FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

SMART: Static and Dynamic Analysis to Reverse Engineer Android Applications

Francisco Miguel Gouveia Serrão

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado em Engenharia de Software

Supervisor: Nuno Honório Rodrigues Flores

Co-Supervisor: Ana Cristina Ramada Paiva

September 7, 2021

SMART: Static and Dynamic Analysis to Reverse Engineer Android Applications

Francisco Miguel Gouveia Serrão

Mestrado em Engenharia de Software

September 7, 2021

Abstract

Mobile applications are constantly evolving and becoming a major need for most businesses' success. Moreover, android is one of the existing mobile operating systems, playing a big role in the world of mobile applications due to its popularity, mainly for being open-source.

However, due to nowadays existing exponential growth in technologies, software maintenance is one of the biggest appearances when it comes to companies' costs and effort from new developer joining the teams. Artifacts need to be constantly updated and generated, usually manually. Thus, automating this process could be beneficial for the research area.

The present dissertation provides an overview of the main android components and intents objects, related work regarding reverse engineering of applications in order to exploit the main gaps that may exist regarding artifact generation, as well as the main advantages of the different reverse engineering techniques and methodologies used in these related works.

Moreover, it is presented a discussion of the problems addressed in previously done related work, as well as, open issues in the research field. It is then describe in detail a tool proposed as a solution to help reduce the constant need for companies to support huge costs to generate and update artifacts.

The dissertation's development is driven by the mentioned problems regarding companies' costs and developers' effort. The goals were to firstly analyse developed research work identifying the main gaps in the field, as well as, methods used to reverse engineer applications in these related works. From the identified gaps and methods, the objective was to design and propose a methodology which could extract information from the apps to further build documentation that can help developers reduce their effort to understand applications.

The proposed tool is able to reverse engineer applications in a first instance through static analysis of the application's java files and extract key information regarding intents and class dependencies. Moreover, source code is transformed in a way which allows understanding activities and fragments life-cycles in real time through a desktop app.

From here, the solution was validated through a case study involving apps from GitHub. Results were analysed in order to understand limitations which are provided and discussed right after.

Finally, conclusions are presented, future work is discussed based on found limitations and contributions are described.

Keywords: Android studio, reverse-engineering, static analysis, dynamic analysis, mobile, applications, source code, program understanding, program comprehension

Resumo

As aplicações móveis estão em constante evolução e a tornar-se numa importante necessidade para o sucesso da maior parte das empresas. Além disso, o android é um dos sistemas operativos móveis existentes, desempenhando um grande papel no mundo das aplicações móveis devido à sua popularidade, principalmente por ser de código aberto.

Porém, devido ao atual crescimento exponencial das tecnologias, a manutenção de software é uma das maiores preocupações quando se trata de custos das empresas e do esforço de novos programadores que ingressam nas equipas de desenvolvimento. Os artefatos precisam de ser constantemente atualizados e criados, geralmente manualmente. Assim, automatizar esse processo pode ser benéfico para a área de pesquisa.

A presente dissertação oferece uma visão geral dos principais componentes bem como de objetos de comunicação "Intent" do android, trabalhos relacionados à engenharia reversa de aplicações com o objetivo de explorar as principais lacunas que possam existir na geração de artefatos, bem como as principais vantagens das diferentes técnicas e metodologias de engenharia reversa. usado nesses trabalhos relacionados.

Além disso, é apresentada uma discussão dos problemas abordados em trabalhos relacionados anteriormente realizados, bem como, questões em aberto no campo da pesquisa. Em seguida, descreve-se em detalhe uma ferramenta que foi proposta como solução, de forma a facilitar e reduzir a necessidade constante de as empresas arcarem com os elevados custos de geração e atualização de artefatos.

O desenvolvimento da dissertação é impulsionado pelos problemas mencionados em relação aos custos das empresas e ao esforço dos programadores. Pretendeu-se, em primeiro lugar, analisar os trabalhos de investigação desenvolvidos identificando as principais lacunas existentes na área, bem como os métodos de aplicação da engenharia reversa nestes trabalhos relacionados. A partir das lacunas e métodos identificados, o objetivo foi projetar e propor uma metodologia que pudesse extrair informações das aplicações para criar documentação adicional que possa ajudar os programadores a reduzir seu esforço para entender as aplicações.

A ferramenta proposta é capaz de realizar a engenharia reversa de aplicações em uma primeira instância por meio da análise estática dos ficheiros java da aplicação e extrair informações importantes sobre intents e dependências de classes. Além disso, o código-fonte é transformado de forma a permitir a compreensão dos ciclos de vida das atividades e fragmentos em tempo real por meio de uma aplicação desktop.

A partir daqui, a solução foi validada por meio de um estudo de caso envolvendo aplicações android do GitHub. Os resultados foram analisados a fim de compreender as limitações que a ferramenta contém e discuti-las em seguida propondo a sua solução em trabalho futuro.

Keywords: Android studio, reverse-engineering, static analysis, dynamic analysis, mobile, applications, source code, program understanding, program comprehension

Acknowledgements

Developing the present document and completing my. academic degree was only possible due to numerous factors and people around me.

First of all I would like to thank my supervisor Professor Nuno Flores and co-supervisor Professor Ana Paiva, whose availability, collaboration and guidance were perpetual and unfaltering. It was a great experience to have the opportunity to develop this project with you.

Secondly, I would also like to thank Professor João Bispo for his continuous collaboration. Without him, it wouldn't had been possible to complete an indispensable component of this dissertation.

Furthermore, I'm beyond grateful to my family. Their love and constant support was crucial throughout my academic path. To all of you, I don't think I can thank you enough, from the bottom of my heart.

Finally, I would like to thank all my friends and colleagues for being part of my personal, academic and professional journey, for their friendship, conversations, contagious joy, trust and advises. I believe relations created will last for life.

Francisco

"Until I began to learn to draw,
I was never much interested in looking at art."

Richard P. Feynman

Contents

List of Figures

List of Tables

Abbreviations

APK	Android Package Kit
AST	Abstract Syntax Tree
CG	Call Graph
DG	Dependency Graph
ET	Execution Tracing
LC	Life Cycle
MIME	Multipurpose Internet Mail Extensions
OCR	Optical Character Recognition
OS	Operating System
RE	Reverse Engineering
UI	User Interface
UML	Unified Modelling Language
URI	Uniform Resource Identifier
US	User Story
TAE	Transformed Application Exploration
TL	Transformation Language
XML	Extensible Markup Language

Chapter 1

Introduction

Reverse engineering of mobile applications is getting more and more traction due to its various applications. With this in mind, it was the research area chosen to be addressed in the context of the present dissertation.

Android Studio is the official framework used to write code for these applications. Moreover, android is the most commonly used mobile operating system due to being open source and having a huge and active community. Specifically, it was mainly developed by Google and has been dominating mobile application's market since 2011 [1], composing at the moment, a total of 72.5% of the total market shares according to [12].

With android's dominance and high demand regarding its number of active users, emerges the need from developers to understand the great variety of elements android has to offer (e.g. intents, broadcast receivers, services, etc).

Reverse engineering is the process of performing a detailed analysis to a product to empower the acquisition of vital information regarding that specific product. In particular, applying the reverse engineering process to a mobile app allows information extraction to, for example, build artifacts or port an application to other platforms. For a detailed explanation regarding RE and its various analysis techniques, please refer to chapter 2.

1.1 Extracting knowledge from Android Apps

Mobile applications developed for android require effort from new comers due to android's steep learning curve associated with the wide variety of components that it has to offer. Also, complexity for the mentioned group of individuals is increased due to the fact that these components may contain life-cycles (e.g., activities/fragments).

Moreover, in a world where apps are constantly changing and evolving, there has to be a commitment to keep artifacts updated so that new developers can be easily integrated in a companies' team. Although documenting software may be time-consuming, this is crucial in a world where many software engineers shift companies on average every 3 to 4 years [9] and artifact maintenance / generation composes a considerable part of companies' costs.

Furthermore, software maintenance is a difficult task for itself and it is usual for companies to give less importance to the documentation. However, lack of documentation composes an even greater challenge when developers are maintaining a software, since without any documentation or artifacts, there is no clear representation or description detailing the software's architecture and workflow.

The reverse engineering process can help tackling this problem by extracting knowledge, i.e, structure and behaviour, from a software and transforming it into information that can be used by new coming developers to better understand the software. However, very few tools exist that may help new developers to understand the android elements.

1.2 Objectives

Motivated by the identified gaps from the initial analysis, which was performed to the research area of reverse engineering of android mobile applications, and taking into consideration which method of extraction best suited this dissertation's context, the objective was to propose a possible solution which could tackle the issues contained in the gap.

The SMART (Static dynaMic Analysis andRoid applicaTions) solution's objective involved proposing a tool which enabled knowledge extraction from android applications. The knowledge was composed of information related to the main android components, such fragments and activities, as well as, intent objects. Additionally, another objective was to find and use a tool which raised the possibility to analyse and transform source code.

Expected results from the development of the present dissertation firstly involved the successful analysis of reverse engineering related work, identifying and describing their main limitations, as well as, gaps.

From the research work performed to the state-of-the-art, it was expected to propose a solution that could fit the existing gaps in the research field, also diminishing the identified open issues.

The proposed solution, SMART, was expected to successfully reverse engineer android applications, regardless of their category, extracting artifacts, that could contribute to the mentioned research area and tackle identified issues.

Artifacts that were extracted will be discussed through the following chapters and these included intents objects and intent filters, classes' dependencies among them, as well as, a real-time visualisation of active activities and fragments while handling a mobile application.

SMART was to be validated through a case study which involved experiments with apps from different categories to verify its versatility. Results from the experiments, shown in the appendix, were expected to be used to answer the pre-defined validation questions.

It was expected that the tool was valid, meaning that it was indeed able to generate the artifacts. Note that questions were constructed so that these were expected to have an affirmative or negative answer, meaning it was able to perform the extraction when answered with a "yes".

SMART is a tool that relies on a mixed analysis reverse engineering (RE) technique to reverse engineer mobile applications and extract defined artifacts. Since it is a mixed analysis technique it is divided into two components them being static and dynamic.

Regarding the static analysis component, it analyses the code in search of key information regarding components that compose the extracted artifacts.

After completion of the analysis, it transforms the source code in a way so that the mobile application communicates with a desktop application.

SMART's main contributions are the fact that it empowers the possibility for new developers to analyse in real-time which activity/fragment is being invoked and at which state it is, as well as, the possibility to view intents' information of the application under analysis.

SMART can be also used in an educational context in order to help learn how the android life-cycle methods work. However, SMART can not be considered a serious game because it does not have all the required features such as the ones presented in [45] and [27].

1.3 Document structure

The complete document structure which is presented in the dissertation is organised as follows: introduction's chapter firstly provides a context regarding the research area which was mitigated, the main goals of the dissertation's development, the expected results and an high level view of the tool that will be proposed.

Chapter 2 presents not only some key concepts related to android and reverse engineering, but also a deep analysis regarding previous published related work.

Chapter 3 presents the existing open issues in the field, a discussion related to all published work analysed in the State-of-The-Art and a solution that could tackle the open issues.

Chapter4 refers to a detailed description of the tool which was developed as a solution to existing gaps in the research field along with its architecture.

In chapter 5 it is presented the case study which was performed with applications from GitHub [8], along with its results in order to validate the previously mentioned solution and limitations that were found in the solution.

Chapter 6 contains the conclusions, future work, as well as, the main contributions from the implemented solution to the research field.

Chapter 2

Reverse Engineering of Android Applications

The present chapter is intended to 1) provide an overview of android and its components in section 2.1; 2) explain the reverse engineering techniques in section 2.2; 3) present the different methods used for the process of reverse engineering of mobile applications with a more in dept definition and explanation, in sections 2.3, 2.4 and 2.5.

2.1 Android

Android is a Linux-based operating system, with its structure for the different layers written in C. Furthermore, it was mainly developed by Google [1] and it is the most used mobile operating system with a world dominance regarding the number of its users [12].

Android applications can be written using the Android SDK or Kotlin language (created to replace Java). The SDK is where a debugger, resources and packages that are used by the applications are stored.

Since the android operating system is open-source, there are a vast of applications developed by third party companies that have their APKs available in the Google Store for users to download them. This can lead to security leaks, for example, due to permissions that these applications request in the background [1].

To reduce security problems, ensuring software quality is crucial. In order to do so, analysis of the various elements of android can provide developers with information related to the application's architecture and workflow to detect failures or weaknesses. The main examples of these elements are fragments and activities, responsible for displaying UIs (user interfaces) to the user in the android system, as well as, intent objects that enable messaging mechanisms between components.

2.1.1 Intents

Intents objects and all their features are made available using the android intent resolver, which is best suited for sending data to other applications as part of a well-defined task flow [15]. Moreover, intents in android are seen as messaging objects that can be used to request actions from other components inside the app or even other applications. The android documentation describes intents has having three main use cases, namely:

- starting activities by passing the intent to the method ***startActivity()***.
- starting services using a **JobScheduler**. [13]
- delivering broadcasts using the ***sendBroadcast()*** or ***sendOrderedBroadcast()*** methods. [4]

These messaging objects have a structure that contains information, such as the name of the component to be started, the action that should be performed on that specific component, the URI object that references the data to be acted upon and some other information regarding the category of the component, a pair of key-value that carry information needed to complete the request done and a flag, that can be seen as "metadata" about that intent "instructing when to launch an activity and how to treat it after it's launched" [15].

Additionally, intents may either be explicit when demanding to launch a specific application component by providing the component name, or implicit when there isn't a specification of the app that is to be launched. In the second case, the system performs an analysis to determine which of the installed apps support that intent, through comparing the information of the intent with the intent filter and further displaying them to the user. An example of this process for selecting the correct activity to receive the implicit intent is shown in Figure 2.1.

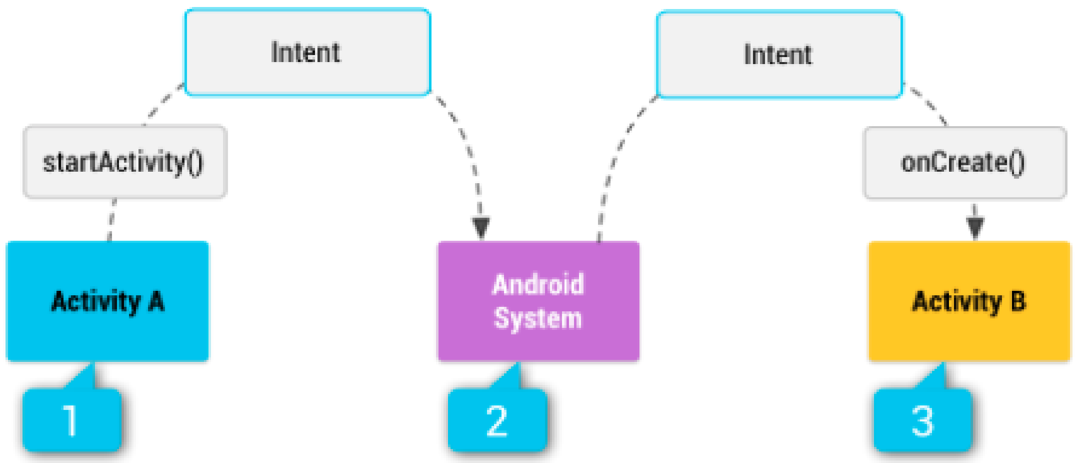


Figure 2.1: Implicit intent being delivered to the correct activity [15]

Moreover, intent filters are expressions contained in the application's manifest file. Also, intent filters define the type of intents that that applications' components can accept by using the **intent-filter** element [15]. The definition is based on the intent's:

- **data** - specifies the accepted data through attributes of the data URI and MIME type.
- **category** - specifies the name of the category of intents that is accepted.
- **action** - specifies the name of the intent actions that are accepted by the intent.

Furthermore, **intent-filters** can contain more than one type of actions, data or category. However, for each intent there should be a different intent-filter.

2.1.2 Activities

Activities are considered to be classes designed specifically to help create the user paths which will take part of the application's workflow of UIs [16]. With this in mind, it is possible that for different applications there might be distinct user interaction's beginnings with the use of activities.

Usually, each activity implements a screen and these are totally decoupled from each other. Thus, each of them contains their own independent life-cycle.

An activity can iterate throughout numerous states in which each state has a callback provided by android and some of these callbacks are mandatory to implement. These callback methods will be referred as life-cycle methods throughout the document.

Every activity has to have the ***onCreate()*** callback which is fired once an activity is created by the system and this method contains information related to the startup logic of the activity.

Once the process of starting the logic is completed, ***onStart()*** method is called. This callback is the method that prepares the activity before being displayed and prepared for interaction with the user [16].

Subsequently, the callback ***onResume()*** is invoked once the activity enters the Resume state. Moreover, as long as there isn't an action that makes another activity to come to foreground, this is the state that stays active and where interaction with the user takes place.

Eventually, the activity enters the pause state and calls the ***onPause()*** method stopping functionalities that are not essential to be running while the activity is on background. Additionally, this is the state where it will defined whether the activity will be used by the user, returning to ***Resumed*** state and calling its corresponding callback method, or if it should be stopped, either due to lack of memory or if the activity isn't visible anymore.

As a consequence of the second option, ***onStop()*** method is invoked and all heavy functionalities and some resources related to that activity are stopped, even though the state and activity object are kept in memory.

From stopped state, the activity is either restarted using the callback ***onRestart()***, or destroyed using ***onDestroy()*** [16]. In such circumstances, all remaining resources are killed and so is the activity.

Figure 2.2 illustrates a more clear view of the transition between these states.

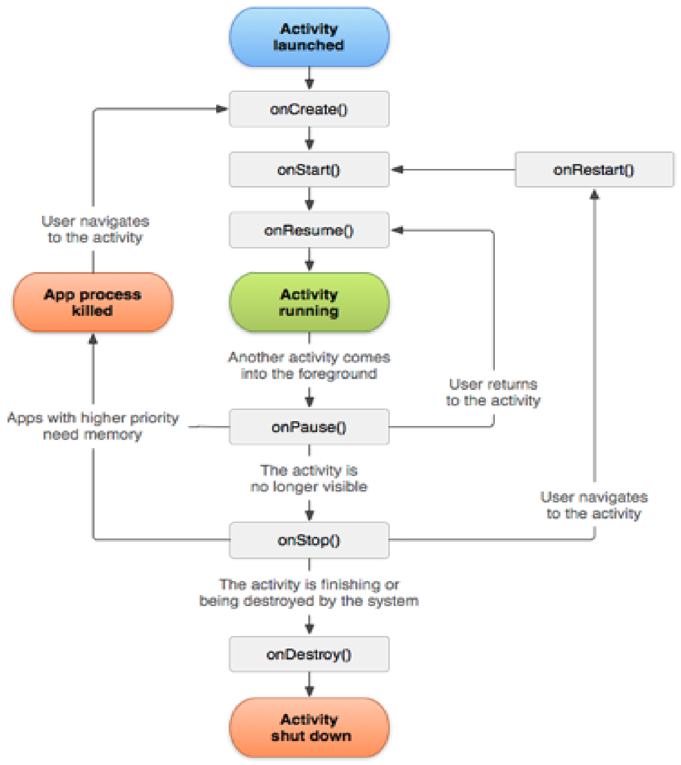

Activity launched
onCreate()
onStart()
onRestart()
onResume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.2: Activities' life-cycle [16]

2.1.3 Fragments

Fragments are perceived as reusable portions of applications' user interface. Although fragments are independent from each other, meaning that they have their own life-cycle (just as activities), events and layout, fragments are necessary to be contained inside another fragment or an activity (using **FragmentActivity class**).

Furthermore, the class **FragmentManager** is in charge of performing actions on fragments, as well as, managing the fragment's back stack. These actions should be done inside a transaction scope, committing the changes to utilized fragment with the **FragmentTransaction** class, in order to avoid data losses and fragment's states. [7]

Android provides a life-cycle object that can be accessed by the ***getLifecycle()*** method. In addition, the possible states throughout a fragment's life cycle are designed as follows:

1. *Initialized*
2. *Created*
3. *Started*
4. *Resumed*
5. *Paused*

6. *Destroyed*

Note that fragments cannot be in a certain state without before iterating through previous states. For example, a fragment cannot be paused if it hasn't been previously created, started and resumed.

