010) couplé avec son module DEXPLER (Bartel *et al.*, 2012) afin d'analyser les applications. SOOT va permettre d'analyser le code binaire de la machine virtuelle Dalvik et de le convertir en une représentation très proche du langage Java. Cette analyse permet de créer un modèle SOOT. Il contient toutes les informations relatives aux entités, à la hiérarchie de classes et aux appels de méthodes. À partir de ce modèle, les métriques et les propriétés sont récupérées et stockées dans une base de données orientée graphe. La base de données est une base de données NEO4J [1]. L'utilisation de ce type de stockage permet d'analyser un grand nombre d'applications et d'avoir toutes les informations relatives à celles-ci stockées au même endroit. La base de données est représentée par un système de noeuds connectés entre eux par des arcs. Chaque noeud représente une entité (classe, méthode), et chaque arc représente une relation. Par exemple, il est possible d'obtenir : Entité1 (appartient à) Entité2 ou encore Entité1 (Appel) Entité2. NEO4J possède un langage de requêtes puissant, CYPHER [2], permettant de récupérer des informations sur les noeuds, ou encore de faire des requêtes permet-

1. (En ligne ; Accès Juin-2017) Neo4j. `https://neo4j.com`.

2. (En ligne ; Accès Juin-2017) Cypher. `https://neo4j.com/developer/cypher-query-language`.

tant de filtrer les résultats. Ce langage de requêtes nous a notamment servis lors de la validation pour l'identification manuel de défaut de code dans des applications Android. De plus, les règles de détection générées par FAKIE sont adaptées pour effectuer une détection à l'aide de requête Cypher. Il est possible de récupérer toutes les informations concernant un noeud par la requête suivante :

```
MATCH (m : Method) WHERE m.full_name = 'onPostExe-
cute#org.wordpress.android.ui.stats.WPComLoginActivity' RE-
TURN m.NoP, m.NoDL, etc.
```

La requête CYPHER ci-dessus se décompose en trois mots clés, à savoir :

- **MATCH** : ce mot clé va permettre d'indiquer quel type d'entité nous cherchons. Ici, le (m : Method) indique que nous recherchons une entité de type *method* qui sera représentée par l'identificateur *m* dans la requête.
- **WHERE :** ce mot clé va permettre d'indiquer les critères de sélection de la requête. Ici, nous recherchons une entité de type *method* possédant comme nom complet *onPostExecute#org.wordpress.android.ui.stats.WPComLoginActivity.* La valeur de l'attribut *full_name* est indiquée par *m.full_name.*
- **RETURN :** ce mot clé permet d'indiquer quelles informations vont être renvoyées lorsqu'une entité correspond aux critères énoncés par le mot clé WHERE.

Les noeuds de la base de données sont constitués principalement de métriques permettant de décrire les entités du système. Le tableau 3.1 contient toutes les métriques et propriétés récupérées ou générées pendant l'analyse du code compilé des applications Android. Certaines métriques ne concernent que les classes d'un système, d'autres seulement les méthodes et certaines concernent les deux types d'entités. Le tableau 3.1 fait référence aux métriques analysées.

Tableau 3.1 Liste des métriques de qualité et des propriétés relative aux noeuds.

Métrique	Entité	Description
NoM	Classe	Nombre de méthodes
DoI	Classe	Profondeur d'héritage
NoII	Classe	Nombre d'interfaces implémentées
NoA	Classe	Nombre d'attributs
NoC	Classe	Nombre de classes filles
CLC	Classe	Complexité de la classe
LOC	Classe	Manque de cohésion entre les méthodes
isAbstract	Classe, Méthode	La classe ou la méthode est abstraite
isFinal	Classe, Méthode	La classe ou la méthode est finale
isStatic	Classe, Méthode	La classe ou la méthode est statique
isInnerClass	Classe	La classe est une classe interne
isInterface	Classe	La classe est une interface
isActivity	Classe	La classe est une activité Android
isBroadcastReceiver	Classe	La classe est un receveur de diffusion
isContentProvider	Classe	La classe est un fournisseur de contenu
isService	Classe	La classe est un service
NoP	Méthode	Nombre de paramètres de la méthode
NoDL	Méthode	Nombre de variables locales
NoI	Méthode	Nombre d'instructions de la méthode
NoDC	Méthode	Nombre d'appels directs à d'autres méthodes
NoC	Méthode	Nombre de références à la méthode courante
CC	Méthode	Complexité Cyclomatique
isGetter	Méthode	La méthode est un getter
isSetter	Méthode	La méthode est un setter
isInit	Méthode	La méthode est un constructeur
isSynchronized	Méthode	La méthode est synchronisée

3.3 Étape 2 : analyse de la base de données Neo4j et génération dynamique d'attributs

Entrée : base de données NEO4J.

Sortie : base de données exploitable pour l'extraction de règles d'association.

Description : Afin d'obtenir des données exploitables par l'algorithme de règles d'association (FP-GROWTH), il est important d'extraire les informations de la base de données. Pour effectuer cette action, FAKIE exécute plusieurs étapes permettant de générer un jeu de données sous la forme d'un fichier ARFF [3], à savoir :

— **Importation de la base de données :** Dans le but de transformer le graphe NEO4J, celui-ci est directement importé en mémoire pendant le processus afin d'être modifié au fur et à mesure des étapes.

— **Application d'un identifiant de défaut de code :** FAKIE va appliquer un identifiant, sur les entités identifiées au préalable, indiquant que celles-ci possèdent un défaut de code. Cette étape va permettre de faire la différence entre les entités dites neutres et celles infectées par un défaut de code. Cette étape est importante afin d'optimiser au maximum les étapes qui suivent étant donné qu'elles demandent énormément de ressources. De ce fait, toutes les prochaines étapes seront appliquées sur les noeuds de la base de données qui sont infectés par un défaut de code.

— **Conversion des labels d'entités en propriétés :** les labels des entités (classe et méthode) sont convertis en propriétés nominales. Cette étape

3. ARFF ou Attribute-Relation File Format est le format de fichier de données utilisé par la librairie d'apprentissage machine WEKA.

permet de connaître le type de l'entité une fois que les informations sont retranscrites sous la forme d'un fichier ARFF.

— **Ajout des propriétés supplémentaires :** Afin d'obtenir un jeu de données le plus complet possible, FAKIE va analyser le graphe et ajouter des informations aux noeuds, tels que la nature de la classe dans laquelle la méthode est contenue. Nous définissons la nature d'une classe par le type de classe auquel elle correspond : classe abstraite, interface, activité, etc. Ces informations peuvent être importantes et corrélées à la présence d'un défaut de code.

— **Suppression des méthodes superflues :** le but premier de la génération du jeu de données est d'obtenir un maximum d'informations sur chaque entité. Cela peut permettre d'extraire des relations de cause à effet entre la présence d'une méthode et la présence d'un défaut de code. La finalité étant de trouver une corrélation entre une méthode et la présence d'un défaut de code. Avant d'ajouter cette étape, les jeux de données avaient un très grand nombre d'attributs, très souvent supérieur à 1500. Ils étaient très complets, mais les ressources demandées par l'algorithme FP-GROWTH pour l'extraction de règles d'association sur de tels jeux de données étaient beaucoup trop importantes. Ainsi, nous avons été dans l'obligation de réduire la taille du jeu de données. Pour cela, nous avons mis en place une liste de méthodes à inclure dans les jeux de données. Pour établir cette liste de méthodes ayant un rapport avec les défauts de code, nous avons fait une revue de littérature. Ensuite, lors de la génération, FAKIE va analyser chaque méthode et les supprimera ou non en fonction de leur présence dans la liste. La suppression des méthodes entraîne inévitablement une perte d'information. Ces informations peuvent être importante mais dans

notre configuration actuel, nous n'avons pas d'autre choix que de réduire le nombre d'attributs au détriment de l'information.

— **Récupération des méthodes possédées par héritage :** lors de l'analyse du code compilé, les méthodes obtenues dans une classe par héritage sont ignorées. Ainsi, lors de la génération des données, FAKIE va parcourir la hiérarchie d'héritage afin de récupérer le nom de ces méthodes.

