TECHNOLOGICAL EDUCATIONAL INSTITUTE
OF CRETE

SCHOOL OF ENGINEERING

DEPARTMENT OF INFORMATICS ENGINEERING

Thesis

An extensible computational framework for the management and analysis of wellness data in an Android environment.

Adamantios I. Kasapakis

Supervisor
Professor Manolis Tsiknakis

HERAKLION
October 2017

Acknowledgements

I would like to express my deepest gratitude to my supervisor Prof. Manolis Tsiknakis for his support during the development of this thesis. His outstanding scientific knowledge and experience were determinant for the accomplishment of this thesis. Additionaly, I would like to thank Dr Matthew Pediaditis and Ms Evagelia Maniadi for their help and guidance.

Abstract

Modern lifestyle leads to increases inactivity in populations around the world, causing health problems, such as hypertension, coronary heart disease, stroke, often related to premature death and an increasing financial burden to the health care system. It is estimated that 31% of the world population is inadequately active and approximately 3.2 million deaths are associated to inactivity. To reduce the impact of inactivity to the society and the health care system, governments are taking actions to motivate people to engage in a heathier way of living. Some of the reason that prevent people from being more active are lack of self-motivation, lack confidence, inability to set personal goals and monitor progress, lack of encouragement and others.

The introduction of smartphones provides a promising way to support such healthier living using mobile applications that can track a person's activity from the sensors available in modern mobile devices. Data collected from those sensors, combined with high accuracy algorithms can distinguish activities such as walking, running or cycling. There is evidence that mobile applications can reduce inactivity and help users engage to a healthier lifestyle by providing insight on workouts, dietary habits sleep and relaxation. Due to those evidences, the computer industry and the research community expresses an increasing interest in fitness and healthcare. Consequently, a great number of fitness applications are available on the market, including features like exercise feedback, personal training sessions, social media integration and others. In addition, big companies have created ecosystems to collect and manage health data and provide software solutions to developers, to easier develop fitness applications.

This thesis aims to create an extensible fitness application as well as libraries that enable the design of various application extensions. Its main characteristic is that it allows functionality to be added by using plug-ins. The application is designed to run on the Android operating system and uses a number of Android's features to distribute plug-ins to the user, provide communication and storage capabilities to the application and the plug-ins, as well as to secure user's data. Finally, for demonstration purposes, two example plug-ins that validate the system's functionality were created.

Περίληψη

Ο σύγχρονος τρόπος ζωής οδηγεί μείωση της φυσική δραστηριότητας των ανθρώπων σε ολόκληρο τον κόσμο, προκαλώντας προβλήματα υγείας, όπως υπέρταση, στεφανιαία νόσο, εγκεφαλικό επεισόδιο, που συχνά σχετίζονται με πρόωρο θάνατο και αυξανόμενη οικονομική επιβάρυνση για το σύστημα υγειονομικής περίθαλψης. Εκτιμάται ότι το 31% του παγκόσμιου πληθυσμού δεν ασκείται επαρκώς και περίπου 3,2 εκατομμύρια θάνατοι συνδέονται με την ελλιπή φυσική δραστηριότητα. Για να μειωθεί ο αντίκτυπος της ελλιπούς άσκησης στην κοινωνία και στο σύστημα υγειονομικής περίθαλψης, οι κυβερνήσεις λαμβάνουν μέτρα για να παρακινήσουν τους ανθρώπους να υιοθετήσουν έναν πιο υγιεινό τρόπο διαβίωσης. Ορισμένοι από τους λόγους που εμποδίζουν τους ανθρώπους να είναι πιο ενεργοί είναι η έλλειψη κίνητρου, η έλλειψη εμπιστοσύνης, η αδυναμία καθορισμού προσωπικών στόχων και παρακολούθησης της προόδου, η έλλειψη ενθάρρυνσης και άλλοι.

Η εισαγωγή των έξυπνων τηλεφώνων, αποτελεί έναν ελπιδοφόρο τρόπο για να υποστηρίξει μια πιο υγιεινή διαβίωση χρησιμοποιώντας εφαρμογές για κινητά τηλέφωνα που μπορούν να παρακολουθήσουν τη δραστηριότητα ενός ατόμου μέσω αισθητήρων που υπάρχουν στις σύγχρονες κινητές συσκευές. Τα δεδομένα που συλλέγονται από αυτούς τους αισθητήρες, σε συνδυασμό με αλγορίθμους υψηλής ακρίβειας, μπορούν να εντοπίσουν δραστηριότητες όπως το περπάτημα, το τρέξιμο ή την ποδηλασία. Υπάρχουν ενδείξεις ότι οι εφαρμογές κινητής τηλεφωνίας μπορούν να αυξήσουν την φυσική δραστηριότητα και να βοηθήσουν τους χρήστες να συμμετάσχουν σε έναν πιο υγιεινό τρόπο ζωής παρέχοντας πληροφορίες σχετικά με τις προπονήσεις, τις διατροφικές συνήθειες, τον ύπνο και τη χαλάρωση. Λόγω αυτών των ενδείξεων, η βιομηχανία ηλεκτρονικών υπολογιστών και η ερευνητική κοινότητα εκφράζουν ένα αυξανόμενο ενδιαφέρον για τη φυσική κατάσταση και την υγειονομική περίθαλψη. Κατά συνέπεια, στην αγορά διατίθενται πολλές εφαρμογές παρακολούθησης της φυσικής δραστηριότητας, που συμπεριλαμβάνουν χαρακτηριστικά όπως ανατροφοδότηση άσκησης, προσωπικές συνεδρίες προπόνησης, συνεργασία με μέσα κοινωνικής δικτύωσης και άλλα. Επιπλέον, μεγάλες εταιρείες δημιούργησαν οικοσυστήματα για τη συλλογή και διαχείριση δεδομένων υγείας και την παροχή λύσεων λογισμικού στους προγραμματιστές, για την ευκολότερη ανάπτυξη εφαρμογών γυμναστικής.

Αυτή η εργασία αποσκοπεί στη δημιουργία μιας επεκτάσιμης εφαρμογής παρακολούθησης της φυσικής δραστηριότητας καθώς και σε βιβλιοθήκες που επιτρέπουν το σχεδιασμό διαφόρων επεκτάσεων για την εφαρμογή. Το κύριο χαρακτηριστικό του είναι ότι επιτρέπει την προσθήκη λειτουργιών χρησιμοποιώντας plug-ins. Η εφαρμογή έχει σχεδιαστεί για λειτουργεί στο λειτουργικό σύστημα Android και χρησιμοποιεί διάφορες λειτουργίες του για τη διανομή των προσθηκών στον χρήστη, την παροχή δυνατότητας επικοινωνίας και αποθήκευσης στην εφαρμογή και τα πρόσθετα, καθώς και για την προστασία των δεδομένων του χρήστη. Τέλος, δημιουργήθηκαν δύο παραδείγματα πρόσθετων που επιδεικνύουν τη λειτουργικότητα του συστήματος.

Table of Contents

Acknowledgements ii
Abstract iii
Περίληψη iv
Table of Contents v
List of Figures vii
List of Tables viii
1. Introduction 1
1.1. State of the art 1
2. User requirements 6
2.1. End user (application requirements) 6
2.2. Developer (API requirements) 6
3. System requirements 7
3.1. System description 7
3.2. Functional requirements 7
3.3. Non-functional requirements 8
4. Component Analysis 9
4.1. Android software stack 9
4.2. Processes and Threads 11
4.3. Manifest 12
4.4. Intents 12
4.5. Binder Framework 13
4.6. Components 14
4.7. Persistent storage [29] 16
4.8. Package Manager 17
4.9. Security 17
5. System design with reuse 19
5.1. Making the plug-in accessible to the user. 19
5.2. Making the plug-in available to Host 20
5.3. The communication mechanism 21
5.4. Security 21
5.5. System overview 22
5.6. Libraries 37
5.7. User interface design 40
6. Development Tools 45
7. Example plugins 46
7.1. PHQ9 plugin (read/write type) 46
7.2. Score plugin (read type) 48
8. Comparison with system requirements 50
8.1. Functional requirements 50
8.2. Non-functional requirements 52
9. Conclusions and future work 53

Διαγράφ

References: 54

ANNEX I: Code 57
a) Core Library 57
b) Client Library 73
c) Host Library 82
d) Host Application 107

Διαγράφ
Διαγράφ

List of Figures

FIGURE 1: THE ANDROID SOFTWARE STACK 9
FIGURE 2: ABSTRACT BINDER COMMUNICATION [21] 13
FIGURE 3: ACTIVITY LIFECYCLE [25] 15
FIGURE 4: COMPONENT DIAGRAM OF THE SYSTEM 23
FIGURE 5: PLUGINDETECTEDRECEIVER CLASS DIAGRAM 24
FIGURE 6: REGISTRY CLASS DIAGRAM 24
FIGURE 7: PLUGINSCANNER CLASS DIAGRAM 25
FIGURE 8: PLUGINSAVER CLASS DIAGRAM 26
FIGURE 9: HOSTSERVICE CLASS DIAGRAM 26
FIGURE 10: DATABASEMANAGER CLASS DIAGRAM 27
FIGURE 11: IPLUGINMODEL CLASS DIAGRAM 27
FIGURE 12: CLIENTSERVICE CLASS DIAGRAM 28
FIGURE 13: CRUD INTERFACE CLASS DIAGRAM 29
FIGURE 14: CRUDREAD INTERFACE CLASS DIAGRAM 29
FIGURE 15: PLUGIN CLASS DIAGRAM 29
FIGURE 16: SCHEMA CLASS DIAGRAM 30
FIGURE 17: TABLE CLASS DIAGRAM 30
FIGURE 18: COLUMN CLASS DIAGRAM 30
FIGURE 19: UPDATEMAP CLASS DIAGRAM 31
FIGURE 20: UPDATETABLEMAP CLASS DIAGRAM 31
FIGURE 21: CUSTOMCURSOR CLASS DIAGRAM 32
FIGURE 22: PLUG-IN DETECTION ACTIVITY DIAGRAM 33
FIGURE 23: HOST-CLIENT CONNECTION ACTIVITY DIAGRAM 35
FIGURE 24: CLIENT SERVICE REMOTE CALLS ACTIVITY DIAGRAM 36
FIGURE 25: HOST LIBRARY CLASS DIAGRAM 38
FIGURE 26: CLIENT LIBRARY CLASS DIAGRAM 39
FIGURE 27: CORE LIBRARY CLASS DIAGRAM 40
FIGURE 28: HOST APPLICATION ACTIVITY MOCK-UP 41
FIGURE 29: SETTINGS MENU SCREENSHOT 41
FIGURE 30: SCREENSHOT OF THE MANAGE PLUGINS ACTIVITY 42
FIGURE 31: SCREENSHOT OF THE UNINSTALL PLUG-INS ACTIVITY 42
FIGURE 33: SWITCH WIREFRAME 43
FIGURE 34: TILEITEM WIREFRAME 44
FIGURE 35: PHQ-9 PLUG-IN'S QUESTIONNAIRE SCREENSHOT 1 47
FIGURE 36: PHQ-9 PLUG-IN'S QUESTIONNAIRE SCREENSHOT 2 47
FIGURE 37: PHQ-9 PLUG-IN'S RESULTS SCREENSHOT 47
FIGURE 38: SCORE PLUG-IN'S SEVERITY TAB SCREENSHOT 49
FIGURE 39: SCORE PLUG-IN'S DISORDER TAB SCREENSHOT 49

List of Tables

TABLE 1: PHQ-9 SCORES AND PROPOSED TREATMENT ACTIONS [37] 46

Διαγράφ

1. Introduction

Modern lifestyle leads people to inactivity, a condition responsible for increased risk of serious health problems and premature death. Such problems have a direct economic impact on the healthcare system and indirect effects e.g. caused by reduced productivity of affected people. For those reasons, health organisations and countries take steps towards motivating the population to be more active, as both the state and its citizens can benefit from it. Due to evidence that mobile applications can positively affect the activity engagement it is becoming an increasing trend among the research community and software industry to develop activity recognition and monitoring algorithms, as well as applications and ecosystems that can help users improve their fitness levels and overall wellbeing. In the same context, the Biomedical Informatics & eHealth Laboratory (BMI Lab) is actively involved in research and development of a broad range of Smartphone-based activity recognition algorithms. Therefore, the BMI Lab could significantly benefit from the development of a fitness application for both testing new algorithms on real users and delivering developed methods to the end-users. In addition, the potential of a plug-in enabled application can simplify the deployment of new components, and avoid monolithic design, providing flexibility to the system by allowing the researcher/developer to customise the application to fit specific research scenarios. From the user's perspective, such a system doesn't force the user to install and use functionality that he/she doesn't need.

1.1. State of the art

Physical activity is defined by WHO as the skeletal muscle movement taking place in activities such as working traveling playing and exercising [1].

Health benefits of physical activity include improvement of muscular and cardiorespiratory fitness, reducing the risk of hypertension, coronary heart disease, stroke, diabetes, various types of cancer (including breast cancer and colon cancer), and depression and weight control [1]. The recommended physical activity for an adult from 18 to 64 years old is at least 150 minutes of moderate-intensity or 75 minutes of vigorous-intensity activity per week and 2 days of muscle-strengthening activities per week [1].

According to WHO, in 2008 around 31% of the world population were inadequately active. Inactivity is associated with approximately 3.2 million deaths every year [2]. The percentage is higher in America where 60% of the population is active bellow the recommended levels[3]. Some of the reasons that are responsible for physical inactivity mentioned by US Centre for Disease Control and Prevention are lack of self-motivation, lack confidence in people's ability to be physically active, inability to set personal goals and monitor progress, lack of encouragement, support, or companionship from family and friends, not finding exercise enjoyable and others [3].

1.1.1. Smartphones and fitness

The introduction of smartphones around the world provides new ways to monitor fitness and health status of people and helps them engage in a more physically active and

healthy lifestyle. According to statista.com the smartphone users were approximately 2.1 billion in 2016.[4]

Mobile devices have a range of sensors (global positioning system, accelerometer, camera, microphone and speaker) and services (calendar) that can collect data from the user's environment. Those data can be used to create high accuracy algorithms for measuring and distinguishing several activities such as walking running and cycling.

1.1.2. Fitness applications

As reported in [5], in march 2017, 23,21% of android users are using Health and Fitness applications and 5,86% medical applications.

They monitor and improve a wide range of fitness activities and behaviours, such as, workouts, weight management, dietary habits, sleep and relaxation. Some of the data that are collected by fitness applications include daily steps, energy output and intake, workout data, sleep quality and weight [6].

According to SMART MOVE randomised control trial [7] smartphone applications significantly improved physical activity over 8 weeks in user older than 16 years old.

There are many different ways to motivate the users to exercise more. Some of the applications use visual or verbal real time feedback to encourage users during exercise and others have virtual coaches to emulate personal training sessions while most of them improve companionship by enabling posting achievements in Social media.[6]

According to [6] a good fitness application should have user-friendly interface, be easy to initiate and reliable during activity, customised to the user's level, enable goal setting, provide review of statistics and periodic summaries and support social networking.

1.1.3. Fitness ecosystems

Due to increasing popularity of fitness applications and the introduction of more and more fitness devices, major mobile companies such as Apple, Google, Samsung, Nike and others, have entered the fitness industry by creating their own fitness ecosystems. The most popular ecosystems will be presented in this paragraph.

1.1.3.1. Google Fit

Along with Google Fit application [8], Google introduced the Google Fit ecosystem that allows developers to upload fitness data to a central repository where users can access their data from different devices and applications in one location. The platform access data from any application, providing user data persistence on the cloud.
Google Fit structural components are the fitness store, the sensor framework, permissions and user controls and the Google Fit APIs.

1. **Fitness Store** is a cloud service, working as a central repository that stores data from the devices and sensors in Google's servers. The framework provides APIs to enable data insert and query operations.
2. **The sensor framework** A set of high-level representations that make it easy to work with the fitness store.
3. **Data sources** represent sensors and consist of a name, the type of data collected, and other sensor details.

4. **Data types** represent different kinds of fitness data, like step count or heart rate. Data types establish a schema through which different applications can understand each other's data.
5. **Data points** consist of a timestamped array of values for a data type, read from a data source
6. **Datasets** represent a set of data points of the same type from a particular data source covering some time interval.
7. **Sessions** represent a time interval during which users perform a fitness activity, such as a run, a bike ride, and so on.
8. **Permissions and user controls** are a set of authorization scopes to request user permission to work with fitness data. The permissions are divided into three groups, activity, location and body each having separate read and write privileges.
9. **Google Fit APIs** enables the creation of applications that support Google Fit on multiple platforms

   a. Android APIs are used for creating Android applications that use Google Fit ecosystem. The APIs include:
   - **the Sensors API** provides access to raw sensor data streams,
   - **the Recording API** provides automated storage of fitness data,
   - **The History API** provides access to the fitness history,
   - **the Sessions API** provides functionality to store fitness data with session metadata.
   - **The Bluetooth Low Energy API** enables your application to look for available BLE devices and to store data from them in the fitness store.
   - **The Config API** provides custom data types and additional settings for Google Fit.

   b. REST API enables storing and accessing user data in the fitness store from applications on any platform.

1.1.3.2. Samsung Health SDK

Similarly to Google, Samsung provides its own fitness ecosystem, Samsung Health SDK [9]. It includes an application and three SDKs designed to provide access to the application and the Samsung's server storage. Those are the Android SDK, Server SDK, and Device SDK.

1. Android SDK

Enables developers to communicate with Samsung's health application from their applications. the SDK is divided in Health Data and Health Service SDKs.

a. **Health Data** includes the following features:

- **Health Store**, that is used to store user's data and provide the connection to the third-party applications in order to read and write data to Samsung Health application.
- **Health Data Type**, that provides custom and predefined data types and is used to access health data for the store.

- **Permissions and User Controls**, provides an interface to the user to manage the access other applications have to his/her health data.

b. **Health Service** includes the following features:
   - **Launching Samsung Health Tracker**, this feature provides the capability to add a tracker from Samsung health to another application to present health data an enable the user to add new data.
   - **Posting Partner Application's Tracker**, that posts a custom tile to the Samsung Health application, containing an icon, health data, and a button for an action.

If a developer wishes to use the Android SDK the Samsung Health application must be installed on the device.

2. Server SDK

It is an SDK that enables developers to use Samsung Health server to access user's health data independently of the Samsung Health application. In addition, there is no restriction in using this library in Android devices like the Android SDK.

3. Device SDK

This SDK is used to provide access to Bluetooth Low Energy devices to Samsung Health, and defines Bluetooth Generic Attributes to enable communication between the devices and the application.

1.1.4. Plug-in Systems

A Plug-in model is a software architecture that allows the extension of a computer program's, the host application, features using pieces of software called Plug-ins. Developers can benefit from plug-in architectures because it enables them to provide modular functionality [10].

One of the most common examples are Web-browsers, that provide extension of the browser's functionality by installing plug-ins, such as Adobe Flash, Java Plug-in, custom themes and other. Another very important example is the Eclipse Intergraded Development Environment, which makes excessive usage of the plugin architecture by extending functionalities of the application almost explicitly with plug-ins.

Benefits from a plug-in architecture [11], [12] include:

- to enable third-party developers to extend application.
- to support addition of new features at run time, without re-compiling and re-distributing the application for new features.
- to reduce the size of the main application and make the system more flexible by using small application cores.
- To remove unnecessary functionalities from the core application and allow the user to customise the application according to his/her needs.

According to [10] there are two types of plug-in architectures, the traditional plug-in architecture and the pure plug-in architecture (everything is a plug-in).

In the traditional architecture, the plug-ins are software bundles that extend a host application's functionality, are compiled independently and communicate with the Host using well defined interfaces. Those plug-ins are mostly used in web-browsers or text editors.

The pure plug-in architecture does not have a Host application, instead it uses a runtime engine for running the plug-ins. Due to lack of a host application, each plug-in act like a host by defining extension points for other extensions to plug.

Historically, due to the numerous benefits that arise from extensible architectures, many frameworks and applications are designed in a way that takes advantage of the characteristics of a plug-in system. Some of the most significant examples are mentioned bellow.

Eclipse

Eclipse [13] is an Integrated Development Environments (IDE) that that can be extended and customised to support programming tasks by pluggable components, the Eclipse Plug-ins.

A plug-in designer declares extension points and a configuration syntax for other plug-ins in an xml file. The platform matches plug-ins with extension points, detects plug-in dependencies and integrates the plug-ins at start-up.

In version 3.01 Eclipse adopted the Equinox an implementation of the OSGi framework specification for the plug-in management.

OSGi standard

The OSGi standard [14] is a specification that describes a dynamic component system for Java, enabling applications to be composed by reusable components, bundles that communicate through services. Bundles can be dynamically integrated on runtime by OSGi framework layers. Those layers include the Execution Environment which defines methods and classes available to a specific platform, modules which provide package management, services that support bundle binding, life cycle layer that enables installation, starting, stopping, updating and uninstalling bundles and the security layer that handles the security.

OSGi is considered the standard component integration framework for Java and it is used by big projects like Atlassian's Confluence and JIRA, Eclipse, IntelliJ, Netbeans and others.

2. User requirements

The system aims to serve two groups of users, the end-user who will use the application and the developer who will design plugins for the system. Therefore, user requirements are divided into the end-user requirements and developer requirements, each representing the needs of the respective group.

The requirements of end-users are called *application requirements* and the developer's requirements are called *API requirements*.

2.1. End user (application requirements)

a. The user should be able to extend the application's (App) functionality without reinstalling it.
b. The end-user should be able to find new plugins and install them on the device.
c. The end-user should be informed during installation time, about which of the system data and features are intended to be used by the plug-in.
d. The user shall be able to uninstall plug-ins.
e. The end-user should be able to activate and deactivate the installed plugins.
f. The end-user should be able to receive notifications related to his/her fitness activity.
g. The end-user should be able to navigate to the Plugin's main activity screen through a button.

2.2. Developer (API requirements)

a. The developer should be provided with an API that allows the creation of plug-ins that extend the Host Application's functionality.
b. The developer should be able to make the plug-in available to the end-user by a plug-in publishing mechanism.
c. The developer should be able to use mechanisms and API components that enables communication between the plug-in store and the Host Application.
d. The developer should be provided with an API that implements create, read, update, delete (CRUD) functionality, which as a result enables storing data to and retrieving data from the Host Application.
e. The developer should be able to define the data structures that will hold the plugin's metadata.
f. The developer should be able to create notifications to be displayed by the Host Application.

3. System requirements

3.1. System description

The plugin system is a system that enables the implementation of an application that stores fitness data as well as other applications that function as plug-ins to extend the functionality of the main application. The system should be supported by an API to provide the necessary tools for the implementation of the Application and the plug-ins and establish effective communication between those parts. The system should also enable the design of two different types of plug-ins, one that is allowed to write data only to data structures it creates and read those data or data from other plug-ins, read/write type or allowed to only read data stored by other plug-ins, read type.
The requirements of the system are described in more detail in the following sub-chapter.

3.2. Functional requirements

a) The system should be able to extend the Host application's functionality on run-time.

b) The system should provide a mechanism to allow the user download and install new plug-ins.

c) The Host application should be able to detect and register installed plug-ins.

d) The system should be able to resolve version compatibility issues.

e) The system should allow the developer to set metadata that describe the plug-ins properties such as API version, plug-in name and packages and others.

f) The system should provide communication mechanisms between the Host and the Plug-ins.

g) The Host should be able to retrieve plug-in metadata and store them in memory during plug-in registration.

h) The Host should be able to initialise a plug-in.

i) The Host Application should not have predefined data-base schema

j) The data base schema should be provided on run-time by each plug-in that requires to store data in the Host Application's memory.

k) The Host Application should be able to create the proper data structures for each plug-in, on run-time to store the plug-in's data.

l) The plug-in should be able to call appropriate methods to store data in the Host application's data-base.

m) A plug-in should be able to call appropriate methods to retrieve its data from Host application's memory.

n) A plug-in should be able to retrieve data stored by other plug-ins if the plug-in developer is aware of the data-base schema used to store those data.

o) The system should provide tools to enable the modification of an existing data-base schema when a plug-in is updated.

p) The system should provide the appropriate methods to implement create, read, update, delete (CRUD) functionality, for data storage.

q) The system should provide tools to enable visual representation of the detected plug-ins in the Host application.

r) The Host application should provide interaction (UI) components to allow activation and deactivation of a plug-in by the user.

s) The Host application should provide user interaction (UI) components to allow the user to navigate to the plugin.

t) The system should protect the user's privacy.

u) User's data should be stored in a private memory.

3.3. Non-functional requirements

a) The system should protect the user's privacy by not exposing data to unauthorised Applications.

b) The system should provide security mechanisms to identify and restrict non-authorised Application communication with the Host Application.

c) The data should be stored in an SQLite data-base.

d) The system should be documented by a Javadoc and developer's guide.

e) The system should be designed in independent code units to make extension, maintenance and testing of the system easier.

f) The system should be validated by unit testing, an automated software testing method which is used to test if small source code units work as expected.

g) The system should be designed in a way that allows future storage migration.

h) The plugin detection operation should run in independent threads to prevent blocking other operations and affect efficiency.

i) The system should implement backwards compatibility when a new API version is implemented in order to support older API versions.

4. Component Analysis

The system is based on Android. It is an open-source operating system based on the Linux kernel and is primarily designed to run on mobile devices. It was developed by Android Inc. in 2003 and in 2005 it was purchased by Google. The first version was released in 2007 and the latest version, Android 8.0 was released in August 2017.

The operating system is designed to run on mobile phones and tablets with a touchscreen interface, and the interaction is basically made by using touch gestures such as tapping and swiping and a virtual keyboard for text input. Google also developed three different versions of android to run on wearable devices, smart televisions and vehicles, the Android Wear, Android TV and Android Auto [15].

4.1. Android software stack

Android is composed of different components. The major building blocks are presented by the following software stack diagram [16].

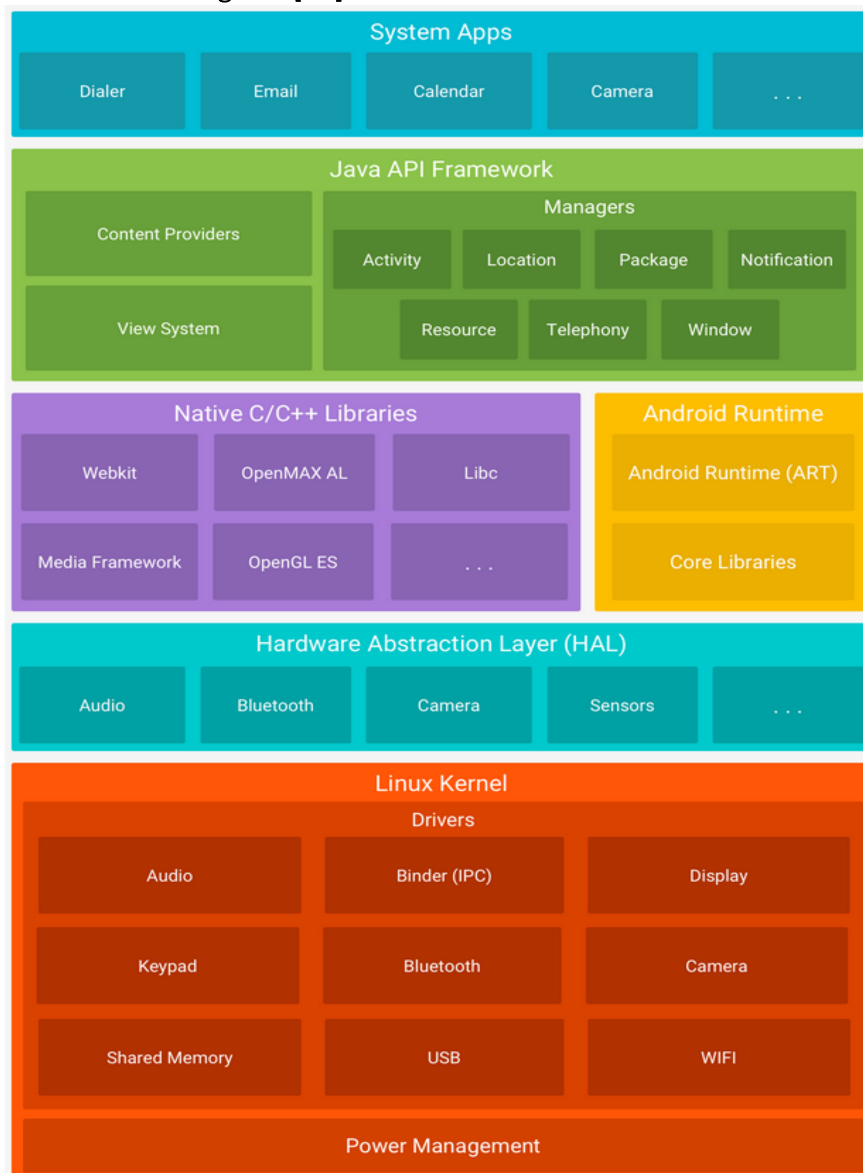

Figure 1: The Android software stack

Each building block is briefly described below.

1. **Linux Kernel**

Android uses a Linux kernel variant, to manage processes and threads and provide applications access to the device's hardware. Every application runs in a process with one or more threads, all of which is scheduled by the kernel. Android kernel has many architectural modifications in order to serve the specific requirements of the Android platform such as optimise the system for low power consumption.

Some of the components present exclusively to the android kernel variant are the binder for inter-process communication, wakelocks for power management, logger and others. Another key difference is that android, unlike Linux, doesn't provide users with Root access to the systems partitions /system [15].

2. **Android Runtime**

Android executes each application in a separate, sandboxed virtual machine. "Dalvik" was the original virtual machine for Android versions and was completely replaced by "ART" in Android 5.0 Lollipop.