Also, fragments have a life-cycle independent from their view's life-cycle. Fragment's life-cycle may be inspected by an observer or by using callback methods, which are similar to callbacks from activities with the addition of two extra callbacks, namely ***onAttach()*** and ***onDettach()***. These are used upon fragments being added or removed from a **FragmentManager** respectively. [7]

Regarding the relation between the view's life-cycle and the fragment's life-cycle, to start and to be able to interact with the fragment, the process starts with the fragment reaching the created state. At this point, the fragment has already became a part of the **FragmentManager** and the creation of the fragment requests the ***onCreate()*** callback. From this point, the view is created only if the fragment provides a non-null view and this view can be returned with ***getView()*** method.

Also, in this stage the initial state of the view is defined and the observation of **LiveData** instances that have callbacks that update the fragment's view are started.

After the creation of the view, both the fragment and its view are started. At this point, if the fragment's view isn't null, the fragment is passed to ***Started*** state right after the fragment is also started. Once the fragment is prepared for interaction with the user, all events and elements have been finished and the ***onResume()*** method is called. [6]

On the other hand, when the user is leaving the interaction with the fragment, firstly the fragment's view life-cycle is sent to observers followed by the fragment's life-cycle. As for the first state, the fragment is still visible to the user. Moreover, both the view and its fragment "owner" are sent back to ***Started*** state and emit the ***ON_PAUSE*** event.

Afterwards, the fragment becomes invisible, the fragment and the view are moved to ***Created*** state and send the ***ON_STOP*** event to observers.

The following step is to exit all animations and transitions so that the fragment's view is totally removed from the window and the view's life-cycle is pushed to ***Destroyed*** state, handling the ***ON_DESTROY*** event to its observers and calling the ***onDestroyView()*** callback method.

Finally the fragment reaches the ***Destroyed*** state, also emits the ***ON_DESTROY*** event to observers of the fragment and ***onDestroy()*** method is called ending the fragment's life-cycle. The following figure 2.3, provides a clear view of this process both for fragment's life-cycle and their views life-cycle. [6]

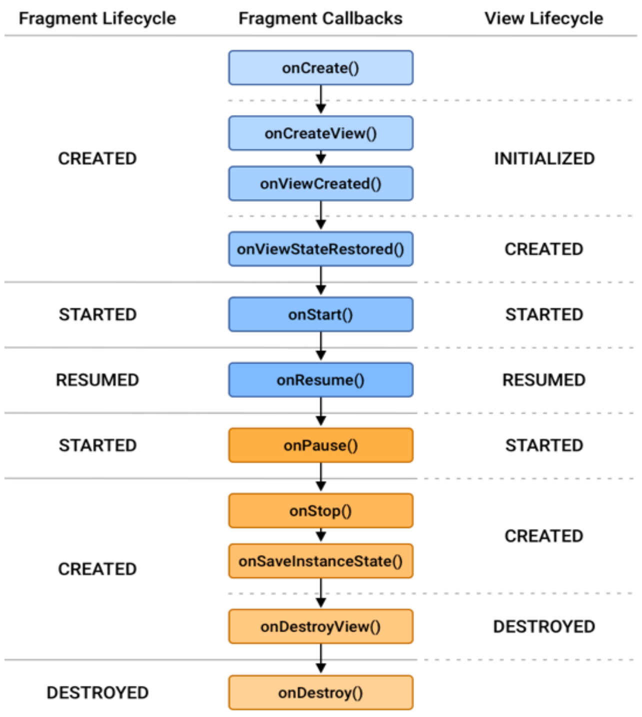


Figure 2.3: Fragment's life-cycle and fragment view's life-cycle [6]

2.2 Reverse Engineering Techniques

Reverse engineering, also known as RE, is the process of extracting knowledge or useful information from any product regardless of its business category [53]. Particularly, in the context of software engineering, the RE process can be anything from extracting knowledge through a simple source code line-by-line analysis, to creating UML class diagrams from source code through analysis, or even searching for faults in the applications by triggering events throughout the application's flow.

Furthermore, knowledge extraction can be done through static analysis ([14]), dynamic analysis ([49]) or using a mixed analysis ([48]) technique that takes advantage of a combination of both techniques.

Static analysis doesn't require the execution of code. It instead analyses code itself identifying methods, events or any other element that can be advantageous for the creation of artifacts depending on the problem being tackled and project's context.

Opposed to static analysis, a **dynamic approach** extracts the fundamental knowledge to create artifacts, by executing the application and its different flows/events, capturing, in most of the developed related work, event triggers or component dependencies.

Finally, a **mixed technique** usually relies on static analysis, on a first stage, in order to find code failures regarding events or workflows of the application, to then complete these paths and further analyse the application using a dynamic tool (e.g., AMOGA [18]).

Each of these techniques present advantages and disadvantages depending on the project's context, as well as, what information is available for analysis. In particular, static analysis of mobile applications may become fairly challenging since these applications are event-driven [31], so dynamic analysis could be considered the best solution to reverse engineer applications.

However, since most of the times applications have failures and may lack correct paths throughout its execution flow, for example missing states in android activities, dynamically analysing applications can lead to incorrect or incomplete results / artifacts.

Thus, usually the best suited approach that developers rely on is the combination of a static analysis to complete missing paths/code using, for example, invoked activities' life-cycle method, followed by a dynamic analysis using tools to extract applications' workflow along with other application's properties.

Relying on the defined research area, reverse engineering of android mobile apps, the first performed step was to find related work already developed in the mentioned research area. From the investigation performed, an analysis of the papers was done in order to identify possible gaps/improvements that could be explored and developed as possible solution tools as a contribution for the present dissertation.

In the following sections papers are presented accorsing to their RE technique along with a description of the phases taken in a specific paper and results / conclusions obtained with the goal of understanding the possible solution tools/methodologies that could be implemented in order to improve the research area's State-of-The-Art.

2.3 Static Analysis Techniques

[30] defines static analysis as "the analysis of computer software which is performed without the actual execution of the programs built from that software". Since there isn't an actual execution of the code, the analysis is considerably limited, not being able to extract the response of, for example, event triggers under certain conditions.

Although static analysis techniques usually involves using simpler methods for knowledge extraction, it can be greatly convenient for smaller applications or analysis. Some examples of these methods are the usage of Abstract Syntax Trees and Call Graphs discussed in the following sections.

2.3.1 Abstract Syntax Trees

Abstract Syntax Trees (ASTs), are identified as tree-like hierarchical structures, which are generated from the source code and composed of nodes where each node represents an if-then condition that should declare the order of executable statements related to the source code.

ASTs are very useful since they allow the abstraction of some details incorporated in the source code such as parenthesis and punctuation. Moreover, these structures can further be used to generate specific artifacts (e.g., Kadabra Java Weaver, Clava, etc [11], [21]).

However, ASTs require the developer to specify the full collection of conditions that empower the creation of such tree hierarchical structure, with a properly defined granularity. This definition may require some effort. A real world example is when a developer needs to reverse engineer multiple mobile applications or even complex and large apps.

As done in [25], when depending on the usage of ASTs to RE apps, for the most part, there are at least three main stages regularly taken into consideration, namely:

- **extraction** - at this moment the source code is extracted to be subsequently used by a tool that empowers the generation of the AST;
- **abstraction** - this stage consists of a source code analysis, followed by the construction of the AST structure by the previously mentioned tool;
- **presentation** - at this point, the structure is analysed in order to generate the artifacts that are desired for the work being developed.

Another example that was built with the usage of Abstract Syntax Trees is proposed in [37]. The authors present a model driven reverse engineering approach for mobile applications with the goal of transforming the native source code from a certain language/platform to native source code of a different language/platform, relying on the usage of abstract syntax trees and graphs models (discussed in 2.3.2).

Regarding [37] in particular, the process involved the main stages of ASTs mentioned in 2.3.1, with the addition of the discovery phase which consists of the creation of these call graphs from ASTs. After the creation process of the call graphs is completed, these are converted to other graphs that "mirror" them to the the target platform native language using a meta-model. The new call graphs are then converted to the target platform source code.

The authors in [37] concluded that, with the increasing need of extracting native source code from mobile applications to convert it to different platforms, it is required a good degree of competence from the developer, in both source and target platforms in order to perform this methodology.

The implementation of this approach can also be useful for generating artifacts related to android components, to reduce time and effort from developers when companies need native applications for multiple platforms and lack artifacts or knowledge about both platforms.

For a better understanding of the process involved, a representation can be seen in Figure 2.4.

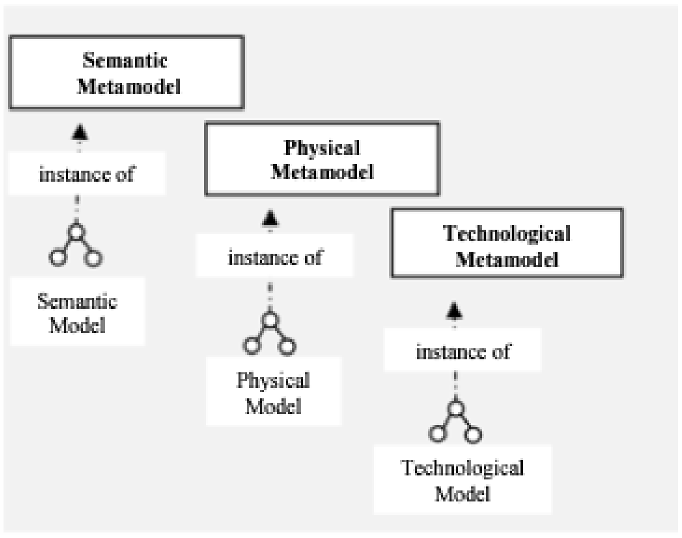


Figure 2.4: Example representation of the RE process using ASTs [37]

Another use of ASTs is presented in [35] where the authors proposed the implementation of a reverse engineering static analysis approach in order to map and transform mobile applications' UIs between platforms, taking advantage of the Appium tool [3].

Appium is another tool apart from the ones mentioned in section 2.3.3 (e.g., Apktool), that can be used to de-compile the source code with the goal of extracting relations of UIs and creating the AST hierarchical structure.

After the creation of the AST, the authors created modules related to functions and controllers, saving these modules between platforms (iOS and Android) in a database. Moreover, through the combination of this data with previously generated modules from a storyboard file (that contains relations from iOS controllers of the source application), [35] obtained the production of Android attributes to after generate the UI elements.

This project comes to show that the use of ASTs can create accurate abstractions of controllers even though the encoding of "Android GUI or Swift are difficult", because it requires the same compiler and process for both platforms.

2.3.2 Call and Dependency Graphs

Using the source code, there is the possibility to create specific types of graphs, namely Call Graphs and Dependency Graphs.

Call graphs are the most common types of graphs and empower knowledge extraction for creating artifacts through static source code analysis. These graphs contain relevant information about all method calls within the method bodies that are being analysed. Nevertheless, native code calls are abstracted from those graphs.

Although having a similar structure to abstract syntax trees, each of the call graphs nodes represent methods with all its variables and information.

The authors of the dissertation that is presented in [42] proposed and developed the implementation of a component that is capable of creating method call graphs and subsequently artifacts to help developers better understand applications, relying on a number of different stages, namely:

- **extract** classes that have methods which are related to each other, mentioned as the "being-used together classes";
- **separate** the classes through the creation of a method call graph containing all methods that exist in the application;
- **remove** method chains "that do no use more than one class from the method call graph" [42];
- **extract** the domain model classes and their multiplicity relations (the tool proposed preserves only classes that have at least 1 field and getter and setter methods);
- **analyse** these fields to reverse engineer multiplicity relations;
- **rank** the domain model classes reusing the previously mentioned call graph output in 2.3.2, adding the spectrum of domain classes used by that method;

The other pertinent type of graphs for this dissertation are data dependency graphs which are used with the goal of extracting data dependencies (e.g., classes), so that the tool analysing the source code is able to extract knowledge to create artifacts or even refactor code.

In particular, the paper in [34], is firstly distinguished from other related work due to the fact that it is the only solution which is implemented as an Android Studio plugin and relies on data dependency graphs, **OSAIFU**. An illustration of the complete methodology is presented in Figure 2.5 followed by a brief clarification and advantages of the approach applied.

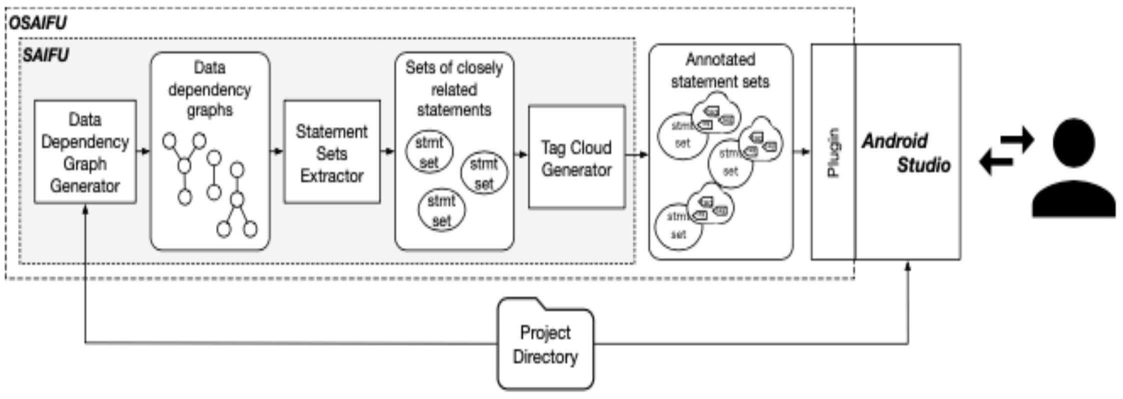


Figure 2.5: OSAIFU [34]

The main goal of this paper was to support program understanding and reverse engineering of android mobile applications. The source code was provided as input to the plugin and the graphs were created.

Furthermore, the "Statement Sets Extractor" creates groups of statements that are related. With these groups OSAIFU extracts features of analysed mobile applications.

A key point is the fact that the tool uses the Java Parser library to be able to generate the dependency graphs in the process.

2.3.3 APK File Analysis

From the vast variety of methods to perform static analysis, one of the most popular techniques in the context of android mobile applications, is to analyse android APK files. These APK files (Android Application Kit) are the files' format that android uses to install and distribute apps. Furthermore, these contain the elements that a specific app demands to install correctly.

A simple example of this technique is [53] where the authors present a paper in which, taking advantage of android manifests, extract properties related to key listening functions and associated to tracing program intrusions such as properties analogous to applications changing devices configurations, installing shortcuts, recording audio, "performing NFC I/O operations" and changing the Wi-Fi state of the device.

This process was done by de-compiling the APK file using Apktools and the android SDK, creating Dalvik code which was then disassembled to Smali byte-code ("similar to Java byte-code").

Apktool is a tool used to extract source code from the application's APK file through reverse engineering. The usage of such tools is deeply common in the context of RE of mobile apps since it allows the analysis and transformation of the source code of the application in an automated form.

From here it generates Smali byte-code and it also possible to rebuild the code to re-generate the APK file.

The authors concluded that over the years, reverse engineering took a shift from only being applied to hardware, to also being applied to software, reporting to be very useful to software quality improvement, as well as a great improvement in security level of android applications [53].

Tools such as Apktools [2], will be mentioned throughout the present dissertation as transformation languages, since these, as the name suggests, allows transformation of source code according to developers' needs.

Since these files extracted from the APK, in particular manifests, contain critical information related to permissions, there are several exploits that can be used for malicious behaviours.

The authors in [50] conducted a study relying on static analysis reverse engineering tools to analyse permissions. For the study, the authors used the APK files and de-compiling tools to complete the source code analysis.

Moreover, a permission analysis was performed based on manual mechanisms after converting the source code to detect and remove malicious activities in mobile applications through unnecessary permissions such as GPS and Location access.

The authors were able to detect several applications that had unnecessary permissions leading to security leaks and in order to improve the work developed, they proposed the creation of an automated mechanism for checking failures [50].

An illustration of the entire reverse engineering process and analysis developed in this paper is visible in Figure 2.6.

The APK analysis approach is best suited for analysing security associated properties, although it could be also applied for improving software quality, as well as artifacts extraction through analysis of other properties of the application, taking developers to need less effort to understand systems.

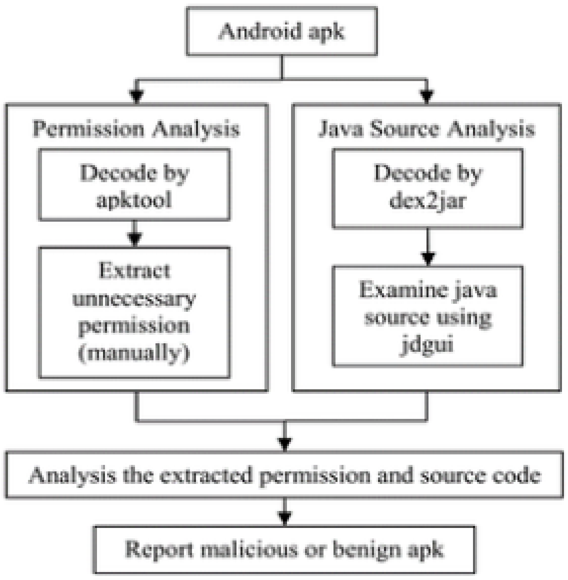

Android apk
Permission Analysis
Decode by apktool
Extract unnecessary permission (manually)
Java Source Analysis
Decode by dex2jar
Examine java source using jdgui
Analysis the extracted permission and source code
Report malicious or benign apk

Figure 2.6: Reverse engineering process [50]

As already specified, Transformation language (TLs) are used to create design structures of the input into a specific output that satisfies some condition. Apart from previously mentioned related work that takes advantage of these TLs such as [50] and Apktool [2] in [53], which is the most popular tool for program understanding/reverse engineering, other tools were analysed in [20], namely: dex2jar [5]; Scoot. These perform the transformation of the executable source into Jasmin and Jimple intermediate languages respectively. Dex2jar then converts Jasmin code into class/jar files.

The content of [20], provides a general overview of all these tools, as well as a comparison between them.

The authors concluded that **Apktool** [2] "performs the most accurate transformation of the Android apps executable since the apps, which are assembled from Smali, most closely pre-serve their original behaviours" [20].

The usage of TLs can be useful when needing to create artifacts since it is possible to specify the output needed to generate these artifacts.

Another example of a paper where the generation of the artifacts is done relying on de-compiling tools, is the work presented in [24]. The authors proposed a tool that relies on these TL tools for artifact generation (UML class diagrams), in order to reverse engineer mobile applications, through static analysis. The main goal was to analyse apps in search for anti-patterns and to create "semantic integration".

The complete process involved three different parts and these were:

1. the **formatting** of the mobile application apk file to Java source code format;

2. the process itself of **reverse engineering the application** to produce the "UML class diagrams", with the analysis of the android apks that were later used to analyse anti-patterns;

3. **anti-pattern search** in order to achieve an increase in software quality by "analysing relations between object-oriented anti-patterns". Also, at this stage of the process, solutions for anti-pattern's semantics are provided by the tool developed, consequently regenerating "the Java code of mobile applications";

The main goal of the work developed in [24] was to improve, as some other already mentioned related work, the software quality of mobile applications, which comes to show that reverse engineering can be seen as an useful tool that companies can rely on to reduce costs and effort from developers. In this case, the usefulness relies on the fact that the tool was proven to be able to detect many anti-patters in the applications utilized in the case study, taking the responsibility of that task from the developer. It also creates UML class diagrams that can be generated for knowledge extraction from new developers.

Still in the context of TL tools, using a static analysis approach, the authors in [19] propose an automation of this process using a toolkit to identify malicious components in applications that could corrupt security for user dealing with those apps.

To execute the reverse engineering process of applications, [19] obtained the APK file of the application and using TLs, they de-compiled the Java byte-code and analysed information contained in the APK file, acquiring key information regarding it, such as source code, manifest files and used resources by the apps, rebuilding the source code after.

The following steps in the process were to analyse permissions and check if there are any vulnerabilities in the APK, through "comparing it to a list of known ones".

Moreover, the authors tested the tool in a real world scenario (for a full case study process inspect [19], sect. 4), and concluded that although a static analysis tool is already helpful, a step further would be to develop a dynamic analysis tool.

This paper is referred to empower a justification for the fact that TLs enrich the process of reverse engineering since it is possible and simple to retrieve key information of mobile applications to then build artifacts.

