Utrecht University
Department of Information and Computing Sciences

Towards architecture documentation for Android apps

Bachelor Thesis

Yorick van Zweeden

Supervisors:
dr. ir. J.M.E.M. van der Werf
drs. J.D. Fokker

1.0

Utrecht, August 2018

Abstract

Android is the largest mobile platform and offers millions of apps. Just as in traditional software engineering, apps benefit from good software architecture. The conducted literature study revealed that no method for documenting Android app architecture currently exists. This thesis sets out to create such a method and identify the fundamental concepts and mechanisms required for documenting Android app architecture. Using the design science research method, the objectives for a solution were established and an artefact was produced. The artefact consists of views at three layers of different abstraction levels. Views at each layer consist of a different set of elements and address different concerns. Demonstration of the artefact was conducted using two case studies and further evaluated. The research has successfully identified viable methods for documenting Android app architecture and encourages future work into this field to improve the artefact.

Preface

I would like to thank Jan Martijn van der Werf for the patience with my slow start and all the support I have received in the last couple of months. Without him, this research would not have gotten as far as it did. Furthermore, I would like to thank Jeroen Fokker for finding time to read my thesis in his recess. Additionally, I would like to thank Rens Rooimans for proofreading the document. Lastly, I would like to thank Paul van Riet for motivating me during our work sessions.

Contents

Contents vii

List of Figures ix

List of Tables x

Listings xi

1 Introduction 1

2 Research Approach 3
- 2.1 Research questions . . . 3
- 2.2 Research methods . . . 3
  - 2.2.1 Conducting a literature study: Snowballing . . . 4
  - 2.2.2 Design Science Research . . . 4

3 Literature Study: Software Architecture 6
- 3.1 What is software architecture? . . . 6
- 3.2 Documenting software architecture . . . 6
- 3.3 Pros and cons of using viewpoints . . . 7

4 Literature Study: Android 9
- 4.1 Platform architecture . . . 9
- 4.2 Android application fundamentals . . . 9
  - 4.2.1 Communication between components . . . 10
  - 4.2.2 App components in detail . . . 10
  - 4.2.3 Architecture components . . . 12
- 4.3 Research into Android app architecture . . . 13

5 Artefact Design 15
- 5.1 Problem identification . . . 15
- 5.2 Defining the objectives . . . 15
  - 5.2.1 Applying viewpoints to Android . . . 15
  - 5.2.2 Objectives . . . 16
  - 5.2.3 Why current research does not meet the requirements . . . 17
- 5.3 The artefact . . . 17
  - 5.3.1 Modelling an example application . . . 18
  - 5.3.2 Artefact summary . . . 26
- 5.4 Creating architecture documentation for large applications . . . 28
  - 5.4.1 Finding the components . . . 28
  - 5.4.2 Finding the connectors . . . 29
  - 5.4.3 Combining components and connectors into views . . . 31

6 Artefact Demonstration & Evaluation **33**
6.1 Creating AD for the K-9 Mail app . . . 33
6.1.1 Introduction to K-9 Mail app . . . 33
6.1.2 Results . . . 33
6.1.3 Evaluation . . . 36
6.2 Creating AD for the OmniNotes app . . . 38
6.2.1 Introduction to OmniNotes . . . 38
6.2.2 Results . . . 38
6.2.3 Evaluation . . . 40
6.3 Evaluation . . . 43
6.3.1 Requirement 1: Providing the right level of detail independent of app size . 43
6.3.2 Requirement 2: Different views should not overlap or be too fragmented . . 43
6.3.3 Requirement 3: Reduce semantic scope with Android-specific elements . . . 44
6.3.4 Requirement 4: The views should address concerns . . . 44
6.3.5 Other remarks . . . 44

7 Conclusions, Limitations and Future Work **46**
7.1 Conclusions . . . 46
7.2 Limitations . . . 46
7.3 Future work . . . 47

Bibliography **49**

Appendix **53**

A Figures & Tables **53**
A.1 Code Layer view of HelloService . . . 53
A.2 Code Layer view of SecondActivity . . . 54
A.3 App Component Layer view of UpgradeDatabases . . . 55
A.4 Lines of Java code per application . . . 56

B Scripts **59**
B.1 ParseManifest Python script . . . 59
B.2 ParseHUSACCT Python script . . . 60

List of Figures

4.1 The model created by Bagheri et al. [1] . . . 13
4.2 The model created by Schmerl et al. [28] . . . 14

5.1 The viewpoint catalog of Rozanski & Woods . . . 16
5.2 Screenshots of the example application . . . 18
5.3 Code Layer view of MainActivity (code snippet of Listing 5.2) . . . 20
5.4 Code Layer view of an if/else statement . . . 21
5.5 Code Layer view of the HelloReceiver . . . 21
5.6 Code Layer view of MainActivity . . . 22
5.7 ACL view showing components of other modules referenced using the grey component type . . . 22
5.8 App Component Layer view of the example application . . . 23
5.9 ACL highlight of the explicit intent . . . 23
5.10 ACL highlight of the implicit intent . . . 24
5.11 ACL highlight of the RPC connector . . . 25
5.12 ACL highlight of the content resolver . . . 25
5.13 ACL highlight of the call/access connector . . . 26
5.14 Two FAML views that depict how FAML hierarchy works . . . 26
5.15 Overview of elements in the Code Layer . . . 27
5.16 Overview of elements in the App Component Layer . . . 27
5.17 Overview of elements in the Functional Architecture Model Layer . . . 27

6.1 Screenshots of the K-9 Mail app . . . 34
6.2 The FAML view of the K-9 mail app . . . 34
6.3 The ACL view of the Message Compose module . . . 35
6.4 The ACL view of the Load Messages module . . . 36
6.5 Screenshots of the OmniNotes app . . . 38
6.6 The FAML view of the OmniNotes app . . . 39
6.7 The ACL view of the Storage module . . . 39
6.8 Custom menu of the GalleryActivity . . . 40
6.9 The CL view of the GalleryActivity . . . 41
6.10 Multiple methods calling the HandleIntents method . . . 42
6.11 Static methods in the CL view . . . 42
6.12 Conflicting decompositions in the FAML views . . . 45

A.1 Code Layer view of HelloService of the example application . . . 53
A.2 Code Layer view of SecondActivity of the example application . . . 54
A.3 App Component Layer view of UpgradeDatabases (K9) . . . 55

List of Tables

4.1 Possible combinations of components and connectors 12

5.1 Testing current research for satisfying the four stated requirements 17

6.1 A comparison of both analysed apps 33

A.1 Lines of Java code per application. The list of applications is taken from Wikipedia's list of free and open-source applications for Android 58

Listings

5.1 The manifest of the example application 19
5.2 Code snippet of MainActivity.java of the example application 20
5.3 Code example of an explicit intent 23
5.4 Code example of an implicit intent 24
5.5 Code example of an RPC for binding to services 24
5.6 Code example of a content resolver 25
5.7 Code example of access/call connector 25
5.8 Code example of a context-registered broadcast receiver 28
5.9 Grep command for finding broadcast receivers 29
5.10 Example output of grep command 5.9 29
5.11 Example dependency of HUSACCT 30
5.12 Example dependency of HUSACCT matching the dependency in 5.11 30
5.13 Code example of content resolver undetected by HUSACCT 30
5.14 Example of an undetected content resolver dependency by HUSACCT 30
5.15 Grep command for finding content resolvers 30
5.16 Example of an *onBound* method declaration of a *ServiceConnection* 31
6.1 Code example of static method that fires an intent 37
B.1 Script used for parsing Android Manifest files 59
B.2 Script used for parsing HUSACCT dependencies 60

Chapter 1

Introduction

Although mobile apps tend to be smaller compared to traditional applications [31], mobile applications are becoming more complex and critical for businesses [34]. Mobile software engineering suffers from the same challenges as traditional software engineering and creating large-scale applications presents different problems. In order for mobile apps to scale up, appropriate techniques must be found for managing increasingly complex projects [34].

One aspect of software engineering that can play a key role in this process is software architecture. Software architecture may help to manage complexity and exposes high-level constraints on system design, as well as the rationale for making specific architectural choices. It may aid in the construction of software, as the architecture provides a partial blueprint for development [5].

Software architecture is captured in architectural descriptions (AD). This is a set of products that documents an architecture in a way its stakeholders can understand and demonstrate that the resulting architecture has met their concerns [27]. They are a helpful instrument to document and communicate architecture. This thesis will focus on creating architectural descriptions (AD) for Android apps. Research in mobile software engineering is not as developed yet compared to traditional software engineering [34]. As a result of that, no Android-specific architectural descriptions exist.

Creating Android-specific architectural descriptions is useful, as using domain-specific constructs are better suited for communication with users within the domain [4]. Android apps are highly constrained by the development framework which results in specific structures and the usage of a small set of components. This thesis tries to identify which elements are useful in communicating Android app architecture effectively.

Contributions

This thesis makes the following contributions:

- Provides an up-to-date overview of the architectural elements in Android apps. This is relevant as the Android platform is rapidly evolving.
- Identifies elements to model the architecture of Android apps. Modelling the right abstraction level is important to create clear and useful models.
- Creates a set of viewpoints which provide a starting point for communicating Android app architecture. Although this set does not cover all aspects of Android software design, it may provide a good start.

Outline

The thesis starts off in Chapter 2 with the research approach, which details the research questions and how these will be answered. This section is followed by a two-part literature study that

answers the research questions posed. Chapter 3 focuses on software architecture and Chapter 4 focuses on Android. In Chapter 5, the problem and the objectives for a solution will be stated. Additionally, the created artefact will be explained. The artefact will be demonstrated using two case studies and evaluated in Chapter 6. Finally, we conclude the thesis, explain the limitations, and give pointers for future work in Chapter 7.

Chapter 2

Research Approach

2.1 Research questions

The main research question that guides this thesis is:

MRQ: How can we document Android app architecture?

We will answer this question with two research questions. First, we want to have context on the current research on documenting Android app architecture.

RQ1: What is the current state of research in documenting Android app architecture?
This research question will be answered by looking at the following subquestions:

- What is software architecture?
- How can we document software architecture?
- How do Android apps work?
- What is the current state of research in documenting Android app architecture?

Once we have established context and an understanding of what has been researched, we try to make steps towards creating Android app architecture documentation. To guide that research, we ask:

RQ2: What are the fundamental concepts and mechanisms of architecture documentation for Android apps?
This research question will be answered by looking at the following subquestions:

- Which views are required to support these concepts?
- Which elements do these models contain?
- How can applications that are large and complex be modelled?

2.2 Research methods

Two research methods will be used: The first method is snowballing, which will be used for RQ1 and its subquestions. The second method is the design science research method, which will be applied to answer RQ2. We will describe both methods briefly.

2.2.1 Conducting a literature study: Snowballing

Snowballing is a search strategy to conduct a systematic literature review [37]. Wohlin [36] describes this in an earlier paper as a method to identify additional papers using the reference list of a paper or the citations to the paper. Our goal is not to produce a literature review, but rather to gain an understanding in previous research and provide context for our research contribution. Snowballing grants us a method to conduct a systematic search, which ensures a relatively complete census of relevant literature [35]. This research method consists of four steps as described by Wohlin [37]:

- *Identifying the research questions:* The research questions dictate the direction of the search process. The questions can be used to assert the relevance of the papers that are found.
- *Identify the starting set of papers:* To initiate the snowballing procedure, a starting set of papers is required. This set provides the basis of snowballing in which other research may be found.
- *Starting the snowballing procedure:* Two types of snowballing exists according to Wohlin [37]. The first type of snowballing is backward snowballing in which the reference list of identified paper is used to discover previous research. This contrasts forward snowballing in which newer papers citing the identified paper are discovered.
- *Examining found papers:* New papers are judged on relevance to the research and relevant papers are added to the set. The method is then applied to the new set of papers. If the procedure does not provide new papers, the method is completed.

The literature study focuses on three topics: Software architecture, Android, and software architecture in Android. Using snowballing, we identified the core principles of software architecture as detailed in Chapter 3. We started with some papers and books on architecture compliance checking and eventually stopped when we deemed to have obtained enough knowledge to answer our research questions. This method was also used for finding the current research on software architecture in Android. To gain an understanding of Android and applications, we mainly used the Android Developer Guides and applied forward snowballing to the links in the examined guides. Additionally, we created apps ourselves to gain a deeper understanding of the Android Development Framework.

2.2.2 Design Science Research

Design science research and behavioural science are two approaches for performing research in the Information Sciences. In [20], the authors explain that behavioural science approaches research "through the development and justification of theories explaining or predicting phenomena that occur". Alternatively, design science acquires knowledge and understanding by the creation and application of a designed artefact [20]. Artefacts are created, evaluated and may demonstrate the feasibility of the designed product [20].

Peffers, Tuunanen, Rothenberger & Chatterjee [25] have reviewed previous research on design science and subsequently developed a method that will be used in this thesis. Their method consists of six activities. Researchers are not expected to proceed with them in a linear fashion but may work at multiple activities at once. The activities are:

- *Problem identification and motivation:* A specific research problem is identified and the value of a solution is justified. The value justification motivates the researcher and audience to pursue the solution and accept the results.
- *Defining objectives for a solution:* The objectives of the solution are inferred from the problem definition and current research. They form the criteria for the evaluation. Peffers et al. [25] mention that some researchers transform the problem into explicit requirements, whereas others express objectives implicitly in the search for a relevant and important problem.

- *Design of the artefact:* Any designed object that inhibits a research contribution in its design, qualifies as an artefact. The artefact may solve unsolved problems or applies existing knowledge in new and innovative ways [20]. To enable implementation and application in an appropriate domain, the artefact should be described [20].
- *Demonstration:* The demonstration of the artefact proves the feasibility of the proposed solution. In this step, the proof may be derived from observational studies, analysis, experiments, testing or descriptive scenarios.
- *Evaluation:* The objectives defined in the second step guide the evaluation. The researcher must show how well the artefact supports a solution to the problem.
- *Communication:* The communication of the research is required for diffusing the resulting knowledge. Peffers et al. [25] suggest to "communicate the problem and its importance, the artifact, its utility and novelty, the rigor of its design, and its effectiveness to researchers and other relevant audiences".

The activities have been used for this research and have guided the structure of this thesis. In Chapter 5, the problem is identified and motivated, the objectives are defined, and the artefact's design is described. Demonstration and evaluation of the artefact are elaborated on in Chapter 6. The communication of the research is achieved by writing this thesis.

Chapter 3

Literature Study: Software Architecture

This chapter is the result of a literature study on software architecture. It gives answers to the following subquestions of RQ1: "What is software architecture?" and "How can we document software architecture?".

3.1 What is software architecture?

Software architecture has been given many definitions. We are using the definition of Bass, Clements & Kazman [2]: *"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both"*. They conclude that the definition implicates that architecture is an abstraction and that certain details are left out in the abstraction. Additionally, every software system has a software architecture, as for each system it can be shown that it comprises of elements and relations. This includes Android apps as well. They point out that the architecture of systems is not necessarily known, because the source code and documentation is missing or the creators are gone. As each element inhibits behaviour, behaviour can be used to reason about the system. If the behaviour is relevant to the architecture, it should be documented, argue Bass et al. [2]. Architecture requires design and evaluation, as not all architectures enable the system to fulfill its requirements: there are good and bad architectures.

Software architecture can play an important role in software engineering: It simplifies the understanding of complexity in systems by reducing them to a high-level overview; it encourages the reuse of software elements; it acts as a partial blueprint in development; architecture shows how the system may evolve in response to changing requirements; it allows for analysis of quality attributes and conformance to specifications [5].

3.2 Documenting software architecture

Software architecture is typically documented in an architectural description (AD). This is a set of products that document an architecture in a way its stakeholders can understand and demonstrates that the architecture has met their concerns [27]. Documenting architecture aids in understanding and communication of architecture and provides opportunities for analysis [5]. Unfortunately, not every system has a valid and up-to-date architectural description [27].

The architectural description consists of a set of views, each of which describes the system in a different way. They are descriptions of the system relative to a set of concerns from a certain viewpoint, and they can be defined by a purpose, a scope, and elements [21]. The purpose describes

the concerns the view is intended to address. The scope defines the boundaries of what is shown in the view and what isn't. The elements consist of the elements which comprise the view and the relations among them. Although software and their architecture may differ from one another, it is common that identical questions are answered by views. Examples of these questions are:

- What are the main functional elements and how is this functionality organised?
- Which elements outside of the app are used and how does the synchronisation work?
- What information is managed and how is this information stored?

Different software raises the same questions, which address the same concerns and leads to views consisting of the same type of elements. To prevent architects from reinventing the wheel, viewpoints have been introduced. A viewpoint is a collection of patterns, templates, and conventions for constructing one type of view [27]. Views are to viewpoints what programs are to a programming language [21]. Viewpoints can be seen as domain-specific languages that codify a specific set of concepts and relations. Different collections of viewpoints have been proposed in the last decades. Examples are the 4+1 view model by Kruchten [23], the Siemens set of viewpoints [22], and the viewpoint catalog by Rozanski& Woods [27].

Hilliard [21] has dissected viewpoints in several characteristics that point out what is required of a viewpoint.

- *Name:* This identifies the viewpoint
- *Addressable concerns:* A range of concerns the viewpoint addresses
- *Viewpoint language:* The elements which occur in a view conforming to the viewpoint
- *Construction rules:* The rules by which elements can be combined
- *Interpretive rules:* The rules by which well-formed views can be interpreted
- *Analytic techniques:* Procedures associated with the viewpoint by which resulting views can be analysed.

3.3 Pros and cons of using viewpoints

Dividing the system's architecture in multiple viewpoints has several benefits as described by Rozanski & Woods [27]:

- *Separation of concerns:* Addressing all concerns in one model leads to a bloated model that is hard to understand and will confuse stakeholders. Separating in viewpoints that address different groups of concerns helps the design, analysis, and communication as each group can be addressed separately.
- *Communication with stakeholder groups:* Different stakeholder groups have different concerns. Using viewpoints helps to select the right set of viewpoints addressing the relevant concerns. Each of the viewpoints can be presented in a notation that the relevant stakeholders are familiar with, which aids the communication process.
- *Management of complexity:* Separating the complexity in different aspects helps to manage it in a divide-and-conquer fashion.
- *Improved developer focus:* Software architecture acts as a blueprint for developers. Instead of having one blueprint which addresses everything, viewpoints provide developers with blueprints that are tailored to their interest.

Viewpoints come with some pitfalls as well:

- *Inconsistency and fragmentation:* Describing the same system with multiple views is bound to bring consistency problems [27]. Furthermore, different concerns lead to different notations. This creates the multiple-view problem in which overlapping issues in the system are described in different notations in different views [29].

- *Fragmentation:* The set of views requires a lot of manual work to create and maintain and this may lead to fragmentation [27].
- *Selection of the wrong set of views:* Architects may select the wrong set of views for describing a system as this is not obvious at all times [27].

Chapter 4

Literature Study: Android

Android is an open-source mobile operating platform developed by Google. As of November 2017, 2.3 billion active smartphones use Android [32]. These smartphones usually ship with an app store, such as the Google Playstore. In the first quarter of 2018, Android users could choose from 3.8 million apps [30]. In the following sections, we will explain the architecture of the platform, how Android apps work, and what has been researched on AD for Android apps. This chapter is the result of a literature study that targets the remaining subquestions of RQ1: "How do Android apps work?" and "What is the current state of research in documenting Android app architecture?".

4.1 Platform architecture

Android runs on a modified version of the Linux kernel. The apps, which are written in Java or Kotlin, are compiled ahead-of-time by the Android Runtime (ART) to Dalvik bytecode. This bytecode is executed by ART (a virtual machine) that is running on the platform. Android provides native libraries written in C/C++ such as SSL and OpenGL ES as well as the ability use C++ code in apps using the Native Development Kit (NDK). The Application framework runs on the top of the ART VM and provides many higher-level services to applications in the form of Java classes. An example of such a service is the Notification Manager, which allows applications to display alerts and notifications to the user. Apps run on top of the ART VM and use these services to interact with the Android platform and the device.

