UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ANALYSE DES APPELS DE SERVICES REST DANS LES APPLICATIONS ANDROID

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

ABDELKARIM BELKHIR

JANVIER 2019

REMERCIEMENTS

J'adresse mes remerciements à ma directrice de recherche, Naouel Moha, pour son temps et sa patience tout au long de ma maîtrise. Je remercie aussi les professeurs Éric Beaudry et Yann-Gaël Guéhéneuc pour leurs conseils et orientations. Je remercie tout particulièrement Manel et Midou pour leur soutien et ce depuis le début de cette aventure.

Je remercie aussi l'équipe IT de Schlumberger Algerie et plus particulièrement Amina, Hakim, Doudou et Nouredine pour tous les conseils, encouragement ainsi que les tout les moments inoubliables que nous avons passé ensemble.

Je remercie également tout ceux qui ont contribué de près ou de loin à la relecture de ce mémoire ; je cite Manel, Yasmine, Reda, Mehdi et Wassim. Je remercie Dylan et Aymen pour les nombreuses heures à la salle et Sara pour toutes les soirées couscous et lasagnes. Je remercie aussi toutes les personnes extraordinaires que j'ai rencontrés pendant ma maîtrise : Jehan, Ellen et Manel. Je remercie mes colocataires Mehdi et Reda pour tous les moments agréables qu'on a passé ensemble et tous les efforts pour rédiger ce *fastitieux* travail (et oui, on a réussi à placer ce mot dans le mémoire). Je tiens à remercier aussi tous ceux, malheureusement oublié ici, qui ont contribué à leur façon à ce document.

Pour finir, je dédie ce travail à mes parents, mes modèles et source de motivation pour leur soutien, et leurs sacrifices : *merci maman, merci papa* !

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

ACRONYMES

ADOCTOR	**An**Dr**O**id **C**ode smells detec**TOR**.
AE	**A**ttribut **E**xterne.
ANE	**A**nnotation **E**xterne.
API	**A**pplication **P**rogram **I**nterface.
AR	**A**ppel **R**EST.
AT	**A**sync**T**ask.
CAR	**C**ontient **A**ppel **R**EST.
CAS	**C**ommunication **AS**ynchrone.
CDE	**C**onfiguration de **D**élais d'**E**xpiration.
CFR	**C**hoix du **F**ormat de **R**éponse.
CI	**C**lasse **I**nvoquée.
CRAN	**C**lasse **R**éférençant le paquet **android.net**.
CRLC	**C**lasse **R**éférençant une **L**ibrairie **C**liente.
DOLAR	**D**etection **O**f **L**inguistic **A**ntiPatterns in **R**EST.
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol.
IAN	**I**nvocation de la méthode **getActiveSystemINfo**.
ICM	**I**nvocation de la classe **ConnectivityManager**.
ISS	**I**nvocation de la méthode **getSystemService**.
LC	**L**ibrairie **C**liente.
LCH	**L**ibrairie **C**liente *Helper*.
LCR	**L**iste des **C**lasses **R**éférençant la librairie.
LICs	**L**iste des **I**nvocations de **C**lasse.
LIMs	**L**iste des **I**nvocations de **M**éthode.

MCR	Mise en **C**ache des **R**éponses.
ME	Méthode **E**xterne.
MH	Modèle **H**ARISSA.
MI	Méthode **I**nvoquée.
NC	**N**om de la **C**lasse.
PL	**P**résence de la **L**ibrairie.
RD	**R**ègle de **D**étection.
RD-CAS	**R**ègle de **D**étection de la pratique **CAS**.
RD-CDE	**R**ègle de **D**étection de la pratique **CDE**.
RD-MCE	**R**ègle de **D**étection de la pratique **MCE**.
RD-VCA	**R**ègle de **D**étection de la pratique **VCA**.
REST	**RE**presentational **S**tate **T**ransfer.
SBD	**S**ignature de la méthode **doInBackground**.
SM	**S**ignature de **M**éthode.
SOA	**S**ervice-**O**riented **A**rchitecture.
SOAP	**S**imple **O**bject **A**ccess **P**rotocol.
SODA-R	**S**ervice **O**riented **D**etection for **A**ntipatterns in **R**EST.
SSL	**S**ecure **S**ockets **L**ayer.
TLS	**T**ransport **L**ayer **S**ecurity.
VCA	**V**érification de la **C**onnectivité de l'**A**ppareil.
VER	**V**érification de l'**É**tat du **R**éseau.
VTR	**V**érification du **T**ype du **R**éseau.

RÉSUMÉ

De nos jours les architectures orientées service jouent un rôle considérable dans l'industrie logicielle. Notamment l'architecture REST (*REpresentational State Transfer*), devenue une référence en matière de communications client/serveur. En effet, les APIs dites *RESTful* sont optimales pour fournir des données à différents types de clients (mobile, Web, etc). Cependant, les clients de type mobile sont ceux qui en profitent le plus, car ces derniers sont très souvent limités en ressources (batterie, mémoire, etc). Dans le cadre de notre travail de recherche, nous nous sommes intéressés sur la qualité des échanges entre clients mobiles et les services REST. Android étant la plateforme mobile la plus populaire, nous nous sommes principalement orientés vers celle-ci. En effet, plusieurs travaux de recherches ont étudié les bonnes et mauvaises pratiques spécifiques au développement d'une application mobile Android, d'autres ont traité les APIs REST mais à date, aucun ne traite de la qualité des échanges entre les deux. Dans le cadre de ce mémoire, nous nous intéressons spécifiquement à l'utilisation des bonnes pratiques pour le développement d'un client REST Android. Nous proposons, dans un premier temps, un catalogue de pratiques de développement spécifiques aux applications Android REST. Ensuite, nous proposons une approche outillée nommée HARISSA pour la détection automatique de ces pratiques. Notre approche nous permet, à date, de détecter quatre pratiques spécifiques aux applications Android REST. Cette approche nous a permis de conduire une étude empirique sur les tendances d'implémentations des pratiques sur 1595 applications. Nous avons observé plusieurs tendances sur l'implémentation des pratiques, en l'occurrence : presque 70% des développeurs adaptent le comportement de leur application selon la connectivité de l'appareil. D'un autre côté seules 10% des applications analysées implémentent la mise en cache des réponses. Ces chiffres nous donnent une idée globale des habitudes de développement des clients mobiles REST au sein de la communauté de développeurs Android. Ceci nous permettra dans des travaux futurs d'émettre des hypothèses concernant la popularité de certaines pratiques par rapport à d'autres.

MOTS CLÉS : Android, API REST, applications mobiles, génie logiciel, détection.

INTRODUCTION

Avec l'évolution considérable des plateformes et applications mobiles ces dernières années, l'utilisation d'appareils mobiles est devenue incontournable. Aujourd'hui, l'industrie du téléphone intelligent compte 4,57 milliards d'utilisateurs et s'approche des 5 milliards en 2020 (Statista, 2018).

Historiquement, les géants actuels des systèmes d'exploitation mobiles, iOS (Apple) et Android (Google), n'ont fait leur apparition qu'à partir de 2007. Avant, les plateformes mobiles dominantes étaient Symbian (Nokia) et BlackBerry OS (BlackBerry). Apple était le premier à introduire les téléphones intelligents nouvelle génération avec son produit : iPhone. Ce dernier possédait un écran tactile ainsi que plusieurs autres fonctionnalités avancées. Entre temps, Google avait racheté le produit Android en 2005 et a décidé de le promouvoir sur le marché en 2007. Android a connu une très forte popularité dans l'industrie téléphonique et est adopté aujourd'hui par la plupart des grands constructeurs mobiles (Samsung. LG, Sony, etc.).

Cette grande popularité au sein des utilisateurs ainsi que l'ouverture sur le développement d'applications au grand public ont encouragé beaucoup de développeurs à se convertir et adopter cette nouvelle plateforme désormais très fructueuse. Le nombre d'applications mobiles sur le marché évolue de façon considérable. En effet, à date, le magasin d'applications Android officiel *Google Play store* compte plus de 2,2 millions d'applications. Ces applications mobiles sont généralement écrites en utilisant des langages de programmation orientés objet comme Java, C++, Objective-C, Swift, Kotlin ou encore C#. Cependant, le développement de

telles applications est différent des applications classiques. En effet, les applications mobiles ont des contraintes supplémentaires liées à la plateforme ou aux ressources de l'appareil (Wasserman, 2010).

Dans le cadre de nos recherches, nous avons opté pour la plateforme Android pour son franc succès chez les développeurs et les utilisateurs (IDC, 2018). La plupart de ces applications accèdent à des données, parfois sensibles, souvent à travers des services distants pour des raisons d'architecture, d'efficacité et de sécurité.

Depuis leur apparition en 2000, les APIs REST sont devenues une référence lorsqu'il s'agit de fournir des services distants. Très pratique pour les applications mobiles, ce type d'architecture permet d'alléger considérablement les applications et assure une plus grande sécurité des informations.

Avec la demande croissante des utilisateurs en termes d'applications mobiles ainsi que la forte concurrence, les développeurs sont constamment obligés d'évoluer et maintenir leurs applications. Cette pression les conduit souvent à négliger les bonnes pratiques dans leur développement de façon intentionnelle ou simplement par ignorance.

Des études ont été menées pour identifier des bonnes et mauvaises pratiques dans le développement d'applications Android. Par exemple, Reimann *et al* proposent un catalogue contenant 30 défauts de code (Reimann *et al.*, 2014). D'autres études sont venues ensuite s'appuyer sur ce catalogue afin de proposer des approches automatisées pour la détection de ces défauts (Hecht *et al.*, 2015b; Palomba *et al.*, 2017).

Similairement, des catalogues ainsi que des approches de détection automatiques des bonnes et mauvaises pratiques ont été proposées concernant le développement des APIs REST (Palma *et al.*, 2014; Petrillo *et al.*, 2017).

Cependant, dans la littérature, les bonnes et mauvaises pratiques dans les applications Android ainsi que dans les APIs REST ont toujours été étudiées de façon séparée. À date, il n'y a aucune étude qui s'est intéressée à l'interaction des services avec les clients Android. Dans le cadre de notre recherche, nous visons essentiellement à identifier des bonnes pratiques spécifiques au développement d'un client Android. Nous pensons qu'il est possible d'optimiser ce dernier en adoptant certains comportements tels que la mise en cache des réponses du serveur, le bon choix du format de réponse ou encore la configuration de délais d'expiration des requêtes.

Nous nous sommes également intéressés aux librairies clientes HTTP utilisées par les applications Android pour communiquer avec les APIs REST. Celles-ci aussi peuvent adopter des bonnes pratiques pour offrir une interface efficace et facile à utiliser par les développeurs. Comme travail préliminaire, nous avons effectué une étude pour identifier des pratiques de développement spécifiques aux librairies clientes. Nous avons également mis en place un sondage auprès de développeurs Android pour valider la liste de pratiques. Ce travail a été publié dans une grande conférence spécialisée dans les systèmes orientés services ICSOC (*International Conference on Service-Oriented Computing*) (Oumaziz *et al.*, 2017). Cependant, ce travail ne sera pas décrit dans le présent document.

Les objectifs de recherche que nous avons définis sont les suivants :

- Construire un catalogue des bonnes pratiques liées aux clients REST Android.
- Proposer une approche automatique pour la détection de pratiques de développement spécifiques aux clients REST Android.
- Étudier les tendances d'implémentation des pratiques adoptées dans les applications Android.

Ainsi, dans ce mémoire, nous décrivons dans un premier temps le catalogue constitué de six bonnes pratiques pour les clients Android que nous avons construit. Nous proposons ensuite une approche outillée automatisée nommée HARISSA pour la détection de ces pratiques. Notre approche est basée sur l'analyse statique du code binaire (*bytecode*) de l'application Android. À date, HARISSA permet la détection de quatre des six pratiques de notre catalogue.

Les contributions de notre travail de recherche sont les suivantes :

- Un catalogue qui regroupe six bonnes pratiques de développement spécifiques aux clients REST Android. Ces pratiques sont déjà bien connues dans la communauté et sont décrites dans le chapitre 3 de ce mémoire.
- Une approche automatique nommée HARISSA permettant la détection de quatre pratiques de développement dans une application Android. La méthode de détection proposée se distingue par son niveau de détail puisque cette dernière est basée sur des la détection de plusieurs actions enclenchées par le développeur dans le code source de l'application (appels de méthodes, instanciations d'objets, assignations de valeurs, etc.)
- Une étude empirique sur plus de 1500 applications Android pour évaluer les tendances d'implémentation des pratiques spécifiques aux clients Android REST.

Ce mémoire est structuré de la manière suivante. Le chapitre I décrit les concepts préliminaires en relation avec notre thématique de recherche. Le chapitre II est dédié à l'état de l'art réalisé au courant de notre recherche. Par la suite, le chapitre III introduit et explique notre catalogue de pratiques de développement spécifiques aux applications Android REST. Ce chapitre est suivi par le chapitre IV qui décrit notre approche outillée pour la détection automatique des pratiques de développement dans une application Android. Dans le chapitre V, nous décrivons

notre étude empirique et présentons les résultats obtenus. Nous concluons ce mémoire avec une synthèse du travail effectué ainsi qu'une présentation des travaux futurs.

CHAPITRE I

CONCEPTS PRÉLIMINAIRES

Dans le but de faciliter la compréhension et la résolution de notre problématique de recherche, nous avons jugé indispensable de revenir sur les principes fondamentaux relatifs à la thématique étudiée afin de comprendre les notions préliminaires qui y sont associées.

Pour ce faire, nous entamons ce chapitre par une présentation de l'architecture orientée services dans son sens large avant de nous focaliser sur l'architecture REST. Nous abordons dans un second temps, la plateforme Android où nous passons en revue son architecture, les composants et le cycle de vie d'une application ainsi que les protocoles de communication avec un service REST.

1.1 Architecture REST

Il est possible de distinguer différents modèles d'architecture lorsqu'on parle d'architectures orientées service (SOA). Nous pouvons notamment citer les modèles SOAP (*Simple Object Access Protocol*) et REST (*REpresentational State Transfer*). Ce dernier n'a cessé de gagner en popularité depuis son apparition en 2000 dans un travail de thèse de Roy Fielding (Fielding et Taylor, 2000).

Une API (*Application Programming Interface*) dite *RESTful* propose des services à différents clients indépendamment des appareils sur lesquels ils reposent (appareil mobile, un ordinateur, etc.). Les communications avec une interface de programmation REST se font grâce au protocole HTTP (Fielding et Taylor, 2000).

1.1.1 Caractéristiques de l'architecture REST

L'architecture REST est définie selon six caractéristiques majeures qui offrent de nombreux avantages non fonctionnels désirables à l'architecture tels que la mise à l'échelle, la performance, la simplicité ou encore la portabilité. Ces caractéristiques sont décrites comme suit :

Client-Serveur. Cette caractéristique désigne un modèle d'architecture qui permet de séparer les responsabilités entre le client et le serveur donnant ainsi la possibilité aux deux composants d'évoluer indépendamment. Cette séparation offre donc une plus grande portabilité à la solution.