2.3.4 Optical Character Recognition

Optical Character Recognition is a technique that can take any type of image or printed text and convert it into the needed components. In particular, regarding mobile applications, as in [43] it is possible to take screenshots from an application and using a tool, reverse engineer them to create a structure of the user interface. From here, it is possible to export the structure to obtain the source code or packages used by the application in another platform's language.

[43] depends on a static analysis reverse engineering technique where the first stage proposed was to detect "off-the-shelf OCR" words from the screenshot provided and use heuristics to remove "false-positives" from this exploration.

From the exploration, a computer vision exploration was done in order to extract a possible view hierarchy of the application. These two explorations were merged, creating a screen with some duplicate elements that were removed by the tool.

The final output was exported containing all packages and source code suited for creating the intended UI, which were later complied generating the mirrored application.

An illustration of the main steps is visible in Figure 2.7 and the authors concluded that automation of this process empowers, not only the tansformation of UIs between platforms using a generic tool REMAUI[43], that can take images independently of the OS that the image was taken in, but also the process of reverse engineering when source code is not immediately available for knowledge extraction.

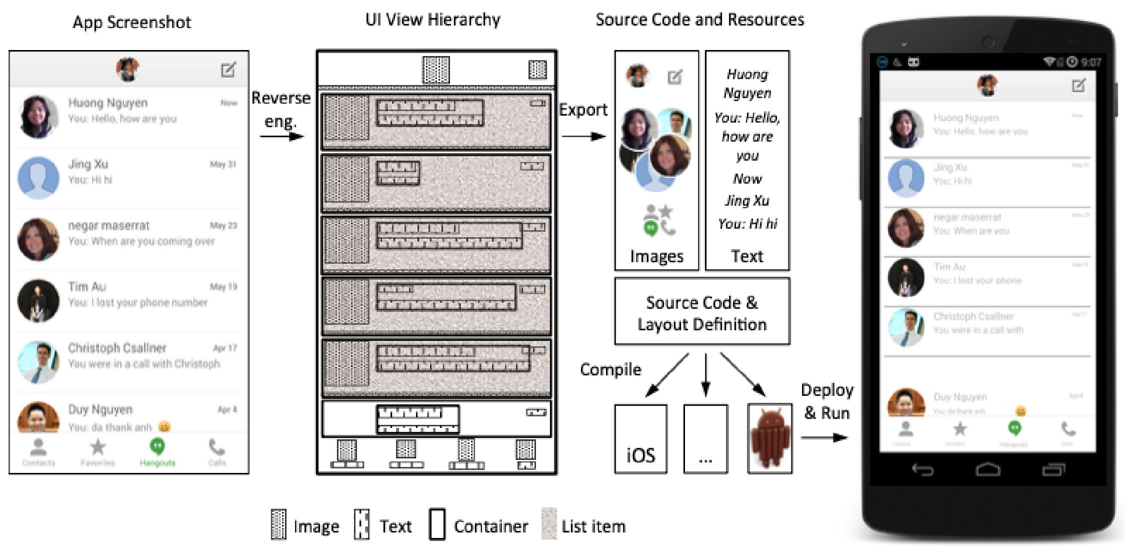


Figure 2.7: Reverse Engineering Tool [43]

The authors concluded that automation of the process empowers, not only the tansformation of UIs between platforms using a generic tool, such as REMAUI[43], that can take images independently of the OS that the image was taken in. Automation also facilitates the process of reverse engineering when source code is not immediately available for knowledge extraction.

2.4 Dynamic Analysis Techniques

As mentioned in 2.2, dynamic analysis takes advantage of the actual execution of the application / program being analysed. This execution allows to capture information that may not be possible to acquire simply by using static analysis, depending on the situation. The information usually refers to events triggers and component dependencies.

However, in order to run applications, as well as, to acquire correct and complete knowledge about them, it is essential to ensure that event execution paths are complete. Furthermore, all inputs needed for the analysis must be provided to the tool which will perform it.

Note that all the analysed dynamic related work relied on a testing and logging techniques although with different goals.

2.4.1 ReIMPaCT

ReIMPaCT [31] is a tool that uses black-box testing and relies on a fully dynamic approach to reverse engineer apps. The authors used this technique to extract hierarchical state machines divided in three different levels, namely "navigation" through activities, "states of activities" and "screens used in this navigation".

The primary advantage that should be taken into consideration is the fact that it does not need to access the source code to reverse engineer applications because it applies Google APIs to "interact with the device" and "to read and extract information from the screen of the device", namely UI Automator and UI Automation.

The tool was submitted to a case study using apps from the Google Store and since there weren't any faults found, the authors intend to inject errors to understand if the tool can help developers to find these faults more easily when being applied to real world scenarios.

In [40] and [26], the authors presented a methodology which used the iMPAcT tool to reverse engineering apps dynamically. It then tested recurring behaviour in these applications. Dynamic explorations were performed to the applications to check whether the UI Patterns were correctly implemented.

The tool was shown to be useful because, although "it was applied on a small sample of mobile applications, it already showed that the iMPAcT tool is able to detect errors in the implementation, such as in the case of a change in orientation"[40].

2.4.2 GUI Reverse Engineering

The authors in [[32], [39]] present a "reverse engineering approach" with the goal of reducing developers' effort to create models according to the user interfaces. The knowledge extraction in the mentioned work is done through a dynamic analysis which executes the UIs to extract its behaviour and architecture.

For illustration purposes, Figure 2.8 represents the complete process developed in [32].

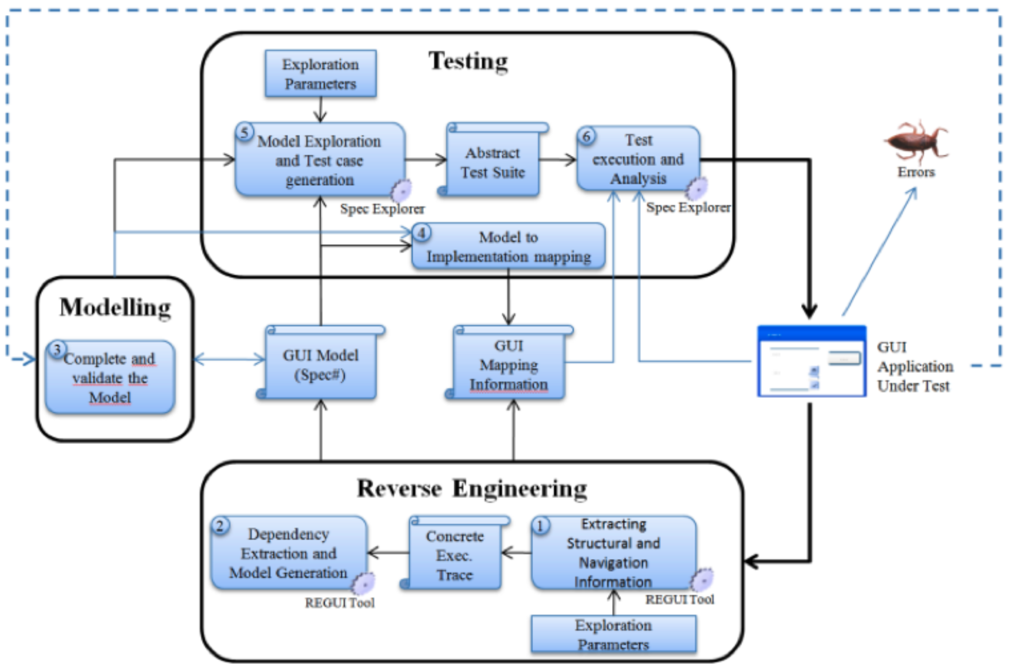

Testing
Exploration Parameters
5 Model Exploration and Test case generation
Abstract Test Suite
6 Test execution and Analysis
Spec Explorer
Spec Explorer
4 Model to Implementation mapping
Errors
Modelling
3 Complete and validate the Model
GUI Model (Spec#)
GUI Mapping Information
GUI Application Under Test
Reverse Engineering
2 Dependency Extraction and Model Generation
Concrete Exec. Trace
1 Extracting Structural and Navigation Information
REGUI Tool
REGUI Tool
Exploration Parameters

Figure 2.8: Reverse Engineering Tool [32]

Through the methodology, the authors were able to reverse engineer applications and automatically generate XML models that can be further tested. The authors concluded that reverse engineering UIs empower testing which can further improve software quality, and reverse engineering apps to automate artifact generation (GUI models in this case) can diminish effort from developers.

Other related work regarding GUI reverse engineering and testing are presented in [44] and [47]. The first work presents a GUI testing tool through reverse engineering similar to [32]. However, in the second work, the tool was applied to web applications and not mobile.

In addition, the paper in [41] also uses a dynamic analysis technique to reverse engineer GUIs. However, in this case the authors relied on a dynamic exploration through the application so that, using machine learning to solve ambiguous conditions, they could "generate state-machine model of the GUI, including guard conditions to remove ambiguity in transitions." [41].

2.5 Mixed Analysis Techniques

Mixed analysis techniques are usually the most complete and involve two main stages relying on both static and dynamic analysis.

The first stage is usually composed of a static analysis which is performed to capture information missing that will be later used as input for a crawler algorithm. The second stage is precisely the dynamic analysis of applications by the crawler algorithm that will capture event triggers or other output specified in the process.

2.5.1 Testing and Event Logging

Testing and event logging is a technique that most of the times relies on a mixed analysis technique just as proposed in [29]. Testing and Event logging technique supports developers in understanding how do applications behave, how they are structured, and in detecting faults such as data losses due to insufficient implementation of activities' life-cycle states. It can likewise help to create models for deriving life-cycle models to other platforms.

The methodology in [29] firstly involved a developer statically adding any missing life-cycle states of activities. The following steps were to create logs for all methods and, through black-box testing, getting a catalog of triggers related to the application so that derivation of life-cycle models could be created.

Furthermore, the authors conducted a study with android apps and concluded that not only often emulators behave differently than real devices, but also the representation of life-cycles for the different platforms diverge a lot.

Dexteroid [36] presents a paper where a framework was developed which, through static analysis, takes the APK of the android application to check activities' life-cycles and creates composite sequential states (states that are related through their transitions), in search of malicious behaviours in these components.

Similarly to previously mentioned works, this paper also depends on logging to detect states of activities. Nevertheless, it differs from others due to the fact that it relies on a static analysis approach and that it has the goal of exploiting applications in search of activities' states sequences that can trigger malicious events such as information leakage and SMS-sending malware attacks.

The dynamic part of the work done is justified because by running applications, it empowers the extraction of these event triggers.

The problem with this approach is the fact that it requires some effort when applications are complex or events require large sequences of states to trigger a malicious event [36].

Although firstly developed to be applied to iOS, the paper in [23] was considered to be inside the topic of this dissertation by cause of the authors citing that they have already "developed an instrumentor for Java", which facilitates the analysis of Android applications.

The work in question presented a mixed reverse engineering technique where firstly the source code was de-compiled and instrumented, through a TL (discussed in 2.3.3) developed by the authors. From here, an AST was built, related to the source code so that the extra events that needed to be created in order to execute the app as a whole using the dynamic crawler could be added.

The conclusions show that there is a down side in using this approach which is mainly because the methodology is based on business relevant use-cases, i.e. whole scenarios from the perspective of the user, "and not on the execution of some arbitrary function"[23]. Thus, it allowed the authors to "make hypothesis regarding the business semantics of the observed patterns of classes' execution" and since they admit not to be sure to "recover all the relevant alternative paths in each of the scenarios" the analysis will be incomplete.

However, the importance of this paper is related with the amount of possible artifacts that it is possible to acquire from state machines, to UML Class Diagrams and even the monitoring of execution of UI classes in the applications.

For a better understanding, an illustration of the reverse engineering methodology that apps are submitted in [23], is visible in Figure 2.9.

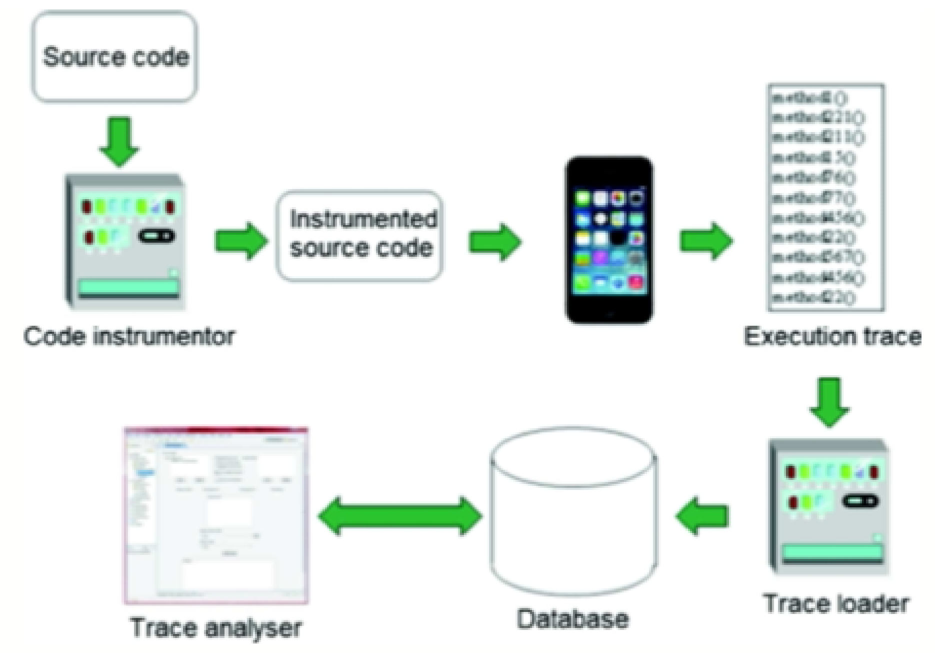


Figure 2.9: Reverse engineering approach [23]

Still in the context of event testing, in [18] there is the presentation of a mixed approach where, statically and, as a first stage, the byte-code is extracted using TLs along with events that the application being reverse engineered contains.

From the extraction process, it is further conducted the byte-code's analysis. Also at this stage, events are analysed by an event tracking algorithm so that it is possible to create groups of events which are supported by the UI of the app.

From here, the events are used as input for dynamic analysis by a crawler algorithm that aims on firing these events in order to generate the model of the application.

With the models, the developed tool generates a state machine where each of the nodes represents the reference to each of the UIs also containing its relations, serving as a tool for generating artifacts that can help understand the paths and events that apps have.

[18] is also referring to this project with a deeper analysis of the entire process, as well as a real implementation of the tool AMOGA.

Reverse engineering of mobile applications through mixed approaches for the extraction/generation of artifacts is the most popular RE technique at the moment of writing from the perspective of related work, and work in [52] is another example of this process. Just as in other examples,

[52] exploits GUI elements searching for its events (actions), using a dynamic crawler algorithm designed by the authors "on top of the Robotium Android test framework" [52].

To achieve the initial input that was later processed and used as input for the dynamic crawler the authors used WALA, a static analysis framework that searches for callback methods and generates a call-graph. ORBIT then used this call graph to map methods that have "intent sending" mechanisms.

Since this was a static, analysis the authors were "unable to infer" some of the "intent-passing mechanisms like intent broadcasting whose behavior is affected by the run-time state of Android", so they used an inferring algorithm to create "action mapping", which finally became the input of the dynamic crawler mentioned before generating a state model of the application.

Once again, the usage of a mixed reverse engineering approach allowed the authors in [52] to achieve better results having complete state models for their case study without any missing paths in their models.

To improve software quality and security in mobile applications [38] presents a paper where a search for anti reverse-engineering techniques is done. This paper, although applied to iOS (authors expect to extend to Android in a near future), is notably relevant because it comes to show an example of refactoring of the app's source code through a reverse engineering process that relied on a mixed analysis technique.

The proposed methodology, in a first part, involves the code instrumentation using a decompiler (e.g. for Android dex2jar mentioned in 2.3.3), extracting byte code of the application as well as the "libraries and framework addresses" contained in the app. The second phase, is composed of a dynamic analysis / instrumentation that relies on a trace logging to be able to save events and interactions in the application.

From the traces, the tool analyses the app identifying similar functions enabling refactor of source code without much effort from developers. The authors concluded that the tool was successfully able to identify similar functions with the correct granularity. [38]

2.5.2 Execution tracing

This reverse engineering technique is mainly used when a mixed approach is being applied, since it requires a code instrumentation before execution of this code to trace the execution flow of the application using an helper tool.

The paper in [51] demonstrates the Execution Tracing technique and depends on a tool developed by the authors, FeatureFinder. The approach presented was composed of three main steps, briefly explained as follows:

1. use FeatureFinder to **instrument** the code and get some execution elements, namely the "call stack at each method call" to identify the methods and their depth , application's packages containing activities and fragments traversed, along with user events. Leaning on the data collected, the tool **creates** a "trace that captures the execution properties observed"[51] through a dynamic analysis;

2. **split** the trace through a bottom-up approach, to be able to perform feature extraction, and joining the events in clusters. When an event is called, all its consequent events are also joined to the cluster;

3. **label** clusters of features in order for humans to understand the knowledge that was extracted (in this case, application's features). The final **output** is a group of terms to that cluster as a label;

The authors also highlighted the importance of getting these elements in particular, defending that these have key information related to "a user execution" and "the features it exercises" [51]. The analysis performed was of great success showing that execution tracing is an applicable technique for reverse engineering of mobile applications, also supported by the fact that Android applications' workflows are event-driven.

Still in the context of Execution tracing, the paper presented in [33] relies on a static analysis of the APK file (2.3.3), followed by a dynamic analysis of the tool in order to trace and create as output a report with logs regarding application properties for complex and large systems.

In order for the static analysis phase to happen, the user has to provide "two pieces of information", these being the byte code that is to be analysed as well as a "PUMAScript", which is a file containing "all information needed for the dynamic analysis"[33].

Further, the script is interpreted by an "interpreter component", which divides the components contained in the script into: "monkey-specified directives and app-specified directives". Then an "instrumenter" performs the static analysis mentioned before, transforming the source code to a version that will be executed in the next phase [33].

The dynamic analysis step takes as input the already instrumented script and executes the application creating logs with events or other key information defined by the user in the PUMAScript that can be traced with this execution.

This paper concluded that the automation of the reverse engineering process of mobile applications can significantly improve effort from companies since the tool is able to take "large-scale" and complex mobile apps, reverse engineer them. From here, it also creates key artifacts for knowledge extraction through a later static analysis by the developers.

2.6 Summary

To summarize, there are numerous contributions from related work regarding tools to reverse engineer mobile android applications. These tools use methods to extract the artifacts, from the simpler AST analysis and testing with logging, all the way to OCR and complex dynamic analysis.

Although the existing related work already extracts numerous artifacts, for instance, event trigger logs, source code's AST and UI information, which empowers the possibility for companies to reduce maintenance costs, there is a lack of tools which are able to reverse engineer android elements including intent messaging objects. These objects contain information regarding application's communication between components and its extraction may represent a considerable part of understanding an application.

Furthermore, related work focuses on mostly building UML diagrams. Even though, activities' analysis is presented in some works, there is still a huge potential for improvement regarding these, as well as, fragments. This is because since activities and fragments contain life-cycles, a real time analysis would enable understanding not only how android iterates through their states, but also the application's workflow.

Additionally, user interface portability is also exploited by a considerable portion of related work. These focus mostly on UI portability, reverse engineering apps, for example, from android to iOS. Reverse engineering UIs can be useful to understand how applications are structured, as well as, their workflow. However, the process itself is much more complex since it has to reverse engineer an image instead of code.

The developed research work scan and mitigation was driven by the goal of analysing the state-of-the-art methodologies regarding reverse engineering of android applications in search of its main gaps. It also aims at exposing the main techniques and methods which are used in the related work, as well as, the produced artifacts in each of them.

The following chapter (3) will present the open issues identified in the context of reverse engineering of android applications. Moreover, a discussion regarding all related work is presented with the goal of identifying the most suitable reverse engineering approach to develop a tool which can tackle the mentioned open issues. Finally, a solution is proposed along with its main contributions, expected results and development tool.

Chapter 3

Research Challenges

The current chapter is organised as follows: 1) explanation of open issues behind the context of this dissertation is presented in section 3.1; 2) analysis to already developed related work is described in section 3.2; 3) a solution is proposed with the goal of tackling the mentioned open issues by reverse engineering applications in section 3.3.