The applications analysed in this thesis run on the top of the ART VM and are exclusively written in Java, as Kotlin is not supported by analysis programs. Currently, the latest version of Android is Android 9.0 (Pie).

4.2 Android application fundamentals

Each app is a process that runs in a sandboxed environment. The entry points of the application are coded in the *manifest* file of the app. This XML-file contains a list of all the permissions required (i.e.: send SMS), the components of the app and the entry points to those components.

Let's consider a mail app as our primary example. The mail app has a manifest file that will declare an activity as the general starting point of the application. Android provides four components that may be used to create an app: activities, services, broadcast receivers and content providers [9]. They will be explained in more depth later on.

- **Activities** represent a single screen with a user interface, which may show a mail inbox with a list of unread emails.
- **Services** are able to keep the app running in the background. In our mail app, a service could run in the background and poll for new emails at certain intervals. Without it, the user can only receive emails when using the application.

- **Broadcast receivers** are able to listen and respond to (system) broadcasts. For a mail app, it is necessary to start services when the device reboots. This can be accomplished with a receiver that can listen for (system) events such as booting up. When this event is broadcasted, the receiver is notified and may start background services of the mail app.
- **Content providers** manage a shared set of data that other applications can query. Another possible requirement for the mail app would be importing contacts. However, each app runs in a sandboxed environment which prohibits communicating with the Contacts app. Using the content provider of the Contacts app, our application is able to request all the known email addresses.

4.2.1 Communication between components

Multiple connectors exist for communication between components. All of them rely on the Binder interface, a lightweight remote procedure call mechanism, that provides interprocess communication (IPC) in Android. Four types can be distinguished:

- **Explicit intents** are a subtype of intents. Intents are asynchronous messages that request an action from other components. They may provide extra data which the responding component may use. Explicit intents are intents where the component is explicitly specified. An explicit intent can start an activity or service.
- **Implicit intents** are the other subtype of intents. This type does not specify a component but instead declares a general action to perform, which allows other components (or other apps) to handle it. Receiving components must declare an intent filter which specifies what actions the component is able to handle. An example would be to send an implicit intent for displaying a photo, to which multiple apps may respond.
- **Remote procedure calls (RPC)** has three different flavours which are very similar. Android allows developers to create an interface using the Android Interface Definition Language (AIDL) that both processes agree upon to communicate with each other. An interface created in AIDL results in a Binder that allows for IPC. This step is more simplified using a Messenger, which uses a default AIDL to establish basic communication. If no IPC is required, components may also use a Binder within the same process. These types may be used between an activity and service to communicate the status of a download or another long-running task.
- **Content resolver** is the method for connecting with content providers. It resolves a content URI to a specific content provider and allows for querying and modifying data of other applications. The default Contacts app on Android phones provides a default content provider through which other applications may be able to access the user's contacts. A messenger app would therefore not require to maintain its own list of known contacts.

4.2.2 App components in detail

Each of the four components is implemented as a subclass of the Android Developer Framework (ADF). For example, an activity is implemented by inheriting the *Activity* class. Developers do not own their implemented app components. The platform does in order to effectively use scarce resources, such as battery and memory. The platform may kill activities and services in the background to free up resources. To deal with this constraint, developers must pay close attention to the lifecycles of their components and what is expected of the platform.

Activity

Activities provide an interface to the user. Multiple activities may form an application, but each activity represents only a single screen with a user interface. For example, an email app may have an activity that displays a list of emails and another activity for readings mails. Each activity can be composed of multiple fragments, which represent a smaller portion of the user interface.

Fragments and activities may be filled with UI elements, such as buttons and listviews. The contents of an activity is defined in an XML layout file, while the logic is written in Java or Kotlin [13].

Activities may be started with implicit or explicit intents and stop when either the *finish* method is called or the user quitting the activity. The lifecycle of activities consists of six callbacks. The activity starts with an *onCreate* event and is destroyed in an *onDestroy* event. *OnPause* and *onStop* are used when the activity is not active and also not visible in the latter. The states *onStart* and *onResume* are used when the activity becomes active again [19].

Service

A service is a general-purpose entry point for keeping an app running in the background. It is a component that runs in the background to perform long-running operations or to perform work for remote processes [18].

There are two types of services; *bound services* and *started services*. In the first type, other components, such as activities, can bind themselves to services using IPC. In the latter type services can be started with intents. Started services can be bound to at a later moment. For example, a started music player service identifies which songs can be played, and an activity binds to the service when the user wants to play a song and control playback. To use either type of service, the related callback methods must be implemented.

Bound services require the implementation of the callback that deals with binding and unbinding. Multiple components may bind to a bound service, but once none are bounded, the service will be destroyed by the system. When a client tries to bind to the service, the service returns a Binder interface, which the client then uses to communicate with the bound service. Communication between bound services and their bound components can be accomplished in three ways. If the service and the component are in the same process, a Binder is recommended. If the service and the component are in different processes, but no multi-threading is required, a *Messenger* is recommended. If multi-threading is required, an *AIDL* can be used [10].

Started services are started with intents and three flavours can be found; Service, IntentService, and ForegroundService. A service has no special additions and developers are required to handle threading, intents and work queues on their own. IntentServices provide default implementations concerning these three topics. Both the Service and the IntentService run in the background and are eligible to be killed by the system if resources are required. In some instances, this behaviour is not preferred. In a music app, a persistent notification is shown which allows the user to control the playback. This type of service is a ForegroundService and they require the creation of a notification so the user is aware that this service is actively running. Communication with the service and a component can be achieved using a local broadcast listener and intents. The component registers a new local broadcast listener with an intent filter to which the service can send an intent with data to. The local broadcast listener is able to relay the data to the activity or perform tasks itself [16].

Broadcast receiver

Broadcast receivers are components which can subscribe to different types of broadcast announcements and respond consequently. For example, when the device just booted, the system sends a *BOOT_COMPLETED* system broadcast allowing apps to start their services. Broadcasts are wrapped within intent messages and receivers use intent filters to specify to which broadcasts it wants to subscribe.

Apps can receive broadcasts via manifest-declared receivers and context-registered receivers. While the app is not active, manifest-registered receivers will receive the broadcasts they are registered to. Context-registered receivers are not declared in the manifest but are registered by a component at runtime in the application with a context. The receiver is active for the lifetime of the given context, which can be the application or a component. Local broadcast receivers are a simplified form of context-registered receivers with less overhead, but are unable to register for

broadcasts outside of the application. Broadcast receivers can be restricted with permissions. In that case, only applications with the required permissions can send broadcasts. Broadcast receivers are intended to perform small tasks in response to events or trigger other components for heavier computational work. Android kills receivers that are running for over 10 seconds. If performing heavier work is required, a background service that performs the work should be started [11].

Content provider

A content provider manages a shared set of app data that you can store in the file system, in an SQLite database, on the web, or on any other persistent storage location that your app can access. Other applications can query or modify the data through the content provider. The content provider is declared in the manifest file and is accessed, given the right permissions, via a content URI that identifies the data. Although it is possible that the application uses its own content provider, they are meant for data sharing with other applications.

Access to the content provider can be achieved in two different ways; using a content resolver via IPC or by passing an intent to an application that has access to the content provider. The content resolver runs by default on the main thread unless specified differently. The latter case is useful whenever an app needs data to which it has no access. For example, if an app wants to get a mail address, it can send an implicit intent for a contacts MIME type. The user will see a selection of apps that can handle this type of request and when the user has picked a contact, this data is set as the result of the intent [12].

Combining app components and connectors

Table 4.1 provides an overview of the relations between the components and connectors. It is considered a security hazard to start a service using an implicit intent, as it is uncertain which services respond and the user is unable to see which services have started. Therefore, this behaviour is prohibited.

	Activity	**Service**	**Broadcast receiver**	**Content provider**
Binder	Communication(bound)	Communication (bound)		
AIDL	Communication(bound)	Communication (bound)		
Messenger	Communication(bound)	Communication (bound)		
Explicit intent	Start	Start		
Implicit intent	Start using intent filters		Trigger using intent filters	
Content resolver				Communication

Table 4.1: Possible combinations of components and connectors

4.2.3 Architecture components

In both 2017 and 2018, Google introduced Architecture Components at Google IO. These components are a response to developers having trouble with complex lifecycles and a lack of recommended architecture [7]. At Google IO 2017, they introduced four components: Lifecycle Components, LiveData, ViewModel and the Room library. All of these components solve architectural problems with UI component lifecycles and handling data persistence [8]. Lifecycle-aware components help managing activity and fragment lifecycles by performing actions in response to a change in lifecycle status of another component. For example, a location provider stops requesting the location when an activity is paused or stopped. LiveData provides a lifecycle-aware link between the database and UI components. It allows developers to observe data changes and notify UI elements only when they are active. Room is a database library that simplifies data persistence and reduces the need to write complicated queries. ViewModels supply activities of data and survive configuration changes, such as rotating the screen.

In 2018, the Navigation Architecture Component and the WorkManager were added to the list of Architecture Components. The Navigation Architecture Component helps developers implementing navigation in apps by taking care of boilerplate code [15]. The boilerplate logic is moved from the activity's Java file to simple lines in the activity's XML layout file. The WorkManager API simplifies the creation of asynchronous tasks and threading [17]. It is another abstraction layer for developers to reduce complexity.

None of these components are declared in the manifest and stay within the lifecycle of an app component. However, they do influence the architecture of the app components themselves.

4.3 Research into Android app architecture

As Android is a relatively new platform that has changed quite rapidly, research in mobile software engineering is not as developed yet compared to traditional software engineering [34]. As a result of that, not much research has been conducted in software architecture for Android apps. Most papers on Android architecture are concerned with the security of apps or automated testing of the user interface. Some papers address architecture of the platform rather than the architecture of apps.

Two papers provided interesting models on Android architecture. The models in the two papers are similar, as they originate from a similar subset of authors. In both papers, the models are meant to clarify the architecture, but are not meant as example of architecture documentation.

The first paper is written by Bagheri, Kang, Malek & Jackson [1]. They have looked into key drivers that created the architecture that is present nowadays. Additionally, they explain the architectural principles (components and connectors) in Android and empirically identify architectural styles the framework offers. They reverse engineered the architecture of the K-9 Mail app and created a model shown in Figure 4.1. They note that "that the complete architecture of K-9 mail is significantly more complex and comprises more than 40 software components" and that they have only shown a subset for clarity. The authors identify four components (activities, services, broadcast receivers and content providers) and four connectors (explicit intents, implicit intents, RPC and content receivers).

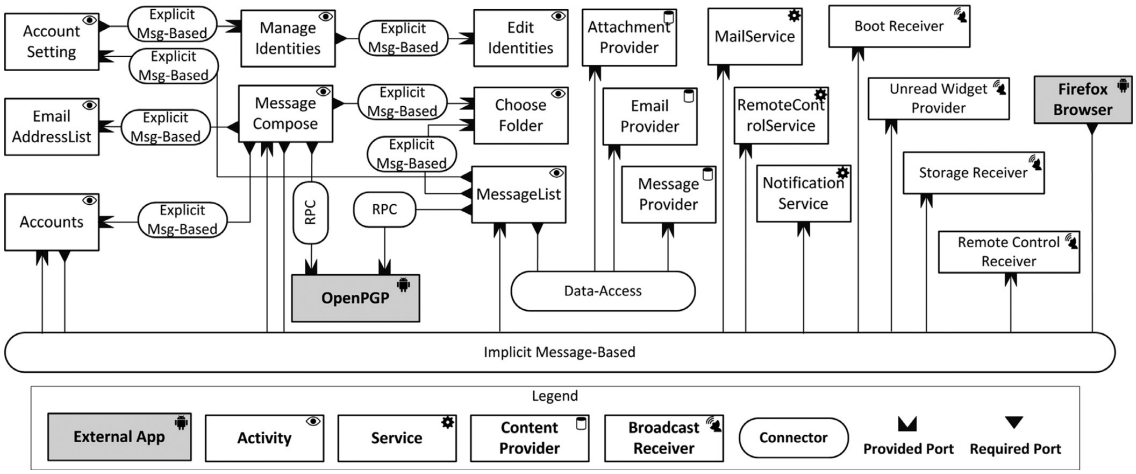

Figure 4.1: The model created by Bagheri et al. [1]

The second paper is by Schmerl et al. [28] and is focused on formal software architecture modelling in order to analyse security. They have defined an architectural style in Acme [6] in order to analyse the intents interactions and the permissions in Android. Their style uses the

same set of components as Bagheri et al. [1]. In their paper, they present a model of two apps that may interact with each other.

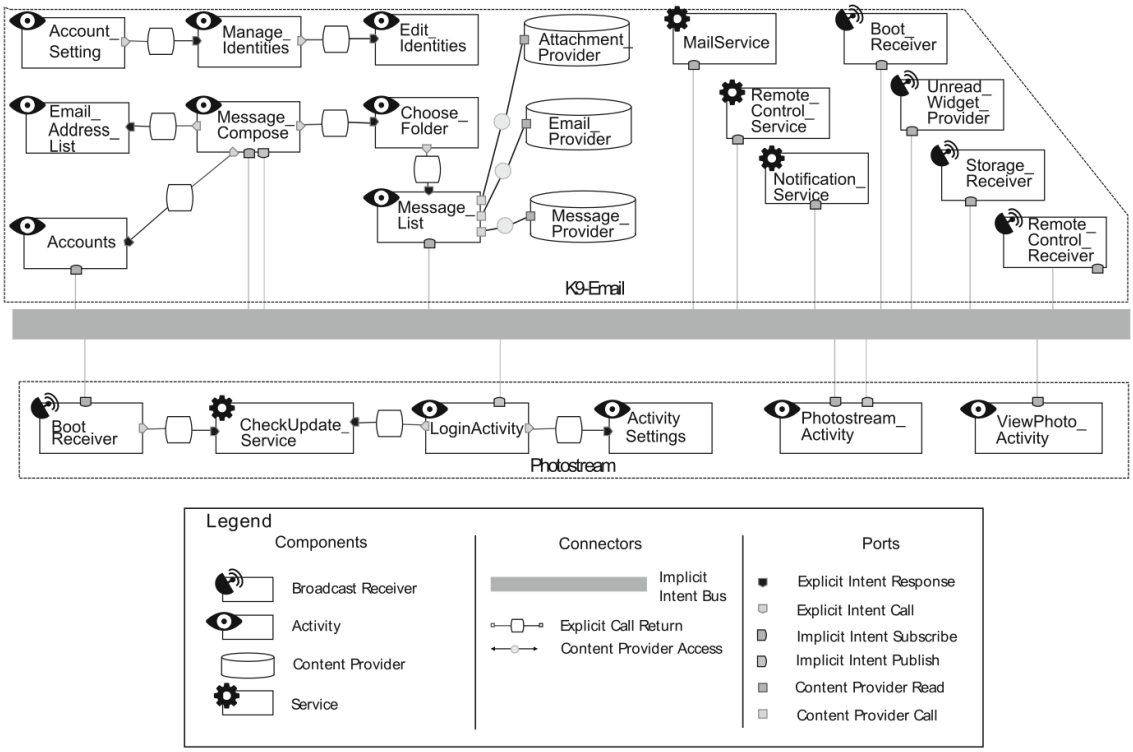

Figure 4.2: The model created by Schmerl et al. [28]

Chapter 5

Artefact Design

We start this chapter with identifying the lack of AD for Android apps as a problem. We follow this section up by stating the objectives for a solution in 5.2. An artefact is developed and described in sections 5.3. We conclude this chapter by laying out a procedure for creating AD for large applications (5.4). This chapter targets research question 2 and its subquestions.

5.1 Problem identification

We have seen from the literature study that there is a gap of research concerning architecture documentation for Android apps. This thesis aims to explore what the fundamental concepts and mechanisms are. As we want to be able to create views, we need a set of elements that allows us to create those views which address the concerns of stakeholders. Therefore, our challenge is to find the right set of elements and to find the right set of views that addresses our needs.

5.2 Defining the objectives

In order to understand what aspects of apps need to be documented, we will apply the viewpoint catalog of Rozanski & Woods [27] to Android apps. We will show that some viewpoints are not relevant due to the characteristics of the Android platform in the scope of documenting the architecture. After this section, we will state the objectives and test current research on these objectives.

5.2.1 Applying viewpoints to Android

The catalog consists of seven viewpoints; context, functional, information, concurrency, development, deployment, and operational. An overview of the viewpoints can be seen in Figure 5.1. The context viewpoint, the deployment viewpoint, and the operational viewpoint are not relevant when documenting Android app architecture. The context viewpoint describes the relationships, dependencies, and interactions between the system and its environment. As the scope is limited to understand how the Android app and its architecture works, placing the app in the context of the organisation and its environment is out of scope. The deployment view describes the environment into which the app will be deployed and the dependencies that the system has. The hardware side is an Android phone and the deployment handled by the Google Playstore. The same applies to the operational viewpoint, which describes how the system will be operated, administered, and supported when it is running in production. A lot of the concerns related to this viewpoint are dealt with by the Google Playstore. Therefore, both viewpoints are not relevant.

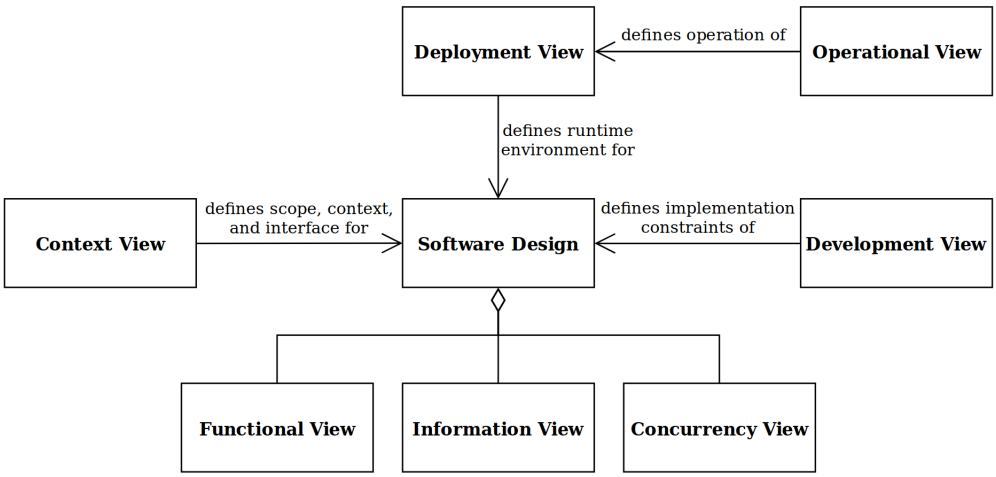

Figure 5.1: The viewpoint catalog of Rozanski & Woods

The remaining viewpoints are of importance:

The development viewpoint describes the architecture that supports the software development process. This includes build and configuration management, design constraints and standards to ensure technical integrity. Although, we are more concerned with the code structure, dependencies, and the organization of modules, as Android apps are created in Java and consist of code and packages that depend on each other.

The functional viewpoint describes the system's runtime functional elements and their responsibilities, interfaces, and primary actions. What is the system required to do, and what and what is the system required not to do? Apps provide functionality to the user and the organisation of this functionality is part of the functional viewpoint.

The information viewpoint describes the way that the system stores, manipulates, manages and distributes information. Examples of typical concerns are the ownership, consistency, and retention of data. These concerns apply to Android applications as well, as they may collect and process data.

The concurrency viewpoint describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled. This viewpoint deals with the coordination of concurrent activities and interprocess communication. Nowadays most smartphones are shipped with multicore processors which allow for multithreading in apps. Additionally, Android provides multiple methods for interprocess communication.