— **Application d'une liste pour les appels de méthodes :** de la même manière que l'étape 3.3, nous avons mis en place une liste d'appels de méthodes à conserver pour les jeux de données. Encore une fois, cette étape permet de réduire le coût en ressources lors de l'extraction de règles d'association effectuée par FP-GROWTH.

— **Conversion des données numériques en données binaires :** l'algorithme FP-GROWTH ne peut extraire des règles d'association qu'à partir de données binaires. Ainsi, nous appliquons un système de seuils détaillé par la suite. Comme expliqué dans les étapes précédentes, les données ne peuvent être exploitées que sous la forme d'attributs binaires. Afin de convertir les métriques numériques en attributs binaires, nous avons effectué une discrétisation. La discrétisation est un procédé permettant de découper des valeurs continues en classes homogènes. Nous avons choisi d'utiliser la boîte de Tukey (Tukey, 1977) pour discrétiser nos valeurs. Le principe de cette boîte de Tukey (ou boîte à moustaches) est une technique permettant d'analyser la distribution d'une population. En utilisant cette méthode, il est possible de souligner les valeurs atypiques d'une population. Pour déterminer nos seuils, nous faisons appel à une relation utilisant l'écart interquartile d'une distribution et nous définissons quatre seuils différents.

L'utilisation de plusieurs seuils permet de définir plus facilement la valeur seuil la plus appropriée lors de l'extraction des règles d'association. Nous nous basons sur la formule suivante : $(Q3 - Q1) * Q3 + X$, où X peut prendre les valeurs : 0.75, 1.5, 3, 4.5. Lorsque X prend la valeur 1.5, la valeur de la frontière haute est calculée. Nous avons ajouté de manière arbitraire d'autres valeurs de X afin d'obtenir un découpage supplémentaire des données continues. Cela permet d'extraire les valeurs atypiques selon différentes valeurs et d'avoir une précision supplémentaire avec l'utilisation de quatre seuils lors de l'extraction de règles d'association. Par exemple, avant application des seuils le jeu de données se présente sous la forme :

NoP NoDL NoI NoDC NoC CC isGetter isSetter...callInit CS
58 45 20 25 36 true true false...false HAS
12 10 22 17 5 true false false...false HAS
2 15 28 117 8 true true false...false HAS

Une fois les seuils appliqués, le jeu de données possède plus d'attributs et est sous la forme :

NoP>25 NoP>52.4 NoDL>12 NoI>22 NoDC>3 NoC>34 CC>18 isGetter
isSetter...callInit CS
true true false false true true true false...false HAS
true false true false true true false false...false HAS
false false true false true true true false...false HAS

Les attributs numériques sont convertis en plusieurs attributs binaires. Cette étape permet d'obtenir un jeu de données exploitables pour l'extraction de règles d'association.

— **Conversion des valeurs nominales en données binaires :** encore une fois, seules les données de type binaire sont exploitables par FP-GROWTH. FAKIE va convertir les noms de méthodes et de classes en valeurs binaires. Pour se faire, chaque valeur nominale est exprimée sous la forme d'un attribut ayant pour valeur vrai ou faux. Par exemple, si le nom d'une classe est *MainActivity* alors l'attribut sera *nameIsMainActivity.*

— **Suppression des entités ne comportant pas de défauts de code :** l'avant-dernière étape est la suppression des noeuds non infectés par un défaut de code. Cela permet d'obtenir un graphe ne comportant que les entités infectées.

— **Conversion de la base de données NEO4J en fichier ARFF :** La toute dernière étape consiste à extraire toutes les informations des entités restantes dans le graphe afin de constituer les jeux de données sous la forme de fichiers ARFF.

Implémentation : toutes ces étapes ont été développées en Java.

3.4 Étape 3 : extraction des règles d'association

Entrée : jeu de données sous la forme d'un fichier ARFF.

Sortie : règles d'association extraites par FP-GROWTH.

Description : Dans cette étape nous appliquons un algorithme de règles d'association sur les données. L'objectif est d'extraire des motifs récurrent impliqué dans la présence d'un défaut de code. Dans notre cas, un item d'un motif représente une des métriques extraites lors de l'analyse des applications An-

droid ainsi que sa valeur. Par exemple, une règle pourra être sous la forme : $LoC > 45 = true, CLC > 15 = true, NoA > 10 = true \Rightarrow BLOB$.

La traduction de la règle ci dessus correspond à : si le manque de cohésion entre les méthodes est élevé, que la complexité est élevée et qu'il y a un très grand nombre d'attributs, alors il y a de fortes chances qu'il y ait un défaut de code de type *Blob*.

Dans notre travail, nous avons fait le choix d'utiliser l'algorithme FP-GROWTH (Han *et al.*, 2000). FP-GROWTH, tout comme APRIORI, va permettre d'extraire des règles d'association en calculant la fréquence des items. FP-GROWTH est moins consommateur de ressources ce qui allège le processus de génération de règles d'association (Kumar et Rukmani, 2010; Győrödi *et al.*, 2004) et nous permet de traiter des jeux de données plus grand. Contrairement à APRIORI, FP-GROWTH ne va effectuer qu'un seul passage sur la base de données afin d'en faire une version compressée : le FP-TREE. Cette version est une projection de la base sous forme d'arbre. La génération des itemsets fréquents est effectuée à partir de cet arbre. La représentation sous forme d'arbre va permettre d'optimiser la génération d'itemsets fréquents, et donc la génération de règles d'association.

Implémentation : pour extraire les règles d'association, FAKIE utilise la libraire d'apprentissage machine WEKA (Witten *et al.*, 2016). Elle implémente l'algorithme FP-GROWTH et va nous permettre de procéder à l'extraction des règles d'association sur notre jeu de données. L'implémentions de WEKA prend en paramètres plusieurs informations, à savoir :

- La confiance minimum des règles extraites : entre 0 et 1 ;
- Le support minimum des règles extraites : entre 0 et 1 ;
- Les attributs du jeu de données devant obligatoirement apparaître dans les règles ;

— Le nombre de règles à générer ;

Concernant la valeur des paramètres choisis pour nos travaux, nous avons sélectionné : 0.1 pour la valeur de confiance ainsi que pour la valeur de support. Concernant la génération nous avons sélectionné 10,000 pour le nombre de règles à générer. L'attribut de jeu de données qui apparaît obligatoirement dans nos règles est celui indiquant la présence d'un défaut de code. Nous avons choisi ces valeurs dans le but d'extraire un maximum de règles d'association et donc d'informations. La valeur de 0.1 pour le support permet d'obtenir des règles couvrant au minimum 10% du jeu de données. Ce seuil est un bon compromis entre pertinence des règles générées et quantité de règles obtenues. Nous avons fait le choix de générer 10,000 règles afin d'avoir la certitude d'extraire presque l'intégralité des règles potentielles. Le but est de générer un maximum de règles afin de sélectionner les règles de détection optimales, ainsi que d'analyser les différentes relations entre les attributs et la présence des défauts de code.

3.5 Étape 4 : sélection des règles de détection

Entrée : liste des règles d'association générées dans l'étape 3.

Sortie : règles de détection.

Description : FAKIE applique un filtre sur les résultats obtenus après utilisation de l'algorithme FP-GROWTH. Cette étape permet d'effectuer une première sélection sur les règles générées. Les règles ne sont pas toutes pertinentes et certaines peuvent être redondantes entre elles. Ainsi, nous avons mis en place deux sélections importantes.

Premièrement, nous ne nous intéressons qu'aux règles sous la forme $n_0...n \Rightarrow X$ où X est l'attribut du jeu de données indiquant la présence d'un défaut de

code et n est un item quelconque. Si une règle possède un attribut indiquant la présence d'un défaut de code en prémisse ou que sa conséquence possède plusieurs attributs, celle-ci est supprimée. Cela nous permet d'obtenir des règles sous la forme plusieurs pour un. Nous sélectionnons ce type de règle puisque notre but premier est d'obtenir des règles indiquant la présence d'un défaut de code en fonction des métriques et des relations.