3.1 Open issues

Open issues were the main motivations behind the development of the solution tool discussed bellow. These issues focused on three main problems, including:

- **Android's steep learning curve** - Mobile applications developed in android require much effort from new comers due to android's steep learning curve associated with the wide variety of components that it has to offer.

  Moreover, complexity for the mentioned group of individuals is increased due to the fact that these components may contain life-cycles (e.g., activities/fragments).

- **Companies costs and developers wasted time** - In a modern era where mobile applications are continuously being evolved and maintained, keeping all the artifacts updated and accordingly to applications' workflows, represents huge costs for companies (e.g. need to contract even more developers to be able to maintain the application).

  With the goal of reducing the great amount of time "wasted" to understand applications, one can ensure that the documentation is always totally correct, complete and updated so that developers fully understands the application and its workflow. However, for this to happen, developers need to dedicate extra time to create and maintain the documentation, time which could be used to produce new features or resolve bugs in the applications.

- **New developers integrated into existing teams** - Also related to automation of artifact extraction, new developers joining a team need to scan through all the documentation related to an application in order to understand its architecture. This process can be tedious, confusing and even misleading if the artifacts are not matching the real application's workflow.

-	APK File Analysis	ASTs	Test and Log	CG/DG	ET	OCR
Static	[53], [50], [36], [24], [19]	[25], [37], [35]	X	[34], [42]	X	[43]
Dynamic	X	X	[29], [31], [23]	X	X	X
Mixed	X	X	[38], [52], [18]	X	[51], [33]	X

Table 3.1: Related work content matrix

In order to minimize the effort necessary from new developers to understand applications, one can extract knowledge from android applications through reverse engineering them, and creating artifacts with the knowledge acquired with the help of tools. By automating this process, it is possible to keep artifacts updated all the time since this creation is faster and effortlessly.

Reverse engineering these applications could help automatize artifact extraction which provides the necessary tools for developers to, not only keep up with application's components and workflow, but also to learn how android life-cycle components iterate through their states.

3.2 Discussion

From the perspective presented in Table 3.1, there is a clear cluster of related work which takes advantage not only of APK files (static analysis technique) to perform their reverse engineering process, as well as testing and logging for the same process. When it comes to these two techniques, APK file analysis comes as the top most used technique due to the fact that there are multiple tools that empower the extraction and analysis of Java byte code, which contains fundamental aspects that describe the application being analysed.

On the other hand, Testing and Event Logging (2.5.1) enables reverse engineering complex and/or large applications. Although frequently requiring log specification of event triggers, which is carried out through a static analysis of the source code, it allows posterior usage of these logs to perform a dynamical analysis throughout the application execution. Application's execution further allows to capture information about these events.

The existing related works that handle Abstract Syntax Trees (2.3.1 all rely on the usage of static analysis techniques by cause of these trees only requiring specification of the rules that should be followed to create the hierarchical tree. Thus, only needing static analysis of the source code to build this tree according to the rules defined by the developer.

Simple to create and work with, ASTs take a great part when it comes to extracting artifacts from applications, even though there may be some issues raised when apps reveal to be complex and/or large such as, unreadable artifacts.

Similar to these, Call and Dependency graphs are straightforward ways to extract knowledge through graph modelling of methods or classes when reverse engineering applications relying on a static analysis approach, despite the fact that these are hard to be applied to complex systems and there are numerous element categories (e.g. method conditions) that define the hierarchical structure of the graph.

Also related to complex systems, even if the developer is able to create all the rules without much effort, there is a chance for the graph to be hard to read and not being of much convenience.

Execution tracing technique is similar to 2.5.1 since it depends on tracing the execution flow of the application through logs. Although Execution tracing also adopts a dynamic approach for analysing the application, the difference relies on the fact that in this methodology, in particular, the already developed related work, relies on the usage of a language that defines the structure and conditions for the output report. It then interprets and transforms the code to a syntax corresponding to the tool component that will do the dynamic analysis and extract the final artifact, which is a report with logs that should be statically analysed by developers to extract knowledge.

Finally, the last column visualized in the table is technique that relies on the usage of OCRs. The OCR technique is one of the least adopted techniques because it requires great knowledge about the subject and can become greatly complex to design and develop.

Although there's very few works developed from my knowledge, at the moment of writing, this section is of great importance because it proposes a solution for the issue that complex and large applications are hard to reverse engineer. In particular, transforming UIs between native source code of platforms in an automated manner that can decrease the effort needed from developers for this conversion.

Taking into consideration other research works and the description of Table 3.1, there was a clear chance for improvement regarding reverse engineering tools for android. Although there were a vast variety of related work that tackle various problems, interaction between applications (also mentioned in [18]), as well as, the extraction of fragment's life-cycles are still to be explored.

To combine the extraction of intents and at the same time to be able to visualise in real time fragments and activities, a mixed technique had to be used to empower a source code transformation before a dynamic analysis to these life-cycles. The intent object extraction, for example, happened during the static analysis stage.

Furthermore, in regard to the methodologies mentioned in the related work and Table 3.1, the one of biggest interest in the context of the present dissertation was the use of APK file analysis, which empowers extraction of information related to intents regarding communication with other applications. These rely on a static analysis technique to reverse engineer applications analysing the file without running the code.

Another use of static analysis that presented much interest for the present's dissertation context, was the usage of ASTs which enables the creation of structures containing knowledge related to apps since these are easy to use and empower code transformation based on the structure using a tool like Kadabra [11].

3.3 Solution Approach

A solution (SMART) was proposed to shorten the gap identified through the open issues.

Although there was a wide variety of reverse engineering tools for android mobile applications, which can help mitigate the open issue regarding the huge costs supported by companies, due to lack of automated artifact extraction, there were still some major gaps regarding some of the main android components, such as intents, fragments and activities.

The proposed solution approach consisted of a tool that is able to execute the reverse engineering process extracting knowledge to further build artifacts regarding the intent messaging objects contained in an application, as well as, dependencies classes may have between each other. This way, SMART can possibly tackle the companies' cost open issue by extracting artifacts that were not extracted in an automated manner in other related work. The present gap is also highlighted in one of the analysed papers ([18]).

Moreover, SMART tackled the open issue regarding new coming developers' excessive effort to learn, by providing a learn-as-you-go dynamic tool. Developers are able to interact with the application and, in real time, visualise which activity or fragment's life-cycle method was called.

Finally, by extracting intent's knowledge and classes' dependencies, as well as, empowering life-cycle method's analysis, new recruited developers can easily be integrated into existing teams, since these can use the documentation produced by SMART in order to understand application's workflow.

3.3.1 Methodology

With the goal of tackling the defined open issues, the tool which was developed had to be able to not only extract information from source code, but also transform that code in order to empower the real time analysis, thus needing the usage of a mixed analysis RE technique.

In order to proceed with the analysis and code transformation, a third party tool had to be used to ease the process of parsing code. With this in mind, to start the SMART's process, the tool (Kadabra [11]) had to acquire the app's source code as an input to further analyse and transform it.

From the analysis performed by the tool, intent object's artifacts were to be produced, as well as, a class dependency graph. Furthermore, as part of the tool, a desktop application was developed using JFrame [10], so that it is possible to create the real time visual representation of fragments and activities life-cycles while the user is exploring the application mentioned above.

A key note is the fact that the source code had to be transformed in a way so that it would be possible to send data about a life-cycle method when it was called to the desktop app.

From the analysis of the produced artifacts, as well as, the real time life cycle method invocation, it was possible to acquire key information related to how a specific application's workflow was designed and understand how android handled its components life-cycles states.

Development wise, agile was the chosen methodology composed by weekly sprints. All requirements were specified, user stories created along with punctuation. From here, USs (User Stories) ranked with least points were developed first, followed by bigger ones. The source code related to SMART can be found in [28].

3.3.2 Validation

In order to validate the tool and its produced artifacts, a case study was developed involving application from GitHub [8]. The applications were selected through a criteria, which ensured that apps included the android components analysed by the tool, as well as, these weren't outdated or archived.

Furthermore, a question set was defined, and a series of tests were performed. The question set were created so that each and every of the artifacts that should be generated by SMART were tested.

Through analysing test results along with answering the pre-defined validations questions, it was determined whether or not the main expected goals, as well as, features were properly designed and implemented.

SMART was expected to be successfully validated meaning that it was able to generate the artifacts (intents' information file and dependency graph script)for all applications used in the case study.

Regarding the dynamic analysis, it was expected to visualise all fragments and activities' life-cycle methods that are called upon during an interaction with a user. It shall not miss any of the methods or display an incorrect order of the life-cycle (e.g., onStart() being called before onCreate()).

The following chapter presents a complete detailed description of the SMART tool along with its key components and algorithms.

Chapter 4

SMART

SMART (Static dynaMic Analysis andRoid applicaTions) was the tool developed as a solution for the existing gap in the field of reverse engineering of android applications.

The sections contained in the current chapter are organized as follows: 1) an high level overview of the full tool in section 4.1; 2) a full explanation of the Kadabra Java Weaver's [22] is provided in section 4.2; 3) the algorithms regarding SMART's logic is shown in section 4.3; 4) the desktop app's architecture is presented in section 4.4.

4.1 Overview

SMART can be considered a learn-as-you-go tool since it allows the developer to reverse engineer applications and interact with these extracting activities' life cycles, fragments, intents and dependency graphs. Thus, the tool empowers the possibility for developers to acquire key information of application without needing to spend long periods of time going through documentation.

Given the fact that SMART is composed of a mixed reverse engineering technique it involves two different analysis, firstly a static analysis responsible for generating artifacts and transforming the source code of the mobile application. The second stage is the dynamic analysis, where the transformed code is used along with a JFrame Desktop Application.

An illustration of the entire tool is visible in Figure 4.1 followed by a brief overview of each component.

The static analysis component of SMART uses the Kadabra Java Weaver [11] as a third party tool for analysing the source code. Kadabra works as a layer on top of Spoon [46], facilitating its usage to generate ASTs and analyse/transform code. The inputs required to be provided to the mentioned tool are the java folder of the application containing all of its code and the r folder, which is a build folder containing information related to the application's elements.

After providing the inputs and configuring Kadabra, the Lara script (fully described in section 4.3) should be ran so that it can analyse the code and create the outputs defined in the context of the tool. These outputs are the intents / intent filters' information and a script inside a ".txt" file that enables the visual representation of a dependency graph.

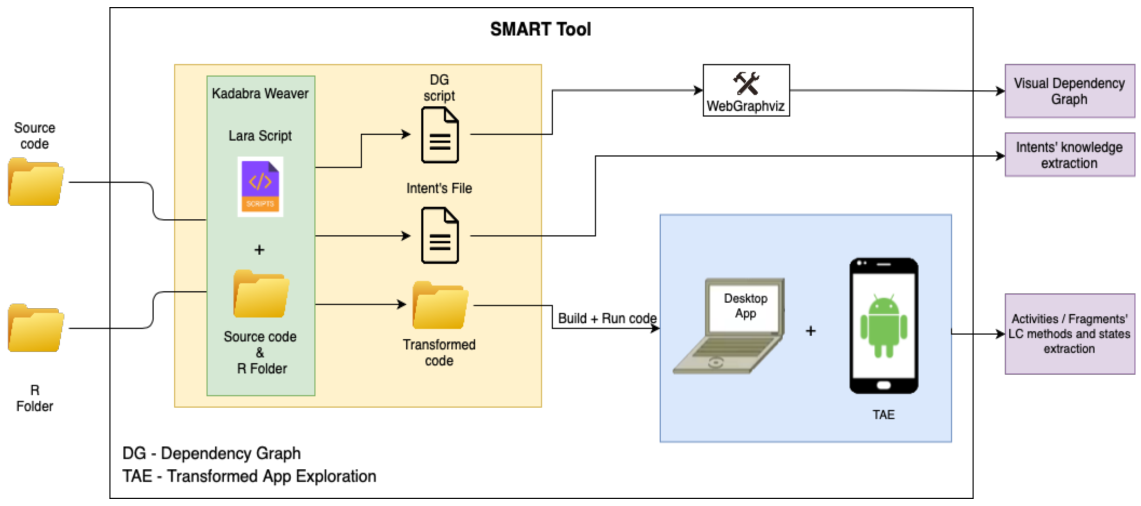


Figure 4.1: SMART Tool

Also, while the Lara script is running, it searches for the launcher activity class to insert a script related to creating the socket that is used in the dynamic analysis, as well as, finding activities and fragments in order to add the code needed to send data related to their life cycle methods. Since not all methods are mandatory to be implemented, if any of these are not implemented in an app, Kadabra adds the code related to them.

As already mentioned, the dynamic analysis part of the tool takes advantage of a JFrame Desktop Application. The application's purpose was to enable the possibility to visualise, in real time, which activity/fragment's life cycle methods were being called. It relies on a socket connection to the mobile app in order to receive the data to each life cycle method call.

Upon receiving data, the app analyses it and displays an image related to the method being called and the name of the activity or fragment.

A more detailed description concerning the architecture of the Desktop app is presented in 4.4.

For full details related to the required configuration by Kadabra and installation refer to the manuals in the SMART's repository [28].

4.2 Kadabra Java Weaver

In order to understand the Kadabra Java Weaver, some concepts are required to be firstly introduced.

LARA [22] is a language that relies on a styles' combination. The first used style is the declarative style responsible for querying source code and the other style is the imperative one, which "applies actions to each code element (e.g., add monitoring code before the code element)" [21].

Kadabra Java Weaver is one of the compilers that LARA relies on, namely for Java language. Another example of compiler, applied to C/C++ languages is Clava ([21]).

The complete Kadabra Java Weaver's architecture is illustrated in Figure 4.2. Furthermore, it can be considered as consisting in three main components, namely: Java Frontend, Kadabra Java Weaver and the LARA Engine.

The Java Frontend component is characterized by the Kadabra AST Dumper and the Kadabra AST Loader. Its main purpose is to transform the source code provided by the user and abstract it into the Kadabra's Abstract Syntax Tree to empower further manipulation based on the referred source code, as well as, transformation back to its original form by the Kadabra Java Weaver.

The Kadabra AST Dumper is a Java app that utilizes a version of Spoon, which is a library that parses source files to further "analyze, rewrite, transform and transpile" [46] code.

From parsing performed by Kadabra AST Dumper, a structure containing syntactic and semantic information linked to the code is passed to Kadabra's AST Loader which is responsible for creating a Kadabra's AST instance that matches the source code. Although similar to the AST generated by Spoon, using a Kadabra's AST allows generating source code from the abstract syntax tree and transforming code also based on the AST. This capability is one of the main reasons for using Kadabra Java Weaver to perform a static analysis since it allows to perform some functions that would be infeasible using Spoon.

Kadabra Java Weaver Engine is the component in charge of granting information around Java. The same component is also responsible for retaining an internal representation of the application's code, according to LARA strategies. Finally, the Kadabra Java Weaver empowers the connection between the Java source code folder and LARA code execution.

LARA Framework [22], is centralized on the belief that source code regarding the definition of functionalities about an app should be described independently tasks such as adaptive behavior. The LARA Engine is contained in the LARA Framework, thus being responsible for executing the Lara Script (described in 4.3). The script contains tasks which are enforced to source code while compiling.

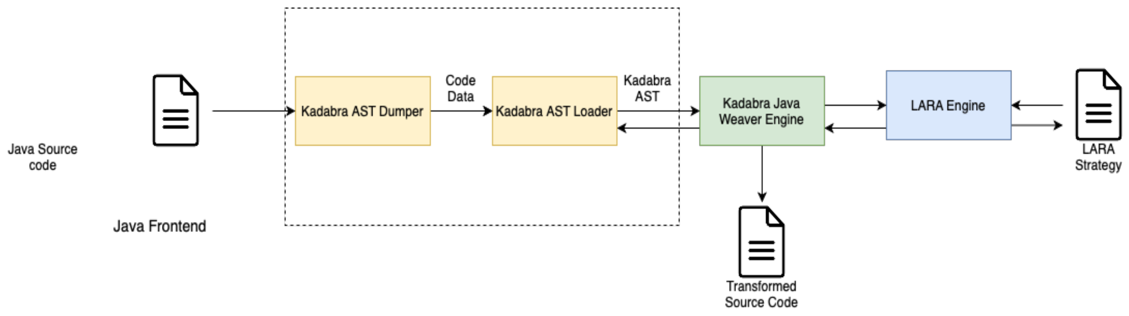


Figure 4.2: Kadabra Java Weaver

4.3 Lara File Script

The Lara File Script refers to a file containing Lara code responsible for performing the entire static analysis process. It is divided into two main moments, these being: 1) artifacts extraction 2) source code transformation. The artifacts extraction moment is when not only intent's information is extracted, but also the dependency graph script is generated.

In order to acquire the artifacts, classes are analysed one at a time and for each of them, local variables are to be scanned in search of the ones that are of the types "Intent" and "Intent Filter". On the condition that an intent exists, the algorithm saves relevant information about it into an array and proceed to the next variable in that class.

Once all variables of all the classes are analysed, Kadabra generates a folder named "Static Analysis" and in it places a text file and adds the information acquired from intents and intent filters. The mentioned information is composed of the following:

- Name of the intent;
- Type of intent (Intent or Intent Filter);
- Line of code, as well as, the code itself;
- File name of the class where the intent is declared;

Regarding the script that concedes the generation of the dependency graph, Kadabra firstly adds some configurations related to the size and direction of the graph, as well as, the shape associated with classes. It then scans the classes in order to save their names to an array.

A second scan is performed in search for local variables whose type matches one of the strings in the array of class names. When a match is accomplished, Kadabra adds a line to the dependency graph file of the type with the following form:

$$\alpha \text{ -> } \beta \text{ [label = "}\gamma\text{"]}$$

Where α is the dependent class, β is the class which has a class dependent on it and γ refers to the file also the line where the dependency is declared.

Once the script is fully generated, it can be copied to the WebGraphviz tool [17], since it is the tool that matches the syntax created by the Lara script, in order to visualise the full dependency graph. For a better understanding related to the logic behind intents and dependency graph's extraction refer to Algorithms 1 and 2 correspondingly.

Algorithm 1 Intent Search and File generation

```
1: var array = []
2: for classes do
3:     for variables of classes do
4:         var type = variables.type
5:         if type === "Intent" || type === "IntentFilter" then
6:             Save to intentVar the intended structure for each intent
7:             array.push(intentVar)
8:         end if
9:     end for
10: end for
11: if !array.length then
12:     Write to file informing no existing intents (Io.writeFile())
13: else
14:     Write array to intended file
15: end if
```

Algorithm 2 Dependency Graph Script Generation

```
1: Add initial configurations to file (size, direction, shape...)
2: var classesArray = []
3: Save classes' names to array classesArray
4: for classes do
5:     for variables from classes do
6:         Save variable type to var type
7:         if classesArray.includes(type) then
8:             Create configuration regarding dependency
9:             classes name -> type [label=variables information]
10:            Append line to dependencyGraph.txt file
11:        end if
12:    end for
13: end for
```

When it comes to the transformation of the source code to enable interaction with the Desktop app, the Lara script is composed of three different stages namely: socket creation, activities' life cycle methods search and fragments' life cycle methods search.

The socket creation involves scanning through the source code of the app searching for its launcher activity. In the context of SMART, and throughout the present dissertation, "MainActivity" was considered to be the name of the launcher activity class. When the mentioned class is found, and since this class is already an activity, Kadabra adds imports related to the socket library along with the code snippet that will create the socket using the IP of the local machine where it is running, as well as, a defined port.

The snippet also contains a PrintWriter that is responsible for sending the life cycle methods data via socket to the desktop app. From here, it checks for implemented life cycle methods on the MainActivity to further add the messages to be sent to the Desktop app. As already mentioned, and for the next discussed code transformations, if a method is not implemented, Kadabra adds the code referring to that method.

The pseudo-code related to the socket creation is presented in Algorithm 3.

Algorithm 3 Main Activity Analysis

```
1:  for $classes$ do
2:      if $classes$="$MainActivity$" then
3:          Insert imports to be used in $Background$ class (socket class)
4:          Insert code regarding the $Background$ class
5:          Instantiate $Background$ class
6:          for var $lifecyclemethods$ in $classes$ do
7:              if $lifecycle$ exists then
8:                  Add $Background.execute(lifecycle)$ to existing $lifecycle$
9:              else
10:                 Create $lifecycle$ method $+$ add $Background.execute(lifecycle)$
11:             end if
12:         end for
13:     end if
14: end for
```

Once the socket creation is completed, activities are analysed next by Kadabra. For this, the script analysis all classes in search for the ones that inherit the android class AppCompatActivity (base class for activities that allows the implementation of newer features). When a class that satisfies the mentioned condition is found, Kadabra transforms the code to contain the snippet related to sending life cycle methods' data.