Dalvik uses Just-in-time compilation. ART instead, uses Ahead-of-Time compilation, improving overall execution efficiency and reducing power consumption by compiling entire applications into native machine code upon their installation [17].

3. **Native C/C++ Libraries**

These are libraries used to handle parts of the system that are written in native languages such as C/C++. Java applications usually interact with native libraries through Java wrappers.

4. **Java API Framework**

This framework provides the necessary Libraries that enable the creating of Android applications and take advantage of the Android OS features.

Those libraries include:

- The View System, which provides classes for creating UIs.
- The Resource Manager to access resources such as Strings and layout files.
- Activity Manager which manages the activity lifecycle.
- Notification Manager to enable applications to issue notifications on the status bar.
- Content Providers, provide access to application's data from another Application.

5. **System Applications**

System applications are default applications included in the OS and provide key-functionalities to the users such as contacts, messaging, internet browsing default keyboard and others.

These applications are the default applications but the user can choose to set any third-party application as the default application for a specific function.

4.2. Processes and Threads

When an application component starts and the application does not have any other components running, the Android system starts a new Linux process for the application with a single thread of execution. By default, all components of the same application run in the same process and thread (called the "main" thread). If an application component starts and there already exists a process for that application (because another component from the application exists), then the component is started within that process and uses the same thread of execution. However, one can arrange for different components in his/her application to run in separate processes, and additional threads for any process can be created [18].

This subsection discusses how processes and threads work in an Android application.

4.2.1. Processes

In Android, each application runs in a Linux process. A new process is started by the operating system when the first component of an application runs. All the additional components of the application that start after the application process is created, will start by default in the same process, unless it is explicitly specified in the manifest file to start in a separate process. Although applications run in different processes it is possible for two application components to share the same process, as long as the applications share the same Linux ID and are signed by the same certificates.

The Linux process's lifetime in Android is controlled by the operating system by evaluating the importance of the running application components to the user and the available system memory. When the operating system does not have enough memory to run important tasks it reclaims memory by terminating inactive processes. The way android is deciding which processes to stop is determined by ordering processes according to their importance, based on the components running in them and the state of those components.

Processes are classified in four types, and are listed below starting from the most to the least important

- **Foreground process** is a process that either has a running activity that user interacts with, has a broadcast receiver that is currently running, or has a Service that is currently executing code.
- **Visible process** is a process that has an activity that is visible to the user on-screen but not in the foreground, has a service that is running as a foreground service, or is hosting a service that the system is using for a particular feature that the user is aware.
- **Service process** is holding a Service that has been started with the *startService()* method.
- **Cached process** is a process that is not needed and the system is free to stop it.

Normally the system will start terminating cached processes when it runs out of memory, and only in critical situations the system will stop all cached processes and start terminating processes from the other classes.

4.2.2. Threads

Android, as every modern operating system, supports multithreading. When a new application is started, the operating system creates a new thread for that application, the main thread. This thread is responsible for updating UI elements on the screen and therefore it is also called UI Thread. All the components of the application that run in the same process by default, are executed in the main thread.

Android UI toolkit, which contains the graphical components used for building user interfaces, such as the activity, text views buttons and others, is not thread-safe and it should always be accessed strictly by the UI thread. Because this thread is responsible for interactions with the user, time consuming operations could block the thread and cause unresponsiveness of the UI. To avoid blocking the UI Thread, long operations should take place in separate Background Threads. Background Threads are created only by the application and are used to execute any task except from UI updates.

4.3. Manifest

The AndroidManifest.xml [19] is responsible for providing the operating system with information about the applications which are necessary for it to execute. Each Android application should have a manifest file. Some of the information provided to system through this file are

- The java package name, which serves as unique identifier for each application.
- Description of the application components and their class names, as well as the intent filters that define the intents they can handle.
- Permission declaration required to define what API parts and which applications the application can access as well as any permissions needed to allow other Applications to interact with its components.
- Declaration of the minimum API-level required by the application. This is the version of Android the application can run on. For example, API-level 21 is Android 5.0 named Lollipop.
- Defines under which processes the application components will run.

4.4. Intents

Intents [20] are objects in the android framework, assist communication between application components. They are used mainly to start an activity or a service component, deliver a broadcast or send data across applications.

There are two types of Intents, the explicit and the implicit Intent.

Explicit Intents start a component by referring to the full class name. This type of Intent is mainly used to start components in the same application, because the developer should know the class names in order to create an explicit Intent.

Implicit Intents do not start a specific component, instead contain a general action that has to be performed and let the system resolve which application will handle this Intent.

One example is capturing a picture using the camera. Using an implicit Intent, the application could ask for the system to find a camera application to capture the picture and return it to the application.

Anytime an implicit Intent is created the operating system is responsible to find all the appropriate applications that can serve this action. This is done by declaring Intent filters for

any component that should be able to respond to an action, in the application's manifest file during development. The OS then is comparing the created Intent with the declared Intent filters to determine if there are any applications containing components than can handle the Intent. If it matches only one filter, the OS automatically starts that component and delivers the Intent to it. If there are multiple matching filters the system displays a dialog to the user to choose which application he/she wants to use to finish the action.

4.5. Binder Framework

Android applications run in isolated processes. Binder is an android framework [21], [22] based on client-server model, that enables Inter-Process Communication (IPC) mechanisms, providing an application running in one process (client) to execute remote methods in another process's service (server).

The basic steps that take place in the binder framework behind the scenes are the following:

- The passing parameters are decomposed into primitives by a method similar to serialization, called "marshalling."
- The marshalled data are passed between a client process to the server process.
- The data are recomposed with a method similar to deserialization, called unmarshalling.
- The returned values are transferred back to the client process.

Binder requires a Proxy and Stub class, common in most IPC systems, to interact with the two parts. Because writing these classes is complicated, android framework provides Android Interface Definition Language (AIDL) that allows the developer to define an interface and the IDE automatically creates proxy and stub classes for the developer. The parameters and return types that can be defined in AIDL are all Java primitive types, Strings, Lists, Maps, and custom classes that implement the Parcelable interface. A developer should create an AIDL file containing the method signatures in both client and server applications.

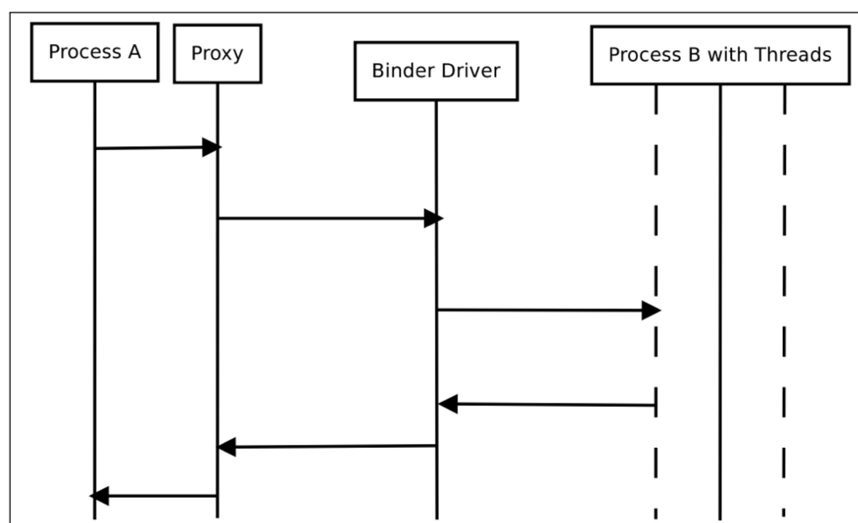

Figure 2: Abstract Binder Communication [21]

At the server application, in a custom service class, a new Binder.Stub() object should be instantiated and implement the auto-generated *stub()* along with all the methods defined in the AIDL file. The interface is exposed to the clients by calling service's *onBind()* which returns the Binder interface used to communicate with the service. The client service should

create a new service connection object to receive a Binder instance that enables remote method calls.

Figure 2 above shows the communication between two processes including the proxy class.

Binder is a very important mechanism and most of the components and call-backs in the operating system, such as Intents, content Providers, and component lifecycle call-backs are invoked via Binders.

4.6. Components

There are four different types of components in the Android Application Framework, that constitute the building blocks of every application. Each component can be an entry point to the Application, and has a distinct purpose and lifecycle that determines the creation and destruction of the component.

Understanding the lifecycle is critical to the effective use of the component. The four types are described below [23].

4.6.1. Activity

One of the most fundamental components of the Android is the Activity class [24]. The Activity provides interaction between the application and the user. Each Activity implements one screen of the Application, with an Application usually having more than one screens to perform different actions. The first screen to come forward when an application starts is the main Activity and each Activity can start others to switch to new screens. It is very common for components to start Activities of another application with explicit Intents or by defining Intent filters that specify which actions that activity can perform.

Activity lifecycle

An *activity* can have four states. It can be active if it is in the foreground of the screen, paused when it has lost focus but is still visible, stopped when it is completely obscured by another activity and shut down when the system asks it to finish or kills the application's process.

Figure 3 shows below presents the activity states as well as the call-back methods that are invoked whenever the state changes.

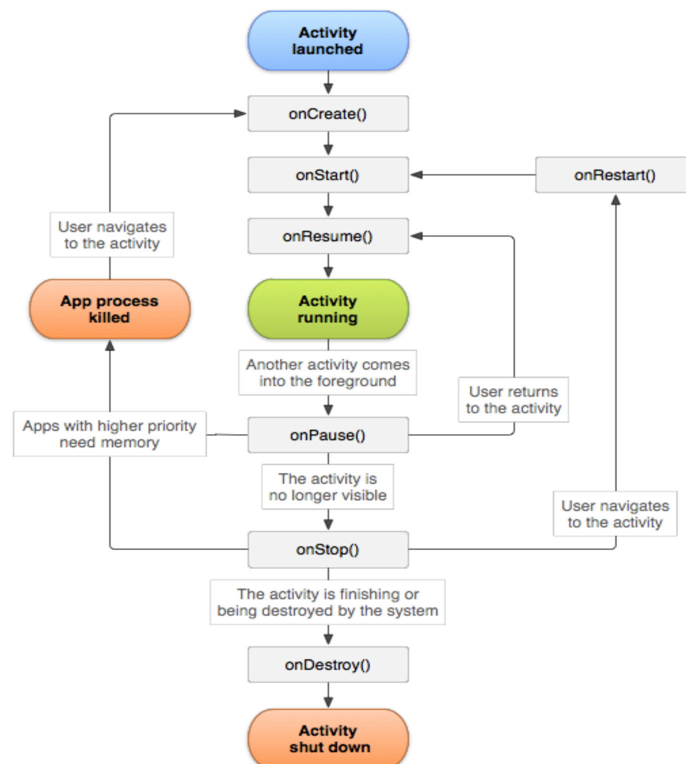

Figure 3: Activity lifecycle [25]

The entire lifetime of the activity takes place between *onCreate()* and *onDestroy()* call-backs. All the set-up of the Activity is done in *onCreate()* and should release the remaining resources in *onDestroy()*.

The visible lifetime takes place between the *onStart()* and *onStop()*. At this time, the activity is visible to the user regardless if it is in the foreground. *onStart()* should be used to register resources that have an effect on the UI such as broadcast receivers and unregister them in *onStop()*.

The foreground lifetime takes place between *onResume()* and *onPause()* and is the time when the activity is in font of all other activities. Those methods can be called very often as the application goes from one screen to another or intents are delivered.

4.6.2. Service

A *Service* [26] is a component that runs in the background without interacting directly with the user through a UI. Services are suitable for long running background operations, such as internet downloads or performing file I/O. A service can keep running in the background even though the user opens another application.

A service can be either started or bound.

- A *Started Service* can be started by calling *startService(Intent)*, passing an explicit or implicit Intent, it runs indefinitely until is terminated by calling *stopService(Intent)* method.
- A Bound Service allows components to bind on it by calling *bindService()* method and communicate with it through *ServiceConnection* interface, and unbind from the service by calling *unbindService()*.

A service is destroyed when all bond components unbind from it and it is not started by any other component. In addition, there are lifecycle call-backs like *onCreate()*, *onStartCommand()*, *onBind()* and *onDestroy()*, to enable developers to run operations at specific service's states.

Bound Services return an IBinder object that provides a communication channel between the binding component and the service. This IBinder object can be a remote interface when the binding component runs in another process. This way the system enables inter-process communication between applications and components that run in different processes.

4.6.3. Content provider

Content provider [27] is a component that enables structured access of an application's data storage. It can be used by the application itself or by other applications running in different processes. It can manage data like video and images or SQLite databases and other structured data.
The advantages of using content providers in an application are that they provide data encapsulation and secure access to the application's storage. Data encapsulation adds an abstraction layer to the storage that helps in data storage migration, without changing the data access interface. In addition, a developer can restrict access to the content provider and allow it only from within the application or define permissions for reading and writing data from other applications.
Content providers are called by passing a URI that determines what table and row to query on. Those URIs are predefined during development time.
As every component, a content provider should be declared during the development of an application in the manifest file.

4.6.4. Broadcast receiver

The Broadcasts [28] is a system similar to a "publish and subscribe" design pattern and is used by Android to send and receive messages when an event takes place. Android system and apps can receive and send broadcasts using intents to notify other applications about an event that they might be interested in.

A broadcast receiver is the component that filters the Intents and determines which ones are send to the receiver. It can be manifest-declared or context-registered. A Manifest-declared receiver is declared in the manifest and registered when the application is installed and any time a relevant broadcast is published, the system invokes the onReceive() method from the associated custom class that implements BroadcastReceiver, to run the code in that method, even if the application is not running.

A context-registered receiver is instantiated within a class along with an intent filter and registered within an application component usually in *onCreate()* or *onResume()* call-backs. It receives broadcasts as long as it is registered while the application is running and should be unregistered in *onPause()* or *onDestroy()* to stop receiving and prevent leakage out of the component's context.

4.7. Persistent storage [29]

Android provides different ways of data storage in or out of the application's private memory. Some of them are shared preferences, internal and external storage and SQLite databases.

4.7.1. Internal and external storage

Files can be saved in the device's internal storage and in a removable storage media like an SD card. By default, internal storage is private to the application and files saved there cannot be accessed by other applications, whereas files in external storage are world-readable and the user can access and modify them by connecting the device to a computer via USB.

4.7.2. Shared preferences

SharedPreferences is a Key-Value storage system in Android that enables saving and retrieving primitive data types in pairs. Data persist in SharedPreferences objects which are saved as files in the private memory of the application and can be obtained by calling *getSharedPreferences()*.

4.7.3. SQLite

Android provides an API to support of SQLite databases, optimised for limited memory devices. The API contains methods to create and update databases and Create, Read, Update, and Delete (CRUD) functionality

SQL Helper class
This class enables the developer to implement call-back methods that help create upgrade and downgrade a database. This class is also responsible for opening a database by returning a SQLiteDatabase object used for reading and writing to the database.

CRUD
The API provides methods *insert()*, *query()*, *delete()* and *update()* to help read, write, delete and modify rows of tables in the database.

4.8. Package Manager

Package Manager [30] is a software tool that manages the installation and upgrade of applications on the device.

The *PackageManager* class provides *PackageInfo* objects that contain important information about the installed packages such as package name and version, and details about all of the components and the permissions required by this package as well as *ApplicationInfo* objects that contain the application's user ID and the path to the application's private data directory.

In addition, it provides methods to retrieve resources from any application and information that are declared in the manifest file, without any security restrictions.

4.9. Security

Android OS is built using Linux kernel, taking advantage of the many security features it provides. Linux is widely used in many security projects and has proven to be a stable and secure during its long history. Some important security features of the kernel [31] are:

- A user-based permission model.
- Process isolation.
- Extensible mechanism for secure IPC.

4.9.1. Sandboxing

Android creates application sandboxes [32] at a kernel level by assigning a unique user ID (UID) to each application and running it in a separate process under that UID. With this technique, the applications have private memory, cannot interact with each other unless they have a permission and have limited access to the OS.

4.9.2. Permissions

Permissions [32] are a security mechanism which provides trusted applications access to resources and data. An application needs to declare permissions in the manifest file by using the *<uses-feature>* element, to get access to features such as camera, internet access and others. The system requests the user's approval on installation time, and if the user grands the permissions the application can hold them until it is uninstalled.

Applications can also define custom permissions by using the *<permission>* element in the manifest file to control which apps can communicate with its components. An important field of custom permissions is the protection level which characterises the risk level of the permission. It can be normal, if the permission is low risk, dangerous, involving high risk actions like accessing private data and controlling the device, signature, a permission that is granted automatically by the system when two applications are signed with the same certificate and system, that the system grants to applications are in the Android system image.

4.9.3. Application signing

Every application that runs on Android must be signed by the developer in order to be successfully installed [33]. Code signing enables the identification of the application author using certificates generated the developer. Those certificates associate the user ID with the application ensuring that one application cannot access another and guaranteeing that the application is installed unmodified to the device. Furthermore, this mechanism allows the system to grand signature-level permissions to components signed with the same signature.

5. System design with reuse

Although there is no dedicated framework to build plug-ins in Android there are many techniques that enable the design of plug-ins.

In android, a user can extend the systems functionality by installing applications, and the operating system provides tools to deploy and manage those applications effectively. In this system, a plug-in is represented as an Android application. This enables to take advantage of the OS's application management and inter-process communication mechanisms that allows installation and communication between the Host and the Plug-ins.

In a plug-in system, the core application provided services to register the plug-ins and a communication interface to allow data interaction. The plug-in's functionality is dependent to those services. More specifically the system:

- Makes the plug-in accessible to the user.
- Makes the plug-in detectible to the Host application.
- Provides communication mechanisms between the plug-ins and the host application.
- Ensures user's data privacy.

5.1. Making the plug-in accessible to the user.

A very important feature of any plug-in system is to be able to expose plug-ins to the user so that the user can download and install them. There are various different ways to provide plug-ins to the user.

Two ways to distribute software in android are through a custom server, by downloading and installing applications from a website and Google's official Android Play-Store.

Distributing through a custom server has many disadvantages. One of them is that it is necessary to create and maintain a server to host the installation files as well as an up-to-date roaster file, for example a JSON file to hold the plug-in line-up. In addition, although the Android OS doesn't restrict the user to install applications from outside of the official store, it requires from the user to change the "allow installation from Unknown sources" security setting, which is generally considered an insecure practice, usually causing suspicion to users.

In contrast distribution by Google Play doesn't require the user to bypass any security settings and the store takes care of any compatibility issues with the device. Furthermore, it is not necessary to maintain a server to keep the installation files.

Our approach is to use Google Play, and take advantage of the advanced features it provides for application distribution.

The plug-ins are made available to the user by a "more plug-ins" settings button that navigates the user to the store to download plug-ins by sending an intent to the operating system to start the Google Play application on the user's device. The intent contains a "*market://search?q=*" URI that includes keywords used by Google Play to search and enlist only our plug-ins. In the application store, the plug-ins contain those keywords in the description text in order the store to be able to detect them during the search. The

user can choose any plug-in from the returned results and install it as any other application.

More information about linking an application to Google Play can be found in [34].

5.2. Making the plug-in available to Host

There are two techniques the Host application can use detect plug-ins. One is the Broadcast-and-respond technique by which the host sends a Broadcast Intent at regular points of time and the plug-ins are responsible to respond to this intent to inform the host they exist. The limitation of this approach is that newly installed plug-ins will not respond to the intent unless they manually run once.

A second approach is scanning with *PackageManager*. In order for this approach to work each plug-in should have a service that declares a custom *IntentAction* in the manifest file. The *PluginManager* queries the system for components that declare this custom ActionIntent and get the *ServiceInfo* of those services. From *ServiceInfo* object the host can retrieve information of the service as well as any meta-data declared helping to describe this component.

In addition to the second approach, the system can watch Package-related broadcasts to determine when an application is installed, updated or uninstalled. By this approach.

In our system, we used the second approach which provides a more deterministic plug-in detection which is being manged from the host's side and does not depend on whether the plug-ins have run. More specifically, the plugins are required to declare an intent filter with the action "**<action android:name="android.intent.action.PLUGIN"** />" in a service. In addition, the same service component should include meta-data definitions which describe the plug-in's name, activities package, version and database version of the plug-in, API level and type of the plug-in. The following code snippet shows the service definition in the manifest file of one of the example plug-ins we designed.

```
...

<service
  android:name=".PHQ9Service"
  android:enabled="true"
  android:exported="true">
  <intent-filter>
    <action android:name="android.intent.action.PLUGIN" />
  </intent-filter>

  <meta-data
    android:name="plugin_name"
    android:value="PHQ-9" />
  <meta-data
    android:name="activity_intent"
    android:value="com.intent.pluginmodel.phq9.MAIN" />
  <meta-data
    android:name="version"
    android:value="1" />
  <meta-data
    android:name="api_level"
    android:value="1" />
  <meta-data
    android:name="dbVersion"
    android:value="1" />
  <meta-data
    android:name="type"
    android:value="@string/plugin_type_read_write" />
</service>
```

```
...
```

5.3. The communication mechanism

In the system developed for the needs of the present thesis, the host communicates with the plug-ins using services. The host exposes a remote binder interface by a host service defined in the host library, on which the client binds any time a data transaction needs to take place. The binder interface provides methods to create, query and insert data to host's database and broadcast notifications as well as methods to expose plug-ins status as it is stored in the host registry, such as whether the plug-in is initialised, and activated and the schema version that is initialised.

The information about the plug-in's status are very important as they determine whether the plug-in is properly set-up to interact with the data-base and that the user approves the interaction with the host by activating the plug-in.

5.4. Security

The main security risk of the system is allowing unauthorised android applications to connect to the host application and perform remote calls to the services methods. This is addressed by defining signature level permissions to the components of the system that enable communication with other applications on the device.

Therefore, we defined a signature level permission in core library's manifest file and declared the permission in the HostService's tag in the systems library's manifest file. The following code snippets show the definition of the permission in core library,

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.kasapakis.adam.plugin_model_library_core">

   <permission
android:name="com.kasapakis.adam.plugin_model_library.permission.HOST_SERVICE_PERMISSIO
N"
           android:label="@string/host_service_permission_name"
           android:description="@string/host_service_permission_description"
           android:protectionLevel="signature"/>
   ...
</manifest>
```

and the declaration of the permission in service's tag:

```
...
<service
   android:name=".service.HostService"
   android:enabled="true"
   android:exported="true"
   android:permission="com.kasapakis.adam.plugin_model_library.permission
         .HOST_SERVICE_PERMISSION"
   />
...
```

The library that contains the ClientSevice requests to use this permission in the library's manifest file, as shown in the following code snippet, and the service is able to communicate with the host.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.kasapakis.adam.plugin_model_library_client">

   <uses-permission android:name="com.kasapakis.adam.plugin_model_library.permission
          .HOST_SERVICE_PERMISSION"/>
   ...
</manifest>
```

Since both libraries are signed with the same signature, the operating system automatically grands them and restricts binding from other applications that don't hold the same permission.

Another part related to security is saving user's data to public memory making them accessible to third party applications. Since the persistent storage mechanisms we use are shared preferences and SQLite data, which both store data in applications private memory, user's data are kept private.

5.5. System overview

The system is designed to work in devices running Android version 5.0 or later also named Lollipop. It is composed from a number of components, which are related to the User interface, the detection of the availability of plug-ins as well as communication and persistent storage.

This sub-chapter aims to describe the major parts of the application and the APIs.

5.5.1. System structure

The application's component diagram is presented in figure 4.

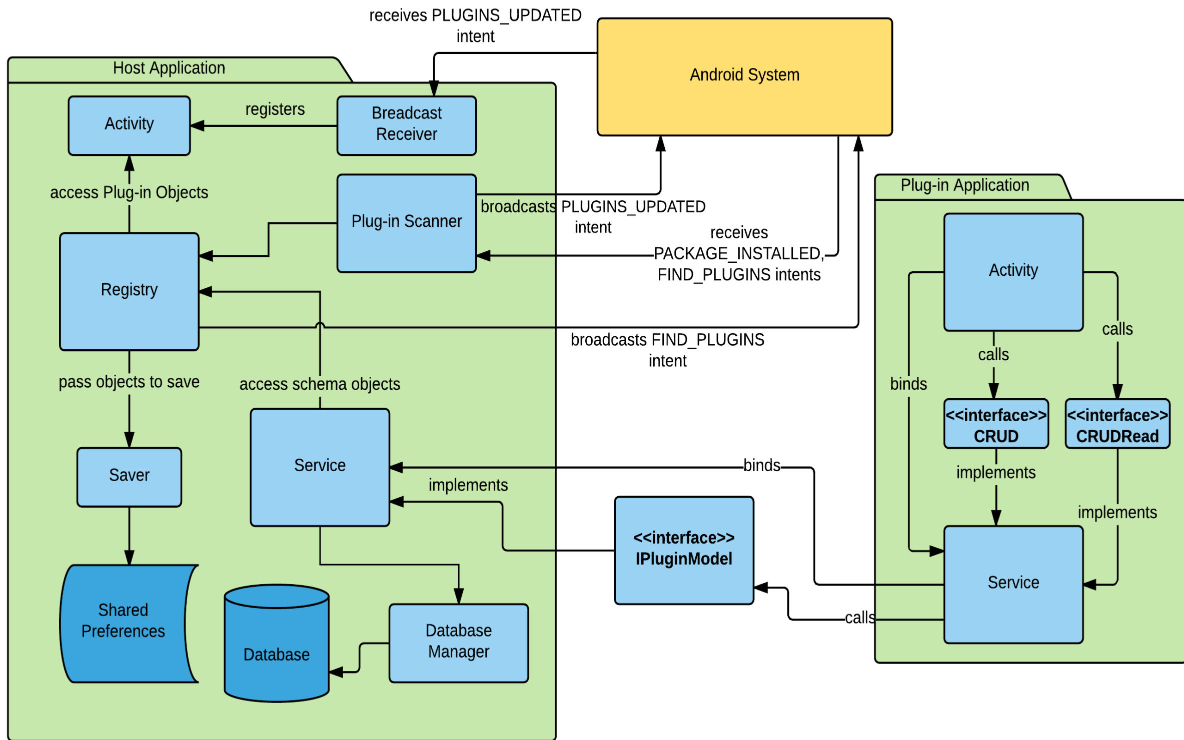

Figure 4: Component Diagram of the System

The Host application package in figure 4 represents the fitness application and the plug-in application package represents one of the plug-ins that extend fitness application's functionality. Each of those packages runs in an isolated process and IPluginModel interface provides the communication between the two processes. Additionaly, android system represents the part of the system that manages the intent broadcasting, used to notify components of the application about events that take place during the execution.

5.5.2. Host Application

5.5.2.1. Activity

The activity is the host application's main entry point. It works as an interaction component between the user and the application. In this class, the user interface is set-up and updated according to the detected and activated plug-ins and is responsible for changing their activated state, and navigating to them as well as installing and removing plug-ins from the device. Details about the user interface design are presented in 5.7.

In addition, the activity interacts with the Registry class to retrieve information about the plug-ins. Furthermore, the PluginDetectedReceiver is registered in the activity to be notified anytime a new plug-in detection has taken place, to update the UI accordingly.

5.5.2.2. Broadcast Receiver

The Broadcast receiver is instantiated and registered in the activity in order to receive broadcasts whenever the *Plug-in Scanner* has finished a new plug-in search.

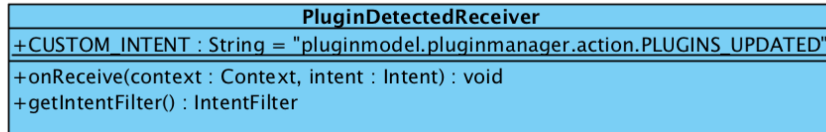

Figure 5: PluginDetectedReceiver class diagram

It extends android's Broadcast receiver class, which requires to implement the onReceive() method that is called when the component receives a broadcast. In our class, this method is declared abstract so it is implemented in the activity, where the appropriate calls to update the UI take place.

5.5.2.3. Registry

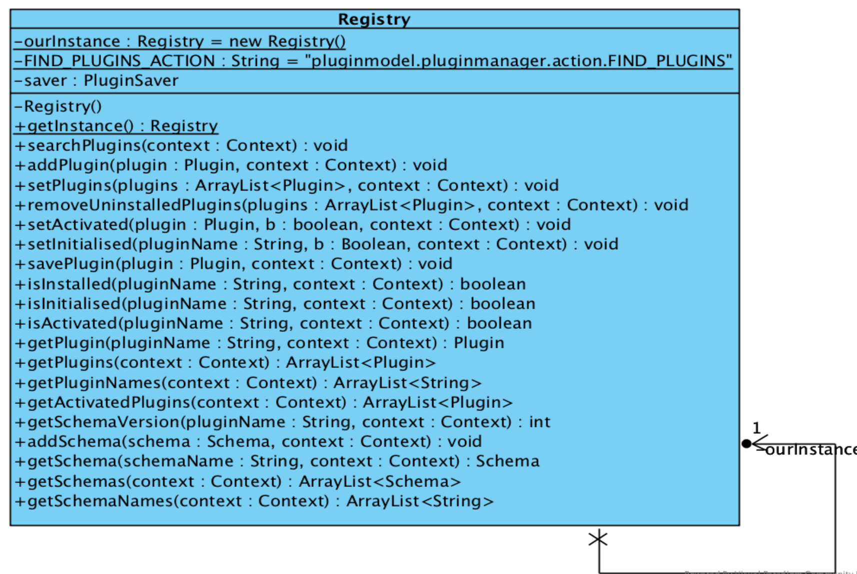

Figure 6: Registry class diagram

The Registry is one of the most important components of the system and is responsible for keeping information about the detected plugins as well as the state of the plug-ins, represented by *Plugin* objects. In addition, it is responsible to keep information about the different tables each plug-in of type *read/write* is creating. This information is described by *Schema* objects.