Deuxièmement, comme expliquée dans la section 3.4, la pertinence d'une règle est mesurée par sa confiance ainsi que son support. Lors d'une génération, il est possible d'obtenir des règles possédant la même confiance et le même support. Dans ce cas, la règle sélectionnée est celle comportant le plus gros itemset pour prémisse. Nous sommes donc sûrs d'obtenir la règle possédant le plus d'informations à support et confiance identiques. Ces deux étapes permettent d'obtenir la règle la plus pertinente générée par l'algorithme FP-GROWTH.

Implémentation : l'Algorithme 1 présente la première étape du processus de sélection des règles. En entrée, nous avons la liste des règles extraites lors de l'analyse avec FP-GROWTH. En sortie, nous obtenons une liste des règles filtrées. D'abord, nous regardons le nombre d'items présents dans la conséquence de la règle. S'il y en a qu'un seul, alors la règle est ajoutée à la liste des règles. Nous vérifions au préalable si chaque règle contient au moins un item indiquant la présence d'un défaut de code en conséquence.

Les Algorithmes 2 et 3 présentent la deuxième étape de sélection des règles. En entrée nous avons la liste des règles d'association après un premier filtre. En sortie nous avons une liste filtrée des règles. L'algorithme va comparer toutes les règles entre elles (taille de prémisse, confiance et support). Si une règle $\boldsymbol{A}$ possède une valeur de prémisse supérieure à une règle $\boldsymbol{B}$ et des valeurs de confiance et support identiques, $\boldsymbol{A}$ est ajoutée à la liste de règles de sortie. Si la règle $\boldsymbol{A}$ possède une

Algorithm 1: Sélection des règles du type : Plusieurs pour Un

Entrée : *liste_regles* : Liste contenant les règles d'associations

Sortie : *regle_filtre* : Une liste contenant les règles d'association filtrées

```
1  Début
2      regle_filtre ← []
3      # Pour chaque règle récupérer la conséquence, vérifier la taille
4      POUR CHAQUE regle ∈ liste_regles FAIRE
5          consequence ← regle.consequence()
6          SI la conséquence contient bien qu'une seule variable ALORS
7              regle_filtre.ajout(regle)
8          FIN
9      FIN
10     retourner regle_filtre
11 Fin
```

meilleure confiance par rapport à la règle ***B*** alors la règle ***A*** est ajoutée à la liste de règles de sortie. Si la règle ***A*** possède la même valeur de confiance, mais un support supérieur alors elle est ajoutée à la liste de règles de sortie. Une fois que ces étapes sont terminées, FAKIE va convertir les règles d'association en requête CYPHER. Celles-ci sont ainsi directement utilisables sur une base de données NEO4J afin d'effectuer une détection.

Algorithm 2: Sélection des règles les plus pertinentes

Entrée: *liste_regle* : Liste des règles ayant subi un premier filtre

Sortie : *regles_selectionnees* : Une liste contenant les règles d'association sélectionnées

```
1  Début
2      regle_filtre ← []
3      temp ← liste_regle
4      FAIRE
5          #temp est un objet de type Deque. Contient une méthode permettant de
6          #renvoyer et de supprimer directement le premier élément de la liste
           element ← temp.pop() #Supprime et renvoi premier élément
7          POUR CHAQUE regle ∈ liste_regle FAIRE
8              #Appel de la méthode présentée dans l'algorithme suivant
9              SI estMeilleur(element, regle) ALORS
10                 temp.supprimer(regle)
11             FIN
12             SINON
13                 temp.supprimer(element)
14                 element ← regle
15             FIN
16         FIN
17         regle_filtre.ajouter(element)
18     TANT QUE !temp.estVide();
19     retourner regle_filtre
20 Fin
```

Algorithm 3: Comparaison de deux règles, méthode estMeilleur

Entrée: $regle1, regle2$: Règles devant être comparée

Sortie : Un booléen indiquant si la règle 1 est plus pertinente que la règle 2

```
1  Début
2      SI regle1.confiance() > regle2.confiance() ALORS
3          retourner vrai
4      FIN
5      SINON SI regle1.confiance() < regle2.confiance() ALORS
6          retourner faux
7      FIN
8      SINON
9          SI regle1.support() > regle2.support() ALORS
10             retourner vrai
11         FIN
12         SINON SI regle1.support() < regle2.support() ALORS
13             retourner faux
14         FIN
15         SINON
16             SI regle1.premisse.taille() > regle2.premisse.taille() ALORS
17                 retourner vrai
18             FIN
19             SINON
20                 retourner faux
21             FIN
22         FIN
23     FIN
24 Fin
```

CHAPITRE IV

ÉTUDES ET RÉSULTATS

Ce chapitre présente les études effectuées ainsi que leurs résultats afin de confirmer l'efficacité de notre approche outillée, FAKIE, décrite dans le chapitre III. Dans ce chapitre, nous présentons les études effectuées dans le cadre de notre travail. Nous mettrons en avant les questions de recherche soulevées. Ensuite, nous présenterons les résultats puis nous discuterons de ceux-ci.

4.1 Description des études et questions de recherche

Nous avons effectué deux études : une étude de validation ainsi qu'une étude empirique. Le but de la première étude est d'évaluer l'efficacité de notre approche outillée FAKIE. Ainsi il sera possible de confirmer la pertinence des règles de détection générées. Afin d'effectuer notre première étude, nous avons constitué un ensemble de données de défauts de codes. Cet ensemble de données a été constitué par identification manuelle de défauts de code dans du code source. Ces défauts de code proviennent de 30 applications Android à code source ouvert. Nous avons sélectionné les applications en nous basant sur la liste des applications utilisés pour la validation de PAPRIKA et ADOCTO ainsi que des ajouts extérieurs. Le but étant de garantir la présence de défauts de code dans ces applications. De plus, la plupart de ces applications possèdent un très grand nombre d'entités (Wordpress,

Wikipedia). La deuxième étude est une étude empirique effectuée sur un très grand nombre de données. Nous avons appliqué FAKIE sur 2,993 applications Android provenant d'un dépôt d'applications mobiles : ANDROZOO (Allix *et al.*, 2016). L'objectif de cette deuxième étude est d'extraire des nouvelles informations pertinentes à propos des défauts de code. Par nouvelles informations, nous parlons de métriques supplémentaires impliquées dans la présence d'un défaut de code. Cela peut être aussi des co-occurrences entre des défauts de code ou des précisions sur certain défaut de code. Par exemple, la présence d'un défaut de code dans un certain type d'entité plus élevé, etc.

Afin de définir un cadre dans lequel effectuer nos recherches, nous avons soulevé deux questions de recherches, une pour chaque étude. Les sous-sections suivantes présentent ces questions de recherches.

4.1.1 QR1 : Les règles de détection générées par FAKIE sont-elles pertinentes ?

Pour répondre à cette question, nous avons effectué une étude permettant de valider les règles de détections générées. Pour constituer les données de notre étude, nous utilisons 30 applications Android à code source ouvert provenant du dépôt F-DROID [1]. Deux membres de notre équipe de recherche ont identifié manuellement la présence de défauts de code dans le code source de chaque application. Le but de cette analyse manuelle est de constituer un jeu de données dit oracle.

La validation des règles de détection se compose de trois étapes. La première étape consiste en la génération de règles de détection avec FAKIE en se basant sur le jeu de données constitué avec les 30 applications Android. Puis, la deuxième étape consiste à effectuer une détection avec les règles précédemment générées par

1. (En ligne ; Accès Juillet-2018). F-Droid. `https://f-droid.org/`.

FAKIE sur le même ensemble d'applications. Enfin, la troisième étape consiste à comparer les règles de détection générées par FAKIE avec des règles de détection manuellement définies. Nous donnons des détails supplémentaires sur le jeu de données et chacune des trois étapes de la validation par la suite.

Jeu de données. Afin d'effectuer l'identification manuelle des défauts de code présents dans les applications Android, nous utilisons des requêtes NEO4J avec des critères très généraux. Le but est d'obtenir un premier filtre sur les entités de l'application. Puis, une analyse manuelle des applications est effectuée en se basant sur le nom des entités. Par exemple, si un défaut de code ne concerne que les récepteurs de diffusion, la requête est :

```
MATCH (cl :Class) WHERE cl.is_broadcast_receiver = true RETURN cl.name
```

La requête retourne une liste des récepteurs de diffusion présents dans les 30 applications Android. Ensuite, une analyse des métriques de chaque entité (classe ou méthode) de la liste ainsi qu'une analyse du code source sont effectuées dans le but d'identifier la présence d'un défaut de code.