When comparing the analysis performed in regard to activities and the fragment analysis which is to be performed next in the process, these are very similar with the main differences being the fact that fragments must inherit "Fragment" android class instead of the "AppCompatActivity" class, and the life cycle method snippets are also different.

Both algorithms behind the implementation of Activities and Fragments' life cycle methods are presented in 4 and 5 as pseudo-code respectively.

Algorithm 4 Remaining Activities' Analysis

```
for $classes$ do
    if $classes$ !== "MainActivity" && $classes.superClass$ === "AppCompatActivity" then
        Instantiate $Background$ class
        for $lifecyclemethods$ in $classes$ do
            if $lifecycle$ exists then
                Add $Background.execute(lifecycle)$ to existing $lifecycle$
            else
                Create $lifecycle$ method + add $Background.execute(lifecycle)$
            end if
        end for
    end if
end for
```

Algorithm 5 Fragments' Analysis

```
for $classes$ do
    if $classes.superClass$ == "Fragment" then
        Instantiate $Background$ class
        for $lifecyclemethods$ in $classes$ do
            if $lifecycle$ exists then
                Add $Background.execute(lifecycle)$ to existing $lifecycle$
            else
                Create Fragment's $lifecycle$ method + add $Background.execute(lifecycle)$
            end if
        end for
    end if
end for
```

At the present moment in the process, the static analysis performed by the Lara Script file is completed. The artifacts are already generated in the Static Analysis folder, empowering the ability to extract knowledge from the app by analysing these. Also, the code of the mobile app is already transformed to be further used along with the desktop application as described in section 4.4. For visualisation example of these artifacts please refer to Figures 5.1 and 5.2.

4.4 Desktop Application

Desktop application refers to the JFrame developed app, designed to receive data from the mobile application to later handle it and display the graph state according life cycle method. Its main goal was to enable knowledge extraction in regard to activities and fragments' life cycle methods.

To achieve the method's data display in real time, the communication between desktop and mobile apps was performed relying on a TCP/IP Protocol Socket. Transformed source code by Kadabra already specifies both the IPv4 related to the user of the tool and the port to which the desktop app should connect so that the communication acts as intended.

Although a pipe line would work as a solution for SMART since the desktop app does not send any type of data back to the mobile app, socket communication was selected enabling future work to implement sending any type of data back to the mobile application without much effort in code refactoring.

When the connection is established, an exploration should be carried out in order to trigger life cycle method's data to be sent from the mobile app. Once a method is triggered, the mobile app uses the socket class to forward the data.

The desktop app receives data and analysis it, firstly checking if it is either a fragment or activity to later verify which of the methods was called. Desktop app displays both the state of the life cycles' graphics and the name of the element containing the method.

To facilitate understanding of the socket architecture in SMART Figure 4.3 represents the described communication between applications.

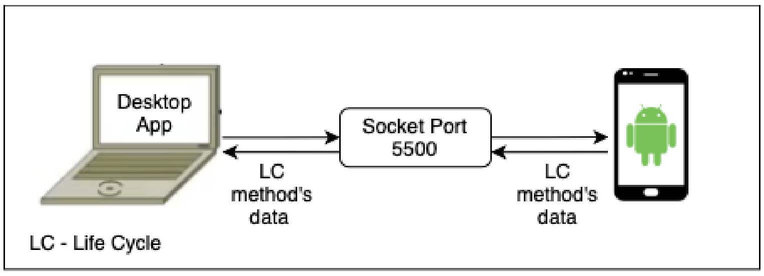


Figure 4.3: Socket connection architecture

Note that although port 5500 is defined as the communication port, the user still needs to verify its IPv4 and place it in the Lara Script's socketCreation() function.

4.5 Summary

To conclude, SMART provides a number of different features, from intent's information and class dependency graphs, to real-time analysis of android activities and fragments. Companies can benefit from all the extracted knowledge to provide documentation to new developers being introduced in teams and reduce their effort to understand the software.

Moreover, new android developers can benefit from using SMART by exploiting applications in a learn-as-you-go approach to understand how android activities and fragments' life-cycles operate and iterate through their states.

The following chapter will present the case study process to which the SMART tool was put through in order to be validated.

Chapter 5

Case Study

The present chapter aims to provide a description of the complete case study to which the SMART tool had to go through in order to be validated.

The sections are organised as follows: 1) section 5.1 presents a brief overview related to each phase; 2) section 5.2 describes the selection criteria for apps used in the case study; 3) section 5.3 introduces all questions that had to be validated; 4) section 5.4 has the goal of describing the testing process of SMART; 5) section 5.5 demonstrates acquired results, conclusions and limitations of SMART.

5.1 Overview

As already mentioned, SMART tool is applicable to mobile apps namely android applications. However, since there are numerous existing apps, a selection had to be made according to certain pre-established constraints. In order to do so, firstly a criteria was defined. It was composed of constraints for selecting the applications that were to be later used in the case study.

Constraints were not only used to ensure that applications contained the components that were to be analysed by SMART, so that all components of the tool could be validated, but also to ensure that applications were from a range of different categories (described in 5.2), so that it could be proven that SMART can analyse different types of android applications.

Following the criteria definition and using the GitHub platform [8] for the selection of apps, filtering was carried out, and four applications were selected 5.2 to later validate SMART. Each of these was adopted as input in Kadabra [11] to not only generate the transformed code that would be later applied as a way of validating the desktop application, but also to generate the Class Dependency Graph and the Intent/IntentFilter.txt file.

After all the artifacts were extracted, results were analysed (discussed in 5.5) in order to verify the consistency, integrity and behaviour in different environments of the tool developed in the context of the present dissertation. Since SMART relies on a mixed analysis technique, the analysis of results from the case study was composed of two different phases.

For the results of the static analysis part, files related to the dependency graphs were tested against the third party tool WebGraphwiz [17] (black-box testing) and the intent files were manually analysed to check for errors or uncaught cases.

Regarding result analysis of the dynamic part of the tool, applications were explored while being connected to the desktop application via a socket where the main goal was to validate if all activities and fragments' life-cycle states were shown correctly, meaning that the code was transformed correctly.

A detailed description of each of the case study phases of the process is presented in the following sections.

5.2 Criteria Definition

Initially, as already mentioned, a collection of constraints for filtering applications that would be used in the case study had to be defined. The definition ensured that the case study involved apps from different categories and that these contained the components that are to be analysed and extracted.

The criteria used in the case study was composed of the following constraints:

1. Applications had to be fully developed in Java language since Kadabra Java Weaver would not capture code developed in Kotlin files;
2. Selected applications had to be from at most 2016;
3. Selected applications had to have commits from at most four years ago;
4. Selected application had to have at least 2 contributors;
5. The pool of applications in the case study had to have activities / fragments / intents / intent filters;
6. There had to be 1 application for each of the 4 different categories and these included:
    - Utility Application - SimpleAlarms
    - Health Applications - FitnessMonitor
    - Business Applications - AppTycoon
    - Educational Applications - Juda
7. Selected applications had to have at least 5 stars;
8. Preferably applications were licensed and had forks;

From the initial 183,672 repositories captured in GitHub [8] related to android mobile applications that were fully developed in Java, only 6,186 of these were considered once we filtered the ones developed before 2016 and with commits from at least 2017. Moreover, after applying filters for stars and licensing, only 704 repositories were left.

At this point, the category filter was applied to reduce the pool of apps even further. The most common category was the gaming category mostly composed by quizzes and the least used category was the business category.

In order to select apps which had fragments, activities and intents, a manual search had to be done through the final collection of projects and for each of the categories one was selected. Also, this manual search of the final pool of apps was justified by the fact that some applications did not generate the R folder, a folder containing build information which was imperative to transform code in Kadabra.

5.3 Case Study Questions

The Case Study Questions, as the name suggests, involved the selection and creation of the questions that had to be answered in order for the SMART tool could be validated. To formulate these questions, the goal outputs of SMART were analysed. On a high level, these were:

1. The ability to extract specific artifacts from applications through source code analysis (dependency graph code snippet and intents' information);
2. The ability to transform the source code to be used along with the desktop application;

From analysing the intended outputs of the tool, the set of questions was created. The following subsections present the questions set along with a description for each of the questions, where each of these aims to validate an output of SMART.

5.3.1 Is SMART able to extract the Intents and Intent Filters' Information?

As previously mentioned, the first step performed in the SMART tool is to analyse an application's source code using Kadabra and the Lara script, to produce artifacts through static analysis, the first one being intents.

In order for SMART to pass the question, it must correctly perform an analysis through the source code. Furthermore, it must generate a file in the Kadabra folder (named "Static Analysis") named "Intents&IntentFilters.txt" and containing for all intents in the application: 1) the name of the intent; 2) its type (Intent or Intent Filter); 3) file name and line referring where it was created; 4) the code of the intent declaration.

5.3.2 Is SMART able to extract the script related to the dependency graph?

Another part of the static analysis involves the generation of a script that, when copied to a third party tool, creates a visualisation of the dependency graph related to the analysed application.

This question aims at validating whether the tool is able to successfully generate the dependency graph script. SMART is validated assuming that all dependencies are generated and there are no existing faults or "extra" dependencies.

5.3.3 Is SMART able to transform the source code correctly?

The main goal of the present question is to validate whether or not SMART is able to successfully generate all the required transformed code of the application. In particular, the socket that will be responsible to create the communication between the Android and Desktop applications.

It will also be in charge of validating the transformed code which sends information about existing activities and fragments' life cycle methods, as well as, added life cycle methods that were missing to these.

If the state methods are displayed for all activities / fragments on the desktop application, it means that the code was correctly transformed to perform the analysis.

5.4 Experiment

After having the four applications to validate SMART along with the "to-be-answered" case study questions, a black-box testing technique was performed, which is a technique that involves providing an input to the software and checking its result output.

Testing was done by providing the source code that was to be analysed to Kadabra, as well as the "r" folder (discussed in 4) along with the packages that the application uses. From here, and after running Kadabra to generate the intended outputs, the intent files were manually analysed in order to check for faults or any missing intent declarations.

Moreover, the script contained in the "dependencyGraph.txt" file was placed in the tool WebGraphviz [17] to generate the graphs and verify if these were correct. The graph visualisation allowed understanding how classes were related on the apps through their dependencies.

A manual analysis for comparing dependencies shown in the visual graph and the ones on the source code was performed by scanning lines presented in the visual version of the graph and verifying if the dependency created in the source code corresponds to its visual representation.

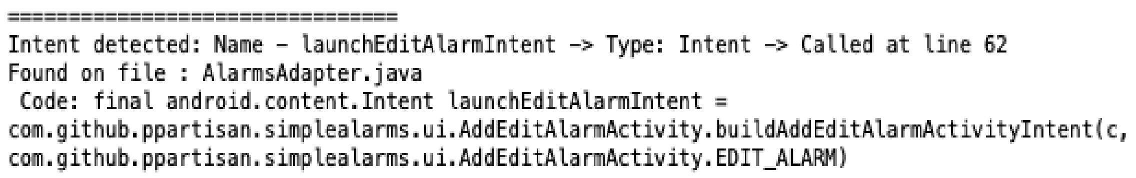
Furthermore, the transformed code was placed in the application's source folder and permissions were added to the manifest file. The desktop application was initialized thus enabling applications' exploration.

The exploration served as a way of validating if transformed code by the Lara script was done correctly and if all fragments and activities' life cycles methods were added.

5.5 Results and Limitations

The results showed that the case study was of much success. This was regarding the static analysis, SMART was able to produce the correct artifacts for all applications related to intents and class dependency graphs. For illustration purposes, a simple example from the case study applications of the intent file and dependency graph are shown in Figures 5.1 and 5.2.

For example, the first element in Figure 5.1 shows that an intent was detected in the source code, named "launchEditAlarmIntent", found at line 62 of the type Intent. It also displays the code for that intent and the name of the file, in this case "AlarmsAdapter.java". If there were no intents in the analysed application, a warning message would be displayed instead. The rest of the case study results for intents can be found in Appendix.



```
================================
Intent detected: Name - launchEditAlarmIntent -> Type: Intent -> Called at line 62
Found on file : AlarmsAdapter.java
 Code: final android.content.Intent launchEditAlarmIntent =
com.github.ppartisan.simplealarms.ui.AddEditAlarmActivity.buildAddEditAlarmActivityIntent(c,
com.github.ppartisan.simplealarms.ui.AddEditAlarmActivity.EDIT_ALARM)
```

Figure 5.1: Intent File example

As for the visual dependency graph example in Figure 5.2, the interpretation can be the fact that the class "MainFragment" has a dependency to class "EmptyRecyclerView" on line 48 of the file "MainFragment.java" and the class "AlarmReceiver" has 2 dependencies to the class "Alarm", in lines 46 and 106 located on the file "AlarmReceiver.java".

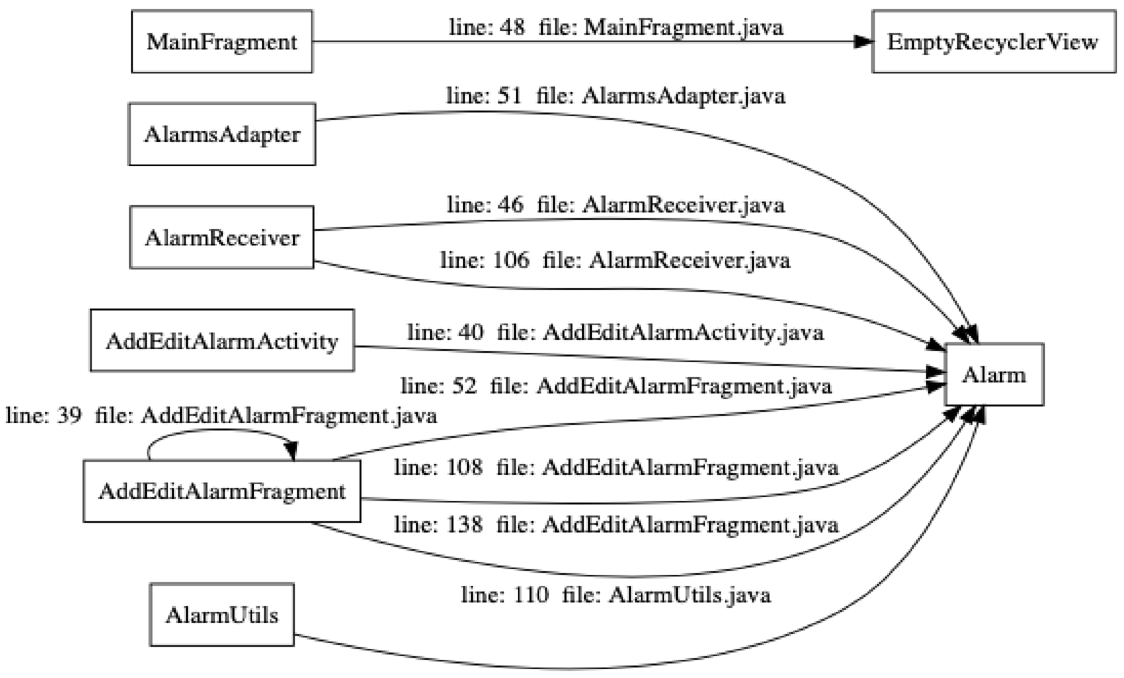

MainFragment
line: 48 file: MainFragment.java
EmptyRecyclerView
AlarmsAdapter
line: 51 file: AlarmsAdapter.java
AlarmReceiver
line: 46 file: AlarmReceiver.java
line: 106 file: AlarmReceiver.java
AddEditAlarmActivity
line: 40 file: AddEditAlarmActivity.java
Alarm
line: 52 file: AddEditAlarmFragment.java
line: 39 file: AddEditAlarmFragment.java
AddEditAlarmFragment
line: 108 file: AddEditAlarmFragment.java
line: 138 file: AddEditAlarmFragment.java
AlarmUtils
line: 110 file: AlarmUtils.java

Figure 5.2: Dependency Graph example

Moreover, SMART was able to transform the source code from applications to be used in the dynamic analysis. In order to validate the results of this phase of the tool, an exploration of the case study apps was performed and results presented in the desktop app were compared with the life cycles in the source code of the application confirming that all fragments and activity life cycle methods were implemented.

An illustration of the "MainActivity" life cycle state transitions is showed is Figure 5.3, where it is visible the call to each method, namely onStart(), onResume() and onPause().

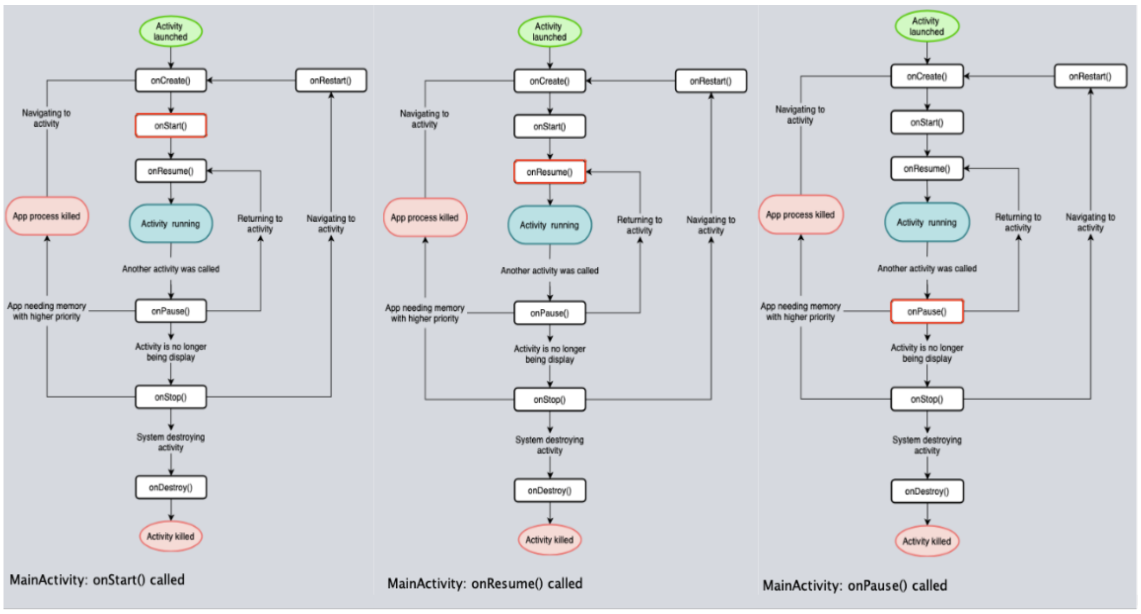

Activity launched
onCreate()
onStart()
onResume()
onRestart()
Navigating to activity
App process killed
Activity running
Returning to activity
Another activity was called
App needing memory with higher priority
onPause()
Activity is no longer being display
onStop()
System destroying activity
onDestroy()
Activity killed
MainActivity: onStart() called
MainActivity: onResume() called
MainActivity: onPause() called

Figure 5.3: SMART - Activities' Life cycle transition example

On the other hand, some limitations emerge with SMART regarding the dynamic analysis and some of the results presented exemplify these limitations. As mentioned in chapter 4 when performing the analysis, Kadabra verifies the name of the superClass of each class. This is due to the fact that in order to transform the source code, it needs to know whether the class is an activity, fragment, etc.

Taking this fact into consideration, if a fragment does not inherit a class named "Fragment" (Android fragment class), the script will not be able to find that fragment and add missing life cycle methods. When Kadabra analyses the source code it needs to add full name of elements, thus searching for the "android.support.v4.app.Fragment" class.

The same limitation applies to activities although these have to inherit "AppCompatActivity" (Android activity class). An example of the described limitations for activities and fragments are found in the business application used in the case study A.2. In the referred app, a base fragment and activity were developed, thus having some of the remaining fragments/activities to inherit these base classes.

Due to the inheritance from these fragments/activities, Kadabra was not able to catch them through a static analysis, to further transform their code and send information to the desktop app.

