UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ÉTUDE DES PATRONS ARCHITECTURAUX DE TYPE MVC DANS LES APPLICATIONS ANDROID

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

AYMEN DAOUDI

DÉCEMBRE 2018

REMERCIEMENTS

Je remercie en premier lieu, Allah de m'avoir permis de mener ce travail de recherche à terme.
Je remercie mes parents, sans leur soutien je ne pourrais guère réaliser ce travail. Je tiens aussi à remercier ma directrice de recherche Prof. Naouel Moha, professeure au département d'informatique. Son soutien, ses conseils et son encadrement m'ont vraiment aidé à achever ce travail dans les meilleurs délais malgré la charge de travail importante. À cet égard elle m'a accordé une grande attention et une présence sans faille tout au long de ces deux années de maîtrise. Je désire aussi remercier ma co-directrice Prof. Ghizlane El Boussaidi, professeure à l'École de Technologies Supérieures de Montréal pour ses orientations et son expertise remarquable. Elle m'a beaucoup aidé à surmonter des défis à la fois d'ordre techniques et d'organisation du travail. Je remercie également le professeur Sègla Jean-Luc Kpodjedo de l'École de Technologies Supérieures pour son aide et ses conseils pertinents.

Je ne peux oublier de remercier mon ami Oualid Boutemine, ancien étudiant de l'UQAM et maître en informatique distingué, qui m'a encouragé et supporté sans hésitation tout au long de ma maîtrise.

J'adresse aussi mes remerciements à l'Université du Québec À Montréal ainsi que son personnel administratif et scientifique, pour l'opportunité qu'elle m'a offerte pour réaliser ce travail, sans oublier tous les professeurs qui m'ont enseigné et tous mes collègues du laboratoire LATECE pour les bons moments que nous avons passé ensemble.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

APK	Android Package Kit
SDK	Software Development Kit (Trousse de développement logiciel)
ART	Android Runtime
API	Application Programming interface (Interface de programmation des Applications)
MVC	Modèle Vue Contrôleur
MVP	Modèle Vue Présentateur
MVVM	Modèle Vue VueModèle
UML	Unified Modeling Language (Langage de Modélisation Unifié)
MARPLE	Metrics and Architecture Reconstruction Plug-in for Eclipse
CI	Classe d'Interaction
CM	Classe Modèle
MOR	Mappage Objet-Relationnel
CLD	Classe de Liaison-De-Données
EIU	Élément Interface Utilisateur
CU	Classe utilitaire
EEE	Évènement d'Entrée à bases d'Écouteurs
LOC	Line Of Code (Ligne de code)

RÉSUMÉ

Le développement des applications mobiles représente maintenant un segment important de l'industrie des logiciels, avec Android étant le plus grand écosystème des applications mobiles. Le développement Android, comme tous les autres types de développement, vient avec ses propres patrons et composantes (mises en page, Activités, etc.). Les développeurs utilisent également d'autres patrons architecturaux établis pour concevoir des logiciels interactifs tels comme MVC, MVP et MVVM. Ils implémentent ces patrons en se basant sur leur compréhension et expérience. Ainsi, la mise en œuvre de tels patrons varie d'un développeur à un autre et rend le choix d'un patron particulier un sujet de débat. À notre connaissance, il n'existe aucun travail qui analyse la façon et la fréquence d'utiliser ces patrons dans les applications mobiles. De plus, il n'y a pas une compréhension claire quand à la distinction du patron qui est le plus adapté ou sur la tendance générale de conception des applications mobiles au moyen de tels patrons. Dans ce mémoire, nous proposons une approche automatique pour identifier le patron architectural basé sur MVC (MVC, MVP et MVVM) qui est utilisé principalement dans une application Android donnée. À cette fin, nous avons défini chacun de ces patrons à travers un certain nombre d'heuristiques selon les implémentations potentielles de chaque patron au sein de la plateforme Android. Nous avons mené une étude empirique sur un grand nombre d'applications mobiles téléchargées à partir de Google Play Store. Nous avons trouvé une dominance du patron populaire MVC, tandis que MVP est rarement utilisé et MVVM est presque inutilisé. Nous avons aussi observé qu'un nombre important d'applications ne suit aucun patron. L'étude empirique nous a également permis d'analyser l'utilisation de ces patrons par catégorie, taille et date de dernière mise à jour. Nous avons observé que MVC a été le modèle le plus utilisé au cours des dernières années et il continue à gagner en popularité. De plus, nous avons constaté qu'il n'existe aucune relation apparente entre le choix d'un patron spécifique et la catégorie d'une application et que les applications de petites tailles sont principalement celles qui n'utilisent aucun patron.

MOTS CLES : Android, applications mobiles, détection, patrons, architecture, couche présentation, MVC, MVP, MVVM.

INTRODUCTION

Le téléphone intelligent (en anglais, *SmartPhone*) est l'une des plus grandes innovations technologiques des vingt dernières années. Ce périphérique a une multitude de facteurs de forme (*form factors*) et fonctionne sous différents systèmes d'exploitation. Les applications fonctionnant sur ce type d'appareils représentent incontestablement un marché en nette croissance, avec une portée qui dépasse sans doute celle des applications bureautiques. La grande majorité de ces applications fonctionne sous le système Android, qui s'est imposé comme la plateforme mobile prédominante avec une part de marché qui est estimée maintenant à environ 88%[1]. Dans le cadre de nos recherches, nous avons opté pour ce système d'exploitation vu sa maturité, son large usage et son franc succès chez les développeurs et les utilisateurs.

Historiquement, jusqu'à l'année 2007, Windows Mobile (Microsoft), Blackberry OS (BlackBerry) et Symbian (Nokia) étaient les plateformes dominantes pour la téléphonie mobile intelligente. Elles visaient au début le marché professionnel, avant de se diriger vers les consommateurs finaux. Ces systèmes offraient une interaction à travers le clavier ou le stylet. Apple en 2007 a introduit son propre téléphone intelligent : le iPhone. Basé sur son système d'exploitation iOS et doté d'un écran tactile capacitif multi-touches permettant une interaction plus fluide avec les applications. Dans la même année, Google a commencé à promouvoir son système d'exploitation Android, qui est basé sur le noyau du système

1. Source : https://www.statista.com/statistics/266136/global-market-share-held-bysmartphone-operating-systems/

d'exploitation Linux et qui ciblait spécifiquement les appareils mobiles. Le système iOS d'Apple est propriétaire et exclusivement utilisé pour ses appareils iPhone. Tandis que le système Android est gratuit, à code source ouvert et offert à tous les fabricants des téléphones intelligents, ce qui a donné un grand avantage à ce dernier. Ces deux systèmes ont connu un succès remarquable grâce à la forte croissance du nombre d'applications ainsi que le nombre de développeurs et de concepteurs s'intéressant à ces plateformes. Chaque système d'exploitation mobile est généralement accompagné par une plateforme de développement qui lui est dédiée et sur laquelle les développeurs se basent pour concevoir des applications mobiles. De plus, certaines entreprises proposent des solutions unifiées et plus génériques qui peuvent être utilisées pour développer des applications pour différents systèmes tout en se basant sur une seule plateforme de développement commune.

Les applications mobiles sont généralement développées au moyen de langages de programmation spécifiques tels que Java, Objective-C, C++, C#, ou encore C, Swift et Kotlin, et qui diffèrent d'une plateforme à une autre. En effet, le développement des applications mobiles est différent du développement des applications traditionnelles (Wasserman, 2010). Les appareils mobiles viennent généralement avec plus de contraintes que les ordinateurs de bureau en termes de mémoire, de batterie, de puissance de calcul ou de concurrence pour les ressources, etc. Pour faire face à ces contraintes, la plateforme Android propose un certain nombre de mécanismes et de composants architecturaux que les développeurs doivent employer pour concevoir des applications de haute qualité. Les développeurs et les concepteurs Android doivent exploiter ces mécanismes et concepts et décider comment les organiser afin de répondre aux exigences fonctionnelles (ou non fonctionnelles) de leurs applications.

L'impact de ces choix sur la qualité d'une application ne peut pas être négligé.

En particulier, pour les applications interactives incluant les applications mobiles, des patrons bien établis dans la couche présentation tels que le Modèle Vue Contrôleur (MVC) et ses variantes telles que Modèle Vue Présentateur (MVP) et le Modèle Vue VueModèle (MVVM) deviennent particulièrement pertinents. Ces patrons proposent généralement trois composantes principales pour développer des applications, une composante représentant l'interface utilisateur (la Vue), une composante représentant la logique d'affaire (le Modèle) et une composante intermédiaire qui lie entre les deux précédentes. Cette décomposition permet de construire des applications facilement maintenables.

MVC, en terme de manière d'implémentation, n'est pas un sujet consensuel pour les chercheurs et les développeurs Android. Certains affirment même que la plateforme Android elle-même est favorable aux anti-patrons MVC (Bagheri *et al.*, 2016). Dans tous les cas, pour les développeurs, il y a une certaine ambiguïté quant aux composants Android qui doivent assumer le rôle du Contrôleur et ceux qui doivent assumer celui de la Vue (StackOverflow, 2018). De plus, il y a un débat sur le choix du patron basé sur MVC le plus adapté et la manière de l'implémenter : le MVC original, le MVP, ou bien le MVVM (Lou, 2016).

Bien que ces patrons soient bien réputés et largement discutés dans la communauté des développeurs Android (StackOverflow, 2018; Maxwell Eric, 2017), il n'existe aucun consensus autour de leur utilisation. En fait, il y a même un débat sur quel patron (entre MVC, MVP et MVVM) est le mieux adapté pour les applications Android. (StackOverflow, 2018; Maxwell Eric, 2017; Musselwhite, 3 01; Lou, 2016). Cependant, il n'y a pas de données ou d'études disponibles qui aident pour décider quel patron convient à quel type d'applications. En outre, chacun de ces patrons peut être implémenté de différentes manières dans la plateforme Android. En fait, pour les développeurs, il existe une certaine ambiguïté, quand à quels composants Android doivent être considérés comme des Vues ou des Contrôleurs.

Dans ce contexte, il est nécessaire de mener des études analysant la manière dont les patrons basés sur MVC sont implémentés dans les applications Android et de fournir des données sur les patrons utilisés dans ces applications. Ces données sont essentielles pour fournir un aperçu des tendances en matière de conception dans les applications mobiles et pour développer spécifiquement une meilleure compréhension de la relation entre les caractéristiques des applications Android et les patrons basés sur MVC.

Dans la littérature, il existe des travaux récents (Lou, 2016; Johnson, 2018; La et Kim, 2010) qui rapportent des analyses manuelles des patrons basés sur MVC dans les applications mobiles. La et Kim rapportent une étude sur la reconstruction d'architectures d'applications mobiles (La et Kim, 2010). Cependant, à notre connaissance, il n'existe aucun travail qui se rapporte spécifiquement à l'identification automatique des patrons de conception de la couche présentation dans les applications mobiles, à l'exception des travaux traitant les anti-patrons et les défauts de code (Hecht *et al.*, 2015; Palomba *et al.*, 2017; Reimann *et al.*, 2014; Kessentini et Ouni, 2017).

Dans ce contexte, le présent travail vise à répondre aux questions de recherche suivantes :

- **RQ1 :** Quelle est la fréquence d'utilisation des patrons basés sur MVC dans les applications mobiles Android ?
- **RQ2 :** Quels sont les types d'applications Android qui utilisent les patrons basés sur MVC ?

Pour répondre à ces questions, nous proposons une approche dénommée RIMAZ qui vise à identifier le patron basé sur MVC dominant dans une application Android. RIMAZ est un outil qui analyse le bytecode des applications Android développées en Java ou Kotlin, en se basant sur des heuristiques qui prennent en

considération les implémentations possibles des variantes de MVC (MVC, MVP et MVVM) sous Android. Nous menons subséquemment une étude sur 5000 applications du Google Play Store pour avoir une idée de l'écosystème Android relativement à l'utilisation des variantes de MVC. Par conséquent, nos contributions se résument essentiellement dans les deux points suivants :

1. une approche basée sur des heuristiques pour identifier les variantes de MVC dans les applications Android ;
2. une étude empirique qui répond aux questions relatives à l'utilisation des patrons basés sur MVC dans les applications Android.

L'intérêt de l'identification de l'utilisation de ces patrons représentent une plateforme de connaissance et une vue globale sur l'état actuel des architectures suivies. D'un côté, ceci facilitera dans le future l'étude de l'influence du choix de ces patrons sur la qualité des applications Android. D'un autre côté, étudier l'évolution de l'adoption de ces patrons avec le temps, aidera à comprendre la tendance suivie par la communauté et son impact sur cette plateforme. De même, analyser ces adoptions par catégories d'applications permet de trouver si un patron spécifique est plus convenable à un type particulier d'applications. Les résultats que nous pouvons obtenir en étudiant l'écosystème Android, auront certainement des bénéfices sur le monde du développement logiciel, vu l'importance que la plateforme Android représente dans cet espace.

L'intérêt de l'identification de l'utilisation de ces patrons par les applications Android constitue une plateforme de connaissance et une vue globale sur l'état actuelle des architectures suivies, ce qui facilitera dans le future l'étude de l'influence du choix de ces patrons sur la qualité des applications Android. D'un autre côté, étudier l'évolution de l'adoption des patrons avec le temps, aide à comprendre la tendance suivie par la communauté et son impact sur cette plateforme. De même,

analyser ces adoptions par catégories permet de trouver s'il existe une convenabilité d'un patron spécifique à un type particulier d'applications. Les résultats que nous pouvons obtenir en étudiant l'écosystème Android, auront certainement des bénéfices sur le développement logiciel, vu que la plateforme Android représente une grande part du développement mobile, qui est a son tour une grande ampleur dans le monde du développement logiciel.

Le reste de ce mémoire est organisé comme suit : le chapitre I présente brièvement quelques concepts clés sur Android, les patrons basés sur MVC ainsi que les différentes façons d'implémenter ces patrons dans cette plateforme. Le chapitre II est dédié à l'état de l'art et détaille les différents travaux connexes à notre sujet. Par la suite, le chapitre III présente RIMAZ notre approche outillée ainsi que les heuristiques sur lesquelles elle se base pour identifier les variantes de MVC dans les applications Android. Le chapitre IV présente l'évaluation de notre approche ainsi que l'étude empirique réalisée pour répondre à nos questions de recherche. À la fin, nous concluons ce mémoire par une synthèse du travail réalisé et une présentation des travaux futurs.

CHAPITRE I

CONCEPTS PRÉLIMINAIRES

Dans ce chapitre, nous décrivons les concepts clés sur lesquels repose notre travail de recherche. Puisque ce travail vise spécifiquement la plateforme Android, nous commençons par présenter brièvement son architecture, ses paradigmes, ainsi que les composantes principales de ses applications. Nous décrivant plusieurs aspects de la plateforme Android pour assurer une meilleure compréhension, tout en se focalisant sur les parties importantes qui sont liées à notre travail. Nous décrivons ensuite la notion des patrons de conception et nous finissons par aborder plus en profondeur les patrons architecturaux de type MVC.

1.1 La plate-forme Android

Dédié principalement aux appareils mobiles, Android est un système d'exploitation conçu et développé par Google. Basé sur le noyau Linux et ayant un code source majoritairement ouvert, il représente le meilleur environnement pour étudier les pratiques suivies dans le code source des applications mobiles. Ces dernières, sont sujettes à de nombreuses contraintes à la fois matérielles et logicielles. Par exemple, l'architecture des systèmes mobiles (Mudge, 2015), le changement de contexte (Context swhitching) qui concerne la sauvegarde et la commutation entre les différents états des processus, les ressources telles que l'écran, la batterie, la mémoire et le processeur, sont tous des contraintes qui affectent la nature et le

comportement des applications sur cette plateforme.

Les applications Android sont organisées autour de quelques concepts clés qui doivent être bien acquis par les développeurs et les concepteurs et que nous présentons dans les sous-sections suivantes.

1.1.1 Architecture et composantes principales

Résumée dans la figure 1.1, l'architecture de la plateforme Android se compose de plusieurs couches (Android-Doc, 2018l) :

i le noyau Linux, tout en bas, est le moteur principal sur lequel se base le système d'exploitation Android ;

ii suivi par la couche d'abstraction matérielle (en anglais, *Hardware Abstraction Layer (HAL)*), qui fournit des interfaces standards exposant les capacités matérielles de l'appareil aux couches supérieures ;

iii au milieu, se trouvent les libraires natives et l'environnement d'exécution. L'environnement d'exécution appelé ART [1] (ou son prédécesseur Dalvik) se charge d'exécuter le code intermédiaire octal (*Bytecode*), obtenu après la compilation du code source de l'application ;

iv le cadre d'applications Android qui fournit un ensemble de services et de composantes réutilisables que tout développeur utilise pour construire les différentes parties de l'application. En plus d'autres composantes, le cadre d'applications Android comprend :

- le gestionnaire des activités : contrôlant le cycle de vie de l'application et des activités ;
- le fournisseur de contenu : qui permet aux applications de partager leurs données avec d'autres applications ;

1. Android Runtime, source : `https://source.android.com/devices/tech/dalvik/`

- le gestionnaire des ressources : qui permet l'accès à d'autres ressources telles que les chaines de caracteres, les couleurs ou les dispositions d'interfaces utilisateurs ;
- le gestionnaire des notifications : permettant d'envoyer et d'afficher des notifications à l'utilisateur.

v tout en haut de la pile se trouve les applications qui sont installées par défaut avec le système.

Nous nous intéressons dans cette étude principalement à la couche (*iii*) ainsi que la couche (*iv*). La première contenant l'environnement d'exécution qui est responsable d'exécuter le bytecode, sur lequel nous nous basons pour analyser le code source des applications. De même, la couche (*iv*) "cadre d'applications" contient les composantes principales d'Android qui sont réutilisées dans chaque application. Elles sont importantes pour comprendre l'architecture et pour inférer l'utilisation de pratiques spécifiques dans le code source.

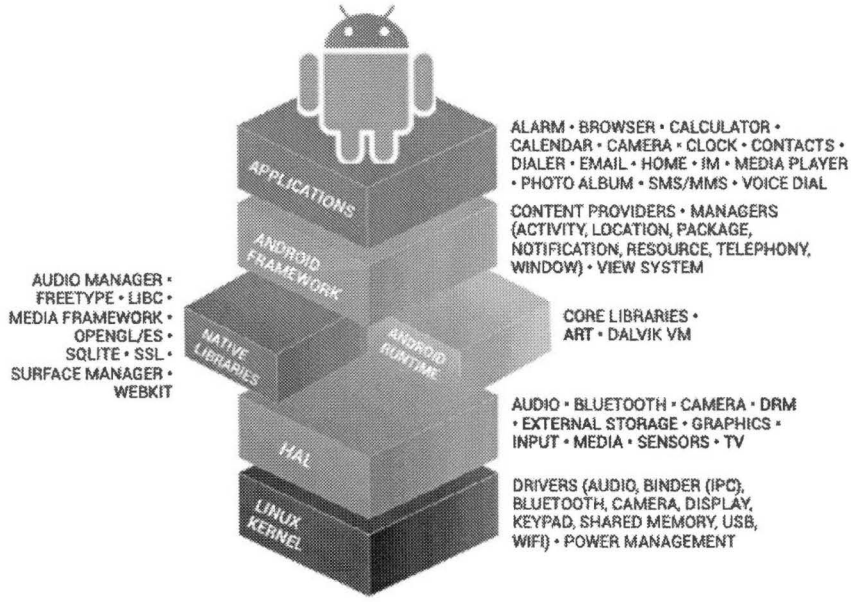


Figure 1.1: Architecture de la plateforme Android [2]

Les composantes de la couche (*iv*) cadre d'applications, comprennent l'*Activité*, le *Service*, le *Récepteur-De-Diffusion* et le *Fournisseur-De-Contenu*, (en anglais respectivement, *Activity*, *Service*, *BroadcastReceiver* et *ContentProvider*). Elles représentent les parties principales des applications Android et jouent un rôle important dans leur fonctionnement. Les Activités sont des composantes clés qui gèrent les écrans à afficher aux utilisateurs, tandis que les trois autres sont des composantes sans interfaces, qui traitent les tâches d'arrière-plan (les Services), les notifications (les Récepteurs-De-Diffusion) et l'accès aux données partagées (les Fournisseurs-De-Contenu).

En particulier, les Activités, servent à définir les interfaces graphiques et gérer les interactions des utilisateurs. De plus, elles jouent un rôle très important en contrôlant le flux et la navigation entre les différentes parties de l'application.

En plus des Activités, il existe quelques composantes de bas niveau qui sont attachées à l'Activité, telles que les Fragments et les Dialogues. Ces deux derniers partagent plusieurs similitudes avec les Activités. Tous les deux peuvent être responsables de gérer toute logique spécifique aux interfaces utilisateurs correspondantes. D'un coté, le Fragment représente le comportement d'une portion d'une interface utilisateur. Il doit être intégré dans une autre interface utilisateur globale supportée par une Activité (Android-Doc, 2018h). Le Dialogue, d'un autre coté, est une petite forme, qui n'occupe pas la totalité de l'écran et qui est affichée sous la forme d'une fenêtre contextuelle. Il invite l'utilisateur à réaliser des actions bien précises ou de rajouter des informations nécessaires pour le flux courant de l'Activité actuelle (Android-Doc, 2018g).