La liste finale est constituée des noms des classes et des méthodes contenant un défaut de code. Par exemple, pour le défaut de code *HAS*, nous avons une liste de nom des méthodes contenant ce défaut de code. À partir de cela, FAKIE va pouvoir constituer un jeu de données en extrayant les informations du graphe NEO4J.

Étape 1. Afin d'effectuer la génération des règles de détection, nous avons appliqué notre méthode outillée FAKIE. Notre méthode va charger le graphe NEO4J en mémoire et va marquer chaque entité contenant un défaut de code. Puis, FAKIE effectue l'étape 2 décrite dans le chapitre III. Cette étape est appliquée sur les 10

Tableau 4.1 Occurrences des défauts de code dans le jeu de données oracle.

Défauts de code	Occurrences	Type
BLOB	309	OO
CC	701	OO
HAS	35	Android
HBR	36	Android
HMU	159	Android
LIC	2043	OO
LM	2345	Android
MIM	1439	Android
NLMR	218	Android
SAK	34	OO

défauts de code présentés dans le chapitre II. Une fois le jeu de données généré, les règles d'associations sont extraites et un filtre est appliqué afin de sélectionner les règles de détection.

Étape 2. Nous avons effectué une détection avec les règles générées dans l'étape 1 sous la forme de requêtes NEO4J. Notre approche est capable de traduire les règles d'associations générées par la librairie WEKA directement en requête NEO4J. Le tableau 4.1 présente les occurrences des défauts de code détectés manuellement dans l'ensemble des 30 applications Android.

Dans le but d'évaluer la pertinence des règles générées, nous utilisons les trois mesures présentées dans le premier chapitre, à savoir : la précision, le rappel et la F-mesure.

Étape 3. Le but de la génération de règles est de réduire la subjectivité des règles de détection. De ce fait, il peut être intéressant de comparer les règles de FAKIE avec celles disponibles dans la littérature. Pour quantifier la différence, nous avons utilisé l'indice de Jaccard (Tan *et al.*, 2006). Cette métrique statistique permet de mettre en avant la similarité entre deux ensembles. Le coefficient de Jaccard (ou l'indice) est le rapport entre l'intersection et l'union de deux ensembles. Cet indice se calcule de cette manière :

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

Le résultat, contenu entre 0 et 1, indique la similarité entre deux ensembles.

Nous avons analysé manuellement le code source de l'outil de détection de Hecht *et al.* (Hecht *et al.*, 2015). Puis, nous en avons extrait les règles de détection afin d'effectuer la comparaison.

4.1.2 QR2 : L'utilisation des règles d'association permet-elle d'extraire des informations intéressantes à propos des défauts de code ?

Lorsque nous générons des règles de détection avec FAKIE, certaines règles ne sont pas sélectionnées en tant que règles de détection. Nous appelons ces règles les *règles périphériques*. Une règle périphérique est dite moins pertinente pour une détection quand celle-ci ne permet pas de détecter le maximum de défauts de code possibles. Les règles d'association sélectionnées pour la détection sont celles ayant le plus haut score de confiance, de support ainsi que le plus grand itemset en prémisse. De ce fait, les règles périphériques sont celles possédant des scores de confiance et de support moindres comparativement aux règles de détection. Ces règles ne permettent pas d'effectuer les détections les plus efficaces. Mais il

est possible que certaines d'entre elles contiennent des informations intéressantes au sujet des défauts de code. Une règle ayant un support de 1 va permettre de détecter l'intégralité des instances de défauts de code du jeu de données utilisé pour la génération. Mais une règle avec un support de 0.8 va mettre en avant une particularité touchant seulement 80% des instances de défauts de code du jeu de données. Il est de ce fait intéressant d'analyser manuellement les règles périphériques.

Nous considérons une règle en tant que *règle périphérique* lorsque celle-ci possède au moins 0.1 en score de support. Ce seuil nous permet de sélectionner les règles d'association concernant au moins 10% des transactions du jeu de données. Notre but ici est d'extraire des informations récurrente et significative. Une règle d'association ne concernant que 1% du jeu de données ne permet pas d'avancer des observations pertinentes. Ce choix s'est aussi fais en nous basant sur les ressources disponibles. Effectivement, plus le seuil de support mininmun est faible, plus la demande en ressource est élevé.

Dans le but de constituer une base de données de défauts de code, nous effectuons une détection avec les règles générées par FAKIE sur 2,993 applications Android. Puis, nous générons des règles d'associations à l'aide de FAKIE sans appliquer la dernière étape qui consiste à filtrer les règles. Pour finir, nous analysons manuellement les règles d'association générées. Le tableau 4.2 présente le nombre d'occurrences des défauts de code détectés dans les applications Android.

Tableau 4.2 Occurrences des défauts de code pour l'étude empirique.

Défauts de code	Occurrences	Type
BLOB	4265	OO
CC	6501	OO
HAS	544	Android
HBR	260	Android
HMU	1440	Android
LIC	26776	OO
LM	49776	Android
MIM	51412	Android
NLMR	3943	Android
SAK	240	OO

4.2 Résultats et discussions

4.2.1 Pertinence des règles et validation

Le tableau 4.3 montre les règles générées par FAKIE. Nous avons sélectionné les règles avec la plus haute valeur de confiance et le plus haute valeur de support afin d'obtenir des règles pertinentes. Le but est d'avoir un maximum de vrai positif lors de la détection. Toutes les règles ont une confiance de 1 et un support de 1. La seule règle ayant un support différent de 1 est la règle pour le défaut de code *HAS* avec un support de 0.96. Ce défaut de code est un peu particulier dans le sens où celui-ci est présent lorsque l'une des méthodes *onPreExecute*, *onPostExecute* et *onProgressUpdate* effectue une tâche lourde à exécuter. De ce fait, la propriété *name* de la méthode peut prendre jusqu'à trois valeurs différentes. Actuellement, FAKIE ne peut pas prendre en compte les valeurs multiples d'une

propriété nominale. Cela est dû à l'algorithme FP-GROWTH. Effectivement, celui-ci génère des règles en se basant sur des transactions et il est de ce fait impossible d'obtenir plusieurs valeurs pour un seul attribut dans une même règle.

Le tableau 4.4 présente les scores obtenus lors de la détection effectuée avec les règles générées par FAKIE. Nous obtenons une F-mesure moyenne de 0.95. Pour les défauts de code *BLOB, CC, LIC, LM, SAK, HMU* et *HBR* nous obtenons une détection très efficace sur notre jeu de données avec des F-mesures égales à 1. La règle obtenue pour *HAS* est pertinente, mais la valeur du rappel est provoquée par la même raison que la valeur du support de la règle de détection. La propriété *name* de la méthode possédant le défaut de code peut prendre trois valeurs et nos règles de détection ne peuvent contenir qu'une seule valeur par propriété. De ce fait, nous n'arrivons pas à détecter l'intégralité des défauts de code *HAS*.

Ensuite, la détection du défaut de code *NLMR* a une précision de 0.66. Lorsque nous utilisons nos règles dans le but de détecter *NLMR*, nous détectons un très grand nombre d'instances. La règle générée par FAKIE ne prend pas en compte la possibilité d'hériter de la méthode *onLowMemory*. Ainsi, nous détectons des faux positifs qui obtiennent la méthode par héritage. Durant la génération des données, FAKIE va ajouter artificiellement les méthodes obtenues par héritage, dont la méthode *onLowMemory*. Cet ajout n'est fais que dans les jeux de données et non directement dans la base de données NEO4J. De ce fait, dans les règles de détection de FAKIE, les méthodes obtenues par héritages sont considérées comme implémenté directement dans la classe. Cela entraîne un manque de précision dans la règle puisque celle-ci ne met pas en avant que la méthode est obtenue par héritage. Il manque donc une information dans la base de données indiquant qu'une classe hérite d'une autre classe. Actuellement, cette information est sous la forme d'une relation et non d'un attribut.