Still regarding activities, and in particular the launcher activity of the analysed applications, another limitation of SMART is the fact that the mentioned activity is obliged to be named "Main-Activity". This is due to the fact that Kadabra searches for a launcher activity where the code related to the socket script, which is responsible for connecting to the desktop application, has to be placed. In this case, Kadabra will search for class "android.support.v7.app.AppCompactActivity".

Regarding launcher activity's limitation naming by SMART, it is exposed in the educational app A.3. In the case of the present application, it was not possible to perform the dynamic analysis component of SMART as a result of the launcher activity being named "HomeActivity".

5.6 Summary

A case study was performed composed of some android applications from different categories picked from GitHub. Experiments were performed in order to validate whether SMART was able to reverse engineer applications. Regarding the static analysis stage, check if SMART was able to extract intent information (code, class, file, etc.), dependency graph scripts. The second stage of the experiments involved an exploration through the applications' UIs verifying if SMART transformed the source code correctly, by looking at the active activities and/or fragments at a given moment.

The results showed that SMART can in fact reverse engineer applications and product the mentioned artifacts, although there are some limitations regarding the launcher activity.

The next chapter will present conclusion regarding the tool and the case study, as well as, contributions from the development of SMART and future work which will focus on tackling the existing SMART limitations.

Chapter 6

Conclusions

Nowadays, where complex/ large applications are always being evolved and maintained, companies have to support enormous costs to be able to maintain apps most of the times due to the lack of correct and updated artifacts. Additionally, companies continuously add new members to their teams. Thus, correct artifacts are essential to ease the effort required from these developers in order to understand applications and their workflows.

The proposed solution which is presented throughout this dissertation is a mixed analysis reverse engineering, thus containing two different stages. The static analysis part is responsible for analysing code searching for elements that are to be extracted as artifacts, these being intents and a class dependency graph. While the static analysis is being performed, the source code is also transformed to contain some code snippets in the activities and fragments life-cycle methods. The static analysis component uses a third party tool, Kadabra Java Weaver to perform these actions.

The second component, the dynamic analysis, relies on a desktop application which was developed as part of the tool. It is responsible for communicating with the mobile application under analysis through a socket, handling data received from this communication and displaying the correct life-cycle method which was invoked at a given moment.

Results were analysed confirming that SMART is valid since the outputs were correctly produced. Some limitations were raised, as for example, the fact that the launcher activity must be names "MainActivity".

6.1 Contributions

The development of SMART enrich its research area with various innovations. Firstly, the implemented tool distinguishes it from other related work as being an innovative reverse-engineering tool that relies on a mixed (static and dynamic code analysis) approach to aid on the understanding of the source code behind an android mobile app, through generating artifacts containing intent objects.

Another innovation of SMART's tool is its ability to raise the awareness on which activity or fragment is being "called upon" at a given moment in time along with its triggered life-cycle state.

This feature can be used in an educational context for new comers to understand how does android work.

6.2 Future Work

Regarding future work, SMART has some limitations that can be improved in future iterations. Starting with code transformation, SMART's future work should involve the implementation of a method to scan for the launcher activity dynamically which would improve SMART's versatility to handle applications.

In the case of activities and fragments the implementation of a way to find the classes correspondent to these elements dynamically will also be implemented as future work. Searching for all fragments' sub classes (e.g., FragmentActivity, ListFragment ,etc.) will enrich SMART's quality widening its use cases to applications that may not use directly the class "Fragment".

Moreover, a nice-to-have requirement that may be implemented is a method that sends information from the desktop app back to the mobile app informing that the correct life-cycle method was already displayed.

References

[1] Android operating system. http://en.wikipedia.org/wiki/Android_(operating_system), Accessed on 2020-10-18.

[2] Apktool: A tool for reverse engineering 3rd party, closed, binary android apps. https://ibotpeaches.github.io/Apktool/, Accessed on 2021-01-03.

[3] Appium: Mobile app automation made awesome. http://appium.io/, Accessed on 2021-01-05.

[4] Broadcasts Overview: Android Developers. https://developer.android.com/guide/components/broadcasts, Accessed on 2020-10-19.

[5] Dex2jar Tool. https://tools.kali.org/reverse-engineering/dex2jar, Accessed on 2021-02-15.

[6] Fragment lifecycle: Android developers. https://developer.android.com/guide/fragments/lifecycle, Accessed on 2020-10-19.

[7] Fragments. https://developer.android.com/guide/fragments, Accessed on 2020-10-19.

[8] GitHub Website. https://github.com/, Accessed on 2021-05-30.

[9] How Long Do Tech Pros Stay in Their Jobs? https://insights.dice.com/2016/07/08/how-long-do-tech-pros-stay-in-their-jobs/, Accessed on 2021-02-11.

[10] JFrame - Java Platform SE 7. https://docs.oracle.com/javase/7/docs/api/javax/swing/JFrame.html, Accessed on 2021-04-24.

[11] Kadabra tool. http://specs.fe.up.pt/tools/kadabra/, Accessed on 2021-03-01.

[12] Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide, Accessed on 2020-10-18.

[13] Services Overview | Android Developers. https://developer.android.com/guide/components/services, Accessed on 2021-04-23.

[14] Static Program Analysis. https://en.wikipedia.org/w/index.php?title=Static_program_analysis&oldid=1029248787, Accessed on 2021-04-14.

[15] Understand Intents and Intent Filters: Android Developers. https://developer.android.com/guide/components/intents-filters, Accessed on 2020-10-19.

[16] Understand the Activity Lifecycle: Android Developers. https://developer.android.com/guide/components/activities/activity-lifecycle, Accessed on 2020-10-19.

[17] WebGraphviz Tool. http://www.webgraphviz.com, Accessed on 2021-05-03.

[18] Salihu Anka, Rosziati Ibrahim, and Aida Mustapha. A Hybrid Approach for Reverse Engineering GUI Model from Android Apps for Automated Testing. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, 9:45–49, 2017.

[19] A. Armando, G. Chiarelli, Gabriele Costa, Gabriele De Maglie, R. Mammoliti, and A. Merlo. Mobile App Security Analysis with the MAVeriC Static Analysis Module. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 5:103–119, 2014.

[20] Yauhen Leanidavich Arnatovich, Lipo Wang, Ngoc Minh Ngo, and Charlie Soh. A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation. *IEEE Access*, 6:12382–12394, 2018.

[21] João Bispo and João M.P. Cardoso. Clava: C/C++ source-to-source compilation using LARA, 2020.

[22] João M.P. Cardoso, Tiago Carvalho, José G.F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. LARA: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development - AOSD '12*, page 179, Potsdam, Germany, 2012. ACM Press.

[23] Philippe Dugerdil and Roland Sako. Dynamic Analysis Techniques to Reverse Engineer Mobile Applications. In Pascal Lorenz, Jorge Cardoso, Leszek A. Maciaszek, and Marten van Sinderen, editors, *Software Technologies*, pages 250–268, Cham, 2016. Springer International Publishing.

[24] Eman Elsayed, Kamal Eldahshan, Enas El-Sharawy, and Naglaa Ghannam. Reverse engineering approach for improving the quality of mobile applications. *PeerJ Computer Science*, 5:e212, 2019.

[25] Esa Fauzi, Bayu Hendradjaya, and Wikan Danar Sunindyo. Reverse engineering of source code to sequence diagram using abstract syntax tree. In *2016 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6, 2016.

[26] Jorge Ferreira and Ana C. R. Paiva. Android Testing Crawler. In Mario Piattini, Paulo Rupino da Cunha, Ignacio García Rodríguez de Guzmán, and Ricardo Pérez-Castillo, editors, *Quality of Information and Communications Technology*, pages 313–326. Springer International Publishing, 2019.

[27] Nuno Flores, Ana C. R. Paiva, and Nuno Cruz. Teaching Software Engineering Topics Through Pedagogical Game Design Patterns: An Empirical Study. *Information*, 11(3), 2020.

[28] Francisco Serrão. Repository: SMART Methodology, 2021. https://github.com/franciscoserrao/SMART.

[29] Dominik Franke, Corinna Elsemann, Stefan Kowalewski, and Carsten Weise. Reverse Engineering of Mobile Application Lifecycles. In *2011 18th Working Conference on Reverse Engineering*, pages 283–292, 2011.

[30] Ivo Gomes, Tiago Gomes, Pedro Morgado, and Rodrigo Moreira. An overview on the Static Code Analysis approach in Software. pages 1–15, 2009.

[31] Marco A. Gonçalves and Ana C. R. Paiva. Reverse Engineering of Android Applications: REiMPAcT. In Martin J. Shepperd, Fernando Brito e Abreu, Alberto Rodrigues da Silva, and Ricardo Pérez-Castillo, editors, *Quality of Information and Communications Technology - 13th International Conference, QUATIC 2020, Faro, Portugal, September 9-11, 2020, Proceedings*, volume 1266 of *Communications in Computer and Information Science*, pages 369–382. Springer, 2020.

[32] André M. P. Grilo, Ana C. R. Paiva, and João Pascoal Faria. Reverse engineering of GUI models for testing. In *5th Iberian Conference on Information Systems and Technologies*, pages 1–6, 2010.

[33] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, page 204–217, New York, NY, USA, 2014. Association for Computing Machinery.

[34] Masahiro Hata, Masashi Nishimoto, Keiji Nishiyama, H. Kawabata, and T. Hironaka. OS-AIFU: A Source Code Factorizer on Android Studio. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 422–425, 2019.

[35] Ruihua Ji, J. Pei, Wenhua Yang, Juan Zhai, Minxue Pan, and Tian Zhang. Extracting Mapping Relations for Mobile User Interface Transformation. *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, 2019.

[36] Mohsin Junaid, Donggang Liu, and David Kung. Dexteroid: Detecting malicious behaviors in Android apps using reverse-engineered life cycle models. *Computers & Security*, 59:92–117, 2016.

[37] Khalid Lamhaddab and Khalid Elbaamrani. Model Driven Reverse Engineering: Graph Modeling for mobiles platforms. In *2015 15th International Conference on Intelligent Systems Design and Applications (ISDA)*, pages 392–397, 2015.

[38] Yibin Liao, Ruoyan Cai, Guodong Zhu, Yue Yin, and Kang Li. MobileFindr: Function Similarity Identification for Reversing Mobile Binaries. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, pages 66–83. Springer International Publishing, 2018.

[39] Inês Coimbra Morgado, A. C. R. Paiva, and J. P. Faria. Dynamic Reverse Engineering of Graphical User Interfaces. *International Journal on Advances in Software*, 5(4):224–236, 2012.

[40] Inês Coimbra Morgado and Ana C. R. Paiva. Testing Approach for Mobile Applications through Reverse Engineering of UI Patterns. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 42–49, 2015.

[41] Inês Coimbra Morgado, Ana C. R. Paiva, João Pascoal Faria, and Rui Camacho. GUI Reverse Engineering with Machine Learning. In *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*, pages 27–31, 2012.

[42] Tuan Anh Nguyen. *Tools For Program Understanding And Reverse-engineering Of Mobile Applications*. PhD thesis, University of Texas, Arlington, 2017.

[43] Tuan Anh Nguyen and Christoph Csallner. Reverse Engineering Mobile Application User Interfaces with REMAUI. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 248–259, 2015.

[44] Ana C. R. Paiva, João C. P. Faria, and Pedro M. C. Mendes. Reverse Engineered Formal Models for GUI Testing. In Stefan Leue and Pedro Merino, editors, *Formal Methods for Industrial Critical Systems, 12th International Workshop, FMICS 2007, Berlin, Germany, 2007, Revised Selected Papers*, volume 4916 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2007.

[45] Ana C.R. Paiva, Nuno H. Flores, André G. Barbosa, and Tânia P.B. Ribeiro. ilearntest – framework for educational games. *Procedia - Social and Behavioral Sciences*, 228:443–448, 2016. 2nd International Conference on Higher Education Advances,HEAd'16, 2016, València, Spain.

[46] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[47] Clara Sacramento and Ana C. R. Paiva. Web Application Model Generation through Reverse Engineering and UI Pattern Inferring. In *9th International Conference on the Quality of Information and Communications Technology, QUATIC 2014, Guimaraes, Portugal*, pages 105–115. IEEE Computer Society, 2014.

[48] Carlos Eduardo Silva and José Creissac Campos. Combining Static and Dynamic Analysis for the Reverse Engineering of Web Applications. In Peter Forbrig, Pason Dewan, Michael Harrison, Kris Luyten, Carmen Santoro, and Simone Barbosa, editors, *Proceedings of the 5th Symposium on Engineering Interactive Computing Systems - EICS*, pages 107–112, London, United Kingdom, 2013. ACM, ACM.

[49] Eleni Stroulia and Tarja Sys. Dynamic analysis for reverse engineering and program understanding. *SIGAPP Applied Computing Review*, 1, 2002.

[50] May Thu Kyaw, Yan Naung Soe, and Nang Saing Moon Kham. Security Analysis of Android Application by Using Reverse Engineering. In *11th International Conference on Future Computer and Communication - WCSE 2019 Spring*, 2019.

[51] Qi Xin, Farnaz Behrang, Mattia Fazzini, and Alessandro Orso. Identifying Features of Android Apps from Execution Traces. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 35–39, 2019.

[52] Wei Yang, Mukul R. Prasad, and Tao Xie. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 250–265, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[53] Kirill Zhigalov and Viacheslav Ivanov. Reverse engineering of mobile applications. *IOP Conference Series: Materials Science and Engineering*, 537:052005, 2019.

Appendix A

Appendix

The present chapter contains all the appendix related to the test experiment results with all four applications.

A.1 Health Application - Case Study

Regarding the static analysis artifacts, Figures A.29, A.30 and A.31 present the extracted Intent's file, the Dependency Graph script and its visualisation correspondingly. Moreover, Figures [A.32,A.33, A.34, A.35, A.36, A.37, A.38, A.39] aim at displaying some of the invoked life cycle methods by android throughout the health application's exploration. The layout differs from others in the 5 with the goal of showing another view of SEMANTIC's desktop app, which performs an analysis and only display methods related to activities.

A.2 Business Application - Case Study

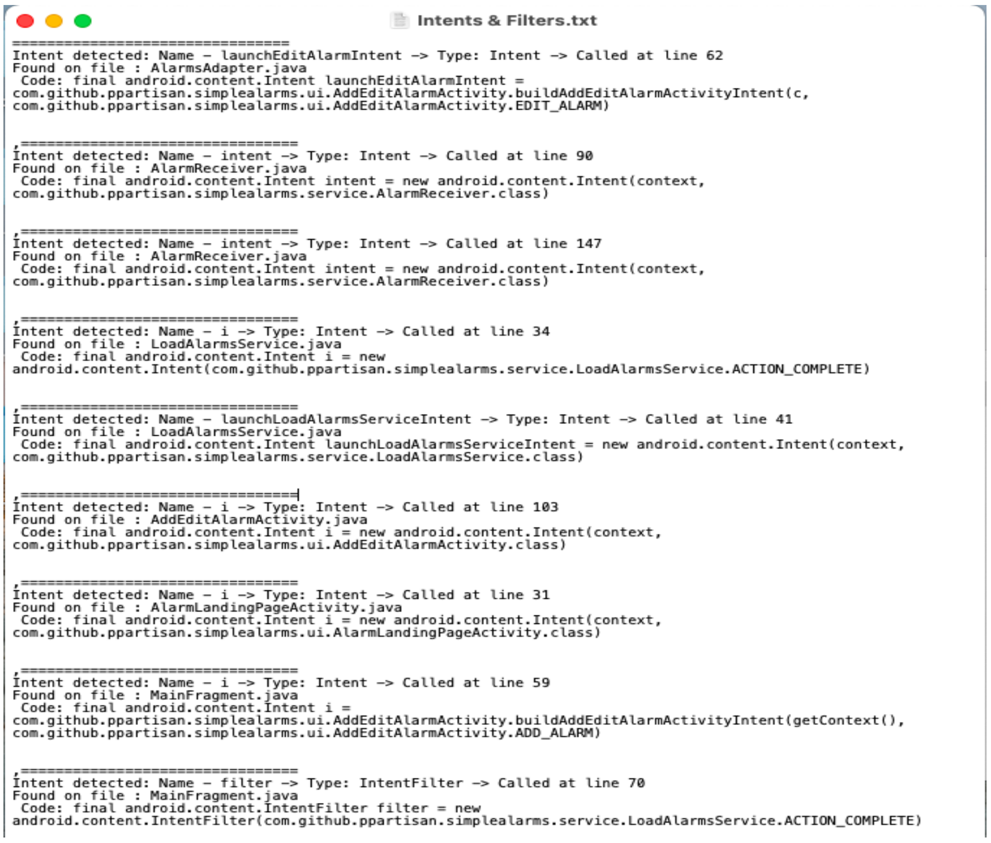
Related to the static analysis artifacts in the business app, Figure A.12 show the extracted Intent's file. Furthermore, Figures in [A.13, A.14] and [A.15, A.16, A.17, A.18] the Dependency Graph script and its visualisation correspondingly. Images [A.19 ,A.20, A.21, A.22, A.23, A.24, A.25] represent some of the life cycle methods invoked by the Android System, thus displayed in the Desktop app throughout the exploration performed.

A.3 Educational Application - Case Study

Launcher Activity in the educational application was not named "MainActivity", therefore it was not possible to perform the dynamic analysis. However, as for the static analysis artifacts, Figures A.26, A.27 and A.28 show the extracted Intent's file, the Dependency Graph script and its visualisation correspondingly.