À un niveau plus granulaire, se situent des objets d'interface utilisateur tels que le `Button` et le `EditText` qui sont appelés des *Vues* (*Views*) dans Android. La figure

---

2. Source : `https://source.android.com/setup/`

1.2 montre un exemple d'un écran d'une application Android. Cet écran est défini par une Activité et contient différents objets tels que des boutons et des champs de texte.

Figure 1.2: Exemple des éléments de l'interface utilisateur des applications Android [3]

Toutes ces composantes aident à la réalisation d'une application Android. Le concept d'Application est explicitement défini dans Android avec une classe de haut niveau déclarée dans un fichier manifest (AndroidManifest.xml).

Les classes Applications ainsi que les Activités, les Fragments et les Dialogues sont

3. Source : https://extremegtx.wordpress.com/2015/01/07/android-studio-the-user-interface/

les principales composantes qui contribuent à la gestion du cycle de vie des applications Android ayant des interfaces utilisateurs. En effet, toutes ces composantes fournissent des évènements qui aident à gérer le cycle de vie d'une application Android. Par exemple, le cycle de vie d'une Activité quelconque peut être défini en gérant les évènements suivants, comme présenté dans la figure 1.3 (Android-Doc, 2018j) :

- **OnCreate()** cette méthode est appelée une seule fois durant tout le cycle de vie de l'Activité. Elle est appelée une fois que l'Activité est créée ;
- **onStart()** cette méthode est appelée lorsque l'Activité passe à l'état *Started*, devient visible à l'utilisateur et se prépare à accepter les interactions de l'utilisateur ;
- **onResume()** cette méthode est appelée lorsque l'Activité passe à l'état *Resumed.* Elle occupe le premier plan et devient interactive avec l'utilisateur ;
- **onPause()** est la méthode appelée lorsque l'Activité est dans l'état *Paused.* Dans ce cas, elle n'occupe plus le premier plan de l'application ;
- **onStop()** cette méthode est appelée lorsque l'Activité est dans l'état *Stopped.* L'Activité n'est plus utilisée et n'est plus affichée à l'utilisateur ;
- **onDestroy()** cette méthode est appelée juste avant la destruction de l'Activité par le système.

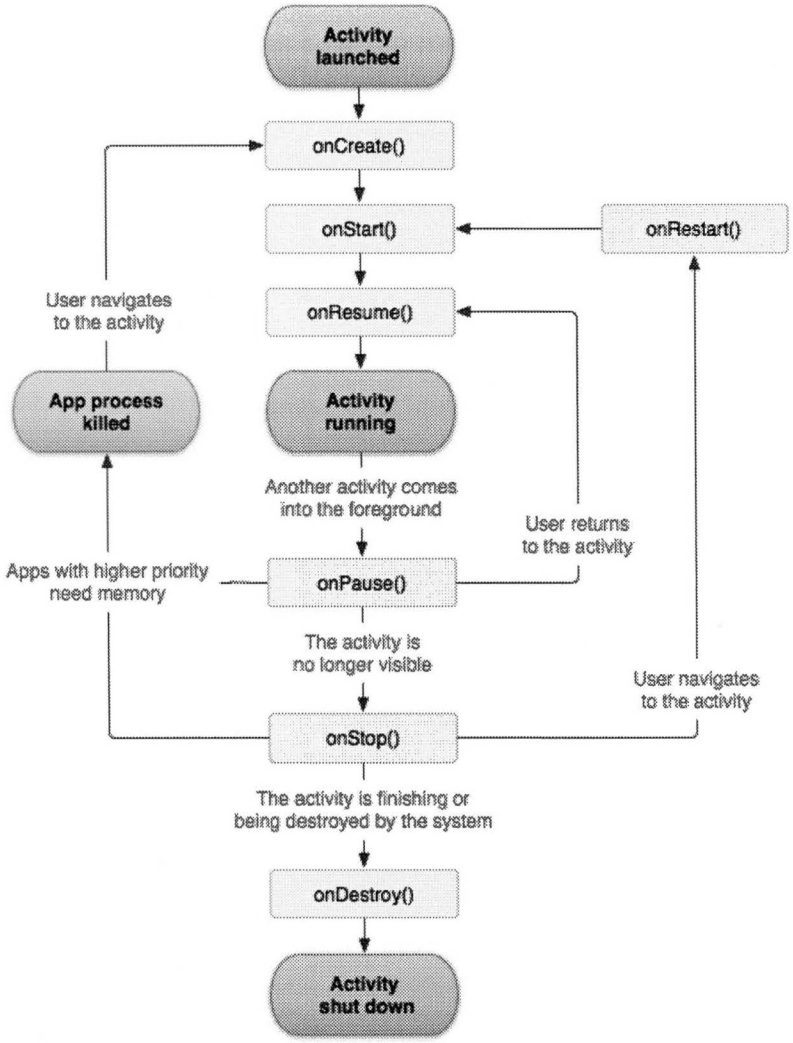


Figure 1.3: Une illustration simplifiée du cycle de vie de l'Activité [4]

1.1.2 Les applications Android

Les applications Android sont généralement développées avec des langages qui sont compatibles avec les environnements Dalvik/Art, tels que Java ou Kotlin, ou avec des langages plus natifs tels que C/C++. Le code source de l'application,

4. Source : `https://developer.android.com/guide/components/activities/activity-lifecycle`

groupé avec ses ressources, est compilé par des compilateurs respectifs à chaque langage (Java, Kotlin, C/C++ etc...) et le SDK (trousse de développement logiciel) (*Software Development Kit*) d'Android vers un code intermédiaire appelé *dex bytecode*. Ce dernier est contenu dans un ou plusieurs fichiers *.dex* (Android-Doc, 2018e). Les fichiers *dex* sont exécutés par l'environnement d'exécution géré (*Managed Environment*) ART ou par son prédécesseur Dalvik (Android-Doc, 2018d).

Les fichiers *dex* ainsi que les informations et les ressources nécessaires sont tous regroupés dans un fichier d'archivage appelé APK [5]. Parmi ces ressources, le fichier manifest (AndroidManifest.xml) illustré dans la figure 1.4 est un fichier *xml* qui contient des informations décrivant la structure et les spécifications de l'application (Android-Doc, 2018c). Ce fichier est responsable de :

1. la déclaration du nom du paquet principal de l'application (figure 1.4, ligne 6) ;
2. l'énumération des différentes permissions nécessaires pour accéder à certaines ressources logicielles ou matérielles (figure 1.4, ligne 32) ;
3. la déclaration des librairies tierces utilisées par l'application ;
4. l'énumération des composantes principales de l'application telles que les Activités, les Services, les Récepteurs-De-diffusion et les Fournisseurs-De-Contenu (figure 1.4, ligne 21).

1.2 Les patrons de conception

La conception et la réalisation d'un logiciel réutilisable et maintenable devient une tâche de plus en plus complexe, compte tenu de la complexité des exigences et des fonctionnalités à implémenter. De plus, développer un logiciel respectant des critères de qualité nécessite une grande expertise.

---

5. Android Package Kit, Source : `https://developer.android.com/guide/components/fundamentals.html`

```
01. <?xml version="1.0" encoding="utf-8"?>
02. <manifest
03.     xmlns:android="http://schemas.android.com/apk/res/android"
04.     android:versionCode="1"
05.     android:versionName="1.0"
06.     package="com.example.myapp">
07.
08.     <!-- Beware that these values are overridden by the build.gradle file -->
09.     <uses-sdk android:minSdkVersion="15" android:targetSdkVersion="26" />
10.
11.     <application
12.         android:allowBackup="true"
13.         android:icon="@mipmap/ic_launcher"
14.         android:roundIcon="@mipmap/ic_launcher_round"
15.         android:label="@string/app_name"
16.         android:supportsRtl="true"
17.         android:theme="@style/AppTheme">
18.
19.         <!-- This name is resolved to com.example.myapp.MainActivity
20.              based upon the package attribute -->
21.         <activity android:name=".MainActivity">
22.             <intent-filter>
23.                 <action android:name="android.intent.action.MAIN" />
24.                 <category android:name="android.intent.category.LAUNCHER" />
25.             </intent-filter>
26.         </activity>
27.
28.         <activity
29.             android:name=".DisplayMessageActivity"
30.             android:parentActivityName=".MainActivity" />
31.     </application>
32.     <uses-permission android:name="android.permission.SEND_SMS"/>
33. </manifest>
```

Figure 1.4: Exemple d'un fichier manifest(Android-Doc, 2018c)

Les concepteurs expérimentés tendent à réutiliser certaines techniques et solutions déjà utilisées pour résoudre des problèmes récurrents. Ces solutions résolvent des problèmes de conception bien précis et rendent la conception du logiciel plus élégante, flexible, réutilisable et maintenable. Ces pratiques, solutions ou techniques sont appelées patrons de conception. Ceux-ci ont été introduits au début pour la programmation orientée-objet dans le livre *Design Patterns : Elements of Reusable Object-Oriented Software* (Gamma *et al.*, 1995), où les auteurs ont catalogué 23 patrons de conception, regroupés en trois catégories : patrons de création, patrons structuraux et patrons comportementaux.

Par définition, un patron de conception (*Design pattern*) est une solution générale réutilisable pour des problèmes communs récurrents dans la conception et l'implémentation des logiciels. Il est caractérisé par son nom, le problème associé, la solution et les conséquences de la solution (Martin, 1996).

1.3 Les patrons architecturaux

Les solutions informatiques ont toujours connu une croissance en terme de complexité. Une partie de cette complexité est due à leurs architectures. L'architecture d'un logiciel contient la description du système en terme de ses composantes ainsi que ses relations et interactions (Garlan *et al.*, 1994).

Un patron architectural (*Architectural design pattern*) est une structure explicative qui se charge de gérer l'exécution des composants et de leurs connecteurs (Alencar *et al.*, 1996).

Nous nous intéressons dans ce mémoire aux patrons architecturaux de la couche présentation, qui sont réputés au sein de la communauté du développement mobile. Un des fameux patrons de ce type est le patron Modèle-Vue-Contrôleur (MVC) (*Model-View-Controller*). Ce dernier, et beaucoup d'autres de ses dérivés, sont largement utilisés pour le but d'assurer une séparation claire entre les données sous-jacentes, afin de concevoir un logiciel maintenable de haute qualité (Buschmann *et al.*, 2007).

D'autres patrons ont été dérivés du patron MVC afin d'offrir plus d'optimisation en terme de découplage entre les différentes classes des applications. Les patrons Modèle-Vue-Présentateur (MVP) (*Model-View-Presenter*) et Modèle-Vue-VueModèle (MVVM) (*Model-View-ViewModel*) en particulier, suscitent de plus en plus l'intérêt des concepteurs et des développeurs des applications mobiles. Nous nous intéressons dans ce travail à ces trois patrons architecturaux (MVC, MVP, MVVM), compte tenu de leur utilisation dans le monde des applications mobiles et le débat qu'ils soulèvent dans l'écosystème Android spécifiquement.

Il existe cependant différentes définitions et perspectives sur la meilleure façon d'implémenter chacun de ces patrons. Vu la complexité de notre approche et la

solution que nous proposons, nous abordons dans les sous-sections qui suivent, en détail, les caractéristiques de chaque patron, ainsi que la manière de les mettre en œuvre dans les applications Android, tout en traitant plusieurs points complexes qui sont nécessaires pour mieux comprendre nos choix techniques dans le chapitre 3.

1.3.1 Modèle-Vue-Contrôleur

MVC a été introduit par *Trygve Reenskaug* en 1979 comme une solution aux problèmes des utilisateurs qui *contrôlent* de grandes quantités de données (Reenskaug, 1979). MVC vise à séparer la logique d'affaire de la couche présentation (Reenskaug, 1979; Dey, 2011). Il définit trois couches principales :

1. la couche **Modèle**, cette couche est constituée d'un ensemble d'entités représentant le savoir (entités du domaine d'affaire), ainsi qu'un groupe de règles qui gèrent les mises à jour et l'accès à ces entités (Reenskaug, 1979). Pour assurer un couplage lâche (*Loose Coupling*) avec les autres couches, le Modèle ne devrait pas connaître la nature des autres composants. Cependant, il peut envoyer des notifications sur l'état de ces objets (Buschmann *et al.*, 2007; Deacon, 2005) ;
2. la **Vue**, qui se charge de fournir une représentation visuelle des entités du Modèle. Elle présente les données récupérées à partir du Modèle en envoyant des requêtes et change son état à travers des messages (Reenskaug, 1979; Dey, 2011).
3. le **Contrôleur**, étant le point d'entrée principal de l'application, il établit le lien entre l'utilisateur et le système. De plus, il manipule la Vue et traduit l'interaction de l'utilisateur au Modèle (Reenskaug, 1979). En outre, le Contrôleur communique directement avec le Modèle et doit être capable de changer son état. Le Modèle, à son tour, peut envoyer des notifications

au contrôleur, sans avoir un lien direct avec ce dernier (Buschmann *et al.*, 2007).

La figure 1.5 (a) montre les trois composantes de l'architecture MVC originale et décrit les connections entre eux.

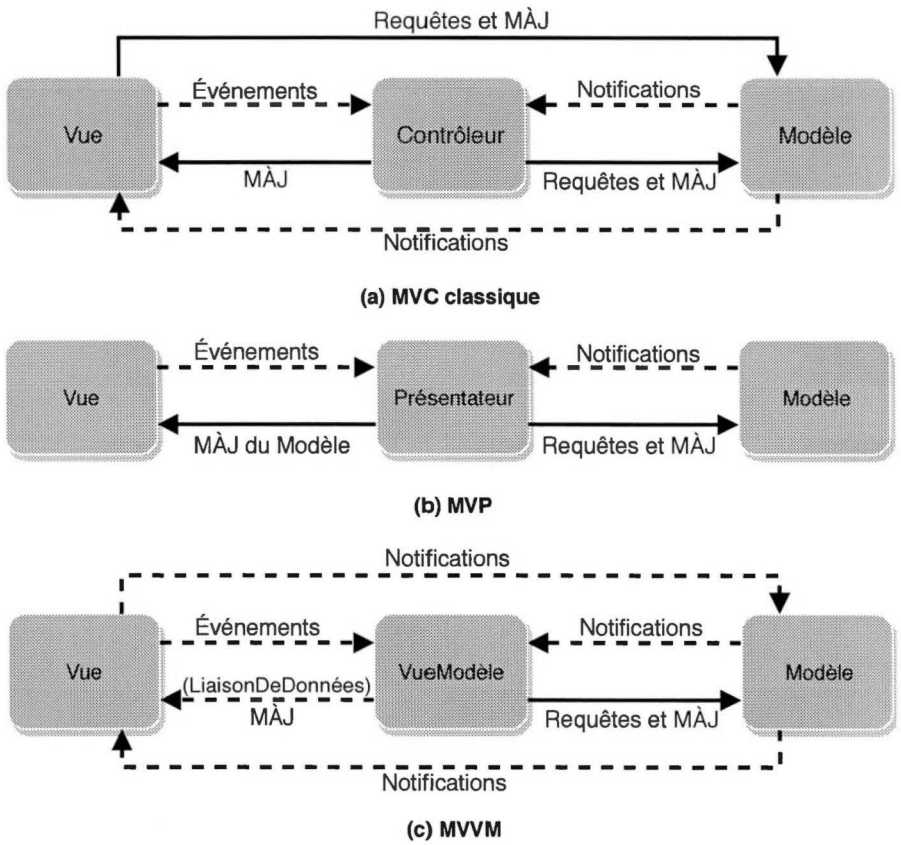


Figure 1.5: Diagrammes présentant les patrons architecturaux discutés

Depuis son introduction, le patron MVC a évolué avec l'apparition de nouveaux cadres de développement. Plusieurs adaptations ont été apportées à ce patron pour s'adapter aux plateformes visées, ce qui a donné lieu à de nouvelles variantes du patron MVC.

1.3.2 Modèle-Vue-Présentateur

MVP, comme le montre la figure 1.5 (b), est une variante de l'architecture MVC. Introduit au début des années 90s, il apporte plus de découplage entre la couche du domaine et l'interface utilisateur (Sokolova et Lemercier, 2014).

Contrairement à MVC, le **Présentateur** dans MVP est plus isolé -que le Contrôleur dans MVC- et ne doit rien connaître sur le système d'exploitation sous-jacent. Les entités du Modèle ont essentiellement le même rôle et les mêmes responsabilités, tandis que les Vues gèrent les actions qui manipulent les composantes du système ou de l'interface utilisateur et délègue aux éléments du présentateur, seulement les actions qui manipulent les entités du Modèle. Si le Présentateur aurait besoin d'apporter des modifications visuelles à l'interface utilisateur ou d'interagir avec le système exploitation sous-jacent, il devrait appeler une méthode appartenant à un élément de la Vue qui implémente ce besoin (Fowler, Martinn,).

1.3.3 Modèle Vue VueModèle

MVVM est une évolution du patron MVP (John, Gossman, 2005). Comme indiqué dans la figure 1.5 (c), il vise à réaliser encore plus de découplage entre l'interface utilisateur et la couche intermédiaire, représentée par les éléments du VueModèle. La communication entre ces deux parties se fait exclusivement à travers des notifications de liaison-de-données bidirectionnelles (*Two-Way DataBinding*).

La Vue, comme dans les patrons précédents, représente l'interface utilisateur. Ses éléments sont généralement écrits avec des langages déclaratifs, comme XML ou une de ses dérivées. La vue doit être aussi passive que possible, elle doit déclarer juste des objets graphiques, et délègue toute la logique au **VueModèle** qui le supporte, et d'où elle récupère tout type de mises à jour (Microsoft-MSDN, 2004). Comme son nom l'indique, le VueModèle n'est que le Modèle de la Vue. Il définit

et implémente les évènements d'entrées de l'utilisateur. De plus, il transforme les données du Modèle et le rend prêt à être affiché par la Vue. Le point le plus important ici, c'est que le VueModèle ne doit pas être couplé directement aux Vues correspondantes. Aucune référence aux Vues ne doit être déclarée ni utilisée au sein du VueModèle. Le rôle de la couche Modèle reste inchangé, comparativement aux deux patrons précédents.

1.3.4 Implémentation des trois patrons dans Android

Le MVC a été le premier patron à être adopté dans le monde Android. Pourtant, son implémentation spécifique dans les applications Android est restée toujours un sujet de débat. Au centre de ce débat, se situe l'interprétation du Contrôleur (StackOverflow, 2018; Maxwell Eric, 2017). Ce dernier est défini comme :

i Le point d'entrée ;

ii La couche qui gère les interactions de l'utilisateur avec le logiciel ;

iii La couche qui traduit l'interaction de l'utilisateur avec la Vue à la couche de données.

En se basant sur ces trois caractéristiques, quelques développeurs considèrent les Activités et d'autres composantes similaires (tels que les Fragments et les Dialogues) comme étant les composantes principales qui jouent le rôle du Contrôleur (StackOverflow, 2018; Maxwell Eric, 2017), pour les raisons suivantes :

i Les Activités sont les points d'entrées principales d'une application ayant une interface utilisateur ;

ii Les Activités représentent l'emplacement le plus naturel pour définir les évènements d'entrée. Même si certains développeurs déclarent les évènements dans d'autres composantes. De plus, un autre type d'évènements, dits gestionnaires

d'évènements, peuvent seulement être déclarés dans ce types de composantes (Activités et d'autres composantes similaires).

Toutefois, d'autres développeurs croient que les Activités, les Fragments ainsi que les Dialogues doivent être considérés comme des composantes de la Vue, et que les composantes du Contrôleur doivent être des classes séparées (StackOverflow, 2018; Musselwhite, 3 01). Les deux choix sont soutenus par des arguments solides et doivent être pris en considération par toute approche de détection.

Le MVP de son coté est simple à implémenter sur Android. La figure 1.6 résume brièvement la relation entre la Vue et le Présentateur qui représente le cœur de ce patron. En effet, en suivant sa définition standard, l'implémentation du patron MVP sur Android est basée sur les points suivants :

1. Le patron MVP garde la déclaration des évènements dans les entités de la Vue. Dans la figure 1.6, l'événement `OnEventListenerAction()` est déclaré et défini dans la Vue (représentée par la classe `Activity`) ;
2. La logique que ces évènements doivent effectuer est déléguée toutefois au Présentateur à travers une de ses méthodes. L'évènement `OnEventListenerAction()` par exemple appelle la méthode `presenterMethod()` appartenant au Présentateur (figure 1.6) qui se charge d'exécuter toute logique désirée (par exemple, la manipulation du Modèle) ;
3. Le Présentateur à son tour doit déléguer la manipulation des composantes de l'interface utilisateur et de la plateforme à la Vue. Dans la figure 1.6, le Présentateur `ConcretePresenter` appelle la méthode `viewMethod()` de la Vue dans sa méthode `presenterMethod()` ;
4. Pour atteindre ce référencement croisé, l'entité représentant la Vue (L'Activité, par exemple) doit contenir un champ qui fait référence au Présentateur et vice versa. Dans le même exemple, la classe `Activity` (représentant

la Vue) contient un champ `presenter` de type `AbstractPresenter`, tandis que la classe `ConcretePresenter` (représentant le Présentateur) contient un champ `view` de type `AbstractView`. Ces deux champs ont des types abstraits et sont initialisés avec des types concrets, explicitement par les développeurs ou implicitement par des techniques d'injection de dépendances (*Dependency Injection*).