Tableau 4.3 Règles de détection générées.

Défauts de code	**Règles générées**	**Confiance - Support**
BLOB	[NoM>14 = true, NoA>8.5=true, LOC>25=true] ⇒ [class=BLOB]	1 - 1
CC	[CLC>28=trye] ⇒ [class=CC]	1 - 1
HAS	[NoI>17=true, natureOfClass=is_async_task, CC>3.5=true, callExternalMethod=true, NoDC>2.5=true, **name=onPostExecute**] ⇒ [class=HAS]	1 - 0.96
LIC	[isStatic=false, isInnerClass=true] ⇒ [class=LIC]	1 - 1
LM	[NoI>17=true] ⇒ [class=LM]	1 - 1
MIM	[isInit=false, isStatic=false, **NoC>5=false**, callExternalMethod=false, callMethod useVariable=false] ⇒ [class=MIM]	1 - 1
NLMR	[ownOnLowMemory=false, isActivity=true] ⇒ [class=NLMR]	1 - 1
SAK	[isInterface=true, **isAbstract=true**, NoM>14=true] ⇒ [class=SAK]	1 - 1
HMU	[useHashMap=true, callExternalMethod=true] ⇒ [class=HMU]	1 - 1
HBR	[NoI>17=tue, **NoDC>2.5=true**, natureOfClass=is_broadcast_receiver, , CC=high, name=onReceive, **isOverridden=true**, callExternalMethod=true] ⇒ [class=HBR]	1 - 1

Tableau 4.4 Précision, Rappel et F-mesure.

Défauts de code	Précision	Rappel	F-mesure
BLOB	1	1	1
CC	1	1	1
HAS	1	0.92	0.95
LIC	1	1	1
LM	1	1	1
MIM	0.67	1	0.80
NLMR	0.66	1	0.79
SAK	1	1	1
HMU	1	1	1
HBR	1	1	1

Pour finir, la détection pour le défaut de code *MIM* obtient une précision de 0.67, un rappel de 1 et une F-mesure de 0.80. La faible valeur de la précision s'explique par l'utilisation de seuils pour les métriques possédant une valeur continue. Dans notre cas, le défaut de code *MIM* est détecté si la métrique *NoC* est en dessous de la valeur 5. Le problème étant que pour obtenir une meilleure précision, il faudrait prendre en compte la valeur 0. Effectivement, détecter une méthode comme contenant un *MIM* si celle-ci n'est jamais appelée par une entité extérieure ne possède pas grand intérêt. De plus lors de la constitution du jeu de données de validation nous n'avons pas pris en compte les méthodes possédant une valeur de *NoC* égale à 0.

Observation 1 : FAKIE génère des règles pertinentes avec une F-mesure moyenne de 0.95 lors d'une détection.

Pour comparer les règles de détection manuellement définies avec les règles générées part FAKIE, nous utilisons l'index de Jaccard afin de calculer la similarité entre les règles. Les règles définies manuellement sont celles de Hecht *et al.*(Hecht *et al.*, 2015). Nous utilisons l'index de Jaccard pour indiquer les similarités des métriques utilisées dans les règles. Nous ne prenons pas en compte leurs valeurs lorsque celle ci sont des métriques à valeur continus. L'utilisation de seuils

Le tableau 4.5 montre les scores d'index de Jaccard obtenus pour chaque défaut de code. On peut observer que les règles pour les défauts de code *BLOB, LM, HMU* et *LIC* obtiennent un index de Jaccard de 1. En d'autres termes, cela indique que les règles de FAKIE sont exactement les mêmes que celles définies manuellement. Concernant les défauts de code *SAK, NLMR, MIM, HBR* et *HAS*, les scores ne sont pas tout à fait les mêmes. Effectivement, il y a plusieurs différences notables concernant la composition de ces règles de détection. Pour commencer, la règle générée pour *SAK* contient une propriété supplémentaire : *is_abstract*. La présence de cette propriété dans les règles générées avec FP-GROWTH est évidente. Une interface est considérée comme une entité abstraite. De ce fait, il est normal d'obtenir cette propriété supplémentaire. L'ajout de cet élément à la règle de détection n'a aucune influence sur les scores de détections puisqu'une interface est de toute manière abstraite.

D'un autre côté, la règle de détection pour le défaut de code *NLMR* est moins pertinente que celle définie manuellement par (Hecht *et al.*, 2015) La différence principale est liée à l'absence de prise en compte de l'obtention de méthode par héritage comme expliqué précédemment. Encore une fois ce manque d'informations remet en cause la pertinence de la règle générée automatiquement pour le défaut de code *NLMR*.

Tableau 4.5 Score de similarité.

Défauts de code	**Index de Jaccard**
BLOB	1
LM	1
SAK	0.66
HMU	1
NLMR	0.66
MIM	0.84
LIC	1
HBR	0.66
HAS	0.5
CC	1

Ensuite, la règle de détection pour le défaut de code *MIM* est très proche de celle définie manuellement. La seule différence porte sur la valeur de la métrique *NoC* utilisée. La règle de détection définie manuellement indique qu'il faut que la métrique *NoC* soit au moins supérieure à zéro alors que celle générée par FAKIE indique que la métrique doit être inférieure à 5.

Concernant le défaut de code *HBR*, les deux règles sont différentes. La règle générée possède des éléments supplémentaires tels que la propriété *is_overriden* qui indique que la méthode est surchargée ou encore que la méthode appelle des méthodes externes à sa classe. Ces attributs supplémentaires n'ont pas d'influences sur la détection, de plus la méthode *onReceive* est de toute façon surchargée.

Pour finir, les règles pour le défaut de code *HAS* possèdent un score de similarité de 0.5. Cette différence est due à notre jeu de données et à la méthode utilisée pour générer les règles. FAKIE intègre la valeur la plus fréquente d'un attribut lors de

la génération de règle. Ainsi, le nom de la méthode contenant un *HAS* dans notre règle est *onPostExecute*. Cependant le défaut de code est présent quand le nom de la méthode est *onPreExecute*, *onPostExecute*, *onProgressUpdate*. Dans ce cas présent, la règle manuelle prend en compte toutes les possibilités contrairement à celle générée par notre approche outillée.

Observation 2 : Les règles générées sont relativement proches de celles créées manuellement. Il existe tout de même des différences notables pour certains défauts de code, mais ces différences n'ont pas toujours d'influence sur l'efficacité de la détection.

4.2.2 Analyse des informations extraites

Afin de répondre à la deuxième question de recherche : *L'utilisation des règles d'associations permet elle d'extraire des informations intéressantes à propos des défauts de code ?*, nous avons effectué une étude empirique sur 2,993 applications Android. Le tableau 4.6 montre le nombre de règles d'associations générées par FAKIE. Nous observons des différences du nombre de règles d'associations pour chaque défaut de code. Le nombre de règles varie en fonction du nombre de métriques et propriétés qui caractérisent le défaut de code. Par exemple, il est possible d'observer dans le tableau 4.6 que le défaut de code *HBR* possède un très grand nombre de règles. Cela indique de ce fait qu'un très grand nombre de propriétés ou de métriques caractérisent ce défaut de code. Parmi toutes les règles générées, nous effectuons une sélection dans le but de conserver celles pouvant potentiellement nous fournir des informations intéressantes. Cette sélection se fait selon quatre critères, à savoir :

Tableau 4.6 Occurrences des règles d'association par défaut de code.

Défauts de code	**Occurrences**
BLOB	4210
CC	1638
HAS	4210
HBR	38342
HMU	1330
LIC	10
LM	3644
MIM	38
NLMR	4118
SAK	38

— Encore une fois, les règles doivent être sous la forme *ManyToOne*. *One* étant l'attribut indiquant la présence d'un défaut de code.
— La règle sélectionnée à un même score de support et de confiance doit être celle contenant le plus grand nombre d'item en prémisse.
— Le score de support doit être au moins supérieur à 0.1.
— La règle ne doit pas être une règle de détection.