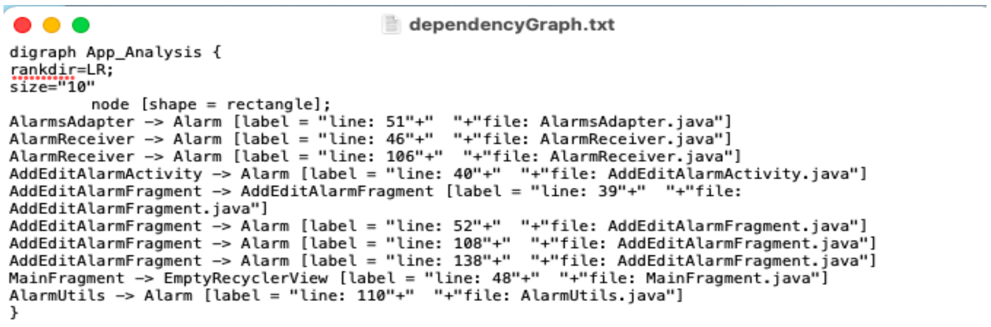
A.4 Utility Application - Case Study

Regarding the static analysis artifacts, Figures A.1, A.2 and A.3 show the extracted Intent's file, the Dependency Graph script and its visualisation correspondingly. Images [A.4 ,A.5, A.6, A.7, A.8, A.9, A.10, A.11] represent some of the life cycle methods invoked by the Android System, thus displayed in the Desktop app throughout the exploration performed in the utility app from the case study.



```
=================================
Intent detected: Name - launchEditAlarmIntent -> Type: Intent -> Called at line 62
Found on file : AlarmsAdapter.java
 Code: final android.content.Intent launchEditAlarmIntent =
com.github.ppartisan.simplealarms.ui.AddEditAlarmActivity.buildAddEditAlarmActivityIntent(c,
com.github.ppartisan.simplealarms.ui.AddEditAlarmActivity.EDIT_ALARM)


,=================================
Intent detected: Name - intent -> Type: Intent -> Called at line 90
Found on file : AlarmReceiver.java
 Code: final android.content.Intent intent = new android.content.Intent(context,
com.github.ppartisan.simplealarms.service.AlarmReceiver.class)


,=================================
Intent detected: Name - intent -> Type: Intent -> Called at line 147
Found on file : AlarmReceiver.java
 Code: final android.content.Intent intent = new android.content.Intent(context,
com.github.ppartisan.simplealarms.service.AlarmReceiver.class)


,=================================
Intent detected: Name - i -> Type: Intent -> Called at line 34
Found on file : LoadAlarmsService.java
 Code: final android.content.Intent i = new
android.content.Intent(com.github.ppartisan.simplealarms.service.LoadAlarmsService.ACTION_COMPLETE)


,=================================
Intent detected: Name - launchLoadAlarmsServiceIntent -> Type: Intent -> Called at line 41
Found on file : LoadAlarmsService.java
 Code: final android.content.Intent launchLoadAlarmsServiceIntent = new android.content.Intent(context,
com.github.ppartisan.simplealarms.service.LoadAlarmsService.class)


,=================================
Intent detected: Name - i -> Type: Intent -> Called at line 103
Found on file : AddEditAlarmActivity.java
 Code: final android.content.Intent i = new android.content.Intent(context,
com.github.ppartisan.simplealarms.ui.AddEditAlarmActivity.class)


,=================================
Intent detected: Name - i -> Type: Intent -> Called at line 31
Found on file : AlarmLandingPageActivity.java
 Code: final android.content.Intent i = new android.content.Intent(context,
com.github.ppartisan.simplealarms.ui.AlarmLandingPageActivity.class)


,=================================
Intent detected: Name - i -> Type: Intent -> Called at line 59
Found on file : MainFragment.java
 Code: final android.content.Intent i =
com.github.ppartisan.simplealarms.ui.AddEditAlarmActivity.buildAddEditAlarmActivityIntent(getContext(),
com.github.ppartisan.simplealarms.ui.AddEditAlarmActivity.ADD_ALARM)


,=================================
Intent detected: Name - filter -> Type: IntentFilter -> Called at line 70
Found on file : MainFragment.java
 Code: final android.content.IntentFilter filter = new
android.content.IntentFilter(com.github.ppartisan.simplealarms.service.LoadAlarmsService.ACTION_COMPLETE)
```

Figure A.1: Utility Application's Intent File



```
digraph App_Analysis {
rankdir=LR;
size="10"
          node [shape = rectangle];
AlarmsAdapter -> Alarm [label = "line: 51"+"  "+"file: AlarmsAdapter.java"]
AlarmReceiver -> Alarm [label = "line: 46"+"  "+"file: AlarmReceiver.java"]
AlarmReceiver -> Alarm [label = "line: 106"+"  "+"file: AlarmReceiver.java"]
AddEditAlarmActivity -> Alarm [label = "line: 40"+"  "+"file: AddEditAlarmActivity.java"]
AddEditAlarmFragment -> AddEditAlarmFragment [label = "line: 39"+"  "+"file:
AddEditAlarmFragment.java"]
AddEditAlarmFragment -> Alarm [label = "line: 52"+"  "+"file: AddEditAlarmFragment.java"]
AddEditAlarmFragment -> Alarm [label = "line: 108"+"  "+"file: AddEditAlarmFragment.java"]
AddEditAlarmFragment -> Alarm [label = "line: 138"+"  "+"file: AddEditAlarmFragment.java"]
MainFragment -> EmptyRecyclerView [label = "line: 48"+"  "+"file: MainFragment.java"]
AlarmUtils -> Alarm [label = "line: 110"+"  "+"file: AlarmUtils.java"]
}
```

Figure A.2: Utility Application's Dependency Graph File Script

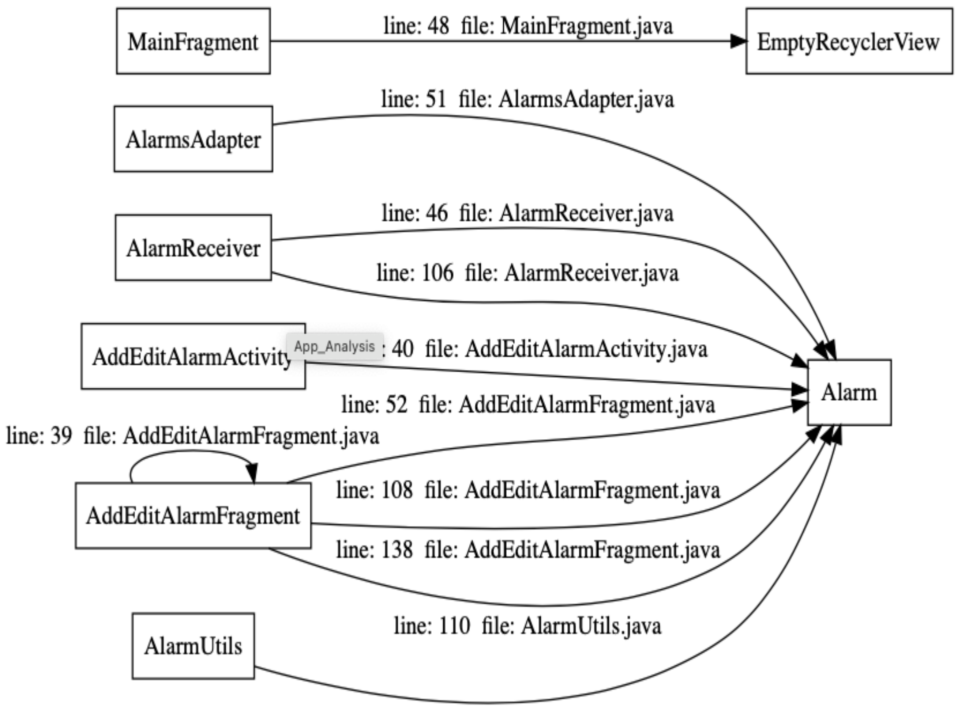
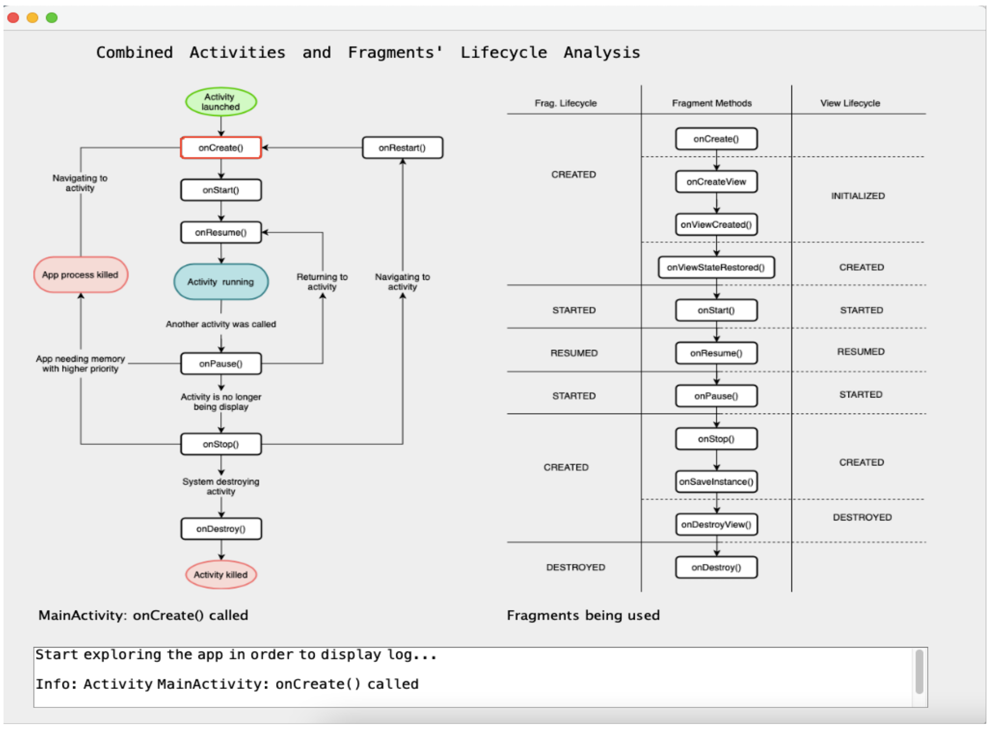

MainFragment
line: 48 file: MainFragment.java
EmptyRecyclerView
AlarmsAdapter
line: 51 file: AlarmsAdapter.java
AlarmReceiver
line: 46 file: AlarmReceiver.java
line: 106 file: AlarmReceiver.java
AddEditAlarmActivity
App_Analysis
40 file: AddEditAlarmActivity.java
Alarm
line: 52 file: AddEditAlarmFragment.java
line: 39 file: AddEditAlarmFragment.java
AddEditAlarmFragment
line: 108 file: AddEditAlarmFragment.java
line: 138 file: AddEditAlarmFragment.java
AlarmUtils
line: 110 file: AlarmUtils.java

Figure A.3: Utility Application’s Dependency Graph Visualisation



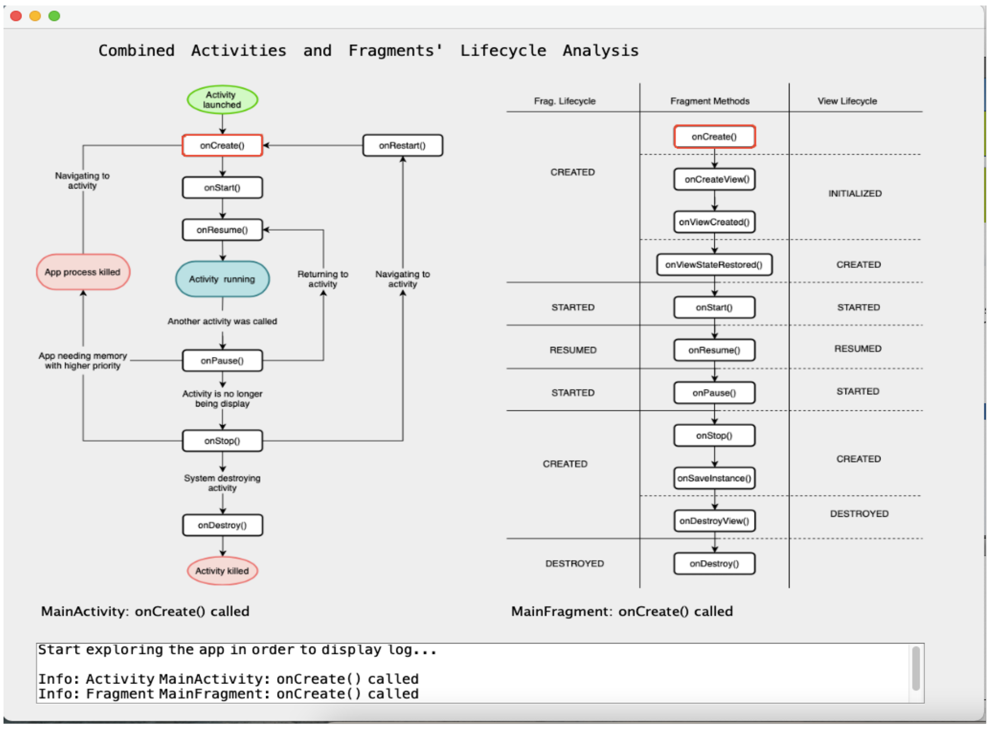
Combined Activities and Fragments' Lifecycle Analysis

MainActivity: onCreate() called

Fragments being used

```
Start exploring the app in order to display log...
Info: Activity MainActivity: onCreate() called
```

Figure A.4: Utility Application: Desktop App Displayed Method 1

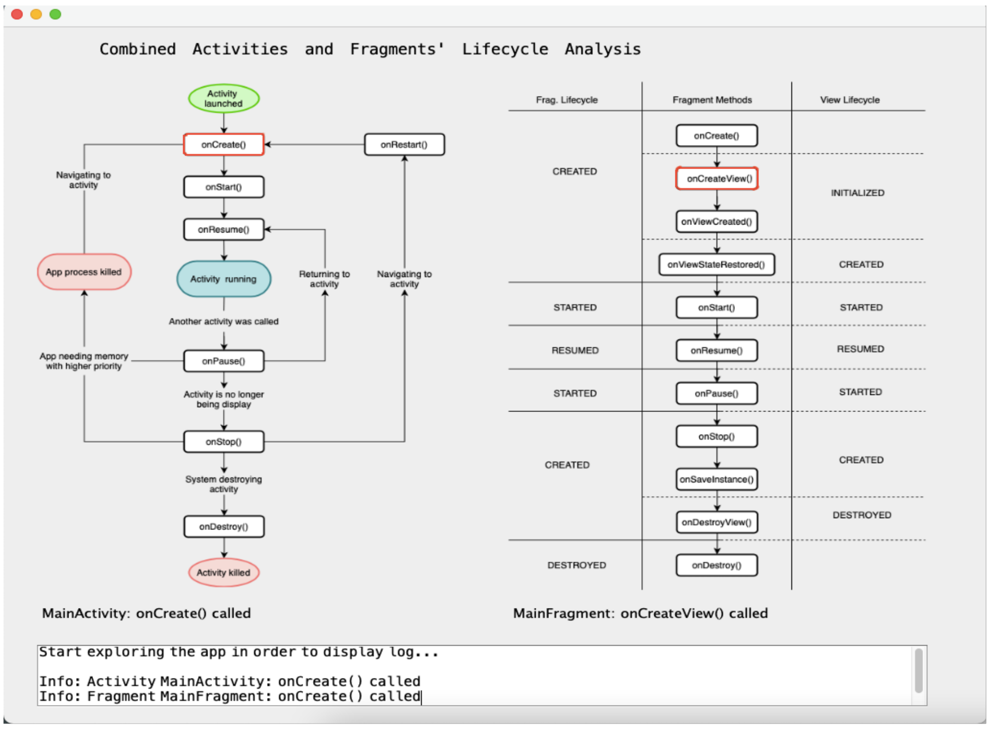

Combined Activities and Fragments' Lifecycle Analysis

MainActivity: onCreate() called

MainFragment: onCreate() called

```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Fragment MainFragment: onCreate() called
```

Figure A.5: Utility Application: Desktop App Displayed Method 2

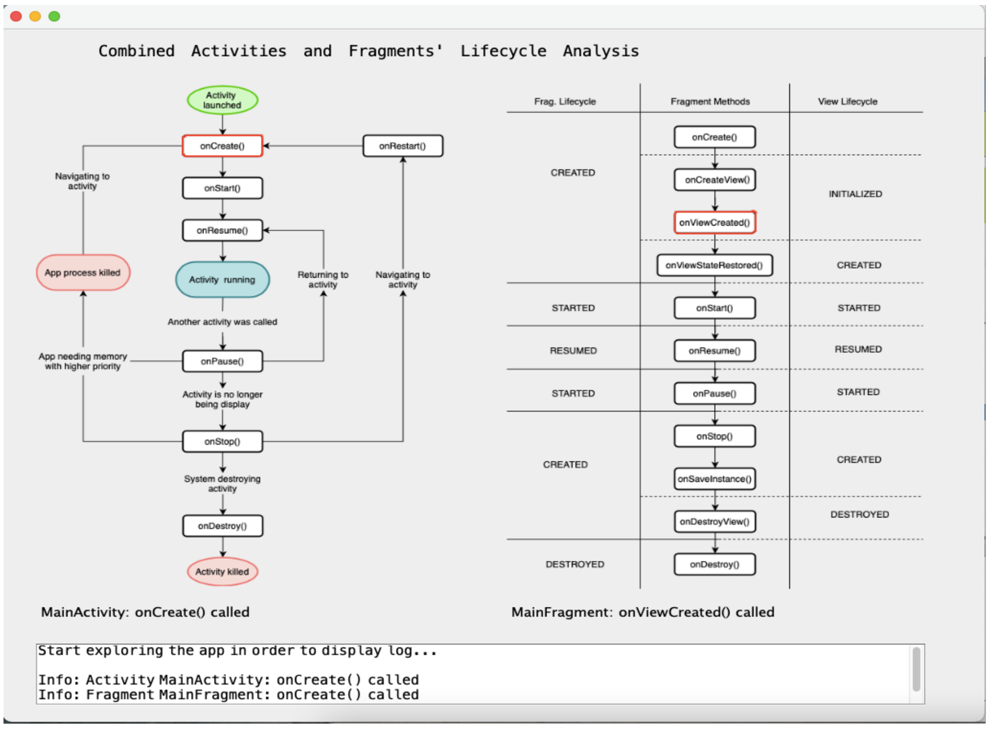

Combined Activities and Fragments' Lifecycle Analysis

MainActivity: onCreate() called

MainFragment: onCreateView() called

```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Fragment MainFragment: onCreate() called
```

Figure A.6: Utility Application: Desktop App Displayed Method 3

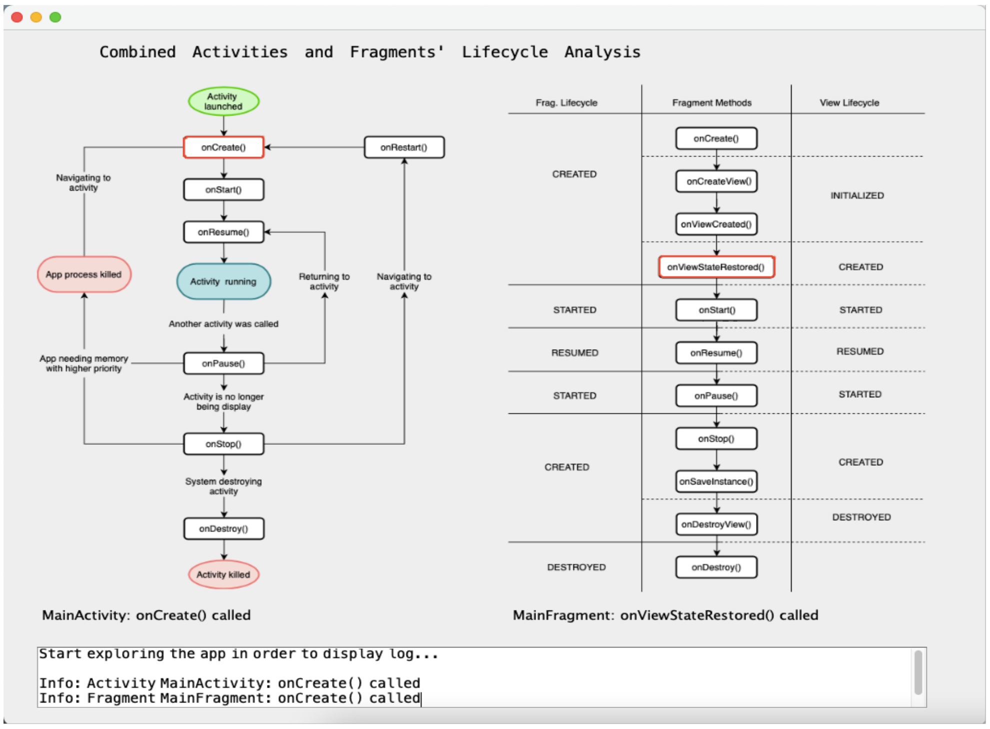


MainActivity: onCreate() called

MainFragment: onViewCreated() called

```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Fragment MainFragment: onCreate() called
```

Figure A.7: Utility Application: Desktop App Displayed Method 4

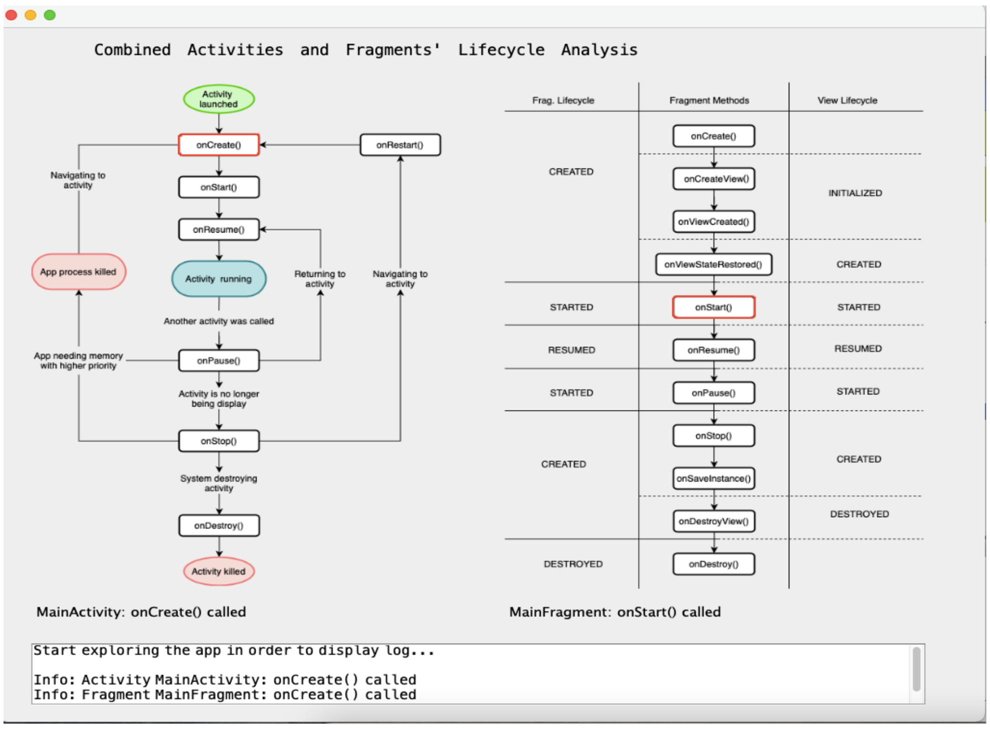


MainActivity: onCreate() called

MainFragment: onViewStateRestored() called

```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Fragment MainFragment: onCreate() called
```