The viewpoint catalog of Rozanski has shown that documentation of Android app architecture requires more focus on the construction and constraints of software design rather than the environment and the operations, as the latter is handled by the platform.

5.2.2 Objectives

The purpose of an architectural description is to describe the complexity of architecture to the people who need to understand it. Due to the maturity of mobile software engineering, no Android-specific architecture documentation exists. This section sets out to explore what is required in terms of modeling architecture.

Requirement 1: Providing the right level of detail independent of app size

One of the challenges when creating architecture descriptions is to provide a good level of detail [27]. One of the requirements of the models is that the level of detail works for all sizes of apps.

Whether we are modelling small or large apps, the models should stay comprehensible and not scale linearly with the size of the application.

Requirement 2: Different views should not overlap or be too fragmented

As established in the literature, it is generally better to document architecture from multiple viewpoints. However, this may lead to the multiple-view problem in which the diversity of approaches leads to fragmented and in some cases conflicting set of models [29]. Therefore, we require that a resulting set of views should not be in conflict, overlapping or be fragmented.

Requirement 3: Reduce semantic scope with Android-specific elements

It is preferred to use Android-specific elements in an Android-specific architecture documentation, as domain specific constructs are better suited for communication with users within the domain [4]. Furthermore, reducing semantic scope can lead to better support for generating implementation from models [4] and may reduce the complexity of the models.

Requirement 4: The views should address concerns

Views conform to viewpoints that should address concerns [27]. The created views should answer questions stakeholders are concerned with. Typical concerns are:

- What is the functionality of the application and how is this structured?
- What information is managed and how is this information stored?
- How are concurrent activities coordinated and IPC handled?

5.2.3 Why current research does not meet the requirements

We have seen two models of Bagheri et al. [1] and Schmerl et al. [28]. In Table 5.1, each paper is tested on all four requirements. Both papers model only a subset of the components of K-9 mail app and are unable to model the whole application. Therefore, these models do not satisfy REQ1. They only consist of one view and consequently can't be fragmented or overlapping with other views. REQ3 is satisfied by each model, as they use Android-specific components. Although they show some information flow and division of functionality in components, they can't serve as a thorough blueprint for the implementation, which is a concern for developers. Other modelling languages do not meet REQ3, which specifies to use Android-specific components in order to reduce the semantic scope and allow for better communication with stakeholders.

Paper	Satisfying REQ1	Satisfying REQ2	Satisfying REQ3	Satisfying REQ4
Bagheri et al.	No	N/A	Yes	Partly
Schmerl et al.	No	N/A	Yes	Partly

Table 5.1: Testing current research for satisfying the four stated requirements

5.3 The artefact

It is clear that handling bigger apps (REQ1) requires a solution. Therefore, three layers of views are used that form a hierarchical decomposition. The highest layer consists of elements that are each decomposed in a view in the middle layer. Each of the elements in the middle layer is decomposed into elements of the lowest layer. Initially, using lower levels of abstraction resulted in large and incomprehensible models. As a result of using a hierarchical decomposition, each view in a lower layer is scoped to the element in the layer above and lower levels of abstraction have

become useful. Additionally, the scoping has benefits towards REQ2, as each element in a higher level would only be modelled once in a lower layer.

The three layers are the Functional Architecture Model Layer (FAML), the Android Component Layer (ACL) and the Code Layer (CL). The FAML is the highest layer and the CL is the lowest layer. We will discuss each layer briefly and then clarify using an example.

Functional Architecture Model Layer

The Functional Architecture Model Layer is based on Functional Architecture Modeling by Brinkkemper & Pachidi [3]. As described by the authors, Functional Architecture Modeling is essential for identifying the functionalities of the software product and translating them into modules, which interact with each other or with third-party products. These views consist of modules, project/-module scopes and information flows. The modules are scoped to a project or higher level module. This layer corresponds strongly with the functional view of viewset catalog [27], as functionality is the main focus.

Android Component Layer

The Android Component Layer consists of the app components and connectors. We identify seven components (activities, services, broadcast receivers, content providers, external apps, custom classes and intent filters) and five connectors (explicit intents, implicit intents, RPC and content receivers, call/access). The notation and idea are derived from the models of Bagheri et al. [1]. However, we added the intent filter and custom classes. Custom classes were added because we found that fragments and viewmodels are strongly coupled with activities and may connect to other components for the purpose of the activity. The intent filters are the entry points of the app and are directly connected to the other components. The addition of custom classes required the call connector. If components instantiate other components or call their methods, this is a call/access connector.

Code Layer

The Code Layer is, as the name describes, directly linked to the code. As Android is very event-driven, the elements in this layer are events, tasks, and choices. Each component in the Android Components Layer is modelled in this layer. This layer indicates how each component exactly works and could act as a very direct blueprint for the implementation.

5.3.1 Modelling an example application

This section showcases the artefact using an example application. Consider the following application: We have a simple application that randomly picks a user's contact and displays "{Contact name} says Hello!" messages to the user. Screenshots of this app can be seen in Figure 5.2.

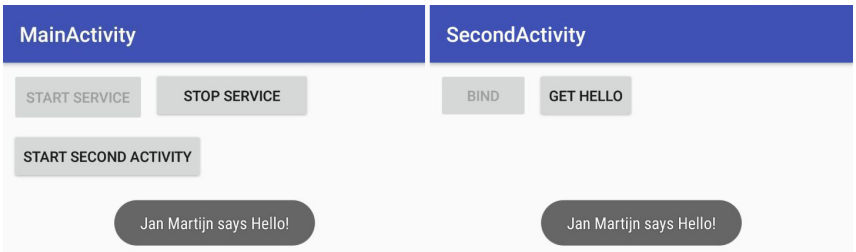

Figure 5.2: Screenshots of the example application

The manifest lists a permission to read contacts and four app components. It is shown in Listing 5.1. The starting activity is *MainActivity* as indicated by the intent filter *action.MAIN*. This activity registers the broadcast receiver *HelloReceiver*. Additionally, it allows for starting and stopping the *HelloService* and firing an implicit intent so *SecondActivity* starts. When the service *HelloService* is started, a random contact name is retrieved using a content resolver and broadcasted to *HelloReceiver*. The receiver propagates this name to a normal Java class called *ToastDisplay*, which displays a toast message in the format "{Contact name} says Hello!". The *SecondActivity* allows for binding to *HelloService* and then shows the 'Get Hello' button. Tapping this button displays the same toast message. The code of all the classes can be found at https://github.com/yorickvanzweeden/bachelor-thesis.

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
<application>

  <activity android:name=".MainActivity"
    android:label="MainActivity">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>

  <receiver android:name=".HelloReceiver"/>

  <service android:name=".HelloService" />

  <activity android:name=".SecondActivity"
    android:label="SecondActivity">
    <intent-filter>
      <action android:name="nl.yorickvanzweeden.intent.action.SECOND_ACTIVITY"/>
      <category android:name="android.intent.category.DEFAULT" />
      <data android:mimeType="text/plain" />
    </intent-filter>
  </activity>
</application>
```

Listing 5.1: The manifest of the example application

Code Layer view of MainActivity

Consider the following edited code snippet of MainActivity.java:

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  findViewById(R.id.button_start).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {  startService(); }
  });

  findViewById(R.id.button_startActivity).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
      startActivity(new Intent("nl.yorickvanzweeden.intent.action.SECOND_ACTIVITY"));
    }});}
```

Listing 5.2: Code snippet of MainActivity.java of the example application

The event *onCreate* defines three specific tasks (disregarding the base call): setting the content view and attaching two onClickListeners. We can denote this in the Code Layer view as shown in Figure 5.3. Yellow triangles indicate events or constructors and they are the starting point in a Code Layer view. The white rectangles imply tasks that are executed. The tasks may represent one line of code or multiple methods depending on the relevance and desired specificity. Dashed lines indicate that multiple tasks are executed by a method, such as the *onCreate* event. Each solid line indicates that a method is called or an event has been set up.

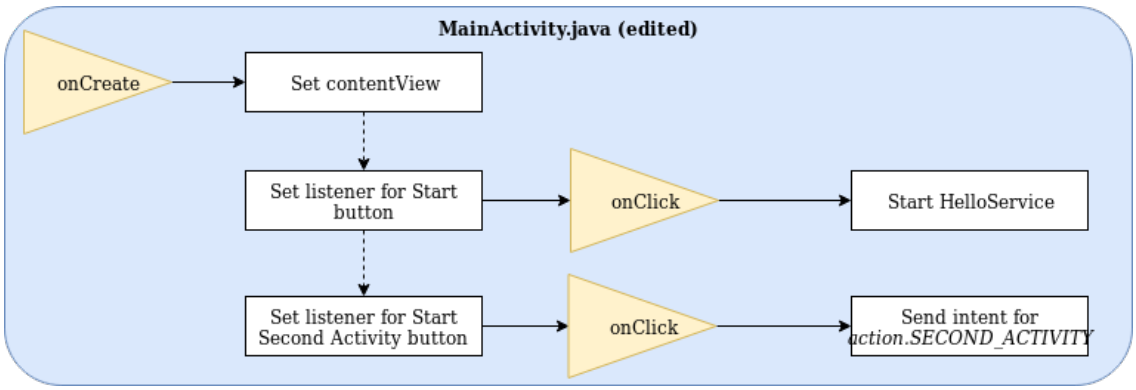

Figure 5.3: Code Layer view of MainActivity (code snippet of Listing 5.2)

If/else statements might be irrelevant for the architecture, but might as well divide the program in diverging flows that should be modelled. If/else statements are modelled as Figure 5.4 shows.

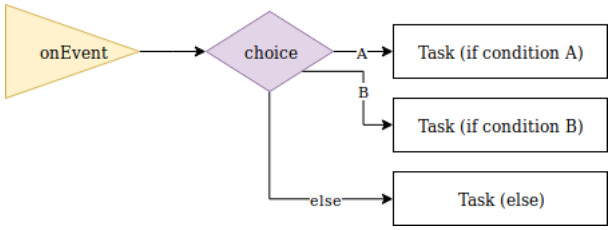

Figure 5.4: Code Layer view of an if/else statement

The Code Layer view may consist of multiple classes. Therefore, it is useful to denote the scope of the code with a blue rounded rectangle. The CodeLayer view of *HelloReceiver* creates, upon receiving a broadcast, a new instance of *ToastDisplay* that will display a toast message. This is shown in Figure 5.5.

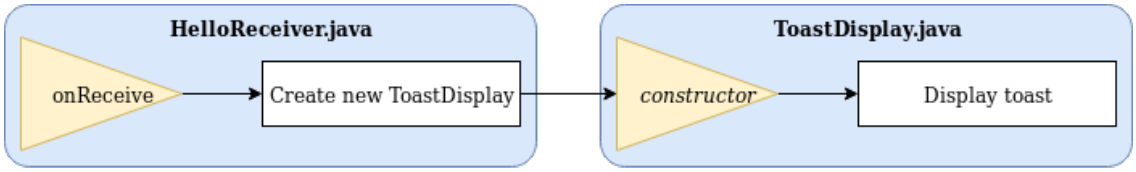

Figure 5.5: Code Layer view of the HelloReceiver

The entire class *MainActivity* has three event listeners and defines onClickListeners for three buttons. One button starts the service *HelloService*, another button stops this service and the third button sends an implicit intent for *action.SECOND_ACTIVITY*, which will be answered by *SecondActivity* as declared in the manifest file. The code is mapped to the Code Layer view in Figure 5.6. The other classes, *HelloService* and *SecondActivity*, have been modelled in a similar fashion and can be seen in Appendix A.1 and A.2.

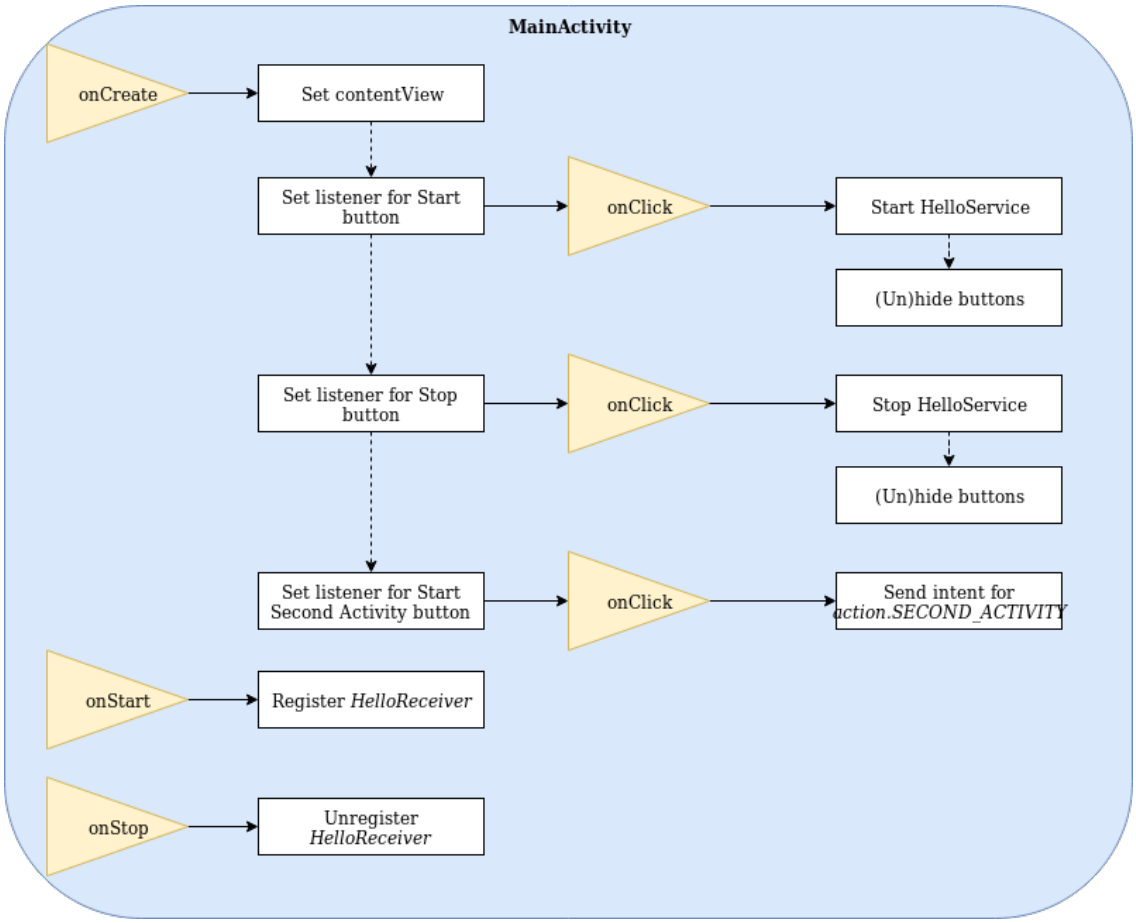

Figure 5.6: Code Layer view of MainActivity

App Components Layer

The ACL view of the example application can be seen in Figure 5.8. All components and connectors can be found in this view, except for the local Content Provider. The *Contacts* Content Provider is a third-party component and thus not local. This layer usually contains all the elements in a module and not the entire app. As such, references to app components or custom classes of other modules are denoted by the component type in grey. An example is shown in Figure 5.7, as the example app has no other modules. We will show the code for each type of connector in the created ACL.

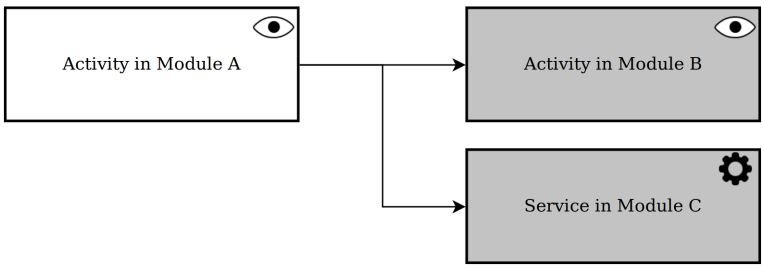

Figure 5.7: ACL view showing components of other modules referenced using the grey component type

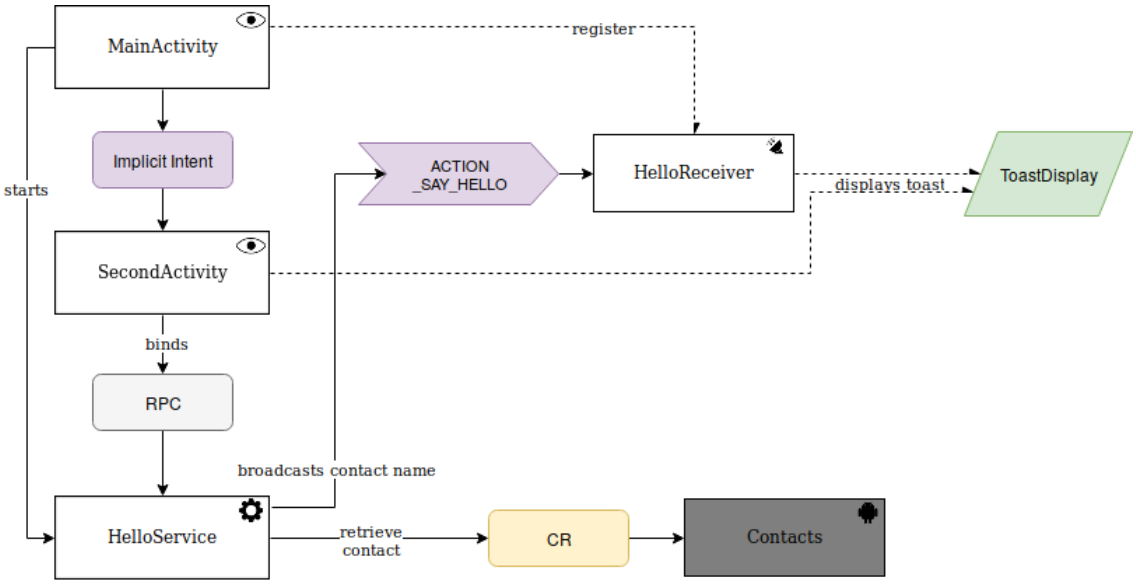

Figure 5.8: App Component Layer view of the example application

Explicit intents: As noted earlier in the literature study on Android, explicit intents specify their app component. The following code snippet (5.3) shows the explicit intent from *MainActivity* to *HelloService*. This is modelled in Figure 5.9.

```
Intent i = new Intent();
i.setClass(this, HelloService.class);
startService(i);
```

Listing 5.3: Code example of an explicit intent

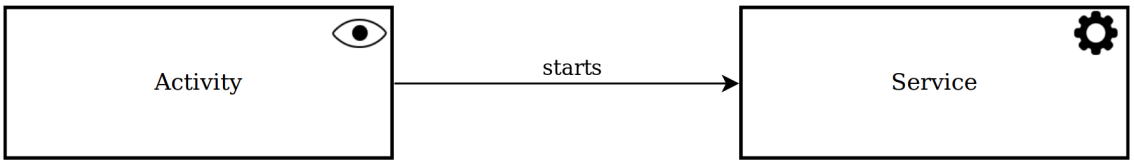

Figure 5.9: ACL highlight of the explicit intent

Implicit intents: This contrasts with the implicit intent. In this type of intent, the action is specified rather than the component. The following code snippet (5.4) shows the implicit intent from *MainActivity* to any component that handles a *SECOND_ACTIVITY* action. We know from the *Manifest* file that *SecondActivity* will respond to such an intent. Figure 5.10 shows how this is represented in an ACL view. Usually, implicit intents are used for generic actions such as sharing a file. The external app component may be used to denote that a 'Sharing app' will handle that intent. Connections to intent filters signify the broadcast of the implicit intent as shown in Figure 5.8. Intent filters can be connected to activities and more often to broadcast receivers. We have included intent filters as a building block for ACL views with the purpose of showing the entry points of the application.

```
Intent i = new Intent();
i.setAction("nl.yorickvanzweeden.intent.action.SECOND_ACTIVITY");
startActivity(i);
```