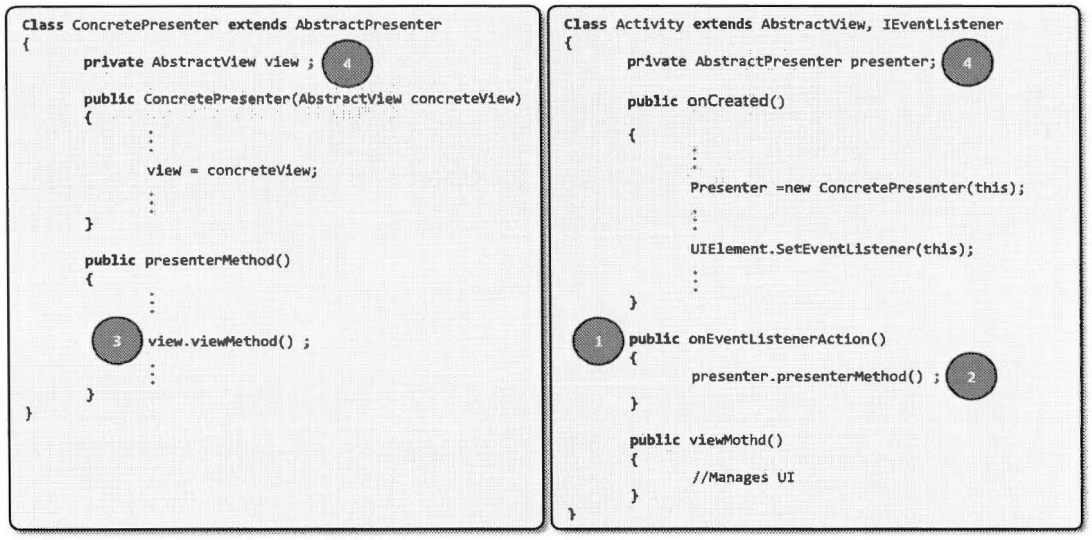


Figure 1.6: La relation entre le Présentateur et la Vue en MVP

La figure 1.7 montre le diagramme de classes de l'exemple illustré par la figure 1.6. Les classes `ConcretePresenter` et `ConcreteView(Activity)` sont des classes concrètes qui étendent (ou implémentent) des classes abstraites `AbstractPresenter` et `AbstractView` respectivement, d'où elles héritent les méthodes `presenterMethod()` et `viewMethod()`.

MVVM (Modèle Vue VueModèle) à son tour est venu tard dans le monde Android. Cependant, depuis l'introduction de nouvelles composantes architecturales par Google en 2017, ce patron a pris un plus grand profil et est maintenant recommandé pour les applications Android (Android-Doc, 2018i). Ces dernières années, la plateforme Android a commencé à fournir un support avancé à ce patron

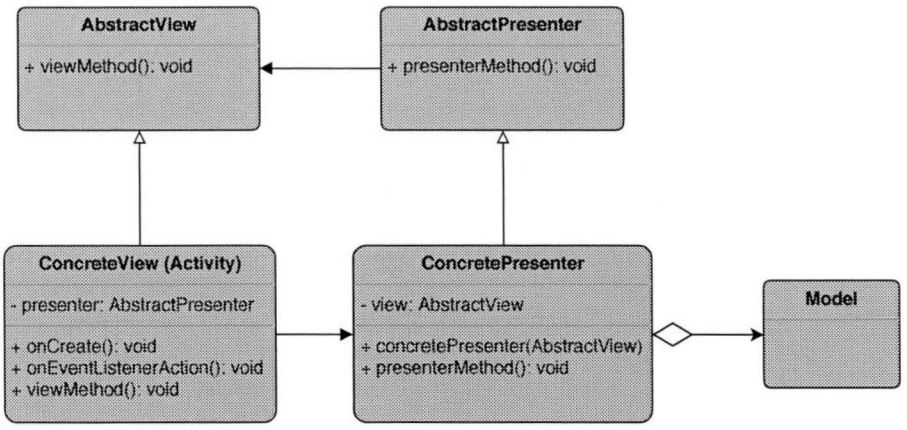


Figure 1.7: Diagramme représentant l'architecture du patron MVP

à travers des types de données spécifiques tels que *ViewModel* (Android-Doc, 2018n) et le *LiveData* (Android-Doc, 2018k). Ceci dit, Google n'a commencé à supporter complètement les techniques avancées de la liaison-de-données qu'en 2016 (Android-Doc, 2018f), ce qui a amené les développeurs à utiliser des librairies tierces pour implémenter MVVM. Le VueModèle dans MVVM est représenté par des classes qui jouent le rôle d'un contexte de liaison-de-données pour chaque élément de la Vue. Lier les éléments de la Vue (représentés par des fichiers de disposition xml) à des classes du VueModèle génère des classes intermédiaires qui définissent toutes les opérations de liaison-de-données et les préparent pour l'exécution (Android-Doc, 2018f). Similairement, les évènements invoqués par les objets graphiques sont définis par la liaison-de-données sous la forme de méthodes dans les classes du VueModèle. Ils seront redéfinis plus tard automatiquement dans les classes de liaison-de-données générées en utilisant le patron Écouteur (*Listener*) traditionnel (Android-Doc, 2018f).

1.4 Conclusion

Nous avons décrit dans ce chapitre les concepts de base concernant la plateforme Android, les patrons de conception et les patrons architecturaux, afin de mieux

comprendre les axes principaux de notre étude. Aussi, afin d'avoir une idée globale sur les travaux qui ont visé les mêmes aspects, nous présentons dans le chapitre qui suit, un état de l'art sur les différentes études qui ont traité l'analyse manuelle ou automatique des patrons, anti-patrons ou de défauts de code en général et dans les applications mobiles Android plus spécifiquement.

CHAPITRE II

ÉTAT DE L'ART

L'étude des pratiques suivies par les développeurs et les concepteurs dans le code source de différents types d'applications a toujours été un sujet d'intérêt pour beaucoup de chercheurs. Étant bonnes ou mauvaises, de niveau conceptuel ou de niveau d'implémentation, plusieurs études ont été menées pour les discuter. Une des raisons de cet intérêt est le besoin de comprendre l'influence de certaines pratiques sur la qualité ou l'évolution des applications, ainsi que l'impact des corrections de certaines mauvaises pratiques.

Une multitude de travaux récents ont été menés visant ces différents axes. La majorité de ces travaux se divisent en deux catégories principales : l'étude des patrons ainsi que l'étude des anti-patrons et des défauts de code.

Ces études proposent généralement des approches d'analyses manuelles ou automatiques du code source des applications. L'analyse automatique, qui représente la majorité de ces approches, vise à identifier la présence des patrons de conception, des anti-patrons ou des défauts de code dans le code source des applications.

Les applications mobiles, particulièrement, ont connu récemment un vif intérêt des chercheurs. Certains travaux ont proposé des approches de détection automatique des anti-patrons et des défauts de code (Hecht *et al.*, 2015; Palomba *et al.*, 2017),

tandis que peu de travaux ont traité les patrons de conception ou les patrons architecturaux. En effet, une seule étude visait la détection des styles architecturaux dans ce type d'applications (Bagheri *et al.*, 2016). Ceci dit, et jusqu'à maintenant, il n'existe aucun travail qui vise spécifiquement la détection automatique des patrons architecturaux de la couche présentation dans les applications mobiles.

Dans les sections qui suivent, nous abordons en détail les différents travaux qui ont été conduits concernant l'analyse automatique de code, l'étude des patrons dans les applications mobiles, la détection automatique des anti-patrons et des défauts de code ainsi que la détection des patrons architecturaux dans les applications mobiles.

2.1 Analyse automatique du code source

Dès les premières années d'utilisation et de documentation des patrons de conception, les chercheurs ont toujours été intéressés par la détection automatique de ces patrons et ce, en proposant des approches d'analyse de code.

Une des premières contributions dans ce domaine est l'étude d'Antoniol et al où ils ont présenté une approche pour la détection des patrons de conception dans les applications orientées objet. Ils ont proposé, à cet effet, un processus de détection basé sur une stratégie de filtrage à plusieurs étapes (Antoniol *et al.*, 1998). La première étape, consiste à fusionner le code source et la conception du programme et les transformer en une représentation intermédiaire abstraite. Cette représentation, basée sur UML, est capable d'exprimer les concepts de base de la programmation orientée objet. La deuxième étape, collecte à partir de cette représentation intermédiaire un nombre de métriques telles que le nombre de champs, méthodes, associations, agrégations et héritages. Ces métriques vont être utilisées après dans la dernière l'étape pour déduire les patrons utilisés.

Certains problèmes de conception peuvent empêcher la réingénierie et l'amélioration des systèmes. Ciupke a présenté une technique qui a pour but l'identification et la localisation de ces problèmes. En spécifiant des problèmes de conception récurrents, cette technique analyse le code légataire, applique des requêtes de détection et localise l'emplacement de ces problèmes (Ciupke, 1999).

Alikacem et Sahraoui, pour leur part ont présenté une approche de détection des violations des bonnes pratiques dans la programmation à objets. Cette approche présente une modélisation de ces pratiques par des systèmes à base de règles floues et les applique sur le code source des applications (Alikacem et Sahraoui, 2006). Comme la figure 2.1 le montre, leur approche se compose de trois étapes principales. La première consiste à convertir le code source de l'application en une représentation aisément manipulable. La deuxième partie ensuite extrait à partir de cette représentation intermédiaire une liste de métriques, qui sera utilisée dans la dernière étape, afin de détecter les non-conformités dans le code de l'application et générer à la fin un rapport général d'évaluation.

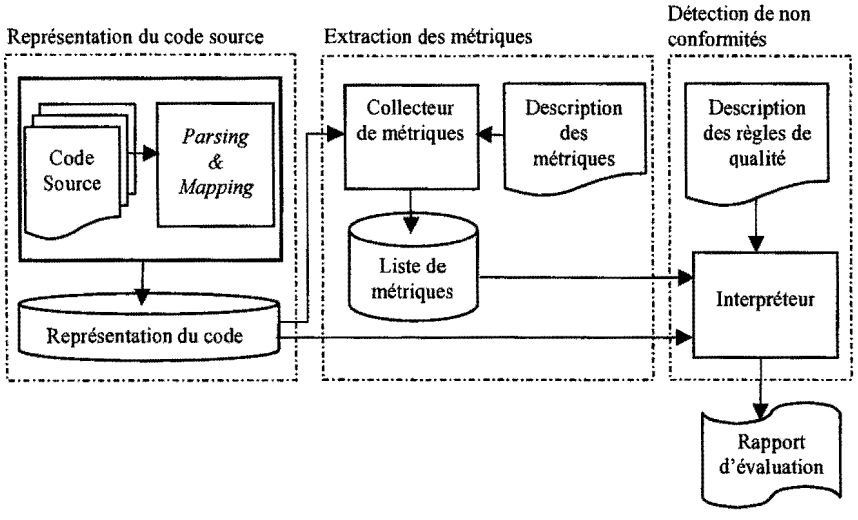


Figure 2.1: Architecture du système d'évaluation de la qualité proposée par Alikacem *et al.* (Alikacem et Sahraoui, 2006)

(Fontana et Zanoni, 2011) ont proposé un module d'extension (*Plug-In* en Anglais) appelé *MARPLE* (Metrics and Architecture Reconstruction Plug-in for Eclipse) pour l'environnement de développement Eclipse. Cet outil permet de détecter les patrons de conception en effectuant la reconstruction de l'architecture du logiciel, tout en se basant sur des métriques qui sont extraites à partir du code source. Il utilise une représentation basée sur l'arbre syntaxique abstrait du programme analysé et visent à détecter des patrons de conception usuels de la programmation orientée objet.

Ces approches, tout comme beaucoup d'autres travaux similaires, ont contribué à prouver l'efficacité de l'analyse et l'instrumentalisation du code source pour la découverte de patrons de conception.

2.2 Étude des patrons dans les applications mobiles

Peu de travaux, récents pour la plupart, traitent les patrons de conception ou les patrons architecturaux dans le monde des applications mobiles. À titre d'exemple, un mémoire de maitrise (Lou, 2016) présente une analyse manuelle et une étude empirique pour vérifier si les architectures MVP et MVVM sont, du point de vue qualité, meilleures que MVC dans les applications Android. Pour répondre à cette problématique, l'auteur a comparé ces architectures selon trois critères, la testabilité, la modifiabilité et la performance. De même, l'auteur a choisi des facteurs clés sur lesquels il s'est basé pour évaluer ces architectures. Pour la testabilité, il a considéré la quantité de cas de test (*Test Cases*), le temps consommé durant ces tests ainsi que la facilité de déboguer en utilisant des points d'arrêt (*Break Points*). Pour la modifiabilité, deux facteurs ont été pris en considération, le niveau de cohésion et le niveau de couplage/découplage entre les différents modules d'une application. En ce qui concerne la performance, seule l'allocation de mémoire a été considérée comme facteur de mesure. L'auteur s'est basé sur les définitions

des trois patrons étudiés et sur des métriques bien documentées pour mener son étude. Cependant, son étude était purement manuelle et a concerné une seule application appelée *to-do*, proposée comme un exemple officiel pour implémenter les trois patrons.

Un autre mémoire de maitrise (Lyngstad, 2013) a proposé une vue globale sur quelques patrons de conception orientés objet tels que le patron *Fabrique* (*Factory Method*), l'*Adaptateur* (*Adapter*), la *Stratégie* (*Strategy*) et le *Composite*, et des patrons architecturaux tels que MVC. De plus, l'auteur a fourni des explications sur comment ces patrons sont implémentés et comment ils s'accordent avec l'environnement Android. Similairement à l'étude précédente, cette étude s'est aussi basée sur une seule application type développée par l'auteur.

Par ailleurs, un livre récent (Johnson, 2018) présente aux développeurs la manière de créer des applications mobiles sophistiquées en utilisant le patron MVVM sur la plateforme Xamarin. Cette dernière permet d'écrire des applications mobiles multi-plateformes (Android, iOS, Windows) en utilisant le langage C#.

(La et Kim, 2010) pour leur part ont présenté une manière de concevoir une architecture MVC équilibrée pour les applications mobiles basées sur les services. D'autres travaux similaires ont eu pour objet l'étude des patrons architecturaux et ont proposé des adaptations à ce type de patron propres aux applications mobiles (Sokolova et Lemercier, 2014; Shahbudin et Chua, 2013).

Tous ces travaux fournissent des contributions intéressantes sur les patrons de conception et les patrons architecturaux dans les applications mobiles. Cependant, très peu de ces travaux proposent des approches de détection automatique pour les identifier. La section suivante montre que les travaux les plus récents concernent plutôt la détection automatique des anti-patrons et des défauts de code dans les applications mobiles.

2.3 Détection automatique des anti-patrons et des défauts de code

Bien que peu nombreux, il existe des travaux pertinents sur l'identification automatique des anti-patrons et des défauts de code dans les applications mobiles. Les anti-patrons sont des erreurs courantes et des mauvaises solutions à des problèmes récurrents au niveau de la conception du code source (Koenig, 1998), tandis que les défauts de code sont des symptômes qui peuvent indiquer des problèmes à un niveau d'implémentation (Tufano *et al.*, 2015).

(Hecht *et al.*, 2015) ont proposé un outil automatique appelé PAPRIKA permettant d'effectuer la détection automatique de sept types de défauts de code, dont trois communs en orienté objet et quatre spécifiques à la plateforme Android. Leur objectif était de traquer l'évolution des applications Android en recensant l'existence de ces défauts de code, et cela en analysant le code compilé de 3568 versions de 106 applications Android différentes.

D'une manière similaire, (Palomba *et al.*, 2017) ont proposé un outil appelé ADOCTOR, qui est capable d'identifier 15 défauts de code spécifiques à la plateforme Android . ADOCTOR se base sur le catalogue de (Reimann *et al.*, 2014) et utilise des règles de détection basées sur des métriques. Ils ont aussi conduit une étude empirique sur un ensemble de 18 applications Android en utilisant l'outil proposé ADOCTOR, qui a montré une précision de 98% et un rappel de 98%.

(Kessentini et Ouni, 2017) ont proposé, pour leur part, une approche alternative pour détecter des défauts de code de type Android, en utilisant un algorithme de programmation génétique multi-objectif. Ils ont aussi évalué leur approche sur 184 applications Android à code source ouvert, en détectant 10 défauts de code.

Ces travaux sont pertinents, mais concernent seulement les anti-patrons et les défauts de code et ne visent en aucun cas les patrons de conception ou les pa-

trons architecturaux. La section suivante présentera la seule étude qui a traité les patrons architecturaux dans les applications mobiles en faisant de la détection automatique.

2.4 Récupération des architectures dans les applications mobiles

(Bagheri *et al.*, 2016) ont effectué la première et la seule analyse poussée qui visait les architectures utilisées dans les applications Android. Ils ont identifié plusieurs facteurs qui ont motivé l'adoption de principes architecturaux dans les logiciels mobiles, spécifiquement pour Android. Ils ont recouvert l'architecture de centaines d'applications et ont pu extraire cinq différents types de styles architecturaux : l'invocation explicite basée sur les messages (Message-Based Explicit-Invocation), l'invocation implicite basée sur les messages (*Message-Based Implicit-Invocation*), publier-s'abonner (*Publish-Subscribe*), l'état partagé (*Shared State*) et les styles d'objets distribués (*Distributed Object Styles*). Ils ont appliqué leur étude sur un ensemble de 1400 applications récupérées depuis plusieurs sources telles que l'officiel Android Play Store [1], F-droid [2], Bazaar [3] et Wandoujia [4]. Ceci leurs a permis de trouver plusieurs utilisations incorrectes dans les styles et les constructions architecturales dans l'architecture inversée de quelques applications populaires.

2.5 Conclusion

Pour conclure, nous avons résumé dans cet état de l'art les différents axes des travaux et les études qui ont été menées autour de l'analyse manuelle et automa-

---

1. Android Play Store, source : `https://play.google.com/store`
2. F-droid, source : `https://f-droid.org/en/`
3. Bazaar, source : `https://cafebazaar.ir/`
4. Wandoujia, source : `https://www.wandoujia.com/`

tique du code source, et qui ont eu comme but de détecter la présence de patrons de conception, patrons architecturaux, anti-patrons et défauts de code en général. Malgré que certains travaux utilisaient des métriques extraites à partir du code source pour détecter l'utilisation des patrons de conception (tel que MARPLE), ces approches sont bien spécifiques à quelques patrons de conceptions, et ne visent pas la plateforme Android particulièrement. Détecter des patrons architecturaux sur la plateforme Android nécessite la conception d'heuristiques fortement liées à cette plateforme. Nous avons relevé le peu d'études qui visaient la récupération des patrons architecturaux dans les applications en général et dans les applications Android spécifiquement. Notre travail de recherche a pour but de combler ce manque en nous intéressant aux patrons architecturaux de type MVC. Ces derniers jouent un rôle très important dans la couche présentation des applications mobiles en général et dans la plateforme Android en particulier. De ce fait, nous nous sommes intéressés à étudier la présence de ces patrons dans les applications Android en proposant une approche outillée pour détecter automatiquement leur présence et pour analyser les différentes tendances accompagnant leurs adoptions. Le chapitre suivant présente notre approche de détection tout en détaillant toutes les étapes et en les illustrant à travers un exemple référent.

CHAPITRE III

IDENTIFICATION DES PATRONS DE TYPE MVC

Dans ce mémoire, nous proposons une approche pour identifier automatiquement le style architectural implémenté dans la couche présentation d'une application Android. Notre approche, appelée RIMAZ, comprend plusieurs étapes et implémente différentes heuristiques que nous présenterons successivement dans les sections suivantes. Nous avons défini ces heuristiques en se basant sur les spécificités de la plateforme Android, des patrons architecturaux visés ainsi que de leurs implémentations sur cette plateforme.

Afin de mieux comprendre les heuristiques proposées et les différentes étapes suivies par l'approche RIMAZ, nous utilisons un exemple référent (*Running Example*) d'une application Android à code source ouvert, appelée *Archi*. Nous commençons ce chapitre par la description de cette application.

3.1 Exemple référent

L'application *Archi*[1] a pour but d'afficher pour un nom d'utilisateur donné une liste d'entrepôts publiques de codes sources sur le site web GitHub[2] (figure 3.1

1. Source : `https://github.com/ivacf/archi`

2. Source : `https://www.github.com/`

(a)). Pour chaque entrepôt sélectionné, des informations sont affichées sur un écran de description séparé (figure 3.1 (b)). L'application a été développée par un des développeurs de la communauté Android en trois versions, afin de présenter et de comparer différents patrons architecturaux pouvant être utilisés pour créer des applications Android. Le même exemple d'application est créé trois fois en utilisant les patrons MVC, MVP et MVVM sur la plateforme Android.