Sans état (*Stateless*). La conservation de l'état de la session entre deux requêtes successives se fait au niveau du client. Les requêtes de ce dernier contiennent l'état de la session ainsi que toutes les informations nécessaires pour y répondre. Ceci permet une meilleure visibilité sur les interactions entre les composants.

En couches. Dans une architecture REST, un client n'est pas en mesure de savoir s'il est connecté directement au serveur final ou à un serveur intermédiaire. L'utilisation de tels serveurs permet d'offrir une certaine extensibilité au système grâce à la mise en place d'une répartition des charges (*load balancing*) et un cache partagé. Elle peut également permettre un renforcement des politiques de sécurité.

Avec mise en cache. Les clients et les serveurs intermédiaires ont la possibilité de mettre en cache les réponses du serveur principal. Ce dernier peut spécifier cette possibilité de manière implicite ou explicite. Cette fonctionnalité a pour but de réduire le nombre d'interactions entre les composants, et, de ce fait, d'augmenter les performances.

Avec code à la demande (facultative). Il est possible aux serveurs d'étendre les fonctionnalités des clients en proposant du code exécutable (des applets Java, scripts JavaScript). Ceci permet non seulement d'alléger le code du client en réduisant le nombre de fonctionnalités à implémenter, mais offre également un moyen d'améliorer l'extensibilité du système.

Interface uniforme. Cette caractéristique est définie par une interface entre le client et le serveur. La communication entre ces deux composants se fait grâce au protocole HTTP et principalement à l'aide des différentes méthodes offertes par ce dernier. Le tableau 1.1 illustre les méthodes HTTP les plus utilisées. Ces méthodes permettent aux clients de viser des ressources spécifiques qui sont identifiées par des URIs (*Uniform Resource Identifier*).

Tableau 1.1: Liste des méthodes HTTP les plus utilisées.

Méthode	Description
`GET`	Demande une ressource au serveur à l'aide de son `URI`
`POST`	Poste une ressource spécifique au serveur
`PUT`	Crée ou modifier une ressource
`DELETE`	Supprime une ressource dans le serveur

1.2 Architecture de la plateforme Android

Android est composé de cinq couches principales (Android-Doc, 2018b). La figure 1.1 illustre ces différentes couches telles que présentées par le site officiel de Android. Ces dernières sont décrites comme suit :

1. **Le noyau Linux.** Cette couche représente le moteur principal sur lequel est basé tout le système.
2. **La couche d'abstraction matérielle.** Cette couche permet l'exposition des fonctionnalités matérielles aux couches supérieures de l'architecture.
3. **Librairies natives et environnement d'exécution (*Android Runtime*).** On retrouve dans cette couche les librairies natives, elles permettent à l'appareil d'effectuer des opérations de base telles que le support de différents types de données, l'affichage des éléments graphiques ou même la gestion des bases de données.

   On y trouve également l'environnement d'exécution (*Android Runtime*) qui tout comme son prédécesseur (*Dalvik*) permet l'exécution du code intermédiaire (*Bytecode*) résultant de la compilation du code de l'application.

4. **Le cadre d'application Android (*Android Framework*).** On y trouve les fonctionnalités et blocs de constructions dans le développement d'une application Android (`ActivityManager`, `ContentProvider`, `TelephonyManager`, etc.).
5. **Applications système.** Cette couche contient les différentes applications natives de la plateforme (navigateur, calendrier, contacts, etc.).

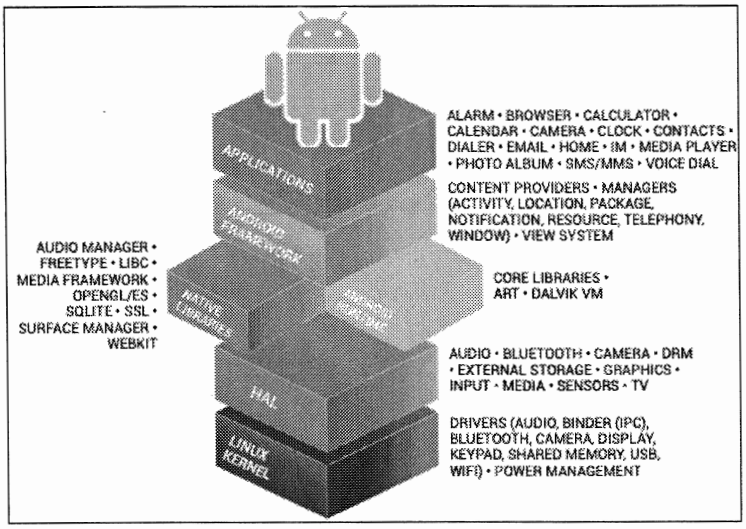

Figure 1.1: Architecture de la plateforme Android (Android-Doc, 2018b).

1.2.1 Applications Android

En plus de l'architecture de la plateforme, il est important de présenter certains concepts clés qui concernent le développement d'une application Android.