Listing 5.4: Code example of an implicit intent

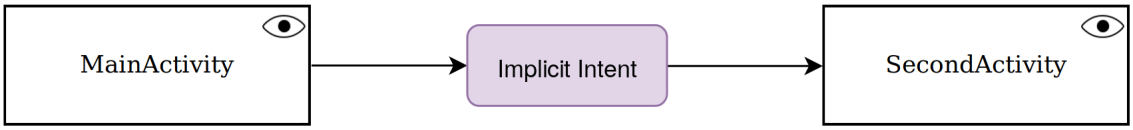

Figure 5.10: ACL highlight of the implicit intent

Remote procedure calls: RPC for services can be implemented in three different ways: Binders, Messengers, and AIDL. Android will generate a Java service file relying on the AIDL file. Both Messengers and AIDL reply on Binders for RPC. When binding to a service, an instance of the service is returned. The following code snippet (5.5) shows the usage of RPC from *SecondActivity* to *HelloService*. The result is that the global variable *mService* is the instance of a running service and its methods are available to *SecondActivity*. In an ACL view, this is modelled as in Figure 5.11.

```
mConnection = new ServiceConnection() {
  @Override
  public void onServiceConnected(ComponentName className,
                                 IBinder service) {
    HelloService.LocalBinder binder = (HelloService.LocalBinder) service;
    mService = binder.getService();
  }
};

Intent intent = new Intent(this, HelloService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
```

Listing 5.5: Code example of an RPC for binding to services

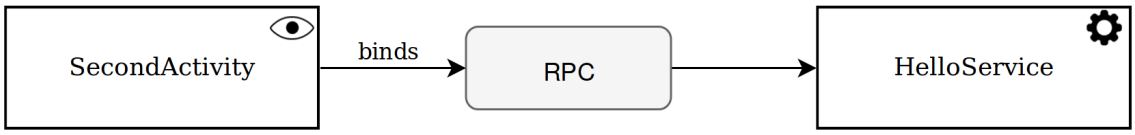

Figure 5.11: ACL highlight of the RPC connector

Content resolvers: Content providers must be connected with using Content Resolver. The code snippet below (5.6) shows how *HelloService* gets a *Cursor* with contacts from the *Contacts* content provider. Figure 5.12 shows how this is modelled in ACL views. Alternatively, Android provides an abstraction that tries to simplify interacting with Content Providers using *CursorLoaders*. However, this construction is deprecated since Android Pie (9.0) [14].

```
ContentResolver cr = getContentResolver();
Cursor cur = cr.query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
```

Listing 5.6: Code example of a content resolver

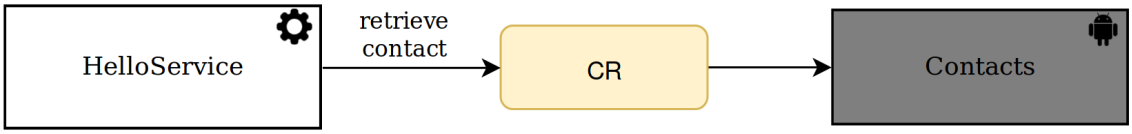

Figure 5.12: ACL highlight of the content resolver

Access/call connectors: Custom classes were included as they may play an important role (such as controllers) and may not be omitted from views. With the addition of custom classes, referencing other classes via calls or access becomes more common. The access/call dashed arrow signifies this relationship. Although we named the connector access/call, declarations such as in Listing 5.7 are included too.

```
public class HelloReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    new ToastDisplay(context, intent.getStringExtra("message"));
  }
}
```

```
class ToastDisplay {
  ToastDisplay(Context context, String name){
    Toast.makeText(context, name + " says Hello!", Toast.LENGTH_SHORT).show();
  }
}
```

Listing 5.7: Code example of access/call connector

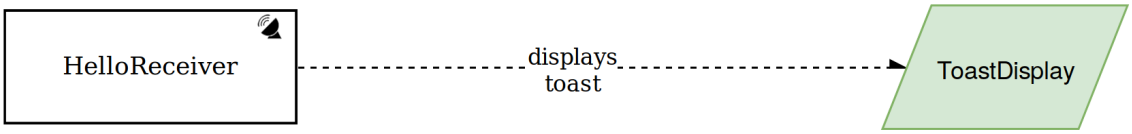

Figure 5.13: ACL highlight of the call/access connector

Functional Architecture Model Layer (FAML) views

One layer of FAML views may be enough in small applications. These will primarily consist of modules. However, in large applications, one layer may not be sufficient and a hierarchy of FAML views may be produced. The example application is too small to contain multiple modules. Actual FAML views are shown in Chapter 6, but these contain no hierarchies. Therefore, an example FAML hierarchy is shown below in Figure 5.14.

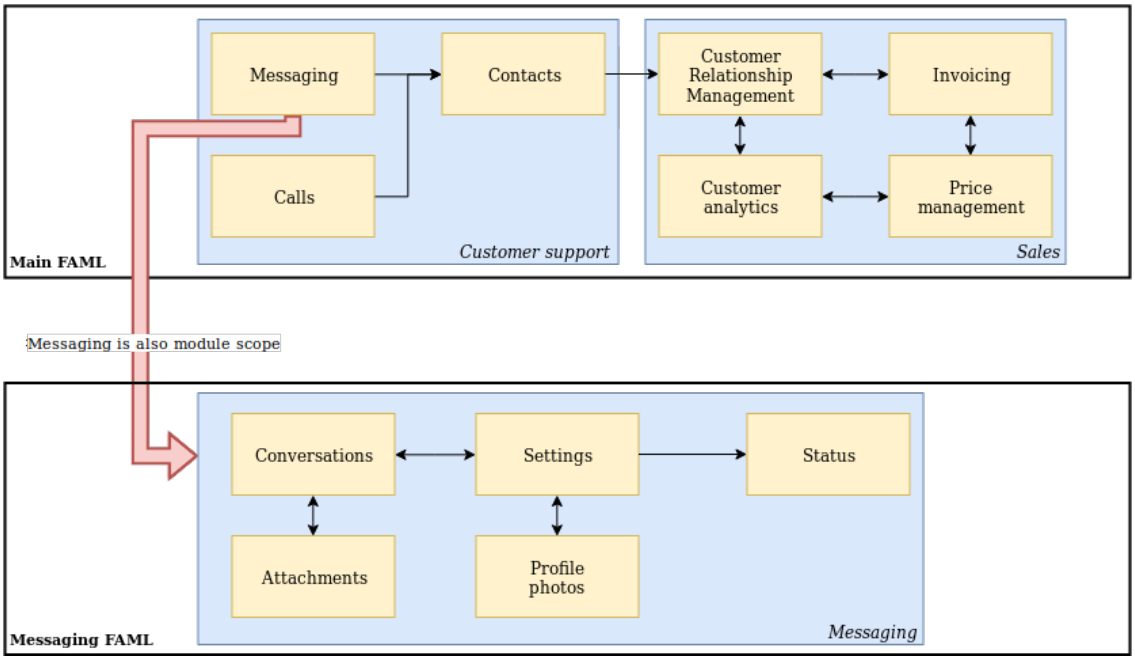

Figure 5.14: Two FAML views that depict how FAML hierarchy works

The yellow boxes are (sub)modules and the blue rectangles define module scope. This scope may be a module of a higher layer, in the same manner as *Messaging* is a module and a module scope. This view primarily shows the functionality an app has to offer and how the functionality is divided.

5.3.2 Artefact summary

Our set of views consists of three layers: the Code Layer, the App Component Layer, and the Functional Architecture Model Layer. Each of the layers addresses different concerns.

Code Layer views are the lowest layer in our artefact. They directly represent code in classes. The views in this layer are scoped to a component in an ACL view. In Figure 5.15, an overview is provided of the elements in the Code Layer views.

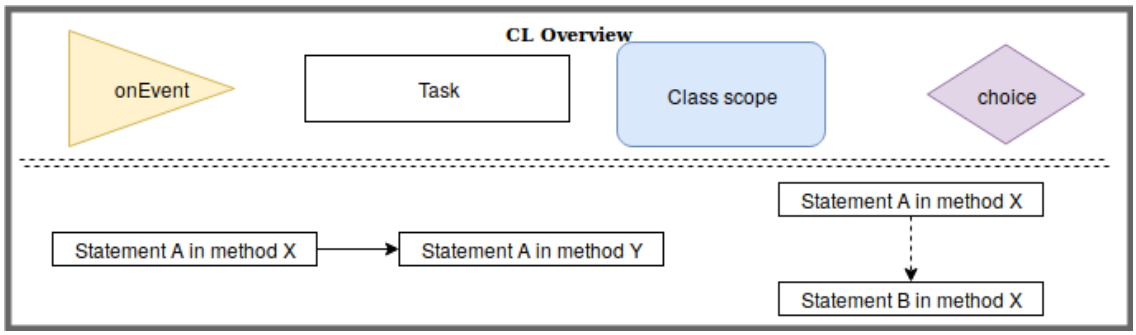

Figure 5.15: Overview of elements in the Code Layer

The App Component Layer views consist of four app components, the external app component, custom classes, intent filters, and grey components. The app components and custom classes are only used once in each view. If references are made to these components in other modules, the grey version is used. Each ACL view is scoped to a module in the FAML view. In Figure 5.16, an overview is provided of the elements in the App Component Layer views.

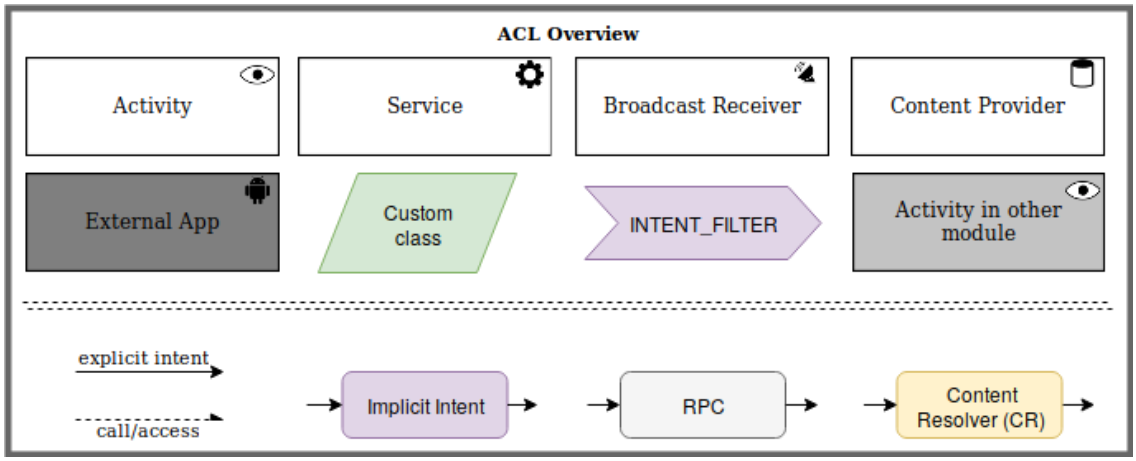

Figure 5.16: Overview of elements in the App Component Layer

The Functional Architecture Model Layer consists of modules, project/module scopes and information flows. It is the highest layer in the artefact. FAML views divide the functionality into modules, which are each scoped to a project. If the application is very large and a hierarchy of FAML views is required, modules may be divided into submodules. In that case, the submodules have a module scope. In Figure 5.17, an overview of the elements in the Functional Architecture Model Layer is shown.

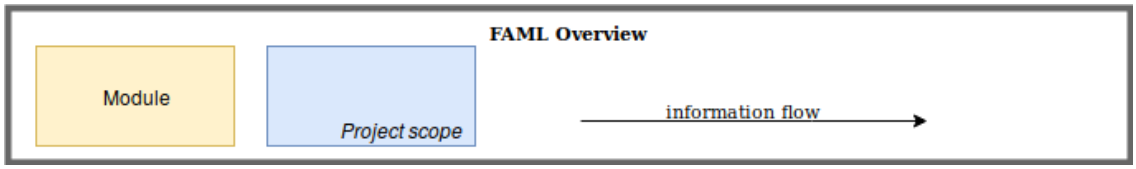

Figure 5.17: Overview of elements in the Functional Architecture Model Layer

5.4 Creating architecture documentation for large applications

Modelling small applications is simple as the complexity is very low. The increasing complexity of larger apps may pose difficulties while creating architecture documentation. A standard procedure for modelling is therefore imperative. Additionally, the use of analysis tools will aid in this process, such as *HUSACCT*, a tool to analyse implemented architecture. Several Python scripts were written to support HUSACCT, and are located in the Appendix and in the previously mentioned GitHub repository. The code was looked at using the *Android Studio IDE*. It also proved useful for finding references to components. The procedure is divided into three parts:

- **Finding all components:** When all app components and intent filters are mapped, only custom classes may be added in the process. The initial list of components allows for a preliminary division into modules.
- **Finding all connectors:** The connectors in the app show which components are connected and help to understand how the components should be grouped.
- **Bringing components and connectors together:** As the views consist of components and connections between components, the model becomes easier to create.

Although we present these steps as sequential, in practice this procedure is more of a cycle. It is possible that not all components are found initially or that all the connectors are properly interpreted. In the process of bringing the components and connectors together, more knowledge is gained about the application. As a result of that, the list of components and connectors may subsequently be revised

5.4.1 Finding the components

This section will list different methods to find components.

Finding components in the manifest

Finding the components first is the easiest way to start. The manifest file contains a record of most of the app components and intent filters. The Python script *parseManifest.py*, located in Appendix B.1, will list all components per category.

Finding context-registered broadcast receivers

Unfortunately, not all components are defined in the *manifest*. Context-registered receivers are declared and registered at run-time. Listing 5.8 is an example:

```
  protected void registerReceiver() {
    CustomReceiver customReceiver = new CustomReceiver();
    this.registerReceiver(customReceiver, new IntentFilter(SOME_ACTION));
  }

public class CustomReceiver extends BroadcastReceiver {
  public void onReceive(Context context, Intent intent) {
    // Things to do
  }
}
```

Listing 5.8: Code example of a context-registered broadcast receiver

To find these receivers, we have two options:

- Finding the registration of receivers using a pattern matching engine (i.e. grep)
- Finding the inheritance relationship to the *BroadcastReceiver* class with HUSACCT and the *parseHUSACCT.py* script (Appendix B.2)

Using grep, we can search for *registerReceiver* using the command (5.9):

```
grep -nr "registerReceiver" -C 2
```

Listing 5.9: Grep command for finding broadcast receivers

This command will recursively search every file for the keyword and also print the four lines of code around it. Example output could be:

```
K9.java-442-     }
K9.java-443-
K9.java:444:     registerReceiver(new ShutdownReceiver(), new IntentFilter(Intent.ACTION_SHUTDOWN));
K9.java-445-     Timber.i("Registered: shutdown receiver");
K9.java-446- }
```

Listing 5.10: Example output of grep command 5.9

Using HUSACCT to find context-registered broadcast receivers

Another possibility is to use HUSACCT. HUSACCT has been developed by Pruijt [26] for the purpose of architecture compliance checking. It has the ability to scan Java files and generate a CSV-file of detected dependencies. We have developed a Python script that analyses this list of dependencies to simplify the creation of our views. The following steps should be followed to get started:

1. Run HUSACCT on the repository
2. Click Analyse Application and then Report Dependencies. This will result in a CSV file with all dependencies.
3. Using *parseHUSACCT.py* (B.2) and the correct path to the CSV-file, function *findContext-DeclaredBroadcastReceivers()* will find all inheritance dependencies to *"xLibraries.android.content.BroadcastReceiver"*. Multiple layers of inheritance in which classes inherit from a class which inherits from BroadcastReceiver, is detected too unless this is hidden behind Android libraries.

Finding references to third-party libraries

Third-party apps may play an important role in the analysed application. To find all third-party dependencies, HUSACCT and *parseHUSACCT.py* (B.2) will be used. Call *Tools.findingThird-PartyDependencies([namespaces])* with the namespace of the app and other known irrelevant namespaces. The script will look for all dependencies with namespaces that are not in that list or the list of known Android namespaces.

5.4.2 Finding the connectors

Finding explicit and implicit intents

All components have been found, but we still need to discover all connectors. Generally, this is trickier as there is no list available in the *manifest*. We will once again use a combination of Python

scripts, HUSACCT and grep commands. The Python script *parseHUSACCT.py* (B.2) supports this process by detecting matches in HUSACCT dependencies. HUSACCT reports dependencies from A to B. An example dependency (5.11) is:

```
com.fsck.k9.activity.MessageCompose, xLibraries.android.content.Intent, Call, Library Method, 270, Direct, false, false,
```

Listing 5.11: Example dependency of HUSACCT

This means that *MessageCompose* calls an intent on line 270. However, this tells us nothing about the nature of the intent and the receiver of the intent. We need the following dependency to figure out that *MessageCompose* directs an intent to *Accounts*.

```
com.fsck.k9.activity.MessageCompose, com.fsck.k9.activity.Accounts, Access, Variable, 270, Direct, false, false,
```

Listing 5.12: Example dependency of HUSACCT matching the dependency in 5.11

However, the fact that *Accounts* is referenced is no evidence that this is the component of the intent. We would have to look at the code (Listing 5.13) to conclude that.

```
MessageCompose.java:270: startActivity(new Intent(this, Accounts.class));
```

Listing 5.13: Code example of content resolver undetected by HUSACCT

As mentioned before, *parseHUSACCT.py* (B.2) offers the functionality to find matches. HUSACCT finds dependencies and the script matches these dependencies on line number and file. Matches have the following relationship: $A \rightarrow B \rightarrow C$. The script allows for filtering on A, B, and C. A and C are called the component filter and B is called the match filter. Intent matches in the *MessageCompose* class can be found by setting the match filter to intents and the component filter to *MessageCompose*. By default, the script will look for intents involving *MessageCompose* (A=*MessageCompose*) and the intents declared in MessageCompose (C=*MessageCompose*). The script will print the found matches and the three lines of code around the match.

Finding content resolvers

Content resolvers are obtained by the *getContentResolver()* method in the Context class. HUSACCT reports only the first dependency and not the ContentResolver reference if the method calls are chained as follows (5.14):

```
Cursor cursor = context.getContentResolver().query(...);
```

Listing 5.14: Example of an undetected content resolver dependency by HUSACCT

Therefore, it is easier to find all content resolvers using the following grep command 5.15:

```
grep -nr "getContentResolver" -C 2
```

Listing 5.15: Grep command for finding content resolvers

Some resolvers are more difficult to find as they may be abstracted away in *CursorLoaders*. We found no proper way to find them other than using grep on the keywords *'CursorLoader'* and *'LoaderCallback'*. Alternatively, picking a ContentProvider and finding all dependencies using the *parseHUSACCT.py* (B.2) is also viable. In that case, the match filter would be *None* to show all dependencies and the component filter *ContentProvider*.

Finding RPC (Binder, AIDL, Messenger)

It is certain that an instance of a service will be returned in a *ServiceConnection* callback. The specific callback depends on the instance of *ServiceConnection*. A code example is shown below in Listing 5.16.

```
public void onBound(IOpenPgpService2 service) {
```

Listing 5.16: Example of an *onBound* method declaration of a *ServiceConnection*

In theory, a script could get all dependencies where a declaration is made and then check all those declarations if they are inherited from *Service*. However, Android provides implementations such as *IntentService* and *MediaBrowserService* that inherit *Service*, but this is not detected by HUSACCT. These inheritance relations are not defined in the source code of the application, but rather in the Android Development Framework. This applies to *ServiceConnection* as well.