The Registry is interacting with most other major components to provide or receive information about plug-in's and database's status. More specifically it receives data from the *Plug-in scanner* and saves them in memory by using the *Saver* component. It provides a list containing *Plugin* objects to the *Activity* to visually represent them to the user and receives calls from it to update the activated status of a plug-in anytime a user activates or de-activates it. In addition, it receives *Schema* objects from the *Service* component and saves them to memory and provides back that information whenever the service requires them. Finally, *Registry* can broadcast a custom intent that notifies the *Plug-in Scanne*r to re-scan for plug-ins.

Since registry is accessed from multiple parts of the system, it can cause conflicting requests for the same resources. To avoid that it was declared as a singleton.

5.5.2.4. Plug-in scanner

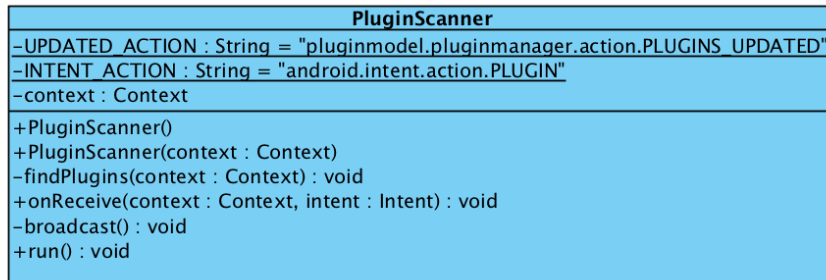

Figure 7: PluginScanner class diagram

The Plug-in scanner is a class that detects services that implement the custom action *"android.intent.action.PLUGIN"* that is declared in the plug-in's manifest file to be able to be detected by the host. The detection is being made using the android *PackageManger*. The scanner constructs the Plug-in objects from the detected services and passes those objects to Registry.

The Scanner extends the *BroadcastReceiver* class to be able to receive broadcasts related to the detection of the services. These broadcasts are produced by the system whenever a new application is installed on the device. More specifically, when the following actions take place:

- <action android:name="android.intent.action.PACKAGE_INSTALL" />
- <action android:name="android.intent.action.PACKAGE_ADDED" />
- <action android:name="android.intent.action.PACKAGE_REPLACED" />
- <action android:name="android.intent.action.PACKAGE_REMOVED" />

the onReceive() method of the receiver is called and executes the code inside the methods body. In addition to the package related broadcasts it receives the custom intent broadcasted by the Registry:

- <action android:name="pluginmodel.pluginmanager.action.FIND_PLUGINS" />

Whenever this component receives any of the broadcasts mentioned above, it is forced to call the findPlugins() method to search for installed plug-ins and then broadcast another custom intent to notify the Activity that a new detection is finished and that it can reload the plug-ins from Registry and refresh the UI.

5.5.2.5. Saver

PluginSaver is called by the Registry to save data to the applications Shared Preferences. Shared preferences can hold map objects in the form of *JSON* format therefore in order for the *PluginSaver* to store objects it uses a Google library called *Gson*[1] that converts Java objects to *JSON* that are accepted by the Shared Preferences.

Plug-in saver is holding Plugin objects in one Shared preferences file and Schema objects in a different file.

[1] https://github.com/google/gson

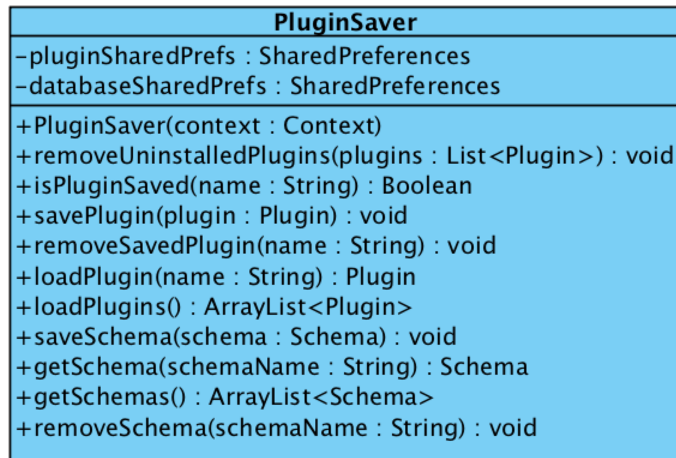

Figure 8: PluginSaver class diagram

5.5.2.6. HostService

The HostService is the main component that works as a slot, and provides an IPluginModel interface used by the plug-ins to communicate with the host application.

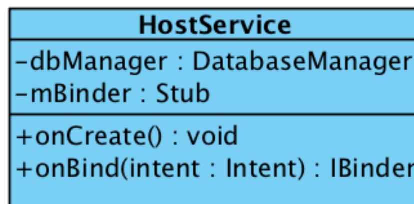

Figure 9: HostService class diagram

It extends the Android's Service class which means that it implements the onBind() method enabling other components to connect to and interact with the service. Service exposes methods through the IPluginModel interface which is explained in more detail in subchapter 5.5.3, to interact with the database, broadcast notifications, as well as provide information about the plug-in's status. In order for the database interaction to take place the service instantiates a *DatabaseManager* object. The plug-in's status is provided by the *Registry* object.

5.5.2.7. DatabaseManager

The *DatabaseManager* is a class which is responsible for creating a database object and for using it to store and load data from the database. This class is instantiated by the *HostService* and uses its methods to create, insert and query data to return to the plugins.

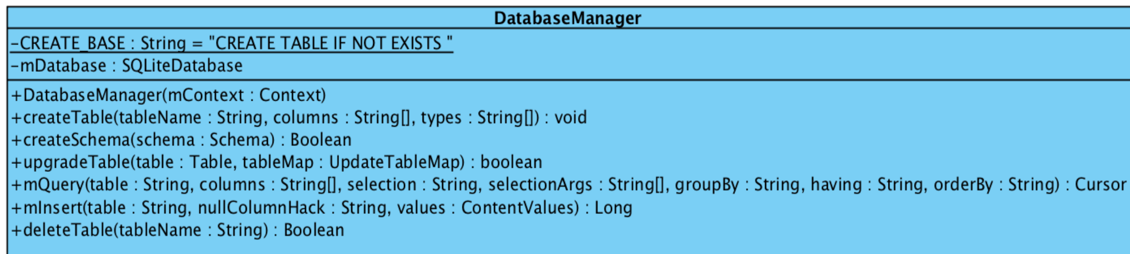

Figure 10: DatabaseManager class diagram

The methods createSchema() and upgradeTable() execute the SQL statements using transactions, guarantying that either all of the changes in the database will be successful or the database will be restored and its previous state.

Each table created in the database includes an _id column that is set as primary key and timestamp column. The timestamp is automatically added each time a new entry is made.

In addition, SQLite supports text, numeric, real, integer and blob types all which are supported by our system.

5.5.3. IPluginModel Interface

In Android, each service that provides binding capabilities needs to implement an interface that is returned to the caller to interact with the service. In this system, the communication takes place between two different processes. For this reason, a special type of interface, a Binder interface needs to be defined in an Android Interface Definition Language (AIDL) file.

IPluginModel interface contains the method signatures that are presented in figure 11.

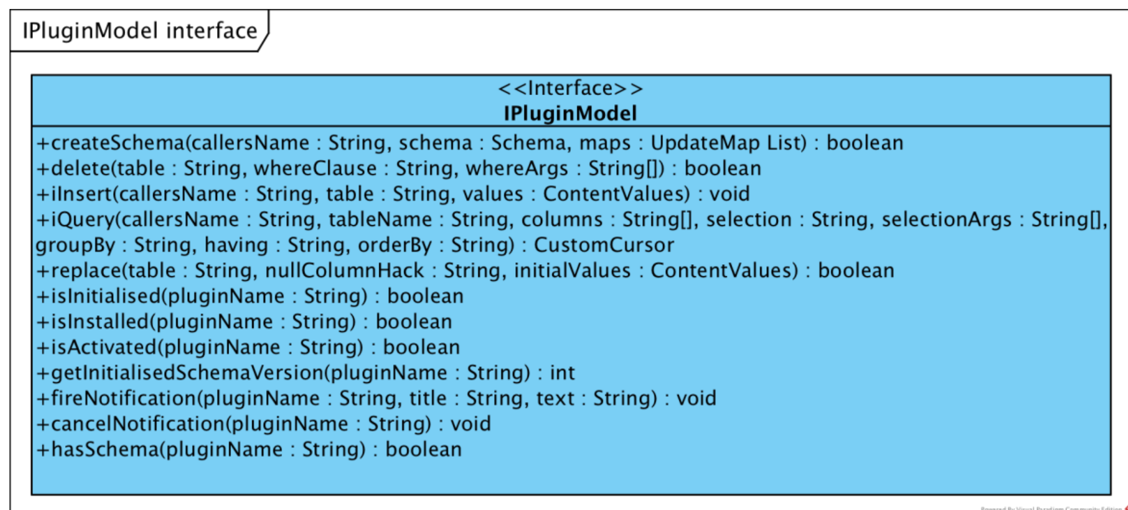

Figure 11: IPluginModel class diagram

5.5.4. Plugin application

5.5.4.1. Activity

The activity in this diagram is an abstract representation the main entry point for the plug-in and is used to exemplify the local binding to the client service. In real plug-ins

design, any of the Android components such as a Service or an Activity can request to bind to the service.

5.5.4.2. Service

The service in the plugin application works as the connecting point to the host application. This class extends the plugin model library's ClientService class and inherits its functionality. ClientService is designed as a generic component to establish connectivity between the host and plugin.

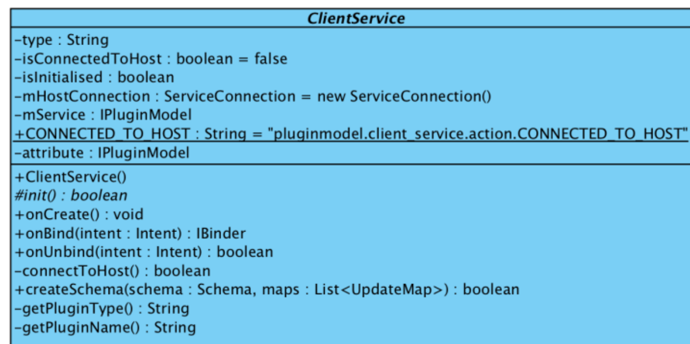

Figure 12: ClientService class diagram

The ClientService class has two purposes, the first is to connect to the HostService, and the second is to enable other classes to connect to client service. Depending on the type of the plugin, the client service provides access to different set of methods to communicate with the host. For example, a read only plug-in should not be able to create tables or modify any data in the database. For that reason, the service returns a different interface to the class that requests to bind on it, depending on the capabilities defined by the plugin type. Those interfaces are implemented by the client service and described in 5.5.4.3.

In addition, the service defines an abstract method, the init(), that should be implemented by the developer of the plug-in, in which the schema of the database should be defined. The service determines if the plug-in is of type read/write and if it is not initialised. If both conditions are met the init() is called. In case the type is "read only" the developer can leave the body of the method blank.

A detailed description of the connection task and the method calls is described in subchapter 5.5.7.

5.5.4.3. CRUD and CRUDRead interfaces

CRUD and CRUDRead are two interfaces that are used within the plug-in to establish communication between the classes that bind to the client service and the service. They are implemented by the library's ClientService class and serve the purpose to expose indirect access to host method calls. This works as a safety measure for preventing incorrect calls directly to the host application and ensuring that the plug-in is properly initialised or activated by the time the calls take place.

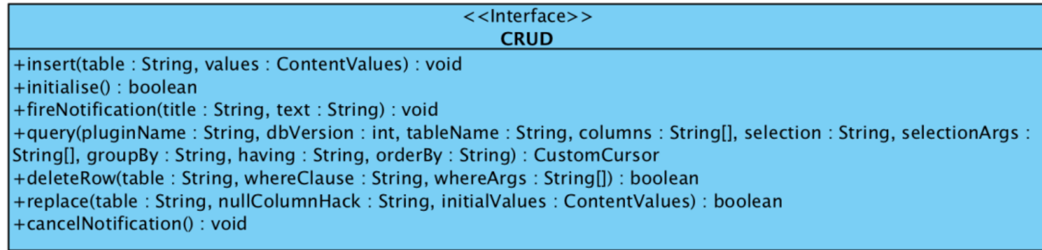

Figure 13: CRUD interface class diagram

CRUD interface defines methods for inserting, querying, and modifying data in the database, as well as broadcasting and cancelling notifications, as shown in figure 13.

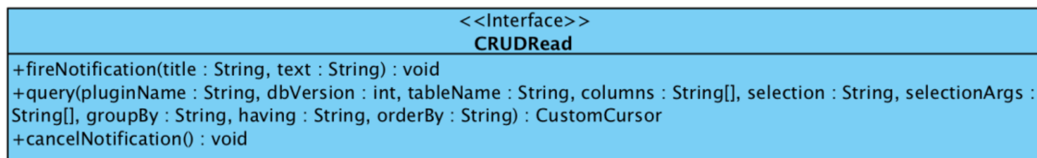

Figure 14: CRUDRead interface class diagram

CRUDRead interface defines only methods for querying data, broadcasting and cancelling notifications.

5.5.5. Other classes

The system uses some other important classes that are not included in figure 4. Those are:

5.5.5.1. Plugin

Each plug-in has characteristics that determine its behaviour and how the host application will handle each particular plug-in. Those characteristics extend from information regarding the versioning to others that define the status of the plug-in in the system.

The system groups those information in Plugin objects defined by the class shown in figure 15

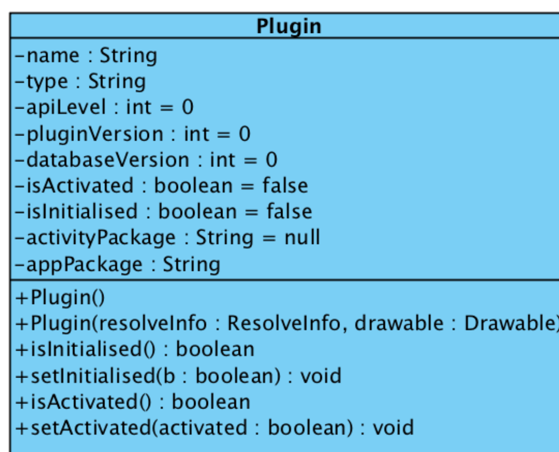

Figure 15: Plugin class diagram

5.5.5.2. Schema

Each plug-in that can write data to the database, needs to define the tables that the host application should create to hold it into. This is accomplished by using Schema objects.

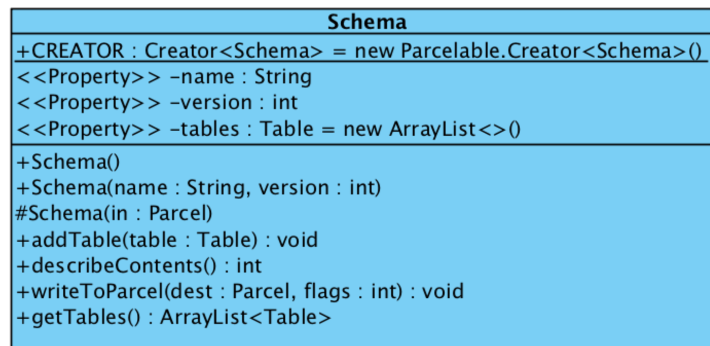

Figure 16: Schema class diagram

Those objects contain the name of the schema the version and an array list of Table objects.

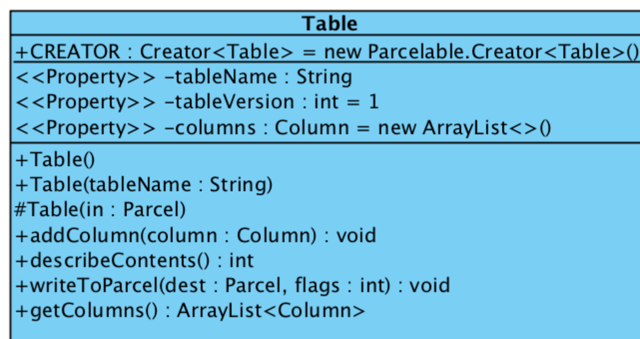

Figure 17: Table class diagram

Table objects contain the name of the table, the table version and an array list of Column objects.

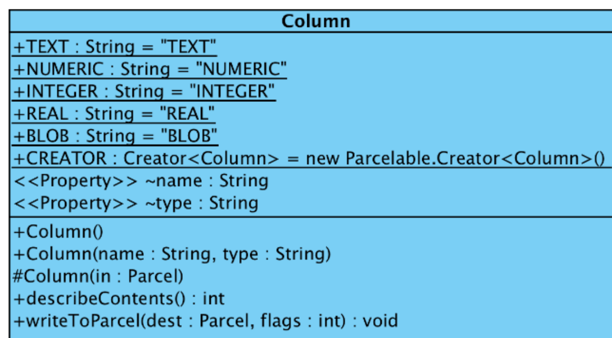

Figure 18: Column class diagram

The *Column* class contains information about the name of the column and the type. Predefined String constants are defined in order to help the developer choose acceptable types. Those constants include supported SQLite data types, text, numeric, integer, real and blob.

All three classes implement the Parcelable interface, a condition necessary in order to be able to be pass from one process to another.

In addition to creating the database tables, schema objects are used for describing the database's current status, since Registry stores those objects in the applications memory.

5.5.5.3. UpdateMap

One of the requirements of the system is to provide the ability to upgrade the structure of the tables for a plug-in if it uses a newer database version when the plug-in is updated. For this reason, the system provides an UpdateMap class that keeps information about the modifications that have to take place in order to update the plug-in tables that where created by a previous schema version.

Figure 19 describes this class.

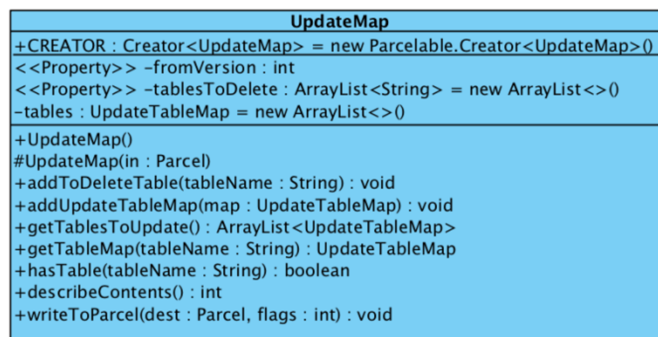

Figure 19: UpdateMap class diagram

Because the update task relies on the schema version that is created at the time of upgrade, the objects hold a fromVersion attribute to denote that this particular object describes the changes needed to be made from that current version. Effectively, the developer needs to create many different objects describing the upgrade from each version up to the newest. For example, if the latest version is 3, one object should describe the modifications from version 1 and another from version 2.

Additionally, the map contains two array lists, one with the tables names to delete and another with the tables that need modification. The latter are represented by UpdateTableMap objects and are described in figure 20.

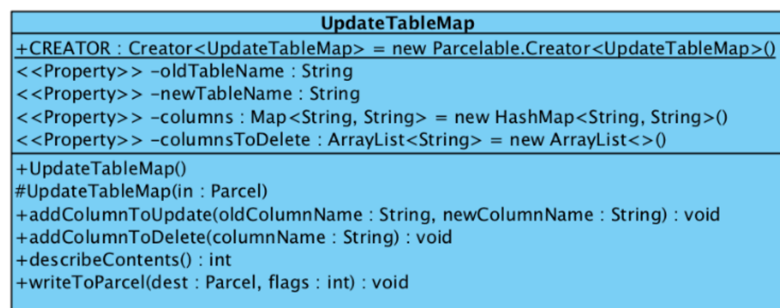

Figure 20: UpdateTableMap class diagram

As Schema object, those objects are passed between processes, therefore it is also necessary to implement Parcelable interface.

5.5.5.4. CustomCursor

SQLite databases in Android return data in cursor objects. Custom cursor supports integer, string, long, and Blob data types.

Since they cannot be transferred across processes, we implemented a CustomCursor class that implements Parcelable interface, to transfer the results returned by query() from host application to the plug-in.

Figure 21 describes the CustomCursor class.

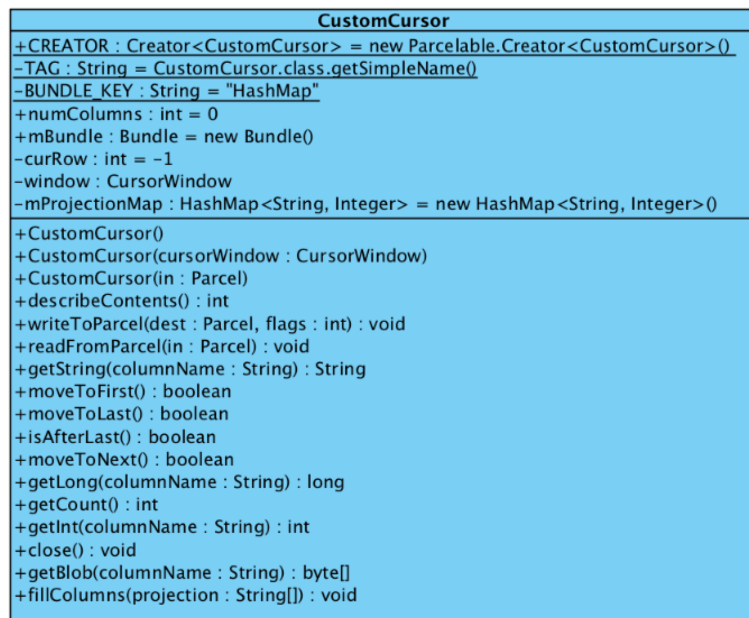

Figure 21: CustomCursor class diagram

5.5.6. Mechanism for detecting Plugins

The following activity diagram describes the activities and actions that take place during plug-in detection by the host application.

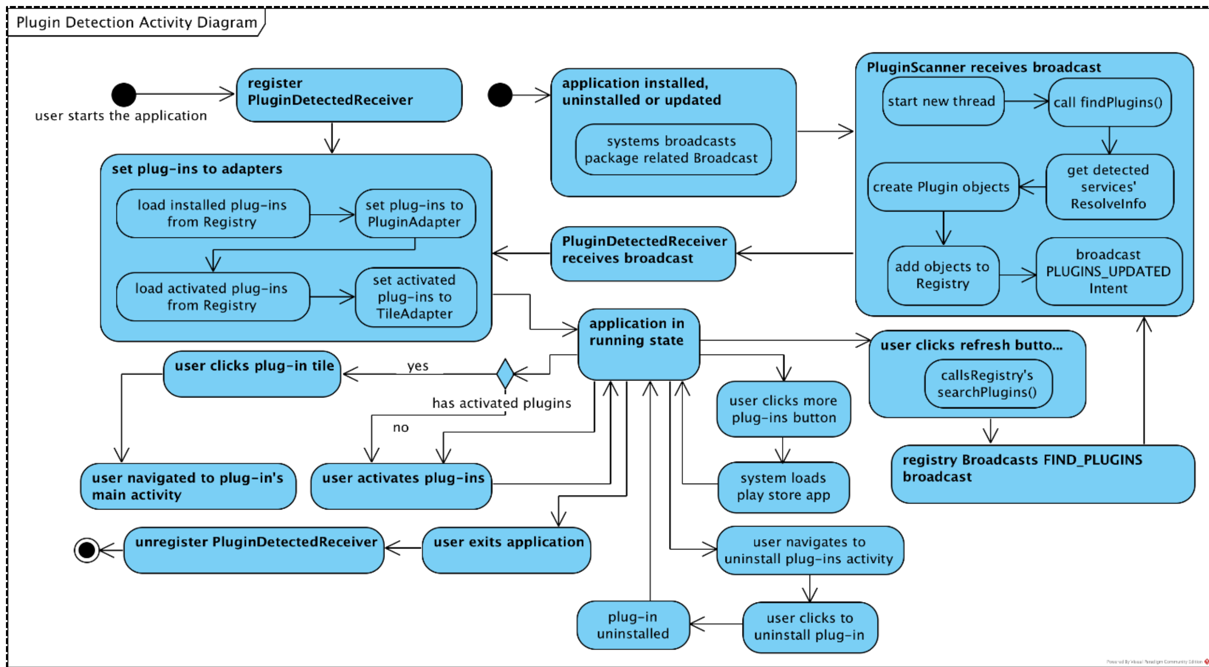

Figure 22: Plug-in detection activity diagram

When the user starts the application, the system loads the main activity and its lifecycle begins. The first call that takes place is the *onCreate()* method. In *onCreate()* the *PluginAdapter* and *TileAdapater* are initialised and are set with the plugin list and the activated plugins list from the Registry. The adapters transform the plugin objects to visual objects to be hosted in the adapter views. *PluginAdapter* provides switch objects that enables the application to enlist the available plug-ins to the user as well as *onCheckedChangedListener* to enable the user to activate or de-activate a plug-in. *TileAdapter* provides custom Tile objects that function as a button and *ItemClickListener* to start a plug-in's activity.

Then the *PluginDetectedReceiver* is initialised and registered. This receiver is notified by the system whenever a *PLUGINS_FOUND* broadcast is broadcasted. This broadcast is sent by the *PluginScanner* to notify the application that a new search for plugins took place and the application should refresh the User Interface.

Now the activity is in running state. If any plug-ins are activated the user can click on a plug-in tile to start a plug-in. The user can activate more than one plug-ins. Whenever this happens the activated state of the plugin in memory is updated and the corresponding tile appears on the screen. If the user chooses so, he/she can click on the refresh button to start a new plug-in detection. In this case, the *onClickListener* method of the refresh button calls the *searchPlugins()* in Registry. The *Registry* in turn sends a Broadcast that notifies the *PluginScanner* to scan for available plug-ins.

The *PluginScanner* receives the broadcast and runs the method *findPlugins()* in a new thread. This method queries with the PackageManager all the services in the device that declare "*android.intent.action.PLUGIN*". The scanner then gets the information from the services that are defined by the developer of the plug-in in the manifest and constructs Plug-in objects. After all objects are constructed they are passed to the *Registry* to save in the memory.

The Registry now contains the current updated plug-in list and the scanner sends a broadcast to notify the *PluginDetectedReceiver* that there was a new detection.

The *PluginDetectedReceiver* calls the method *setPluginAdapterContents()* that gets the new plugin and activated plugin lists from the Registry and sets it to the adapters.

The *PluginScanner* also receives broadcasts by the system whenever an application is installed upgraded or uninstalled. Anytime the receiver gets notified, either by the Registry broadcast that was mentioned previously or by the system, the same process of detecting plug-ins takes place. *PluginScanner* is set to receive broadcasts in the manifest file instead of being registered on run-time as the *PluginDetectedReceiver*, which means that the receiver is started even if the application is not running.

Finally, the user can click "more plugins" button and the system navigates to Play Store to download new plugins or navigate to the uninstall plug-ins activity to uninstall a plug-in.

5.5.7. Communication between Plugin and Host

In the system developed for the needs of the present thesis, the host and the plug-ins communicate using an android Binder interface. This interface provides the capability to the plug-in to conduct remote process method calls to the host to create database tables, insert and query data from host's database broadcast notifications and request information for the plug-in's state.

In order to connect to host, a plug-in needs to connect to the local client service. The local service then connects to the Host's service to be able to calls its methods. The HostService's binder interface is not exposed to the plug-in's developer directly but it is wrapped around a local interface in the client's service and the developer can only interact with the local methods. This design provides control over what functionality a plug-in is exposed to depending on its type by returning a different interface for a read-only and a read/write type of plugin. Furthermore, various checks take place to guaranty that each call is forwarded to the host's service only when the plug-in is properly initialised and activated, to eliminate system crashes.

1. The Activity diagram shown in figure 23 describes the binding to the host and table creation by the plug-in.

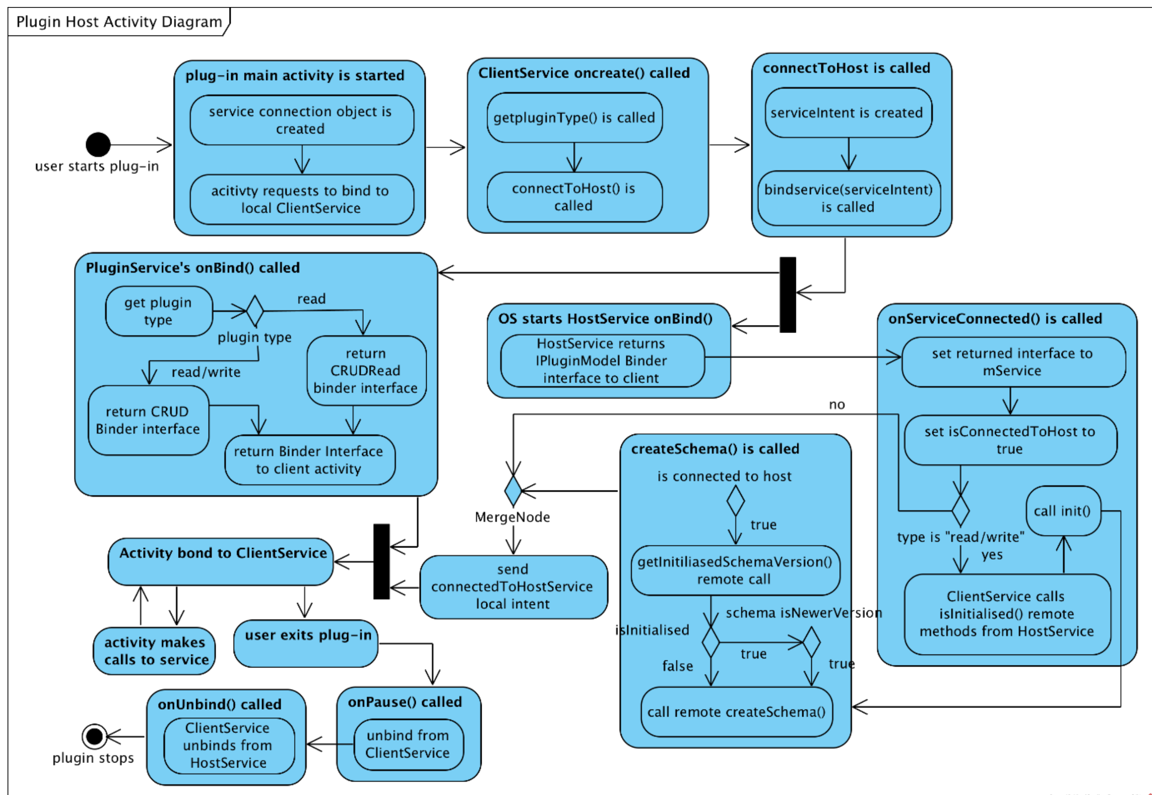

Figure 23: Host-client connection activity diagram

Although in this scenario the local binding takes place in the main activity, it is possible for any activity or service in the plug-in application to bind to the local service.