Une application Android est principalement développée en Java, celle-ci est distribuée par la suite dans un format de paquet compressé nommé APK (*Android Package Kit*). On y retrouve le code intermédiaire de l'application et celui de toutes les librairies utilisées (natives et tierces), les ressources ainsi que le fichier *Manifest*.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     android:versionCode="1"
5     android:versionName="1.0"
6     package="com.example.myapp">
7
8     <!-- Beware that these values are overridden by the build.gradle file -->
9     <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 </manifest>
```

Figure 1.2: Exemple d'un fichier *Manifest* (Android-Doc, 2018c).

Composants d'une application Android

Fichier *Manifest*. AndroidManifest est un fichier de format XML qui contient des informations décrivant la structure globale de l'application. La figure 1.2 illustre un exemple de fichier *Manifest*. Ce fichier comporte les informations suivantes :

1. Le nom du paquet principal de l'application ;
2. La liste des permissions nécessaires au fonctionnement de l'application ;
3. La déclaration des librairies tierces utilisées par l'application ;
4. La liste des composants principaux de l'application (activités, services, récepteurs de diffusion, fournisseurs de contenu).

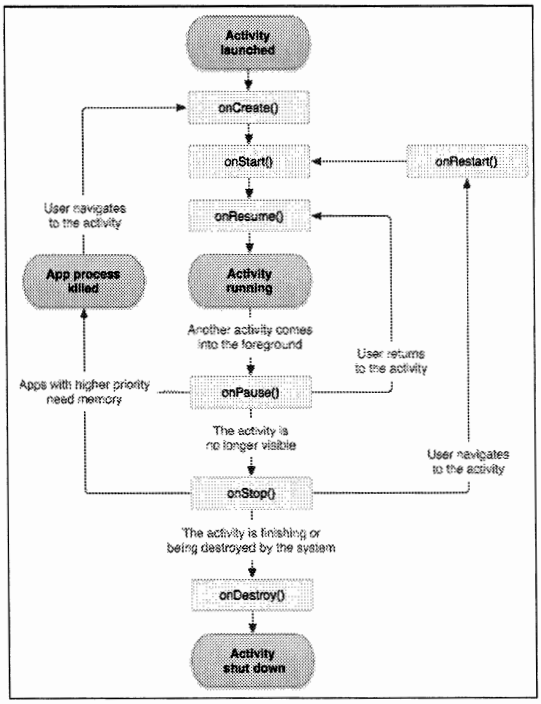

Figure 1.3: Cycle de vie d'une activité dans une application Android (Android-Doc, 2018d).

Ressources. Chaque application Android comporte un ensemble de ressources nécessaires à son bon fonctionnement. Il s'agit généralement d'images, de textes traduits, de couleurs, de styles, des différentes dispositions (*layouts*), etc. Android génère une classe nommée `R` dans laquelle chaque ressource est affiliée à un identifiant unique. Ceci permet le référencement de ces ressources à partir du code source de l'application.

Cycle de vie d'une Activité Android

Au cours de l'utilisation d'une application Android, une activité peut passer par différents états soit en fonction de l'environnement (système Android) ou bien des actions de l'utilisateur. La figure 1.3 illustre les différents états que peut avoir une activité au cours de l'exécution de l'application.

- `onCreate()`. Cette méthode est appelée une seule fois, au moment de la création de l'activité.
- `onStart()`. Cette méthode est appelée au moment où l'état de l'activité passe à l'état *Started*. Celle-ci est dorénavant visible à l'utilisateur et se prépare à recevoir toute interaction de ce dernier.
- `onResume()`. Cette méthode est appelée au moment où l'état de l'activité passe à l'état *Resumed*. Celle-ci occupe le premier plan et est interactive avec l'utilisateur.
- `onPause()`. Cette méthode est appelée au moment où l'état de l'activité passe à l'état *Paused*. Celle-ci n'occupe plus le premier plan.
- `onStop()`. Cette méthode est appelée au moment où l'état de l'activité passe à l'état *Stopped*. Celle-ci n'est plus utilisée et donc plus affichée à l'utilisateur.
- `onDestroy()`. Cette méthode est appelée au moment où l'activité s'apprête à être détruite par le système.

1.2.2 Protocole de communication avec un service REST

L'interaction entre une application Android et une API REST ressemble à une interaction classique entre un client et un serveur distant. En effet, l'échange entre ces deux acteurs se résume dans la préparation et l'envoi de la requête par le client, puis la réception et le traitement de la réponse du serveur. Les aspects à considérer lors de l'implémentation d'un client REST sur une plateforme Android sont : (A) la communication avec le service, (B) la conception de l'application et (C) les optimisations possibles. Ces aspects sont décrits plus en détail dans les sections suivantes.

A. La communication avec le service

Le client Android prépare une requête HTTP avec (1) l'URL de base de l'API, (2) la méthode HTTP, ainsi que (3) les paramètres de la requête. Cette requête est transmise à l'API REST à l'aide d'une librairie cliente HTTP.

Il est possible de distinguer trois types de librairies clientes :

1. **Librairies clientes de bas niveau.** Le plus souvent, ce type de librairies est fourni avec le cadre d'application de la plateforme Android (ex. la librairie `HttpURLConnection`).
2. **Librairies clientes tierces.** Au cours des dernières années, plusieurs librairies tierces ont fait surface (ex. `Retrofit`, `Google Volley`). Ces librairies sont souvent des façades (*wrappers*) de librairies de bas niveau, offrant ainsi une interface logicielle plus simple à utiliser et maintenir.
3. **Librairies clientes *Helper*(*Helper Client Library*).** Ce type de librairies vise une API ou un ensemble d'APIs bien défini. Ce sont des librairies d'un niveau plus haut que les librairies tierces et offrent beaucoup plus de facilité dans la consommation de l'API. Un exemple est de retourner les réponses sous la forme d'objets Java spécifiques à l'API au lieu d'une réponse dans un format JSON ou XML (ex. `Tweet` pour l'API de *Tweeter* ou bien `Post` pour *Facebook*).

B. La conception de l'application

Il existe différents choix de conception lorsqu'il s'agit de consommer une API REST. En effet, le développeur peut choisir entre différents styles architecturaux, certains exemples ont été proposés lors de la conférence annuelle *Google IO 2010*.

C. Les optimisations possibles

Il existe un certain nombre d'optimisations possibles lors de la conception d'une application Android cliente d'une API REST (mise en cache, choix du type de retour, compression, etc.).

1.3 Contraintes liées le développement d'une application cliente mobile

Bien que l'implémentation d'un client REST dans une application mobile Android ne soit pas très différente d'une application Java ordinaire, il existe un certain nombre d'éléments à prendre en considération lors du développement de cette dernière :

1. **Les ressources limitées.** Il est très important de prendre en considération les limitations en matière de ressources lorsque l'on développe une application pour un appareil mobile.
2. **La consommation énergétique.** La consommation énergétique est l'aspect le plus important dans l'expérience utilisateur. De plus, les opérations utilisant le réseau mobile (Wi-Fi, 3G, Edge, etc.) sont les plus lourdes en termes de consommation de batterie (Li *et al.*, 2014; Sharkey, 2009). C'est pour cette raison qu'il est très important d'adapter le comportement de l'application afin de conserver le niveau de la batterie.
3. **L'utilisation des données.** Un autre point important concernant l'expérience utilisateur est la consommation des données. En effet, il est possible de réduire la consommation des données en réduisant le taux de données téléchargées, ceci permet de réduire les coûts financiers associés à leurs téléchargement. Plusieurs techniques permettent d'accomplir ceci, en particulier la compression des données ou la réduction de la fréquence de requêtes lorsque l'appareil n'est pas en Wi-Fi.

4. **Contraintes en relation avec la plateforme.** Certaines contraintes sont directement liées à la plateforme Android. Nous pourrons citer à titre d'exemple le fait de ne pas exécuter des communications réseau ou toute autre opération lourde dans le contexte du fil d'exécution principal (*UI Thread*). La non-prise en considération de ces contraintes pourrait entraîner le dysfonctionnement voir l'interruption de l'application dans certaines versions de la plateforme Android.

CHAPITRE II

ÉTAT DE L'ART

L'étude des applications Android ainsi que les pratiques suivies par les développeurs sont des sujets d'actualité et qui, par conséquent, suscitent un grand intérêt au sein de la communauté des chercheurs dans le domaine. En effet, durant les dernières années, un grand nombre de travaux qui traitent des différents axes de notre travail a vu le jour, venant ainsi alimenter en ressources cette vaste thématique de recherche.

Les travaux dont il est question peuvent être répartis en deux catégories distinctes. D'un côté, il y a les travaux qui traitent des différentes pratiques de développement, que ce soit en dans le domaine des applications Android ou de l'architecture REST. De l'autre côté, nous trouvons les travaux traitant de l'analyse automatique desdites pratiques.

Dans ce chapitre, il est question de discuter les différents travaux existants à ce jour. Nous commencerons par les différentes bonnes et mauvaises pratiques dans le développement d'une application Android. Nous abordons ensuite les mêmes aspects concernant les architectures REST. Une fois ces deux axes discutés, nous présentons les différentes techniques existantes permettant l'analyse d'applications Android.

2.1 Pratiques liées à la plateforme Android

Au même titre que sur toute plateforme logicielle, il existe des bonnes et des mauvaises pratiques lorsqu'il s'agit de développer une application Android. Ces pratiques sont parfois appelées patrons et anti-patrons lorsqu'elles deviennent récurrentes au sein de la communauté des développeurs (Gamma, 1995). Néanmoins, chaque plateforme peut être sujette à un type bien spécifique de pratiques liées aux contraintes qui lui sont associées.

Bien qu'ils soient récents pour la majorité, il existe peu de travaux scientifiques qui proposent une étude sur les bonnes et mauvaises pratiques dans le développement des applications Android. En effet, cette thématique est beaucoup plus discutée par les développeurs dans des espaces de partage de connaissances prévus à cet effet comme les forums, des groupes spécifiques dans les réseaux sociaux ou présentée dans la documentation officielle de la plateforme (Android-Doc, 2018a).

Une équipe de recherche de l'université de Dresden propose un catalogue comportant 30 défauts de qualité liés aux applications Android (Reimann *et al.*, 2014). Un exemple des défauts de qualité est l'interruption de l'interface utilisateur par des processus fonctionnant en arrière-plan de l'application. Plusieurs travaux par la suite ont repris le catalogue introduit par cette étude pour proposer des approches de détection automatique pour les défauts.

Yoonsik Cheon, un professeur à l'université du Texas, présente une étude (Yoonsik, 2016) qui démontre que les bonnes pratiques en développement Java ne le sont pas forcément sur la plateforme Android. Effectivement, même si Java est un langage de développement officiel sur Android, certaines des bonnes pratiques en Java peuvent même être identifiées comme anti-patrons sur la plateforme Android. Dans son travail, l'auteur propose une étude de cas qui démontre qu'en effet,

certaines bonnes pratiques causent une utilisation excessive des ressources sur un appareil mobile. Nous rappelons bien-sûr que ce type d'appareil est souvent très limité par les ressources matérielles ainsi que par la batterie.

Nous avons constaté qu'il existait un certain nombre d'optimisations possibles quant à la consommation énergétique lors du développement d'une application Android. Un groupe de chercheurs de l'université de South California a proposé un ensemble de travaux à ce sujet. Dans leur étude (Li *et al.*, 2014), les auteurs proposent une étude empirique sur 405 applications dans laquelle ils essayent de répondre à plusieurs questions de recherche en relation avec la consommation énergétique des applications Android. Le point le plus important à soulever dans cette étude est que l'utilisation du réseau est l'activité la plus gourmande en énergie, particulièrement les requêtes HTTP. L'étude stipule que 75% des applications utilisent plus de 89% de l'énergie associée à l'utilisation du réseau dans les communications HTTP. Les auteurs présentent également des travaux en relation avec certaines pratiques de développement concernant la consommation énergétique sur les appareils mobiles tournant sous Android (Li et Halfond, 2014; Li et Halfond, 2015). Parmi les pratiques étudiées, nous retrouvons l'optimisation du processus de communication utilisant le protocole HTTP. En l'occurrence, les expérimentations conduites par les auteurs ont permis de démontrer que la transmission d'un gros paquet était beaucoup plus économe en énergie que la transmission de multiples paquets de taille inférieure (une différence de 50% a été observée). Les auteurs proposent donc de regrouper plusieurs requêtes de petites tailles en une seule requête lorsque ceci est possible.

Lors de la conférence annuelle de *Google IO 2010*, Virgil Dobjanschi a proposé trois patrons de conception pour les clients REST Android (Dobjanschi, 2010). Il a également expliqué qu'il était possible de proposer son propre patron de conception à condition de prendre en considération certains points essentiels :

— **Le bon choix du format de retour.** Il est important de bien choisir le format de la réponse envoyée par le serveur lorsque le choix est possible. Le conférencier propose de procéder dans l'ordre suivant : binaire, JSON, XML. Plusieurs autres travaux ont démontré la véracité de cet ordre (Gil et Trezentos, 2011; Rodrigues *et al.*, 2011; Sumaray et Makki, 2012).

— **Activer la compression.** Dans certains cas, il est possible d'échanger des données compressées avec l'API REST dans le cas où celle-ci le permet. Ceci peut réduire considérablement la consommation de la bande passante (Palomba *et al.*, 2017; Reimann *et al.*, 2014).

— **Lancer l'appel REST dans un fil d'exécution séparé.** Il est fortement déconseillé de lancer des appels REST à partir d'une activité Android ou bien dans le contexte du fil d'exécution principal, ceci peut causer le dysfonctionnement voire l'interruption de l'application (à partir de Android 3.0) (Android-Doc, 2018f).

— **Minimiser les communications.** Comme expliqué précédemment, il est préférable d'adopter un comportement qui permet de minimiser au maximum le nombre d'interactions avec le serveur distant. Plusieurs solutions peuvent être considérées dans ce cas : la mise en cache, la réduction de la fréquence de rafraîchissement, etc.

Dans une autre étude intéressante (Naylor *et al.*, 2014), les auteurs ont essayé de démontrer que sécuriser les communications avec la fonctionnalité HTTPS pourrait avoir un coût sur la performance, la consommation énergétique et même sur l'utilisation de la bande passante. En effet, leurs expérimentations montrent que lors d'une utilisation excessive de ce protocole, la bande passante est significativement plus sollicitée. Ceci est dû principalement au mécanisme d'échange de certificats (TLS/SSL) et aussi à l'impossibilité d'optimiser les communications grâce à l'utilisation de serveurs mandataires (*Proxy Servers*) pour la mise en cache

ou encore la compression de données. L'utilisation de l'HTTPS a également un impact négatif non-direct sur la batterie, ce protocole implique le cryptage/décryptage de données et garde la radio éveillée pendant une plus longue période. Les auteurs suggèrent donc de réduire l'utilisation de ce mécanisme de sécurité seulement aux opérations sensibles.

2.2 Analyse des pratiques dans les architectures REST

Plusieurs travaux portant sur les patrons et anti-patrons dans une architecture REST ont été proposés. Cependant, ces travaux ciblent pour la plupart la partie serveur de l'interaction.

Dans leurs travaux, Palma *et al.* présentent dans un premier temps une approche basée sur les heuristiques nommée SODA-R (*Service Oriented Detection for Antipatterns in REST*) pour la détection d'anti-patrons dans l'architecture REST (Palma *et al.*, 2014). Cette approche leur a permis de détecter huit anti-patrons et cinq patrons dans 12 APIs REST populaires. Parmi ces patrons nous pouvons citer comme exemples le fait de fournir différents formats de réponses et permettre leur mise en cache locale.

D'autres travaux ont également été proposés pour la détection des (anti-)patrons linguistiques liés à l'architecture REST. Ces travaux sont basés sur des techniques d'analyses sémantiques et lexicales à l'aide de dictionnaires lexicaux tels que WordNet [1] ou encore Stanford CoreNLP [2] afin de détecter les anomalies linguistiques dans des APIs REST. Parmi ces travaux, nous citons les articles suivants :

Palma *et al.* ont proposé une nouvelle approche qui permet la détection de cinq

---

1. `https://wordnet.princeton.edu` (Miller *et al.*, 1990)

2. `https://stanfordnlp.github.io/CoreNLP/` (Manning *et al.*, 2014).

anti-patrons linguistiques dans une architecture REST (Palma *et al.*, 2015; Palma *et al.*, 2017). À l'instar de leur premier travail (Palma *et al.*, 2014), l'approche DOLAR (*Detection Of Linguistic Antipatterns in REST*) est basée sur le framework SOFA (*Service Oriented Framework for Antipatterns*) (Moha *et al.*, 2012) qui permet de générer des algorithmes de détection d'anti-patrons dans les architectures orientées services. Les auteurs ont testé leur approche sur 15 APIs REST populaires et ont montré une précision moyenne de 81% avec un rappel de 78% pour tous les (anti-)patrons.

Petrillo *et al.* ont proposé quant à eux une autre approche CLOUDLEX (Petrillo *et al.*, 2017) permettant également la détection de quatre (anti-)patrons linguistiques avec une précision moyenne de 84.82% et un rappel de 63.57%.

2.3 Approches outillées pour la détection des pratiques

Depuis l'apparition des (anti-)patrons dans le développement logiciel, les chercheurs ont rapidement été intéressés par la possibilité de détecter automatiquement la présence de ces derniers.

Une équipe de recherche propose une approche outillée nommée PAPRIKA pour la détection de défauts de code dans les applications Android (Hecht *et al.*, 2015b). L'outil proposé permet la détection de sept défauts de code grâce à une analyse statique du code intermédiaire. La liste des défauts étudiés est composée de défauts de code populaire en programmation orientée objet ainsi que d'autres qui sont spécifiques à la plateforme Android. Cet outil leur a permis, par la suite, de conduire une étude empirique sur l'évolution de la présence des défauts de code dans les applications Android à travers les versions (Hecht *et al.*, 2015a).

Un outil similaire nommé ADOCTOR (*AnDrOid Code smell detecTOR*) est proposé par une autre équipe (Palomba *et al.*, 2017). Cet outil permet la détection d'un plus grand nombre de défauts de code spécifiques à la plateforme Android que PAPRIKA puisqu'il est capable de détecter 15 défauts de code (contre quatre pour PAPRIKA). ADOCTOR se base sur le catalogue proposé dans l'étude (Reimann *et al.*, 2014). L'outil a montré une précision de 98% avec un rappel de 98% lors d'une étude empirique sur un ensemble de 18 applications.

D'autres approches pour la détection de défauts de code spécifiques à la plateforme Android ont également été proposées. Une troisième équipe de recherche propose une approche basée sur un algorithme de programmation génétique (Kessentini et Ouni, 2017). Les auteurs ont validé leur approche avec une détection de dix défauts de code sur 181 applications Android à code source ouvert.

Dans ce chapitre, nous avons discuté plusieurs travaux qui traitent des différents axes de notre travail de recherche. Nous avons présenté des travaux qui portent sur l'analyse de la qualité des interfaces d'applications orientés services (API REST) du côté serveur, ainsi que d'autres travaux qui portent sur la détection de pratiques de développement dans les applications Android.

Cependant, à ce jour, il n'existe aucun travail sur l'analyse de la qualité des échanges entre clients mobiles et les services REST. Notre travail de recherche a pour but de combler ce manque et de présenter ainsi une étude qui ciblerait principalement les clients mobiles Android.

Dans notre travail de recherche, nous avons identifié dans un premier temps, à partir de la littérature, une liste de bonnes et mauvaises pratiques dans le développement d'un client Android. Nous nous sommes ensuite intéressés à la détection automatique de ces dernières à l'aide d'une approche outillée.

CHAPITRE III

LES PRATIQUES DE DÉVELOPPEMENT ADOPTÉES DANS LES CLIENTS REST ANDROID

Lors du développement d'un client REST, le développeur peut choisir d'utiliser un certain nombre de bonnes pratiques qui s'offrent à lui. En effet, il existe des pratiques de développement spécifiques aux clients REST qui peuvent être considérées comme bonnes pratiques compte tenu des nombreux avantages qu'elles offrent en termes de performance, de consommation énergétique et même de protection de la bande passante. Cependant, la non-utilisation desdites pratiques ne peut être considérée systématiquement comme une mauvaise pratique à moins que le développeur soit capable de mesurer les impacts liés à leur non-utilisation.

Dans ce chapitre, nous présentons un nombre de pratiques de développement liées aux clients REST Android. Nous abordons la vérification de la connectivité de l'appareil, la mise en cache des réponses, le choix des techniques de communication (synchrone ou asynchrone), la spécification des délais d'expiration, le choix des formats de réponse ainsi que la prise en compte des codes de réponse du serveur. Chacune de ces pratiques est ensuite détaillée et nous listons les avantages qu'elles offrent ainsi que les impacts directs ou indirects sur l'expérience utilisateur.

3.1 Description des pratiques de développement

Cette section a pour but de décrire chacune des pratiques étudiées en mettant l'accent sur leurs avantages. Il sera également question de discuter les impacts directs et indirects de ces pratiques sur l'expérience utilisateur.

3.1.1 Vérification de la Connectivité de l'Appareil (VCA)

Cette pratique consiste à vérifier l'état ou le type de la connexion utilisée par l'appareil avant d'effectuer les appels REST. En effet, il est recommandé d'offrir la possibilité à l'utilisateur de contrôler le comportement de l'application quant à l'utilisation des données, surtout si celle-ci fonctionne en arrière-plan et est gourmande en bande passante, par exemple : chargement et téléchargement de données, synchronisation, mises à jour, etc. Ceci permet d'éviter de mauvaises surprises à l'utilisateur qui pourrait se voir perdre involontairement et sans connaissances des faits plusieurs méga-octets de données.

Afin de remédier à ce problème, le cadre applicatif Android fournit une classe `ConnectivityManager` qui permet aux développeurs d'avoir un suivi en temps réel de la connectivité de l'appareil (Android-Doc, 2018a). L'utilisation de cette classe offre ainsi la possibilité d'optimiser le comportement de l'application selon l'état ou le type de la connexion utilisée. Concrètement, il est possible de réduire la fréquence de sollicitation de la bande passante lorsque l'appareil n'est pas connecté en Wi-Fi ce qui permet de réduire la consommation de la batterie ainsi que les coûts encourus par l'utilisateur suite à une utilisation excessive du réseau mobile.

3.1.2 Mise en Cache des Réponses (MCR)

Une étude (Qian *et al.*, 2012) menée par une équipe de recherche a démontré que certaines applications populaires ont un taux très élevé de redondance dans les échanges avec les services REST. Ce taux varie entre 18% et 20% des échanges.

Une des approches permettant de pallier ce problème est la mise en cache des réponses faisant référence au stockage local des données fréquemment requêtées.

Les trois formes de mise en cache au niveau du client les plus populaires sont (Ollite et Mohamudally, 2015) :

1. *Implémentation à travers le cadre applicatif de la plateforme.* Le cadre d'application Android fournit des techniques pour mettre en cache les réponses du serveur (Android-Doc, 2018e) ;
2. *Implémentation à travers des librairies tierces.* Plusieurs librairies officielles ou tierces proposent la possibilité d'implémenter la mise en cache, notamment les librairies clientes HTTP ;
3. *Implémentation personnalisée de la mise en cache.* Il se trouve que la possibilité de mettre en cache les réponses n'est pas présente dans toutes les librairies (Qian *et al.*, 2012). Dans d'autres cas, le développeur préfère implémenter sa propre façon de mettre en cache les réponses du serveur.

Cette pratique a pour impact de réduire les interactions avec le service distant ce qui se traduit par une réduction de la consommation de la batterie et les données du réseau ainsi qu'une nette amélioration des performances.

3.1.3 Configuration des Délais d'Expiration (CDE)

Lors d'un échange avec une API REST, il arrive que celle-ci soit surchargée par le nombre de requêtes causant un temps de latence de plusieurs secondes. Un article populaire (Nielsen, 2010) stipule que l'utilisateur perd patience à partir de dix secondes d'attente.

Pour pallier ce problème, le développeur peut configurer un délai d'expiration. Ce dernier fait référence au temps d'attente maximal pour une réponse ou une interaction de la part de l'entité visée. L'utilisation de tels délais est commune dans plusieurs cas. On peut citer à titre d'exemple les communications avec une base de données ou un fil d'exécution séparé.

Le développeur peut spécifier un comportement en particulier en cas de dépassement de ce délai afin d'améliorer la fluidité de l'application. Il est possible par exemple d'afficher un message d'erreur pour informer l'utilisateur que l'application fonctionne toujours (Mionskowski, 2017).

Cette pratique peut être implémentée à travers une librairie cliente. En effet, cette dernière propose généralement un moyen de configurer un délai d'expiration pour les requêtes HTTP. Il est également possible de proposer une implémentation personnalisée de cette pratique à l'aide de méthodes asynchrones.

Cette pratique est rarement prise en considération par les développeurs, alors qu'elle permet grandement de faciliter la maintenance et la gestion d'erreur de l'application.

3.1.4 Communications Asynchrones (CAS)

Sur la plateforme Android, lorsqu'une application est lancée, un fil d'exécution lui est alloué. Ce dernier est connu sous le nom du fil d'exécution principal ou fil d'exécution de l'interface (*UI Thread*). Toutes les opérations exécutées dans le contexte de ce fil d'exécution le sont de façon synchrone (Android-Doc, 2018c).

Ce type d'exécution n'est cependant pas optimal pour les opérations longues (plus de cinq secondes) puisque ceci bloque l'interface utilisateur en attente de la fin de l'opération. C'est pour cette raison qu'il est hautement recommandé d'exécuter les opérations longues de façon asynchrone y compris les appels de services REST (Dobjanschi, 2010; Android-Doc, 2018f).

L'exécution synchrone des appels REST peut causer le dysfonctionnement voir l'interruption de l'application dans certaines versions de la plateforme Android, ceci peut se traduire par l'affichage d'un message ANR (*Application Not Responding*) ou par la levée de l'exception `NetworkOnMainThreadException`.

3.1.5 Choix du format de réponse (CFR)

Une API REST peut retourner la réponse à une requête HTTP sous différents formats. Les plus populaires restent JSON et XML. Le bon choix du format de réponse peut avoir un impact sur les performances de l'application et notamment sur les applications mobiles où les ressources sont limitées.

Le listing 3.1 illustre la différence dans la présentation d'informations identiques encodées respectivement dans les formats XML et JSON.

XML