Comme le montre la figure 3.1, l'application contient deux écrans, le premier représente la page principale où l'utilisateur peut voir la liste d'entrepôts pour un nom d'utilisateur précis. En cliquant sur un entrepôt donné, l'utilisateur sera conduit vers une deuxième page dédiée uniquement à décrire l'entrepôt choisi.

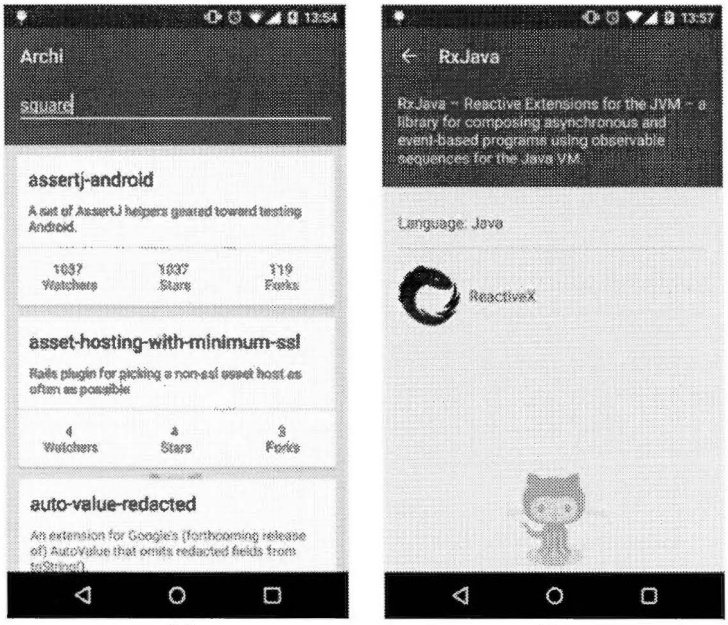
Les figures 3.2, 3.3 et 3.4 présentent des diagrammes de classes de cette application pour chaque patron architectural implémenté. Cette application est caractérisée par les composantes suivantes :

1. Les classes et les interfaces suivantes qui représentent les entités décrivant les informations de base de la logique d'affaires :
   - `Repository` : est une classe qui représente l'entrepôt du code source du projet hébergé sur le site GitHub ;
   - `User` : classe qui représente l'utilisateur propriétaire d'un entrepôt de données ;
   - `GithubService` : une interface qui aide à la récupération des données à partir du site web GitHub.
2. Les classes suivantes qui gèrent les éléments visuels et les écrans de l'application :
   - `MainActivity` : est une classe qui gère l'écran contenant la liste des entrepôts pour un nom d'utilisateur donné (figure 3.1 (a)) ;
   - `RepositoryActivity` : est une classe qui gère l'écran qui affiche les in-

formations d'un entrepôt précis (figure 3.1 (b)) ;

- `RepositoryAdapter` : est une classe qui adapte visuellement un entrepôt pour qu'il soit affiché dans une liste dans l'écran géré par `MainActivity`.

3. La classe `ArchiApplication` : gère l'instance de l'application Archi.



(a) (b)

Figure 3.1: Aperçu de l'application Archi [3]

Ces composantes sont réutilisées dans chaque version en les combinant avec d'autres classes dépendamment de l'architecture visée.

La figure 3.2 montre le diagramme de classes de l'application Archi implémentée avec le patron MVC. Les entités qui gèrent les écrans, telles que `MainActivity` et `RepositoryActivity`, manipulent les objets de la logique d'affaires (`Repository` et `User`) directement sans l'aide de classes intermédiaires, à l'exception de la classe `RepositoryAdapter` qui joue le rôle d'un adaptateur.

---

3. Source : `https://github.com/ivacf/archi/blob/master/README.md`

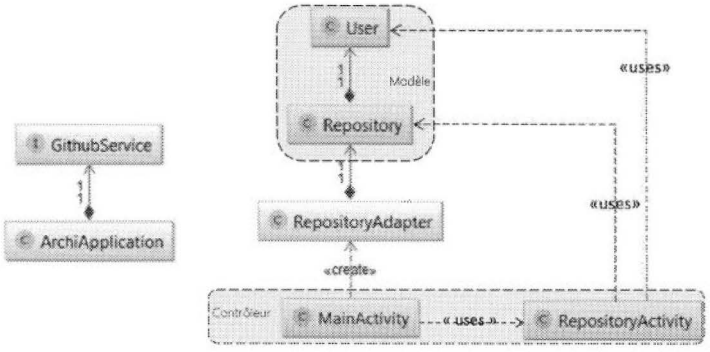


Figure 3.2: Diagramme de classes de la version MVC de l'application Archi

La figure 3.3 montre le diagramme de classes de l'application implémentée avec le patron MVP. Les classes Activités manipulent les classes `Repository` et `User` à travers des classes intermédiaires telles que la classe `MainPresenter` ou la classe `RepositoryPresenter`.

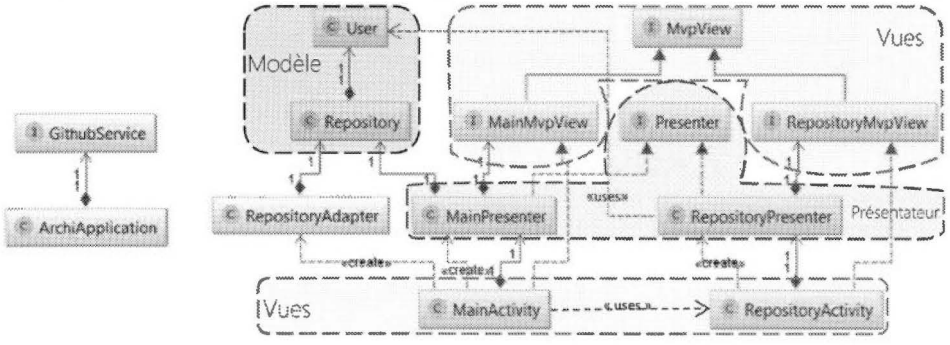


Figure 3.3: Diagramme de classes de la version MVP de l'application Archi

Dans la figure 3.4, où l'application Archi est implémentée en utilisant le patron MVVM, les deux classes d'Activité `MainActivity` et `RepositoryActivity` manipulent les classes `Repository` et `User` à travers les classes `MainViewModel`, `RepositoryViewModel` et les classes de liaison-de-données auto-générées `MainActivityBinding`, `RepositoryActivityBinding`.

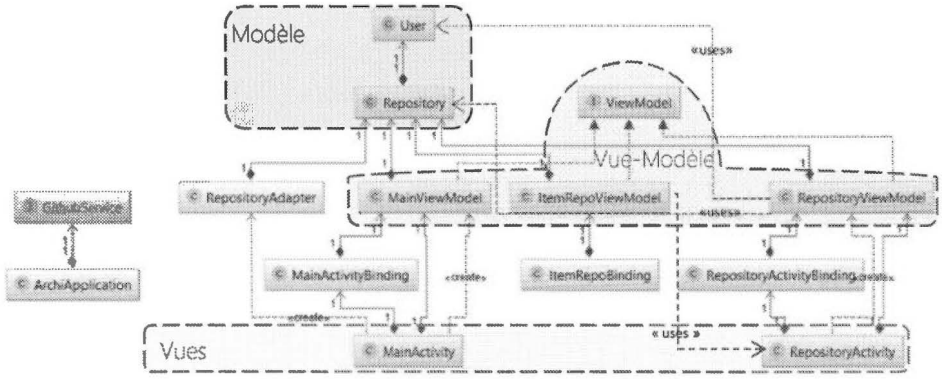


Figure 3.4: Diagramme de classes de la version MVVM de l'application Archi

À partir de ce simple exemple, on peut voir que l'identification du patron appliqué nécessite :

1. l'identification des entités représentant les classes intermédiaires entre l'utilisateur et l'application (ex. les Activités) ;
2. l'analyse des différentes manières d'accès et de manipulation des classes de la logique d'affaires par ces entités.

Les sections qui suivent expliquent en détails comment nous nous basons sur cette approche pour inférer le style architectural utilisé tout en référant à chaque fois à cet exemple.

3.2 Aperçu sur l'approche proposée

Notre approche RIMAZ vise à identifier les patrons architecturaux de type MVC dans les applications Android. RIMAZ se base sur un nombre d'heuristiques qui décrivent les différentes implémentations potentielles des patrons basés sur MVC dans les applications Android. En utilisant ces heuristiques, RIMAZ identifie le patron de type MVC dominant dans une application Android. Nous nous focalisons spécifiquement sur le patron MVC et deux de ses variantes, à savoir MVP et MVVM.

Développé en Java, RIMAZ s'appuie sur la plateforme SOOT[4] pour analyser le bytecode Dalvik se trouvant dans les fichiers compressés APK (Vallée-Rai *et al.*, 1999). SOOT convertit le bytecode Dalvik en une représentation intermédiaire plus facile à comprendre et à analyser. De ce fait l'outil RIMAZ supporte à la fois les applications développées en Java ou en Kotlin, car ces deux derniers génèrent du bytecode après l'étape de compilation.

L'approche RIMAZ comporte plusieurs étapes pour identifier lequel des patrons MVC, MVP ou MVVM domine dans une application Android. Comme le montre la figure 3.5, les étapes 1 à 3 représentent des étapes de pré-traitement qui visent à récupérer à partir des applications analysées les composantes et les classes nécessaires pour notre outil de détection. L'étape 4, en revanche, est l'étape de traitement où nous appliquons des algorithmes de détection sur l'application Android basés sur des heuristiques afin d'identifier le patron dominant dans une application Android. Nous détaillons dans la suite chacune de ces étapes.

3.3 Étape 1 : Filtrage du code source de l'application

Cette étape consiste à différencier entre le code source de l'application en cours d'analyse et le code source appartenant aux librairies utilisées. Cette opération de séparation représente un défi souvent souvent rencontré lors de l'analyse du code source des applications Android. Ceci est dû au fait que les fichiers APK contiennent le bytecode du code source de l'application ainsi que celui des librairies utilisées.

L'analyse automatique de la totalité du contenu du fichier APK contenant des librairies tierces augmente le temps d'exécution et embrouille l'algorithme d'analyse, engendrant des résultats moins précis (Li *et al.*, 2015). Afin de résoudre

4. Source : `https://github.com/Sable/soot`

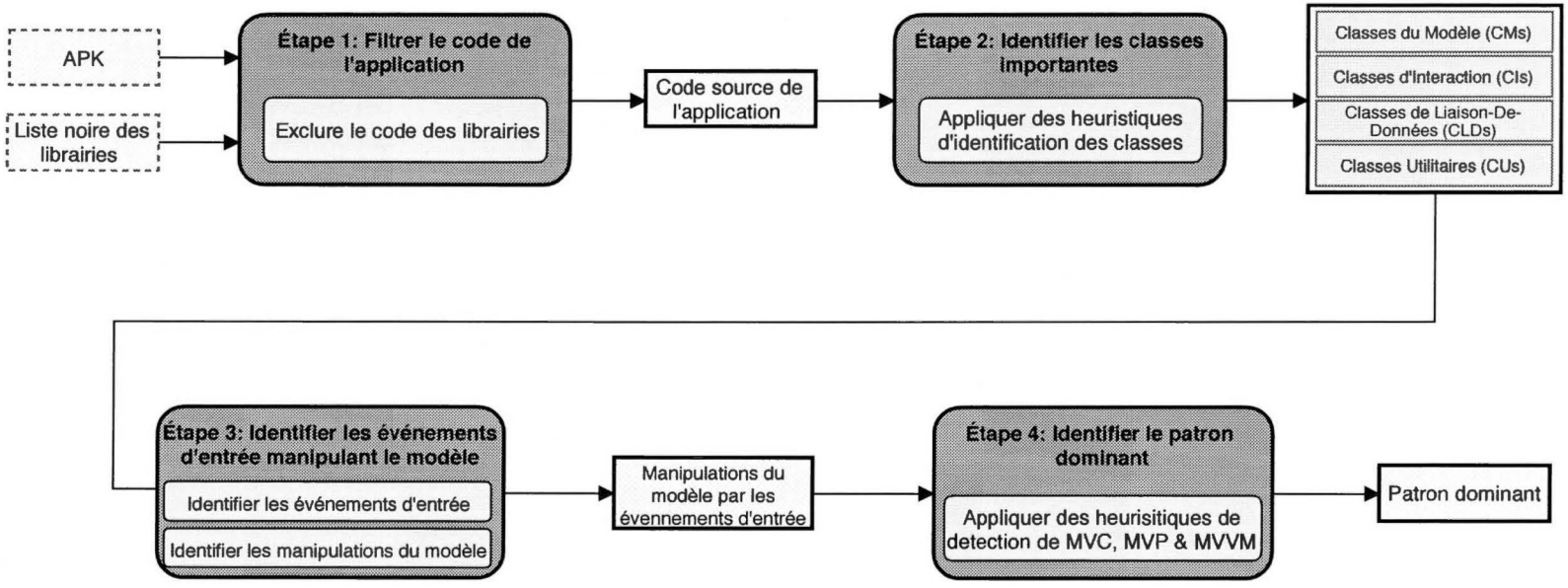


Figure 3.5: Aperçu sur l'approche RIMAZ pour détecter le patron de type MVC dominant dans une application Android

ce problème, quelques approches utilisent le nom du paquet principal se trouvant dans le fichier manifest de l'application afin de restreindre l'analyse aux classes de ce paquet. Ce nom de paquet est utilisé pour exclure tout code source et tout type dont le nom ne débute pas par ce nom. Cette approche souffre cependant de deux limitations :

1. Parfois, certains développeurs changent volontairement le nom du paquet et utilisent un nom complètement différent dans le fichier manifest ;
2. Plusieurs projets Android peuvent contenir plusieurs dossiers racines. Utiliser seulement le nom du paquet récupéré depuis le fichier manifest pourrait certainement écarter les autres dossiers.

Jusqu'à maintenant, la meilleure manière pour contourner ce problème serait d'utiliser une liste noire contenant les noms de paquets des librairies les plus utilisées dans les applications Android. Li *et al.* (Li *et al.*, 2015) ont conduit une étude sur un grand jeu de données d'applications Android et ont pu construire une liste contenant les noms de paquets de plus de 1350 librairies parmi les plus utilisées dans les applications Android. La figure 3.6 montre les 20 librairies les plus populaires avec le nombre d'applications qui les utilisent dans l'étude de Li *et al.*

Cependant, cette liste n'a pas été mise à jour depuis l'année 2015. Nous avons avons donc procédé à la mise à jour de cette liste en l'étendant par une autre liste contenant plus de 1150 noms de paquets de librairies largement utilisées par la communauté. Nous avons collecté cette liste d'une manière manuelle à partir des sites web communautaires tels que *Android Arsenal*[5] et *Ultimate Android Reference*[6].

---

5. Source : `https://android-arsenal.com/`

6. Source : `https://github.com/aritraroy/UltimateAndroidReference/`

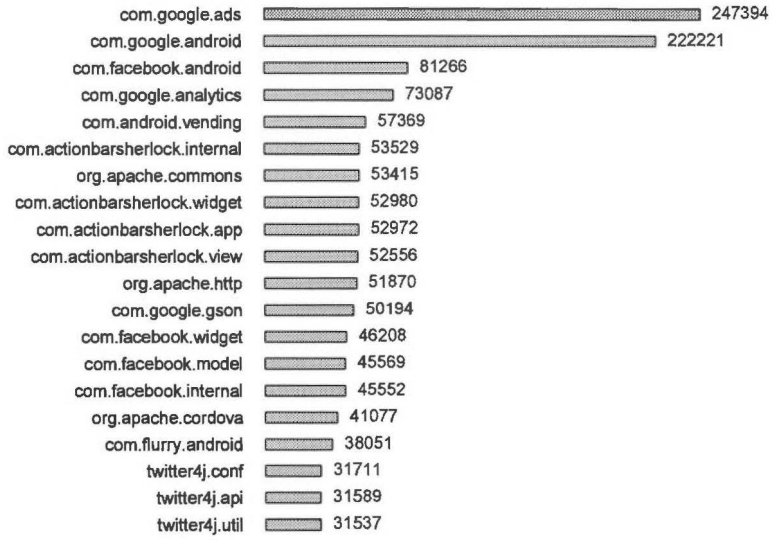


Figure 3.6: Popularité des 20 librairies les plus utilisées dans l'étude de Li *et al.* (Li *et al.*, 2015)

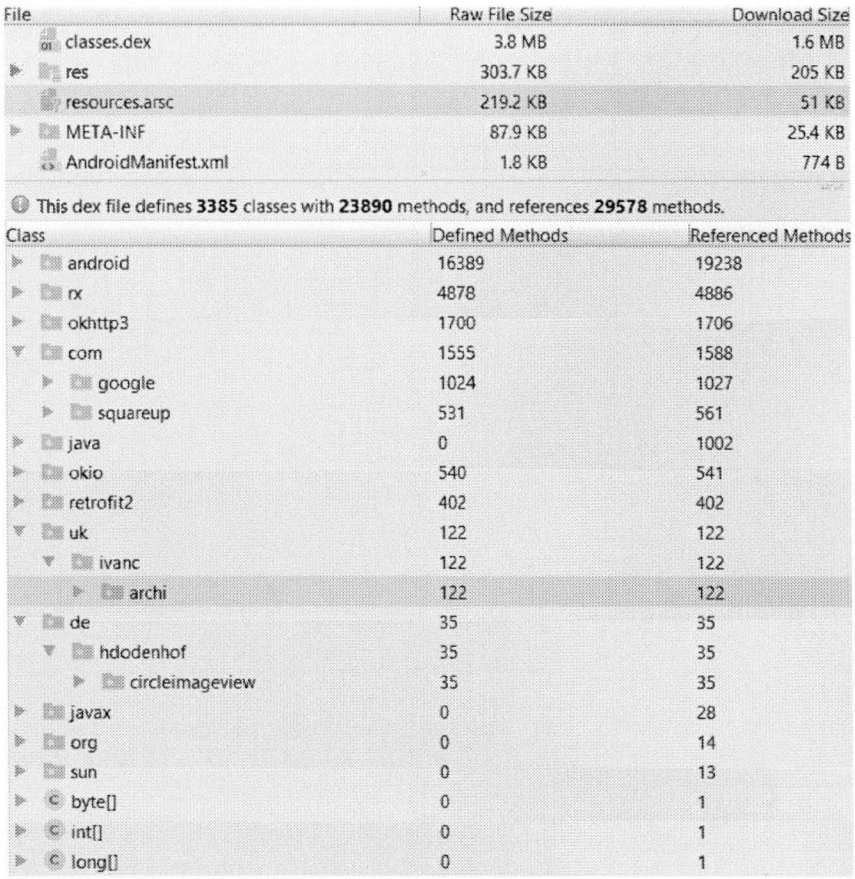
Si nous appliquons cette étape sur l'application Archi, beaucoup de librairies seraient ignorées telles que la librairie *okhttp* et *retrofit*. La figure 3.7 montre les différents paquets contenus dans le fichier dex de l'application Archi. En particulier, seul le paquet principal `uk.ivanc.archi` serait considéré dans l'analyse de l'application Archi.

3.4 Étape 2 : Identifications des classes importantes

Tel que détaillé dans la section 1.3, les patrons architecturaux basés sur MVC visent à réaliser un découplage entre l'interface utilisateur et les entités représentant le Modèle. Pour déterminer lequel de ces patrons est utilisé dans une application, nous devons identifier les classes jouant certains rôles dans l'implémentation de ces patrons. En particulier, nous devons distinguer entre : *(i)* les classes à travers lesquelles l'utilisateur interagit avec l'application, *(ii)* les classes représentant les entités du Modèle, *(iii)* les classes responsables de la liaison-de-

---

`blob/master/README.md`



File	Raw File Size	Download Size
classes.dex	3.8 MB	1.6 MB
res	303.7 KB	205 KB
resources.arsc	219.2 KB	51 KB
META-INF	87.9 KB	25.4 KB
AndroidManifest.xml	1.8 KB	774 B

This dex file defines **3385** classes with **23890** methods, and references **29578** methods.

Class	Defined Methods	Referenced Methods
android	16389	19238
rx	4878	4886
okhttp3	1700	1706
com	1555	1588
google	1024	1027
squareup	531	561
java	0	1002
okio	540	541
retrofit2	402	402
uk	122	122
ivanc	122	122
archi	122	122
de	35	35
hdodenhof	35	35
circleimageview	35	35
javax	0	28
org	0	14
sun	0	13
byte[]	0	1
int[]	0	1
long[]	0	1

Figure 3.7: Les différents paquets contenus dans le fichier APK de l'application Archi

données, *(iv)* les objets visuels personnalisés et les autres classes restantes appelées classes utilitaires.