The only other option is to grep on keywords such as *'Bind'*, *'Bound'*, *'Connection'* and *'Service'*. Unfortunately, this is very cumbersome and prone to missing RPC connectors.

5.4.3 Combining components and connectors into views

As this procedure is cyclic, it is not required that all components and connectors have been found. It is very difficult to immediately understand the purpose of all components and how they are influencing each other. While creating documentation, more understanding in this complexity is gained, which improves the documentation.

Creating a preliminary FAML view

As large applications may have lots of components, it may be confusing how these relate. Therefore, it is useful to open the application as a user and categorise large chunks of functionality. In a banking app, this could be 'Checking balances' and 'Transferring money'. When a list of functionality is compiled, it is easier to divide the components into categories of functionality going by their name. This division forms the basis for the FAML views and ACL views. It is very likely to be incorrect but reduces initial complexity.

Add intent filters to ACL components

An easy step to continue with is to add all intent filters to the components. Most intent filters that have been found, will be manifest-declared. Context-declared intent filters will be found later and can be added too.

Modeling the first ACL view

The first ACL view to be modelled should not be the centerpiece of the app. It is easier to start with functionality that sparsely connected to other functionality. An example might be 'Widgets' or 'Settings'. Any component can be the starting point of the search for connectors. In a breadth-first-search manner, the relations to other components can be found. This will require looking at the code and using the Python script to find connectors. It even may be easier to model the starting component in the Code Layer, as this requires to obtain a deeper understanding of the component rather than scanning briefly.

Revising the FAML view

An ACL view with very few components could be an indication that the functionality is not properly divided into modules. Contrarily, having too many components could also require revising the FAML views.

Custom classes

Viewmodels, helpers, and fragments are very common in Android apps. They often connect to other components in the name of activities and should also be included in the ACL views. The methods for finding connectors described earlier, help to uncover these important classes.

Getting stuck

If the complexity remains to be too high, it may be a good option to start with providers or receivers first. Content providers and broadcast receivers generally don't have many connectors and are predictable in their purpose. Additionally, it might help to create the Code Layer views of complicated components as this visualises the component in events and the resulting tasks, rather than a huge chunk of code. This applies especially to activities and fragments, which have a lot of lifecycle events.

This procedure should result in creating the desired views. In our experience, some creativity and analytical skills are required in order to complete the documentation of the architecture. In the next chapter, we will apply this procedure in two case studies.

Chapter 6

Artefact Demonstration & Evaluation

To illustrate the applicability of our artefact, two case studies are presented in this chapter. Two open-source applications of different sizes were picked for creating architecture documentation. The first one is the K-9 Mail app and the second one is the OmniNotes app. In Table 6.1, a comparison between K-9 Mail app and OmniNotes is shown. The lines of code have been calculated using CLOC, a tool to count lines of code in different programming languages.

	K-9 Mail app (v5.403)	OmniNotes (v5.5.2)
Java LoC	90K	14K
Downloads in Playstore	5M	100K
GitHub contributors	190	18

Table 6.1: A comparison of both analysed apps

6.1 Creating AD for the K-9 Mail app

We will describe the K-9 Mail app first. This will be followed by an explanation of the results. Lastly, we evaluate the results and the process.

6.1.1 Introduction to K-9 Mail app

The K-9 Mail app is an open-source email client with support for multiple accounts, PGP, and different mail providers. The four most important activities of the app show a list of accounts, a list of folders, a list of emails and a screen to compose emails. All of these screens can be seen in Figure 6.1. Additionally, the app provides a setup for adding new accounts, three different settings menus, and three different widgets.

6.1.2 Results

The K-9 Mail apps architecture was documented using the artefact with additional support of analysis tools. These tools include Python scripts, grep commands, regex queries, HUSACCT dependency analysis and git. Git was used in order to understand past design choices and the evolution of architecture. All of the views created for the K-9 mail app can be found at https://github.com/yorickvanzweeden/bachelor-thesis. Because the K-9 mail app is quite large and the time for writing this thesis was limited, no views for the Code Layer have been created. The second demonstration does include Code Layer views.

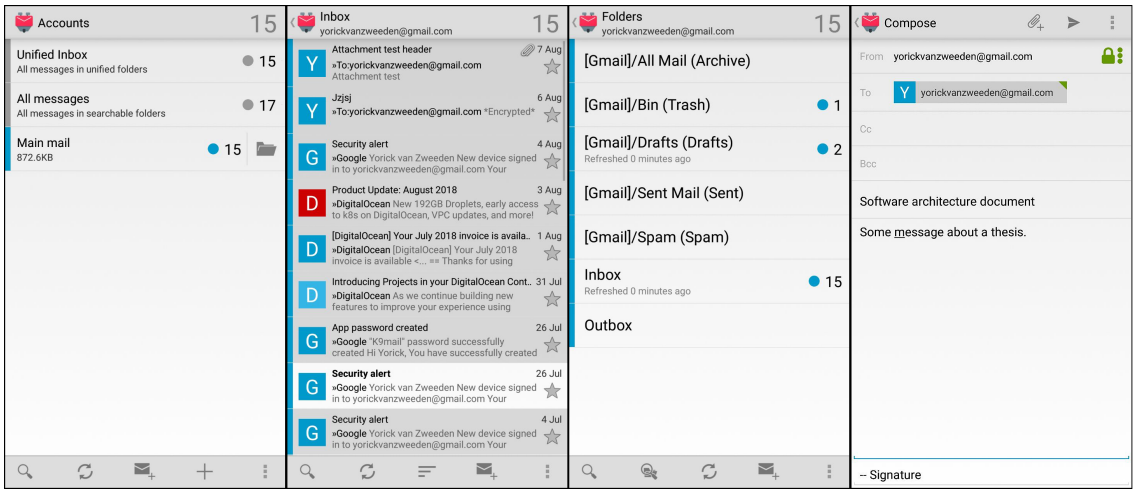

Figure 6.1: Screenshots of the K-9 Mail app

FAML

The FAML was created after all ACL views were mapped. As the FAML contains a lot of relations, Graph visualisation software was used to find a minimal version with fewer overlapping connections. The result is shown in Figure 6.2. Three modules provided the core functionality: Compose mails, Message list, and Accounts. Accounts includes the Folder functionality as these were very similar in their dependencies on other components in the ACL views. The other modules provide smaller functionality or supported these three modules. An example of a support module would be the Upgrade Databases module (Appendix A.3). This module could be triggered by any of the three main modules and would upgrade the databases. On completion, the initiating core module would be started.

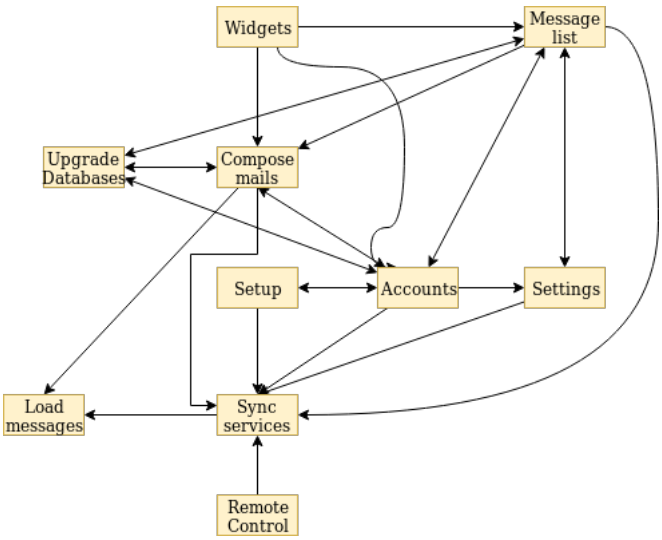

Figure 6.2: The FAML view of the K-9 mail app

ACL Views

We created an ACL view for each module for a total of ten. We'll discuss two of them and the rest can be found in the previously mentioned GitHub repository. We are unable to include the

views in the Appendix, as some views are too large to fit on one sheet of paper. The first view to discuss is the *Compose Mail* ACL view, as this view includes multiple interesting connectors. The starting point is the *MessageCompose* activity, which lies at the center of the view.

Compose Mail ACL view

The first connector is the RPC connection with the *RecipientPresenter*. As seen in Figure 6.1, a green lock appears next to contacts with verified PGP keys. This functionality is provided by *RecipientPresenter*, which is declared and initiated in *MessageCompose*. *RecipientPresenter* creates a bound service connection to *IOpenPgpService2*. This service is generated from an AIDL file and generates a service class to which connection can be made. The AIDL file is part of the plugin package *openpgp-api-lib*, which also provides support for encrypting messages. *MessageCompose* uses this functionality to encrypt messages when the user sends the message. A list of contacts is provided by a content resolver that connects to the built-in *Contacts* content provider. Using an implicit intent, the *MessageCompose* activity provides the ability to use other apps to pick attachments, such as a gallery app. All the light grey components are from other modules and are thus described in other modules. This also means that the *Compose Mail* module has outgoing connections to the modules that include these grey components (*Accounts*, *Upgrade Databases*, *Load Messages*). During the analysis of all components, we found an unused component. The activity *EmailAddressList* has not been used since 2012 but is declared in the *manifest* file. It used to provide the contacts but has been replaced. Developers of the K-9 Mail app may have forgotten to remove this class.

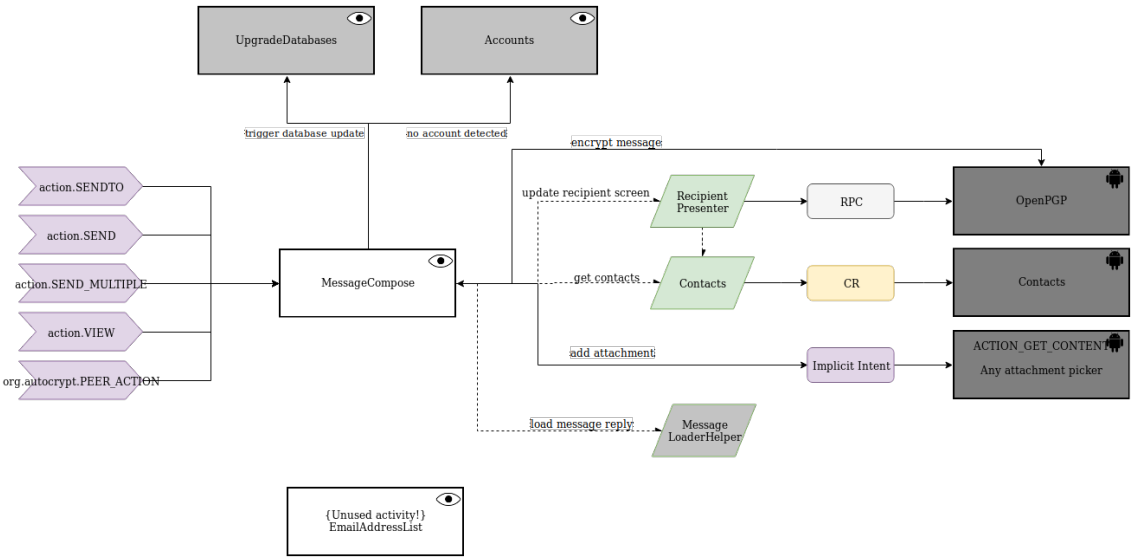

Figure 6.3: The ACL view of the Message Compose module

Load Messages ACL view

The previous module uses a *MessageLoaderHelper* class to load the message reply. This module shows how this is accomplished and explains why this is a separate module. Two modules use this functionality: *Compose Mail* and *MessageList*. Combining this view with either view of the two modules would create an unpleasantly large view. As both modules seemed to equally depend on this piece of functionality and combining was not an option, we chose to separate these components into a module. This module shows how Android architecture is not solely mediated by app components, but could heavily depend on other classes as well. The resulting view can be seen in Figure 6.4. For brevity, three classes were combined into a custom class called 'Multiple

classes'. While loading messages, attachments may be included. The *AttachmentProvider* contains a static method that provides the URI's for these attachments. If the message was encrypted, the attachment has a different URI. The *DecryptedFileProvider* uses Android's *FileProvider* to retrieve that URI. When the user turns off the screen, a receiver cleans these temporary decrypted files. This receiver is registered by the *DecryptedFileProvider*.

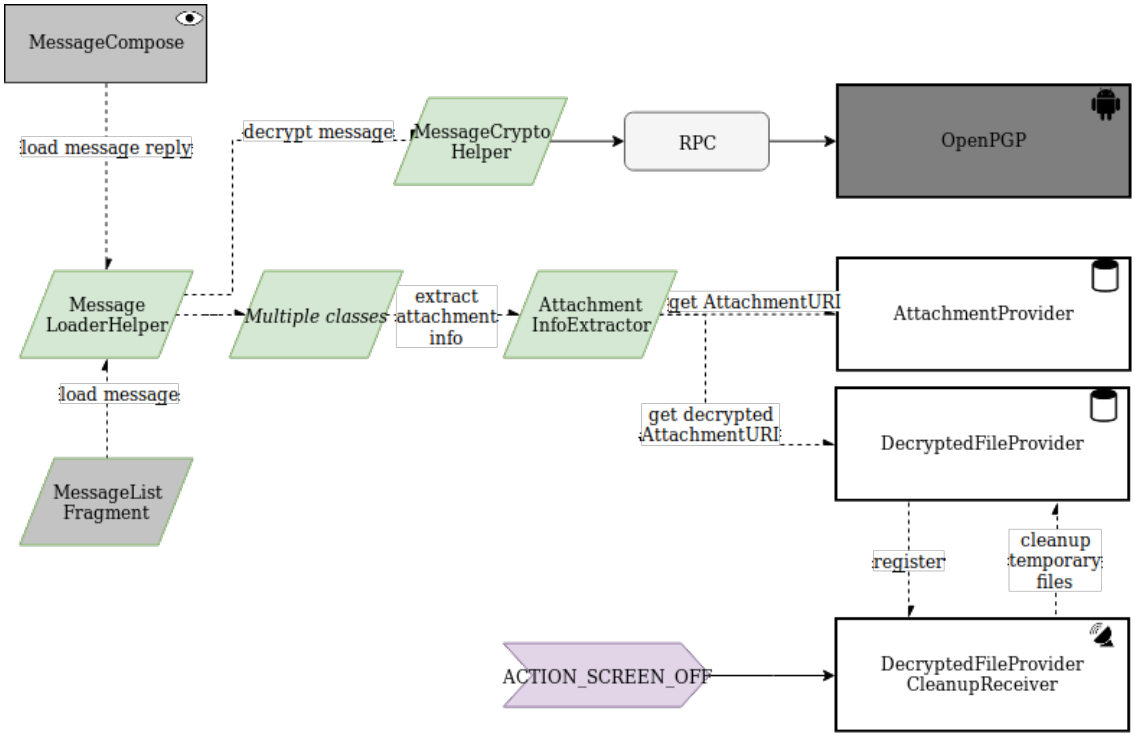

Figure 6.4: The ACL view of the Load Messages module

6.1.3 Evaluation

Positive findings

The case study of the K-9 Mail app indicates it is possible to use the artefact for creating AD for larger apps. Having three layers provides us with the ability to split up the application into manageable pieces, which can individually be modelled. The addition of the custom class component allows for documenting those classes which play an important role but are not an app components themselves. The process of documenting architecture requires looking at all the defined app components. In the process, we found multiple chunks of code and an entire activity that were unused but used to have a function. The code size of the app could be reduced by removing these unused methods and classes. IDE's usually detect when methods are unused as they are not called. However, the *onCreate* events are shown to be always called. Detecting unused app components in large applications is, therefore, more difficult.

Creating the modular division

The hardest part of creating AD for K-9 was developing a proper FAML. Some components, such as content providers and activities, were used in different modules. As we would like to designate each component to one module and thus reducing the overlap between views, choosing between modules becomes a problem. It is not obvious to which module a component should be assigned. A solution could be to create a separate module, but it would be illogical if the

module consists of one component. An example of such a problematic component is *ChooseFolder* activity, which is used by *Widgets*, *Accounts*, and *MessageList*. As we didn't want to create a module consisting of one component, we decided to let the component reside in *MessageList*. Aside from activities, providers are often used throughout the application and bring about these difficult choices. *UpgradeDatabases* is an example in which we made the choice to separate the components in a different module (Appendix A.3). A downside of this choice is that a module and three connections are added to the FAML and thereby further increasing complexity in this view.

Static methods

Another difficulty posed by the K-9 mail app were the statically defined methods. A lot of components defined static methods for creating an intent to themselves. An example code snippet is shown below (6.1):

```
public class PollService extends CoreService {
    public static void startService(Context context) {
        Intent i = new Intent();
        i.setClass(context, PollService.class);
        i.setAction(PollService.START_SERVICE);
        context.startService(i);
    }
}
```

Listing 6.1: Code example of static method that fires an intent

Any component that calls this method is no longer using intents directly. As a result of that, HUSACCT reports an intent from *PollService* to *PollService* and a call from any class that calls this method. The script *parseHUSACCT.py* (B.2) could not simplify detection and a lot of manual work was needed to find these static intent methods.

Android libraries abstracting and removing code

Android tries to remove boilerplate code for developers and speed up development by creating libraries. K-9 uses such a library called the Preferences API. This API allows developers to create a full-fledged settings menu more easily. The developer is required to define the settings structure and menu in XML-files and the library takes care of the rest. This becomes a problem when the XML-files allow intent specification which library converts to Java intents. This clouds the ability to understand where intents are defined, what they target and what their purpose is. Another upcoming library called Navigation should simplify the implementation of navigation in Android apps. Navigation was announced in 2018 and uses XML and a special editor to accomplish this.

Inconsistency on modelling incoming connections and providing text

Intent filters indicate the entry points of applications. However, intent filters also show the entry point of a module. It would be inconsistent to exclude other types of entry points, such as components in other modules that call components in the current module. In the process of documenting the K-9 Mail app, we did not model these 'incoming connecting components' in the *Compose Mail* ACL view, but we did for *Load Messages* ACL view. The *Load Messages* ACL view would have no entry points if the components using *MessageLoaderHelper* were not modelled, but this does not apply to *Compose Mail*. It remains unclear if components of other modules should always be modelled, as the addition of external components gives more insight, but increases the complexity of the views.

6.2 Creating AD for the OmniNotes app

We will describe the OmniNotes app first. This will be followed by an explanation of the results. Lastly, we evaluate the results and the process.

6.2.1 Introduction to OmniNotes

The OmniNotes app is an open-source note-taking application with support for setting reminders, attaching attachments and creating categories for each note. The most important activity of the app is the *MainActivity*, which shows the list of notes and allows for taking notes. Other relevant activities allow for changing settings, creating and editing categories, showing attachments and showing information about the notes. Figure 6.5 shows the *MainActivity*, the *NavigationDrawer* (fragment of *MainActivity*) and the *Settings* activity.