```xml
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
</employees>
```

JSON

```json
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" }
]}
```

Listing 3.1: Comparaison des formats XML et JSON (W3schools, 2018).

Les deux principales caractéristiques que nous pouvons noter à partir de cette comparaison sont :

1. **La lisibilité.** Le format JSON est significativement plus lisible par un humain que XML.
2. **La taille.** Le format JSON est beaucoup plus compact que son concurrent, et donc beaucoup moins volumineux.

Ces différences peuvent avoir un impact sur les performances d'une application mobile. Quelques travaux ont été présentés concernant la comparaison entre ces deux formats. À titre d'exemple, l'étude (Sumaray et Makki, 2012) dans laquelle les auteurs comparent dans une application Android les temps de sérialisation et désérialisation de la représentation d'une entité `Book` dans les deux formats (JSON et XML). La figure 3.1 illustre les résultats de la comparaison et montre que le temps de sérialisation/désérialisation est nettement moins important pour le format JSON, presque six fois moins lent.

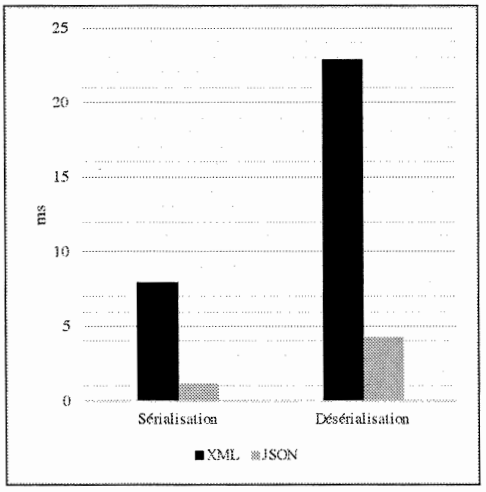

Figure 3.1: Comparaison du temps de sérialisation/désérialisation en millisecondes entre JSON et XML (Sumaray et Makki, 2012).

3.1.6 Gestion des Codes de Statut (GCS)

Pour plusieurs raisons, il arrive qu'une application ne fonctionne pas correctement. Dans ce cas, si elle est bien conçue, celle-ci devrait fournir un maximum d'informations à l'utilisateur ou au développeur concernant le problème survenu. En effet, n'afficher qu'un simple message du type : "*Une erreur est survenue !*" crée une frustration chez l'utilisateur et peut altérer son appréciation de l'application.

Une application cliente Android doit être en mesure d'interpréter le code de statut HTTP retourné par le serveur à chaque communication. Ce code à trois chiffres désigne le résultat de la requête envoyée par le client (Fielding *et al.*, 1999). Ce code peut désigner le succès ou l'erreur survenue lors du traitement de la requête. Il existe différents codes répartis en cinq catégories qui se distinguent par le premier chiffre. Le tableau 3.1 décrit les différentes catégories.

Tableau 3.1: Description des catégories de codes de statut HTTP.

Catégorie	**Description**
1xx	**Information** : Codes informatifs sur le traitement de la requête.
2xx	**Succès** : Codes relatifs au succès de la requête.
3xx	**Redirection** : Codes relatifs à une redirection
4xx	**Erreur Client** : Codes relatifs à une erreur provenant du client.
5xx	**Erreur Serveur** : Codes relatifs à une erreur provenant du serveur.

Très souvent, dans les applications clientes Android, les développeurs ne prennent en considération que le cas de la réussite de la requête (HTTP 2xx) et adoptent le même comportement pour le cas échéant, en l'occurrence un message d'erreur généralisé pour tous les autres cas.

A

```
1  @Override
2  public void onResponse(Response<Model> response) {
3      int statusCode = response.code();
4      if (statusCode >= 200 && statusCode <= 300) {
5        // Récupérer le résultat de la requête.
6      }
7      else {
8      // Afficher un message d'erreur
9      showErrorDialog("Une erreur est survenue !");
10     }
11 }
12
13
14
15
16
17
18
19
20
```

B