3.4.1 Identification des classes d'interaction

En général, dans le contexte des applications Android, les classes qui supportent l'interaction de l'utilisateur sont : les Activités, les Fragments, les Dialogues ainsi que les classes Applications. Dans ce travail, nous appelons ce type de composantes les *Classes d'Interaction* (**CIs**). L'heuristique qui aide à les identifier dans une application donnée est définie comme suit :

- Une Activité, un Fragment, un Dialogue ou une classe Application est toute classe qui a au moins un type parent ayant un nom entièrement qualifié (*Fully Qualified Name*) se terminant par la signature `.app.Activity`, `.app.Fragment`, `.app.Dialog` ou `.app.Application` respectivement.

La justification de cette heuristique est basée sur le raisonnement suivant : les types parents de ces composantes peuvent être sous deux formes : les types appartenant au paquet `android.app.*`, ou bien les types provenant des librairies de support et qui ont le format `android.support.vX.app.*`.

Par exemple, une classe Fragment peut étendre la classe `android.app.Fragment` ou la classe `android.support.v4.app.Fragment`, selon le choix du développeur.

En effet, lorsque les développeurs visent plusieurs versions d'API Android, il se peut qu'ils utilisent des types parents fournis par les librairies de support (Android-Doc, 2018m). Ces dernières sont fournies avec la plateforme Android permettant aux développeurs de rajouter de nouvelles fonctionnalités à leurs applications tout en restant supportées par les versions antérieures d'Android (Android-Doc, 2018m).

En général, les classes fournies par les librairies de support ont un nom entièrement qualifié qui commence par `android.support.vX` où `X` est la version de la librairie de support.

3.4.2 Identification des classes du Modèle

Les *Classes Modèles* (**CMs**) sont des classes décrivant l'état de la logique d'affaire qui peut être persistant ou pas. Dans une application Android, ces classes peuvent être implémentées sous plusieurs formes ; elles peuvent être des simples *classes Bean* ou peuvent être développées à l'aide des librairies *MORs*[7], des librairies

7. Mappage Objet-Relationnel (en anglais, ORM (*Objet-Relational Model*))

de sérialisation ou des librairies de base de données SQL légères. Pour couvrir ces différentes possibilités, nous avons d'abord construit à partir des sites web communautaires tels que *Android Arsenal* et *Ultimate Android Reference* une liste contenant un nombre de 50 librairies les plus utilisées pour implémenter les CMs. Nous avons appliqué ensuite des heuristiques qui vérifient si ces librairies ont bien été utilisées afin d'identifier les CMs.

Nous ne pouvons, vu le grand nombre de ces méthodes, détailler les différentes heuristiques utilisées. Nous nous sommes basés cependant sur la logique suivante :

- pour les librairies utilisées, nous nous sommes basés sur la détection des différents champs, méthodes et classes ayant un nom entièrement qualifié (*Fully qualified name*) contenant au début le nom du paquet de la librairies utilisée ;
- pour les classes POJO (*Plain Old Java Object*), nous nous sommes basés sur les différentes caractéristiques de ce type de classes, telles que le fait qu'elles contiennent que des constructeurs ainsi que des champs privés ayant des getters et des setters correspondants.

3.4.3 Identification des classes de liaison de données

La liaison de données (*Data Binding*) permet encore plus de découplage entre l'interface utilisateur et la logique d'affaires. Le patron MVVM en particulier, repose sur la liaison de données pour découpler la Vue et le VueModèle. Lors de l'utilisation des techniques de liaison de données dans Android, des classes étendant le type `ViewDataBinding` sont automatiquement générées durant la compilation et traitées durant l'exécution afin d'effectuer les liaisons désirées. Dans ce travail, nous appelons ce type de classes, les *Classes de Liaison-De-Données* (**CLDs**). Nous utilisons l'heuristique suivante pour les identifier dans une application Android :

- Une CLD, est n'importe quelle classe qui étend la classe `android.databinding.ViewDataBinding`.

3.4.4 Identification des objets visuels personnalisés

La plateforme Android offre une multitude d'objets visuels pour programmer des interfaces utilisateurs. Cependant, les développeurs tendent des fois à créer leurs propres objets personnalisés pour différentes raisons. Afin d'avoir des résultats de détection précis, nous prenons en considération l'identification de ces objets. La convention dans le monde Android est d'appeler les objets constituant l'interface utilisateur tels que les boutons (Button), les champs de texte (EditText), etc. des *Vues* (*Views*), puisqu'ils sont tous des sous-types de la classe `android.view.View` (Android-Doc, 2018b). Ce terme (Vue) peut être confondu avec le terme Vue utilisé dans les patrons architecturaux discutés dans ce travail. Pour éviter cette confusion, nous avons décidé d'utiliser l'appellation ÉlémentIU (abrégée **EIU**) pour tous les objets des interfaces utilisateurs. L'heuristique qui aide à identifier ce type de classes est définie comme suit :

- Un EIU est n'importe quelle classe qui :
  - étend la classe `android.view.View`; ou
  - étend une classe interne de toute classe étendant `android.view.View`.

3.4.5 Identification des classes utilitaires

Nous considérons comme classes utilitaires (*Helper Classes*), toutes les classes qui ne sont ni des CIs, ni des CMs, ni des CLDs et ni des EIUs, et nous les symbolisons par (**CUs**). Dans ce travail, le concept de classe utilitaires ne reflète pas forcément la signification classique d'une classe utilitaire dans la programmation orientée objet. Dans notre contexte, les CUs sont généralement des classes qui sont utilisées par les CIs et qui ne jouent en aucun cas un rôle significatif dans un des patrons

architecturaux MVC, MVP ou MVVM.

Dans l'exemple de l'application Archi, nous détectons tous les éléments essentiels mentionnés dans cette section. Ces éléments diffèrent d'une architecture à une autre. En effet, comme mentionné dans le tableau 3.1, les CIs qui ont pu être détectées sont la classe `ArchiApplication` ainsi que les Activités `MainActivity` et `RepositoryActivity`. De plus, les entités du Modèles détectés sont les classes `User` et `Repository`. Dans la version implémentée en MVVM, l'approche a détecté trois classes de liaison-de-données CLDs, à savoir, `MainActivityBinding`, `RepositoryActivityBinding` et `ItemRepoBinding`. De plus, la classe `RepositoryAdapter` est considérée comme un EIU, puisqu'elle étend une classe interne de la classe `TextView` qui étend à son tour la classe `android.view.View`. Le reste des classes détectées sont considérées comme des classes utilitaires CUs.

3.5 Étape 3 : Identification des évènements d'entrée manipulant le Modèle

En plus de connaître les classes significatives dans une application Android, nous avons besoin d'analyser les interactions entre ces classes. Ceci nous aide à identifier lequel des patrons MVC, MVP ou MVVM est appliqué. Les évènements déclenchés par l'interaction de l'utilisateur représentent le mécanisme qui influence le plus la communication entre ces composantes. Dans la documentation officielle de la plateforme Android, ce type d'évènements est appelé évènements d'entrée (*Input Events*) (Android-Doc, 2018j).

De ce fait, nous avons besoin de savoir où ces évènements sont déclenchés et comment ils sont propagés aux CMs spécifiquement. En effet, identifier les évènements d'entrée, leurs endroits de déclaration et/ou de déclenchement, ainsi que la manière dont ils accèdent et manipulent les CMs, est au cœur de notre approche pour identifier le patron architectural utilisé.

Tableau 3.1: Identification des classes importantes dans les différentes implémentations de l'application Archi

	CIs	CMs	CLDs	CUs	EIUs
MVC	• MainActivity • RepositoryActivity • ArchiApplication	• Repository • User	-	-	RepositoryAdapter
MVP			-	• MainPresenter • RepositoryPresenter	
MVVM			• MainActivityBinding • RepositoryActivityBinding • ItemRepoBinding	• MainViewModel • RepositoryViewModel • ItemRepoViewModel	

Nous présentons dans cette section, les trois parties principales de cette étape, à savoir : (*i*) la détection des évènements d'entrée, (*ii*) la détection des manipulations des entités du Modèle par ces évènements, et finalement, (*iii*) la classification de ces manipulations selon l'emplacement des évènements. Cette dernière étape sert à différencier entre les différentes manipulations selon l'endroit de définition de chaque évènement initiant des manipulations des entités du Modèle.

3.5.1 Identification des évènements d'entrée

Dans la documentation officielle de la plateforme Android, les évènements d'entrée sont catégorisés en deux types :

1. Les Évènements d'Entrée à base d'Écouteurs : ces évènements sont déclenchés par les ÉlémentIUs et sont implémentés en utilisant le patron Écouteur (*Listener pattern*) (Android-Doc, 2018j). Dans tout ce qui suit, nous utiliserons l'abréviation **EEE**s pour faire référence à ce type d'évènements.
2. Gestionnaires d'évènements (*Event Handlers*) : ce type d'événements est représenté par des méthodes redéfinies (*Overriden*) en étendant d'autres classes mères ou en implémentant certaines interfaces de la plateforme Android. Ces méthodes, sont invoquées par Android quand des actions spécifiques sont déclenchées (Android-Doc, 2018j). Les gestionnaires d'évènements peuvent être :
   - Des gestionnaires d'évènements de cycle de vie : tels que les méthodes `onCreate`, `onStart`, `onResume`, `onPause`, `onStop`, `onDestroy` et `onRestart` d'une Activité par exemple ;
   - Des gestionnaires d'évènements qui ne gèrent pas les cycles de vie : Ceux-ci représentent des actions spécifiques différentes, telles que la méthode `onKeyDown`. Cette dernière est appelée lorsque l'utilisateur appuie sur l'un des boutons de volume de son appareil Android.

a) Identification des Évènements d'Entrée à base d'Écouteurs

D'un coté, une des spécificités du patron Écouteur dans le monde Java en général et dans le monde d'Android plus spécifiquement est que les écouteurs sont à la base des types abstraits représentant des contrats. Ils sont généralement définis comme étant des types internes dans l'objet qui déclenchent l'évènement en question (Android-Doc, 2018j). D'un autre coté, certains développeurs, lorsqu'ils développent des composantes personnalisées omettent cette pratique et tendent à déclarer ces contrats comme des objets externes.

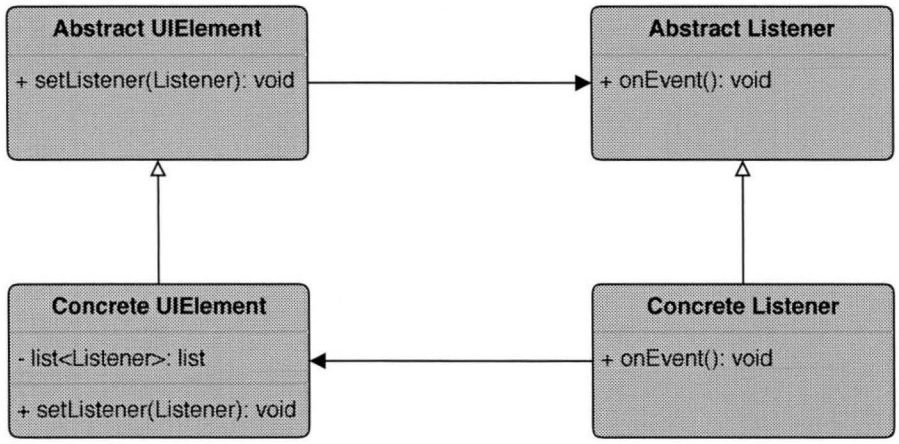


Figure 3.8: Diagramme de classes du patron Écouteur

De ce fait, et pour identifier les EEEs, nous nous basons principalement sur les caractéristiques du patron Écouteur. Ce dernier est une implémentation particulière du patron Observateur (Gamma *et al.*, 1995). La figure 3.8 décrit le fonctionnement de ce patron qui est constitué de deux éléments principaux :

i Un contrat représenté par une interface ou une classe abstraite (`AbstractListener`, figure 3.8). Réimplanté après par un objet écouteur (`ConcreteListener`, figure 3.8), il contient une ou plusieurs méthodes qui sont invoquées quand l'évènement est déclenché (`onEvent()`, figure 3.8) ;

ii Un objet ÉlémentIU par lequel l'événement est déclenché. Il doit avoir un champ représentant l'écouteur (ou une collection d'écouteurs), ainsi qu'une méthode qui définit les écouteurs de cet objet.

Nous n'omettons pas, cependant, la caractéristique de définir les contrats comme des classes internes, nous l'utilisons dans nos tests pour accélérer les calculs de détections quand ces cas sont rencontrés.

L'Algorithme 1, décrit les étapes suivies pour détecter des EEEs. Pour qu'il détecte la présence de l'utilisation des EEEs, l'algorithme a besoin d'avoir comme entrée une ligne de code qui contient :

- l'objet EIU concerné pour lequel cet évènement est défini et qui est un sous type de la classe `"android.app.View"`.
- la méthode que cet objet appelle, pour injecter et définir l'écouteur implémenté.

Pour optimiser l'algorithme, nous exploitons le fait que souvent les écouteurs sont définis comme des classes internes aux ÉlementIUs (lignes 10 à 14 de l'algo). Dans ce cas, l'algorithme teste si le type externe est un EIU ou une classe interne d'un EIU (lignes 11 à 13 de l'algo). Ainsi il retourne une utilisation d'un EEE.

Dans l'autre cas (lignes 15 à 18 de l'algo), si l'écouteur n'est pas une classe interne, l'algorithme teste tout simplement la présence du patron écouteur décrit dans la figure 3.8 (lignes 16 à 18 de l'algo).

b) Identification des gestionnaires d'évènements

Les gestionnaires d'évènements sont généralement offerts par un ensemble de types que le développeur doit étendre pour définir le gestionnaire d'évènement désiré. En plus de ça, des composantes telles que les CIs et les ÉlémentIUs offrent leurs propres gestionnaires d'évènements (tels que les évènements de cycle de vie). Pour

Algorithme 1 : Détection d'un Évènement d'entrée à base d'écouteur

```
1  Entrée : Ligne de code, un objet Obj invoquant une de ces méthodes M;
2  Sortie : EEE : Évènement d'Entrée à base d'Écouteur;
3    Définir :
4    EIU : ElementIU, objet étendant la classe "android.app.View";
5    MI : Méthode Initialisatrice, méthode publique non statique ayant un
       seul paramètre de type référence;
6  si (Obj est un EIU) et (M est une MI) alors
7      P ← Paramètre passé dans MI;
8      si P est défini dans l'application alors
9          L ← Types parents de P;
10         si Un des L est un type interne alors
11             O ← Type externe de L;
12             si O est un EIU ou une classe interne de EIU alors
13                 retourner new EEE(O, MI, L, P);
14             fin
15         sinon
16             S ← Types parents de Obj;
17             si (Un des S a un champ ayant le même type que celui de l'un
                 des L) ou (Un des S contient un champ générique ayant un
                 type paramètre identique à celui de l'un des L) alors
18                 retourner new EEE(Obj, MI, L, P);
19             fin
20         fin
21     fin
22 fin
```

identifier ces évènements, nous avons recueilli, à partir de la documentation Android, une liste de tous les types parents potentiels pouvant avoir des gestionnaires d'évènements. Cette liste inclut les types suivants :

- L'interface `KeyEvent.Callback`;
- L'interface `Application.ActivityLifecycleCallbacks`;
- L'interface `Window.Callback`;
- L'interface `LayoutInflater.Factory`;
- L'interface `LayoutInflater.Factory2`;
- L'interface `ComponentCallbacks`;
- L'interface `ComponentCallbacks2`;
- L'interface `View.OnCreateContextMenuListener`.

Nous considérons comme gestionnaire d'événements, toute méthode qui :

- redéfinit une des méthodes des types précédents; ou
- représente une méthode de gestion de cycle de vie.

De plus, certaines composantes telles que les CIs par exemple ont leurs propres méthodes qui représentent des gestionnaires d'évènements spécifiques, mais qui ne sont pas des méthodes de cycles de vie. Nous nous basons sur les caractéristiques suivantes pour les identifier :

- la méthode n'est pas une méthode de gestion du cycle de vie; et
- elle doit être une méthode d'instance (*c-à-d.* qui n'est pas statique) qui est publique ou protégée (`public` ou `protected`); et
- Le nom de la méthode commence avec le préfixe `On`.

À titre d'exemple, la méthode `onBackPressed` de la classe `Activity` est considérée comme un gestionnaire d'évènements. Celle-ci est appelée lorsque l'Activité a détecté que l'utilisateur a appuyé sur la touche retour de l'appareil (Android-Doc,

2018a).

Le listing 3.1 illustre une portion du code source de l'application Archi dans sa version MVC. Elle appartient au gestionnaire d'évènement `OnCreate` de l'Activité `MainActivity` et définit quelques évènements qui gèrent l'interaction de l'utilisateur avec le premier écran (figure 3.1 (a)). La ligne numéro `6` par exemple définit l'évènement du clique du bouton `searchButton`, qui déclenche la recherche des entrepôts pour un nom d'utilisateur introduit dans le champ de texte `editTextUsernam`. Cet évènement est à base de l'écouteur `OnClickListener` qui est une interface interne de la classe `android.view.View`.

De même, les lignes numéro `14` et `15` définissent les évènements de l'interaction avec le champs de texte `editTextUsername`. Le premier est déclenché lorsque l'utilisateur change la valeur textuelle, tandis que le second est déclenché lorsqu'une action particulière est effectuée sur cet objet. Étant aussi à base d'écouteurs, le premier utilise une extension de la classe `android.text.TextWatcher` (Listing 3.2), tandis que le deuxième redéfinit l'écouteur `TextView.OnEditorActionListener` qui est une interface interne de la classe `android.widget.TextView` qui est elle même une descendante de la classe `android.view.View`.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    // Set up search button
    searchButton = (ImageButton) findViewById(R.id.button_search);
    searchButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            loadGithubRepos(editTextUsername.getText().toString());
        }
    });
    //Set up username EditText
    editTextUsername = (EditText) findViewById(R.id.edit_text_username);
    editTextUsername.addTextChangedListener(mHideShowButtonTextWatcher);
    editTextUsername.setOnEditorActionListener(new TextView.OnEditorActionListener() {
        @Override
        public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
            if (actionId == EditorInfo.IME_ACTION_SEARCH) {
                String username = editTextUsername.getText().toString();
                if (username.length() > 0) loadGithubRepos(username);
                return true;
            }
            return false;
        }
    });
    ...
}
```

Listing 3.1: Définition des EEEs dans la version MVC de l'application Archi (Classe MainActivity)

```
1     private TextWatcher mHideShowButtonTextWatcher = new TextWatcher() {
2     @Override
3     public void beforeTextChanged(CharSequence charSequence, int start, int count, int after) {
4     }
5
6     @Override
7     public void onTextChanged(CharSequence charSequence, int start, int before, int count) {
8         searchButton.setVisibility(charSequence.length() > 0 ? View.VISIBLE : View.GONE);
9     }
10
11    @Override
12    public void afterTextChanged(Editable editable) {
13    }
14 }
```

Listing 3.2: Implémentation de l'écouteur `TextWatcher`

En appliquant notre approche sur le code source présenté dans le listing 3.1, nous détectons les évènements suivants :

- un gestionnaire d'évènement `OnCreate` ;
- un évènement à base de l'écouteur `OnClickListener` ;
- un évènement à base de l'écouteur `addTextChangedListener` ;
- un évènement à base de l'écouteur `TextWatcher`.

3.5.2 Identification des manipulations du Modèle par les évènements d'entrée

Comme précisé au début de cette section, l'identification des manipulations des entités du Modèle par les évènements d'entrée est importante pour inférer le style architectural appliqué. La manipulation d'une CM peut être réalisée en récupérant ses informations (lecture) ou en les modifiant (écriture). Un évènement d'entrée peut manipuler en lecture une CM donnée en :

- récupérant la valeur d'une instance d'une CM ; ou

- récupérant la valeur d'un champ d'instance ; ou
- récupérant la valeur d'un champ statique ; ou
- appelant une de ses méthodes *getters*.

Similairement, un évènement d'entrée peut modifier les informations d'une CM en :

- performant des initialisations ; ou
- modifiant la valeur d'un champ d'instance ; ou
- modifiant la valeur d'un champ statique ; ou
- appelant une de ses méthodes *setters* ; ou
- appelant toute autre méthode statique ou d'instance qui est ni un *getter*, ni un *setter*.

Dans notre approche, les manipulations du Modèle sont identifiées en utilisant un nombre d'heuristiques. Dépendamment du patron architectural appliqué, les évènements d'entrée peuvent manipuler les classes du Modèle directement ou indirectement. Par conséquent, nous distinguons et identifions les situations suivantes :

- **Une manipulation directe :** Dans ce cas, la manipulation de la CM est effectuée dans le corps de la méthode d'exécution de l'évènement. Nous appelons une *méthode d'exécution* toute méthode qui est invoquée lorsqu'un EEE est déclenché, ou bien toute méthode redéfinie dans le cas d'un gestionnaire d'évènements. Dans le listing 3.1 par exemple, la méthode `OnCreate()` est elle-même une méthode d'exécution car c'est un gestionnaire d'évènements. De même, la méthode `onClick(View v)` est une méthode d'exécution de l'écouteur `View.OnClickListener` ;
- **Une manipulation indirecte :**
  - Une manipulation via une méthode appartenant à la même classe : Ceci est le cas d'une manipulation indirecte effectuée à travers l'appel d'une autre méthode appartenant à la même classe ;

— Une manipulation via une autre classe : Qui représente aussi une manipulation indirecte de la CM en appelant des méthodes appartenant à d'autres classes telles que les CUs ou les CLDs.

Dans la version MVC de l'application Archi, aucune manipulation du Modèle n'est effectuée dans les méthodes d'exécutions de l'évènement présenté dans le listing 3.2. Par contre, le listing 3.1, montre que les méthodes d'exécution `onEditorAction` (ligne 17) et `onClick` (ligne 8) font toutes les deux appel à la méthode `loadGithubRepos` (appartenant à la même classe `MainActivity`). Le listing 3.4 montre un extrait de la méthode `loadGithubRepos` qui manipule l'entité `Repository` (ligne 15).

La méthode `OnCreate` de la classe `RepositoryActivity` cependant effectue une manipulation directe comme montré dans les lignes 4 et 6 du listing 3.3. Le listing 3.3 montre la méthode `OnCreate` de la classe `RepositoryActivity`. Cette méthode effectue une manipulation directe de la CM Repository (lignes 4 et 6).