Le tableau 4.7 présente les règles sélectionnées manuellement. Certains défauts de code ne sont pas présents dans le tableau puisqu'aucune règle périphérique ne dépassait le score de 0.1 de support.

1. ***BLOB*** : nous observons qu'une très grande proportion de *BLOB* possède une complexité supérieure à 28. Une complexité supérieure à 28 indique la présence du défaut de code *CC*. Ce résultat a du sens étant donné

Tableau 4.7 Règles périphériques.

Règles	Confiance	Support	Défauts de code
[NoM>14=true, NoA>8.5=true, LOC>25=true, **CLC>28=true**] ⇒ [class=BLOB]	1	0.81	*BLOB*
[NoM>14=true, NoA>8.5=true, LOC>25=true, **DoI>5=true**] ⇒[class=BLOB]	1	0.48	*BLOB*
[NoM>14=true, NoA>8.5=true, LOC>25=true, **DoI>5=true, isActivity=true**] ⇒ [class=BLOB]	1	0.44	*BLOB*
[NoM>14=true, NoA>8.5=true, LOC>25=true, **NoII>2=true**] ⇒ [class=BLOB]	1	0.19	*BLOB*
[NoM>14=true, NoA>8.5=true, LOC>25=true, **CLC>28=true, DoI>5=true**] ⇒ [class=BLOB]	1	0.41	*BLOB*
[NoM>14=true, NoA>8.5=true, LOC>25=true, **CLC>28=true, DoI>5=true, isActivity=true**] ⇒ [class=BLOB]	1	0.38	*BLOB*
[CLC>28=true, **LOC>25=true**] ⇒ [class=CC]	1	0.79	*CC*
[CLC>28=true, **NoM>14=true**] ⇒ [class=CC]	1	0.76	*CC*
[CLC>28=true, **NoA>8.5=true**] ⇒ [class=CC]	1	0.69	*CC*
[CLC>28=true, **DoI>5=true**] ⇒ [class=CC]	1	0.39	*CC*
[CLC>28=true, **DoI>5=true, isActivity=true**] ⇒ [class=CC]	1	0.34	*CC*
[CLC>28=true, **LOC>25=true, NoM>14=true, NoA>8.5=true**] ⇒ [class=CC]	1	0.57	*CC*
[NoI>26=true, natureOfClass=is_async_task, CC>5=true, callExternalMethod=true, name=OnPostExecute, NoDC>4=true,**callInit=true** ⇒ [class=HAS]]	1	0.82	*HAS*
[NoI>26=true, name=OnPostExecute, NatureOfClass=is_async_task, CC>5=true, **callExternalMethod=true, NoDC>4=true, NoDL>9=true**] ⇒ [class=HAS]	1	0.76	*HAS*
[useHashMap=true, **callInit=true** , callExternalMethod=true, **NoI>26=true**] ⇒ [class=HMU]	1	0.58	*HMU*
[useHashMap=true, **callInit=true**, callExternalMethod=true, **CC>5=true**] ⇒ [class=HMU]	1	0.37	*HMU*
[NoI>26=true,**CC>5=true**] ⇒ [class=LM]	1	0.9	*LM*
[NoI>26=true, **isOverridden=true**] ⇒ [class=LM]	1	0.42	*LM*

qu'un grand nombre d'attributs, un grand nombre de méthodes ainsi qu'un manque de cohésion entre les méthodes sont des facteurs augmentant la complexité d'une classe. Nous en concluons que 81% des instances de *BLOB* possèdent aussi le défaut de code *CC*. De plus, un grand nombre de *BLOB* possède une profondeur d'héritage supérieur à 5. Cela nous indique que 48% des *BLOB* héritent d'une grande partie de leurs méthodes et attributs. Pour finir, la troisième règle du tableau 4.7 indique qu'un très grand nombre de *BLOB* sont en fait des activités Android.

2. ***CC*** : de manière triviale, les classes infectées par le défaut de code *CC* possèdent potentiellement un manque de cohésion entre les méthodes, un grand nombre de méthodes ou encore un grand nombre d'attributs (respectivement 79%, 76% et 69%). Cela appuie les observations relevées concernant le défaut de code *BLOB*. De plus on observe aussi que 38% des instances de *CC* sont des activités Android. Pour finir, la dernière règle du tableau 4.7 concernant les règles périphériques de *CC* indique encore une fois la co-occurrence des *BLOB* et des *CC*.

3. ***HAS*** : pour ce défaut de code, nous pouvons observer qu'une majorité des méthodes contenant *HAS* possèdent un grand nombre d'appels directs et font appel à des constructeurs.

4. ***HMU*** : les règles périphériques pour le défaut de code *HMU* mettent en avant plusieurs choses. Tout d'abord 58% des méthodes contenant ce défaut de code possèdent un nombre d'instructions supérieur à 26. Ensuite, 37% possèdent une complexité cyclomatique supérieure à 5. Ces informations indiquent que le défaut de code *HMU* aurait tendance à être présent lorsque les méthodes sont très complexes et possèdent un grand nombre d'instructions. Le fait est qu'un grand nombre d'instructions indique aussi

la présence du défaut de code *LM*. Pour conclure, nous observons une co-occurrence récurrente entre les défauts de code *HMU* et *LM* (58%).

5. ***LM*** **:** un très grand nombre des méthodes contenant le défaut de code *LM* possèdent une grande complexité cyclomatique (90%). Ce n'est pas surprenant puisqu'un grand nombre d'instructions est un facteur augmentant potentiellement la complexité cyclomatique.

Pour synthétiser, certaines règles périphériques mettent en avant des tendances concernant quelques défauts de code. Comme expliqué précédemment, 44% des *BLOB* sont présents dans des activités Android alors que ces entités représentent seulement 7% de toutes nos classes de notre jeu de données. La co-occurrence entre *BLOB* et *CC* est mise en avant. Enfin, les co-occurrences des défauts de code *LM* et *HMU* sont mises en avant par ces règles périphériques.

Observation 3.1 : *BLOB* et *CC* co-occurrent très souvent ensemble (81% des instances de *BLOB* possèdent aussi *CC*).
Observation 3.2 : 44% des *BLOB* sont contenus dans des activités Android alors qu'elles ne représentent que 7% des entités du jeu de données.
Observation 4 : Le défaut de code *HMU* co-occurre de manière significative avec *LM*(58%).

4.3 Réponses aux questions de recherche

Pour répondre aux questions de recherche précédemment énoncées, nous nous basons sur les observations de la section 4.2.

QR1 : *Les règles de détection générées par* FAKIE *sont-elles pertinentes ?*

Le tableau 4.4 indique les résultats d'une détection faite avec les règles générées par FAKIE. L'obtention d'une moyenne de 0.95 pour la F-mesure souligne la pertinence de nos règles. De plus, le tableau 4.5 indique une certaine similarité entre les règles définies manuellement par hecht2015tracking et les règles de FAKIE. L'outil développé par Hecht *et al.* (Hecht *et al.*, 2015) étant considéré comme efficace, nous pouvons affirmer que nos règles le sont aussi. Ces deux observations confirment la pertinence des règles générées par notre approche outillée FAKIE.

QR2 : *L'utilisation des règles d'associations permet elle d'extraire des informations intéressantes à propos des défauts de code ?*

Le tableau 4.7 présente une partie des règles périphériques obtenues à l'aide de FAKIE. À partir ces règles, nous avons extrait des tendances de développement telles que : *BLOB* et *CC* co-occurrent de manière significative ensemble (81%). 44% des *BLOB* sont présents dans les activités Android. Enfin, *HMU* est régulièrement contenu dans des méthodes infectées également par *LM* (58%). Ces observations permettent de dire que l'utilisation des règles d'association sur des données en rapport avec les défauts de code permet d'extraire de l'information.

4.4 Menaces à la validité

Dans cette section, nous présentons les différentes menaces à la validité concernant nos études. Nous nous basons sur la classification de (Hyman, 1982).