```
1  @Override
2  public void onResponse(Response<Model> response) {
3      int statusCode = response.code();
4      if (statusCode >= 200 && statusCode <= 300) {
5        // Récupérer le résultat de la requête.
6      }
7      else {
8          if (statusCode == 401) {
9              // Afficher un message d'erreur
10             showErrorDialog("L'authentification a
                   échouée");
11         }
12         if (statusCode == 404) {
13             // Afficher un message d'erreur
14             showErrorDialog("Ressource non-trouvée");
15         }
16         // ... Gérer un maximum de codes de statu
                pour le cas de l'échec.
17     }
18 }
```

Listing 3.2: Exemples d'implémentation de gestion de codes de statut.

Cependant, il est fortement recommandé de prévoir un comportement approprié pour chacune des catégories de code statut HTTP. Le listing 3.2 illustre la différence entre une gestion d'erreur classique (A) contre une gestion plus élaborée (B). Cette dernière approche permet une meilleure flexibilité dans le débogage de l'application.

CHAPITRE IV

HARISSA, UN OUTIL POUR ANALYSER LES CLIENTS REST ANDROID

Dans ce chapitre, il est question de présenter l'approche outillée qui constitue le cœur de notre travail de recherche, à savoir, l'outil HARISSA. Cet outil permet l'analyse automatique des clients REST Android dans le but de détecter la présence ou l'absence des pratiques de développement citées dans le chapitre précédent.

Dans un premier temps, nous présentons notre approche dans sa globalité pour permettre une vue d'ensemble, puis nous nous attardons sur chacune de ses phases de détection. Nous passons ensuite en revue la construction du modèle de l'application mobile à analyser, ainsi que les règles de détection de pratiques de développement. L'objectif de ce chapitre est donc de permettre une compréhension de notre approche outillée et de ses mécanismes internes.

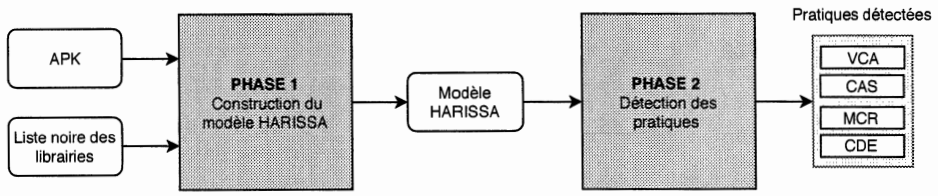

Figure 4.1: Vue globale de l'approche HARISSA

Comme illustré dans la figure 4.1, nous avons adopté une approche en deux phases. La première phase vise à créer une représentation sous forme de modèle de l'application Android à analyser. Cette représentation est le résultat de l'analyse du fichier compressé APK. Dans la seconde phase, il est question d'analyser ce modèle à l'aide de règles de détection qui permettent de vérifier la présence ou l'absence des pratiques de développement liées aux clients REST.

4.1 Phase 1 : Création du modèle HARISSA.

Entrées : Un fichier APK de l'application à analyser, une liste noire de librairies tierces.

Sortie : Un modèle HARISSA contenant des entités, des métriques et des propriétés.

La première phase de notre approche consiste en l'analyse de l'APK de l'application Android à inspecter afin d'en extraire un modèle contenant toutes les informations nécessaires à la détection. Cette phase est constituée de plusieurs étapes comme illustré dans la figure 4.2.

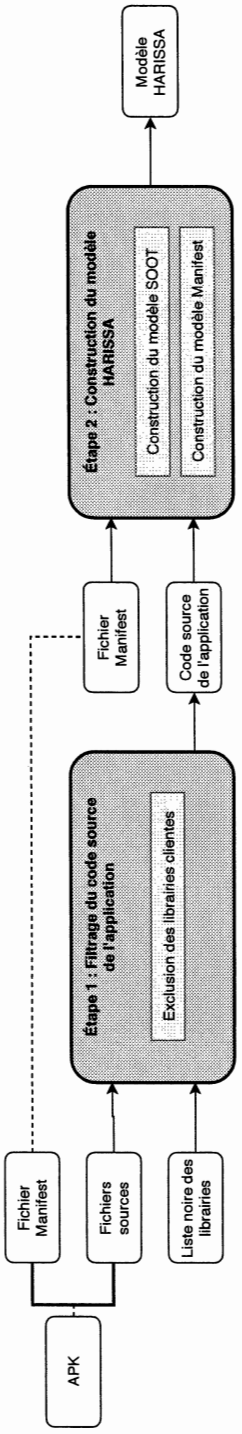

Figure 4.2: Vue détaillée sur la construction du modèle HARISSA.

4.1.1 Filtrage du code source

Cette étape consiste à différencier les classes Java appartenant à l'application analysée de celles provenant des librairies tierces qu'elle utilise. Cette opération constitue présentement un défi courant dans l'analyse statique des applications Android. En effet, l'analyse des fichiers sources non-filtrés complique et fausse sensiblement les résultats de la détection (Li *et al.*, 2016).

Une des solutions possibles à cette problématique serait de restreindre l'analyse aux classes appartenant au paquet principal de l'application. Le nom de ce dernier est explicitement présent dans le fichier *Manifest*. Cependant, cette solution a toutefois ses limites car :

1. Il est possible que le développeur change le nom du paquet principal de l'application sans pour autant le mettre à jour dans le fichier *Manifest*.
2. Il est possible que l'application soit constituée de plusieurs paquets indépendants du paquet principal.

Une autre solution serait d'utiliser une liste noire contenant les noms de paquets des librairies les plus populaires dans les applications Android. Une équipe de recherche (Li *et al.*, 2016) propose une étude sur un très grand nombre d'applications Android (1.5 million d'applications) qui leur a permis de proposer une liste noire contenant plus de 1350 librairies tierces parmi les plus utilisées. La figure 4.3 montre les 20 librairies les plus utilisées avec leur nombre d'apparition.

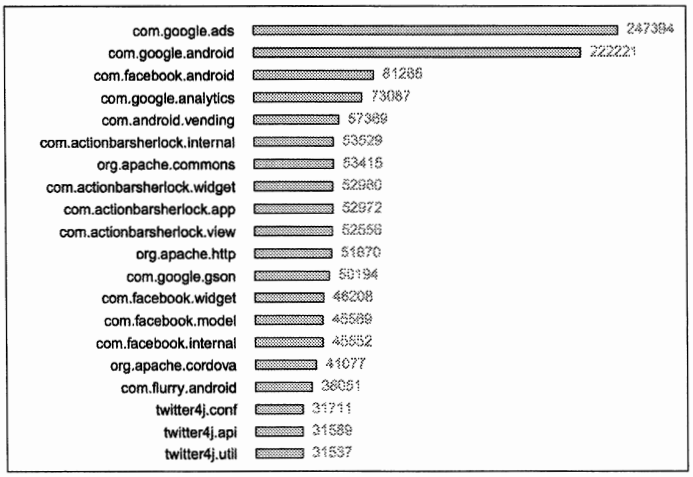

Figure 4.3: Popularité des 20 librairies tierces les plus utilisées (Li *et al.*, 2016).

Notre choix s'est porté sur cette dernière solution avec l'utilisation d'une liste noire. Nous avons cependant étendu manuellement la liste proposée par Li et al. (2016) avec plus de 1150 noms de paquets de librairies populaires dans la communauté des développeurs. Pour ce faire, nous nous sommes référés à des forums tels que *Android Arsenal*[1] et *Ultimate Android Reference*[2]. La liste complète finale est disponible sur notre répertoire GitHub[3].

Pour le filtrage des classes de l'application, nous vérifions pour chacune des classes, la présence du nom de paquet de celle-ci dans la liste noire. Si c'est le cas, elle sera considérée comme provenant d'une librairie tierce et par conséquent filtrée.

---

1. `https://android-arsenal.com/`

2. `https://github.com/aritraroy/UltimateAndroidReference`

3. `https://github.com/kbelkhir/Harissa/blob/master/data/libraries_blist.txt`

4.1.2 Construction du modèle HARISSA

Le modèle HARISSA est construit à partir d'informations provenant du fichier *Manifest* ainsi que des fichiers sources de l'application en cours d'analyse. Certaines informations sont ignorées tandis que d'autres sont mises en évidence pour faciliter et optimiser la détection des pratiques étudiées.

Traitement du fichier Manifest

Il s'agit de créer une représentation objet du fichier `AndroidManifest`. Ce fichier fournit des informations à propos de l'application dans un format XML. Nous avons suivi le processus suivant :

1. **Reconstruction du fichier `AndroidManifest.xml`.** Ce fichier est situé dans le répertoire parent de l'APK. Cependant, la décompression de ce répertoire ne suffit pas pour récupérer un fichier *Manifest* lisible puisque ce dernier est en binaire. Notre outil effectue un traitement sur le binaire à l'aide de la librairie `AXML`[4] dans le but de reconstruire un fichier XML utilisable.
2. **Extraction des différentes balises du fichier.** Une fois le fichier reconstruit. Nous utilisons un analyseur de fichier XML afin d'extraire le contenu des différentes balises contenant les informations sur l'application.
3. **Construction du modèle Manifest.** Les informations extraites dans l'étape précédente sont utilisées pour construire un modèle complet du fichier *Manifest*. Le tableau 4.1 montre le contenu de ce dernier.

---

4. `https://github.com/xgouchet/AXML`

Tableau 4.1: Liste des composants du modèle Manifest.

Composant	Description
`ApplicationName`	Le nom de l'application.
`PackageName`	Le nom du paquet principal de l'application.
`MinSDK`	La version du SDK Android minimale requise.
`UsedPermissions`	La liste des permissions requises pour l'application.

Traitement des fichiers sources

Pour le traitement des fichiers sources, nous avons utilisé un analyseur de code intermédiaire Java : SOOT FRAMEWORK (Vallée-Rai *et al.*, 2000). SOOT est un outil qui permet l'analyse et le traitement de code intermédiaire Java. Ce dernier convertit le *bytecode* en une représentation intermédiaire plus simple à interpréter et analyser. L'outil prend en entrée un dossier compressé dans le format JAR ou APK ainsi qu'un certain nombre de configurations (choix de la représentation intermédiaire, mode d'analyse, etc.). Il retourne par la suite une représentation complète sous forme d'un modèle SOOT correspondant au code intermédiaire présent dans ce dossier.

Construction du modèle HARISSA

La construction de ce modèle repose principalement sur le regroupement des informations recueillies à partir des deux modèles précédents. La création de ce modèle passe par plusieurs étapes :

1. **Tri du modèle SOOT.** Certaines informations extraites de SOOT dépassent le périmètre couvert par notre modèle. En effet, le modèle généré par SOOT contient beaucoup d'informations et de détails qui ne sont pas nécessaires pour notre détection (un graphe complet des appels de méthode, une hiérarchie des classes, etc.). Ces informations alourdissent le

modèle et ralentissent l'analyse de l'application. Un tri est donc nécessaire dans ces informations pour ne garder que celles qui seront utilisées dans la détection.

2. **Identification des entités Android.** Pour la construction du modèle final, nous étiquetons les entités liées à la plateforme Android telles que `Activity`, `AsyncTask` ou encore les classes `R`[5]. Ces dernières ne sont pas identifiées dans le modèle SOOT puisque SOOT est prévu pour l'analyse de code Java seulement.

   L'identification de ces entités facilite et permet d'optimiser la détection de certaines pratiques de développement. Pour identifier lesdites entités, nous avons effectué un parcours de l'arbre d'héritage des classes de l'application afin de déterminer celles qui héritent de classes provenant du cadre d'application Android.

3. **Calcul de métriques.** Bien que peu nombreuses, certaines métriques comme le nombre de classes et le nombre de lignes de code sont également proposées dans le modèle HARISSA.

4. **Construction du modèle HARISSA.** Enfin, le modèle HARISSA est généré à partir du regroupement des modèles SOOT filtré et Manifest, des métriques et entités liées à la plateforme Android. Le tableau 4.2 illustre les différentes entités du modèle final (voir annexe A.1 pour le tableau des propriétés).

---

5. Classes créées automatiquement par le système Android qui servent à l'identification des ressources

Tableau 4.2: Liste des composants du modèle HARISSA.

Composant	Description
HarissaApp	Le modèle final de l'application analysée. Elle contient toutes les informations et métriques de l'application.
`HarissaClass`	Une classe dans l'application. Elle peut soit être écrite par le développeur ou provenir d'une librairie tierce.
`HarissaField`	Une propriété dans une classe.
`HarissaMethod`	Une méthode dans une classe.
`HarissaBody`	Le corps d'une méthode.
`AssignExpression`	Une assignation de valeur.
`InvokeExpression`	Une invocation de méthode.

4.2 Phase 2 : Détection des pratiques

Entrés : Le modèle HARISSA

Sortie : Un fichier listant les pratiques détectées dans l'application.

Contrairement à la détection des défauts de code, la détection des pratiques de développement spécifiques aux clients REST Android n'est pas une tâche simple. En effet, l'implémentation de la plupart de ces pratiques se fait à l'aide des librairies clientes HTTP. Ceci complique considérablement la détection statique de ces pratiques puisque chaque librairie les implémente différemment.

Il est impossible de proposer une méthode de détection statique commune qui soit indépendante du choix de la librairie HTTP cliente (sauf pour la pratique VCA puisque celle-ci est implémentée à l'aide du cadre d'application Android). La seule solution est donc de choisir les librairies les plus utilisées et d'implémenter la détection des pratiques pour chacune d'elles.

Nous avons remarqué en consultant plusieurs discussions dans des sites communautaires de développement Android que les librairies les plus dominantes sont :

- `Java HttpURLConnection`
- `Google Volley`
- `Retrofit`
- `OkHttp`

Cependant, nous n'avons trouvé aucune étude ou classement officiel sur l'utilisation de ces dernières. Nous avons donc décidé de mener notre propre étude qui nous permettra d'identifier les librairies HTTP les plus populaires. Cela a aussi pour but de limiter notre détection aux applications utilisant ces librairies.

Pour cette étude, nous nous sommes basés sur une liste de 70 librairies clientes HTTP récupérées à partir du dépôt de librairies Java : *Maven*[6]. Nous avons étudié leurs occurrences dans plus de 1500 applications Android REST. Nous avons constaté effectivement une prédominance des quatre librairies citées ci-dessus. La figure 4.4 illustre les résultats de notre étude.

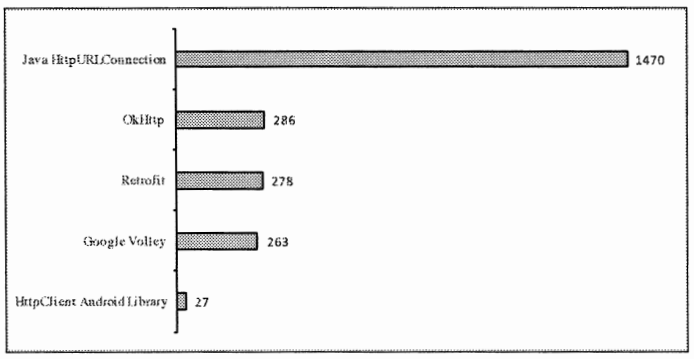

Figure 4.4: Résultats de l'étude sur la dominance des librairies clientes HTTP.

6. https ://mvnrepository.com/tags/http

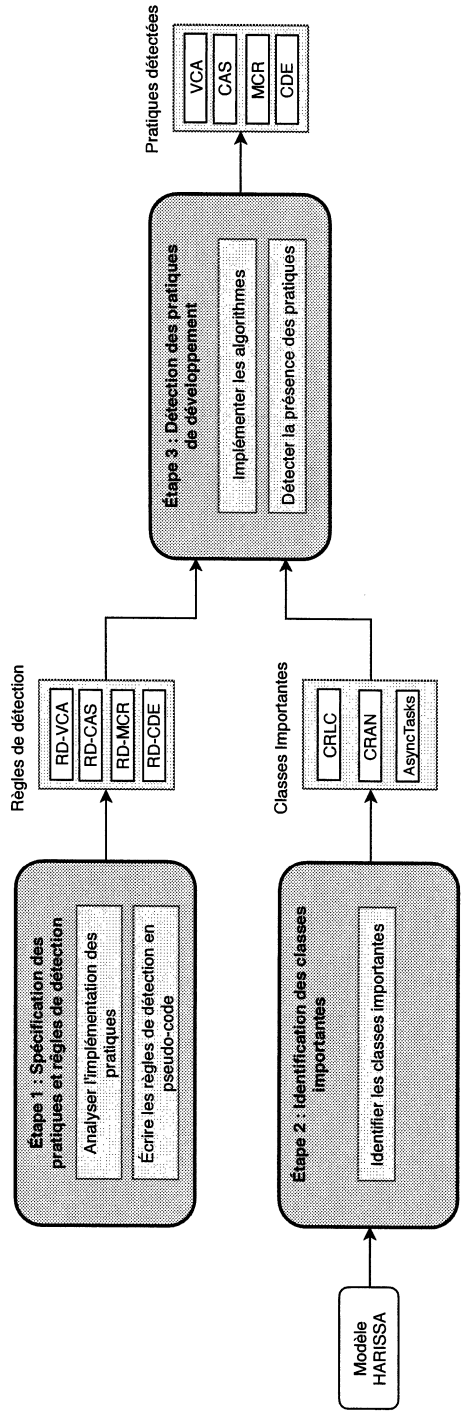

Figure 4.5: Vue globale sur le processus de détection des pratiques HARISSA.

4.2.1 Spécification des pratiques et règles de détection

Après avoir choisi et décrit les pratiques à détecter (chapitre 3), l'étape suivante est la spécification des règles de détection pour ces dernières. Comme expliqué précédemment, il faudrait mettre en place une règle de détection par librairie pour chacune des pratiques, excepté la pratique (VCA) dont la détection se base sur le cadre d'application Android.

Afin de comprendre l'implémentation de chaque pratique, nous nous sommes appuyés principalement sur la documentation officielle des librairies (Retrofit-Doc, 2018; OkHttp-Doc, 2018; HttpUrlConnection-Doc, 2018; Volley-Doc, 2018) ainsi que celle de la plateforme Android (Android-Doc, 2018b).

Afin de détecter la présence ou l'absence de l'une des pratiques étudiées dans l'application, nous nous basons sur certains algorithmes de fouille (AF) que nous avons mis en place pour identifier des actions particulières dans le code intermédiaire de l'application (voir la section A.2 pour les représentations en pseudo-code). Ces algorithmes sont décrits ci-dessous :

AF 1 : Détection de références vers une librairie

Grâce à cet algorithme (voir annexe A.2.1), l'outil Harissa est capable de détecter la présence de références vers une librairie externe donnée dans l'application analysée. Cet algorithme sert principalement dans l'identification des classes importantes. Une librairie est dite référencée si une classe appartenant à l'application contient au mois un des éléments suivants :

1. Un attribut de type externe provenant de la librairie ;
2. Une méthode avec une signature contenant un type provenant de la librairie (un des paramètres ou le retour de la méthode) ;

3. Une annotation avec un type provenant de la librairie.

AF 2 : Détection des invocations de méthodes.

Ce deuxième algorithme (voir annexe A.2.2) permet de détecter et retracer un appel d'une méthode à l'aide de sa signature dans le code intermédiaire. Dans cet algorithme, tous les corps des méthodes provenant des classes ciblées sont parcourus à la recherche d'une invocation avec la signature de la méthode recherchée. Si trouvée, cette méthode est considérée comme invoquée et donc ajoutée par la suite à la liste des invocations.

AF 3 : Détection des instanciations d'une classe.

Cet algorithme (voir annexe A.2.3) permet de détecter et retracer l'instanciation d'une classe en particulier. De la même façon que l'algorithme précédent (AF 2), toutes les méthodes sont parcourues à la recherche d'une instanciation d'une classe. Celle-ci est dite instanciée si au moins une instanciation existe. Les instanciations sont groupées dans une liste.

Règle de détection de la pratique VCA (RD-VCA)

Le code dans le Listing 4.1 est un exemple qui illustre la vérification de l'état ainsi que le type de connexion de l'appareil Android.

```
1 ConnectivityManager cm =
      (ConnectivityManager)context.getSystemService(Context.CONNECTIVITY_SERVICE);
2
3 NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
4
5 boolean isConnected = activeNetwork != null && activeNetwork.isConnectedOrConnecting();
6
7 boolean isWiFi = activeNetwork.getType() == ConnectivityManager.TYPE_WIFI;
```

Listing 4.1: Détection de l'état de connexion de l'appareil (Android-Doc, 2018).

À partir de cet exemple, les étapes clés de cette opération sont :

1. Instanciation d'un objet de type `ConnectivityManager` provenant du cadre d'application Android.
2. Invocation de la méthode `getSystemService()` avec `"connectivity"` en paramètre.
3. Instanciation d'un objet de type `NetworkInfo` provenant du cadre d'application Android.
4. Invocation de la méthode `getActiveNetworkInfo()` provenant de la classe `ConnectivityManager`.
5. Invocation de la méthode `isConnectedOrConnecting()` provenant de la classe `NetworkInfo`.
6. Invocation de la méthode `getType()` provenant de la classe `NetworkInfo`.

L'extraction de ces étapes clés facilite grandement la détection de la pratique VCA. Il suffit maintenant de repérer la présence de ces étapes dans le code intermédiaire de l'application à l'aide des algorithmes de fouilles décrits plus haut (AF1, AF2 et AF3).

L'algorithme 1 illustre la règle de détection de la pratique VCA sous forme de pseudo-code. Cette pratique est dite présente si :

1. Une instanciation de la classe `android.net.ConnectivityManager` est détectée (`ICM`).
2. Une invocation de la méthode `getSystemService()` avec le paramêtre `"connectivity"` est détectée (`ISS`).
3. Une invocation de la méthode `getActiveNetworkInfo()` avec sa signature complète (`IAN`).

4. Une invocation d'une méthode qui permet de vérifier l'état du réseau (VER) ou le type de ce dernier (VTR). Une liste des méthodes est fournie pour cette détection.

Entrée : Modèle HARISSA (MH).

Sortie : Pratique détecté VCA

```
1  ICM ←AppReferenceClasse(ConnectivityManager);
2  ISS ←AppContientInvocationsDe(getSystemService("connectivity"));
3  IAN ←AppContientInvocationsDe(getActiveNetwork);
4  VCA, VTR, VTR ← faux;
5  MRTs ← Liste des signatures des méthodes permettant la récupération du type de
     connexion;
6  MREs ← Liste des signatures des méthodes permettant la récupération de l'état de
     connexion;
7  pour toutes les signatures s dans dans la liste MREs faire
8  |  si AppContientInvocationsDe(s) alors
9  |  |  VTR ← vrai;
10 |  fin
11 fin
12 pour toutes les signatures s dans dans la liste MRTs faire
13 |  si AppContientInvocationsDe(s) alors
14 |  |  VET ← vrai;
15 |  fin
16 fin
17 si (ICM et IAN et (VTR ou VET) ) alors
18 |  VCA ← vrai;
19 fin
20 retourner VCA;
```

Algorithme 1 : Règle de détection de la pratique VCA.

RÈGLE DE DÉTECTION DE LA PRATIQUE CSA (RD-CSA)

Mis à part la librairie `HttpUrlConnection`, toutes les autres librairies proposent deux méthodes différentes pour exécuter les appels de service REST. Une pour les appels synchrones et une autre pour les appels asynchrones.

A

```
1  private final OkHttpClient client = new
       OkHttpClient();
2
3    public void run() throws Exception {
4      Request request = new Request.Builder()
5          .url("url")
6          .build();
7
8      try (Response response =
           client.newCall(request).execute()) {
9        if (!response.isSuccessful()) throw new
             IOException("Unexpected code " +
             response);
10
11       System.out.println(response.body().string());
12     }
13   }
14
15
16
17
18
19
20
21
22
23
```

B

```
1  private final OkHttpClient client = new
       OkHttpClient();
2
3    public void run() throws Exception {
4      Request request = new Request.Builder()
5          .url("url")
6          .build();
7
8      client.newCall(request).enqueue(new Callback() {
9        @Override public void onFailure(Call call,
             IOException e) {
10         e.printStackTrace();
11       }
12
13       @Override public void onResponse(Call call,
             Response response) throws IOException {
14         try (ResponseBody responseBody =
               response.body()) {
15           if (!response.isSuccessful()) throw new
                 IOException("Unexpected code " +
                 response);
16
17           System.out.println(responseBody.string());
18         }
19       }
20     });
21   }
```

Listing 4.2: Différence entre un appel synchrone (A) et un appel asynchrone (B) à l'aide de la librairie `OkHttp` (Okhttp-Doc, 2018).

À titre d'exemple, le listing 4.2 illustre la différence entre un appel synchrone (A) et un appel asynchrone (B) à l'aide de la librairie `OkHttp`. La ligne 8 montre la différence dans la méthode utilisée dans les deux styles de communication :

— La méthode `Response execute()` (Listing 4.2.A, ligne 8) : Elle est proposée par la librairie `OkHttp` pour exécuter les appels de façon synchrone, sauf si celle-ci est exécutée dans un contexte asynchrone (par exemple : `AsyncTask`).

— La méthode `void enqueue(Callback response)` (Listing 4.2.B, ligne 8) : Elle permet de faire des appels asynchrones sans avoir à utiliser les `AsyncTasks`.

Toutes les autres librairies tierces (`Retrofit`, `Google Volley`) proposent des techniques similaires pour distinguer facilement les deux styles de communication. Par contre, pour le cas de la librairie native `Java HttpUrlConnection`, aucune méthode n'est proposée pour l'exécution des appels de façon asynchrone. Le développeur est donc contraint à utiliser les `AsyncTasks`.

Le listing 4.3 représente un exemple d'implémentation d'un appel asynchrone à l'aide de librairie `HttpUrlConnection` en utilisant les `AsyncTasks`. Dans ce code, nous avons une classe `HttpGetRequest` qui hérite de la classe mère `AsyncTask` et qui redéfinit la méthode `doInBackground(Params... params)`. Cette méthode ainsi que toutes les méthodes invoquées par celle-ci sont exécutées dans un fil d'exécution en arrière plan ce qui permet donc une exécution asynchrone (Android-Doc, 2018d).