When the user starts a plug-in that main activity is loaded. In the onCreate() method the activity binds to the local service. The service's lifecycle begins and onCreate() is being called. In this method the plug-in type is determined and a connectToHost() is called, which is responsible to create and send an intent to the system to request to bind to the HostService. At this point the operating system is managing a handshake between the host and the client to establish the connection. This part is independent from the service's lifecycle and since there are no further instructions in onCreate() it terminates. Subsequently, the onBind() method is called to process a plug-in activity's binding request and return a local binder interface. Depending on the type of the plug-in, onBind() returns either a CRUD Binder object for a read/write or a CRUDRead Binder object for a read only plug-in. After onBind() is terminated the activity is bond to the local service.

In the meanwhile, the system calls the host service's respective onBind() method to get a remote interface to bind to the host. When the connection with the host's service is established the onServiceConnected() method is called, in which the returned interface is held in an attribute and the isConnected state is updated to "true".

Furthermore, if the plug-in is of type "read/write", the client service retrieves the "isInitialised" state of the plug-in from the host and the init() method is called. init() is implemented by the plug-in developer and in this method he/she should create the Schema object that defines the structure of the tables the plug-in needs to use and call the local createSchema(). Additionally to the Schema object, if the version of the plug-in's database is higher that one, an UpdateMap object is passed in createSchema() to

describe the modifications that are necessary to upgrade the tables of the plug-in. In local createSchema() the client service retrieves the Schema version that is initialised in the host and checks if the plug-in is initialised. If it is not, then the remote createSchema() is called in the host to conduct the creation of the tables. In addition, the createSchema() is called if the plug-in is already initialised and the current database version is higher that the version of the previously initialised plug-in. In createSchema() in the host, if the current version is higher the modifications that should take place to update the database tables are determined and methods upgradeTable() or createSchema() of database manager are called to create or upgrade the tables.

If the plug-in is not of type “read/write” the init() method is not called. Finally, the system broadcasts a “CONNECTED_TO_HOST” intent to notify the activity that the ClientService is connect to Hostservice.

The onServiceConnected() terminates and the client service continues at the state of being bond to the activity. At this state, the activity can conduct calls to interact with the host’s service. This process is explained in Figure 24.

When the user chooses to exit the plug-in the onPause() is called and it unbinds from the local service and the local service then unbinds from the host’s service.

2. Figure 24 describes in more detail the steps that take place when the plug-in is connected to host’s service and conducts remote calls. Those calls depend on the type of the plug-in. Both types can query the database, broadcast and cancel notifications. The read/write type can additionally insert into the database, update an entry and delete a row on a table.

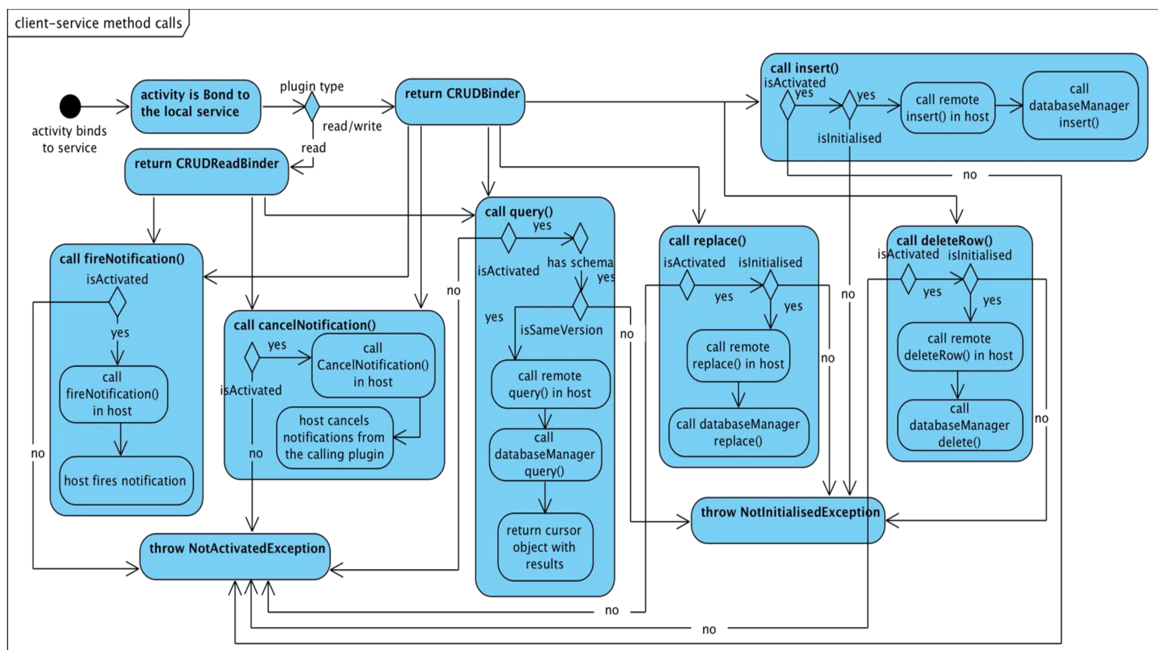

Figure 24: Client service remote calls activity diagram

In every call, a check is made to make sure the plug-in is activated by the user. If it is not activated then a “not activated” exception is thrown and the plug-in developer is responsible to catch this exception whenever he/she calls those methods. Additionally, the methods replace(), insert(), and deleteRow() can only be safely called if the plug-in is properly initialised. For that reason, it is checked whether it is initialised before the

remote call is propagated to the host's service. If it is not initialised a "NotInitialised" exception is thrown.

In contrast, query() method doesn't require the calling plug-in to be initialised. Instead, it is required to check if the table queried exists in the database, therefore, the query() method contains a parameter pluginName describing the plugin that created the table and is used from the system to check if that plugin's Schema exists in the database.

Additionally, the Schema version passed in query() is used to check the version of Schema that is initialised. If the Schema version is different than the version expected by the plug-in developer, a "NotInitialised" exception is thrown.

5.6. Libraries

One the most important goals of this project is the creation of APIs that composes the different parts of the plugin system and enable developers to produce extensions for it. To accomplish that we organised the components of the system in three libraries, the core, the host and the client library.

Android, apart from the Java libraries (JAR), defines an Android-specific type of library saved in "Android Archive" files (AAR). In addition to the java classes, this type of file contains resource files such as XML layout files, drawables and others. In our system, the generated libraries are AAR libraries.

This subchapter demonstrates the structure of the libraries of the system.

5.6.1. Plugin model host library

This library is composed by classes that provide the Host application with capabilities to manage and visually represent plug-ins to the user and provide communication slots for the plug-ins to connect on, as well as database storage.
The library package is presented in figure 25.

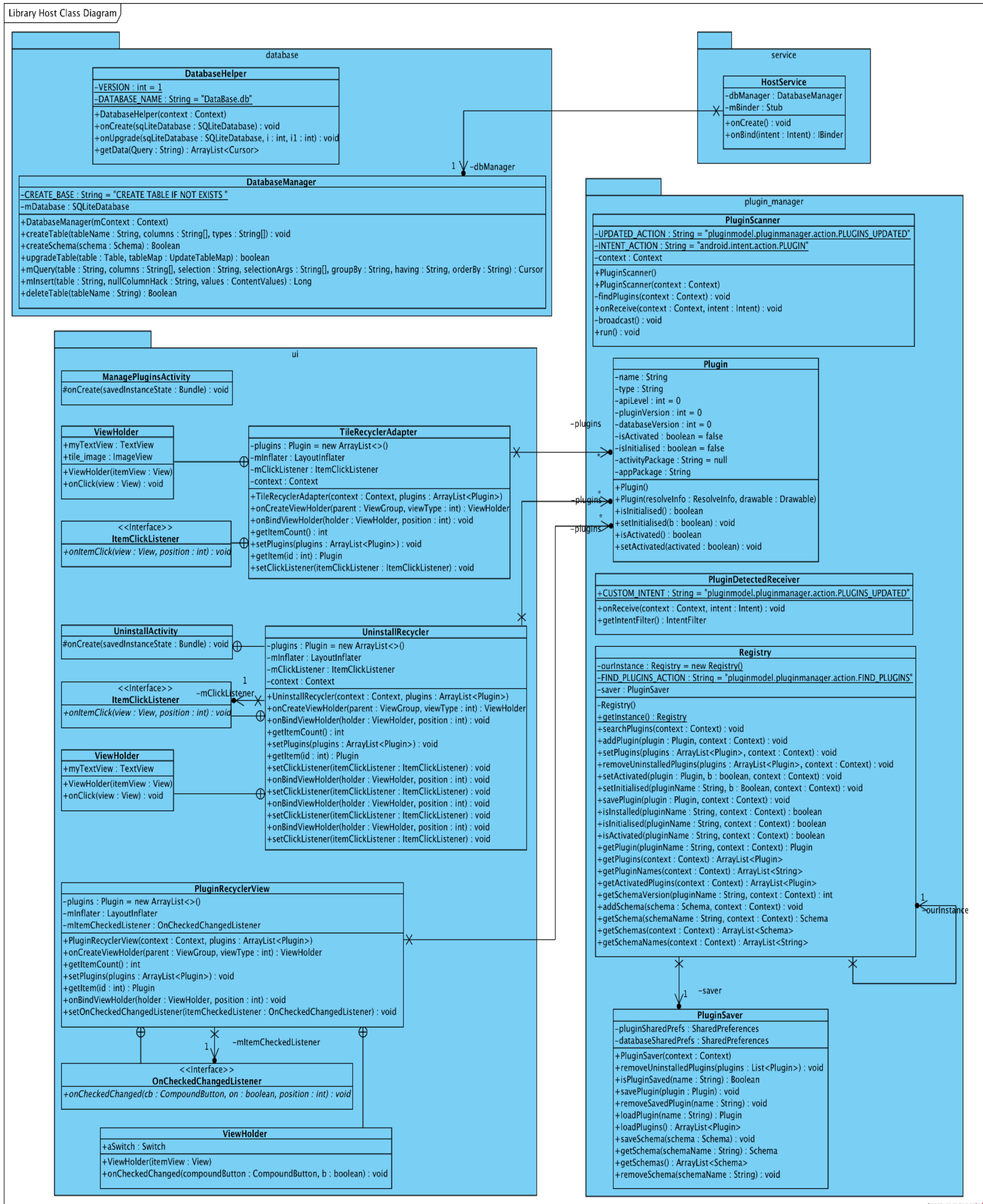

Figure 25: Host library class diagram

The library is divided in four sub-packages grouping the classes according to the category of the functionality they provide to the system.

UI package contains recycler view classes that are related to the visual representation of the plug-ins. Database package consisting of the database manager and database helper,

used to create and interact with the database. Plugin manager package contains the classes used for detecting and storing the plug-ins. Finally, the service package contains the HostService class.

In addition to java classes, the library includes xml files that define the layouts for components such as the switch and the plugin tile, described in 5.7.

5.6.2. Plugin model client library

The client library is composed by the ClientService class and two interfaces, CRUD and CRUDread. The binder classes included in figure 26 represent inner classes of service that implement the interfaces.

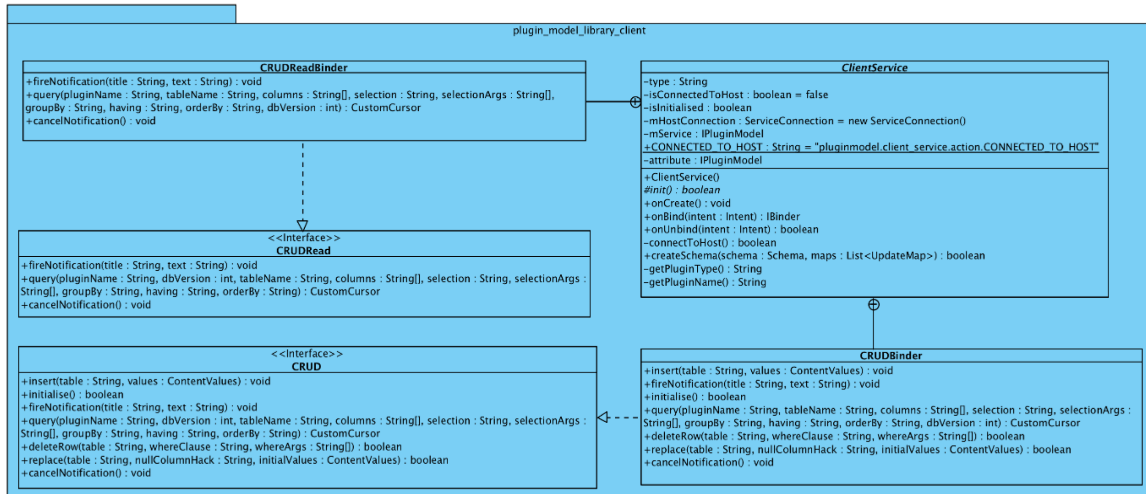

Figure 26: Client library class diagram

5.6.3. Plugin model core library

The core library contains the common classes that host and client library depend on. In Android, in order for binder interfaces to be compiled and function properly the AIDL files that define it should be strictly created in identical packages. Therefore, those files are included in the core library and imported by the other two libraries.

In addition, it provides three types of exceptions described in 5.5.7.2, thrown by the client service.

The class diagram in figure 27 describes the classes contained in the core library.

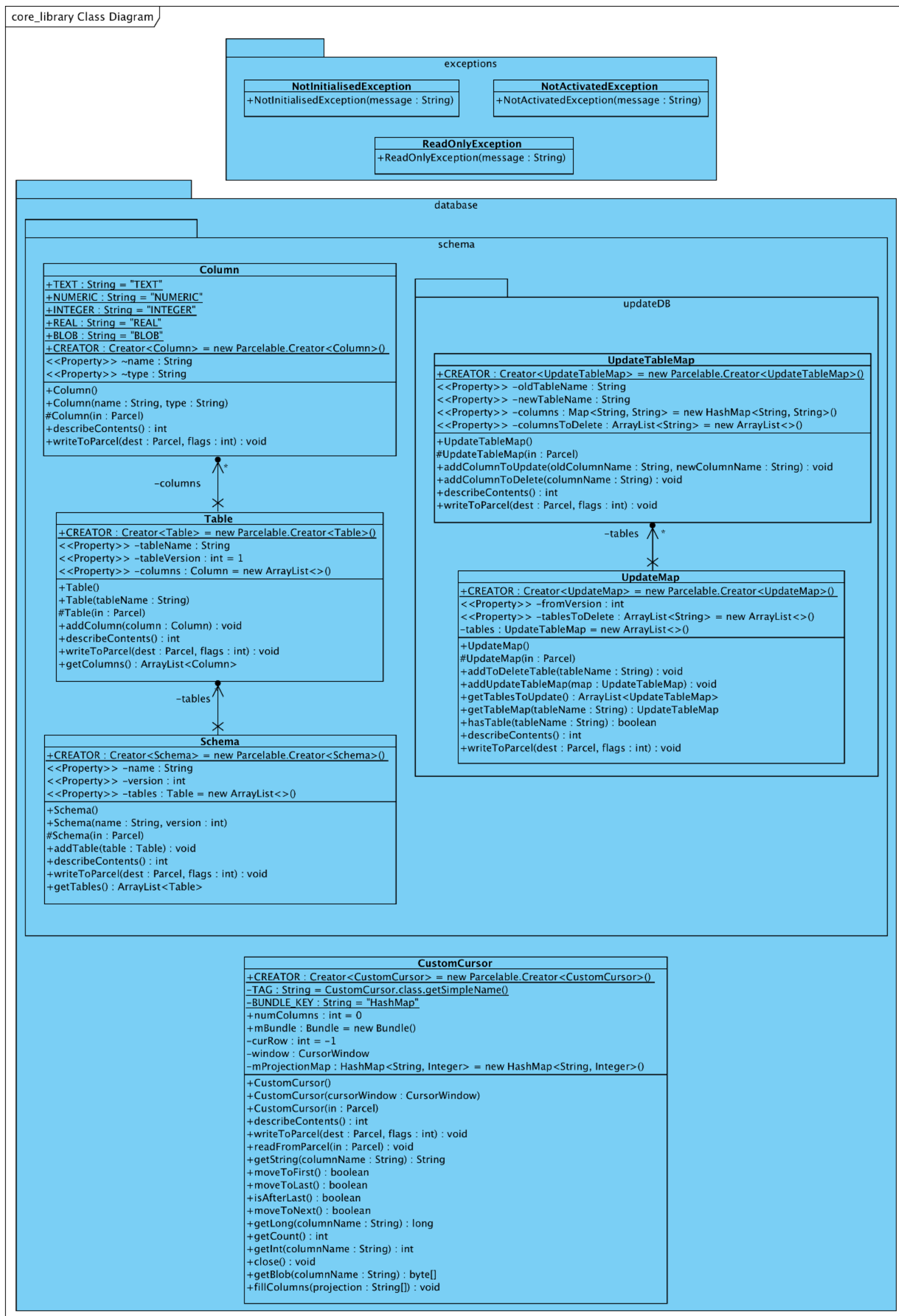

Figure 27: Core library class diagram

5.7. User interface design

Chapters 4 and 5 among others, define requirements related to the interaction between the user and the application. The user should be provided with mechanisms to install and uninstall plug-ins, as well as with a UI that visually presents, enables activation or de-activation and navigation to the installed plug-ins.

Those requirements are addressed by the UI design which is described in this subchapter.

5.7.1. Activity UI

The overall design of the application's activity is composed by parts merged together in one UI.

The activity includes a toolbar on the top of the screen that holds the settings menu. Additionally, on the main part of the UI the TileRecyclerView takes the rest of the space, presenting the tiles of the activated plug-ins. Furthermore, a floating button with a "plus" icon in the bottom-right corner can be clicked and reveal the container that holds switches that represent the installed plug-ins.

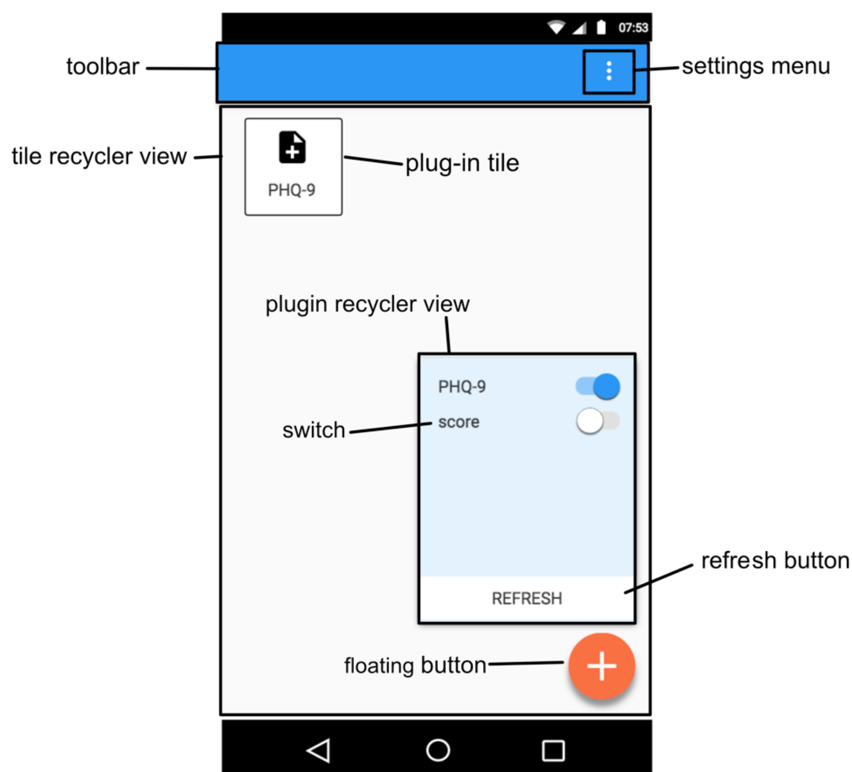

Figure 28: Host application activity mock-up

5.7.2. Installing and Un-Installing Plugins UI

The user can access the install and un–install plug-ins functionality by clicking on the three-dot settings icon on the application's toolbar. When the user clicks on the icon the settings menu appears and displays two buttons to the user.

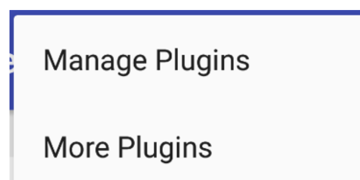

Figure 29: Settings menu screenshot

By clicking the “More Plugins” button the user is navigated to Google Play Store to download plug-ins. Alternatively the user can click on the “Manage Plugins” button which starts an activity that works as a menu with options to manage the plug-ins. In the current version of the application, the user is provided with the option to uninstall plug-ins. A screen-shot of the manage plug-ins menu is shown in the picture bellow.

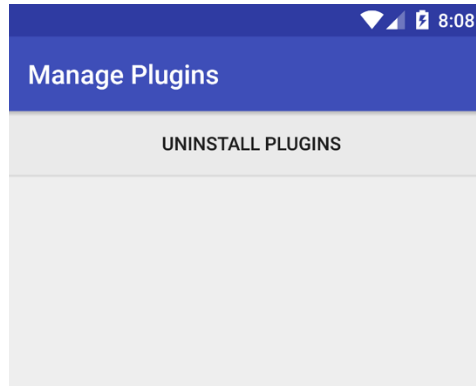

Figure 30: Screenshot of the Manage plugins activity

The user can then click the “UNINSTALL PLUGINS” option and be navigated to another activity that lists all the detected plug-ins and if he/she clicks on one of items on the list the application sends an intent to the system to uninstall the relevant package from the device.

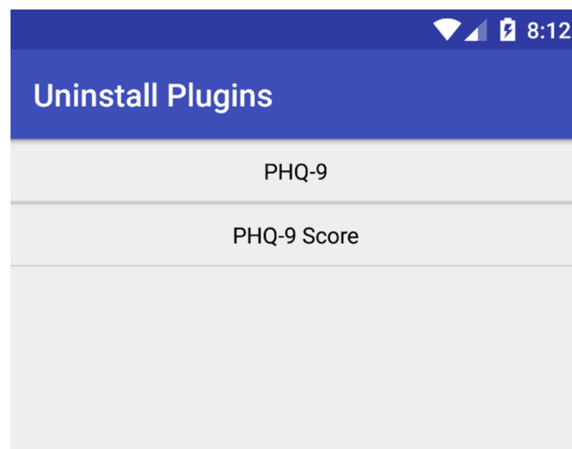

Figure 31: Screenshot of the Uninstall plug-ins activity

The user is asked by the system if he/she wishes to uninstall the particular application and can choose to continue or cancel the task.

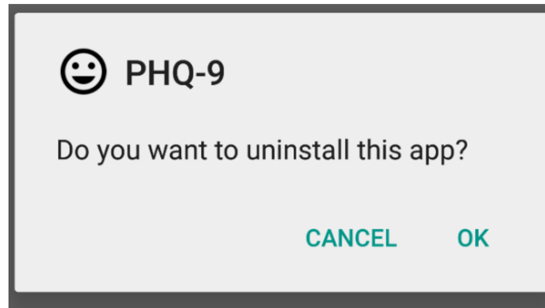

Figure 32: Screenshot of the Uninstall application dialog box

Due to the undetermined number of the installed plug-ins on the system, the “Uninstall Plugins” activity is dynamic. To implement a dynamic list of UI components in Android we have to make use of the AdapterView which is a class that creates UI objects from a list of java objects. In this case it maps Plugin objects to TextView objects and attaches onClick listeners to respond to clicks.

5.7.3. Presenting plugins to the user

Another requirement is to present the detected plugins to the user and provide the interaction components for activation or deactivation of the plug-ins.

For this task, we designed a UI component that makes use of a switch that has two parts, the name of the switch that holds the plugin’s name and a switch part that responds to clicks, to change the activation status of the Plugin object that is stored in the application’s memory.

Figure 33: Switch wireframe

All the switches are contained on an RecyclerView that is implemented by PluginRecyclerView class. Similarly to the “Uninstall Plugins” activity the recycler class handles the mapping between Plugin objects and the switch components to dynamically list the detected plug-ins and respond to clicks.

In addition, the container that holds the switches contains an extra “Refresh” button that enables the user to re-detect the plugins on-demand.

Every time the user clicks on a switch the state of the switch changes and system’s registry is notified to change the activated state of the current plug-in. In addition, the application creates a tile that the user can click on to navigate to the plug-in’s main activity.

5.7.4. Plugin Tiles

Each activated plug-in is visually represented by a Tile that contains the plug-in application’s icon and a TextView that displays the name of the plug-in, as shown in the wireframe bellow.

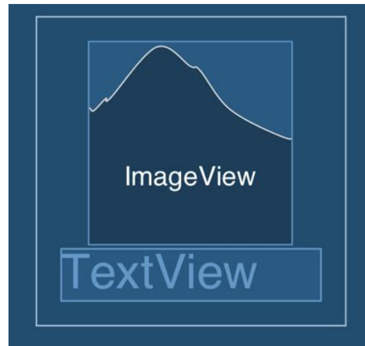

Figure 34: TileItem wireframe

This tile is contained in a custom *RecyclerView* class, the *TileRecyclerView*, that works as the rest of the recycler views described previously.

Additionally, the click listener of the tiles, when clicked, generates an intent using the application package that is stored in the Plugin object and sends that intent to the system to start the main activity of the plug-in.

6. Development Tools

The design and implementation of the project was made using multiple software tools.

a) Android Studio IDE

Android Studio[2] is an Integrated Development Environment (IDE) developed by Google and is dedicated to Android development. It is built on Jetbrain's IntelliJ IDEA, a popular Java IDE and is currently the official Android Development IDE.
Among the many features provided by Android Studio are:

- Android specific refactoring.
- Android Virtual Device Emulator used for debugging in the IDE.
- Layout editor that enables the developers to easily design user interfaces by drag and drop.
- Lint tools to evaluate performance, usability, version compatibility of the applications.
- Application signing capabilities and others.

The version of Android Studio used in this project is 2.3.3.

b) Visual Paradigm

The UML class diagrams and activity diagrams used in this project where designed in Visual Paradigm Community Edition[3] version 14.2. Visual paradigm is a UML design tool that supports UML 2. It can be used to design class, activity, use case, state machine and many other types of diagrams and supports functionalities such as report and code generation and reverse engineering. The community edition is a free non-commercial edition and supports all 13 UML diagrams types.

c) Moqups

The design of the application prototypes was made using Moqups[4], which is an online cloud application dedicated for mock-up and wireframe design.

d) Genymotion Android Emulator

Genymotion[5] is an Android emulator used by many developers for application testing. It supports a very big number of devices and configurations.

[2] https://developer.android.com/studio/index.html
[3] https://www.visual-paradigm.com/download/community.jsp
[4] https://moqups.com/
[5] https://www.genymotion.com/

7. Example plugins

For demonstration purposes, two plug-ins related to health monitoring designed and implemented. For the first plug-in, the Patient Health Questionnaire 9 (PHQ-9) was implemented. The second plug-in is designed to present the data that are produced by PHQ-9 plug-in. These two examples represent the two different types of plug-ins, the read and read/write type.

PHQ-9 [35] is a self-administered questionnaire that can effectively diagnose the presence and severity of depression. It is a 9-question subset of the PHQ questionnaire which was developed in 1999 by Dr. Robert J. Spitzer, Dr. Janet B.W. Williams, Dr. Kurt Kroenke with an educational grand from Pfizer Inc.[6] and consists of questions the patient's everyday experience in the past 2 weeks.

The patient can answer the questions by "not at all", "several days", "more than half of the days" and "nearly every day", and each question adds from 0 up to 3 points to the final score. According to the score the patient's depression is classified in one of the following severity levels.

Table 1: PHQ-9 Scores and Proposed Treatment Actions [37]

PHQ-9 Score	Depression Severity	Proposed Treatment Actions
0 – 4	None-minimal	None
5 – 9	Mild	Watchful waiting; repeat PHQ-9 at follow-up
10 – 14	Moderate	Treatment plan, considering counselling, follow-up and/or pharmacotherapy
15 – 19	Moderately Severe	Active treatment with pharmacotherapy and/or psychotherapy
20 – 27	Severe	Immediate initiation of pharmacotherapy and, if severe impairment or poor response to therapy, expedited referral to a mental health specialist for psychotherapy and/or collaborative management

In addition to the score, the questionnaire can assess if the patient meets the criteria of "Major depressive syndrome" by evaluating the number of answers with a score more than two. The detailed scoring methodology is described in the PHQ instructions document [37].

7.1. PHQ9 plugin (read/write type)

The plug-in has two screens, one for the questionnaire and another for the presentation of the score.

a) The questionnaire screen is a scrollable activity that presents an exact copy of the PHQ-9 according to [35]. Each question is followed by radio buttons for the user to answer the question. At the bottom of the activity there is a submit button that can be clicked when the user answers all the questions. If one of the questions is not answered the activity does not proceed to calculate the score and displays a message to the user asking to answer all questions.