```
1     @Override
2  protected void onCreate(Bundle savedInstanceState) {
3     ...
4     Repository repository = getIntent().getParcelableExtra(EXTRA_REPOSITORY);
5     bindRepositoryData(repository);
6     loadFullUser(repository.owner.url);
7  }
```

Listing 3.3: Manipulation directe de la CM `Repository` depuis l'évènement `OnCreate` de la classe `RepositoryActivity`

```
1  public void loadGithubRepos(String username) {
2      ...
3      ArchiApplication application = ArchiApplication.get(this);
4      GithubService githubService = application.getGithubService();
5      subscription = githubService.publicRepositories(username)
6                  .observeOn(AndroidSchedulers.mainThread())
7                  .subscribeOn(application.defaultSubscribeScheduler())
8                  .subscribe(new Subscriber<List<Repository>>() {
9                      ...
10                     @Override
11                     public void onNext(List<Repository> repositories) {
12                         Log.i(TAG, "Repos loaded " + repositories);
13                         RepositoryAdapter adapter =
14                                 (RepositoryAdapter) reposRecycleView.getAdapter();
15                         adapter.setRepositories(repositories);
16                         adapter.notifyDataSetChanged();
17                     }
18                 });
19
20 }
```

Listing 3.4: Manipulation de la CM `Repository` à travers la méthode `loadGithubRepos`

La version MVP en revanche utilise dans la méthode `OnCreate` de l'Activité `MainActivity` une manipulation via une autre classe. Comme le montre le listing 3.5, les méthodes d'exécution `OnClick` et `onEditorAction` appellent toutes les deux la méthode `loadRepositories`(Lignes 12 et 22 respectivement) d'un objet de type `MainPresenter`. Cette méthode manipule la classe `Repository` du Modèle.

Dans la version MVVM de l'application, les EEEs sont définis dans les CLDs. Ces EEEs sont automatiquement générés dans les classes CLDs comme décrit dans la section 1.3. Le listing 3.6 montre la définition de l'évènement basé sur l'écouteur `OnClickListenerImpl` dans la version MVVM de l'application. La méthode

d'exécution `onClick` de ce dernier appelle la méthode `onClickSearch` de la classe `MainViewModel`, qui elle même appelle la méthode `loadGithubRepos` qui manipule la classe `Repository` du Modèle, représentant un cas d'une manipulation via une autre classe.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    presenter = new MainPresenter();
    presenter.attachView(this);
    ...
    // Set up search button
    searchButton = (ImageButton) findViewById(R.id.button_search);
    searchButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            presenter.loadRepositories(editTextUsername.getText().toString());
        }
    });
    //Set up username EditText
    editTextUsername = (EditText) findViewById(R.id.edit_text_username);
    editTextUsername.addTextChangedListener(mHideShowButtonTextWatcher);
    editTextUsername.setOnEditorActionListener(new TextView.OnEditorActionListener() {
        @Override
        public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
            if (actionId == EditorInfo.IME_ACTION_SEARCH) {
              presenter.loadRepositories(editTextUsername.getText().toString());
                return true;
            }
            return false;
        }
    });
}
```

Listing 3.5: Définition des EEEs dans la version MVP de l'application Archi

```
1     public static class OnClickListenerImpl implements android.view.View.OnClickListener{
2     private uk.ivanc.archimvvm.viewmodel.MainViewModel value;
3     public OnClickListenerImpl setValue(uk.ivanc.archimvvm.viewmodel.MainViewModel value) {
4         this.value = value;
5         return value == null ? null : this;
6     }
7     @Override
8     public void onClick(android.view.View arg0) {
9         this.value.onClickSearch(arg0);
10    }
11 }
```

Listing 3.6: Imlémentation de l'écouteur `OnClickListener` dans la CLD `MainActivityBinding`

3.5.3 Classifications des manipulations du Modèle selon l'emplacement de la définition des EEEs

Nous pouvons déterminer le patron architectural dominant dans une application, dépendamment des classes dans lesquelles les évènements d'entrée sont définis, ainsi que du type des manipulations que ces évènements effectuent sur les classes du Modèle (directes ou indirectes). Par exemple, lorsque la plupart des manipulations de CMs sont initiées par des EEEs définies dans des CLDs, nous soupçonnons que le patron MVVM a été appliqué dans l'application analysée. Par conséquent, nous devons classer chaque manipulation du Modèle en fonction de l'emplacement de la définition de l'événement d'entrée associé. À ce stade, nous visons spécifiquement les évènements d'entrée à base d'écouteurs. En effet, les gestionnaires d'évènements, comme expliqué dans la section 3.5.1, sont définis dans les classes d'interaction (CIs). Cet emplacement de définition est imposé par la plateforme Android, et est indépendant du patron architectural appliqué. Nous classons donc les manipulations de modèle initiées par des EEEs en deux ensembles :

1. Les manipulations du Modèle initiées par des EEEs déclarées dans des classes

d'`Activités`, de `Fragments` ou de `Dialogues`, (Nous excluons les classes d'application des CI dans cette partie) ;

2. Les manipulations du Modèle initiées par des EEEs déclarées dans des classes utilitaires (CUs) ou de liaison de données (CLDs).

3.6 Étape 4 : Identification du patron dominant

Les développeurs ont tendance à être flexibles quant à leurs implémentations des patrons architecturaux, parfois en les violant ou en les combinant. Notre algorithme est intéressé à révéler le patron architectural dominant dans une application, qui est probablement le choix conscient du concepteur de cette application.

L'approche RIMAZ repose sur un algorithme basé sur des heuristiques pour identifier automatiquement des patrons basés sur l'architecture MVC. Évidemment, quand il n'y a pas de classes du Modèle [8], il n'y a aucun intérêt à chercher des variantes MVC. Cependant, quand notre approche détecte des classes du Modèle, l'algorithme essaie de déterminer le type dominant de manipulations de ces classes. Le diagramme d'activité présenté dans la Figure 3.9 résume l'algorithme avec les différentes heuristiques pour la détection des patrons basés sur MVC.

Comme montré par la figure 3.9 notre algorithme est guidé par la réponse à trois questions. Une heuristique (notée [Hx] dans la figure) correspond à une réponse aux trois questions. Une question centrale dans notre algorithme est la première question de la figure 3.9 : *Où sont définies la plupart des manipulations du modèle initiées par les EEEs ?* Selon la réponse à cette question, notre algorithme procède comme suit :

---

8. Ce cas représente généralement des applications n'ayant pas besoin d'implémenter des classes représentant la logique d'affaire, telles que les petites applications utilitaires.

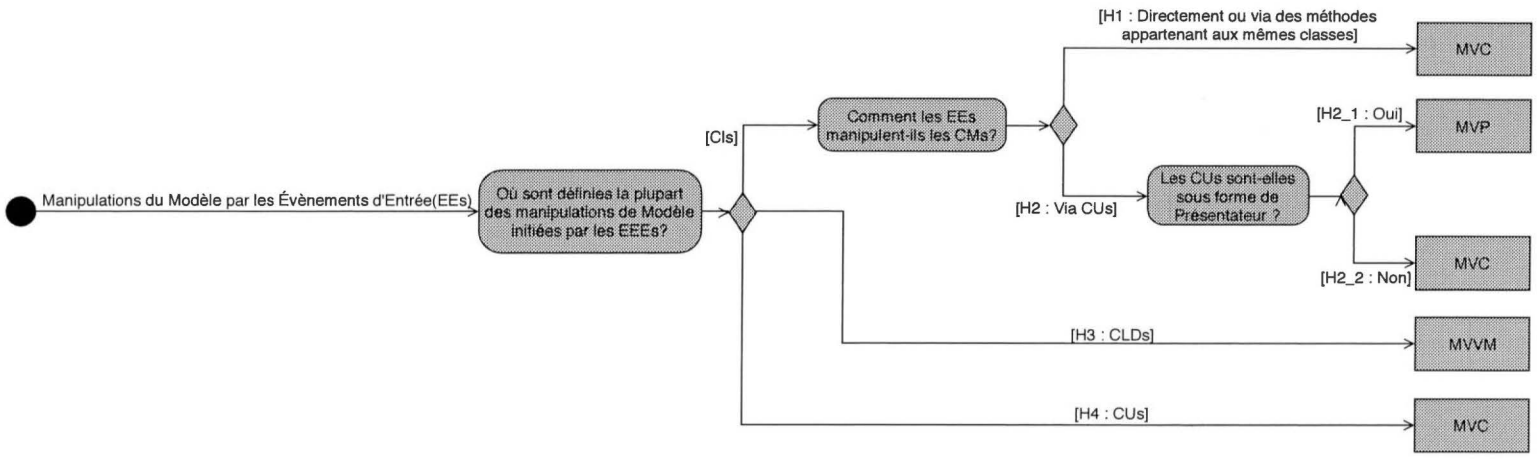


Figure 3.9: Diagramme d'activité résumant l'algorithme de détection des patrons basés sur MVC

1. Cas 1 : Ceci est le cas où la plupart des manipulations du Modèle sont initiées par des EEEs qui ont été définies et/ou déclenchées dans des CIs. Dans ce cas, la détection peut être réduite au patron MVC ou à MVP, un tel cas est illustré par les listings 3.1 et 3.5 dans l'exemple de l'application Archi. Une deuxième question clé est soulevée alors : *Est-ce que la plupart des manipulations du Modèle se produisent dans les CIs directement, via des méthodes appartenant aux mêmes classes ou via des classes utilitaires ?*

   Dans ce cas, où la plupart des manipulations du Modèle sont initiées par des EEEs qui ont été définies et/ou déclenchées dans des CIs, deux heuristiques principales sont appliquées pour pouvoir distinguer entre le patron MVC ou le patron MVP. Au fait, après que nous avons vérifié que la majorité des manipulations du Modèle sont initiées par des EEEs qui sont définies dans des CIs, nous nous intéressons à toutes les manipulations des CMs, y compris celles initiées par des gestionnaires d'évènements.

   La première heuristique **[H1]** : est l'heuristique appliquée lorsque les manipulations du Modèle initiées dans les CIs sont faites directement dans les corps des méthodes d'exécutions des évènements, ou à travers des méthodes appartenant aux mêmes classes. Ceci signifie que le développeur de l'application a considéré les CIs en question, comme des entités du **Contrôleur**, et nous aurions par conséquent, MVC comme un patron dominant. Dans ce cas, la Vue est représentée par les fichiers de disposition *.xml* ainsi que tout autre ÉlémentIU personnalisé, tandis que le Contrôleur est constitué d'Activités, de Fragments et de Dialogues.

   La deuxième heuristique **[H2]** : s'applique lorsque les manipulations du Modèle initiées dans les CIs sont faites via des classes utilitaires CUs. Ceci signifie que les CIs délèguent la manipulation des entités du Modèle à d'autres classes (*c-à-d.* CUs) (par exemple le listing 3.5). Dans ce cas, et comme présenté dans la figure 3.9, deux autres heuristiques sont appliquées (**[H2$_1$]** et

[H2$_2$]), dépendamment de la nature de ces CUs :

- **[H2$_1$]** : si la majorité de ces CUs sont sous la forme d'un Présentateur, alors le patron appliqué est MVP. Les CUs représentent dans ce cas des entités du Présentateur, tandis que la Vue est constituée des classes d'Activités, de Fragments, de Dialogues, leurs fichiers de disposition *.xml* associés, ainsi que tout ÉlémentIU personnalisé. Dans la version MVP de l'application Archi par exemple, et comme le montre le listing 3.5, les EEEs sont définis dans les classes CIs. De plus, ils manipulent les CMs à travers des méthodes d'autres classes telles que `MainPresenter`. Cette dernière a la forme d'une classe Présentateur tel que décrit dans la section 1.3.4.
- La deuxième heuristique **[H2$_2$]** est appliquée si les CUs n'ont pas la forme d'un Présentateur. Dans ce cas ces classes représenteraient des éléments du Contrôleur et le patron appliqué est donc le MVC. Le Contrôleur dans ce cas est constitué de ces classes utilitaires ainsi que toutes les classes d'Activités, de Fragments et de Dialogues, tandis que la Vue est constituée seulement des fichiers de disposition *.xml* et tous les autres ÉlémentIUs personnalisés.

2. Cas 2 : Ceci, est le cas où la plupart des manipulations du Modèle sont initiées par des EEEs, qui n'ont pas été définies et/ou déclenchées par des CIs. Comme présenté par la figure 3.9, deux heuristiques sont appliquées dans cette situation :
   - L'heuristique **[H3]** est appliquée lorsque les EEEs sont majoritairement définies dans des CLDs, ce qui signifie que les évènements ont été déclarés en utilisant les techniques de liaisons de données de la plateforme Android, indiquant que le patron MVVM est potentiellement suivi. Pour le confirmer, trois conditions doivent être vérifiées :
     — La CLD doit avoir un champ privé dont le type est un des CUs ;

— Ce type de CUs est également utilisé pour définir un champ dans une CI ;

— Cette CI doit aussi définir un autre champ ayant le même type que celui de la CLD discutée.

Dans ce cas, le patron MVVM est effectivement utilisé où le VueModèle correspond aux CUs détectés dans les trois conditions précédentes, tandis que la Vue est constituée des classes d'Activités, de Fragments, de Dialogues, leurs fichiers de disposition *.xml* associés ainsi que tout ÉlémentIU personnalisé. Dans la version MVVM de l'application Archi, les EEEs sont définis dans les CLDs tels que présenté dans le listing 3.6. De plus, les trois dernières conditions sont satisfaites et la CU utilisée est la classe `MainViewModel`.

- La deuxième heuristique **[H4]** est appliquée dans le cas où les EEEs sont majoritairement définies dans des CUs. Ce cas est généralement rencontré lorsque les développeurs préfèrent déclarer les EEEs dans des classes utilitaires pour assurer plus de séparation de préoccupations (*separation of concerns (SoC)*), mais sans avoir l'intention d'implémenter MVP ou MVVM. Dans ce cas, nous considérons que le patron MVC a été appliqué. Le Contrôleur est constitué de ces CUs ainsi que des CIs, tandis que la Vue est constituée des fichiers de disposition *.xml* et tous les autres ÉlémentIUs personnalisés.

3.7 Conclusion

Dans ce chapitre, nous avons exposé l'approche RIMAZ qui vise à détecter le patron architectural de type MVC dominant dans une application Android. De plus, nous avons présenté en détail les différentes heuristiques constituant cette approche, toute en les accompagnant par un exemple illustratif pour permettre une meilleure compréhension.

Pour évaluer l'efficacité de notre approche, nous proposons dans le chapitre suivant une évaluation préliminaire où nous expérimentons notre approche sur un ensemble d'applications Android à code source ouvert. Pour valider les résultats de ce teste, nous vérifions ces applications manuellement.

Additionnement, nous proposons dans le chapitre qui suit une étude empirique en appliquant notre outil sur un grand jeux de données d'application Android. Nous étudions les différentes tendances de l'adoption des patrons architecturaux étudiés afin de répondre à nos questions de recherche.

CHAPITRE IV

EXPÉRIMENTATION ET RÉSULTATS

Nous abordons dans ce chapitre les expérimentations que nous avons menées pour tester notre approche RIMAZ et pour répondre à nos questions de recherche. Nous présentons les différents jeux de données des applications que nous avons utilisées pour évaluer notre approche. De plus, nous décrivons les différentes méta-données sur lesquelles nous nous sommes basées pour étudier les différentes tendances d'adoption des patrons architecturaux étudiés. Nous analysons les résultats obtenus et nous présentons les menaces à la validité concernant leur exactitude ainsi que la reproductibilité de nos expérimentations.

4.1 Questions de recherche

Le but de nos expérimentations est d'identifier les différentes tendances en terme d'utilisation des patrons MVC, MVP et MVVM au sein de la communauté du développement mobile. Plus spécifiquement, nos expérimentations visent à répondre aux questions de recherche suivantes :

1. **QR1 :** Quelle est la fréquence d'utilisation des patrons basés sur MVC dans les applications mobiles Android ? L'objectif visé par cette question se décline en deux sous-objectifs :

(a) identifier lequel des patrons MVC, MVP et MVVM est plus utilisé par les développeurs Android.

(b) étudier l'évolution de la fréquence d'utilisation de ces patrons à travers le temps.

2. **QR2 :** Quels sont les types d'applications Android qui utilisent les patrons basés sur MVC ? À travers cette question, nous voulons savoir si l'utilisation de ces patrons a une corrélation avec quelques propriétés des applications mobiles. Plus précisément, nous analysons la corrélation entre la taille et la catégorie d'une application et le patron appliqué. Ceci nous permet d'avoir une idée sur les facteurs qui peuvent affecter le choix d'un patron donné.

4.2 Sujets

Les sujets de notre recherche représentent les trois patrons architecturaux de la couche présentation à savoir le MVC, MVP et MVVM.

4.3 Objets

Pour répondre à nos questions de recherche, nous avons mené une évaluation empirique en appliquant notre approche sur des milliers d'applications obtenus à partir de *Google Play Store*[1]. Nous avons avant cela conduit une étude préliminaire sur un ensemble d'applications Android à code source ouvert pour évaluer la précision de nos algorithmes de détection.

Dans le reste de ce chapitre, nous décrivons en détail notre évaluation préliminaire, notre étude empirique ainsi que la discussion des résultats obtenus et les réponses à nos questions de recherche.

---

1. Source : `https://play.google.com/store`

4.4 Évaluation préliminaire

À notre connaissance, il n'existe aucune étude sur la détection des patrons basés sur MVC dans les applications Android. Avant de mener une étude à grande échelle sur des applications récupérées à partir de Google Play Store, nous avons réalisé une évaluation préliminaire qui vise à évaluer la précision de notre approche outillée RIMAZ. Pour ce faire, nous avons sélectionné d'une manière aléatoire un ensemble de 100 applications à code source ouvert, que nous avons téléchargées à partir de F-Droid, un entrepôt d'applications Android à code source ouvert. Nous avons appliqué notre outil RIMAZ sur chaque application. Nous avons aussi inspecté manuellement le code source de ces applications pour identifier le patron appliqué. Ensuite, nous avons comparé le résultat de notre analyse manuelle avec le résultat donné par Rimaz pour pouvoir évaluer la précision de notre approche.

RIMAZ a réussi à détecter des instances des trois patrons étudiés dans l'ensemble des 100 applications. L'architecture MVC était la plus populaire avec 73% d'utilisation, MVP était rarement utilisé (16%) tandis que MVVM a été détecté seulement une seule fois. 10% des applications cependant n'utilisaient aucun patron.

Pour valider les résultats obtenus avec notre outil RIMAZ, nous avons fait appel à trois experts ayant une bonne connaissance et expérience en matière de développement mobile en général et de la plateforme Android plus précisément. Nous avons demandé à ces experts, d'une manière indépendante, de manuellement inspecter le code source des 100 applications étudiées et de vérifier quel patron (entre MVC, MVP et MVVM) est majoritairement utilisé dans chaque application. Nous avons ensuite comparé leurs résultats avec le résultat obtenu par notre outil automatique RIMAZ. Pour ce faire, nous avons considéré le vote majoritaire entre les avis des experts et le comparer avec le résultat obtenu par notre outil RIMAZ. Les experts devaient étiqueter chaque application avec une des valeurs : MVC, MVP, MVVM

et NONE (Pour les applications n'implémentant aucun patron).

Cette comparaison a révélé que RIMAZ a correctement classifié 90 applications sur les 100 étudiées.

Pour mieux expliquer les résultats obtenus dans cette étape, nous avons calculé quelques métriques telles que la précision, le rappel et la F-mesure. Selon les résultats obtenus par notre validation manuelle, RIMAZ a pu classifié ces applications avec une précision de 85.87% et un rappel de 90.71%. En sachant que la précision correspond au nombre d'éléments pertinents détectés par rapport au nombre total d'éléments détectés, une précision de 85.87% signifie que pour le nombre d'occurrences détectées par l'outil RIMAZ pour chaque patron, 85.87% (en moyenne) de ces occurrences ont été correctement classifiées. De même, le rappel correspond au nombre d'éléments pertinents détectés par rapport au nombre total des éléments réellement pertinents. Ceci signifie que pour tous les patrons réellement utilisés dans les 100 applications, 90.71% (en moyenne) ont été correctement identifiés par l'outil RIMAZ. La valeur de la F-mesure, qui correspond à une moyenne harmonique entre la précision et le rappel et donnée par 88%.