```
1  public class HttpGetRequest extends AsyncTask<String, Void, String> {
2    @Override
3    protected String doInBackground(String... params){
4      String result;
5      try
6      {
7        URL myUrl = new URL("stringURL");
8        HttpURLConnection connection =(HttpURLConnection) myUrl.openConnection();
9        connection.setRequestMethod("GET");
10       connection.connect();
11       InputStream inputStream = connection.getInputStream();
12       result = inputStreamIntoString(inputStream);
13     }
14     catch(IOException e){
15       e.printStackTrace();
16       result = null;
17     }
18     return result;
19   }
20   protected void onPostExecute(String result){
21     super.onPostExecute(result);
22   }
23 }
```

Listing 4.3: Exemple d'une implémentation d'un appel asynchrone à l'aide d'une `AsyncTask` et la librairie `HttpUrlConnection`.

Les deux approches pour détecter une communication asynchrone sont :

1. Invocation d'une méthode provenant de la librairie qui permet d'effectuer un appel asynchrone.
2. Invocation d'une méthode provenant de la librairie qui permet d'effectuer un appel synchrone dans un contexte asynchrone (`AsyncTask`).

Ces étapes nous permettent de mettre en place notre règle de détection pour la pratique CAS comme illustré dans l'algorithme 2. Il s'agit d'identifier dans un premier temps les appels asynchrones. Tous les autres appels seront identifiés comme synchrones par la suite.

L'algorithme 2 illustre la détection des appels asynchrones (CAS). Il parcourt les redéfinitions de la méthode `doInBackground(Params... params)` dans les classes filles de `AsyncTask` (`ATs`) à la recherche d'un appel REST. Cet appel peut ne pas être présent directement dans le corps de cette méthode mais dans l'une des méthodes invoquées par celle-ci. C'est pour cette raison que nous utilisons un algorithme récursif pour la recherche (voir annexe A.3).

Les appels de méthodes synchrones exécutés dans un contexte asynchrone à l'aide de la classe `AsyncTask` sont marqués comme asynchrones.

Entrée : Modèle HARISSA (`MH`).
Sortie : Communication Asynchrone détectée (`CAS`).

```
1  ATs ← Liste des AsyncTask dans le modèle MH;
2  SBD ← Signature de la méthode doInBackground(Params... params);
3  CA ← faux;
4  pour toutes les classes c dans la liste ATs faire
5      pour toutes les méthodes m dans la classe c faire
6          si la signature de m est égale à SBD et ContientUnAppelREST(m) alors
7              CAS ← vrai;
8              si l'appel est implémenté à l'aide d'une MPL permettant de faire des
                 appels synchrones alors
9                  Marquer la méthode comme asynchrone;
10             fin
11         fin
12     fin
13 fin
14 retourner CSA;
```

Algorithme 2 : Règle de détection de la pratique CAS.

Jusqu'à présent, notre approche permet de détecter les pratiques de développement suivantes :

— Mise en cache des réponses (MCR).

— Configuration de délais d'expiration (CDE).

— Communication asynchrones (CAS).

— Vérification de la connectivité de l'appareil (VCA).

4.2.2 Extraction des classes importantes

Afin d'optimiser la détection de pratiques à partir du modèle HARISSA. Il est important de réduire préalablement le nombre de classes à analyser et filtrer donc toutes les classes qui n'ont pas de relation avec la logique du client de l'application. Cette opération permet de réduire considérablement le temps d'exécution de l'analyse, notamment sur les applications de grande taille.

Pour filtrer les classes de l'application à analyser nous utilisons principalement l'algorithme de fouille AF1 afin d'identifier les classes que nous voulons garder pour l'analyse. Ces classes sont décrites comme suit :

1. **Les classes qui référencent une des librairies clientes étudiées (CRLC).** Identifier ces classes optimise grandement la détection des pratiques qui sont implémentées à l'aide de la librairie cliente HTTP (CAS, MCR, SDE).
2. **Les classes qui référencent le paquet `android.net` (CRAN).** Ce paquet provient du cadre d'application de la plateforme Android. Il contient toutes les classes relatives à l'utilisation du réseau. Il est important d'identifier ces classes puisque c'est dans celles-ci que sont potentiellement implémentées certaines des pratiques étudiées, par exemple, la pratique CAS, à l'aide de la classe `ConnectivityManager`, ou encore une implémentation

possible de la mise en cache à l'aide de la classe : `HttpResponseCache`.

3. **Les classes `AsyncTasks`.** Comme expliqué dans la section précédente, l'identification des classes filles de la classe `AsyncTask` est primordiale pour la détection de certains appels asynchrones dans la pratique CAS.

4.2.3 Détection des pratiques

L'étape suivante est l'implémentation réelle en Java des algorithmes de détection à partir des règles de détection qui sont en pseudo-code. Une fois tous les algorithmes implémentés, il suffit juste de fournir à l'outil l'APK de l'application à analyser ainsi que les paramètres de configuration qui sont décrits dans le tableau 4.3.

Tableau 4.3: Liste des configuration de l'outil HARISSA.

Configuration	**Description**
`-android`	Chemin vers le cadre d'application Android.
`-apk`	Chemin vers l'APK de l'application à analyser.
`-src`	Chemin vers un dossier comportant des APKs.
`-output`	Chemin vers l'emplacement de sortie.
`-exclude`	Chemin vers la liste des nom de paquets des librairies à exclure.

Une fois l'application analysée, HARISSA retourne deux fichiers de sortie dans le format CSV qui contiennent les résultats de l'analyse. Un exemple de chacun de ces fichiers est illustré dans le listing 4.4. Ces derniers sont le résultat d'une analyse sur plusieurs applications Android. Le fichier A contient les résultats de la détection des pratiques pour chaque application. Chaque ligne contient le nom du paquet principal de l'application, les librairies clientes utilisées ainsi que des booléens qui désignent la présence des pratiques de développement étudiées. Le deuxième fichier de sortie B, quant à lui, expose l'implémentation des pratiques MCR, CAS, SDE pour chaque librairie cliente utilisée par l'application. Ceci est

très pratique pour garder une trace sur quelle librairie implémente quelle pratique dans le cas où l'application utilise plusieurs librairies.

A

```
1 PackageName,LibraryClients,CCA,MC,SDE,CA,CS
2 ac.mujeramujer,HttpURLConnection,false,true,true,false
3 com.anantshivam.smartapps.mcqb,OkHttp,false,false,true,false
4 com.apptegy.caneyisd,Retrofit,false,false,true,true
5 com.appshouse.fatimagul,Volley,false,false,false,false
6 com.itrustore.ratengoods,[Retrofit|OkHttp],true,false,true,true,true
7 com.ksl.android,[Volley|HttpURLConnection],true,true,false,true,false
```

B