Validité interne : l'une des menaces majeures est provoquée par le jeu de données de validation. Le but principal de FAKIE est de diminuer la subjectivité des règles de détection. Cependant, il n'existe pas à ce jour de jeux de données publiques contenant des instances de défauts de code. De ce fait, nous avons été dans l'obligation de constituer un jeu de données. La création manuelle d'un jeu de données peut ajouter de la subjectivité puisque la présence d'un défaut de

code dépend de la compréhension de la définition de celui-ci. De plus, certains faux positifs peuvent être ajoutés au jeu de données. Afin d'essayer de minimiser l'impact de ces menaces, deux développeurs ont identifié les défauts de code sous les mêmes conditions. Cela permet d'avoir une double confirmation pour chaque instance de défaut de code. Concernant la présence de faux positifs, l'utilisation de l'algorithme FP-GROWTH permet de réduire l'impact de ceux-ci lors de la génération de règles. Effectivement, comme expliqué dans la section 1.4.4, FP-GROWTH est un algorithme basé sur la fréquence des items d'un jeu de données. De ce fait, un faux positif qui est considéré comme une instance de données avec des valeurs atypiques sera mise de côté naturellement par l'algorithme.

Validité externe : la menace principale ici est la représentativité de nos résultats. Effectivement, nous avons effectué notre validation sur seulement 30 applications Android. Un plus grand nombre d'applications auraient été favorables pour la validation de notre méthode outillée FAKIE. Néanmoins, la construction d'un jeu de données est un processus long et du à des contraintes techniques nous n'avons pas constitué un jeu de données plus conséquent. Toutefois, nous avons tout de même sélectionné des applications contenant un très grand nombre de classes, ce qui augmente la probabilité d'obtenir un grand nombre d'instances de défaut de code.

Validité de construction : comme expliqué dans la section 3.3, nous utilisons une liste de méthodes et d'appels de méthodes pour générer notre jeu de données. Initialement toutes les méthodes appelées ou déclarées étaient ajoutées au jeu de données. Nous obtenions un jeu de données contenant plus de 1000 attributs différents. Due à des contraintes techniques, il n'était pas possible de traiter de tels jeux de données. Effectivement, la génération des données pouvait prendre jusqu'à plusieurs jours voir semaine. De plus, lors de l'application de l'algorithme FP-GROWTH, le jeu de données était beaucoup trop dense et il était impossible

d'extraire des règles d'association. De ce fait nous avons réduit le nombre d'attributs en utilisant une liste de méthode à conserver. Cette façon de faire permet de réduire la taille du jeu de données et permet d'utiliser FAKIE dans un temps raisonnable d'exécution. L'utilisation de cette liste de méthode induit automatiquement une perte d'informations dans notre jeu de données et peut être qu'une méthode non référencée dans la littérature est un facteur important concernant la présence d'un défaut de code.

CONCLUSION

La détection de défauts de code est une pratique permettant d'augmenter la qualité globale d'une application ou d'un système. La qualité d'une application est particulièrement importante en développement d'application mobile en raison des ressources limitées des périphériques mobiles. Une des autres contraintes très importantes du développement d'applications mobile est le temps de conception. Le temps de conception d'une application mobile est en constante diminution pour pouvoir répondre à la demande du marché de plus en plus demandeur. Ces facteurs incitent les développeurs à faire de mauvais choix de conception qui conduisent à la présence de défauts de code dans les systèmes. Il n'existe à ce jour que peu de recherches axées sur les défauts de code Android. De plus, les outils de détection existants utilisent des règles de détection définies manuellement. Afin de diminuer la subjectivité des règles de détection, nous avons choisi de développer une approche permettant de générer des règles automatiquement.

Notre approche outillée FAKIE permet de générer des règles de détection de défauts de code dans les applications Android. Avec cette approche, nous avons pu générer des règles pour 10 défauts de code différents, à savoir : *BLOB, LM, CC, HMU, HAS, HBR, LIC, NLMR, MIM, SAK.* FAKIE va analyser des instances de défauts de code et utiliser l'algorithme FP-GROWTH afin de générer des règles d'association. Puis, les règles sont filtrées afin d'obtenir les règles de détection les plus pertinentes.

Afin de valider notre approche, nous avons effectué une étude sur 30 applications Android à code source ouvert. Nous avons généré les règles de détection de 10

défauts de code à partir d'instances de défauts identifiées dans ces applications. Puis, nous avons effectué une détection de défauts de code avec les règles précédemment inférées et nous avons obtenu une F-mesure moyenne de 0.95. Ces résultats confirment la pertinence de la génération de règles avec notre approche outillée FAKIE. Nous avons également effectué une étude empirique sur 2,993 applications Android dans le but d'identifier des informations intéressantes à l'aide des règles d'association. Cette étude nous a permis d'extraire des tendances de co-occurrence ou de développement. Par exemple, 44% des *BLOB* et 34% des *CC* sont présents dans des activités Android. Cela met en avant la tendance des développeurs à créer des Activités trop complexes et possédant un trop grand nombre de responsabilités.

Concernant les travaux futurs, nous avons pour but d'ajouter des métriques et des propriétés au modèle afin d'obtenir un jeu de données plus complet lors de l'analyse de la base de données NEO4J. Nous projetons de générer des règles de détection pour des défauts de code supplémentaires. Il serai intéressant d'appliquer notre méthode sur des applications Java classique afin d'obtenir des résultats sur les défauts de code spécifiques aux applications OO classique. L'utilisation d'un cluster de calculs pourrai permettre d'augmenter le nombre de méthode prises en compte lors de la génération de règles. Cela pourrai permettre d'extraire des informations importantes que nous n'avons pas pu obtenir dû à nos limitations technique. Finalement, nous espérons que cette recherche permettra de faciliter la création de règles de détection de défauts de code. La finalité étant d'améliorer la qualité générale des applications Android.

APPENDICE A

RÈGLES EXTRAITE POUR LA COMPARAISON

Listing A.1 Règle de détection pour le NLMR dans PAPRIKA

```
String query = "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.app_key as app_key";
```

Listing A.2 Règle de détection pour le HMU dans PAPRIKA

```
String query = "MATCH
    (m:Method)-[:CALLS]->(e:ExternalMethod{full_name:'<init>#java.util.HashMap'})
    return m.app_key";
```

Listing A.3 Règle de détection pour le LIC dans PAPRIKA

```
String query = "MATCH (cl:Class) WHERE HAS(cl.is_inner_class) AND NOT
    HAS(cl.is_static) RETURN cl.app_key as app_key";
```

Listing A.4 Règle de détection pour le MIM dans PAPRIKA

```
String query = "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 (m1)-[:CALLS]->(:ExternalMethod) AND
    NOT (m1)-[:CALLS]->(:Method) AND NOT HAS(m1.is_init) RETURN
    m1.app_key as app_key";
```

Listing A.5 Règle de détection pour le SAK dans PAPRIKA

```
String query = "MATCH (cl:Class) WHERE HAS(cl.is_interface) AND
    cl.number_of_methods > " + veryHigh + " RETURN cl.app_key as
    app_key";
```

Listing A.6 Règle de détection pour le LM dans PAPRIKA

```
String query = "MATCH (m:Method) WHERE m.number_of_instructions >" +
    veryHigh + " RETURN m.app_key as app_key";
```

Listing A.7 Règle de détection pour le HBR dans PAPRIKA

```
String query = "MATCH
    (c:Class{is_broadcast_receiver:true})-[:CLASS_OWNS_METHOD]->(m:Method{name:'onReceive'})
    WHERE m.number_of_instructions > "+veryHigh_noi+" AND
    m.cyclomatic_complexity>"+veryHigh_cc+" return m.app_key as app_key";
```

Listing A.8 Règle de détection pour le HAS dans PAPRIKA

```
String query = "MATCH
    (c:Class{parent_name:'android.os.AsyncTask'})-[:CLASS_OWNS_METHOD]->(m:Method)
    WHERE (m.name='onPreExecute' OR m.name='onProgressUpdate' OR
    m.name='onPostExecute') AND m.number_of_instructions
    >"+veryHigh_noi+" AND m.cyclomatic_complexity > "+veryHigh_cc+"
    return m.app_key as app_key";
```

Listing A.9 Règle de détection pour le BLOB dans PAPRIKA

```
String query = "MATCH (cl:Class) WHERE cl.lack_of_cohesion_in_methods >"
    + veryHigh_lcom + " AND cl.number_of_methods > " + veryHigh_nom + "
    AND cl.number_of_attributes > " + veryHigh_noa + " RETURN cl.app_key
    as app_key";
```

Listing A.10 Règle de détection pour le CC dans PAPRIKA

```
String query = "MATCH (cl:Class) WHERE cl.class_complexity > "+ veryHigh
    +" RETURN cl.app_key as app_key";
```

RÉFÉRENCES

Aggarwal, K., Singh, Y., Kaur, A. et Malhotra, R. (2009). Empirical analysis for investigating the effect of object-oriented metrics on fault proneness : a replicated case study. *Software process : Improvement and practice*, *14*(1), 39–62.