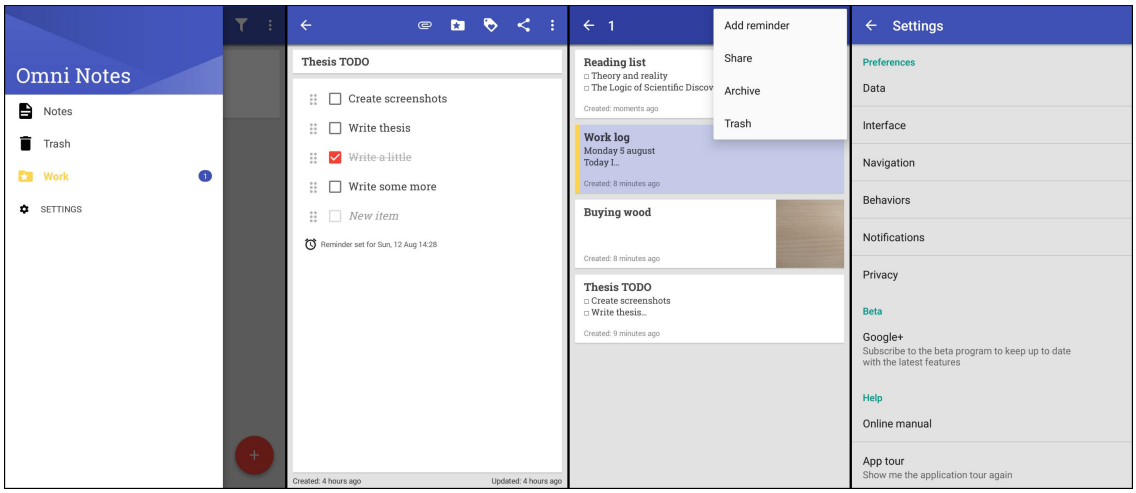

Figure 6.5: Screenshots of the OmniNotes app

6.2.2 Results

The architecture of the OmniNotes app was documented using the artefact an additional support of analysis tools. These tools include Python scripts, grep commands, regex queries, and HUSACCT dependency analysis. HUSACCT failed to report all dependencies and was only used partly. All of the views created can be found at https://github.com/yorickvanzweeden/bachelor-thesis. We modelled some Code Layer views, but not all of them due to time constraints.

FAML views

We started by identifying modules on the basis of the features presented by the app. Originally, we included a module called *Passwords*, but removed this as we could easily place it within *Settings*. During the process of modelling the ACL views, we included the Storage module as this was deemed necessary. As shown in Figure 6.6, the centerpiece of the application is the *MainActivity* and all of its fragments. All of the other modules connect to it and this module is also the largest in terms of lines of code (around 5K). It is a simpler FAML than the K-9 Mail app, which could be expected from a smaller application.

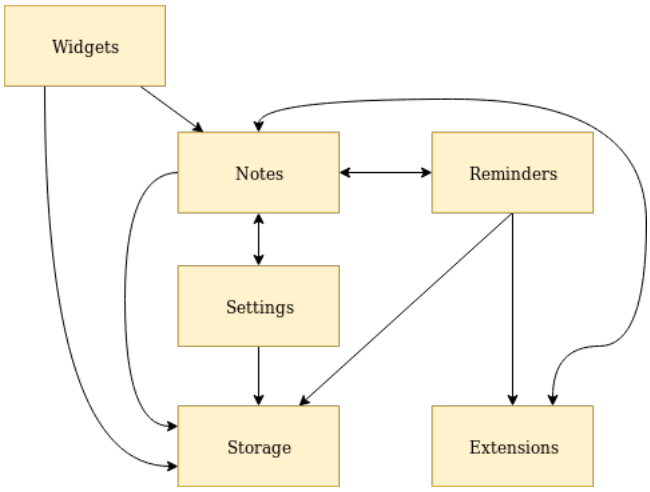

Figure 6.6: The FAML view of the OmniNotes app

ACL views

Each module has one corresponding ACL view. In the previous case study, we showed two ACL views. The views created for this case study are no different in terms of concepts and structures. The ACL view of the Notes module is the most relevant to the app, but this view is too big to properly show on paper. The Storage ACL view is interesting as it contains no app components. The view is shown in Figure 6.7. This view was created because three classes are using content resolvers and a lot of other classes depend on this module. In the evaluation of this case study, we will explain why this view is problematic.

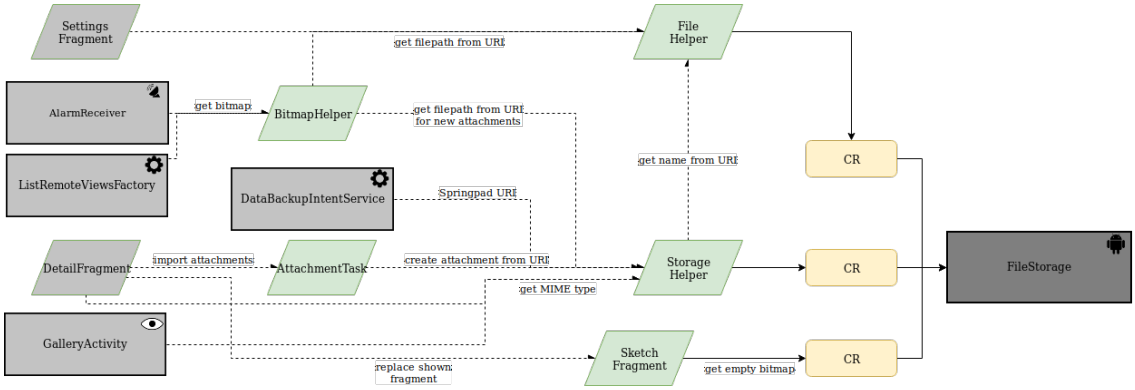

Figure 6.7: The ACL view of the Storage module

CL views

Due to time constraints, we limited ourselves to creating CL views of app components. In the previously mentioned GitHub repository not all created CL views are shown, as some of the views are very similar or are not interesting enough. We encourage the reader to look at the CL view of *MainActivity*, which we can not discuss here, as the view is too large to display properly on paper. Instead, the view of *GalleryActivity* will be explained (Figure 6.9). As the structure of CL views is very simple, we will only clarify one view.

The *GalleryActivity* shows an image or video and a menu. It is accessed via the *DetailFragment* which shows a note and its attachments. The activity overrides four events: *onCreate*, *onStart*,

onCreateOptionsMenu, and *onOptionsItemSelected*. The *onCreateOptionsMenu* event is overridden in order to set a custom menu as shown in Figure 6.8. The menu is created by inflating an XML file.

Figure 6.8: Custom menu of the GalleryActivity

The *onOptionsItemSelected* callback is used to handle menu button presses of the user. Three actions are defined for three buttons. In the CL view, this is represented using an event triangle and three choice events. Each of the choices leads to a task. The *onCreate* callback is mainly responsible for showing the attachments and making sure that scrolling left and right works. Two listeners are registered by this method. The first listener *onViewTouchOccurred* plays the video if the attachment is a video and the user has clicked (instead of swiped). The second listener *onPageSelected* changes the subtitle to '(2/2)', if the user swipes to the second attachment. In the CL view, these two event listeners are connected to the tasks which create them. The *onStart* method just starts analytics tracking, a library that sends anonymous data to the developer for improvement. The complete Code Layer view of this activity is shown in Figure 6.8.

6.2.3 Evaluation

Positive findings

This case study shows that CL views can be created for Android apps. Although HUSACCT did not report all dependencies, using Android Studio and grep this only proved to be a minor inconvenience. We suspect this would not be the case for larger applications.

Problems with the CL views

We detected a couple of problems with the CL views:

- CL views are unsuitable for very large classes. Although this may be a code smell, classes of 500 lines already proved to be too large to display on a single piece of paper.
- Loops are not supported by the CL view and will be represented by tasks. This may result in a loss of understanding of the code.
- Switch statements are very tedious to model in the CL view if they contain a lot of cases.
- Some methods are used multiple times by other methods. An example of this can be found in the ACL view of the MainActivity. We did not want to create duplicates, so we connected the task to the chain of the called method. Our solution is shown in Figure 6.10. It would be increasingly difficult to model more calls to the same method.

PendingIntents and AlarmManager

Android provides constructs to create Intents and pass them to other classes for them to be fired later. These intents are called PendingIntents. OmniNotes used *PendingIntents* in order to create actions for each row in a listview. It is unclear whether the intents should be modelled at the component of creation or the component that fires the intent. A similar problem arises with the *AlarmManager* library which allows developers to schedule intent broadcasts at a later moment. Such constructs make analysis and modelling harder.

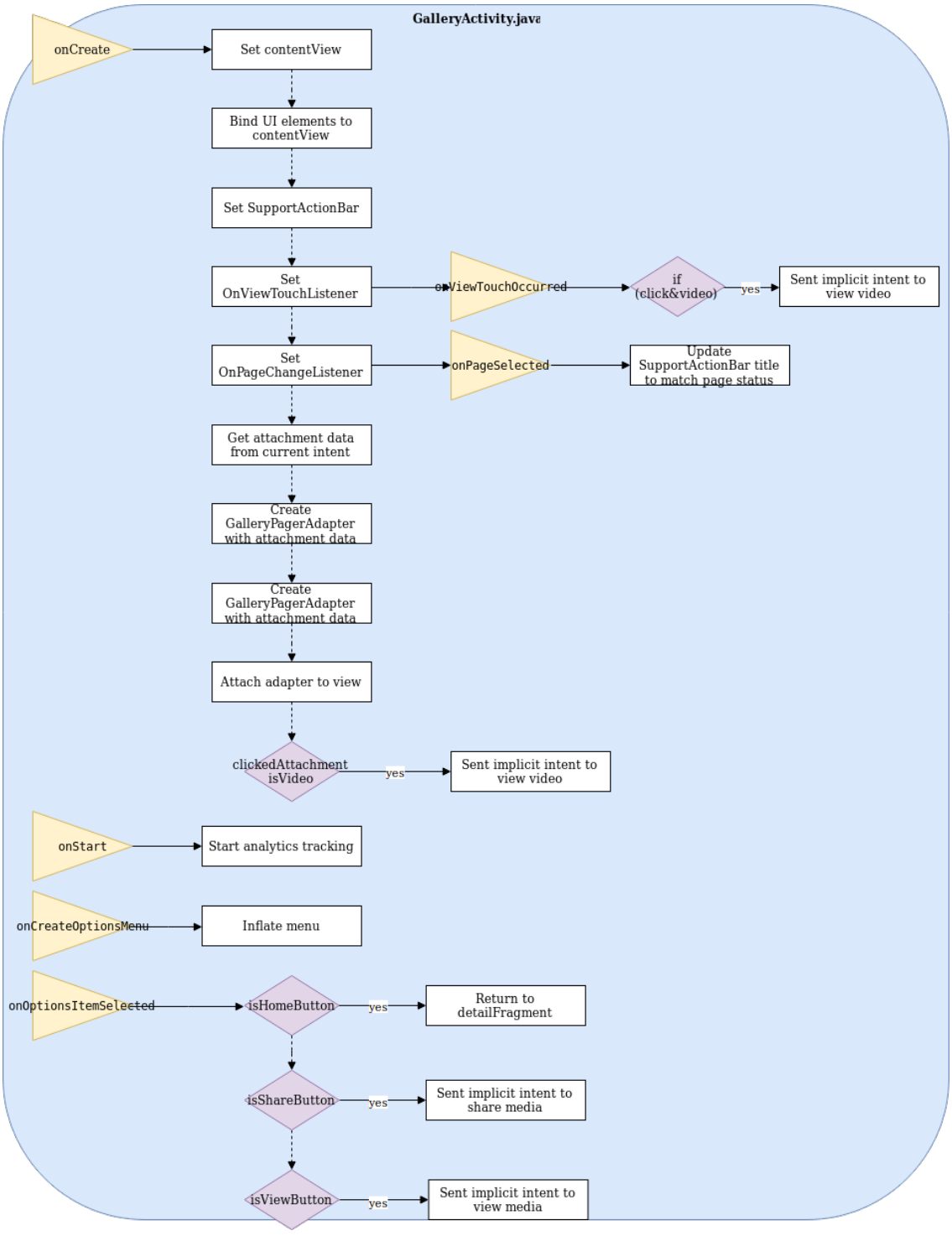

Figure 6.9: The CL view of the GalleryActivity

EventBus

OmniNotes uses the *EventBus* library. This library creates an implicit event bus to which events may be posted and components may subscribe to. This allows for communication throughout the app. Mapping where events were created and what components subscribed to these events was

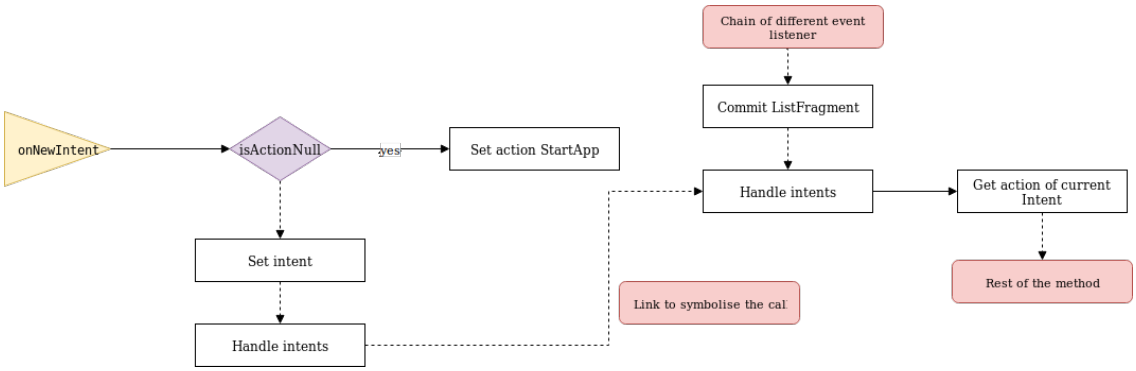

Figure 6.10: Multiple methods calling the HandleIntents method

not difficult. However, modelling this behaviour proved to be harder. In order to reduce overlap (REQ2), we enforced the restriction to not model the same component in different modules. If a reference was made to a component in another module, we used the grey symbol of the component type. The EventBus was is used in a lot of different modules and thus connected different modules. This did not create problems in this case study, as very few subscribed classes created intents or other connections. However, if these kinds of libraries are used heavily, it may be a headache to create correct views.

Static methods

In the ACL view and the CL view static methods are challenging. If a class has a static method that performs a generic function, it is unclear if the model should mention the class. Figure 6.11 shows an example in which we have to make a decision whether to exclude or to include Class B in our model. The problem extends to the ACL with the static methods of custom classes.

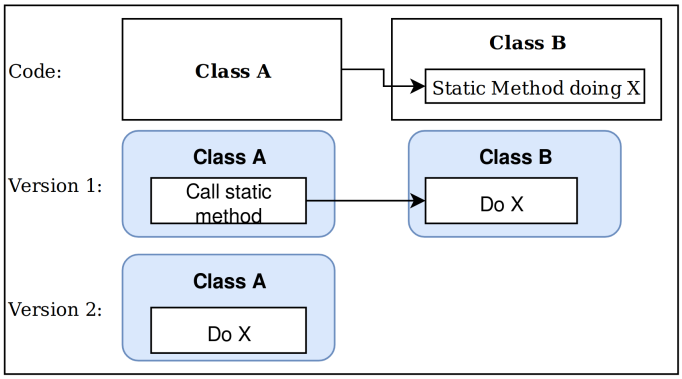

Figure 6.11: Static methods in the CL view

Content providers

We made the choice to exclude calls to the data storage in our set of views. We will discuss this choice in the evaluation of the artefact. In short, we made this choice as data storage is required in most parts of applications and including this in views would complicate them. Nevertheless, we had to model some parts of data storage for the OmniNotes app. The *Storage* module is unsurprisingly used by most other modules. The reason we had to create a separate module for *Storage* is that this module included three content resolvers which are connected to content providers. As we wanted to include all components and connectors, we were forced to map the

elements from the rest of the application to the content resolvers. This is shown in the ACL view of the *Storage* module (Figure 6.7). OmniNotes uses internal file storage for attachment data storage. The helpers were used to convert content URIs to file paths and acted as a middleman for saving attachments to internal file storage. This has resulted in a view used by most modules without any actual app components.

StartActivityForResult & setResult

We have identified an asymmetric connection in our set of views. Components may use the *StartActivityForResult* class to start an activity with an intent. The started activity would return the result using the method *setResult*. In our ACL views, we would model the intents fired, but not the intent communication back to the activity. This is asymmetrical and the communication should be modelled in both directions.

HUSACCT did not report all dependencies

Although this did not pose a problem in this case study, HUSACCT failed to show all the dependencies for the OmniNotesApp. If the application under examination would be larger, this may be harder to detect. As HUSACCT could be of great value in the process of creating architecture documentation, it is paramount that all dependencies are reported.

6.3 Evaluation

We will first evaluate the artefact on the basis of the requirements of Section 5.2. We will then present key problems with the artefact that need to be solved in later research.

6.3.1 Requirement 1: Providing the right level of detail independent of app size

The goal was to be able to handle every app size. Our first case study was the K-9 Mail app which contains 90K lines of Java code and for which we were able to create a FAML view and ACL views. To underscore that creating architecture documentation using the FAML views and ACL views is possible for most applications, we analysed the size of 94 applications on Wikipedia's list of free and open-source applications for Android. Of the 94 applications, 11 were written in C or C++ using Android NDK. Only 7 applications of the remaining 83 were larger than the K-9 Mail app. The largest application written in Java is the Chromium browser with 636.288 lines of code. The list of apps with their size can be found in Appendix A.1. Although it is uncertain that the artefact is suitable for modelling the Chromium browser, it will be suitable for a very large portion of Android apps.

However, the Code Layer views have proved to be tedious to create and unfit for large classes. We don't consider them suitable for large applications as we find it improbable that all classes would contain less than 500 lines of code.

6.3.2 Requirement 2: Different views should not overlap or be too fragmented

The second requirement specified that different views should not overlap or be too fragmented. In order to reduce fragmentation, we model every component, and in order to reduce overlap, we model each component once. References to components from other modules are marked using the grey icon of the component type. However, three specific problems exist with this strategy that were detected in the OmniNotes study:

- It is not certain that all components have been identified, as Android libraries may conceal connectors from the procedure. Intents coded in XML that are fired via the Preferences API are an example.
- If the app uses internal file storage and we require to model all content resolvers, we may end up with a storage module that is used in most other modules.
- Third-party libraries, such as EventBus, may be used in every module due to their purpose. Custom solutions must be found for exceptions of this kind.

6.3.3 Requirement 3: Reduce semantic scope with Android-specific elements

As apps use app components very often, most of the complexity of the apps could be captured using the elements of the artefact. However, during the case studies, we found that the complexity of libraries like the Preferences API can't be fully mapped using the standard set of components. Other aspects of the Android apps were missing as well; such as data storage and lifecycle management.

6.3.4 Requirement 4: The views should address concerns

The process of creating architecture documentation using the artefact allows architects to obtain a clear understanding of what functionality is offered by the application and how this is structured. Additionally, 'dead code' was often found in the process and should be removed to reduce code size. The views show the usage of patterns and could enable more reuse in future development. They also show where IPC is used and which components communicate with each other.

Not all concerns are addressed by the views:

- Lifecycle management is very important in Android and poor management results in memory leaks and mediocre performance. All the activities, fragments and services have a lifecycle which should be managed.
- Information storage on Android comes in many forms; there are databases, shared preferences, and file storage. This is important as most app components process information.
- Some functionality is enabled by custom classes or Android specific libraries. As we only concern ourselves with the defined set of components and connectors, some of the complexity driving the functionality is excluded from the views.

Addressing these concerns in the existing set of views would merely complicate them, but we currently offer no solution for addressing them.

6.3.5 Other remarks

Hierarchical decomposition rules

One of the key problems of this artefact is that there are no exact rules for going from one layer to another layer. Dividing the components in the ACL view into modules for the FAML view requires creativity and interpretation. An example: Components A to D must be divided into three modules. Figure 6.12 shows two different decompositions that are deemed reasonable for this set of components and connectors. However, the different decompositions lead to different FAML views and ACL views. Although the app stays the same, different views are created.

This problem does not exist in the translation of ACL components into CL views, as the scope of components requires little interpretation. However, the CL views offer no method to go to the ACL view. Therefore, it is possible to create architecture documentation without CL views, which raises questions about the usefulness of the Code Layer. As creating CL views is tedious and every code change would require a view update, we suspect that this view would not be used often in practice.