```
1 PackageName,Client,MC,SDE,CA,CS
2 ac.mujeramujer,HttpURLConnection,false,true,true,false
3 com.anantshivam.smartapps.mcqb,OkHttp,false,false,true,false
4 com.apptegy.caneyisd,Retrofit,false,false,true,true
5 com.appshouse.fatimagul,Volley,false,false,false,false
6 com.itrustore.ratengoods,Okhttp,false,true,false,true
7 com.itrustore.ratengoods,Retrofit,false,false,true,true
8 com.ksl.android,Volley,true,false,true,false
9 com.ksl.android,HttpURLConnection,false,false,true,false
```

Listing 4.4: Exemple des fichiers de sortie de l'outil HARISSA.

CHAPITRE V

EXPÉRIMENTATIONS ET RÉSULTATS

L'approche outillée HARISSA présentée dans le chapitre précédent permet la détection de quatre pratiques de développement liées aux applications mobiles REST. Le fait que cette approche soit automatique nous permet de mettre en place une étude empirique sur un grand nombre d'applications. Une telle étude permet de donner des informations sur les tendances de développement des applications Android REST, et d'avoir une idée générale sur comment ces applications sont développées dans le marché du mobile.

Nous avons réalisé une étude observationnelle, sur plus de 1500 applications Android REST. Nous abordons dans ce chapitre l'expérimentation effectuée afin de répondre à nos questions de recherche. Nous décrivons nos sujets et objets, nous analysons les résultats obtenus et mettons en avant les menaces à la validité concernant l'exactitude et la reproductibilité de notre expérimentation.

5.1 Questions de recherche

Notre étude par observation permet de répondre aux questions de recherche suivantes :

QR1: *Quelles tendances se dégagent en terme d'utilisation des pratiques de développement par les applications Android REST?*

QR2: *Quelles tendances se dégagent en terme d'utilisation des librairies clientes par les applications Android REST ?*

5.2 Sujets

Notre expérimentation porte sur les quatre pratiques que nous avons précédemment décrites dans le chapitre 3 :

- Vérification de la connectivité de l'appareil (VCA).
- Mise en cache des réponses (MCR).
- Configuration des délais d'expiration (CDE).
- Communications asynchrones (CAS).

L'expérimentation porte également sur les quatre librairies clientes HTTP : `HttpURLConnection`, `Volley`, `Retrofit`, `OkHttp`.

5.3 Objets

Dans le but de réaliser notre étude empirique, nous avons procédé au téléchargement, de façon aléatoire, de 9173 applications Android provenant du magasin

d'applications *Google Play Store*[1] à partir du dépôt d'applications *Androzoo*[2]. Nous avons récupéré par la suite les métadonnées de ces applications depuis l'API officielle du magasin. Cette procédure nous a permis de constituer notre ensemble d'applications à analyser.

L'ensemble téléchargé ne comprenant pas exclusivement des applications REST, nous avons dû effectuer un tri afin de filtrer cet ensemble et ne garder que les applications clientes REST. Le tri en question s'est fait en s'appuyant sur deux indices :

1. *L'utilisation de la permission* ***INTERNET***. Les applications Android nécessitent l'utilisation de cette permission afin d'exécuter des tâches en relation avec l'utilisation du réseau internet. Cette information est explicitement définie dans le fichier *Manifest* de l'application.
2. *Le référencement d'une librairie cliente.* La communication avec des APIs REST se faisant principalement à l'aide du protocole HTTP, nous nous sommes donc basés sur la détection de l'une des quatre librairies les plus populaires.

Une fois ces deux indices détectés dans une application, celle-ci est marquée comme "client REST potentiel". Pour une plus grande précision dans la détection de nos pratiques, nous nous sommes assurés à l'aide de notre outil que l'application exécute bel et bien un appel vers une API REST. De ce fait, notre ensemble d'applications Android REST est composé de 1595 applications.

---

1. https ://play.google.com/

2. https ://androzoo.uni.lu/

La figure 5.1 illustre la distribution des applications clientes par catégorie dans le graphe A et par nombre de lignes de code (LOC) dans le graphe B.

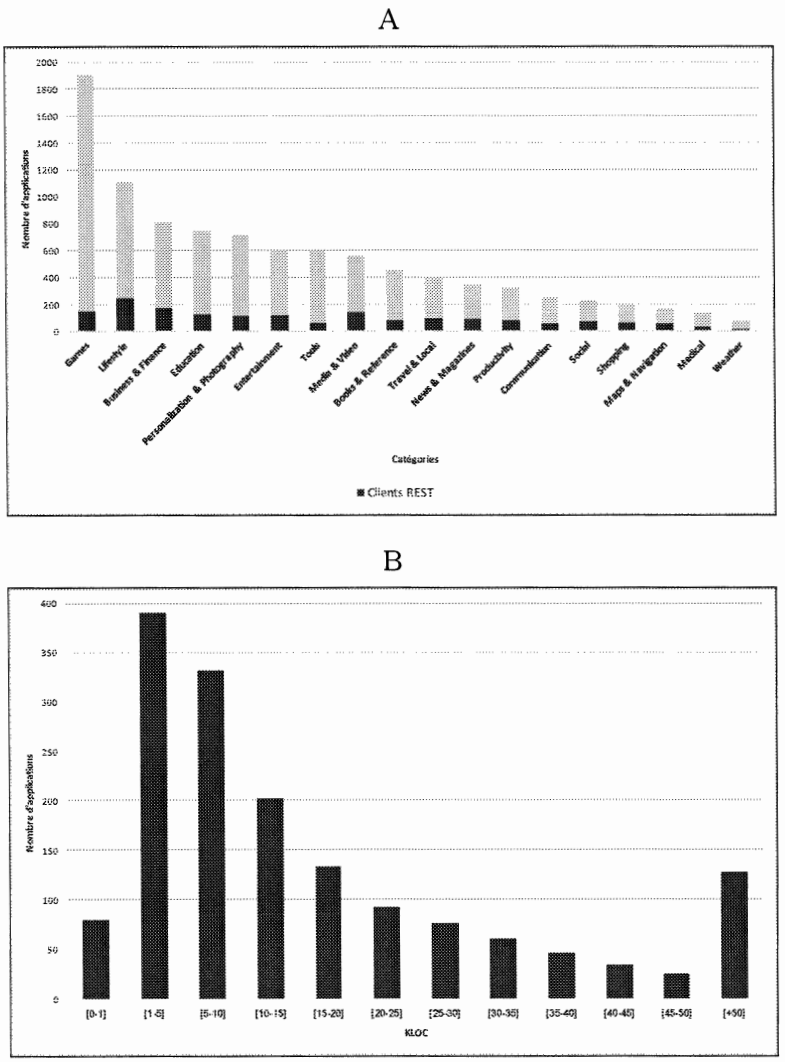

Figure 5.1: Distribution des applications par catégorie (A) et LOC (B).

À travers ces deux graphes, nous pouvons constater la diversité de tailles et de catégories des applications.

5.4 Validation et calcul de précision

Pour la validation de nos résultats ainsi que le calcul de la précision de notre outil HARISSA, nous avons téléchargé 1448 applications supplémentaires à partir du magasin d'applications *F-Droid*[3] car ce dernier contient des applications à code source ouvert. Ceci nous permet de valider manuellement les résultats de notre approche automatique. Pour ce faire, nous avons dans un premier temps effectué les mêmes analyses que sur notre ensemble d'applications principal. Dans un second temps, nous avons validé les résultats manuellement en consultant directement le code source de 80 applications sélectionnées de façon aléatoire.

Le choix du nombre de 80 applications a été motivé par l'objectif d'atteindre un niveau de confiance de 95%. Le tableau 5.1 résume les résultats de notre validation et indique la précision de détection pour chacune des pratiques étudiées. Celle-ci est satisfaisante puisqu'elle varie entre 72.5% et 100% donnant ainsi une moyenne de 92.81%, ce qui démontre la fiabilité de notre outil.

Tableau 5.1: Aperçu sur la précision de détection de HARISSA.

Pratique	Précision
VCA	100.0%
CAS	72.5%
MCR	98.7%
SDE	97.5%
Précision moyenne	**92.8%**

3. https ://f-droid.org

5.5 Résultats et observations

Dans cette section, il s'agit de présenter les observations et résultats constatés durant notre étude empirique. En premier lieu, nous présentons lesdits résultats et observations pour ce qui est de l'utilisation des librairies clientes. En second lieu, nous présentons nos observations et résultats concernant les pratiques identifiées.

5.5.1 L'utilisation des librairies clientes

`HttpURLConnection` est la librairie fournie par le cadre d'application Android pour les communications HTTP avec les APIs REST. De ce fait, elle est la librairie la plus utilisée par les développeurs. Cependant, l'apparition des librairies tierces en 2014 et l'évolution très rapide de ces dernières ont fait perdre près de 50% des parts d'utilisation à `HttpURLConnection` en 2018 comme l'illustre la figure 5.3.

Nous avons observé que depuis 2014 les évolutions des tendances de l'utilisation des librairies `OkHttp` et `Retrofit` sont très semblables. Cette similitude est expliquée par le fait que ces deux librairies se complètent et sont donc souvent utilisées ensemble pour communiquer avec les APIs REST. L'utilisation de ces librairies a significativement évolué entre 2017 et 2018 (60% et 64% respectivement) tandis que `Google Volley` et `HttpURLConnection` n'ont enregistré que 37% et 35% respectivement.

Nous avons observé également que certaines catégories d'applications Android sont davantage portées vers l'utilisation des librairies tierces. On peut citer à titre d'exemple les applications traitant du domaine médical ou encore du domaine social.

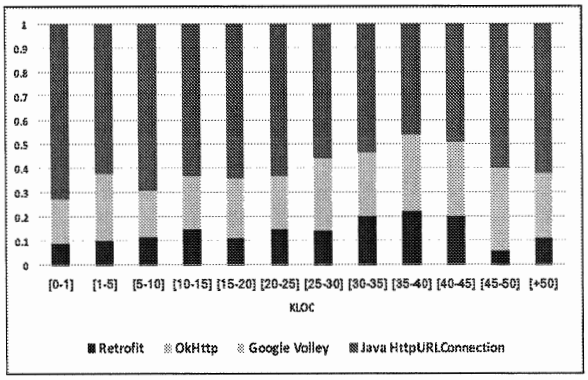

Figure 5.2: Librairies par LOC.

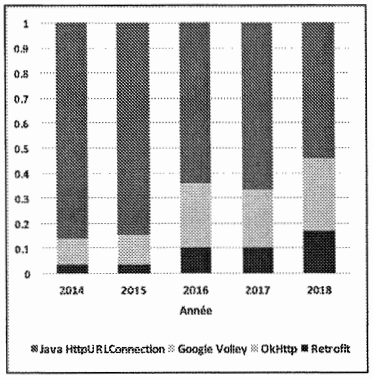

Figure 5.3: Librairies par année.

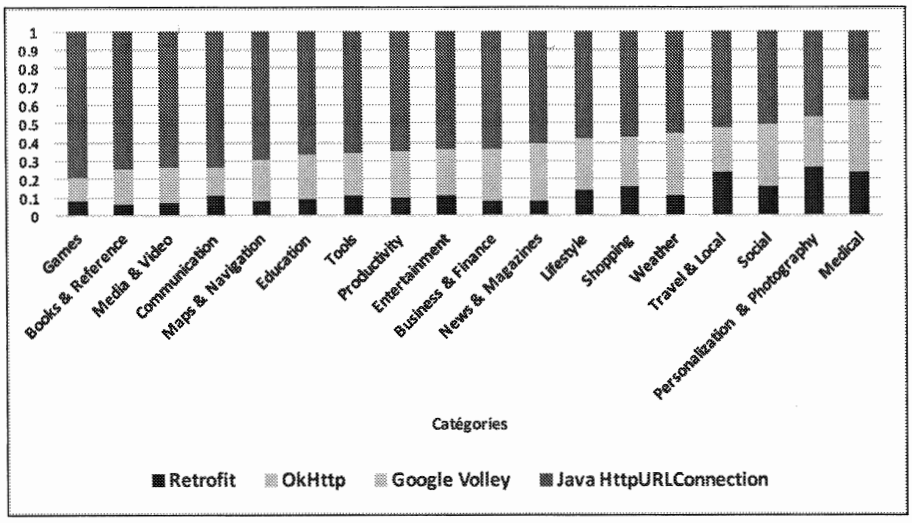

Figure 5.4: Librairies par catégorie.

Observation 1 : Depuis leur apparition, les librairies HTTP tierces occupent une place de plus en plus importante au sein de la communauté de développeurs pour les nombreux avantages qu'elles offrent.

5.5.2 Les pratiques identifiées

Les résultats et observations concernant la détection des pratiques implémentées à l'aide d'une librairie cliente, en l'occurrence la mise en cache des réponse (MCR), la configuration des délais d'expiration (CDE) et les communications asynchrones CAS, sont illustrés dans la figure 5.5. Il est question dans cette section de présenter nos observations pour chacun des résultats obtenus.

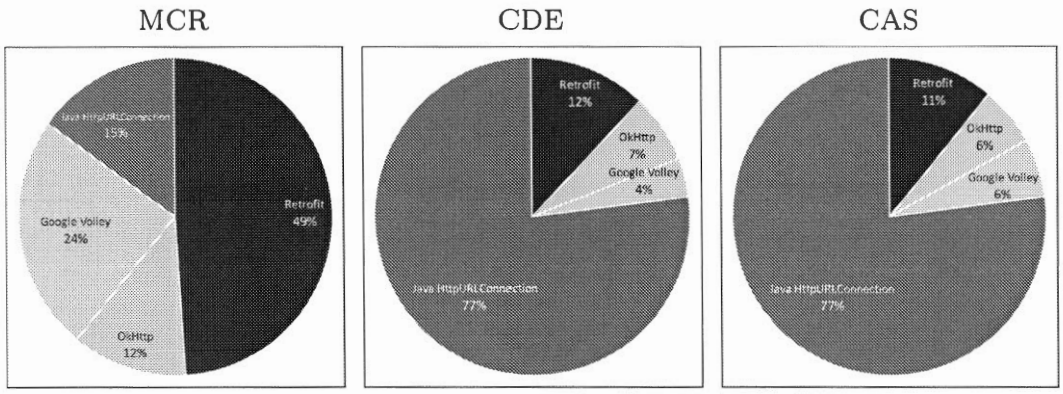

Figure 5.5: Pratiques par librairie cliente.

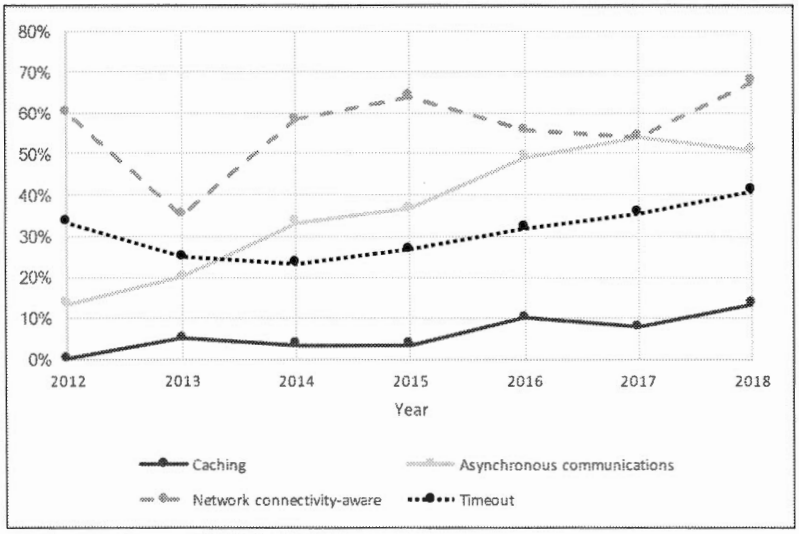

Figure 5.6: Évolution des pratiques à travers les années.

Mise en cache des réponses (MCR)

Bien que la mise en cache des réponses permette une communication plus performante avec l'API REST, nous avons observé un faible pourcentage d'implémentation par les développeurs (figure 5.6). Cependant, il est possible de voir que l'utilisation de cette pratique a évolué ces cinq dernières années.

Nous avons également observé que 85% des applications implémentant cette pratique le font à l'aide des librairies tierces contre seulement 15% pour la librairie native `HttpURLConnection` (figure 5.5.MCR).

Observation 3.1 : Même si la plupart des développeurs ignorent la fonctionnalité du cache, son utilisation a été de plus en plus observée ces cinq dernières années.

Observation 3.2 : Les applications Android qui utilisent les librairies tierces ont tendance à implémenter plus souvent la pratique MCR.

Communications asynchrones (CAS)

Nous avons observé à partir de la figure 5.6 une nette évolution de l'utilisation des communications asynchrones durant les quatre dernières années. Cette évolution de près de 40% peut s'expliquer par les facilités qu'offrent les librairies tierces. Aussi, les communications synchrones sont davantage problématiques dans les dernières versions de la plateforme Android.

Observation 4 : Depuis l'apparition des librairies tierces en 2014, l'utilisation des communications asynchrones avec les API REST connaît une nette progression.

Vérification de la connectivité de l'appareil (VCA)

Nous avons observé que 62% des applications Android REST analysées prennent en considération l'état ou le type de connexion utilisée par l'appareil.

Observation 5 : La vérification de l'état et le type de connexion de l'appareil est une bonne pratique répandue au sein de la communauté des développeurs de clients Android REST.

Configuration des délais d'expiration (CDE)

Cette étude nous a permis d'observer que 36% des clients Android REST configurent des délais dans le cas d'échec d'une requête.

Nous avons également observé que 77% des applications implémentant la pratique CDE le font à travers la librairie `HttpURLConnection`.

Observation 6.1 : Notre étude démontre que l'utilisation des délais dans le cas d'échec d'une requête est une pratique répandue parmi les développeurs de clients Android REST de par sa facilité d'implémentation.

Observation 6.2 : Le configuration de délais est très largement répandue par les clients qui utilisent la librairie `HttpURLConnection`. Ceci s'explique par le fait que celle-ci est la seule librairie étudiée qui n'offre pas une configuration de délai par défaut.

5.6 Réponses aux questions de recherche

En nous basant sur les observations précédemment présentées, nous répondons à nos questions de recherche comme suit :

QR1: *Quelles tendances se dégagent en terme d'utilisation des pratiques de développement par les applications Android REST?*

La figure 5.6 montre que l'utilisation des pratiques considérées comme bonnes ne cesse d'évoluer à travers les années et que les développeurs sont plus conscients des avantages apportés par les bonnes pratiques.

QR2: *Quelles tendances se dégagent en terme d'utilisation des librairies clientes par les applications Android REST ?*

Les développeurs ont tendance à utiliser de plus en plus les librairies tierces depuis leur apparition. Les applications qui en utilisent ont tendance à implémenter davantage de bonnes pratiques. Nous constatons donc une corrélation entre l'utilisation des librairies tierces et l'implémentation des bonnes pratiques.

Ainsi, les développeurs utilisent et devraient utiliser les librairies tierces afin de bénéficier de la facilité d'implémentation des "bonnes" pratiques.

5.7 Menaces à la validité

Dans cette section, nous mettons en évidence les menaces à la validité concernant notre étude observationnelle.

Validité interne : Nous avons supposé que les pratiques de développement étudiées étaient représentatives d'une application Android REST. Cependant, il est possible d'en trouver d'autres qui le soient davantage. Nous avons collecté

manuellement les pratiques étudiées à partir de la littérature, mais il est possible dans des travaux futurs d'étudier la présence d'autres pratiques de développement.

Le code source de certaines des applications analysées a été obfusqué. Même dans sa forme la plus basique, une obfuscation du code source affecte grandement la précision de notre détection puisque les noms des paquets, classes et méthodes sont partiellement ou entièrement modifiés.

Concernant la pratique VCA, on peut se poser des questions sur la pertinence de la détection systématique de la pratique sur 1500 applications. D'un côté, beaucoup d'applications utilisent que très peu de données par jours, on peut citer comme exemple une application pour consulter la météo. Pour ce type d'applications il n'existe pas de réel enjeu sur la connectivité. D'un autre coté, même si la pratique VCA permet d'éviter par exemple de télécharger plusieurs mégaoctets de données sans que l'utilisateur en soit conscient, il faudrait prendre en considération la conscience de l'utilisateur à la consommation de données. En effet, dans la plupart des applications, la plus grande majorité de l'utilisation de données est la source de l'interaction direct avec l'utilisateur (défilement dans un fil d'actualité, visionnage d'une vidéo, ...). Aussi, notre méthode de détection pour cette pratique est limitée. Une application à plusieurs scénario d'utilisation et peut donc consommer les données à partir de plusieurs endroits dans le code, notre méthode de détection ne permet pas de juger la pertinence de l'utilisation de la pratique VCA dans un cas précis. On ne peut pas donc savoir si cette dernière est utilisée adéquatement par le développeur. Ceci est également valable pour les autres pratiques.

Concernant la pratique MCR, nous détectons seulement les implémentations à travers les librairies clientes HTTP. La détection des implémentations personnalisées de cette pratique est compromise puisqu'elles varient beaucoup d'une application à une autre. Aussi, la non-utilisation de la mise en cache peut être légitime dans

certains cas. Il se peut que dans certains scénarios, le développeur veuille s'assurer de toujours avoir l'information la plus à jour possible. Cela est tellement vrai que certains développeurs ont comme pratique l'utilisation de diverses stratégies pour éviter qu'une requête retourne un résultat précédemment mis en cache (ex. entête HTTP spécifiant `no-cache`, l'ajout d'un paramètre spécifiant un *timestamp* ou compteur variant d'une requête à l'autre, etc.).

Validité externe : Les menaces à la validité externe font référence aux obstacles à la généralisation de nos résultats. Même si nous proposons la seule étude sur les pratiques de développement pour les applications Android REST, nous ne pouvons pas généraliser les résultats pour les autres plateformes. Des travaux futurs qui visent d'autres plateformes mobiles sont nécessaires pour confirmer ou infirmer nos observations.

CONCLUSION

Les services REST sont idéals pour fournir des informations aux applications fonctionnant sur des appareils mobiles, comme les téléphones intelligents et les tablettes. De nombreux travaux de recherche ont été proposés concernant l'étude des pratiques de développement des services REST. Cependant, très peu abordent la façon dont les applications Android utilisent/consomment ces services.

Dans le cadre de notre travail de recherche, nous avons proposé un catalogue composé de six bonnes pratiques spécifiques aux clients Android. Nous avons aussi présenté une approche outillée HARISSA qui permet, au moment de la rédaction de ce mémoire, de détecter la présence de quatre des six pratiques de ce catalogue : Vérification de la Connectivité de l'Appareil (VCA), Mise en Cache des Réponses (MCR), Configuration des Délais d'Expiration (CDE) et Communications AsynchroneS (CAS). Notre approche est basée sur l'analyse du fichier *Manifest* et du code binaire (*bytecode*) de l'application. Nous avons validé cette approche manuellement sur 80 applications Android REST et avons obtenu une précision moyenne de 92.81% pour les quatre pratiques détectées.

Nous avons conduit une étude empirique sur plus de 1500 clients Android afin d'investiguer les tendances d'implémentation des quatre pratiques que nous sommes capables de détecter ainsi que les choix des développeurs en matière de librairies clientes pour communiquer avec les APIs REST. À travers notre étude, nous avons observé que certaines pratiques étaient assez populaires chez les développeurs comme la vérification de la connectivité de l'appareil (VCA), alors que d'autres étaient souvent ignorées telles que la mise en cache des réponses (MCR).

Nous avons également observé que l'utilisation des librairies tierces a très rapidement évolué depuis leur apparition en 2014 et devrait facilement dans le futur dépasser l'utilisation de la librairie native `HttpURLConnection`.

Actuellement, plusieurs perspectives d'évolution sont projetées par notre équipe de recherche. La première est d'étendre notre outil afin de compléter la détection pour toutes les pratiques de notre catalogue. Par la suite, il est toujours possible d'étendre également notre catalogue avec de nouvelles bonnes pratiques spécifiques aux clients Android. Une autre perspective est de généraliser notre approche HARISSA pour la détection d'autres types de pratiques en relation avec les applications Android.

Finalement, nous espérons que nos travaux permettront aux développeurs de prendre en considération les bonnes pratiques que nous proposons pour le développement d'une application Android REST. Nous espérons également que notre approche et nos résultats inspirent d'autres travaux de recherche dans ce sens.

APPENDICE A

L'APPROCHE OUTILLÉE HARISSA

A.1 Liste exhaustive des propriétés du modèle HARISSA

Nom	Entité	Description
`name`	`HarissaApp`, `HarissaClass`, `HarissaMethod`	Nom de l'entité
`package`	`HarissaApp`, `HarissaClass`	Nom de paquet de l'entité
`minSdkVersion`	`HarissaApp`	Version minimale compatible du SDK de Android
`permission`	`HarissaApp`	Permission nécessaire à l'application
`type`	`Field`, `Variable`	Type (Objet) de l'entité
`annotation`	`HarissaClass`, `HarissaMethod`, `HarissaField`	Une annotation associée à l'entité
`signature`	`HarissaMethod`	Signature de la méthode
`argument`	`HarissaMethod`	Argument de la méthode
`returnType`	`HarissaMethod`	Type (Objet) du retour de la méthode
`invoker`	`InvokeExpression`	L'objet invoquant une méthode dans l'expression
`invokedMethod`	`InvokeExpression`	La méthode invoquée dans l'expression
`argumentValue`	`InvokeExpression`	La valeur de l'argument de la méthode invoquée dans l'expression

A.2 Algorithmes de fouille

A.2.1 AF 1 : Détection des références vers une librairie

Entrées : Modèle HARISSA (MH), nom de paquet de la librairie externe. (NPL)

Sorties : Présence d'une référence vers la librairie (PL), liste des classes référençant la librairie (LCR).