Agrawal, R., Imieliński, T. et Swami, A. (1993). Mining association rules between sets of items in large databases. Dans *Acm sigmod record*, volume 22, 207–216. ACM.

Agrawal, R., Srikant, R. *et al.* (1994). Fast algorithms for mining association rules. Dans *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, 487–499.

Allix, K., Bissyandé, T. F., Klein, J. et Le Traon, Y. (2016). Androzoo : Collecting millions of android apps for the research community. Dans *Proceedings of the 13th International Conference on Mining Software Repositories*, 468–471. ACM.

Bartel, A., Klein, J., Le Traon, Y. et Monperrus, M. (2012). Dexpler : converting android dalvik bytecode to jimple for static analysis with soot. Dans *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, 27–38. ACM.

Basili, V. R., Briand, L. C. et Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, *22*(10), 751–761.

Brown, W. H., Malveau, R. C., McCormick, H. W. et Mowbray, T. J. (1998). *AntiPatterns : refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

Carette, A., Younes, M. A. A., Hecht, G., Moha, N. et Rouvoy, R. (2017). Investigating the energy impact of android smells. Dans *24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER)*, p. 10. IEEE.

Chidamber, S. R. et Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, *20*(6), 476–493.

Ciupke, O. (1999). Automatic detection of design problems in object-oriented reengineering. Dans *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30 Proceedings*, 18–32. IEEE.

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A. et De Lucia, A. (2018). Detecting code smells using machine learning techniques : are we there yet ? Dans *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 612–621. IEEE.

Fontana, F. A., Mäntylä, M. V., Zanoni, M. et Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, *21*(3), 1143–1191.

Fontana, F. A., Zanoni, M., Marino, A. et Mantyla, M. V. (2013). Code smell detection : Towards a machine learning-based approach. Dans *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, 396–399. IEEE.

Fowler, M., Beck, K., Brant, J., Opdyke, W. et Roberts, D. (1999). *Refactoring : improving the design of existing code.* Addison-Wesley Professional.

Goutte, C. et Gaussier, E. (2005). A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. Dans *European Conference on Information Retrieval*, 345–359. Springer.

Győrödi, C., Győrödi, R. et Holban, S. (2004). A comparative study of association rules mining algorithms. Dans *Hungarian Joint Symposium on Applied Computational Intelligence, Oradea.*

Haase, C. (2015). Developing for android, i : Understanding the mobile context. `https://goo.gl/KUN6XC`. [En ligne ; Accès Janvier-2018].

Han, J., Pei, J. et Yin, Y. (2000). Mining frequent patterns without candidate generation. Dans *ACM sigmod record*, volume 29, 1–12. ACM.

Hecht, G., Benomar, O., Rouvoy, R., Moha, N. et Duchien, L. (2015). Tracking the software quality of android applications along their evolution (t). Dans *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, 236–247. IEEE.

Hyman, R. (1982). Quasi-experimentation : Design and analysis issues for field settings (book). *Journal of Personality Assessment*, *46*(1), 96–97.

Jiang, D., Ma, P., Su, X. et Wang, T. (2014). Distance metric based divergent change bad smell detection and refactoring scheme analysis. *Proc. of the 26th Int. Journal of Innovative Computing, Information and Control (ICIC'2014)*, *10*(4).

Kessentini, M. et Ouni, A. (2017). Detecting android smells using multi-objective genetic programming. Dans *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, 122–132. IEEE Press.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G. et Sahraoui, H. (2011). Bdtex : A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, *84*(4), 559–572.

Kim, D. K. (2017). Finding bad code smells with neural network models. *International Journal of Electrical and Computer Engineering (IJECE)*, *7*(6), 3613–3621.

Kothari, S. C., Bishop, L., Sauceda, J. et Daugherty, G. (2004). A pattern-based framework for software anomaly detection. *Software Quality Journal*, *12*(2), 99–120.

Kumar, B. S. et Rukmani, K. (2010). Implementation of web usage mining using apriori and fp growth algorithms. *Int. J. of Advanced networking and Applications*, *1*(06), 400–404.

Li, W. et Shatnawi, R. (2007). An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software*, *80*(7), 1120–1128.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R. et Poshyvanyk, D. (2013). Api change and fault proneness : a threat to the success of android apps. Dans *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 477–487. ACM.

Lockwood, A. (2013). How to leak a context : Handlers & inner classes.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Gueheneuc, Y.-G. et Aimeur, E. (2012a). Smurf : A svm-based incremental anti-pattern detection approach. Dans *Reverse engineering (WCRE), 2012 19th working conference on*, 466–475. IEEE.

Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y.-G., Antoniol, G. et Aïmeur, E. (2012b). Support vector machines for anti-pattern

detection. Dans *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 278–281. ACM.

Mäntylä, M. V. et Lassenius, C. (2006). Subjective evaluation of software evolvability using code smells : An empirical study. *Empirical Software Engineering*, *11*(3), 395–431.

Marinescu, R. (2004). Detection strategies : Metrics-based rules for detecting design flaws. Dans *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, 350–359. IEEE.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4), 308–320.

Moha, N., Gueheneuc, Y.-G., Duchien, L. et Le Meur, A.-F. (2010). Decor : A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, *36*(1), 20–36.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D. et De Lucia, A. (2015). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, *41*(5), 462–489.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A. et De Lucia, A. (2017a). Lightweight detection of android-specific code smells : The adoctor project. Dans *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, 487–491. IEEE.

Palomba, F., Oliveto, R. et De Lucia, A. (2017b). Investigating code smell co-occurrences using association rule learning : A replicated study. Dans *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), IEEE Workshop on*, 8–13. IEEE.

Reimann, J., Brylski, M. et Aßmann, U. (2014). A tool-supported quality smell catalogue for android developers. Dans *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung-MMSM*, volume 2014.

Riel, A. J. (1996). *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc.

Singh, Y., Kaur, A. et Malhotra, R. (2010). Empirical validation of object-oriented metrics for predicting fault proneness models. *Software quality journal*, *18*(1), 3.

Suryanarayana, G., Samarthyam, G. et Sharma, T. (2014). *Refactoring for software design smells : managing technical debt*. Morgan Kaufmann.

Tan, P.-N. *et al.* (2006). *Introduction to data mining.* Pearson Education India.

Tukey, J. W. (1977). *Exploratory data analysis*, volume 2. Reading, Mass.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. et Sundaresan, V. (2010). Soot : A java bytecode optimization framework. Dans *CASCON First Decade High Impact Papers*, 214–224. IBM Corp.

Van Emden, E. et Moonen, L. (2002). Java quality assurance by detecting code smells. Dans *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, 97–106. IEEE.

Verloop, D. (2013). Code smells in the mobile applications domain.

Wasserman, A. I. (2010). Software engineering issues for mobile application development. Dans *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 397–400. ACM.

Witten, I. H., Frank, E., Hall, M. A. et Pal, C. J. (2016). *Data Mining : Practical machine learning tools and techniques.* Morgan Kaufmann.

Xie, G., Chen, J. et Neamtiu, I. (2009). Towards a better understanding of software evolution : An empirical study on open source software. Dans *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 51–60. IEEE.

Yamashita, A. et Moonen, L. (2012). Do code smells reflect important maintainability aspects ? Dans *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, 306–315. IEEE.

Yamashita, A. et Moonen, L. (2013). Exploring the impact of inter-smell relations on software maintainability : An empirical study. Dans *Software Engineering (ICSE), 2013 35th International Conference on*, 682–691. IEEE.