[6] http://www.pfizer.com/

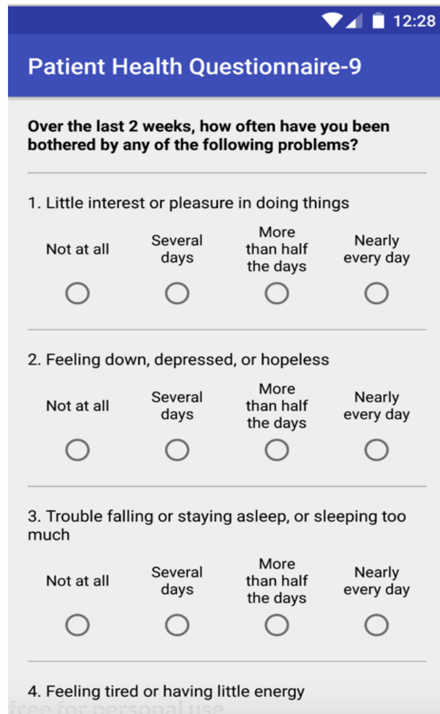

Figure 35: PHQ-9 plug-in's questionnaire screenshot 1

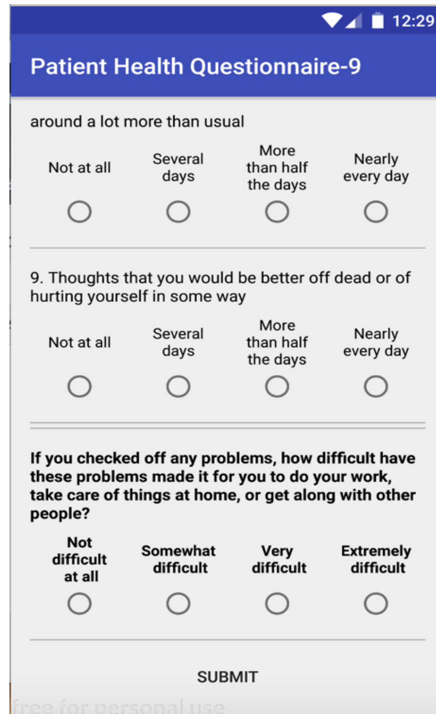

Figure 36: PHQ-9 plug-in's questionnaire screenshot 2

When the submit button is clicked the plug-in calculates the depression severity score and if the depression is major or other type. Subsequently it calls the insert() method to save the data to the host's database and sends the results to the score activity to be displayed to the user.

b) The score screen includes a text view on the top of the activity that presents the type of depression according to the assessment. Additionally, it includes a progress bar that displays the score as a proportion of the maximum score and a text view bellow the progress bar with the score number.

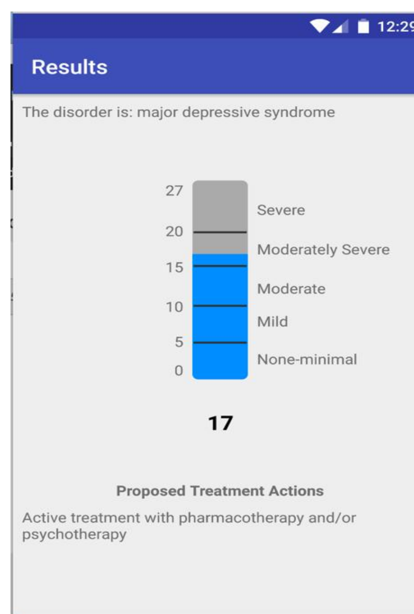

Figure 37: PHQ-9 plug-in's results screenshot

Finally, at the bottom of the screen there is a text view that contains a text with the proposed treatment action that corresponds to the proposed treatment mentioned in the PHQ-9 instructions document.

The part of the code that connects to the host application is the following:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
...
   connection = new ServiceConnection() {
      @Override
      public void onServiceConnected(ComponentName name, IBinder service) {
         localBinder = (CRUD) service;
         isConnected = true;
      }

      @Override
      public void onServiceDisconnected(ComponentName name) {
         localBinder = null;
         isConnected = false;
      }
   };

   final Intent clientServiceIntent = new
            Intent().setClass(getApplicationContext(), PHQ9Service.class);
   bindService(clientServiceIntent, connection, BIND_AUTO_CREATE);
...
```

The following code snippet describes storing the results to the host's database.

```
Button submit = (Button) findViewById(R.id.submit_button);
submit.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
...
ContentValues contentValues = new ContentValues();
contentValues.put("score", score);
contentValues.put("syndrome", syndromeType);
try {
   localBinder.insert("phq_9_results", contentValues);
} catch (NotActivatedException e) {
   e.printStackTrace();
} catch (NotInitialisedException e) {
   e.printStackTrace();
} . . .
```

7.2. Score plugin (read type)

The Score plug-in is created in order to demonstrate the type of plug-in that can only read data stored in the host application by another plug-in. It presents the results of the PHQ-9 questionnaire in a visually understandable way for the user to evaluate his/her progress.

It contains one screen that is composed by two different parts. The first part is a tab layout that holds two charts, one line and one bar chart in two different tabs. The line chart contains the score values on the y axis and the timestamp to the x axis. The bar chart contains a bar that show if the depression detected was major, other or none. The second part of the screen contains a list of all the database entries.

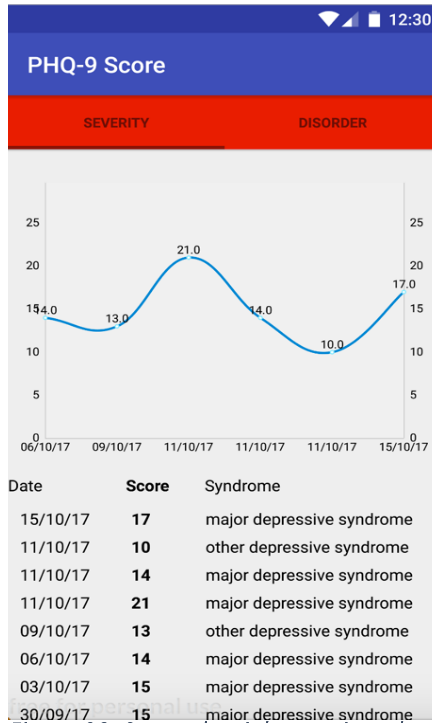

Figure 38: Score plug-in's severity tab screenshot

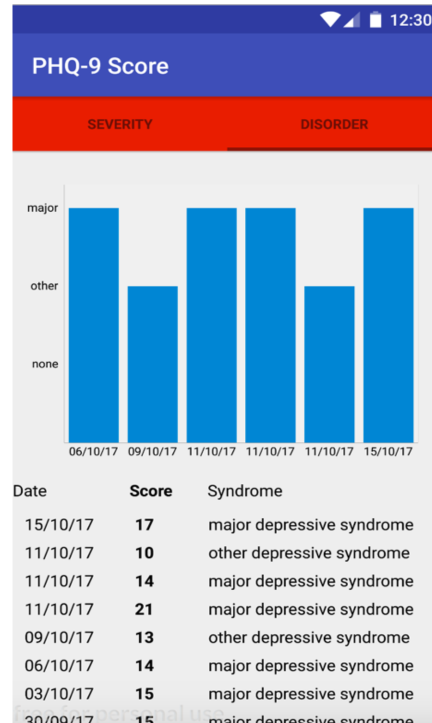

Figure 39: Score plug-in's disorder tab screenshot

Both charts and the list are scrollable so that the user can access all data even if they do not fit in the views. The chart where created by using MPAndroidChart[7] a custom library that enables developers to create line, bar and other types of charts in Android applications.

[7] https://github.com/PhilJay/MPAndroidChart

8. Comparison with system requirements

This chapter aims to compare the system requirement as described in chapter 3 and the implemented system's features in an attempt to evaluate which of originally planned and needed requirements are actually fulfilled.

8.1. Functional requirements

This subchapter describes how the implementation of the system addresses the functional requirements.

a. The system extends the host application's functionality on runtime because the plug-in can be installed after the installation of the Host application.

b. The system allows the user to download plug-ins via Google Play.

c. The host application is able to detect installed plug-ins with PluginScanner class and register them in the Registry.

d. The system resolves version compatibility issues during plug-in detection, by ignoring plugins that use API level different the host application.

e. Android provides the ability to define metadata for an Android component in the manifest file. In our system, it is necessary that the plug-in developer provides metadata about the plug-in's name, type, version, database version and the intent that starts the activity in the manifest file in the format demonstrated in the following code snippet.

```
<service
  android:name=".ScoreService"
  android:enabled="true"
  android:exported="true">
  <intent-filter>
    <action android:name="android.intent.action.PLUGIN" />
  </intent-filter>

  <meta-data
    android:name="plugin_name"
    android:value="@string/app_name" />
  <meta-data
    android:name="type"
    android:value="@string/plugin_type_read" />
  <meta-data
    android:name="activity_intent"
    android:value="android.intent.action.phq_9score.MAIN" />
  <meta-data
    android:name="api_level"
    android:value="1" />
  <meta-data
    android:name="version"
    android:value="1" />
  <meta-data
    android:name="dbVersion"
    android:value="0" />
</service>
```

f. The system provides communication mechanisms between the Host and the Plug-ins using services and binder interfaces.
g. The Host retrieves plug-in metadata in PluginScanner class and stores them in memory by constructing Plugin objects that are saved by the Registry class.
h. The initialisation of a plug-in happens in by the plug-in service instead of the Host. This simplifies the inter-process communication interfaces while maintaining the reliability of the system.
i. The Host Application when installed does not have any predefined data-base. The database tables are created upon plug-in's initialisation process.
j. See 8.1.i
k. The Host App is able to create the proper data structures plug-ins, on run-time by enabling a plug-in to call the createSchema() method that is implemented by the HostService.
l. The IPluginModel interface provides the insert() method that enables a read/write type plug-in to store data in the tables it created in the host application's database during the initialisations process.
m. Additionally to the 8.1.l, the IPluginModel interface provides the query() method to be able to retrieve data stored in the host application's database.
n. The system does not restrict the query() method from providing data to plug-ins that are stored by other plug-ins.
o. The system provides the upgradeTable() method in the DatabaseManager class that handle modifications on database tables. The task of modification is automatically handled by the HostService according to the database version of the plug-in. In addition, the plug-in developer is provided with UpdateMap classes to define any changes to the plug-ins schema during the upgrade.
p. The ClientService provides CRUD and CRUDRead interfaces to the plug-in, which its developer can use to interact with the applications database.
q. The Host library defines classes that assist the design of UI components in the Host application.
r. The Host application presents the plug-ins to the user in a list of switches. Those switches serve the purpose of enabling the user to activate and de-activate a plug-in. For more details about the design of the switches refer to 5.7.3.
s. The Host application places a clickable plug-in tile in the main layout of the application that navigates the user to the plug-in's main activity when clicked. Plug-in tiles are explained in more detail in 5.7.4.
t. User's privacy is achieved by storing his/her data to the applications private memory

and securing the communication between the Host and the plug-ins with signature level permissions.

u. User's data are stored in an SQLite database which is saved in the application's private memory.

8.2. Non-functional requirements

This subchapter describes how the implementation of the system addresses the non-functional requirements.

1. See 8.1.t.
2. See 8.1.t.
3. User's data is stored in an SQLite database.
4. Along with the libraries and the host application we delivered a Javadoc and a developer's guide.
5. The system was designed taking into account the maintenance requirement, therefore, the classes are designed to be independent whenever possible.
6. The relatively small scale of this project enables alternative testing methods. Aiming to reduce implementation time this system was tested by ad hoc methods.
7. The database is managed by the DatabaseManager class which is independent of the components that interact with it. A storage migration can be accomplished by changing the functionality of this class without major changes in the interacting components.
8. The PluginScanner class executes the plugin detection operation on a new thread and does not block the main thread of the application.
9. Since this is the first version of the system the backwards compatibility implementation is not relevant. Although the flexible design of the database tries to capture future requirements.

9. Conclusions and future work

Activity and health monitoring is a key factor for improving global health and reducing social and economic impact on the society attracting the attention of a high number of companies and researchers.

This thesis investigated solutions to the problem of health monitoring and management by taking advantage of Android OS mechanisms in order to provide a flexible, fully extensible application framework. The Android's inter-process communication mechanisms provided effective communication between the plug-ins and the host application. The system was designed in accordance with the requirements that were defined in this study, and the Android's features provided sufficient mechanisms for the implementation, without any need for modifications to the requirements. The plug-ins communicate with the application without any significant impact on the performance of the plug-ins. In addition to the application and the APIs, a Javadoc and a developer's guide were delivered to provide documentation to the developers that will use the system.

A major challenge in the design of the system was the data storage. Although the SQLite database provided all the necessary data storage functionalities, the system was designed to not have a predefined database structure. In theory, this would eliminate limitations regarding the data structure that can be saved in the database. Inevitably, this causes a number of problems. Since there is no predefined structure in the database the developer of a plug-in needs to know exactly what types of data are currently stored in the database by other plug-ins and the exact structure of the tables, in order to be able to access them. A limitation like this could be manageable in a system where all plug-ins are designed by one programming team but can cause problems to third-party developers.

An important feature of most fitness applications is data visualisation. Due to the lack of predefined data structures in the system, the host application in unaware of the data that is stored in the database, thus it cannot visualise them. Instead, the visualisations are handled by plug-ins.

To address the problems that accrue from the database abstraction, in addition to the custom data structures, it would be beneficial to provide data standardisation for the most common activities.

The example plug-ins demonstrated that the system works as expected. Due to the limited number of example plug-ins the system is not tested thoroughly, therefore, more plug-ins should be created in order to further test that the system is useful for more advanced scenarios. Additionally, it can be benefitial to determine the robustness of the system in scenarios that exceed the normal operation (stress testing), for example by conducting simultaneous method calls from a very high number of plug-ins or simulating high frequency interaction with the database. Although, no significant impact on the performance of the application and no delays during the interaction with the database were observed if a stress test detects such performance issues, it could be considered to implement the interaction with the database through separate threads. The database in our system can support multiple threads since it is thread safe as it is accessed by one database object it.

References:

[1] World Health Organization, "WHO | Physical activity," *World Health Organization*, 2017. [Online]. Available: http://www.who.int/mediacentre/factsheets/fs385/en/. [Accessed: 24-Sep-2017].

[2] WHO, "WHO | Physical Inactivity: A Global Public Health Problem," *WHO*, 2014. [Online]. Available: http://www.who.int/dietphysicalactivity/factsheet_inactivity/en/. [Accessed: 03-Nov-2016].

[3] CDC, "Overcoming Barriers to Physical Activity," *Centers for Disease Control and Prevention*, 2011. [Online]. Available: http://www.cdc.gov/physicalactivity/basics/adding-pa/barriers.html. [Accessed: 28-Nov-2016].

[4] "• Number of smartphone users worldwide 2014-2020 | Statista." [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. [Accessed: 08-Aug-2017].

[5] Statista, "• Leading Android app categories worldwide 2017 | Statistic." [Online]. Available: https://www.statista.com/statistics/200855/favourite-smartphone-app-categories-by-share-of-smartphone-users/. [Accessed: 09-Aug-2017].

[6] J. P. Higgins, "Smartphone Applications for Patients' Health and Fitness," *Am. J. Med.*, vol. 129, no. 1, pp. 11–19, 2016.

[7] L. G. Glynn, P. S. Hayes, M. Casey, F. Glynn, A. Alvarez-Iglesias, J. Newell, G. ??laighin, D. Heaney, M. O'Donnell, and A. W. Murphy, "Effectiveness of a smartphone application to promote physical activity in primary care: The SMART MOVE randomised controlled trial," *Br. J. Gen. Pract.*, vol. 64, no. 624, p. 871, Jan. 2014.

[8] "Platform Overview | Google Fit | Google Developers." [Online]. Available: https://developers.google.com/fit/overview. [Accessed: 09-Aug-2017].

[9] Samsung, "Samsung Health | SAMSUNG Developers." [Online]. Available: http://developer.samsung.com/health. [Accessed: 03-Oct-2017].

[10] D. Birsan and Dorian, "On plug-ins and extensible architectures," *Queue*, vol. 3, no. 2, p. 40, Mar. 2005.

[11] R. Wolfinger and D. Dhungana, "A component plug-in architecture for the. NET platform," *Modul. Program. …*, pp. 1–20, 2006.

[12] M. Murphy, *The Busy Coder's Guide to Android Development version 8.0*. CommonsWare, LLC, 2016.

[13] "Notes on the Eclipse Plug-in Architecture." [Online]. Available: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html. [Accessed: 10-Aug-2017].

[14] "Architecture – OSGi™ Alliance." [Online]. Available: https://www.osgi.org/developer/architecture/. [Accessed: 10-Aug-2017].

[15] Wikipedia, "Android (operating system)," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Android_(operating_system). [Accessed: 05-Apr-2017].

[16] Google, "Platform Architecture | Android Developers," 2017. [Online]. Available: https://developer.android.com/guide/platform/index.html. [Accessed: 16-Sep-2017].

[17] Wikipedia, "Android Runtime," *Wikipedia, the free encyclopedia*. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Android_Runtime&oldid=681316060. [Accessed: 05-May-2017].

[18] Google, "Processes and Threads | Android Developers." [Online]. Available:

https://developer.android.com/guide/components/processes-and-threads.html. [Accessed: 22-Aug-2017].
[19] A. Developers, “App Manifest | Android Developers,” 2015. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro.html. [Accessed: 18-May-2017].
[20] Google, “Intents and Intent Filters | Android Developers,” *https://developer.android.com/guide/components/intents-filters.html*. [Online]. Available: https://developer.android.com/guide/components/intents-filters.html. [Accessed: 20-Sep-2017].
[21] Thorsten Schreiber, “Android Binder Android Interprocess Communication,” Ruhr-Universitat Bochum, 2011.
[22] Google, “Bound Services | Android Developers.” [Online]. Available: https://developer.android.com/guide/components/bound-services.html. [Accessed: 10-Aug-2017].
[23] Google, “Application Fundamentals - Android Developers,” 2013. [Online]. Available: https://developer.android.com/guide/components/fundamentals.html. [Accessed: 20-Apr-2017].
[24] Android and Google, “Introduction to Activities | Android Developers,” 2015. [Online]. Available: https://developer.android.com/guide/components/activities/intro-activities.html. [Accessed: 17-May-2017].
[25] Google, “The Activity Lifecycle | Android Developers,” 2017. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle.html. [Accessed: 16-Oct-2017].
[26] Google, “Services | Android Developers.” [Online]. Available: https://developer.android.com/guide/components/services.html. [Accessed: 22-Aug-2017].
[27] Google, “Content Providers | Android Developers.” [Online]. Available: https://developer.android.com/guide/topics/providers/content-providers.html. [Accessed: 15-Sep-2017].
[28] Google, “Broadcasts | Android Developers.” [Online]. Available: https://developer.android.com/guide/components/broadcasts.html. [Accessed: 03-Oct-2017].
[29] Google, “Storage Options | Android Developers.” [Online]. Available: https://developer.android.com/guide/topics/data/data-storage.html. [Accessed: 17-Sep-2017].
[30] Google, “PackageManager | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/content/pm/PackageManager.html. [Accessed: 18-Sep-2017].
[31] Google, “System and kernel security | Android Open Source Project.” [Online]. Available: https://source.android.com/security/overview/kernel-security. [Accessed: 10-Oct-2017].
[32] Google, “Application security | Android Open Source Project.” [Online]. Available: https://source.android.com/security/overview/app-security. [Accessed: 30-Sep-2017].
[33] Google, “Application Signing | Android Open Source Project.” [Online]. Available: https://source.android.com/security/apksigning/. [Accessed: 05-Oct-2017].

[34] Google, “Linking to Google Play | Android Developers.” [Online]. Available: https://developer.android.com/distribute/marketing-tools/linking-to-google-play.html. [Accessed: 10-Sep-2017].
[35] “PHQ-9 english.” [Online]. Available: http://www.phqscreeners.com/sites/g/files/g10016261/f/201412/PHQ-9_English.pdf. [Accessed: 15-Oct-2017].
[36] “Pfizer: One of the world’s premier biopharmaceutical companies.” [Online]. Available: http://www.pfizer.com/. [Accessed: 15-Oct-2017].
[37] “INSTRUCTION MANUAL Instructions for Patient Health Questionnaire (PHQ) and GAD-7 Measures.” [Online]. Available: https://phqscreeners.pfizer.edrupalgardens.com/sites/g/files/g10016261/f/201412/instructions.pdf. [Accessed: 15-Oct-2017].

ANNEX I: Code

a) Core Library

CustomCursor.java

```
package com.kasapakis.adam.plugin_model_library_core.database;

import android.database.CursorWindow;
import android.os.Bundle;
import android.os.Parcel;
import android.os.Parcelable;
import android.util.Log;

import java.util.HashMap;

/**
 * Class used to return data from an SQLite query from a remote process.
 */

public class CustomCursor implements Parcelable {
    /**
     * TAG.
     */
    private static final String TAG = CustomCursor.class.getSimpleName();
    /**
     * String constant for bundle kay.
     */
    private static final String BUNDLE_KEY = "HashMap";
    /**
     * How many columns we have
     */
    public int numColumns = 0;
    /**
     * passed projection
     */
    public Bundle mBundle = new Bundle();
    /**
     * Current row
     */
    private int curRow = -1;
    /**
     * Cursor Window.
     */
    private CursorWindow window;
    /**
     * Hashmap for columns and index.
     */
    private HashMap<String, Integer> mProjectionMap = new HashMap<String, Integer>();


    public static final Parcelable.Creator<CustomCursor> CREATOR =
            new Parcelable.Creator<CustomCursor>() {
                @Override
                public CustomCursor createFromParcel(Parcel in) {
                    return new CustomCursor(in);
                }

                @Override
                public CustomCursor[] newArray(int size) {
                    return new CustomCursor[size];
                }
            };

    /**
```

```
     * Constructor
     */
    public CustomCursor() {
    }

    /**
     * CustomCursor constructor with CursorWindow parameter.
     *
     * @param cursorWindow contains the data of the cursor object returned from a
     *query.
     */
    public CustomCursor(CursorWindow cursorWindow) {
        window = cursorWindow;
    }

    private CustomCursor(Parcel in) {
        readFromParcel(in);
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeParcelable(window, 0);
        dest.writeInt(numColumns);
        dest.writeInt(curRow);
        dest.writeBundle(mBundle);
    }

    public void readFromParcel(Parcel in) {
        window = in.readParcelable(CursorWindow.class.getClassLoader());
        numColumns = in.readInt();
        curRow = in.readInt();
        mBundle = in.readBundle();
        if (mBundle != null) {
            mProjectionMap =
                    (HashMap<String, Integer>) mBundle.getSerializable(BUNDLE_KEY);
        }
    }

    /**
     * Returns String value in the column.
     *
     * @param columnName
     * @return
     */
    public String getString(String columnName) {
        if (mProjectionMap != null) {
            if (mProjectionMap.containsKey(columnName)) {
                int columnIndex = mProjectionMap.get(columnName);
                return window.getString(curRow, columnIndex);
            } else {
                Log.e(TAG, "Didn't match the column name.");
                return null;
            }
        } else {
            Log.e(TAG, "Projecion map is null.");
            return null;
        }
    }

    /**
     * Moves to the first row of cursorWindow.
     *
```

```
 * @return
 */
public boolean moveToFirst() {
    if (window.getNumRows() == 0) {
        return false;
    }
    curRow = 0;
    return true;
}

/**
 * Moves to the last row of cursor window.
 *
 * @return
 */
public boolean moveToLast() {
    if (window.getNumRows() == 0) {
        return false;
    }
    curRow = window.getNumRows() - 1;
    return true;
}

/**
 * Checks if its not last row.
 *
 * @return true if the row if the last of after the last
 */
public boolean isAfterLast() {
    return (curRow >= window.getNumRows());
}

/**
 * Moves to the next row of the cursor window.
 *
 * @return
 */
public boolean moveToNext() {
    curRow++;
    if (isAfterLast()) {
        curRow = window.getNumRows();
        return false;
    }
    return true;
}

/**
 * Returns the long value.
 *
 * @param columnName
 * @return
 */
public long getLong(String columnName) {
    if (mProjectionMap != null) {
        if (mProjectionMap.containsKey(columnName)) {
            int columnIndex = mProjectionMap.get(columnName);
            return window.getLong(curRow, columnIndex);
        } else {
            Log.e(TAG, "Didn't match the column name.");
            return 0;
        }
    } else {
        Log.e(TAG, "Projecion map is null.");
        return 0;
    }
}
```

```
/**
 * Returns count of rows.
 *
 * @return
 */
public int getCount() {
    return window.getNumRows();
}

/**
 * Returns the int value.
 *
 * @param columnName
 * @return Int
 */
public int getInt(String columnName) {
    if (mProjectionMap != null) {
        if (mProjectionMap.containsKey(columnName)) {
            int columnIndex = mProjectionMap.get(columnName);
            return window.getInt(curRow, columnIndex);
        } else {
            Log.e(TAG, "Didn't match the column name.");
            return 0;
        }
    } else {
        Log.e(TAG, "Projecion map is null.");
        return 0;
    }
}

/**
 * Close the window cursor.
 */
public void close() {
    window.close();
}

/**
 * Returns the Blob type.
 *
 * @param columnName
 * @return byte[]
 */
public byte[] getBlob(String columnName) {
    if (mProjectionMap != null) {
        if (mProjectionMap.containsKey(columnName)) {
            int columnIndex = mProjectionMap.get(columnName);
            return window.getBlob(curRow, columnIndex);
        } else {
            return null;
        }
    } else {
        return null;
    }
}

/**
 * This method fill the projection elements in a list.
 *
 * @param projection String[]
 */
public void fillColumns(String[] projection) {
    int length = projection.length;
    HashMap projectionMap = new HashMap();
    for (int i = 0; i < length;
         i++) {
        projectionMap.put(projection[i], i);
```

```java
        }
        mBundle.putSerializable(BUNDLE_KEY, projectionMap);
    }
}
```

Schema.java

```java
package com.kasapakis.adam.plugin_model_library_core.database.schema;

import android.os.Parcel;
import android.os.Parcelable;

import java.util.ArrayList;

/**
 * Class used to define the database schema the plug-in needs to be able to save
 *data into Host's database.
 * The schema should have a name, version and contain Tables to be created.
 */

public class Schema implements Parcelable {
    public static final Parcelable.Creator<Schema> CREATOR = new
            Parcelable.Creator<Schema>() {
        @Override
        public Schema createFromParcel(Parcel source) {
            return new Schema(source);
        }

        @Override
        public Schema[] newArray(int size) {
            return new Schema[size];
        }
    };
    private String name;
    private int version;
    private ArrayList<Table> tables = new ArrayList<>();

    public Schema() {
    }

    /**
     * Constructs Schema object by setting Schema name and version.
     *
     * @param name    name of Schema object
     * @param version version of Schema object
     */
    public Schema(String name, int version) {
        this.name = name;
        this.version = version;
    }

    private Schema(Parcel in) {
        this.name = in.readString();
        this.version = in.readInt();
        this.tables = in.createTypedArrayList(Table.CREATOR);
    }

    /**
     * Adds a Table object to the schema.
     *
     * @param table Table object to be added in Schema
     */
    public void addTable(Table table) {
        tables.add(table);
    }

    /**
     * @return ArrayList of tables contained in Schema
```

```java
     */
    public ArrayList<Table> getTables() {
        return tables;
    }

    /**
     * @return name of the Schema object
     */
    public String getName() {
        return name;
    }

    /**
     * Set Schema name.
     *
     * @param schemaName name to set for Schema object
     */
    public void setName(String schemaName) {
        this.name = schemaName;
    }

    /**
     * Get version of Schema.
     *
     * @return the version of the Schema
     */
    public int getVersion() {
        return version;
    }

    /**
     * Sets the Schema version.
     *
     * @param version version number of the Schema object
     */
    public void setVersion(int version) {
        this.version = version;
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(this.name);
        dest.writeInt(this.version);
        dest.writeTypedList(this.tables);
    }
}
```

Table.java

```java
package com.kasapakis.adam.plugin_model_library_core.database.schema;

import android.os.Parcel;
import android.os.Parcelable;
import java.util.ArrayList;

/**
 * Class used to define a database table.
 * It is represented by a table name, an optional table version and a list of
 * column objects
 * that describe the columns.
 */
```

```java
public class Table implements Parcelable {
    public static final Parcelable.Creator<Table> CREATOR = new Parcelable.Creator<Table>() {
        @Override
        public Table createFromParcel(Parcel source) {
            return new Table(source);
        }

        @Override
        public Table[] newArray(int size) {
            return new Table[size];
        }
    };
    private String tableName;
    private int tableVersion = 1;
    private ArrayList<Column> columns = new ArrayList<>();

    public Table() {
    }

    /**
     * Create a Table object by passing its name.
     *
     * @param tableName Name of the Table.
     */
    public Table(String tableName) {
        this.tableName = tableName;
    }

    private Table(Parcel in) {
        this.tableName = in.readString();
        this.tableVersion = in.readInt();
        this.columns = in.createTypedArrayList(Column.CREATOR);
    }

    /**
     * Add a column to the Table object.
     *
     * @param column the column to add.
     */
    public void addColumn(Column column) {
        columns.add(column);
    }

    /**
     * Returns the name of the table.
     *
     * @return the table name.
     */
    public String getTableName() {
        return tableName;
    }

    /**
     * Get Table version.
     *
     * @return version of the Table
     */
    public int getTableVersion() {
        return tableVersion;
    }

    /**
     * Sets the version of the Table.
     *
     * @param tableVersion version of the Table
     */
    public void setTableVersion(int tableVersion) {
```

```java
        this.tableVersion = tableVersion;
    }

    /**
     * Returns the Column objects.
     *
     * @return ArrayList of Column objects
     */
    public ArrayList<Column> getColumns() {
        return columns;
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(this.tableName);
        dest.writeInt(this.tableVersion);
        dest.writeTypedList(this.columns);
    }
}
```

Column.java