Parmi les 10 applications incorrectement classifiées, deux application étaient classifiées comme des applications utilisant MVC alors qu'elles implémentaient le patron MVP de façon dominante. Ceci est dû à des librairies tierces qui n'ont pas été filtrées au début. En effet, ces librairies contiennent quelques classes qui ont été identifiées comme des CMs et qui sont directement manipulées par plusieurs Activités.

Sept autres applications ont été classifiées comme des applications développées en MVP alors qu'elles implémentaient MVC. Ceci est dû au référencement croisé qui est des fois fait par les développeurs entre la Vue (représenté généralement par des Activités) et le Contrôleur (représenté par des CUs). Ceci n'est pas une

pratique acceptable dans MVC, au contraire c'est une mauvaise pratique qui cause plus de couplage étroit. Ce référencement croisé entre ces classes a la forme de référencement croisé entre les entités de la Vue et les Présentateurs comme présenté dans la figure 1.6, où la Vue référence le Présentateur, et le Présentateur référence la Vue. En effet, les développeurs utilisent parfois des références croisées entre les éléments de la Vue et les éléments du Contrôleur pour pouvoir accéder aux fonctionnalités de chacun à partir de l'autre.

À la fin, une application a été classifiée comme étant une application n'utilisant aucun patron, alors qu'elle implémentait MVC réellement. En effet, l'outil RIMAZ n'aurait pas pu détecté les classes Modèles utilisées dans cette application, d'où le mauvais classement.

4.5 Étude empirique

Afin de réaliser notre étude empirique nous avons suivi un processus donné que nous présentons dans la figure 4.1.

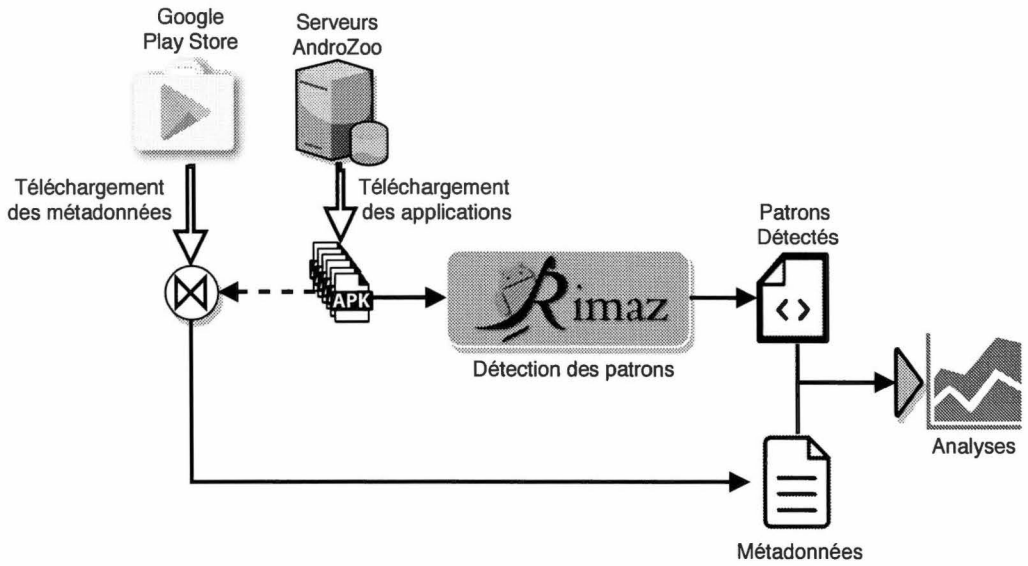


Figure 4.1: Processus d'expérimentation de l'approche outillée RIMAZ

Pour conduire notre étude empirique, nous avons construit un lot d'applications Android accompagnées de métadonnées décrivant leurs propriétés telles que la taille et la catégorie. Nous nous sommes basés sur l'entrepôt *AndroZoo* qui contient des millions d'applications Android. *AndroZoo* est un entrepôt mis à jour d'une manière hebdomadaire qui récupère les applications à partir de plusieurs sources dont *Google Play Store* (Allix *et al.*, 2016).

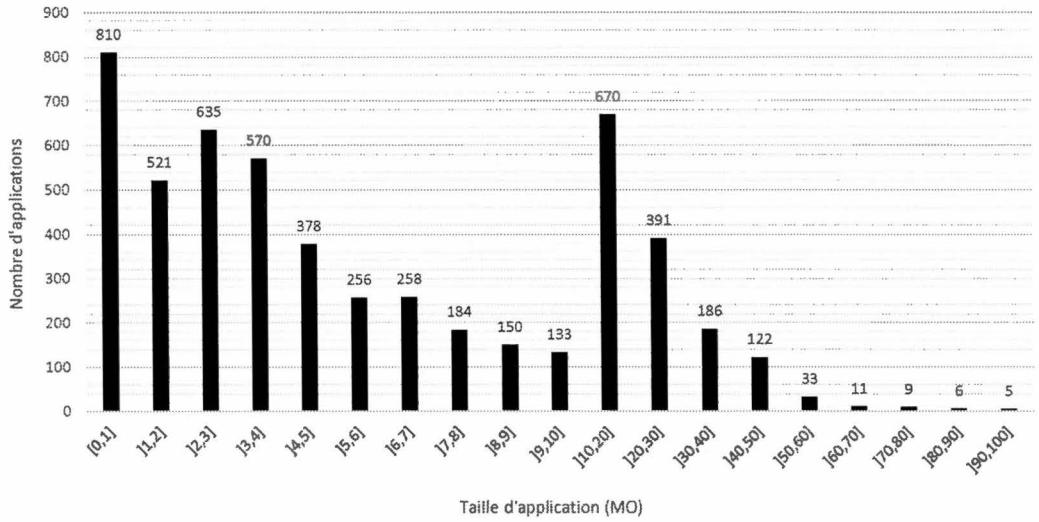


Figure 4.2: Nombre d'applications par taille du fichier Apk

Nous avons sélectionné aléatoirement 5480 applications Android dont la source est le *Google Play Store*, pour pouvoir obtenir les métadonnées précises.

Pour connaitre la distribution des patrons étudiés et pour étudier leurs différentes tendances d'adoption, nous avons considéré les méta-données suivantes :

1. **la taille** : Les applications de notre jeu de données ont des tailles différentes allant de 20 Ko à 100 Mo. Nous mesurons la taille des applications selon trois dimensions :
   - la taille des fichiers Apk ;
   - le nombre de classes ;
   - le nombre de lignes de code (LOC) (*Line Of Code*).

Nous nous attendions à ce que les applications de plus grandes tailles implémentent des patrons architecturaux comparativement aux applications de petites tailles. Les figures 4.2, 4.3 et 4.4 montrent le nombre d'applications par taille de fichiers Apk, nombre de classes et nombre de lignes de code. Dans la première, tout le contenu du fichier est considéré (y compris les librairies utilisées par l'application), tandis que pour les deux autres, les librairies sont omises et les classes ou le nombre de lignes de code ne concernent que les fichiers de l'application ;

2. **la catégorie** : Dans le *Google Play Store*, les applications sont classifiées par catégories. Ces dernières donnent quelques indications sur le domaine d'une application. L'intérêt de considérer ces catégories dans notre analyse, est pour voir s'il y a une relation ou pas, entre l'utilisation d'un patron précis et une des catégories. La figure 4.5 montre le nombre d'applications par catégorie ;

3. **la dernière date de mise à jour** : Un des objectifs de notre étude est d'analyser comment la fréquence d'utilisation d'un patron a évolué avec le temps. Pour cette raison, nous nous ne basons pas sur la date de création des applications mais plutôt sur la dernière date de mise à jour. Ceci est du au fait qu'une application qui aurait pu être créée sans utiliser l'un des patrons étudiés, a évolué en utilisant un de ces patrons. La figure 4.6 montre le nombre d'applications par leurs dates de mise à jours la plus récente.

4.6 Explication des résultats obtenus

Nous rapportons dans cette section les résultats de notre étude empirique et nous répondons à chacune de nos questions de recherche.

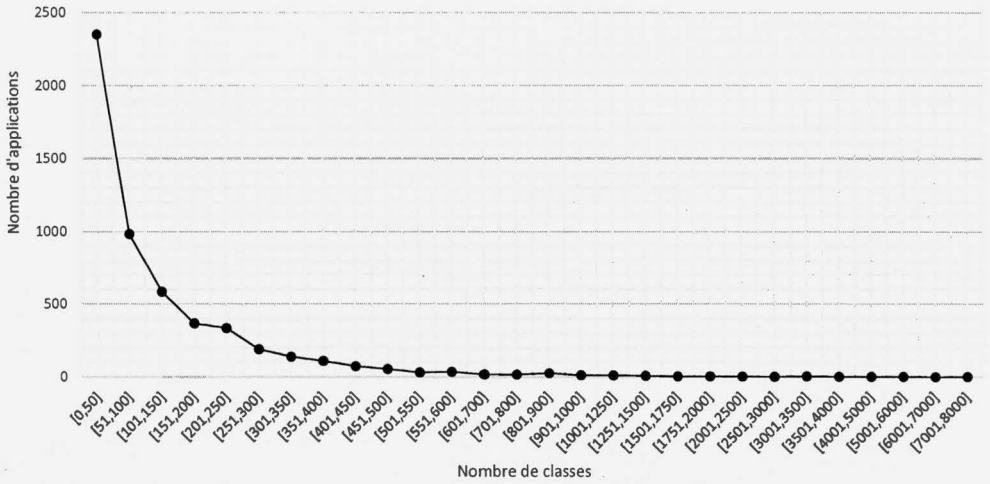


Figure 4.3: Nombre d'applications par nombre de classes

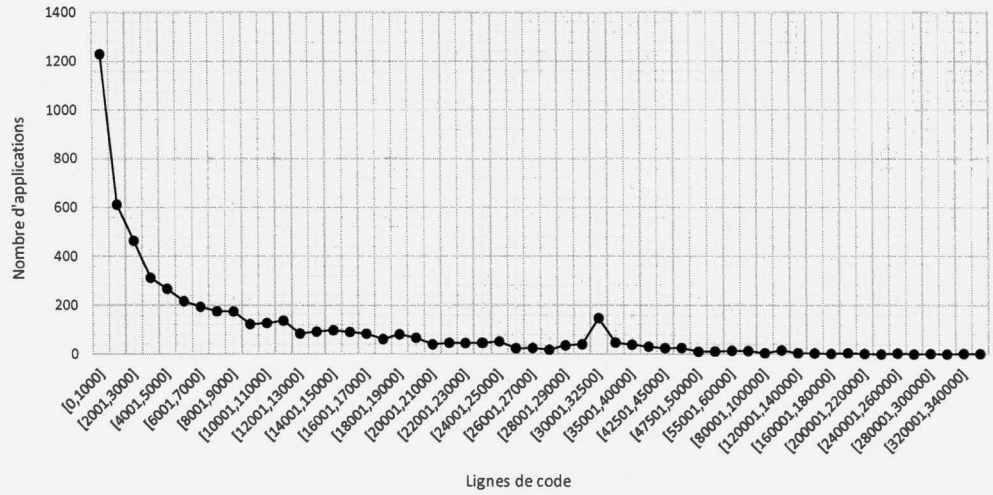


Figure 4.4: Nombre d'applications par nombre de lignes de code

4.6.1 QR1 : Quelle est la fréquence d'utilisation des patrons basés sur MVC dans les applications mobiles Android ?

Nous avons analysé plus de **5000** applications en utilisant notre outil RIMAZ. Les résultats de cette analyse sont présentés dans la figure 4.7.

Comme le montre cette figure, 57% des applications analysées utilisent le patron

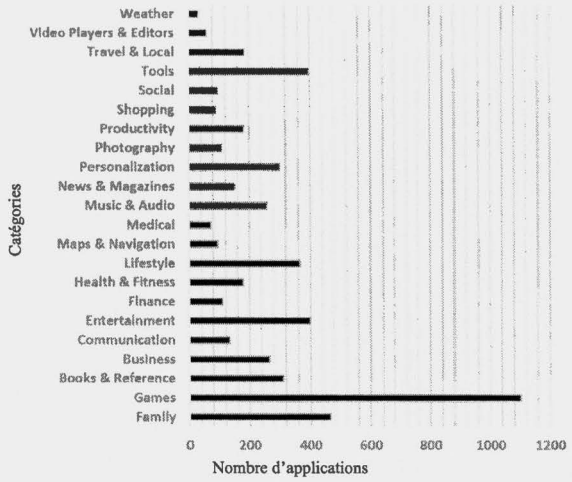


Figure 4.5: Nombre d'applications par catégorie

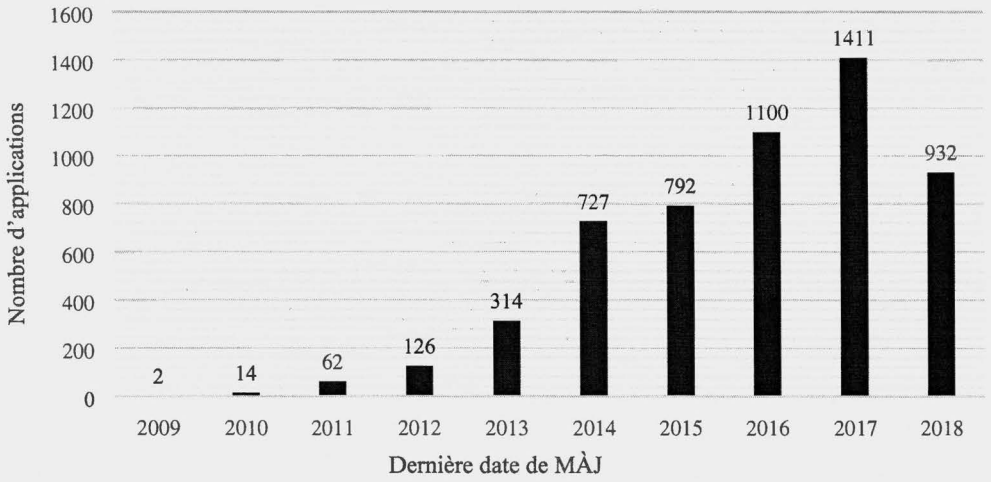


Figure 4.6: Nombre d'applications selon leur dernière année de MÀJ

MVC, 8% utilisent MVP, 35% n'utilisent aucun des patrons étudiés et aucune occurrence du patron MVVM n'a été détectée. Ainsi, le patron MVC est le plus utilisé par les développeurs Android. Ceci peut être expliqué par différents facteurs :

1. le patron MVC était historiquement adopté depuis le début par la communauté Android ;

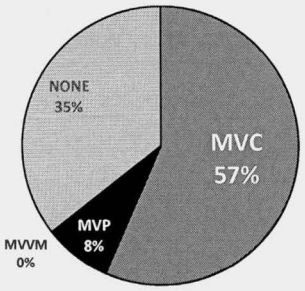


Figure 4.7: Pourcentage des patrons détectés dans 5480 applications Android

2. un très grand nombre d'applications mobiles sont très simples avec un petit nombre d'écrans. MVC offre suffisamment de séparation de préoccupations pour supporter ces applications et est effectivement le patron le plus utilisé pour ce type d'application, comme nous le soulignons dans les parties qui suivent ;
3. les développeurs tendent à déclarer la plupart des évènements d'entrée dans les Activités, ce qui crée naturellement plus de couplage comparativement à ce qui est suggéré par MVP ou MVVM.

Malgré le fait que MVP et MVVM offrent plus de découplage et de séparation de préoccupations, très peu d'applications les utilisent, ceci peut être expliqué par les facteurs suivants :

1. la complexité introduite par les patrons MVP et MVVM, comparé à MVC, fait en sorte que les développeurs inexpérimentés ne les utilisent pas ;
2. la relation entre la taille des applications et la patron adopté, que nous expliquons dans la partie qui suit ;
3. l'absence du patron MVVM n'est pas une surprise, ceci est du au support tardif par la plateforme Android de ce patron et des composantes nécessaires telles que la librairie de la liaison de données, comme c'est expliqué dans la section 1.3.4. De plus, il faut noter aussi que l'utilisation du patron MVVM

est très réputé par les développeurs utilisant d'autres technologies telles que la plateforme Xamarin utilisant le langage C# ce qui n'est pas supporté dans notre étude (Johnson, 2018).

QR1- Observation 1 : MVC est le patron le plus utilisé par les développeurs Android.

Nous avons aussi analysé comment la fréquence d'utilisation des patrons MVC, MVP et MVVM a évolué à travers le temps. Pour ce faire, nous avons utilisé la date de la dernière mise à jour des applications. La figure 4.8 montre l'évolution du pourcentage de l'application des patrons discutés par année.

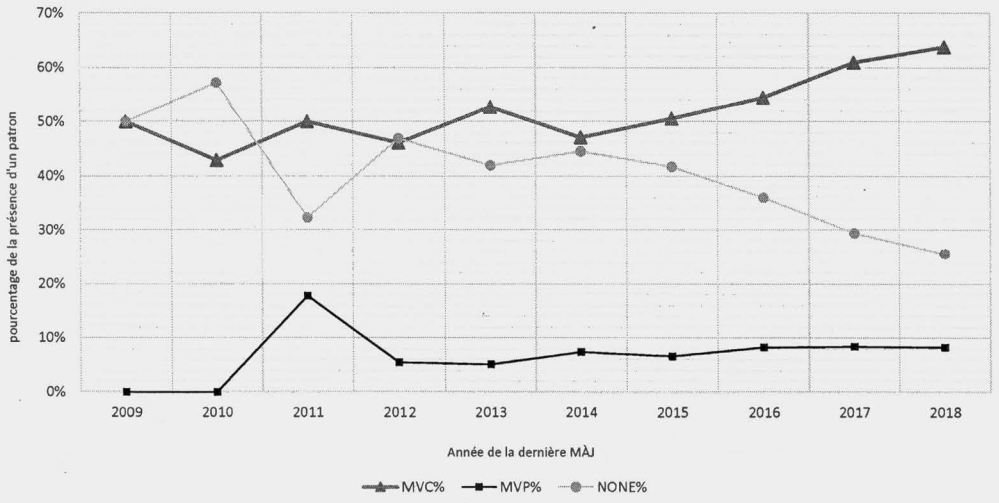


Figure 4.8: Évolution de l'adoption des patrons détectés par année

Ces résultats montrent que les développeurs Android ont commencé à appliquer le patron MVP autour de l'année 2010. Compte tenu de tous les récents débats autour de MVC et MVP, nous nous attendions à ce que les développeurs soient plus intéressés par MVP et l'utilisent plus dans les récentes années. Cependant, les résultats montrent que l'augmentation globale de l'utilisation de MVC au cours des quatre dernières années est plus importante que celle de MVP. En fait MVP

continue à être utilisé mais dans une petite partie des applications Android (8%).

Cela suggérerait que le débat dans la communauté des développeurs sur l'utilisation de patrons alternatifs à MVC, tels que MVP et MVVM, ne s'est pas traduit par une utilisation accrue de ces alternatives. De plus, le pourcentage d'applications qui n'utilisent aucun patron a considérablement diminué (d'environ 40% à 25%) pour les applications qui ont été mises à jour au cours des quatre dernières années. Cette baisse était en faveur d'une augmentation de l'utilisation de MVC.

QR1- Observation 2 : MVC a été le patron le plus utilisé et continue de gagner en popularité au cours des dernières années.

4.6.2 QR2 : Quels sont les types des applications Android qui utilisent les patrons basés sur MVC ?

Pour obtenir des informations sur les facteurs pouvant affecter le choix d'un patron donné, nous avons analysé les relations entre l'utilisation des patrons basés sur MVC et les types d'applications faisant partie de notre jeu de données.

La figure 4.9 montre la distribution par catégorie des occurrences de MVC et de MVP trouvées dans notre jeu de données.