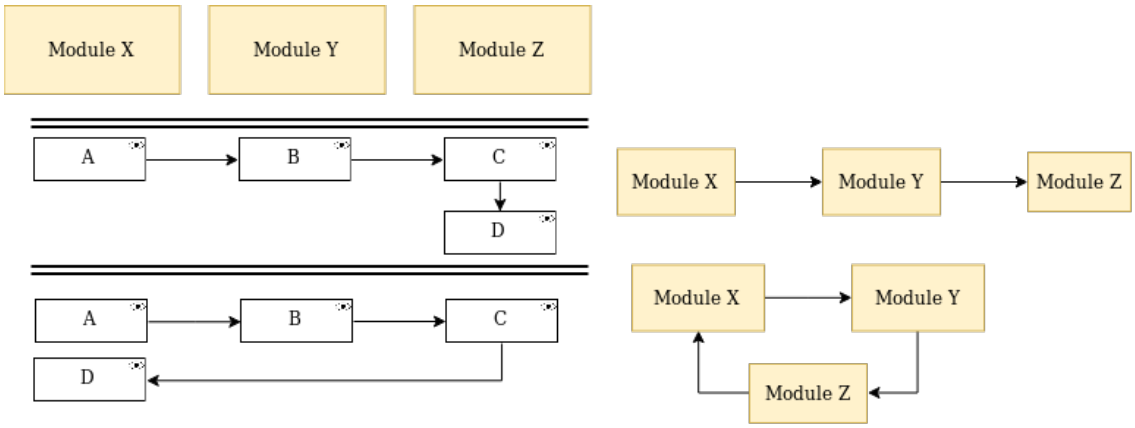

Figure 6.12: Conflicting decompositions in the FAML views

Components

The intent filter component has different roles, which can be confusing. In the early stages of the artefact, it was meant to show the entry points of the application. However, intent filters also show entry points of modules and are used as broadcasts.

Content providers have caused trouble during the case studies. They are often used in multiple modules which obstructs the designation to a specific module. They are interacted with in two ways: content resolvers and static methods. In the previous section, it was explained why static methods are challenging. Content providers may be used closely with internal file storage resulting in storage modules.

Some concepts are poorly modelled in the views, such as PendingIntents, the Preference API and Viewmodels. A solution for Android specific concepts that are currently not properly expressed, should be developed.

Chapter 7

Conclusions, Limitations and Future Work

7.1 Conclusions

In the last decade, Android has grown to a platform that has more than 2.3 billion active devices. The platform often ships with the Google Playstore, which is the largest app store with 3.8 million apps. Despite this enormous growth and the societal impact Android has brought about, research in mobile software engineering is not as developed yet compared to traditional software engineering. This is very unfortunate, as mobile apps face the same challenges as traditional software engineering. Moreover, Android is rapidly evolving and creating new libraries while deprecating older libraries. The case studies showed that both apps were using deprecated libraries. Architectural descriptions may help architects design apps with future changes in mind. Additionally, documenting architecture aids in the development of apps as it can serve as a blueprint and allow better communication between stakeholders.

This thesis is first in trying to establish architectural descriptions for Android apps with the purpose of documentation. Using the design science research method, three layers of views have been developed that target different concerns for Android apps. The three types of views have little overlap or fragmentation and are suitable for documenting any Java-written app. The case studies have shown that it is possible to document smaller and larger apps. Although more research is needed, this thesis provides a good indication of what elements are required for creating Android architectural descriptions.

7.2 Limitations

This research has a couple of limitations concerning the demonstration and evaluation of the artefact. The case studies were conducted on two open-source apps. These apps were selected without a specific rationale other than picking a smaller and larger app. Appendix A.1 has shown that the K-9 Mail app is a relatively large app, but the biggest apps are at least five times larger in terms of LoC. It would have been more convincing to create architectural descriptions for these apps. Additionally, both apps in the case studies did not use new libraries that Google announced in 2017 to combat architectural problems. Examining how these new libraries would get along with the created artefact is an opportunity we missed out on. The products of the case studies were not evaluated by the creators of the documented apps. Therefore, we have no good understanding of how the set of views will be perceived by real stakeholders.

7.3 Future work

Towards a perfect architectural description for Android apps

The evaluation of the artefact has shown some critical flaws in the set of views. Future research is required to improve the procedure for creating architectural descriptions. A solution must be found for dealing with the variety of Android libraries, the modelling of static methods and the scalability of the Code Layer. Additionally, the content provider and intent filter in the App Component Layer should be reviewed.

The procedure for the process of documenting requires better specification while still allowing the creativity of the architect and adaptability for different kinds of apps. The three layers are connected, but this connection is not strictly defined. This has resulted in difficulties creating the Functional Architecture Model Layers and a one-sided connection between the Code Layer and the App Component Layer.

Automation

In future work, more improvements in automation can be made. The mechanical detection of the selected elements has greatly reduced the labor for creating architectural description in our experience. When the detection is fully automated, the cost of creating documentation is vastly reduced. However, key problems for automation are Android libraries that hide Java code behind XML-files, the introduction of the Kotlin language for building Android apps and the usage of the Android NDK to develop apps with C/C++. Furthermore, the rules for assigning components to modules are unclear, and encounters with exceptions constrain the possibilities of automation and necessitate human creativity and insight.

Software architecture evolution

Nevertheless, reduced costs of creating software architecture may provide great benefits. In the process of conducting case studies, git was used to understand design decisions. Future work may show how the evolution of the architecture can be displayed using version control and a more automated version of this artefact. Showing the evolution of architecture may help architects plan for future feature requests and handling the rapid changes of the Android platform.

Applying graph algorithms

Multiple graph algorithms were used during the case studies. We used graph algorithms for reducing the energy of our FAML models and make them more readable. Additionally, we realised that the dependencies formed a graph and existing algorithms could be applied to gain more knowledge. The K-9 Mail app had an unused activity, which formed a disconnected island in the dependencies. Previous research has shown that reachability analysis can be used to show these kinds of 'dead code' [24], but has not explicitly linked this to Android software architecture. The combination of reachability analysis and software architecture evolution may explain why parts of the code have died and what design decisions lead to this.

Graphing algorithms may aid in the detection of modules. Future work may use algorithms for finding strongly connected components (SCC) in Android apps and finding ways to map the SCCs to FAML modules.

Compliance checking

Lastly, we point out that our research may aid in developing architecture compliance checking for Android apps. Currently, only an approach has been published for detecting architectural technical debt through automated compliance checking [33]. Verdecchia [33] mentions the usage of model comparison tools with respect to the Android reference architecture. Our research may aid in modelling the implemented architecture and defining the reference architecture.

Bibliography

[1] H. Bagheri, E. Kang, S. Malek, and D. Jackson. Detection of Design Flaws in the Android Permission Protocol through Bounded Verification. In *International Symposium on Formal Methods*, pages 73–89. Springer, 2015. ix, 13, 14, 17, 18

[2] L. Bass, P. Clements, and R. Kazman. Software architecture in practice. 2012, 2012. 6

[3] S. Brinkkemper and S. Pachidi. Functional Architecture Modeling for the Software Product Industry. In *European Conference on Software Architecture*, pages 198–213. Springer, 2010. 18

[4] R. France and B. Rumpe. Domain specific modeling. *Software and Systems Modeling*, 4(1):1–3, 2005. 1, 17

[5] D. Garlan. Software Architecture: a Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101. ACM, 2000. 1, 6

[6] D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of component-based systems*, 68:47–68, 2000. 13

[7] Google. Android and architecture. Available at `https://android-developers.googleblog.com/2017/05/android-and-architecture.html`. 12

[8] Google. Android architecture components. Available at `https://developer.android.com/topic/libraries/architecture/`. 12

[9] Google. Application fundamentals. Available at `https://developer.android.com/guide/components/fundamentals`. 9

[10] Google. Bound services overview. Available at `https://developer.android.com/guide/components/activities/activity-lifecycle`. 11

[11] Google. Broadcasts. Available at `https://developer.android.com/guide/components/broadcasts`. 12

[12] Google. Content provider basics. Available at `https://developer.android.com/guide/topics/providers/content-provider-basics`. 12

[13] Google. Introduction to activities. Available at `https://developer.android.com/guide/components/activities/intro-activities`. 11

[14] Google. Loaders. Available at `https://developer.android.com/guide/components/loaders`. 25

[15] Google. The navigation architecture component. Available at `https://developer.android.com/topic/libraries/architecture/navigation/`. 13

[16] Google. Running a service in the foreground. Available at `https://developer.android.com/guide/components/services#Foreground`. 11

[17] Google. Schedule tasks with workmanager. Available at https://developer.android.com/topic/libraries/architecture/workmanager. 13

[18] Google. Services overview. Available at https://developer.android.com/guide/components/services. 11

[19] Google. Understand the activity lifecycle. Available at https://developer.android.com/guide/components/activities/activity-lifecycle. 11

[20] A.R. Hevner, S.T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *Management Information Systems Quarterly*, 28(1):6, 2004. 4, 5

[21] R. Hilliard. Views and Viewpoints in Software Systems Architecture. In *Position paper from the First Working IFIP Conference on Software Architecture, San Antonio*, 1999. 6, 7

[22] C. Hofmeister, R.L. Nord, and D. Soni. Describing Software Architecture with UML. In *Software Architecture*, pages 145–159. Springer, 1999. 7

[23] P.B. Kruchten. The 4+1 View Model of Architecture. *IEEE software*, 12(6):42–50, 1995. 7

[24] É. Payet and F. Spoto. Static Analysis of Android Programs. *Information and Software Technology*, 54(11):1192–1201, 2012. 47

[25] K. Peffers, T. Tuunanen, M.A. Rothenberger, and S. Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 2008. 4, 5

[26] L. Pruijt. *Instruments to Evaluate and Improve IT Architecture Work*. Utrecht University, 2015. 29

[27] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives*. Addison-Wesley, 2012. 1, 6, 7, 8, 15, 16, 17, 18

[28] B. Schmerl, J. Gennari, A. Sadeghi, H. Bagheri, S. Malek, J. Cámara, and D. Garlan. Architecture Modeling and Analysis of Security in Android Systems. In *European Conference on Software Architecture*, pages 274–290. Springer, 2016. ix, 13, 14, 17

[29] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In *Computer Science Today*, pages 307–323. Springer, 1995. 7, 17

[30] Statista. Number of apps available in leading app stores as of 1st quarter 2018. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. 9

[31] M.D. Syer, M. Nagappan, A.E. Hassan, and B. Adams. Revisiting Prior Empirical Findings for Mobile Apps: An Empirical Case Study on the 15 Most Popular Open-Source Android Apps. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pages 283–297. IBM Corp., 2013. 1

[32] Bernd van der Wielen (Newzoo). Insights into the 2.3 billion android smartphones in use around the world. Available at https://newzoo.com/insights/articles/insights-into-the-2-3-billion-android-smartphones-in-use-around-the-world/. 9

[33] R. Verdecchia. Identifying Architectural Technical Debt in Android Applications through Automated Compliance Checking. 2018. 47

[34] A.I. Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 397–400. ACM, 2010. 1, 13

[35] J. Webster and R.T. Watson. Analyzing the Past to Prepare for the Future: Writing a Literature Review. *MIS Quarterly*, pages xiii–xxiii, 2002. 4

[36] C. Wohlin. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, page 38. ACM, 2014. 4

[37] C. Wohlin. Second-Generation Systematic Literature Studies using Snowballing. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, page 15. ACM, 2016. 4

Appendix A

Figures & Tables

A.1 Code Layer view of HelloService

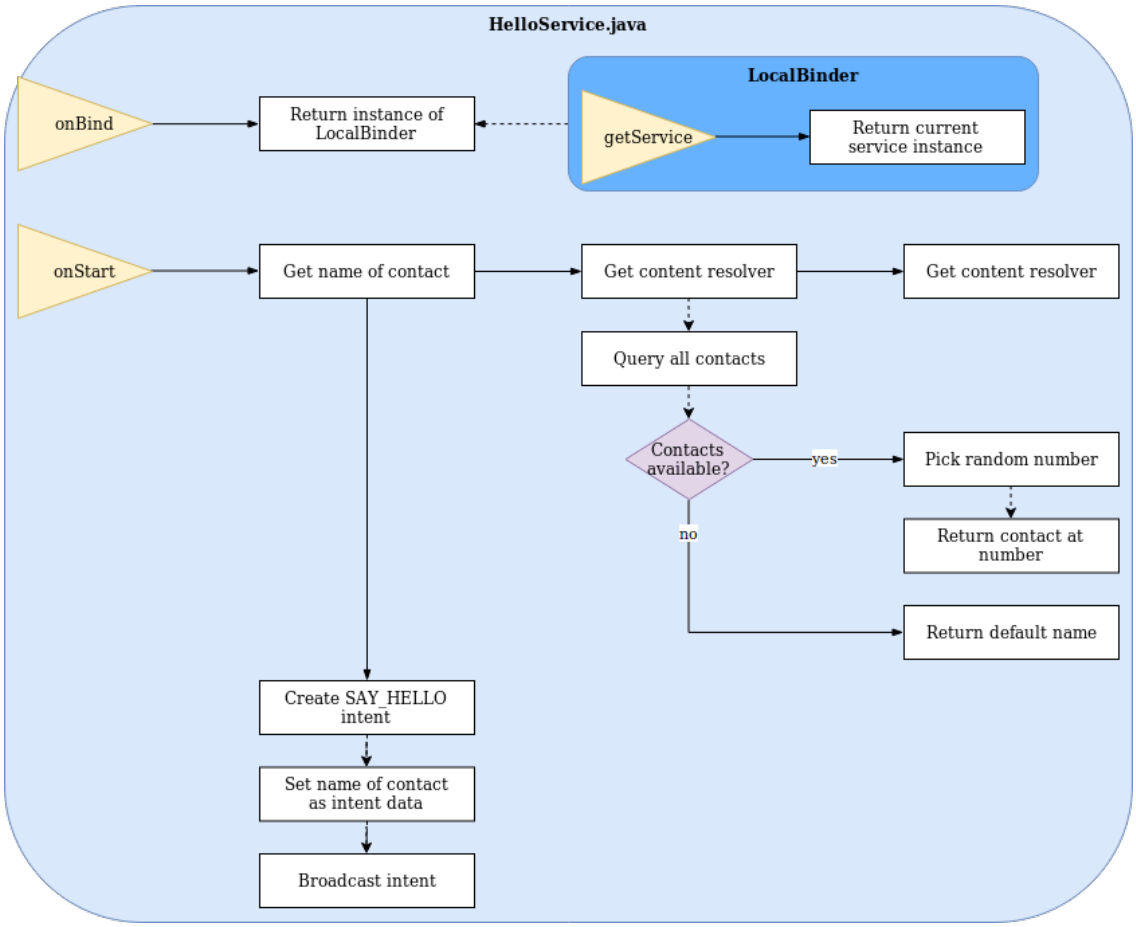

Figure A.1: Code Layer view of HelloService of the example application

A.2 Code Layer view of SecondActivity

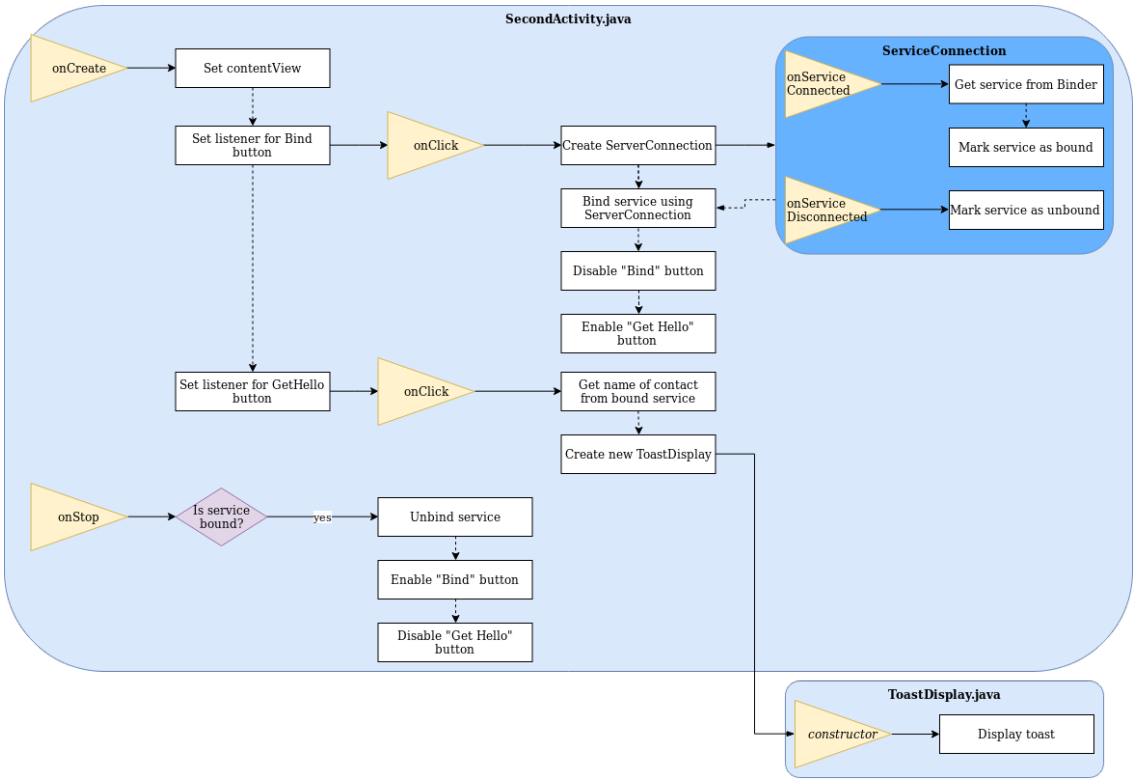

Figure A.2: Code Layer view of SecondActivity of the example application

A.3 App Component Layer view of UpgradeDatabases

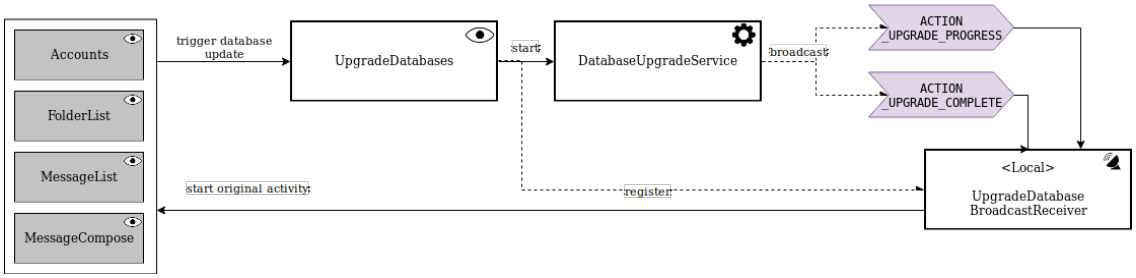

Figure A.3: App Component Layer view of UpgradeDatabases (K9)