```java
package com.kasapakis.adam.plugin_model_library_core.database.schema;

import android.os.Parcel;
import android.os.Parcelable;

/**
 * Class used to define a database table column.
 * It should have a name and the data type the column holds into.
 */

public class Column implements Parcelable {
    /**
     * Set column type as TEXT.
     */
    public static final String TEXT = "TEXT";
    /**
     * Set column type as NUMERIC.
     */
    public static final String NUMERIC = "NUMERIC";
    /**
     * Set column type as INTEGER.
     */
    public static final String INTEGER = "INTEGER";
    /**
     * Set column type as REAL.
     */
    public static final String REAL = "REAL";
    /**
     * Set column type as BLOB.
     */
    public static final String BLOB = "BLOB";

    public static final Parcelable.Creator<Column> CREATOR = new Parcelable.Creator<Column>() {
        @Override
        public Column createFromParcel(Parcel source) {
            return new Column(source);
        }

        @Override
        public Column[] newArray(int size) {
```

```
            return new Column[size];
        }
    };
    private String name;
    private String type;

    public Column() {
    }

    public Column(String name, String type) {
        this.name = name;
        this.type = type;
    }

    private Column(Parcel in) {
        this.name = in.readString();
        this.type = in.readString();
    }

    /**
     * method that returns column name
     *
     * @return name of column
     */
    public String getName() {
        return name;
    }

    /**
     * method that returns column type
     *
     * @return type of column
     */
    public String getType() {
        return type;
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(this.name);
        dest.writeString(this.type);
    }
}
```

UpdateMap.java

```
package com.kasapakis.adam.plugin_model_library_core.database.schema.updateDB;

import android.os.Parcel;
import android.os.Parcelable;

import java.util.ArrayList;

/**
 * Class used to map changes in old table versions to new versions.
 * Contains information about the schema version to update from,
 * TableMapObjects that describe how tables should be changed, and tables to
 * delete.
 * example: if updating from version 1 to any newer version
 * the fromVersion should be set 1 and the tables should contain old names from
 * version 1
 * and new names from current version.
```

```java
 * Should create one UpdateMap object for each version it is possible to update
 * from.
 */

public class UpdateMap implements Parcelable {
    public static final Parcelable.Creator<UpdateMap> CREATOR =
            new Parcelable.Creator<UpdateMap>() {
                @Override
                public UpdateMap createFromParcel(Parcel source) {
                    return new UpdateMap(source);
                }

                @Override
                public UpdateMap[] newArray(int size) {
                    return new UpdateMap[size];
                }
            };
    private int fromVersion;
    private ArrayList<UpdateTableMap> tables = new ArrayList<>();
    private ArrayList<String> tablesToDelete = new ArrayList<>();

    public UpdateMap() {
    }

    private UpdateMap(Parcel in) {
        this.fromVersion = in.readInt();
        this.tables = in.createTypedArrayList(UpdateTableMap.CREATOR);
        this.tablesToDelete = in.createStringArrayList();
    }

    /**
     * add table name to remove this table from the database.
     *
     * @param tableName table name to remove
     */
    public void addToDeleteTable(String tableName) {
        tablesToDelete.add(tableName);
    }

    /**
     * add UpdateTableMap object to describe how the table should be changed.
     *
     * @param map
     */
    public void addUpdateTableMap(UpdateTableMap map) {
        tables.add(map);
    }

    /**
     * Get the database version from which this object describes the update from.
     *
     * @return version to update from
     */
    public int getFromVersion() {
        return fromVersion;
    }

    /**
     * Set the database version from which this object describes the update from.
     *
     * @param fromVersion version to update from
     */
    public void setFromVersion(int fromVersion) {
        this.fromVersion = fromVersion;
    }

    /**
```

```java
     * Returns the table names that will be deleted from the database.
     *
     * @return
     */
    public ArrayList<String> getTablesToDelete() {
        return tablesToDelete;
    }

    /**
     * Returns the TableMaps that will be updated at the database.
     *
     * @return
     */
    public ArrayList<UpdateTableMap> getTablesToUpdate() {
        return tables;
    }

    public UpdateTableMap getTableMap(String tableName) {
        for (UpdateTableMap tbl : tables) {
            if (tbl.getNewTableName().equals(tableName)) {
                return tbl;
            }
        }
        return null;
    }

    /**
     * Checks if the object has a specific table name in the list of tables to be updated.
     *
     * @param tableName name of table to check
     * @return true if contains the table
     */
    public boolean hasTable(String tableName) {
        for (UpdateTableMap tableMap : tables) {
            if (tableMap.getNewTableName().equals(tableName)) {
                return true;
            }
        }
        return false;
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeInt(this.fromVersion);
        dest.writeTypedList(this.tables);
        dest.writeStringList(this.tablesToDelete);
    }
}
```

UpdateTableMap.java

```java
package com.kasapakis.adam.plugin_model_library_core.database.schema.updateDB;

import android.os.Parcel;
import android.os.Parcelable;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

/**
 * Class that describes the modification that should take place to turn an old
```

```
 * database table
 * to the new table.
 */

public class UpdateTableMap implements Parcelable {

    public static final Parcelable.Creator<UpdateTableMap> CREATOR =
            new Parcelable.Creator<UpdateTableMap>() {
                @Override
                public UpdateTableMap createFromParcel(Parcel source) {
                    return new UpdateTableMap(source);
                }

                @Override
                public UpdateTableMap[] newArray(int size) {
                    return new UpdateTableMap[size];
                }
            };
    private String oldTableName;
    private String newTableName;
    private Map<String, String> columns = new HashMap<String, String>();
    private ArrayList<String> columnsToDelete = new ArrayList<>();

    public UpdateTableMap() {
    }

    private UpdateTableMap(Parcel in) {
        this.oldTableName = in.readString();
        this.newTableName = in.readString();
        int columnsSize = in.readInt();
        this.columns = new HashMap<String, String>(columnsSize);
        for (int i = 0; i < columnsSize; i++) {
            String key = in.readString();
            String value = in.readString();
            this.columns.put(key, value);
        }
        this.columnsToDelete = in.createStringArrayList();
    }

    /**
     * @return the name of the existing table that should be changed
     */
    public String getOldTableName() {
        return oldTableName;
    }

    /**
     * @param oldTableName sets the name of the existing table that should be
     * changed
     */
    public void setOldTableName(String oldTableName) {
        this.oldTableName = oldTableName;
    }

    /**
     * @return new name of the table
     */
    public String getNewTableName() {
        return newTableName;
    }

    /**
     * @param newTableName new name of the table
     */
    public void setNewTableName(String newTableName) {
        this.newTableName = newTableName;
    }
```

```java
    /**
     * Returns a map containing the new column name as key and old column name as
     * value.
     *
     * @return map with new name as key and old name as value
     */
    public Map<String, String> getColumns() {
        return columns;
    }

    /**
     * @param columns map holding new column name and old table name sets
     */
    public void setColumns(Map<String, String> columns) {
        this.columns = columns;
    }

    /**
     * Adds a set in the columns map that contains the new column name and the
     * old column name.
     * new name should ALWAYS be the FIRST argument.
     *
     * @param newColumnName new name of the column
     * @param oldColumnName old name of the column that should be changed
     */
    public void addColumnToUpdate(String newColumnName, String oldColumnName) {
        columns.put(newColumnName, oldColumnName);
    }

    /**
     * @param columnName column name that should be removed from the table during
     * upgrade
     */
    public void addColumnToDelete(String columnName) {
        columnsToDelete.add(columnName);
    }

    /**
     * @return arrayList of the names of columns to delete
     */
    public ArrayList<String> getColumnsToDelete() {
        return columnsToDelete;
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(this.oldTableName);
        dest.writeString(this.newTableName);
        dest.writeInt(this.columns.size());
        for (Map.Entry<String, String> entry : this.columns.entrySet()) {
            dest.writeString(entry.getKey());
            dest.writeString(entry.getValue());
        }
        dest.writeStringList(this.columnsToDelete);
    }
}
```

NotActivatedException.java

```java
package com.kasapakis.adam.plugin_model_library_core.exceptions;

/**
```

```
 * Exception thrown if the plug-in is not activated in host application.
 */

public class NotActivatedException extends Exception {
    public NotActivatedException(String message) {
        super(message);
    }
}
```

NotInitialisedException.java

```
package com.kasapakis.adam.plugin_model_library_core.exceptions;

/**
 * Exception thrown if the plug-in is not initialised in host application.
 */

public class NotInitialisedException extends Exception {
    public NotInitialisedException(String message) {
        super(message);
    }
}
```

ReadOnlyException.java

```
package com.kasapakis.adam.plugin_model_library_core.exceptions;

/**
 * Exception thrown if the plug-in is read only type and tries to perform write actions.
 */

public class ReadOnlyException extends Exception {
    public ReadOnlyException(String message) {
        super(message);
    }
}
```

IPluginModel.aidl

```
package com.kasapakis.adam.plugin_model_library_core;

import com.kasapakis.adam.plugin_model_library_core.database.schema.updateDB.UpdateMap;
import com.kasapakis.adam.plugin_model_library_core.database.CustomCursor;
import com.kasapakis.adam.plugin_model_library_core.database.schema.Schema;

/**
* Interface for communication with Host Service.
*/

interface IPluginModel {

     /**
      * Remote method to create Schema in Database.
      *
      * @param callerName String: name of the plugin that calls the remote method.
      * @param schema     Schema: Schema object to be created in the database.
      * @param maps       List of UpdateMaps: holds the changes to, the newer version of the
      *                   schema.
      *                   Can be null if database version is 1.
      * @return returns true if Schema is successfully created.
      * @throws RemoteException
      */
     boolean createSchema(in String callersName, in Schema schema, in List<UpdateMap> maps);


     /**
      * Remote method to insert data in a table.
      *
      * @param callerName String: name of the plugin that calls the remote method.
      * @param table      String: the name of the table to insert the dta into.
```

```
 * @param values     ContentValues: values to insert in the table.
 * @throws RemoteException
 */
void iInsert(in String callersName, in String table, in ContentValues values);
/**
 * Remote method to query tables in the database.
 *
 * @param callerName    name of the plugin that calls the remote method.
 * @param tableName     String: The table name to compile the query again
 * @param columns       String: A list of which columns to return. Passing null will return
 *                  all columns, which is discouraged to prevent reading data from
 *                  storage that isn't going to be used.
 * @param selection     String: A filter declaring which rows to return, formatted as an
 *                  SQL WHERE clause (excluding the WHERE itself). Passing null will
 *                  return all rows for the given table.
 * @param selectionArgs String: You may include ?s in selection, which will be replaced by
 *                  the values from selectionArgs, in order that they appear in
 *                  the selection. The values will be bound as Strings.
 * @param groupBy       String: A filter declaring how to group rows, formatted as an
 *                  SQL GROUP BY clause (excluding the GROUP BY itself).
 *                  Passing null will cause the rows to not be grouped.
 * @param having        String: A filter declare which row groups to include in the cursor,
 *                  if row grouping is being used, formatted as an SQL HAVING clause
 *                  (excluding the HAVING itself). Passing null will cause all row
 *                  groups to be included, and is required when row grouping is
 *                  not being used.
 * @param orderBy       String: How to order the rows, formatted as an SQL ORDER BY
 *                  clause (excluding the ORDER BY itself). Passing null will use
 *                  the default
 *                  sort order, which may be unordered.
 * @return CustomCursor: containing the results from the database query.
 */
CustomCursor iQuery(in String callersName, in String tableName, in String[] columns,
       in String selection, in String[] selectionArgs, in String groupBy,
       in String having, String orderBy);

/**
 * Remote method for deleting rows in the database.
 *
 * @param table       String: the table to delete from.
 * @param whereClause String: the optional WHERE clause to apply when deleting.
 *                 Passing null will delete all rows.
 * @param whereArgs   String: You may include ?s in the where clause, which will be
 *                 replacedby the values from whereArgs. The values will be bound
 *                 as Strings.
 * @return true if delete was successful.
 */
boolean delete(in String table, in String whereClause, in String[] whereArgs);

/**
 * Remote method for replacing a row in the database.
 * Inserts a new row if a row does not already exists
 *
 * @param table          String: the table in which to replace the row
 * @param nullColumnHack String: optional; may be null. SQL doesn't allow inserting a
 *                  completely empty row without naming at least one column name.
 *                  If your provided initialValues is empty, no column names are
 *                  known and
 *                  an empty row can't be inserted. If not set to null, the
 *                  nullColumnHack
 *                  parameter provides the name of nullable column name to explicitly
 *                  insert a NULL into in the case where your initialValues is empty.
 * @param initialValues  ContentValues: this map contains the initial column values for
 *                  the row.
 *                  The keys should be the column names and the values the column
 *                  values.
 * @return true if replacement is successful.
```

```
     */
    boolean replace(in String table, in String nullColumnHack, in ContentValues
            initialValues);

    /**
     * Checks if a plug-in is initialised.
     *
     * @param pluginName String: name of the plug-in to check if is initialised.
     * @return true if plug-in is initialised.
     */
    boolean isInitialised(in String pluginName);

    /**
     * Checks if a plug-in is installed and detected by the Host application.
     *
     * @param pluginName String: name of the plug-in to check if is installed.
     * @return true if plug-in is installed.
     */
    boolean isInstalled(in String pluginName);

    /**
     * Checks if a plug-in is activated.
     *
     * @param pluginName String: name of the plug-in to check if is activated.
     * @return true if plug-in is activated.
     */
    boolean isActivated(in String pluginName);

    /**
     * Returns the version of schema for a particular plug-in if it is initialised before.
     *
     * @param pluginName String: name of the plug-in.
     * @return Int: version of the initialised schema.
     */
    int getInitialisedSchemaVersion(in String pluginName);

    /**
     * Broadcasts a notification.
     *
     * @param callerName String: name of the plug-in that fires the notification.
     * @param title      String: title of the notification.
     * @param text       String: text contained in the notification.
     */
    void fireNotification(in String pluginName, in String title, in String text);

    /**
     * deletes all notifications fired by a specific plug-in.
     *
     * @param callerName String: name of the plug-in to cancel it's notifications.
     */
    void cancelNotification(in String pluginName);

    /**
     * checks if a plug-in has an initialised schema.
     *
     * @param pluginName String: name of the plug-in.
     * @return true if host has a schema version for this plug-in.
     */
    boolean hasSchema(in String pluginName);
}
```

AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.kasapakis.adam.plugin_model_library_core">

    <permission android:name="com.kasapakis.adam.plugin_model_library.permission
                                    .HOST_SERVICE_PERMISSION"
```

```
            android:label="@string/host_service_permission_name"
            android:description="@string/host_service_permission_description"
            android:protectionLevel="signature"/>

    <application
        android:allowBackup="true"
        android:label="@string/app_name"
        android:supportsRtl="true"></application>
</manifest>
```

b) Client Library

ClientService.java

```
package com.kasapakis.adam.plugin_model_library_client;

import android.app.Service;
import android.content.ComponentName;
import android.content.ContentValues;
import android.content.Intent;
import android.content.ServiceConnection;
import android.content.pm.PackageManager;
import android.os.Binder;
import android.os.Bundle;
import android.os.IBinder;
import android.os.RemoteException;
import android.support.v4.content.LocalBroadcastManager;

import com.kasapakis.adam.plugin_model_library_core.IPluginModel;
import com.kasapakis.adam.plugin_model_library_core.database.CustomCursor;
import com.kasapakis.adam.plugin_model_library_core.database.schema.Schema;
import com.kasapakis.adam.plugin_model_library_core.database.schema.updateDB.UpdateMap;
import com.kasapakis.adam.plugin_model_library_core.exceptions.NotActivatedException;
import com.kasapakis.adam.plugin_model_library_core.exceptions.NotInitialisedException;

import java.util.List;

/**
 * Manages the connection to the Host service and returns CRUD or CRUDRead interface when a client
 * requests to bind on it.
 */
public abstract class ClientService extends Service {

    /**
     * Action that is broad-casted locally by the ClientService, when the service is connected to
     * Host service.
     * To be notified when service is connected, declare a local broadcast receiver and the set
     * it to receive intents whit this action.
     */
    public static final String CONNECTED_TO_HOST =
            "pluginmodel.client_service.action" + ".CONNECTED_TO_HOST";

    private String type;
    private IPluginModel mService;
    private boolean isConnectedToHost = false;

    private ServiceConnection mHostConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder iBinder) {
            mService = IPluginModel.Stub.asInterface(iBinder);
            isConnectedToHost = true;
            //attempts to initialise only when plugin type is read/write
            if (type.equals(getString(R.string.plugin_type_read_write))) {
                init();
            }
            Intent intent = new Intent(CONNECTED_TO_HOST);
            LocalBroadcastManager.getInstance(getApplicationContext()).sendBroadcast(intent);
```

```
    }

    @Override
    public void onServiceDisconnected(ComponentName componentName) {
        mService = null;
        isConnectedToHost = false;
    }
};

/**
 * ClientService constructor;
 */
public ClientService() {
}

/**
 * Abstract method used to initialise your Plugin.
 * <p>
 * This method will be called by the Host, if the plugin is not initialised yet,
 * or the Schema version is higher than the current one.
 * <p>
 * Example code:
 * <pre>
 * {@code
 *  Schema schema = new Schema();
 *  Table t1 = new Table("test_table1");
 *  t1.addColumn(new Column("col1", Column.INTEGER));
 *  t1.addColumn(new Column("col2", Column.INTEGER));
 *  schema.addTable(t1);
 *  Table t2 = new Table("test_table2");
 *  t2.addColumn(new Column("col1", Column.INTEGER));
 *  t2.addColumn(new Column("col2", Column.REAL));
 *  schema.addTable(t2);
 *  //remote method call to create schema.
 *  createTable(schema);
 * }
 * </pre>
 */
protected abstract boolean init();

@Override
public void onCreate() {
    super.onCreate();

    type = getPluginType();
    connectToHost();
}

@Override
public IBinder onBind(Intent intent) {
    //return local interface according to plug-in type.
    if (type.equals("read")) {
        return new CRUDReadBinder();
    } else if (type.equals("read/write")) {
        return new CRUDBinder();
    } else {
        return null;
    }
}

@Override
public boolean onUnbind(Intent intent) {
    unbindService(mHostConnection);
    return super.onUnbind(intent);
}

/**
```

```java
 * Method implementing connection to Host.
 *
 * @return True if Connection was successful.
 */
private boolean connectToHost() {
    Intent serviceIntent = new Intent()
            .setComponent(new ComponentName(
                    "com.kasapakis.adam.pluginmodeldemo",
                    "com.kasapakis.adam.pluginmodeldemo.MyServiceHost"));
    return bindService(serviceIntent, mHostConnection, BIND_AUTO_CREATE);
}

/**
 * Method used to create the schema in Host application's database.
 *
 * @param schema Schema: the schema of the database.
 * @param maps   ArrayList UpdateMap: map that describes changes from a lower version.
 * @return returns true if plugin is successfully initialised.
 */
public boolean createSchema(Schema schema, List<UpdateMap> maps) {
    boolean result = false;
    try {
        boolean isInit = mService.isInitialised(getPluginName());
        schema.setName(getPluginName());
        if (isConnectedToHost) {
            boolean isNewerVer = schema.getVersion() > mService.getInitialisedSchemaVersion
                    (getPluginName());
            if (!isInit) {
                result = mService.createSchema(getPluginName(), schema, maps);
            } else if (isNewerVer) {
                result = mService.createSchema(getPluginName(), schema, maps);
            }
        }
    } catch (RemoteException e) {
        e.printStackTrace();
    }
    return result;
}

/**
 * Determines the type of plug-in according to the metadata in manifest.
 *
 * @return String: type.
 */
private String getPluginType() {
    ComponentName myService = new ComponentName(this, this.getClass());
    Bundle data = null;
    try {
        data = getPackageManager().getServiceInfo(myService, PackageManager.GET_META_DATA)
                .metaData;
    } catch (PackageManager.NameNotFoundException e) {
        e.printStackTrace();
    }
    return data.getString("type", null);
}

/**
 * Helper method to return application label
 *
 * @return application label
 */
private String getPluginName() {
    return getApplicationInfo().loadLabel(getPackageManager()).toString();
}

/**
 * CRUDBinder class used to create a binder interface to communicate locally with the service
```

```
     * and call CRUD interface methods.
     */
    private class CRUDBinder extends Binder implements CRUD {

        @Override
        public void insert(String table, ContentValues values) throws NotActivatedException,
                NotInitialisedException {
            if (isConnectedToHost) {
                try {
                    if (mService.isActivated(getPluginName())) {
                        if (mService.isInitialised(getPluginName())) {
                            mService.iInsert(getPluginName(), table, values);
                        } else {
                            throw new NotInitialisedException("attempt to call insert(), plugin " +
                                    "is not initialised in Host");
                        }
                    } else {
                        throw new NotActivatedException("attempt to call insert(), plugin is not " +
                                "activated in Host");
                    }
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
            }
        }

        @Override
        public void fireNotification(String title, String text) throws NotActivatedException {
            if (isConnectedToHost) {
                try {
                    if (mService.isActivated(getPluginName())) {
                        mService.fireNotification(getPluginName(), title, text);
                    } else {
                        throw new NotActivatedException("attempt to fire notification, plugin is " +
                                "not activated in Host");
                    }

                } catch (RemoteException e) {
                    e.printStackTrace();
                }
            }
        }

        @Override
        public boolean initialise() {
            init();
            return true;
        }

        @Override
        public CustomCursor query(String pluginName, int dbVersion, String tableName, String[]
                columns, String selection, String[] selectionArgs, String groupBy, String having,
                              String orderBy)
                throws NotActivatedException, NotInitialisedException {

            CustomCursor cursor = null;
            //if pluginName is null this means the requested data are created by the plugin that
            // calls the method, otherwise, it request data saved from another plugin.
            if (pluginName == null) {
                pluginName = getPluginName();
            }
            if (isConnectedToHost) {
                try {
                    if (mService.isActivated(getPluginName())) {
                        if (mService.hasSchema(pluginName) && mService.getInitialisedSchemaVersion
                                (pluginName) == dbVersion) {
                            cursor = mService.iQuery(pluginName, tableName, columns,
```

```
                        selection, selectionArgs, groupBy, having, orderBy);
                } else {
                    throw new NotInitialisedException("attempt to call query(), plugin "
                        + pluginName + " is not initialised in Host");
                }
            } else {
                throw new NotActivatedException("attempt to call query(), plugin is not " +
                    "activated in Host");
            }
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
    return cursor;
}

@Override
public boolean deleteRow(String table, String whereClause, String[] whereArgs)
        throws NotActivatedException, NotInitialisedException {
    if (isConnectedToHost) {
        try {
            if (mService.isActivated(getPluginName())) {
                if (mService.isInitialised(getPluginName())) {
                    return mService.delete(table, whereClause, whereArgs);
                } else {
                    throw new NotInitialisedException("attempt to call deleteRow(), " +
                        "plugin is not initialised in Host");
                }
            } else {
                throw new NotActivatedException("attempt to call deleteRow(), plugin is " +
                    "not activated in Host");
            }
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
    return false;
}

@Override
public boolean replace(String table, String nullColumnHack, ContentValues initialValues)
        throws NotActivatedException, NotInitialisedException {
    if (isConnectedToHost) {
        try {
            if (mService.isActivated(getPluginName())) {
                if (mService.isInitialised(getPluginName())) {
                    return mService.replace(table, nullColumnHack, initialValues);
                } else {
                    throw new NotInitialisedException("attempt to call replace(), plugin " +
                        "is not initialised in Host");
                }
            } else {
                throw new NotActivatedException("attempt to call replace(), plugin is not" +
                    " activated in Host");
            }
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    return false;
}

@Override
public void cancelNotification() throws NotActivatedException {
    if (isConnectedToHost) {
        try {
```

```
                if (mService.isActivated(getPluginName())) {
                    mService.cancelNotification(getPluginName());
                } else {
                    throw new NotActivatedException("attempt to delete notification, plugin " +
                            "is not activated in Host");
                }
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    }
}

private class CRUDReadBinder extends Binder implements CRUDRead {

    @Override
    public CustomCursor query(String pluginName, int dbVersion, String tableName, String[]
            columns, String selection, String[] selectionArgs, String groupBy, String having,
                          String orderBy)
            throws NotActivatedException, NotInitialisedException {

        CustomCursor cursor = null;
        if (isConnectedToHost) {
            try {
                if (mService.isActivated(getPluginName())) {
                    if (mService.hasSchema(pluginName) && mService.getInitialisedSchemaVersion
                            (pluginName) == dbVersion) {
                        cursor = mService.iQuery(pluginName, tableName, columns,
                                selection, selectionArgs, groupBy, having, orderBy);
                    } else {
                        throw new NotInitialisedException("attempt to call query(), plugin "
                                + pluginName + " is not initialised in Host");
                    }
                } else {
                    throw new NotActivatedException("attempt to call query(), plugin is not " +
                            "activated in Host");
                }
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
        return cursor;
    }

    @Override
    public void fireNotification(String title, String text) throws NotActivatedException {
        if (isConnectedToHost) {
            try {
                if (mService.isActivated(getPluginName())) {
                    mService.fireNotification(getPluginName(), title, text);
                } else {
                    throw new NotActivatedException("attempt to fire notification, plugin is " +
                            "not activated in Host");
                }

            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void cancelNotification() throws NotActivatedException {
        if (isConnectedToHost) {
            try {
                if (mService.isActivated(getPluginName())) {
                    mService.cancelNotification(getPluginName());
```

```java
                } else {
                    throw new NotActivatedException("attempt to delete notification, plugin " +
                            "is not activated in Host");
                }
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    }
  }
}
```

CRUD.java

```java
package com.kasapakis.adam.plugin_model_library_client;

import android.content.ContentValues;

import com.kasapakis.adam.plugin_model_library_core.database.CustomCursor;
import com.kasapakis.adam.plugin_model_library_core.exceptions.NotActivatedException;
import com.kasapakis.adam.plugin_model_library_core.exceptions.NotInitialisedException;

/**
 * Interface that exposes methods to a read/write plug-in.
 */
public interface CRUD {

    /**
     * Method for inserting a row into the database.
     *
     * @param table  String: the table to insert the row into
     * @param values ContentValues: this map contains the initial column values for the row.
     *               The keys should be the column names and the values the column values.
     * @throws NotActivatedException   thrown if the plug-in is not activated in host application.
     * @throws NotInitialisedException thrown if the plug-in is not initialised.
     */
    void insert(String table, ContentValues values) throws NotActivatedException,
        NotInitialisedException;

    boolean initialise();

    /**
     * @param pluginName    String: name of the plugin that created the table.
     *                      This is necessary to check if the table exists.
     *                      If reading from your own tables set to NULL.
     * @param dbVersion     int: version of the database Schema that defines the table.
     * @param tableName     String: The table name to compile the query against.
     * @param columns       A list of which columns to return. Passing null will return all columns,
     *                      which is discouraged to prevent reading data from storage that isn't
     *                      going to be used.
     * @param selection     String[]: A filter declaring which rows to return,
     *                      formatted as an SQL WHERE clause (excluding the WHERE itself).
     *                      Passing null will return all rows for the given table.
     * @param selectionArgs String[]: You may include ?s in selection, which will be replaced by the
     *                      value from selectionArgs, in order that they appear in the selection.
     *                      The values will be bound as Strings.
     * @param groupBy       String: A filter declaring how to group rows, formatted as an
     *                      SQL GROUP BY clause (excluding the GROUP BY itself).
     *                      Passing null will cause the rows to not be grouped.
     * @param having        String: A filter declare which row groups to include in the cursor,
     *                      if row grouping is being used, formatted as an SQL HAVING clause
     *                      (excluding the HAVING itself).
     *                      Passing null will cause all row groups to be included, and is required
     *                      when row grouping is not being used.
     * @param orderBy       String: How to order the rows, formatted as an SQL ORDER BY clause
     *                      (excluding the ORDER BY itself).
     *                      Passing null will use the default sort order, which may be unordered.
```

```java
     * @return CustomCursor: containing the results from the database query.
     * @throws NotActivatedException   thrown if the plug-in is not activated in host application
     * @throws NotInitialisedException thrown if the plug-in that create the tables that are queried
     *                                 is not initialised in host application.
     */
    CustomCursor query(String pluginName, int dbVersion, String tableName, String[] columns,
                       String selection
            , String[] selectionArgs, String groupBy, String having, String orderBy)
            throws NotActivatedException, NotInitialisedException;

    /**
     * Method for deleting rows in the database.
     *
     * @param table       String: the table to delete from.
     * @param whereClause String: the optional WHERE clause to apply when deleting.
     *                    Passing null will delete all rows.
     * @param whereArgs   String: You may include ?s in the where clause, which will be replaced
     *                    by the values from whereArgs. The values will be bound as Strings.
     * @return true if delete was successful.
     * @throws NotActivatedException   thrown if the plug-in is not activated in host application.
     * @throws NotInitialisedException thrown if the plug-in is not initialised.
     */
    boolean deleteRow(String table, String whereClause, String[] whereArgs)
            throws NotActivatedException, NotInitialisedException;

    /**
     * Method for replacing a row in the database.
     * Inserts a new row if a row does not already exists
     *
     * @param table         String: the table in which to replace the row
     * @param nullColumnHack String: optional; may be null. SQL doesn't allow inserting a
     *                      completely empty row without naming at least one column name.
     *                      If your provided initialValues is empty, no column names are known and
     *                      an empty row can't be inserted. If not set to null, the nullColumnHack
     *                      parameter provides the name of nullable column name to explicitly
     *                      insert a NULL into in the case where your initialValues is empty.
     * @param initialValues  ContentValues: this map contains the initial column values for the row.
     *                      The keys should be the column names and the values the column values.
     * @return true if replacement is successful.
     * @throws NotActivatedException   thrown if the plug-in is not activated in host application.
     * @throws NotInitialisedException thrown if the plug-in is not initialised.
     */
    boolean replace(String table, String nullColumnHack, ContentValues initialValues)
            throws NotActivatedException, NotInitialisedException;

    /**
     * Sends a notification to be broadcasted by the host.
     *
     * @param title String: title of the notification
     * @param text  String: message of the notification
     * @throws NotActivatedException thrown if the plug-in is not activated in host application
     */
    void fireNotification(String title, String text) throws NotActivatedException;

    /**
     * Removes notifications broadcasted by this plug-in.
     *
     * @throws NotActivatedException thrown if the plug-in is not activated in host application¬
     */
    void cancelNotification() throws NotActivatedException;
}
```