```
1  pour toutes les classes c dans dans le modèle MH faire
2      AE, ME, ANE, PL ← faux;
3      LCR ← vide;
4      pour tous les attributs a dans c faire
5          si Le nom du type de l'attribut a commence avec NPL alors
6              AE ← vrai;
7          fin
8          si l'attribut a est annotée par un type dont le nom commence avec NPL alors
9              ANE ← vrai;
10         fin
11     fin
12     pour toutes les méthodes m dans la classe c faire
13         si la signature de m commence par NPL ou son corps contient NPL alors
14             ME ← vrai;
15         fin
16         si la méthode m est annotée par un type dont le nom commence par NPL alors
17             ANE ← vrai;
18         fin
19     fin
20     si AE ou ME ou ANE alors
21         PL ← vrai;
22         LCR ← LCR + c;
23     fin
24 fin
25 retourner PL, LCR;
```

A.2.2 AF 2 : Détection des invocations de méthode

Entrée : Modèle HARISSA (MH), Signature de la méthode (SM).

Sortie : Liste des invocations de la méthode.

```
1  MI ← faux;
2  LIMs ← vide;
3  pour toutes les classes c dans dans le modèle MH faire
4  |  pour toutes les méthodes m dans la classe c faire
5  |  |  pour toutes les invocations de méthode i dans le corps de la méthode m faire
6  |  |  |  si la signature de la méthode invoquée est égale à SM alors
7  |  |  |  |  MI ← vrai;
8  |  |  |  |  LIMs ← LIMs + i ;
9  |  |  |  fin
10 |  |  fin
11 |  fin
12 fin
13 retourner MI, LIMs;
```

A.2.3 AF 3 : Détection des instanciations de classe

Entrée : Modèle HARISSA (`MH`), Nom complet de la classe (`NC`).

Sortie : Liste des instanciations de la classe.

```
1  CI ← faux;
2  LICs ← vide;
3  pour toutes les classes c dans dans le modèle MH faire
4      pour toutes les méthodes m dans la classe c faire
5          pour toutes les instanciations d'objet i dans le corps de la méthode m faire
6              si le nom complet du type de l'objet instancié est égale à NC alors
7                  CI ← vrai;
8                  LICs ← LICs + i ;
9              fin
10         fin
11     fin
12 fin
13 retourner CI, LICs;
```

A.3 Algorithme récursif pour la recherche d'appels REST dans une méthode

```
1  Function ContientUnAppelREST(M) :
2      CAR ← faux;
3      AR ← nul
4      si le corps c de la méthode M contient une invocation i d'un appel REST alors
5          CAR ← vrai;
6          AR ← i;
7          retourner CAR,AR;
8      sinon
9          pour toutes les invocations i dans le corps de la méthode M faire
10             si La méthode invoquée mi appartient à l'application et et mi n'a pas
                 déjà été analysée alors
11                 si ContientUnAppelREST(mi) alors
12                     CAR ← vrai;
13                     AR ← i;
14                     retourner CAR,AR;
15                 fin
16                 Marquer mi comme déjà analysée;
17             fin
18         fin
19     fin
20     retourner CAR,AR;
```

RÉFÉRENCES

Android-Doc (2018a). Android Developers. https://developer.android.com/. (Accédé le 07/27/2018).

Android-Doc (2018b). Android-doc. https://developer.android.com/. (Accédé le 08/07/2018).

Android-Doc (2018c). App manifest overview | Android Developers. https://developer.android.com/guide/topics/manifest/manifest-intro. (Accédé le 08/09/2018).

Android-Doc (2018d). Asynctask | Android Developers. https://developer.android.com/reference/android/{OS}/AsyncTask.html#doInBackground(Params...). (Accédé le 08/08/2018).

Android-Doc (2018e). Avoid redundant downloads | Android Developers. https://developer.android.com/training/efficient-downloads/redundant_redundant. (Accédé le 08/10/2018).

Android-Doc (2018f). Connect to the network. https://developer.android.com/training/basics/network-ops/connecting. (Accédé le 07/28/2018).

Android-Doc (2018). Determine and monitor the connectivity status. https://developer.android.com/training/monitoring-device-state/connectivity-monitoring#{Java}. (Accédé le 08/03/2018).

Android-Doc (2018a). Manage network usage | Android Developers. https://developer.android.com/training/basics/network-ops/managing. (Accédé le 08/10/2018).

Android-Doc (2018b). Platform architecture | Android Developers. https://developer.android.com/guide/platform/. (Accédé le 08/08/2018).

Android-Doc (2018c). Processes and threads overview | Android Developers. https://developer.android.com/guide/components/processes-and-threads. (Accédé le 08/13/2018).

Android-Doc (2018d). Understand the activity lifecycle | Android Developers. `https://developer.android.com/guide/components/activities/activity-lifecycle`. (Accédé le 08/09/2018).

Dobjanschi, V. (2010). Developing Android REST client applications. *Google I/O 2010*.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. et Berners-Lee, T. (1999). *Hypertext transfer protocol–HTTP/1.1*. Rapport technique.

Fielding, R. T. et Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation.

Gamma, E. (1995). *Design patterns : elements of reusable object-oriented software*. Pearson Education India.

Gil, B. et Trezentos, P. (2011). Impacts of data interchange formats on energy consumption and performance in smartphones. Dans *Proceedings of the 2011 workshop on open source and design of communication*, 1–6. ACM.

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

Hecht, G., Rouvoy, R., Moha, N. et Duchien, L. (2015b). Detecting antipatterns in Android apps. Dans *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, 148–149. IEEE Press.

HttpUrlConnection-Doc (2018). Httpurlconnection documentation. `https://developer.android.com/reference/{Java}/net/HttpURLConnection`. (Accédé le 08/07/2018).

IDC (2018). Smartphone market share - OS. `https://www.idc.com/promo/smartphone-market-share/OS`. (Accédé le 10/10/2018).

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.

Li, D. et Halfond, W. G. (2014). An investigation into energy-saving programming practices for Android smartphone app development. Dans *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, 46–53. ACM.

Li, D. et Halfond, W. G. (2015). Optimizing energy of http requests in Android applications. Dans *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, 25–28. ACM.

Li, D., Hao, S., Gui, J. et Halfond, W. G. (2014). An empirical study of the energy consumption of Android applications. Dans *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, 121–130. IEEE.

Li, L., Bissyandé, T. F., Klein, J. et Le Traon, Y. (2016). An investigation into the use of common libraries in Android apps. Dans *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, 403–414. IEEE.

Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. et McClosky, D. (2014). The stanford corenlp natural language processing toolkit. Dans *Proceedings of 52nd annual meeting of the association for computational linguistics : system demonstrations*, 55–60.

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D. et Miller, K. J. (1990). Introduction to wordnet : An on-line lexical database. *International journal of lexicography*, *3*(4), 235–244.

Mionskowski, P. (2017). The importance of timeouts - bright inventions blog. `https://brightinventions.pl/blog/http-client-timeouts/`. (Accédé le 08/12/2018).

Moha, N., Palma, F., Nayrolles, M., Conseil, B. J., Guéhéneuc, Y.-G., Baudry, B. et Jézéquel, J.-M. (2012). Specification and detection of soa antipatterns. Dans *International Conference on Service-Oriented Computing*, 1–16. Springer.

Naylor, D., Finamore, A., Leontiadis, I., Grunenberger, Y., Mellia, M., Munafò, M., Papagiannaki, K. et Steenkiste, P. (2014). The cost of the s in https. Dans *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 133–140. ACM.

Nielsen, J. (2010). Website response times. *Nielsen Norman Group*, *21*(06).

OkHttp-Doc (2018). Okhttp documentation. `http://square.github.io/okhttp/`. (Accédé le 08/07/2018).

Okhttp-Doc (2018). Okhttp github documentation. `https://github.com/square/okhttp/wiki/Recipes`. (Accédé le 08/07/2018).

Ollite, I. et Mohamudally, N. (2015). Performance analysis of a 2-tier caching proxy system for mobile restful services. Dans *EUROCON 2015-International Conference on Computer as a Tool (EUROCON), IEEE*, 1–7. IEEE.

Oumaziz, M. A., Belkhir, A., Vacher, T., Beaudry, E., Blanc, X., Falleri, J.-R. et Moha, N. (2017). Empirical study on REST apis usage in Android mobile applications. Dans M. Maximilien, A. Vallecillo, J. Wang, et M. Oriol (dir.). *Service-Oriented Computing*, 614–622., Cham. Springer International Publishing.

Palma, F., Dubois, J., Moha, N. et Guéhéneuc, Y.-G. (2014). Detection of REST patterns and antipatterns : a heuristics-based approach. Dans *International Conference on Service-Oriented Computing*, 230–244. Springer.

Palma, F., Gonzalez-Huerta, J., Founi, M., Moha, N., Tremblay, G. et Guéhéneuc, Y.-G. (2017). Semantic analysis of restful apis for the detection of linguistic patterns and antipatterns. *International Journal of Cooperative Information Systems*, *26*(02), 1742001.

Palma, F., Gonzalez-Huerta, J., Moha, N., Guéhéneuc, Y.-G. et Tremblay, G. (2015). Are restful apis well-designed ? detection of their linguistic (anti) patterns. Dans *International Conference on Service-Oriented Computing*, 171–187. Springer.

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, Klagenfurt, Austria, February 20-24, 2017*, 487–491. `http://dx.doi.org/10.1109/SANER.2017.7884659`. Récupéré de `https://doi.org/10.1109/SANER.2017.7884659`

Petrillo, F., Merle, P., Palma, F., Moha, N. et Guéhéneuc, Y.-G. (2017). A lexical and semantical analysis on REST cloud computing apis. Dans *International Conference on Cloud Computing and Services Science*, 308–332. Springer.

Qian, F., Quah, K. S., Huang, J., Erman, J., Gerber, A., Mao, Z., Sen, S. et Spatscheck, O. (2012). Web caching on smartphones : ideal vs. reality. Dans *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 127–140. ACM.

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

Retrofit-Doc (2018). Retrofit documentation. `http://square.github.io/retrofit/`. (Accédé le 08/07/2018).

Rodrigues, C., Afonso, J. et Tomé, P. (2011). Mobile application webservice performance analysis : Restful services with json and xml. Dans *International Conference on ENTERprise Information Systems*, 162–169. Springer.

Sharkey, J. (2009). Coding for life–battery life, that is. Dans *Google IO Developer Conference*, volume 2009.

Statista (2018). Number of mobile phone users worldwide 2015-2020. `https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/`. (Accédé le 10/10/2018).

Sumaray, A. et Makki, S. K. (2012). A comparison of data serialization formats for optimal efficiency on a mobile platform. Dans *Proceedings of the 6th international conference on ubiquitous information management and communication*, p. 48. ACM.

Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P. et Sundaresan, V. (2000). Optimizing Java bytecode using the soot framework : Is it feasible ? Dans *International conference on compiler construction*, 18–34. Springer.

Volley-Doc (2018). Volley overview Android Developers. `https://developer.android.com/training/volley/`. (Accédé le 08/07/2018).

W3schools (2018). Json vs xml. `https://www.w3schools.com/js/js_json_xml.asp`. (Accédé le 08/12/2018).

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

Yoonsik, C. (2016). *Are Java Programming Best Practices Also Best Practices for Android.* Rapport technique, Technical Report 16-76, Department of Computer Science, The University of Texas at El Paso, El Paso, TX.