A.4 Lines of Java code per application

Name	Java LoC	Code repository
Chromium	636288	https://github.com/chromium/chromium.git
Brave	623525	https://github.com/brave/browser-android-tabs.git
Telegram	382829	https://github.com/DrKLO/Telegram.git
OsmAnd	291818	https://github.com/osmandapp/Osmand.git
XOWA	241764	https://github.com/gnosygnu/xowa.git
Firefox for Android	123465	https://hg.mozilla.org/mozilla-central
WordPress	118190	https://github.com/wordpress-mobile/WordPress-Android.git
K-9 Mail	92839	https://github.com/k9mail/k-9.git
Signal	82923	https://github.com/signalapp/Signal-Android.git
OpenKeychain	72732	https://github.com/open-keychain/open-keychain.git
FBReader	70860	https://github.com/geometer/FBReaderJ.git
Wikipedia	60758	https://github.com/wikimedia/apps-android-wikipedia.git
CSipSimple	58871	https://github.com/tqcenglish/CSipSimple.git
Google IO	57854	https://github.com/google/iosched.git
Nextcloud	57297	https://github.com/nextcloud/android.git
PasswdSafe	51402	https://github.com/Rushera/passwdsafe.git
Conversations	48171	https://github.com/siacs/Conversations.git
AntennaPod	47384	https://github.com/AntennaPod/AntennaPod.git
Popcorn Time	45135	https://github.com/popcorn-official/popcorn-android.git
Barcode Scanner	43668	https://github.com/zxing/zxing.git
APG	41992	https://github.com/thialfihar/apg.git
PressureNET	41882	https://github.com/Cbsoftware/PressureNet.git
ownCloud	40243	https://github.com/owncloud/android.git
Avare	40045	https://github.com/apps4av/avare.git
Fennec F-Droid	38948	https://github.com/f-droid/fdroidclient.git
Jitsi	37116	https://github.com/jitsi/jitsi-android.git
Surespot	36659	https://github.com/surespot/android.git
Orbot	29397	https://github.com/n8fr8/orbot.git
VLC	29176	https://github.com/videolan/vlc-android.git
Linphone	27527	https://github.com/BelledonneCommunications/linphone-android.git
GNU Ring	27315	https://git.ring.cx/savoirfairelinux/ring-client-android.git
NewPipe	25739	https://github.com/TeamNewPipe/NewPipe.git
Wire	25725	https://github.com/wireapp/wire-android.git
Mupen64Plus	24690	https://github.com/mupen64plus-ae/mupen64plus-ae.git
Sipdroid	24257	https://github.com/mconf/sipdroid.git
Mozilla Stumbler	23907	https://github.com/mozilla/MozStumbler.git
iFixit	21037	https://github.com/iFixit/iFixitAndroid.git
Wikimedia Commons	19019	https://github.com/commons-app/apps-android-commons.git
ConnectBot	18442	https://github.com/connectbot/connectbot
Prey	17809	https://github.com/prey/prey-android-client.git
KeePassDroid	17404	https://github.com/bpellin/keepassdroid.git
Angband	17002	https://github.com/sergeybe/angdroid.git
I2P	15994	https://github.com/i2p/i2p.android.base.git
Firefox Focus/Klar	14583	https://github.com/mozilla-mobile/focus-android.git
BOINC	13521	https://github.com/BOINC/boinc.git
micro:bit	13102	https://github.com/Samsung/microbit.git

Table A.1 continued from previous page

Name	Java LoC	Code repository
	12760	https://github.com/cryptape/Neuron-Android.git
OpenLP	10507	https://github.com/mcruncher/worshipsongs-android.git
MAPS.ME	10429	https://github.com/mapsme/omim.git
Kiwix	10164	https://github.com/kiwix/kiwix-android.git
Tribler	9433	https://github.com/Tribler/tribler-android.git
Briar	9291	https://github.com/kiggundu/briar/briar.git
Galaxy Zoo	8132	https://github.com/murraycu/android-galaxyzoo.git
Dasher	8020	https://github.com/GNOME/dasher.git
SageMath	7921	https://github.com/sagemath/android.git
AdAway	7744	https://github.com/AdAway/AdAway.git
Dolphin	7704	https://github.com/boonex/dolphin-android.git
Sugar environment	7492	https://github.com/chennaione/sugar.git
The White House	6882	https://github.com/WhiteHouse/wh-app-android.git
Dungeon Crawl Stone Soup	6214	https://github.com/crawl/sdl-android.git
Haven	5027	https://github.com/guardianproject/haven.git
DNS66	4824	https://github.com/julian-klode/dns66.git
Impress Remote	4704	https://gerrit.libreoffice.org/impress_remote.git
Ringdroid	4195	https://github.com/google/ringdroid.git
GLtron	3986	https://github.com/Chluverman/android-gltron.git
Warmux	3241	https://github.com/a-team/wormux.git
Frozen Bubble	2786	https://github.com/robinst/frozen-bubble-android.git
Tux Paint	2760	https://github.com/tux4kids/Tuxpaint-Android.git
MuPDF	2376	https://github.com/muennich/mupdf.git
PPSSPP	2362	https://github.com/hrydgard/ppsspp.git
Rockbox	2083	https://github.com/Rockbox/rockbox.git
Fish Fillets NG	2012	https://github.com/FishFilletsNG/ffng-android.git
Freeciv	1416	https://github.com/zielmicha/freeciv-android.git
Tox	1269	https://github.com/Antox/Antox.git
Wiktionary	1232	https://github.com/wikimedia/WiktionaryMobile.git
Psiphon	1179	https://github.com/Psiphon-Labs/psiphon-tunnel-core.git
RetroArch	1023	https://github.com/libretro/RetroArch.git
ScummVM	944	https://github.com/scummvm/scummvm.git
robotfindskitten	681	https://github.com/xxv/robotfindskitten.git
Kodi (formerly XBMC)	500	https://github.com/xbmc/xbmc.git
DuckDuckGo	402	https://github.com/duckduckgo/Android.git
Lantern	342	https://github.com/nordprojects/lantern.git
GCompris	188	https://github.com/gcompris/GCompris-qt.git
2048	136	https://github.com/uberspot/2048-android.git
Battle for Wesnoth	N/A (NDK)	https://github.com/cjhopman/Wesnoth-1.8-for-Android.git
Brogue	N/A (NDK)	https://bitbucket.org/rmtew/brogue
GNU IceCat	N/A (NDK)	https://ftp.gnu.org/gnu/gnuzilla/
H-Craft Championship	N/A (NDK)	https://bitbucket.org/mzeilfelder/trunk_hc1
OpenArena	N/A (NDK)	https://github.com/pelya/openarena-engine
openMSX	N/A (NDK)	https://github.com/openMSX/openMSX.git
OpenTTD	N/A (NDK)	https://github.com/pelya/openttd-android.git
OpenTyrian	N/A (NDK)	https://github.com/meh/opentyrian.git
Orfox / Tor Browser	N/A (NDK)	https://github.com/guardianproject/Orfox
Simon Tatham's Puzzle Collection	N/A (NDK)	https://github.com/ghewgill/puzzles.git

Table A.1 continued from previous page

Name	Java LoC	Code repository
VICE	N/A (NDK)	https://github.com/lubomyr/vice-2.4.git

Table A.1: Lines of Java code per application. The list of applications is taken from Wikipedia's list of free and open-source applications for Android

Appendix B

Scripts

B.1 ParseManifest Python script

```
# Get XML file
tree = ET.parse('/PATH/TO/APPLICATION/MANIFEST/src/main/AndroidManifest.xml')
root = tree.getroot()

# Create reversed tree
parent_map = {c:p for p in tree.iter() for c in p}

# Tag obtaining name attribute
name_tag = '{http://schemas.android.com/apk/res/android}name'

# Print a list of components from the Manifest file
def getComponent(componentType):
  print(f"##{componentType}##")
  for component in root.iter(componentType):
    print(component.attrib[name_tag])

# Print a list of intent filters from the Manifest file
def getIntentFilters():
  for component in root.iter('intent-filter'):
    f = component.find("action").attrib[name_tag]
    c = parent_map[component].attrib[name_tag]
    print(f'{f} -> {c}')

getIntentFilters()

getComponent("activity")
getComponent("service")
getComponent("receiver")
getComponent("provider")
```

Listing B.1: Script used for parsing Android Manifest files. This script is also available at https://github.com/yorickvanzweeden/bachelor-thesis

B.2 ParseHUSACCT Python script

```
import csv
import os
import re
import subprocess
from threading import Thread
from enum import Enum

JAVACLASSES = {}
DEPENDENCIES = []
MATCHES = {}

## Support for multithreading with return value
class ThreadWithReturnValue(Thread):
    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs={}, Verbose=None):
        Thread.__init__(self, group, target, name, args, kwargs)
        self._return = None
    def run(self):
        if self._target is not None:
            self._return = self._target(*self._args,
                                                **self._kwargs)
    def join(self):
        Thread.join(self)
        return self._return

## Analyses the source code files and imports the HUSACCT dependencies
class Setup(object):
  def __init__(self, filepath_repository, filepath_dependencies, namespace, matchfilter):
    self.regex_p1 = re.compile('\.[a-z].*\.[A-Z].*$|^[a-z]')
    self.regex_p2 = re.compile('\.[A-Z]')
    self.ignore = [namespace + ".R"]
    self.getJavaClasses(filepath_repository)
    self.getDependencies(filepath_dependencies)
    self.getMatches(matchfilter)

  # Create a dictionary of all Java classes
  # This is used for getting a code block using a HUSACCT dependency
  def getJavaClasses(self, filepath):
    global JAVACLASSES

    for root, dirs, files in os.walk(filepath):
        for file in files:
            if file.endswith(".java"):
              JAVACLASSES[file[:-5]] = os.path.join(root, file)

  # Detect innerclasses in HUSACCT dependencies and adds this relation to JAVACLASSES
  # This only works if it is actually an inner class. Two classes on the same level won't work
  def detectInnerclass(self, s, namespace):
    if namespace in s and not(s in self.ignore):
      s = s[12:]

      # Format string to [(Object.)*Class]
      x1 = self.regex_p1.search(s)

      if x1 != None:
        s = s[x1.span()[0]+1:]
        x2 = self.regex_p2.search(s)
        if x2 != None:
          s = s[x2.span()[0]+1:]

      # Split string
      values = s.split('.')
      path = JAVACLASSES[values[0]]
      for i in reversed(range(1, len(values))):
        JAVACLASSES[values[i]] = path

  # Parse the XML file of HUSACCT dependencies
  def getDependencies(self, filepath):
    with open(filepath, newline='') as csvfile:
      spamreader = csv.reader(csvfile, delimiter=',', quotechar='|')
      for row in spamreader:
        self.detectInnerclass(row[0], namespace)
        self.detectInnerclass(row[1], namespace)
        DEPENDENCIES.append(row)

  # Generate an array of matches based on file and linenumber
  def getMatches(self, dependency):
    for i in range(0, len(DEPENDENCIES)):
```

```
      row = DEPENDENCIES[i]
      if (dependency in row[1] and row[2] != "Import"):
        t1 = ThreadWithReturnValue(target=self.searchDependencies, args=("Search up ", row[0],
            i, -1))
        t2 = ThreadWithReturnValue(target=self.searchDependencies, args=("Search down", row[0],
             i, 1))
        t1.start()
        t2.start()
        results = t1.join()  + t2.join()

        if len(results) == 0:
          continue

        try:
          old = MATCHES[row[0]]
          new = [row[4], row[2], results]
          old.append(new)
          MATCHES[row[0]] = old
        except:
          MATCHES[row[0]] = [[row[4], row[2], results]]

  # Search for matching dependencies on file and linenumber
  def searchDependencies(self, threadName, file, startlinenr, searchDirection):
    i = startlinenr + searchDirection
    results = []
    while i >= 0 and i < len(DEPENDENCIES) - 1 and DEPENDENCIES[i][0] == file:
      if (int(DEPENDENCIES[i][4]) == int(DEPENDENCIES[startlinenr][4])):
        results.append(i)
      i += searchDirection
    return results

## Static class that prints blocks of code
class Lines():
  # Get lines using filename
  def getLines(file, linenr, offset):
    try:
      filepath = JAVACLASSES[file]
    except:
      file = Tools.convertNot(file)
      filepath = JAVACLASSES[file]

    cmd = f'cat {filepath} | head -{linenr + offset} | tail -{1 + 2 * offset}'
    return subprocess.check_output(cmd, shell=True).decode('UTF-8')

  # Get lines by filename and linenumber
  def getLinesByRange(file, start_linenr, end_linenr, offset):
    try:
      filepath = JAVACLASSES[file]
    except:
      file = Tools.convertNot(file)
      filepath = JAVACLASSES[file]

    cmd = f'cat {filepath} | head -{end_linenr + offset} | tail -{end_linenr - start_linenr +
        2*offset + 1}'
    return subprocess.check_output(cmd, shell=True).decode('UTF-8')

  # Get lines using dependency
  def printDependencyWithCodeLines(dependency, offset):
    print(Lines.getLines(dependency[0], int(dependency[4]), offset))

  # Get lines of code where matches start in target
  def getCodeLinesStartingFromTarget(component, offset):
    keys = list(MATCHES.keys())
    for key in keys:
      if component != None and Tools.convertNot(key) != component:
        continue
      # Preprocessing of matches
      matches = MATCHES[key]
      # Group by line number distance of max 3
      grouped_matches = Lines.groupLinesByLinenumber(matches, 3)

      for group in grouped_matches:
        # [match1, match2] but sorted by linenr
        linenr_start = group[0][0]
        linenr_end = group[len(group) - 1][0]

        for match in group:
          print(f"{Tools.convertNot(key)}:{match[0]}| {match[1]} \
            --> {[Tools.convertNot(DEPENDENCIES[x][1]) for x in match[2]]}")
```

```
        print(Lines.getLinesByRange(key, int(linenr_start), int(linenr_end), offset))

  # Get lines of code where matches end in target
  def getCodeLinesEndingAtTarget(component, offset):
    keys = list(MATCHES.keys())
    for key in keys:
      # Preprocessing of matches
      matches = MATCHES[key]
      # Group by line number distance of max 3
      grouped_matches = Lines.groupLinesByLinenumber(matches, 3)

      for group in grouped_matches:
        # [match1, match2] but sorted by linenr
        linenr_start = group[0][0]
        linenr_end = group[len(group) - 1][0]

        shouldPrint = False

        for match in group:
          if component in [Tools.convertNot(DEPENDENCIES[x][1]) for x in match[2]]:
            shouldPrint = True

        if not(shouldPrint):
          continue

        for match in group:
          print(f"{Tools.convertNot(key)}:{match[0]}| {match[1]} \
            --> {[Tools.convertNot(DEPENDENCIES[x][1]) for x in match[2]]}")

        print(Lines.getLinesByRange(key, int(linenr_start), int(linenr_end), offset))

  # Groups matches by distance of line numbers --> [[match1, match2], [match3]]
  def groupLinesByLinenumber(matches, separation):
    matches.sort(key=lambda x: int(x[0]))
    linenrs = [int(item[0]) for item in matches]
    grouped = []

    current_group = []
    prev_linenr = -1
    for i in range(0, len(linenrs)):
      # New matches --> Fill first group
      if (prev_linenr == -1):
        current_group.append(matches[i])
        prev_linenr = linenrs[i]
        continue

      # Already worked with matches
      current_linenr = linenrs[i]
      if (current_linenr - prev_linenr < separation):
        current_group.append(matches[i])
      elif len(current_group) > 0:
          grouped.append(current_group)
          current_group = []

      prev_linenr = linenrs[i]

    # Still have groups left at the end
    if len(current_group) > 0:
      grouped.append(current_group)
      current_group = []

    return grouped

  # Print lines of code of components that occur in the matches
  def getCodeLines(componentfilter, offset, bothDirections = True):
    for target in componentfilter:
      Lines.getCodeLinesStartingFromTarget(target, offset)
      print("_______________\n")

      # If target is None, all matches are shown. This should not be displayed twice
      if bothDirections and target is not None:
        Lines.getCodeLinesEndingAtTarget(target, offset)
        print("_______________\n")
        print("_______________\n")


class Tools(object):
  # Convert notation (HUSSACT: x.x.x.y -> y)
  def convertNot(s):
      regex_p1 = re.compile('\.[a-z].*\.[A-Z].*$|^[a-z]')
```

```
      regex_p2 = re.compile('\.[A-Z]')
      x1 = regex_p1.search(s)

      if x1 != None:
        s = s[x1.span()[0]+1:]
        x2 = regex_p2.search(s)
        if x2 != None:
          s = s[x2.span()[0]+1:]

      if "." in s:
        return s.split(".")[0]

      return s

  # Search the dependencies using criteria
  def searchDependencies(component, relationship):
    result = []
    for dep in DEPENDENCIES:
      if (component in dep[1] and dep[2] == relationship):
         result.append(dep)

    return result

  # Find a list of context-declared broadcast receivers
  def findContextDeclaredBroadcastReceivers():
    result = Tools.searchDependencies("xLibraries.android.content.BroadcastReceiver", "
        Inheritance")

    if len(result) == 0:
      print("No BroadcastReceivers found")
      return

    for r in result:
      print(f"{Tools.convertNot(r[0])} \t\t <-- {r[0]}")
      result += Tools.searchDependencies(r[0], "Inheritance")

  # Find list of third party dependencies
  def findingThirdPartyDependencies(namespaces):
    namespaces += ["android.support", "butterknife", "xLibraries.android", "xLibraries.com.
        bumptech",
                   "xLibraries.com.google", "xLibraries.com.squareup", "xLibraries.java", "
                       xLibraries.org.apache",
                   "xLibraries.org.json", "xLibraries.org.jsoup", "xLibraries.org.junit", "
                       xLibraries.org.powermock",
                   "xLibraries.rx", "xLibraries.timber"]
    thirdParties = set()
    for dep in DEPENDENCIES:
      dep_0 = False
      dep_1 = False
      for namespace in namespaces:
        if namespace in dep[0]:
          dep_0 = True
        if namespace in dep[1]:
          dep_1 = True

      if dep_0 == False:
        thirdParties.add(dep[0])
      if dep_1 == False:
        thirdParties.add(dep[1])

    thirdParties = list(thirdParties)
    thirdParties.sort()
    for thirdParty in thirdParties:
      print(thirdParty)


class MatchFilters(Enum):
  ## Match filters ##
  INTENT_URI = "xLibraries.android.content.Intent"
  PENDING_INTENT_URI = "xLibraries.android.app.PendingIntent"

  # Content resolver does not work if context.getContentResolver() is used (and context is too
      broad)
  CONTENT_RESOLVER_URI = "Loader"

  # Detecting binders
  BINDER_URI = "xLibraries.android.os.Binder"

  # Used for finding all references from/to a component
  NONE_URI = ""
```

```
  # Detecting context-declared broadcast receivers
  LOCAL_RECEIVERS = "BroadcastReceiver"

  CUSTOM_COMPONENT = "Any component"

filepath_repository = "/home/yorick/Repositories/Omni-Notes"
filepath_dependencies = "/home/yorick/Repositories/OZP/dependencies_OmniNotes.csv"
namespace = "it.feio.android.omninotes"
matchFilter = MatchFilters.INTENT_URI

# Stupid exceptions when class files contain two separate classes :<
JAVACLASSES["ImageAndTextViewHolder"] = "/home/yorick/Repositories/Omni-Notes/omniNotes/src/
    main/java/it/feio/android/omninotes/models/adapters/ImageAndTextAdapter.java"
JAVACLASSES["NoteDrawerAdapterViewHolder"] = "/home/yorick/Repositories/Omni-Notes/omniNotes/
    src/main/java/it/feio/android/omninotes/models/adapters/NavDrawerAdapter.java"
JAVACLASSES["NoteDrawerCategoryAdapterViewHolder"] = "/home/yorick/Repositories/Omni-Notes/
    omniNotes/src/main/java/it/feio/android/omninotes/models/adapters/NavDrawerCategoryAdapter
    .java"

# Run setup; find java files and matches
Setup(filepath_repository, filepath_dependencies, namespace, matchFilter.value)


## Component filters ##
componentFilter = [None] # Show all matches
componentFilter = ["CategoriesUpdatedEvent","DynamicNavigationReadyEvent","
    NavigationUpdatedEvent",
  "NavigationUpdatedNavDrawerClosedEvent", "NotesLoadedEvent",
  "NotesMergeEvent", "NotesUpdatedEvent", "NotificationRemovedEvent", "PasswordRemovedEvent",
  "PushbulletReplyEvent", "SwitchFragmentEvent"]

Lines.getCodeLines(componentFilter, 1, True)

# Tools.findContextDeclaredBroadcastReceivers()
# Tools.findingThirdPartyDependencies([namespace])
```

Listing B.2: Script used for parsing HUSACCT dependencies. This script is also available at https://github.com/yorickvanzweeden/bachelor-thesis