CRUDRead.java

```java
package com.kasapakis.adam.plugin_model_library_client;

import com.kasapakis.adam.plugin_model_library_core.database.CustomCursor;
```

```java
import com.kasapakis.adam.plugin_model_library_core.exceptions.NotActivatedException;
import com.kasapakis.adam.plugin_model_library_core.exceptions.NotInitialisedException;

/**
 * Interface that exposes methods to a read only plug-in.
 */
public interface CRUDRead {

    /**
     * Method to query host's database.
     *
     * @param pluginName    String: name of the plugin that created the table.
     *                      This is necessary to check if the table exists.
     * @param dbVersion     int: version of the database Schema that defines the table.
     * @param tableName     String: The table name to compile the query against.
     * @param columns       String[]: A list of which columns to return. Passing null will return
     *                      all columns, which is discouraged to prevent reading data from storage
     *                      that isn't going to be used.
     * @param selection     String: A filter declaring which rows to return,
     *                      formatted as an SQL WHERE clause (excluding the WHERE itself).
     *                      Passing null will return all rows for the given table.
     * @param selectionArgs String[]: You may include ?s in selection, which will be replaced by the
     *                      values from selectionArgs, in order that they appear in the selection.
     *                      The values will be bound as Strings.
     * @param groupBy       String: A filter declaring how to group rows, formatted as
     *                      an SQL GROUP BY clause (excluding the GROUP BY itself).
     *                      Passing null will cause the rows to not be grouped.
     * @param having        String: A filter declare which row groups to include in the cursor,
     *                      if row grouping is being used, formatted as an SQL HAVING clause
     *                      (excluding the HAVING itself).
     *                      Passing null will cause all row groups to be included, and is required
     *                      when row grouping is not being used.
     * @param orderBy       String: How to order the rows, formatted as an SQL ORDER BY clause
     *                      (excluding the ORDER BY itself).
     *                      Passing null will use the default sort order, which may be unordered.
     * @return CustomCursor: containing the results from the database query.
     * @throws NotActivatedException   thrown if the plug-in is not activated in host application
     * @throws NotInitialisedException thrown if the plug-in that create the tables that are queried
     *                                 is not initialised in host application.
     */
    CustomCursor query(String pluginName, int dbVersion, String tableName, String[] columns, String
            selection
            , String[] selectionArgs, String groupBy, String having, String orderBy)
            throws NotActivatedException, NotInitialisedException;

    /**
     * Sends a notification to be broadcasted by the host.
     *
     * @param title String: title of the notification
     * @param text  String: message of the notification
     * @throws NotActivatedException thrown if the plug-in is not activated in host application
     */
    void fireNotification(String title, String text) throws NotActivatedException;

    /**
     * Removes notifications broadcasted by this plug-in.
     *
     * @throws NotActivatedException thrown if the plug-in is not activated in host application
     */
    void cancelNotification() throws NotActivatedException;
}
```

AndroidManifest.xml

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.kasapakis.adam.plugin_model_library_client">
```

```
    <uses-permission android:name="com.kasapakis.adam.plugin_model_library.permission
                        .HOST_SERVICE_PERMISSION"/>
    <application
        android:allowBackup="true"
        android:label="@string/app_name"
        android:supportsRtl="true">
        <service
            android:name=".ClientService"
            android:enabled="true"
            android:exported="false"
            />
    </application>
</manifest>
```

c) Host Library

HostService.java

```
package com.kasapakis.adam.plugin_model_library_host.service;

import android.app.NotificationManager;
import android.app.PendingIntent;
import android.app.Service;
import android.content.ContentValues;
import android.content.Context;
import android.content.Intent;
import android.database.CrossProcessCursorWrapper;
import android.database.Cursor;
import android.database.CursorWindow;
import android.graphics.Bitmap;
import android.graphics.drawable.BitmapDrawable;
import android.graphics.drawable.Drawable;
import android.os.IBinder;
import android.os.RemoteException;
import android.support.v4.app.NotificationCompat;
import android.util.Log;

import com.kasapakis.adam.plugin_model_library_core.IPluginModel;
import com.kasapakis.adam.plugin_model_library_core.database.CustomCursor;
import com.kasapakis.adam.plugin_model_library_core.database.schema.Schema;
import com.kasapakis.adam.plugin_model_library_core.database.schema.Table;
import com.kasapakis.adam.plugin_model_library_core.database.schema.updateDB.UpdateMap;
import com.kasapakis.adam.plugin_model_library_host.database.DatabaseManager;
import com.kasapakis.adam.plugin_model_library_host.plugin_manager.Registry;

import java.util.ArrayList;
import java.util.List;

/**
 * Service that provides extension point to the Host application. Plug-ins bind on this service to
 * communicate with Host.
 */
public class HostService extends Service {
    private DatabaseManager dbManager;
    //remote interface implementation
    private final IPluginModel.Stub mBinder = new IPluginModel.Stub() {
        /**
         * Remote method to create Schema in Database.
         *
         * @param callerName String: name of the plugin that calls the remote method.
         * @param schema     Schema: Schema object to be created in the database.
         * @param maps       List of UpdateMaps: holds the changes to, the newer version of the
         *                   schema.
         *                   Can be null if database version is 1.
         * @return returns true if Schema is successfully created.
```

```java
     * @throws RemoteException
     */
    @Override
    public boolean createSchema(String callerName, Schema schema, List<UpdateMap> maps)
            throws RemoteException {
        // check if is initialised
        if (isInitialised(callerName)) {
            //if new schema version higher updates schema
            if (getInitialisedSchemaVersion(schema.getName()) < schema.getVersion()) {
                if (maps != null) {
                    UpdateMap map = new UpdateMap();
                    boolean successful = false; //will hold db transaction result.

                    // iterate to get the appropriate map according to which version updating
                    // from.
                    for (UpdateMap temp : maps) {
                        if (getInitialisedSchemaVersion(schema.getName()) == temp
                                .getFromVersion()) {
                            map = temp;
                            break;
                        }
                    }
                    if (map != null) {
                        Schema tempSchema = new Schema(); //temporary schema to hold new tables

                        //don't update new tables, just normally create
                        for (Table table : schema.getTables()) {
                            if (!map.hasTable(table.getTableName())) {  //if new add to list
                                // to create later.
                                tempSchema.addTable(table);
                            } else {  //if needs update
                                successful = dbManager.upgradeTable(table, map.getTableMap
                                        (table.getTableName()));
                            }
                        }
                        //create new tables.
                        successful = successful && dbManager.createSchema(tempSchema);
                        //iterate tables to delete and delete them.
                        for (String t : map.getTablesToDelete()) {
                            successful = successful && dbManager.deleteTable(t);
                        }
                    } else {        //if map is null return false.
                        return false;
                    }
                    return successful;
                }
            }
        } else {            //runs if not initialised, to initialise schema normally.
            if (dbManager.createSchema(schema)) {
                Registry.getInstance()
                        .addSchema(schema, getApplicationContext());
                Registry.getInstance()
                        .setInitialised(schema.getName(), true, getApplicationContext());
                return true;
            }
        }
        return true;
    }

    /**
     * Remote method to insert data in a table.
     *
     * @param callerName String: name of the plugin that calls the remote method.
     * @param table      String: the name of the table to insert the dta into.
     * @param values     ContentValues: values to insert in the table.
     * @throws RemoteException
     */
```

```
@Override
public void iInsert(String callerName, String table, ContentValues values) throws
      RemoteException {
   dbManager.mInsert(table, null, values);
}

/**
 * Remote method to query tables in the database.
 *
 * @param callerName    name of the plugin that calls the remote method.
 * @param tableName     String: The table name to compile the query again
 * @param columns       String: A list of which columns to return. Passing null will return
 *                all columns, which is discouraged to prevent reading data from
 *                storage that isn't going to be used.
 * @param selection     String: A filter declaring which rows to return, formatted as an
 *                SQL WHERE clause (excluding the WHERE itself). Passing null will
 *                return all rows for the given table.
 * @param selectionArgs String: You may include ?s in selection, which will be replaced by
 *                the values from selectionArgs, in order that they appear in
 *                the selection. The values will be bound as Strings.
 * @param groupBy       String: A filter declaring how to group rows, formatted as an
 *                SQL GROUP BY clause (excluding the GROUP BY itself).
 *                Passing null will cause the rows to not be grouped.
 * @param having        String: A filter declare which row groups to include in the cursor,
 *                if row grouping is being used, formatted as an SQL HAVING clause
 *                (excluding the HAVING itself). Passing null will cause all row
 *                groups to be included, and is required when row grouping is
 *                not being used.
 * @param orderBy       String: How to order the rows, formatted as an SQL ORDER BY clause
 *                (excluding the ORDER BY itself). Passing null will use the default
 *                sort order, which may be unordered.
 * @return CustomCursor: containing the results from the database query.
 */
@Override
public CustomCursor iQuery(String callerName, String tableName, String[] columns, String
      selection, String[] selectionArgs, String groupBy, String having, String orderBy) {

   //do the query
   Cursor cursor = dbManager.mQuery(tableName, columns, selection, selectionArgs,
         groupBy, having, orderBy);
   //cursor window will hold the data.
   CursorWindow window = new CursorWindow("MyCursorWindow");

   //create and fill window with data fro query.
   CrossProcessCursorWrapper crossProcessCursor = new CrossProcessCursorWrapper(cursor);
   crossProcessCursor.fillWindow(0, window);

   //create CustomCursor and add data from query.
   CustomCursor customCursor = new CustomCursor(window);
   customCursor.numColumns = columns.length;
   customCursor.fillColumns(columns);

   crossProcessCursor.close();
   return customCursor;
}

/**
 * Remote method for deleting rows in the database.
 *
 * @param table       String: the table to delete from.
 * @param whereClause String: the optional WHERE clause to apply when deleting.
 *               Passing null will delete all rows.
 * @param whereArgs   String: You may include ?s in the where clause, which will be replaced
 *               by the values from whereArgs. The values will be bound as Strings.
 * @return true if delete was successful.
 */
@Override
```

```java
    public boolean delete(String table, String whereClause, String[] whereArgs) {
        return dbManager.delete(table, whereClause, whereArgs);
    }

    /**
     * Remote method for replacing a row in the database.
     * Inserts a new row if a row does not already exists
     *
     * @param table         String: the table in which to replace the row
     * @param nullColumnHack String: optional; may be null. SQL doesn't allow inserting a
     *                      completely empty row without naming at least one column name.
     *                      If your provided initialValues is empty, no column names are
     *                      known and
     *                      an empty row can't be inserted. If not set to null, the
     *                      nullColumnHack
     *                      parameter provides the name of nullable column name to explicitly
     *                      insert a NULL into in the case where your initialValues is empty.
     * @param initialValues  ContentValues: this map contains the initial column values for
     *                      the row.
     *                      The keys should be the column names and the values the column
     *                      values.
     * @return true if replacement is successful.
     */
    @Override
    public boolean replace(String table, String nullColumnHack, ContentValues initialValues) {
        return dbManager.replace(table, nullColumnHack, initialValues);
    }


    /**
     * Checks if a plug-in is initialised.
     *
     * @param pluginName String: name of the plug-in to check if is initialised.
     * @return true if plug-in is initialised.
     */
    @Override
    public boolean isInitialised(String pluginName) {
        Boolean result = Registry.getInstance().isInitialised(pluginName,
                getApplicationContext());
        Log.d("in isInitialised", pluginName + result);
        return result;
    }

    /**
     * Checks if a plug-in is installed and detected by the Host application.
     *
     * @param pluginName String: name of the plug-in to check if is installed.
     * @return true if plug-in is installed.
     */
    @Override
    public boolean isInstalled(String pluginName) {
        return Registry.getInstance().isInstalled(pluginName, getBaseContext());
    }

    /**
     * Checks if a plug-in is activated.
     *
     * @param pluginName String: name of the plug-in to check if is activated.
     * @return true if plug-in is activated.
     */
    @Override
    public boolean isActivated(String pluginName) {
        return Registry.getInstance().isActivated(pluginName, getApplicationContext());
    }

    /**
     * Returns the version of schema for a particular plug-in if it is initialised before.
```

```java
 *
 * @param pluginName String: name of the plug-in.
 * @return Int: version of the initialised schema.
 */
@Override
public int getInitialisedSchemaVersion(String pluginName) {
   return Registry.getInstance().getSchemaVersion(pluginName, getApplicationContext());
}

/**
 * checks if a plug-in has an initialised schema.
 *
 * @param pluginName String: name of the plug-in.
 * @return true if host has a schema version for this plug-in.
 */
@Override
public boolean hasSchema(String pluginName) {
   ArrayList<String> a = Registry.getInstance().getSchemaNames(getApplicationContext());
   return a.contains(pluginName);
}

/**
 * Broadcasts a notification.
 *
 * @param callerName String: name of the plug-in that fires the notification.
 * @param title      String: title of the notification.
 * @param text       String: text contained in the notification.
 */
@Override
public void fireNotification(String callerName, String title, String text) {
   //assign a hash code for each new plugin to be able to update notifications for each
   // plugins separately
   int pluginHash =
         Registry.getInstance().getPlugin(callerName, getApplicationContext())
               .hashCode();
   //load and convert application icon to drawable
   Context context = getApplicationContext();
   Drawable drawable = getApplicationInfo().loadIcon(getPackageManager());
   Bitmap bitmap = ((BitmapDrawable) drawable).getBitmap();

   //load notification icon
   int smallIconId =
         getResources().getIdentifier("notification_icon", "drawable", getPackageName());

   Intent intent = new Intent()
         .setAction(Registry.getInstance()
               .getPlugin(callerName, getApplicationContext())
               .getActivityPackage());
   PendingIntent pendingIntent =
         PendingIntent
               .getActivity(getApplicationContext(), 0, intent, PendingIntent
                     .FLAG_CANCEL_CURRENT);

   //define new notification
   NotificationCompat.Builder mBuilder =
         new NotificationCompat.Builder(context)
               .setLargeIcon(bitmap)
               .setSmallIcon(smallIconId)
               .setContentTitle(title)
               .setContentText(text)
               .setContentIntent(pendingIntent);

   //retrieve NotificationManager
   NotificationManager mNotificationManager =
         (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

   //publish notification
```

```java
            mNotificationManager.notify(pluginHash, mBuilder.build());
        }

        /**
         * deletes all notifications fired by a specific plug-in.
         *
         * @param callerName String: name of the plug-in to cancel it's notifications.
         */
        @Override
        public void cancelNotification(String callerName) {
            NotificationManager mNotificationManager =
                    (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
            int pluginHash = Registry.getInstance().getPlugin(callerName, getApplicationContext())
                    .hashCode();
            //cancel notification
            mNotificationManager.cancel(pluginHash);
        }
    };

    @Override
    public void onCreate() {
        super.onCreate();
        dbManager = new DatabaseManager(getApplicationContext());
    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }
}
```

DatabaseManager.java

```java
package com.kasapakis.adam.plugin_model_library_host.database;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;

import com.kasapakis.adam.plugin_model_library_core.database.schema.Column;
import com.kasapakis.adam.plugin_model_library_core.database.schema.Schema;
import com.kasapakis.adam.plugin_model_library_core.database.schema.Table;
import com.kasapakis.adam.plugin_model_library_core.database.schema.updateDB.UpdateTableMap;

import java.util.ArrayList;
import java.util.Map;

/**
 * Class that provides databse crud methods.
 */

public class DatabaseManager {
    private static String CREATE_BASE = "CREATE TABLE IF NOT EXISTS ";
    private SQLiteDatabase mDatabase;

    public DatabaseManager(Context mContext) {
        super();
        mDatabase = new DatabaseHelper(mContext).getWritableDatabase();
    }

    /**
     * Method used to create a table in the database.
     *
     * @param tableName String: name of table.
     * @param columns   String[] column names.
```

```java
 * @param types     String[] types.
 */
public void createTable(String tableName, String[] columns, String[] types) {
    String createQuery = CREATE_BASE + tableName + "(" + " _id integer primary key " +
            "autoincrement";
    createQuery = createQuery
            + ", timestamp DATETIME DEFAULT (DATETIME(CURRENT_TIMESTAMP, " + "'LOCALTIME'))";
    if (columns.length > 0) {
        for (int i = 0; i < columns.length; i++) {
            createQuery = createQuery + ", " + columns[i] + " " + types[i];
        }
    }
    createQuery = createQuery + ")";
    mDatabase.execSQL(createQuery);
}

/**
 * method used to create a schema in the database.
 *
 * @param schema Schema: the object that contains information about the schema of teh database.
 * @return true if transaction is successful.
 */
public Boolean createSchema(Schema schema) {
    boolean successful = false;
    if (schema == null) {
        return false;
    }
    mDatabase.beginTransactionNonExclusive();
    try {
        for (Table table : schema.getTables()) {
            String tableName = table.getTableName();
            String createQuery = CREATE_BASE + tableName
                    + "(" + " _id integer primary key " + "autoincrement";
            createQuery = createQuery
                    + ", timestamp DATETIME DEFAULT (DATETIME"
                    + "(CURRENT_TIMESTAMP, 'LOCALTIME'))";
            if (!table.getColumns().isEmpty()) {
                for (Column column : table.getColumns()) {
                    if (!(column.getType().isEmpty() || column.getName().isEmpty())) {
                        createQuery = createQuery + ", " + column.getName() + " " + column
                                .getType();
                    }
                }
            }
            createQuery = createQuery + ")";
            mDatabase.execSQL(createQuery);
        }
        successful = true;
        mDatabase.setTransactionSuccessful();

    } catch (SQLException e) {
        successful = false;
    } finally {
        mDatabase.endTransaction();
        return successful;
    }
}

/**
 * Upgrades a table according to the changes discribed in the table map.
 *
 * @param table    Table: table to upagrade.
 * @param tableMap UpdateTableMap: map the changes to the old table.
 * @return true if transaction is successful.
 */
public boolean upgradeTable(Table table, UpdateTableMap tableMap) {
    boolean successful = false;
```

```
        mDatabase.beginTransactionNonExclusive();
        try {
            //rename existing table to tmp.
            String tmpName = "tmp" + tableMap.getOldTableName();
            String statement = "ALTER TABLE "
                    + tableMap.getOldTableName() + " RENAME TO " + tmpName;
            mDatabase.execSQL(statement);

            String[] columnNames = new String[table.getColumns().size()];
            String[] types = new String[table.getColumns().size()];

            ArrayList<Column> colList = table.getColumns();
            for (int i = 0; i < colList.size(); i++) {
                columnNames[i] = colList.get(i).getName();
                types[i] = colList.get(i).getType();
            }
            createTable(table.getTableName(), columnNames, types); //create the new table

            Map<String, String> colMap = tableMap.getColumns();
            String selectCol = "_id";
            String insertCols = "_id";
            for (int i = 0; i < columnNames.length; i++) {
                if (!tableMap.getColumnsToDelete().contains(columnNames[i])) {
                    if (colMap.containsKey(columnNames[i])) {
                        selectCol = selectCol + ", ";
                        insertCols = insertCols + ", ";
                        selectCol = selectCol + colMap.get(columnNames[i]);
                        insertCols = insertCols + columnNames[i];
                    } else {
                        selectCol = selectCol + ", ";
                        insertCols = insertCols + ", ";
                        selectCol = selectCol + columnNames[i];
                        insertCols = insertCols + columnNames[i];
                    }
                }
            }
            //build statement to transfer data from old table to new.
            statement = "INSERT INTO "
                    + table.getTableName() + "("
                    + insertCols + ")"
                    + " SELECT "
                    + selectCol
                    + " FROM " + tmpName;
            //execute data transfer from old table to new.
            mDatabase.execSQL(statement);
            //drop old table.
            mDatabase.execSQL("DROP TABLE " + tmpName);
            successful = true;
            mDatabase.setTransactionSuccessful();
        } catch (SQLException e) {
            successful = false;
        } finally {
            mDatabase.endTransaction();
            return successful;
        }
    }

    /**
     * Query the given table, returning a Cursor over the result set.
     *
     * @param table        String: The table name to compile the query against.
     * @param columns      String: A list of which columns to return. Passing null will return all
     *                     columns, which is discouraged to prevent reading data from storage that
     *                     isn't going to be used.
     * @param selection    String: A filter declaring which rows to return, formatted as an
     *                     SQL WHERE clause (excluding the WHERE itself). Passing null will return
     *                     all rows for the given table.
```

```
 * @param selectionArgs String: You may include ?s in selection, which will be replaced by the
 *                  values from selectionArgs, in order that they appear in the selection.
 *                  The values will be bound as Strings.
 * @param groupBy      String: A filter declaring how to group rows, formatted as an
 *                  SQL GROUP BY clause (excluding the GROUP BY itself). Passing null will
 *                  cause the rows to not be grouped.
 * @param having       String: A filter declare which row groups to include in the cursor,
 *                  if row grouping is being used, formatted as an SQL HAVING clause
 *                  (excluding the HAVING itself). Passing null will cause all row groups
 *                  to be included, and is required when row grouping is not being used.
 * @param orderBy      String: How to order the rows, formatted as an SQL ORDER BY clause
 *                  (excluding the ORDER BY itself). Passing null will use the default sort
 *                  order, which may be unordered.
 * @return A Cursor object, which is positioned before the first entry. Note that Cursors are
 * not synchronized, see the documentation for more details.
 */
public Cursor mQuery(String table, String[] columns, String selection,
                     String[] selectionArgs, String groupBy, String having, String orderBy) {
    Cursor cursor;
    cursor = mDatabase.query(table,
            columns,
            selection,
            selectionArgs,
            groupBy,
            having,
            orderBy);
    return cursor;
}

/**
 * Method for inserting a row into the database.
 *
 * @param table         String: the table to insert the row into.
 * @param nullColumnHack String: optional; may be null. SQL doesn't allow inserting a completely
 *                      empty row without naming at least one column name. If your provided
 *                      values is empty, no column names are known and an empty row can't be
 *                      inserted. If not set to null, the nullColumnHack parameter provides
 *                      the name of nullable column name to explicitly insert a NULL into in
 *                      the case where your values is empty.
 * @param values        ContentValues: this map contains the initial column values for the row.
 *                      The keys should be the column names and the values the column values
 * @return the row ID of the newly inserted row, or -1 if an error occurred
 */
public Long mInsert(String table, String nullColumnHack, ContentValues values) {
    return mDatabase.insert(table, nullColumnHack, values);
}

/**
 * Drops table from database.
 *
 * @param tableName name of table to drop.
 * @return true i succesfully executed.
 */
public Boolean deleteTable(String tableName) {
    try {
        mDatabase.execSQL("DROP " + tableName);
        return true;
    } catch (SQLException e) {
        return false;
    }
}

/**
 * Method for replacing a row in the database. Inserts a new row if a row does not already
 * exist.
 *
 * @param table         String: the table in which to replace the row
```

```java
     * @param nullColumnHack String: optional; may be null. SQL doesn't allow inserting a completely
     *                     empty row without naming at least one column name. If your provided
     *                     initialValues is empty, no column names are known and an empty row
     *                     can't be inserted. If not set to null, the nullColumnHack parameter
     *                     provides the name of nullable column name to explicitly insert a NULL
     *                     into in the case where your initialValues is empty.
     * @param initialValues  ContentValues: this map contains the initial column values for the row.
     *                     The keys should be the column names and the values the column values.
     * @return the row ID of the newly inserted row, or -1 if an error occurred
     */
    public boolean replace(String table, String nullColumnHack, ContentValues initialValues) {
        try {
            mDatabase.replace(table, nullColumnHack, initialValues);
            return true;
        } catch (SQLException e) {
            return false;
        }
    }

    /**
     * Method for deleting rows in the database.
     *
     * @param table       String: the table to delete from.
     * @param whereClause String: the optional WHERE clause to apply when deleting. Passing null
     *                    will delete all rows.
     * @param whereArgs   String: You may include ?s in the where clause, which will be replaced by
     *                    the values from whereArgs. The values will be bound as Strings.
     * @return the number of rows affected if a whereClause is passed in, 0 otherwise.
     * To remove all rows and get a count pass "1" as the whereClause.
     */
    public boolean delete(String table, String whereClause, String[] whereArgs) {
        try {
            mDatabase.delete(table, whereClause, whereArgs);
            return true;
        } catch (SQLException e) {
            return false;
        }
    }
}
```

DatabaseHelper.java

```java
package com.kasapakis.adam.plugin_model_library_host.database;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

/**
 * Provides a database object or if it the first time called it creates a new one.
 */

public class DatabaseHelper extends SQLiteOpenHelper {
    private static final int VERSION = 1;
    private static final String DATABASE_NAME = "DataBase.db";

    public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase) {
    }

    @Override
    public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int i1) {
    }
```

```
}
```

Plugin.java

```
package com.kasapakis.adam.plugin_model_library_host.plugin_manager;

/**
 * Class that represents a plug-in.
 */
public class Plugin {

    private String name;
    private String type;
    private int apiLevel = 0;
    private int pluginVersion = 0;
    private int databaseVersion = 0;
    private boolean isActivated = false;
    private boolean isInitialised = false;
    private String activityPackage = null;
    private String appPackage;

    public Plugin() {
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public boolean isInitialised() {
        return isInitialised;
    }

    public void setInitialised(boolean b) {
        isInitialised = b;
    }

    public boolean isActivated() {
        return isActivated;
    }

    public void setActivated(boolean activated) {
        isActivated = activated;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPluginVersion() {
        return pluginVersion;
    }

    public void setPluginVersion(int pluginVersion) {
        this.pluginVersion = pluginVersion;
    }

    public int getDatabaseVersion() {
        return databaseVersion;
    }
```

```java
    public void setDatabaseVersion(int databaseVersion) {
        this.databaseVersion = databaseVersion;
    }

    public String getActivityPackage() {
        return activityPackage;
    }

    public void setActivityPackage(String activityPackage) {
        this.activityPackage = activityPackage;
    }

    public String getAppPackage() {
        return appPackage;
    }

    public void setAppPackage(String appPackage) {
        this.appPackage = appPackage;
    }

    public int getApiLevel() {
        return apiLevel;
    }

    public void setApiLevel(int apiLevel) {
        this.apiLevel = apiLevel;
    }
}
```

PluginScanner.java

```java
package com.kasapakis.adam.plugin_model_library_host.plugin_manager;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;

import java.util.ArrayList;
import java.util.List;

/**
 * class that scans for plugins installed in the device.
 * the class extends BroadcastReceiver, and receives
 * <p>android.intent.action.PACKAGE_INSTALL</p>
 * <p>android.intent.action.PACKAGE_ADDED</p>
 * <p>android.intent.action.PACKAGE_REPLACED</p>
 * <p>android.intent.action.PACKAGE_REMOVED</p>
 * <p>pluginmodel.pluginmanager.action.FIND_PLUGINS</p>
 */
public class PluginScanner extends BroadcastReceiver implements Runnable {

    private static final String UPDATED_ACTION = "pluginmodel.pluginmanager.action.PLUGINS_UPDATED";
    private static final String INTENT_ACTION = "android.intent.action.PLUGIN";
    private static final int apiVersion = 1;
    private Context context;

    public PluginScanner() {
    }

    public PluginScanner(Context context) {
        this.context = context;
    }

    /**
     * Method used to detect plug-in on the device.
```

```java
     *
     * @param context
     */
    private void findPlugins(Context context) {
        List<ResolveInfo> pluginServices = context.getPackageManager().queryIntentServices(
                new Intent(INTENT_ACTION), PackageManager.GET_META_DATA);
        ArrayList<Plugin> plugins = new ArrayList<>();
        for (ResolveInfo pluginService : pluginServices) {
            if (pluginService.serviceInfo.metaData.getInt("api_level", 0) != apiVersion) {
                continue;
            }
            Plugin plugin = new Plugin();
            plugin.setApiLevel(pluginService.serviceInfo.metaData.getInt("api_level", 0));
            plugin.setName(pluginService.serviceInfo.metaData.getString("plugin_name", null));
            plugin.setType(pluginService.serviceInfo.metaData.getString("type", null));
            plugin.setActivityPackage(
                    pluginService.serviceInfo.metaData.getString("activity_intent", "not found"));
            plugin.setAppPackage(pluginService.serviceInfo.packageName);
            plugin.setPluginVersion(pluginService.serviceInfo.metaData.getInt("version", 0));
            plugin.setDatabaseVersion(pluginService.serviceInfo.metaData.getInt("dbVersion", 0));
            plugins.add(plugin);
        }
        Registry.getInstance().setPlugins(plugins, context);
        Registry.getInstance().removeUninstalledPlugins(plugins, context);
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        new Thread(new PluginScanner(context)).run();
    }

    private void broadcast() {
        Intent i = new Intent();
        i.setAction(UPDATED_ACTION);
        context.sendBroadcast(i);
    }

    @Override
    public void run() {
        findPlugins(context);
        broadcast();
    }
}
```

Registry.java

```java
package com.kasapakis.adam.plugin_model_library_host.plugin_manager;

import android.app.Application;
import android.content.Context;
import android.content.Intent;

import com.kasapakis.adam.plugin_model_library_core.database.schema.Schema;

import java.util.ArrayList;

/**
 * Holds plugin objects and information about installed plugins and information about plugin's
 * database structure.
 * Provides to access plugin's activated and initialised status, and methods to modify them.
 * Provides methods to read and modify plugin's databse schema structure.
 */
public class Registry extends Application {
    private static final Registry ourInstance = new Registry();