En effet, plus de 50% des applications utilisent un patron pour chaque catégorie. À première vue, ces résultats n'indiquent aucune corrélation entre l'utilisation ou non d'un patron et la catégorie d'une application. Les catégories *Finance*, *Shopping* et *News & Magazines* ont les pourcentages les plus élevés d'applications utilisant un patron et utilisent généralement MVC. Ces catégories incluent généralement des applications professionnelles telles que *France24* (5 millions de téléchargements), *Daily Expenses 3* (1 million de téléchargements) et *OLX-Jual Beli Online* (10 millions de téléchargements). Ces trois applications ont toutes été implémentées

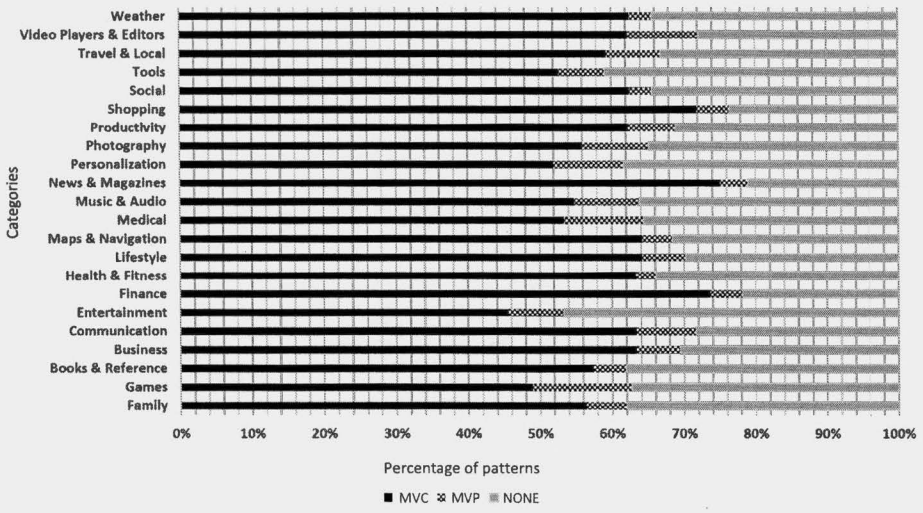


Figure 4.9: Distribution des patrons basés sur MVC par catégorie d'applications en utilisant MVC.

Les catégories *Entertainment* et *Tools* ont le plus grand nombre d'applications qui n'utilisent aucun patron. Nous avons investigué les applications de ces catégories pour comprendre la raison pour laquelle elles n'utilisent aucun patron et nous avons trouvé que 50% de ces applications ont une taille inférieure à 2MO. Quelques exemples des applications de ces catégories sont *DroidCam Wireless Webcam* (taille : 2,9MO) et *Real Horn Sounds* (taille : 1,3MO).

Le patron MVP de son côté a été utilisé dans la catégorie *Games* plus qu'aucune autre catégorie, ceci suggère que ces applications nécessitent un plus grand découplage des couches Vue, Contrôleur et Modèle.

QR2- Observation 1 : Il n'existe aucune relation apparente entre le choix d'un patron spécifique et la catégorie d'une application.

Cependant, notre analyse précédente montre qu'il y aurait une relation entre l'utilisation d'un patron et la taille d'une application. Les figures 4.10, 4.11 et 4.12

montrent la distribution des patrons basés sur MVC par la taille des applications. Nous pouvons clairement voir que 50% des applications ayant une taille inférieure à 1Mo, un nombre de classes inférieur à 50 classes et un nombre de lignes de code inférieur à 2000 n'utilisent aucun patron.

Ceci est dû au fait que la majorité des applications de petites tailles sont des applications utilitaires telles que celles appartenant aux catégories *Entertainment* et *Tools*. Généralement, ces applications n'implémentent pas des entités du modèle, et de ce fait n'utilisent aucun des patrons étudiés.

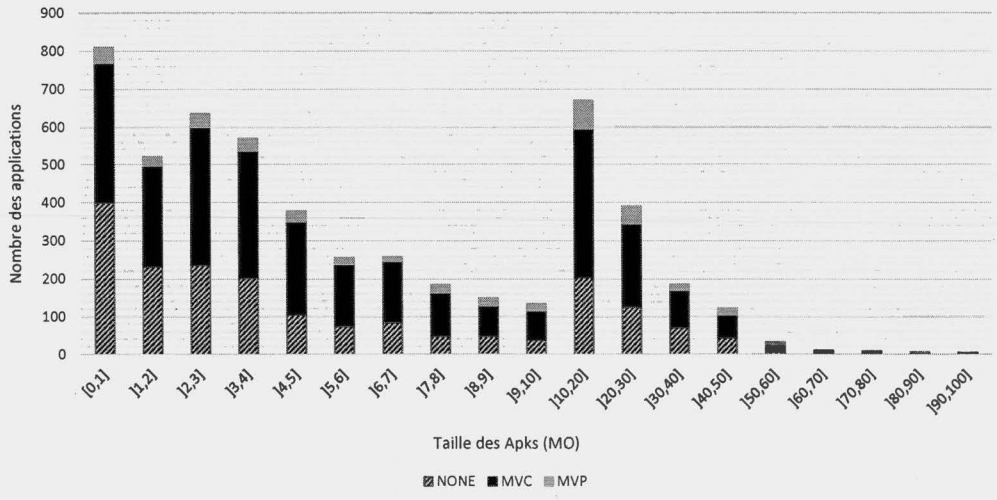


Figure 4.10: Distribution des patrons par taille du fichier Apk

De même, pour les plus grandes tailles, le nombre d'applications n'implémentant aucun patron diminue remarquablement comme le montre ces figures.

QR2- Observation 2 : Les applications de petites tailles sont celles qui implémentent le moins les patrons étudiés.

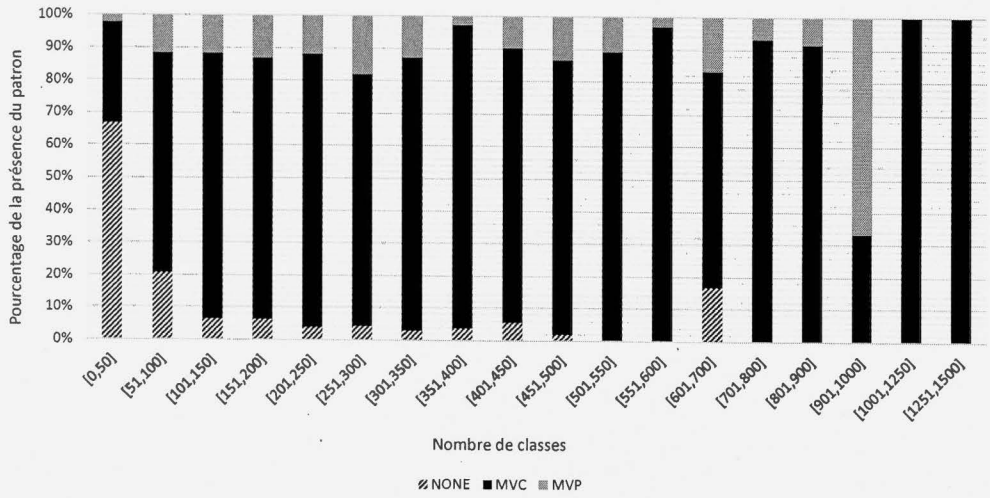


Figure 4.11: Distribution des patrons par par taille en nombre de classes

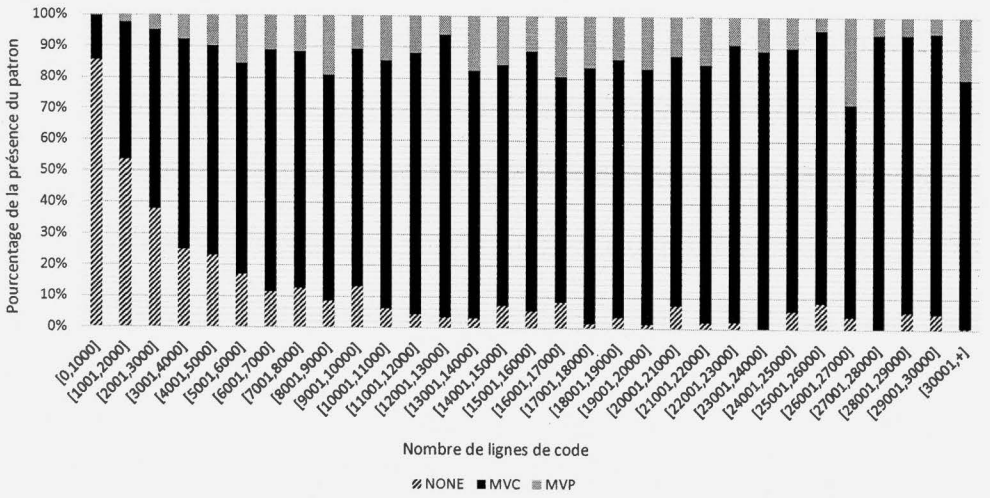


Figure 4.12: Distribution des patrons par nombre de lignes de code

4.7 Menaces à la validité

Dans cette section, nous mettons en avant les menaces à la validité concernant nos expérimentations en se basant sur le guide proposé par (Wohlin *et al.*, 2012).

Validité interne : La menace à la validité interne représente la relation entre les changements effectués aux variables manipulées (variables indépendantes) et

les résultats obtenus. Dans cette étude, ces menaces peuvent être causées par une détection incorrecte des patrons étudiés avec l'outil RIMAZ. Ceci peut être dû à l'analyse du code qui appartient à une librairie utilisée qui n'a pas été filtrée lors de la première étape du processus. C'est pour cette raison que nous avons utilisé, comme mentionné dans la section 3.3, une liste des librairies les plus utilisées fournie par (Li *et al.*, 2015). Nous avons également mis à jour cette liste par une liste supplémentaire construite manuellement en fonction des recommandations et des discussions de la communauté. Cette liste sera continuellement mise à jour pour améliorer la précision de RIMAZ. De plus, ces menaces peuvent aussi être causées par certaines manières d'implémentation de quelques patrons, et qui ne peuvent pas être détectés par notre approche. Par exemple, le patron MVVM peut être implémenté en utilisant des librairies tierces pour implémenter ce patron, ces dernières ne sont pas supportées dans notre approche. Nous avons adopté une heuristique générique et indépendante des librairies. Une autre cause pour ces menaces serait aussi les différentes pratiques, mauvaises ou controversées, qui peuvent être suivies par certains développeurs et qui font qu'on a des résultats non précis. Pour atténuer ces menaces, nous avons d'abord évalué manuellement notre approche sur un nombre représentatif d'applications avec une grande précision et l'avons appliqué sur un grand nombre d'applications. Finalement, nous avons essayé de faire attention en interprétant les résultats de notre analyse tout en considérant la tendance générale connue dans la communauté des développeurs Android.

Validité externe : La menace à la validité externe représente la possibilité de généraliser nos résultats. Les résultats obtenus durant nos expérimentations sont dépendants de l'état actuel de la plateforme de développement Android, des applications analysées, des patrons étudiés et de la liste noire des librairies à filtrer au début du processus de détection. Nous avons néanmoins sélectionné au hasard un

grand nombre d'applications ayant différentes tailles et appartenant à plusieurs catégories pour pouvoir appuyer nos résultats. Il est donc possible que ces résultats soient applicables sur un bon nombre d'applications.

Validité à la reproductibilité : La menace à la reproductibilité représente la possibilité de pouvoir répliquer nos expérimentations. Nous avons essayé de détailler le plus possible notre étude et de fournir toutes les informations nécessaires pour pouvoir reproduire notre approche et notre analyse. L'outil RIMAZ, la liste noire des librairies tierces et les applications analysées sont disponibles sur Github [2].

Validité des conclusions : La menace à la validité des conclusions appuie le fait que les conclusions déduites durant l'étude sont justes. Durant notre étude, nous n'avons pas tiré de conclusions qui ne peuvent être validées avec les résultats présentés.

4.8 Conclusion

Dans ce chapitre, nous avons présenté les expérimentations que nous avons effectuées avec notre outil RIMAZ. Nous avons aussi présenté des réponses à nos questions de recherche et nous avons essayé de donner des explications, en tenant compte des spécificités de la plateforme Android et les pratiques de la communauté.

Pour évaluer l'efficacité de notre outil, nous l'avons testé en analysant 100 applications à code source ouvert et en validant les résultats manuellement. L'outil RIMAZ a montré une précision de 85.87%.

En se basant sur ce degré de confiance, nous avons mené une large expérimentation

2. Source : `https://github.com/TheRimaz/Rimaz`

sur un jeu de données de plus de 5000 applications. Nous avons trouvé que MVC est le patron le plus utilisé par les développeurs Android et continue de gagner en popularité ces dernières années, alors que très peu d'applications utilisent le patron MVP et presqu'aucune application n'implémente le patron MVVM. De plus, nous avons remarqué que l'adoption d'un patron spécifique ne corrèle pas nécessairement avec la catégorie d'une application et que les applications de petites tailles sont celles qui implémentent le moins les patrons étudiés.

CONCLUSION

Le développement des applications mobiles Android est une option attrayante pour de plus en plus de développeurs. L'interactivité de l'utilisateur est un élément clé pour la plupart des applications. Ainsi, les choix des développeurs quant à la façon d'organiser les composants d'interaction de base fournis par la plateforme Android sont importants. Les patrons interactifs tels que MVC, MVP et MVVM sont donc particulièrement pertinents pour les applications Android. De nombreuses études ont été réalisées pour analyser le code source des applications Android afin de comprendre l'impact des bonnes ou des mauvaises pratiques sur la qualité des applications mobiles. Ces études ont donné naissance à différents outils d'analyses permettant aux développeurs et aux chercheurs de comprendre le comportement et l'évolution des applications ainsi que d'améliorer la qualité de leur code source. Néanmoins, certains types de ces pratiques, particulièrement les patrons architecturaux, n'ont pas eu un intérêt suffisant de la communauté en recherche.

Dans ce travail, nous proposons une approche basée sur des heuristiques pour identifier lequel des trois patrons architecturaux de la couche présentation (MVC, MVP et MVVM) est dominant dans une application Android. Nous avons validé notre approche en la testant sur un ensemble d'applications à code source ouvert. Ensuite nous avons conduit une expérimentation à grande échelle sur plus de 5000 applications que nous avons téléchargées depuis *Google Play Store*. Notre étude a montré que le patron MVC est le plus utilisé avec un pourcentage de 57% et qu'il continue à gagner en popularité ces dernières années. MVP est moins populaire (autour de 8% d'utilisation dans notre jeu de données). Quand à MVVM, bien qu'il

soit apparu dans l'échantillon des applications de validation à code source ouvert, il n'a pas été détecté dans notre jeu de données. Enfin, de façon significative, un bon pourcentage d'applications n'utilisait aucun patron. Notre étude a révélé aussi qu'il n'existe aucune relation entre l'utilisation d'un patron spécifique et la catégorie d'une application. De plus, Nous avons pu remarquer que la majorité des applications de petite taille tendent à être monolithiques en n'implémentant aucun des patrons étudiés.

D'une façon générale, notre étude fournit une vue globale sur le choix des développeurs Android en ce qui concerne les patrons architecturaux de la couche présentation. Néanmoins, comme expliqué dans la section 4.7, les résultats obtenus sont dépendants de plusieurs facteurs tels que l'état actuel de la plateforme de développement Android, les applications analysées, les librairies tierces utilisées dans ces applications et même les patrons étudiés. Nous ne pouvons pas donc généraliser nos résultats.

Actuellement, nous projetons plusieurs perspectives d'évolution dans cet axe de recherche. La première serait d'investiguer d'autres manières d'implémentation des patrons étudiés afin de renforcer notre outil de détection. De plus, nous prévoyons dans nos futurs travaux d'étudier à quel point ces patrons sont bien implémentés en détectant les violations les plus communes au sein de la communauté Android. Aussi, nous comptons intégrer notre outil RIMAZ dans les environnements de développement (tel que Android Studio) sous la forme d'un module d'extension afin de permettre aux développeurs d'analyser leurs applications. Finalement, nous espérons que cette recherche permet d'offrir une bonne compréhension sur l'utilisation des patrons basés sur MVC dans les applications Android et de servir comme un premier pas quand à l'utilité et l'impact de ce type de pratiques dans les applications mobiles en général.

RÉFÉRENCES

Alencar, P. S. C., Cowan, D. D. et de Lucena, C. J. P. (1996). A Formal Approach to Architectural Design Patterns. *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, 576 594.

Alikacem, H. et Sahraoui, H. (2006). Détection d'anomalies utilisant un langage de description de règle de qualité. *in actes du 12e colloque LMO.*

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

Android-Doc (2018a). Activities. `https://developer.android.com/reference/android/app/Activity`.

Android-Doc (2018b). Android view. `https://developer.android.com/reference/android/view/View.html`.

Android-Doc (2018c). App manifest. `https://developer.android.com/guide/topics/manifest/manifest-intro.html`.

Android-Doc (2018d). Art and dalvik. `https://source.android.com/devices/tech/dalvik/`.

Android-Doc (2018e). Dalvik executable format. `https://source.android.com/devices/tech/dalvik/dex-format`.

Android-Doc (2018f). Data binding library. `https://developer.android.com/topic/libraries/data-binding/index.html`.

Android-Doc (2018g). Dialogs. `https://developer.android.com/guide/topics/ui/dialogs.html`.

Android-Doc (2018h). Fragments. `https://developer.android.com/guide/components/fragments.html`.

Android-Doc (2018i). Guide to app architecture. https://developer.android.com/topic/libraries/architecture/guide.

Android-Doc (2018j). Input events. https://developer.android.com/guide/topics/ui/ui-events.html.

Android-Doc (2018k). Live data. https://developer.android.com/topic/libraries/architecture/livedata.html.

Android-Doc (2018l). Platform architecture. https://developer.android.com/guide/platform/.

Android-Doc (2018m). Support library. https://developer.android.com/topic/libraries/support-library/index.html.

Android-Doc (2018n). View model. https://developer.android.com/topic/libraries/architecture/viewmodel.html.

Antoniol, G., Fiutem, R. et Cristoforetti, L. (1998). Design pattern recovery in object-oriented software. Dans *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, 153 160. IEEE.

Bagheri, H., Garcia, J., Sadeghi, A., Malek, S. et Medvidovic, N. (2016). Software architectural principles in contemporary mobile software : from conception to practice. *Journal of Systems and Software*, *119*, 31 44.

Buschmann, F., Henney, K. et Schmidt, D. (2007). *Pattern-Oriented Software Architecture Volume 4 : A Pattern Language for Distributed Computing.*

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.

Deacon, J. (2005). Model-view-controller (mvc) architecture. *Computer Systems Development*, (Mvc), 1 6. Récupéré de https://techsimplified2.com/Uploads/Agendas/October28,2011.pdf

Dey, T. (2011). A Comparative Analysis on Modeling and Implementing with MVC Architecture. 44 49.

Fontana, F. A. et Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. *Information sciences*, *181*(7), 1306 1324.

Fowler, Martinn. Gui architectures. https://www.martinfowler.com/eaaDev/uiArchs.html.

Gamma, E., Helm, R., Johnson, R. et Vlissides, J. (1995). *Design Patterns :*

Elements of Reusable Object-oriented Software. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.

Garlan, D., Allen, R. et Ockerbloom, J. (1994). Exploiting style in architectural design environments. *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 175 188.

Hecht, G., Omar, B., Rouvoy, R., Moha, N. et Duchien, L. (2015). Tracking the software quality of android applications along their evolution. Dans *30th IEEE/ACM International Conference on Automated Software Engineering*, p. 12. IEEE.

John, Gossman (2005). Introduction to model-view-viewmodel pattern for building wpf apps. https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/.

Johnson, P. (2018). *Using MVVM Light with your Xamarin Apps*. Apress.

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.

Koenig, A. (1998). *Patterns and antipatterns*, volume 13.

La, H. J. et Kim, S. D. (2010). Balanced mvc architecture for developing service-based mobile applications. Dans *IEEE 7th International Conference on E-Business Engineering*, 292 299.

Li, L., Bissyandé, T. F., Klein, J. et Traon, Y. L. (2015). An Investigation into the Use of Common Libraries in Android Apps.

Lou, T. (2016). *A Comparison of Android Native App Architecture MVC, MVP and MVVM*. (Mémoire de maîtrise). Aalto University, Finland.

Lyngstad, G. B. S. (2013). Thesis : Design Patterns In A Smartphone Environment.

Martin, R. C. (1996). Design Principles and Design Patterns. (c), 1 34.

Maxwell Eric (2017). The mvc, mvp, and mvvm smackdown. https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/.

Microsoft-MSDN (2004). Implementing the model-view-viewmodel pattern. https://msdn.microsoft.com/en-us/library/ff798384.aspx.

Mudge, T. (2015). The architecture of smart phones. Dans *2015 IEEE 22nd*

International Conference on High Performance Computing (HiPC), 355 355.

Musselwhite, J. (Visité le 2018-03-01). Android architecture. Récupéré de http://www.therealjoshua.com/2011/11/android-architecture-part-1-intro/

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

Reenskaug, T. (1979). THING-MODEL-VIEW-EDITOR : an Example from a Planning System. *Xerox PARC Technical Note (May 1979)*.

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 2014*.

Shahbudin, F. E. et Chua, F.-F. (2013). Design patterns for developing high efficiency mobile application. *Journal of Information Technology & Software Engineering*, *3*(3), 667 684.

Sokolova, K. et Lemercier, M. (2014). Towards high quality mobile applications : Android passive mvc architecture. *7*, 123 138.

StackOverflow (2018). Mvc pattern on android. https://stackoverflow.com/questions/2925054/mvc-pattern-on-android.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A. et Poshyvanyk, D. (2015). When and why your code starts to smell bad. Dans *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, 403 414. IEEE.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. et Sundaresan, V. (1999). Soot - a java bytecode optimization framework. Dans *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, 13 . IBM Press.

Wasserman, A. I. (2010). Software engineering issues for mobile application development. Dans *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, 397 400., New York, NY, USA. ACM.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B. et Wessln, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.