    /**
     * intent received by plugin scanner to start plugin detection.
```

```java
     */
    private static final String FIND_PLUGINS_ACTION = "pluginmodel.pluginmanager.action" +
            ".FIND_PLUGINS";

    private PluginSaver saver;

    private Registry() {
    }

    /**
     * Returns and instance of Registry singleton
     *
     * @return Registry instance
     */
    public static Registry getInstance() {
        return ourInstance;
    }

    /**
     * Fires intent to notify receiver to detect plugins.
     *
     * @param context application context
     */
    public void searchPlugins(Context context) {
        Intent i = new Intent();
        i.setAction(FIND_PLUGINS_ACTION);
        context.sendBroadcast(i);
    }

    /**
     * add plugin to registry
     *
     * @param plugin  plugin object to be added
     * @param context application context
     */
    public void addPlugin(Plugin plugin, Context context) {
        saver = new PluginSaver(context);
        if (saver.isPluginSaved(plugin.getName())) { //existing or updated plugin
            if (plugin.getPluginVersion() > saver.loadPlugin(plugin.getName()).getPluginVersion()) {
                //upgraded plugin
                plugin.setActivated(saver.loadPlugin(plugin.getName()).isActivated());

                if (plugin.getDatabaseVersion() == saver.getSchema(plugin.getName()).getVersion()) {
                    //higher plugin version but same db version
                    plugin.setInitialised(true);
                }
                savePlugin(plugin, context);
            }
        } else { //newly installed plugin
            if (saver.getSchema(plugin.getName()) != null) {
                if (plugin.getDatabaseVersion() == saver.getSchema(plugin.getName()).getVersion()
                        ) {
                    plugin.setInitialised(true);
                }
            }
            savePlugin(plugin, context);
        }
    }

    /**
     * Adds multiple plugins to registry
     *
     * @param plugins ArrayList of plugins to be added
     * @param context application context
     */
    public void setPlugins(ArrayList<Plugin> plugins, Context context) {
        saver = new PluginSaver(context);
```

```
        ArrayList<Plugin> savedPlugins = saver.loadPlugins();
        for (Plugin plugin : plugins) {
            addPlugin(plugin, context);
        }
    }

    /**
     * Method to remove no longer installed plugins
     *
     * @param plugins ArrayList of currently detected plugins
     * @param context Context: application context
     */
    public void removeUninstalledPlugins(ArrayList<Plugin> plugins, Context context) {
        saver = new PluginSaver(context);
        saver.removeUninstalledPlugins(plugins);
    }

    /**
     * Method to modify activated state of a plugin in memory.
     *
     * @param plugin  Plugin: plugin to be modified
     * @param b       boolean: new status
     * @param context Context: application context
     */
    public void setActivated(Plugin plugin, boolean b, Context context) {
        Plugin temp = getPlugin(plugin.getName(), context);
        temp.setActivated(b);
        savePlugin(temp, context);

    }

    /**
     * method to modify initialised state of a plugin
     *
     * @param pluginName String: plugin to be modified
     * @param b          boolean: new status
     * @param context    Context: application context
     */
    public void setInitialised(String pluginName, Boolean b, Context context) {
        saver = new PluginSaver(context);
        Plugin temp = saver.loadPlugin(pluginName);
        temp.setInitialised(b);
        saver.savePlugin(temp);
    }

    /**
     * Saves a plugin object in memory.
     *
     * @param plugin  Plugin: plugin to be saved.
     * @param context Context: application context.
     */
    public void savePlugin(Plugin plugin, Context context) {
        saver = new PluginSaver(context);
        saver.savePlugin(plugin);
    }

    /**
     * Check if a plugin is installed
     *
     * @param pluginName String: plugin name to check for
     * @param context    Context: application context
     * @return boolean: installed status
     */
    public boolean isInstalled(String pluginName, Context context) {
        return new PluginSaver(context).isPluginSaved(pluginName);
    }
```

```java
/**
 * check if a plugin is initialised
 *
 * @param pluginName String: plugin name to check for
 * @param context    Context: application context
 * @return boolean: installed status
 */
public boolean isInitialised(String pluginName, Context context) {
    Plugin p = new PluginSaver(context).loadPlugin(pluginName);
    if (p == null) {
        return false;
    } else {
        return p.isInitialised();
    }
}

/**
 * check if plugin is activated
 *
 * @param pluginName String: plugin name to check for
 * @param context    Context: application context
 * @return boolean: activated status
 */
public boolean isActivated(String pluginName, Context context) {
    return new PluginSaver(context).loadPlugin(pluginName).isActivated();
}

/**
 * returns a Plugin object given its name
 *
 * @param pluginName String: plugin's name to return
 * @param context    Context: application context
 * @return Plugin object
 */
public Plugin getPlugin(String pluginName, Context context) {
    return new PluginSaver(context).loadPlugin(pluginName);
}

/**
 * return all installed plugins
 *
 * @param context Context: application context
 * @return ArrayList of plugins installed
 */
public ArrayList<Plugin> getPlugins(Context context) {
    return new PluginSaver(context).loadPlugins();
}

/**
 * returns names of plugins installed
 *
 * @param context Context: application context
 * @return ArrayList with names of the installed plugins
 */
public ArrayList<String> getPluginNames(Context context) {
    ArrayList<String> pluginNames = new ArrayList<>();
    for (Plugin plugin : getPlugins(context)) {
        pluginNames.add(plugin.getName());
    }
    return pluginNames;
}

/**
 * returns all activated plugins
 *
 * @param context Context: application context
 * @return ArrayList of Plugin objects containing all activated plugins
```

```
     */
    public ArrayList<Plugin> getActivatedPlugins(Context context) {
        saver = new PluginSaver(context);
        ArrayList<Plugin> activated = new ArrayList<>();
        for (Plugin plugin : saver.loadPlugins()) {
            if (plugin.isActivated()) {
                activated.add(plugin);
            }
        }
        return activated;
    }

    /**
     * returns the version of schema of a Plugin
     *
     * @param pluginName String: name of the plugin
     * @param context    Context: application context
     * @return schema version
     */
    public int getSchemaVersion(String pluginName, Context context) {
        saver = new PluginSaver(context);
        if (saver.isPluginSaved(pluginName)) {
            return saver.loadPlugin(pluginName).getDatabaseVersion();
        }
        return 0;
    }
    //database

    /**
     * add a Schema object to registry
     *
     * @param schema  Schema: object to be added
     * @param context Context: application context
     */
    public void addSchema(Schema schema, Context context) {
        new PluginSaver(context).saveSchema(schema);
    }

    /**
     * returns the corresponding Schema object
     *
     * @param schemaName String: name of schema to return
     * @param context    Context: application context
     * @return Schema object
     */
    public Schema getSchema(String schemaName, Context context) {
        return new PluginSaver(context).getSchema(schemaName);
    }

    /**
     * return all schema objects
     *
     * @param context Context: application context
     * @return ArrayList containing Schema objects
     */
    public ArrayList<Schema> getSchemas(Context context) {
        return new PluginSaver(context).getSchemas();
    }

    /**
     * returns the names of all Schema objects
     *
     * @param context Context: application context
     * @return ArrayList containing names of Schema objects
     */
    public ArrayList<String> getSchemaNames(Context context) {
        ArrayList<String> names = new ArrayList<>();
```

```java
        ArrayList<Schema> schemas = new PluginSaver(context).getSchemas();
        for (Schema schema : schemas) {
            names.add(schema.getName());
        }
        return names;
    }
}
```

PluginSaver.java

```java
package com.kasapakis.adam.plugin_model_library_host.plugin_manager;

import android.content.Context;
import android.content.SharedPreferences;

import com.google.gson.Gson;
import com.kasapakis.adam.plugin_model_library_core.database.schema.Schema;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import static android.content.Context.MODE_PRIVATE;

/**
 * Class used to save Plugin and Schema objects to shared preferences memory.
 */
public class PluginSaver {
    private SharedPreferences pluginSharedPrefs;
    private SharedPreferences databaseSharedPrefs;


    public PluginSaver(Context context) {
        pluginSharedPrefs = context.getSharedPreferences("pluginSharedPrefs", MODE_PRIVATE);
        databaseSharedPrefs = context.getSharedPreferences("databaseSharedPrefs", MODE_PRIVATE);
    }

    /**
     * Removes the list of plugins that are passed as parameter from the shared preferences.
     *
     * @param plugins list of plugins to remove.
     */
    public void removeUninstalledPlugins(List<Plugin> plugins) {
        Map<String, ?> savedPlugins = pluginSharedPrefs.getAll();
        ArrayList<String> toRemove;
        for (Map.Entry<String, ?> entry : savedPlugins.entrySet()) {
            boolean isInstalled = false;
            for (int i = 0; i < plugins.size(); i++) {
                if (entry.getKey().equals(plugins.get(i).getName())) {
                    isInstalled = true;
                    break;
                }
            }
            if (!isInstalled) {
                removeSavedPlugin(entry.getKey());
            }
        }
    }

    /**
     * Check if a specific Plugin object is saved to SharedPreferences.
     *
     * @param name String: Name of the Plugin to check upon.
     * @return true if plugin is saved, false if not saved.
     */
    public Boolean isPluginSaved(String name) {
        if (pluginSharedPrefs.contains(name)) {
            return true;
```

```
        } else {
            return false;
        }
    }

    /**
     * Save a Plugin object to SharedPreferences in JSON form.
     *
     * @param plugin Plugin: object to save.
     */
    public void savePlugin(Plugin plugin) {
        Gson gson = new Gson();
        String json = gson.toJson(plugin);
        pluginSharedPrefs
                .edit()
                .putString(plugin.getName(), json)
                .apply();
    }

    /**
     * Remove a Plugin object from SharedPreferences.
     *
     * @param name String: Name of the Plugin to remove.
     */
    public void removeSavedPlugin(String name) {
        pluginSharedPrefs
                .edit()
                .remove(name)
                .apply();
    }

    /**
     * Load the plugin object from SharedPreferences by passing the name of the Plugin.
     *
     * @param name String: Name of the Plugin to return.
     * @return Plugin object.
     */
    public Plugin loadPlugin(String name) {
        Gson gson = new Gson();
        String json = pluginSharedPrefs.getString(name, null);
        return gson.fromJson(json, Plugin.class);
    }

    /**
     * Load the saved plug-in objects.
     *
     * @return ArrayList with Plugin objects.
     */
    public ArrayList<Plugin> loadPlugins() {
        Gson gson = new Gson();
        ArrayList<Plugin> plugins = new ArrayList<>();
        Map<String, ?> pluginsMap = pluginSharedPrefs.getAll();

        for (Map.Entry<String, ?> entry : pluginsMap.entrySet()) {
            plugins.add(gson.fromJson((String) entry.getValue(), Plugin.class));
        }
        return plugins;
    }

    //database

    /**
     * Saves a Schama object to shared preferences
     *
     * @param schema Schema: object to save.
     */
    public void saveSchema(Schema schema) {
```

```java
        Gson gson = new Gson();
        String json = gson.toJson(schema);
        databaseSharedPrefs
                .edit()
                .putString(schema.getName(), json)
                .apply();
    }

    /**
     * Returns a Schema object given its name.
     *
     * @param schemaName String: name of Schema to return.
     * @return Schema object.
     */
    public Schema getSchema(String schemaName) {
        Gson gson = new Gson();
        String json = pluginSharedPrefs.getString(schemaName, null);
        return gson.fromJson(json, Schema.class);
    }

    /**
     * Returns an array list containing all Schema objects saved in Shared Preferences.
     *
     * @return ArrayList of Schema objects.
     */
    public ArrayList<Schema> getSchemas() {
        Gson gson = new Gson();
        ArrayList<Schema> schemas = new ArrayList<>();
        Map<String, ?> databaseMap = databaseSharedPrefs.getAll();

        for (Map.Entry<String, ?> entry : databaseMap.entrySet()) {
            schemas.add(gson.fromJson((String) entry.getValue(), Schema.class));
        }
        return schemas;
    }

    /**
     * Removes a Schema object from the Shared preferences.
     *
     * @param schemaName String: name of the Schema to remove.
     */
    public void removeSchema(String schemaName) {
        databaseSharedPrefs
                .edit()
                .remove(schemaName)
                .apply();
    }
}
```

PluginDetectedReceiver.java

```java
package com.kasapakis.adam.plugin_model_library_host.plugin_manager;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;

/**
 * Broadcast receiver class that receives "pluginmodel.pluginmanager.action.PLUGINS_UPDATED" when
 * PluginScanner has finished detecting new plug-ins.
 */
public class PluginDetectedReceiver extends BroadcastReceiver {
    public static final String CUSTOM_INTENT = "pluginmodel.pluginmanager.action.PLUGINS_UPDATED";

    @Override
    public void onReceive(Context context, Intent intent) {
```

```java
    }

    /**
     * Returns the intent filter object that is used to receive when plugins are detected.
     *
     * @return IntentFilter.
     */
    public IntentFilter getIntentFilter() {
        IntentFilter packageFilter = new IntentFilter();
        packageFilter.addAction(CUSTOM_INTENT);
        return packageFilter;
    }
}
```

TileRecyclerView.java

```java
package com.kasapakis.adam.plugin_model_library_host.ui;

import android.content.Context;
import android.content.pm.PackageManager;
import android.graphics.drawable.Drawable;
import android.support.v7.widget.RecyclerView;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.TextView;

import com.kasapakis.adam.plugin_model_library_host.R;
import com.kasapakis.adam.plugin_model_library_host.plugin_manager.Plugin;

import java.util.ArrayList;

/**
 * Adapter class that represents Plugin objects as Tiles in a RecyclerView.
 */

public class TileRecyclerView extends RecyclerView.Adapter<TileRecyclerView.ViewHolder> {

    private ArrayList<Plugin> plugins = new ArrayList<>();
    private LayoutInflater mInflater;
    private ItemClickListener mClickListener;
    private Context context;

    /**
     * Constructor
     *
     * @param context Context: Application context
     * @param plugins ArrayList of Plugin.
     */
    public TileRecyclerView(Context context, ArrayList<Plugin> plugins) {
        this.mInflater = LayoutInflater.from(context);
        this.plugins = plugins;
        this.context = context;
    }

    // inflates the cell layout from xml when needed
    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = mInflater.inflate(R.layout.tile_item, parent, false);
        ViewHolder viewHolder = new ViewHolder(view);
        return viewHolder;
    }

    // binds the data to the textview in each cell
    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
```

```
        String name = plugins.get(position).getName();
        holder.myTextView.setText(name);
        Drawable icon = null;
        try {
            icon = context.getPackageManager().getApplicationIcon(plugins.get(position)
                    .getAppPackage());
        } catch (PackageManager.NameNotFoundException e) {
            e.printStackTrace();
        }
        if (icon != null) {
            holder.tile_image.setImageDrawable(icon);
        }
    }

    /**
     * @return number of items in the recycler view.
     */
    @Override
    public int getItemCount() {
        if (plugins == null) {
            return 0;
        }
        return plugins.size();
    }

    /**
     * Sets Plugin objects to be displayed in the recycler view.
     *
     * @param plugins ArrayList of Plugin.
     */
    public void setPlugins(ArrayList<Plugin> plugins) {
        this.plugins = plugins;
        notifyDataSetChanged();
    }

    /**
     * Convenience method for getting Plugin object according to its position in the Adapter.
     *
     * @param id int: position in the adapter.
     * @return Plugin
     */
    public Plugin getItem(int id) {
        return plugins.get(id);
    }

    /**
     * Sets TileRecyclerView.OnClickListener to respond to clicks.
     *
     * @param itemClickListener OnClickListener
     */
    public void setClickListener(ItemClickListener itemClickListener) {
        this.mClickListener = itemClickListener;
    }

    /**
     * Interface that defines onClickListener().
     * Implement this interface's method to respond to clicks.
     */
    public interface ItemClickListener {
        void onItemClick(View view, int position);
    }

    // stores and recycles views as they are scrolled off screen
    public class ViewHolder extends RecyclerView.ViewHolder implements View.OnClickListener {
        public TextView myTextView;
        public ImageView tile_image;
```

```java
        public ViewHolder(View itemView) {
            super(itemView);
            tile_image = (ImageView) itemView.findViewById(R.id.tile_image);
            myTextView = (TextView) itemView.findViewById(R.id.info_text);
            itemView.setOnClickListener(this);
        }

        @Override
        public void onClick(View view) {
            if (mClickListener != null) mClickListener.onItemClick(view, getAdapterPosition());
        }
    }
}
```

PluginRecyclerView.java

```java
package com.kasapakis.adam.plugin_model_library_host.ui;

import android.content.Context;
import android.support.v7.widget.RecyclerView;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.CompoundButton;
import android.widget.Switch;

import com.kasapakis.adam.plugin_model_library_host.R;
import com.kasapakis.adam.plugin_model_library_host.plugin_manager.Plugin;

import java.util.ArrayList;

/**
 * Adapter class that represents Plugin objects as switches in a RecyclerView.
 */

public class PluginRecyclerView extends RecyclerView.Adapter<PluginRecyclerView.ViewHolder> {
    private ArrayList<Plugin> plugins = new ArrayList<>();
    private LayoutInflater mInflater;
    private PluginRecyclerView.OnCheckedChangedListener mItemCheckedListener;

    /**
     * Constructor
     *
     * @param context Context: Application context
     * @param plugins ArrayList of Plugin.
     */
    public PluginRecyclerView(Context context, ArrayList<Plugin> plugins) {
        this.mInflater = LayoutInflater.from(context);
        this.plugins = plugins;
    }

    // inflates the cell layout from xml when needed
    @Override
    public PluginRecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = mInflater.inflate(R.layout.plugin_item, parent, false);
        PluginRecyclerView.ViewHolder viewHolder = new PluginRecyclerView.ViewHolder(view);
        return viewHolder;
    }

    // binds the data to the textview in each cell
    @Override
    public void onBindViewHolder(PluginRecyclerView.ViewHolder holder, int position) {
        String name = plugins.get(position).getName();
        holder.aSwitch.setText(name);
        holder.aSwitch.setChecked(plugins.get(position).isActivated());
    }
```

```java
    /**
     * @return number of items in the recycler view.
     */
    @Override
    public int getItemCount() {
        return plugins.size();
    }

    /**
     * Sets Plugin objects to be displayed in the recycler view.
     *
     * @param plugins ArrayList of Plugin.
     */
    public void setPlugins(ArrayList<Plugin> plugins) {
        this.plugins = plugins;
        notifyDataSetChanged();
    }

    /**
     * Convenience method for getting Plugin object according to its position in the Adapter.
     *
     * @param id int: position in the adapter.
     * @return Plugin
     */
    public Plugin getItem(int id) {
        return plugins.get(id);
    }

    /**
     * Sets PluginRecyclerView.OnCheckedChangeListener to respond to switch changes.
     *
     * @param itemCheckedListener OnCheckedChangeListener
     */
    public void setOnCheckedChangedListener(PluginRecyclerView.OnCheckedChangedListener
                                                    itemCheckedListener) {
        this.mItemCheckedListener = itemCheckedListener;
    }

    /**
     * Interface that defines onCheckChangedListener().
     * Implement this interface's method to respond to changes on switches.
     */
    public interface OnCheckedChangedListener {
        void onCheckedChanged(CompoundButton cb, boolean on, int position);
    }

    // stores and recycles views as they are scrolled off screen
    public class ViewHolder extends RecyclerView.ViewHolder implements CompoundButton
            .OnCheckedChangeListener {
        public Switch aSwitch;

        public ViewHolder(View itemView) {
            super(itemView);
            aSwitch = (Switch) itemView.findViewById(R.id.plugin_item_name);
            aSwitch.setOnCheckedChangeListener(this);
        }

        @Override
        public void onCheckedChanged(CompoundButton compoundButton, boolean b) {
            if (mItemCheckedListener != null)
                mItemCheckedListener.onCheckedChanged(compoundButton, b, getAdapterPosition());
        }
    }
}
```

tile_item.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:layout_margin="10dp"
    android:background="@android:color/background_light"
    android:orientation="vertical">

    <ImageView
        android:id="@+id/tile_image"
        android:layout_width="66dp"
        android:layout_height="66dp"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.0"
        tools:layout_constraintBottom_creator="1"
        tools:layout_constraintLeft_creator="1"
        tools:layout_constraintRight_creator="1"
        tools:layout_constraintTop_creator="1" />

    <TextView
        android:id="@+id/info_text"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:gravity="center"
        android:padding="0dp"
        android:text="TextView"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="1.0"
        tools:layout_constraintLeft_creator="1"
        tools:layout_constraintRight_creator="1"
        tools:layout_constraintTop_creator="1" />
</android.support.constraint.ConstraintLayout>
```

plugin_item.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <Switch
        android:id="@+id/plugin_item_name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:text="Switch" />
</LinearLayout>
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.kasapakis.adam.plugin_model_library_host">

    <application
        android:allowBackup="true"
        android:label="@string/app_name"
        android:supportsRtl="true">

        <service
            android:name=".service.HostService"
            android:enabled="true"
            android:exported="true"
            android:permission="com.kasapakis.adam.plugin_model_library.permission
                                .HOST_SERVICE_PERMISSION"
            />

        <receiver
            android:name="com.kasapakis.adam.plugin_model_library_host.plugin_manager.PluginScanner"
            android:enabled="true"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.PACKAGE_INSTALL"/>
                <action android:name="android.intent.action.PACKAGE_ADDED"/>
                <action android:name="android.intent.action.PACKAGE_REPLACED"/>
                <action android:name="android.intent.action.PACKAGE_REMOVED"/>

                <data android:scheme="package"/>
            </intent-filter>
            <intent-filter>
                <action android:name="pluginmodel.pluginmanager.action.FIND_PLUGINS"/>
            </intent-filter>
        </receiver>
        <receiver
            android:name="com.kasapakis.adam.plugin_model_library_host.plugin_manager
                                .PluginDetectedReceiver"
            android:enabled="true"
            android:exported="false"/>

        <activity
            android:name="com.kasapakis.adam.plugin_model_library_host.ui.UninstallActivity"
            android:label="@string/uninstall_plugins_activity_name"/>
        <activity
            android:name="com.kasapakis.adam.plugin_model_library_host.ui.ManagePluginsActivity"
            android:label="@string/manage_plugins_activity_name"/>
    </application>

</manifest>
```

d) Host Application

Host.java

```
package com.kasapakis.adam.pluginmodeldemo;

import android.content.Context;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
```

```java
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.GridLayoutManager;
import android.support.v7.widget.LinearLayoutManager;
import android.support.v7.widget.RecyclerView;
import android.support.v7.widget.Toolbar;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.Button;
import android.widget.CompoundButton;
import android.widget.Toast;

import com.gordonwong.materialsheetfab.MaterialSheetFab;
import com.kasapakis.adam.plugin_model_library_host.plugin_manager.Plugin;
import com.kasapakis.adam.plugin_model_library_host.plugin_manager.PluginDetectedReceiver;
import com.kasapakis.adam.plugin_model_library_host.plugin_manager.Registry;
import com.kasapakis.adam.plugin_model_library_host.ui.ManagePluginsActivity;
import com.kasapakis.adam.plugin_model_library_host.ui.PluginRecyclerView;
import com.kasapakis.adam.plugin_model_library_host.ui.TileRecyclerView;

import java.util.ArrayList;

public class Host extends AppCompatActivity {
    TileRecyclerView tileAdapter;
    PluginRecyclerView pluginAdapter;
    RecyclerView tileRecyclerView;
    RecyclerView pluginRecyclerView;

    PluginDetectedReceiver pluginDetectedReceiver = new PluginDetectedReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            super.onReceive(context, intent);
            setPluginAdapterContents();
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_host);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        //search for plug-ins when activity is created.
        Registry.getInstance().searchPlugins(getApplicationContext());

        //load plug-ins.
        ArrayList<Plugin> plugins = Registry.getInstance().getPlugins(getApplicationContext());

        //set up the pluginRecyclerView
        pluginRecyclerView = (RecyclerView) findViewById(R.id.plugin_recycler_view);
        pluginRecyclerView.setLayoutManager(
                new LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false));

        pluginAdapter = new PluginRecyclerView(this, plugins);

        //set listener to respond to changes whenever a plug-in gets activated/de-activated.
        pluginAdapter.setOnCheckedChangedListener(
                new PluginRecyclerView.OnCheckedChangedListener() {
                    @Override
                    public void onCheckedChanged(CompoundButton cb, boolean on, int position) {
                        Plugin plugin = pluginAdapter.getItem(position);
                        Registry.getInstance().setActivated(plugin, on, getApplicationContext());
                        tileAdapter.setPlugins(Registry.getInstance().getActivatedPlugins
                                (getApplicationContext()));
                    }
                });
```

```
        pluginRecyclerView.setAdapter(pluginAdapter);

        // set up the RecyclerView for tiles
        tileRecyclerView = (RecyclerView) findViewById(R.id.tile_recycler_view);
        int numberOfColumns = 3;
        tileRecyclerView.setLayoutManager(new GridLayoutManager(this, numberOfColumns));

        tileAdapter = new TileRecyclerView(this, Registry.getInstance().getActivatedPlugins
                (getApplicationContext()));

        //set listener to respond to clicks on tiles to navigate to the plug-ins activity.
        tileAdapter.setClickListener(new TileRecyclerView.ItemClickListener() {
            @Override
            public void onItemClick(View view, int position) {
                String activityIntent = tileAdapter.getItem(position).getActivityPackage();
                Intent intent = new Intent();
                intent.setAction(activityIntent);
                startActivity(intent);
            }
        });
        tileRecyclerView.setAdapter(tileAdapter);

        // Setup floating action button
        Fab fab = (Fab) findViewById(R.id.fab);
        View sheetView = findViewById(R.id.fab_sheet);
        View overlay = findViewById(R.id.overlay);
        int sheetColor = getResources().getColor(R.color.cardPrimary);
        int fabColor = getResources().getColor(R.color.cardPrimary);

        // Initialize material sheet FAB, the container of the plugin switches.
        new MaterialSheetFab<>(fab, sheetView, overlay,
                sheetColor, fabColor);
        //setup refresh button
        Button refresh = (Button) findViewById(R.id.refresh_button);
        refresh.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Registry.getInstance().searchPlugins(getApplicationContext());
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu;
        // his adds items to the action bar.
        getMenuInflater().inflate(R.menu.menu_host, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();

        if (id == R.id.action_manage_plugins) {
            Intent settingsIntent =
                    new Intent(getApplicationContext(), ManagePluginsActivity.class);
            startActivity(settingsIntent);
        }
        if (id == R.id.action_play_store) {
            try {
                String search_query = "";
                Intent intent = new Intent(Intent.ACTION_VIEW);
                intent.setData(Uri.parse("market://search?q=" + search_query + "&c=apps"));
                startActivity(intent);
            } catch (Exception e) {
                Toast.makeText(Host.this, "PlayStore not found!", Toast.LENGTH_SHORT).show();
```

```java
            }
        }
        return super.onOptionsItemSelected(item);
    }

    @Override
    protected void onResume() {
        super.onResume();
        //set plugins before the activity is displayed.
        setPluginAdapterContents();
        //register receiver to be notified whenever pluginScanner detected new plug-ins.
        registerReceiver(pluginDetectedReceiver, pluginDetectedReceiver.getIntentFilter());
    }


    @Override
    protected void onPause() {
        //Unregister pluginDetectedReceiver.
        unregisterReceiver(pluginDetectedReceiver);
        super.onPause();
    }

    /**
     * Sets Plugin objects to pluginAdapter and tileAdapter.
     */
    public void setPluginAdapterContents() {
        pluginAdapter.setPlugins(
                Registry.getInstance().getPlugins(getApplicationContext()));
        tileAdapter.setPlugins(
                Registry.getInstance().getActivatedPlugins(getApplicationContext()));
    }
}
```

MyServiceHost.java

```java
package com.kasapakis.adam.pluginmodeldemo;

import android.content.Intent;

import com.kasapakis.adam.plugin_model_library_host.service.HostService;

public class MyServiceHost extends HostService {

    public MyServiceHost() {
    }

    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        return START_STICKY;
    }
}
```

activity_host.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context="com.kasapakis.adam.pluginmodeldemo.Host">
```

```xml
    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_host" />

    <com.kasapakis.adam.pluginmodeldemo.Fab
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        app:srcCompat="@android:drawable/ic_input_add" />

    <!-- Overlay that dims the screen -->
    <com.gordonwong.materialsheetfab.DimOverlayFrameLayout
        android:id="@+id/overlay"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <!-- Circular reveal container for the sheet -->
    <io.codetail.widget.RevealLinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="end|bottom"
        android:orientation="vertical">

        <!-- Sheet that contains your items -->
        <android.support.v7.widget.CardView
            android:id="@+id/fab_sheet"
            android:layout_width="250dp"
            android:layout_height="wrap_content">

            <android.support.v7.widget.RecyclerView
                android:id="@+id/plugin_recycler_view"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:layout_alignParentStart="true"
                android:paddingBottom="50dp">

            </android.support.v7.widget.RecyclerView>

            <Button
                android:id="@+id/refresh_button"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_gravity="bottom"
                android:background="?android:attr/selectableItemBackground"
                android:text="@string/refresh_button" />

        </android.support.v7.widget.CardView>
    </io.codetail.widget.RevealLinearLayout>

</android.support.design.widget.CoordinatorLayout>
```

content_host.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/content_host"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:weightSum="3"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="com.kasapakis.adam.pluginmodeldemo.Host"
    tools:showIn="@layout/activity_host">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/tile_recycler_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```

AndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.kasapakis.adam.pluginmodeldemo">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".Host"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service
            android:name=".MyServiceHost"
            android:enabled="true"
            android:exported="true" />

        <activity android:name="com.kasapakis.adam.plugin_model_library_host.ui.UninstallActivity" />
    </application>
</manifest>
```