Figure A.8: Utility Application: Desktop App Displayed Method 5



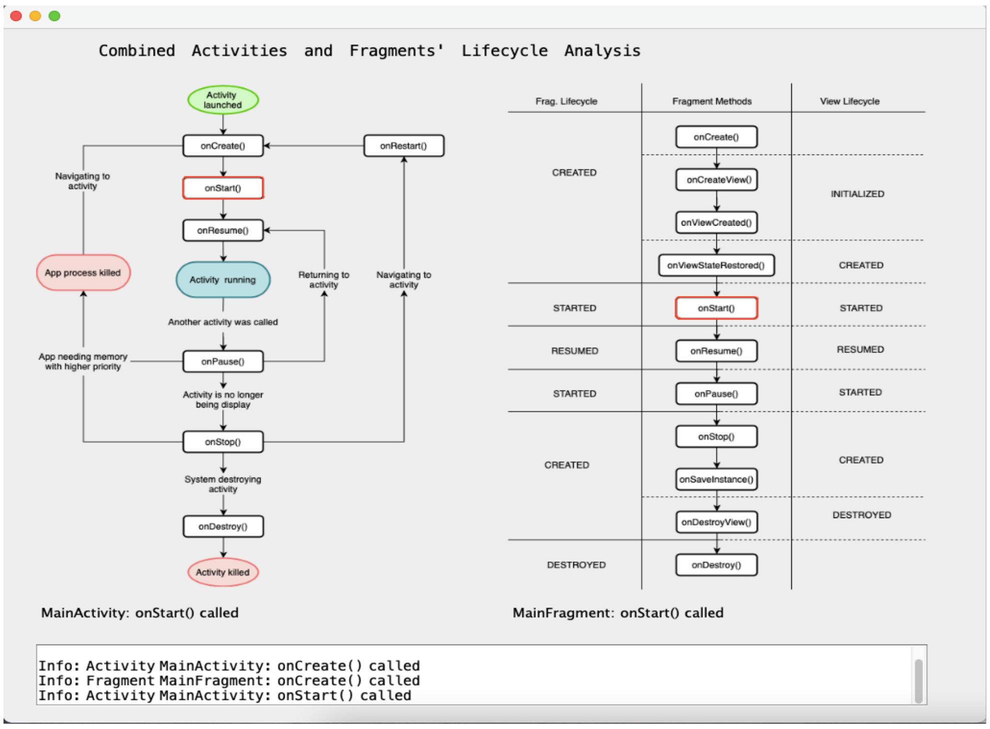
Combined Activities and Fragments' Lifecycle Analysis

MainActivity: onCreate() called

MainFragment: onStart() called

```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Fragment MainFragment: onCreate() called
```

Figure A.9: Utility Application: Desktop App Displayed Method 6



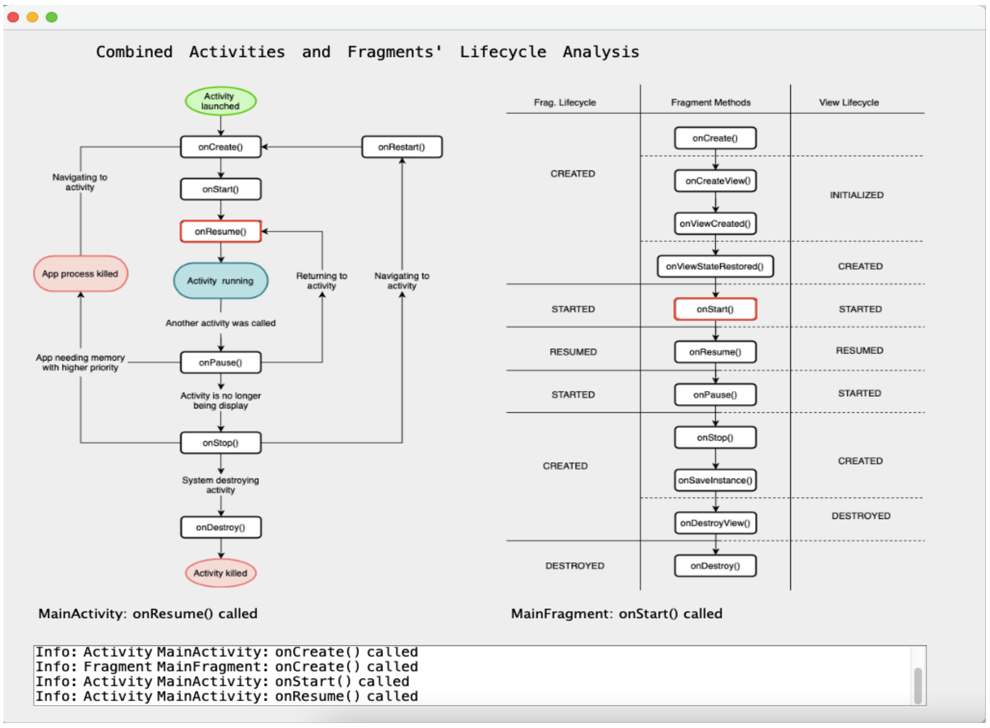
Combined Activities and Fragments' Lifecycle Analysis

MainActivity: onStart() called

MainFragment: onStart() called

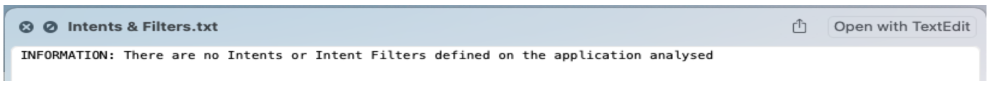
```
Info: Activity MainActivity: onCreate() called
Info: Fragment MainFragment: onCreate() called
Info: Activity MainActivity: onStart() called
```

Figure A.10: Utility Application: Desktop App Displayed Method 7



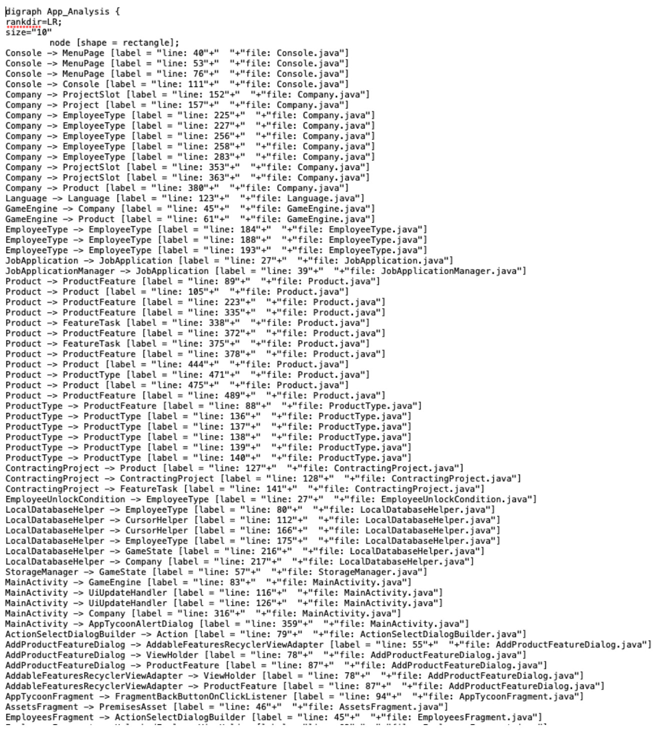
```
Info: Activity MainActivity: onCreate() called
Info: Fragment MainFragment: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
```

Figure A.11: Utility Application: Desktop App Displayed Method 8



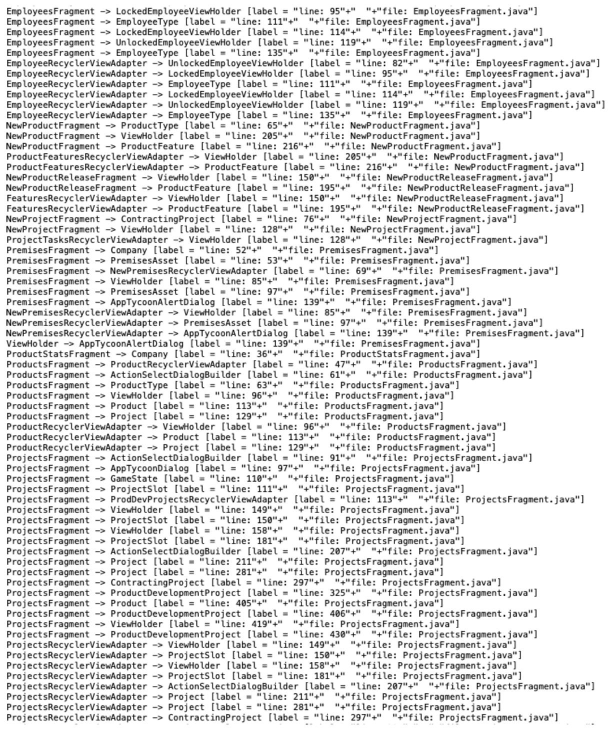
```
INFORMATION: There are no Intents or Intent Filters defined on the application analysed
```

Figure A.12: Business Application's Intent File



```
digraph App_Analysis {
rankdir=LR;
size="10"
        node [shape = rectangle];
Console -> MenuPage [label = "line: 40"+"  "+"file: Console.java"]
Console -> MenuPage [label = "line: 53"+"  "+"file: Console.java"]
Console -> MenuPage [label = "line: 76"+"  "+"file: Console.java"]
Console -> Console [label = "line: 111"+"  "+"file: Console.java"]
Company -> ProjectSlot [label = "line: 152"+"  "+"file: Company.java"]
Company -> Project [label = "line: 157"+"  "+"file: Company.java"]
Company -> EmployeeType [label = "line: 225"+"  "+"file: Company.java"]
Company -> EmployeeType [label = "line: 227"+"  "+"file: Company.java"]
Company -> EmployeeType [label = "line: 256"+"  "+"file: Company.java"]
Company -> EmployeeType [label = "line: 258"+"  "+"file: Company.java"]
Company -> EmployeeType [label = "line: 283"+"  "+"file: Company.java"]
Company -> ProjectSlot [label = "line: 353"+"  "+"file: Company.java"]
Company -> ProjectSlot [label = "line: 363"+"  "+"file: Company.java"]
Company -> Product [label = "line: 380"+"  "+"file: Company.java"]
Language -> Language [label = "line: 123"+"  "+"file: Language.java"]
GameEngine -> Company [label = "line: 45"+"  "+"file: GameEngine.java"]
GameEngine -> Product [label = "line: 61"+"  "+"file: GameEngine.java"]
EmployeeType -> EmployeeType [label = "line: 184"+"  "+"file: EmployeeType.java"]
EmployeeType -> EmployeeType [label = "line: 188"+"  "+"file: EmployeeType.java"]
EmployeeType -> EmployeeType [label = "line: 193"+"  "+"file: EmployeeType.java"]
JobApplication -> JobApplication [label = "line: 27"+"  "+"file: JobApplication.java"]
JobApplicationManager -> JobApplication [label = "line: 39"+"  "+"file: JobApplicationManager.java"]
Product -> ProductFeature [label = "line: 89"+"  "+"file: Product.java"]
Product -> Product [label = "line: 105"+"  "+"file: Product.java"]
Product -> ProductFeature [label = "line: 223"+"  "+"file: Product.java"]
Product -> ProductFeature [label = "line: 335"+"  "+"file: Product.java"]
Product -> FeatureTask [label = "line: 338"+"  "+"file: Product.java"]
Product -> ProductFeature [label = "line: 372"+"  "+"file: Product.java"]
Product -> FeatureTask [label = "line: 375"+"  "+"file: Product.java"]
Product -> ProductFeature [label = "line: 378"+"  "+"file: Product.java"]
Product -> Product [label = "line: 444"+"  "+"file: Product.java"]
Product -> ProductType [label = "line: 471"+"  "+"file: Product.java"]
Product -> Product [label = "line: 475"+"  "+"file: Product.java"]
Product -> ProductFeature [label = "line: 489"+"  "+"file: Product.java"]
ProductType -> ProductFeature [label = "line: 88"+"  "+"file: ProductType.java"]
ProductType -> ProductType [label = "line: 136"+"  "+"file: ProductType.java"]
ProductType -> ProductType [label = "line: 137"+"  "+"file: ProductType.java"]
ProductType -> ProductType [label = "line: 138"+"  "+"file: ProductType.java"]
ProductType -> ProductType [label = "line: 139"+"  "+"file: ProductType.java"]
ProductType -> ProductType [label = "line: 140"+"  "+"file: ProductType.java"]
ContractingProject -> Product [label = "line: 127"+"  "+"file: ContractingProject.java"]
ContractingProject -> ContractingProject [label = "line: 128"+"  "+"file: ContractingProject.java"]
ContractingProject -> FeatureTask [label = "line: 141"+"  "+"file: ContractingProject.java"]
EmployeeUnlockCondition -> EmployeeType [label = "line: 27"+"  "+"file: EmployeeUnlockCondition.java"]
LocalDatabaseHelper -> EmployeeType [label = "line: 80"+"  "+"file: LocalDatabaseHelper.java"]
LocalDatabaseHelper -> CursorHelper [label = "line: 112"+"  "+"file: LocalDatabaseHelper.java"]
LocalDatabaseHelper -> CursorHelper [label = "line: 166"+"  "+"file: LocalDatabaseHelper.java"]
LocalDatabaseHelper -> EmployeeType [label = "line: 175"+"  "+"file: LocalDatabaseHelper.java"]
LocalDatabaseHelper -> GameState [label = "line: 216"+"  "+"file: LocalDatabaseHelper.java"]
LocalDatabaseHelper -> Company [label = "line: 217"+"  "+"file: LocalDatabaseHelper.java"]
StorageManager -> GameState [label = "line: 57"+"  "+"file: StorageManager.java"]
MainActivity -> GameEngine [label = "line: 83"+"  "+"file: MainActivity.java"]
MainActivity -> UiUpdateHandler [label = "line: 116"+"  "+"file: MainActivity.java"]
MainActivity -> UiUpdateHandler [label = "line: 126"+"  "+"file: MainActivity.java"]
MainActivity -> Company [label = "line: 316"+"  "+"file: MainActivity.java"]
MainActivity -> AppTycoonAlertDialog [label = "line: 359"+"  "+"file: MainActivity.java"]
ActionSelectDialogBuilder -> Action [label = "line: 79"+"  "+"file: ActionSelectDialogBuilder.java"]
AddProductFeatureDialog -> AddableFeaturesRecyclerViewAdapter [label = "line: 55"+"  "+"file: AddProductFeatureDialog.java"]
AddProductFeatureDialog -> ViewHolder [label = "line: 78"+"  "+"file: AddProductFeatureDialog.java"]
AddProductFeatureDialog -> ProductFeature [label = "line: 87"+"  "+"file: AddProductFeatureDialog.java"]
AddableFeaturesRecyclerViewAdapter -> ViewHolder [label = "line: 78"+"  "+"file: AddProductFeatureDialog.java"]
AddableFeaturesRecyclerViewAdapter -> ProductFeature [label = "line: 87"+"  "+"file: AddProductFeatureDialog.java"]
AppTycoonFragment -> FragmentBackButtonOnClickListener [label = "line: 94"+"  "+"file: AppTycoonFragment.java"]
AssetsFragment -> PremisesAsset [label = "line: 46"+"  "+"file: AssetsFragment.java"]
EmployeesFragment -> ActionSelectDialogBuilder [label = "line: 45"+"  "+"file: EmployeesFragment.java"]
```

Figure A.13: Business Application's Dependency Graph File Script 1



```
EmployeesFragment -> LockedEmployeeViewHolder [label = "line: 95"+"  "+"file: EmployeesFragment.java"]
EmployeesFragment -> EmployeeType [label = "line: 111"+"  "+"file: EmployeesFragment.java"]
EmployeesFragment -> LockedEmployeeViewHolder [label = "line: 114"+"  "+"file: EmployeesFragment.java"]
EmployeesFragment -> UnlockedEmployeeViewHolder [label = "line: 119"+"  "+"file: EmployeesFragment.java"]
EmployeesFragment -> EmployeeType [label = "line: 135"+"  "+"file: EmployeesFragment.java"]
EmployeeRecyclerViewAdapter -> UnlockedEmployeeViewHolder [label = "line: 82"+"  "+"file: EmployeesFragment.java"]
EmployeeRecyclerViewAdapter -> LockedEmployeeViewHolder [label = "line: 95"+"  "+"file: EmployeesFragment.java"]
EmployeeRecyclerViewAdapter -> EmployeeType [label = "line: 111"+"  "+"file: EmployeesFragment.java"]
EmployeeRecyclerViewAdapter -> LockedEmployeeViewHolder [label = "line: 114"+"  "+"file: EmployeesFragment.java"]
EmployeeRecyclerViewAdapter -> UnlockedEmployeeViewHolder [label = "line: 119"+"  "+"file: EmployeesFragment.java"]
EmployeeRecyclerViewAdapter -> EmployeeType [label = "line: 135"+"  "+"file: EmployeesFragment.java"]
NewProductFragment -> ProductType [label = "line: 65"+"  "+"file: NewProductFragment.java"]
NewProductFragment -> ViewHolder [label = "line: 205"+"  "+"file: NewProductFragment.java"]
NewProductFragment -> ProductFeature [label = "line: 216"+"  "+"file: NewProductFragment.java"]
ProductFeaturesRecyclerViewAdapter -> ViewHolder [label = "line: 205"+"  "+"file: NewProductFragment.java"]
ProductFeaturesRecyclerViewAdapter -> ProductFeature [label = "line: 216"+"  "+"file: NewProductFragment.java"]
NewProductReleaseFragment -> ViewHolder [label = "line: 150"+"  "+"file: NewProductReleaseFragment.java"]
NewProductReleaseFragment -> ProductFeature [label = "line: 195"+"  "+"file: NewProductReleaseFragment.java"]
FeaturesRecyclerViewAdapter -> ViewHolder [label = "line: 150"+"  "+"file: NewProductReleaseFragment.java"]
FeaturesRecyclerViewAdapter -> ProductFeature [label = "line: 195"+"  "+"file: NewProductReleaseFragment.java"]
NewProjectFragment -> ContractingProject [label = "line: 76"+"  "+"file: NewProjectFragment.java"]
NewProjectFragment -> ViewHolder [label = "line: 128"+"  "+"file: NewProjectFragment.java"]
ProjectTasksRecyclerViewAdapter -> ViewHolder [label = "line: 128"+"  "+"file: NewProjectFragment.java"]
PremisesFragment -> Company [label = "line: 52"+"  "+"file: PremisesFragment.java"]
PremisesFragment -> PremisesAsset [label = "line: 53"+"  "+"file: PremisesFragment.java"]
PremisesFragment -> NewPremisesRecyclerViewAdapter [label = "line: 69"+"  "+"file: PremisesFragment.java"]
PremisesFragment -> ViewHolder [label = "line: 85"+"  "+"file: PremisesFragment.java"]
PremisesFragment -> PremisesAsset [label = "line: 97"+"  "+"file: PremisesFragment.java"]
PremisesFragment -> AppTycoonAlertDialog [label = "line: 139"+"  "+"file: PremisesFragment.java"]
NewPremisesRecyclerViewAdapter -> ViewHolder [label = "line: 85"+"  "+"file: PremisesFragment.java"]
NewPremisesRecyclerViewAdapter -> PremisesAsset [label = "line: 97"+"  "+"file: PremisesFragment.java"]
NewPremisesRecyclerViewAdapter -> AppTycoonAlertDialog [label = "line: 139"+"  "+"file: PremisesFragment.java"]
ViewHolder -> AppTycoonAlertDialog [label = "line: 139"+"  "+"file: PremisesFragment.java"]
ProductStatsFragment -> Company [label = "line: 36"+"  "+"file: ProductStatsFragment.java"]
ProductsFragment -> ProductRecyclerViewAdapter [label = "line: 47"+"  "+"file: ProductsFragment.java"]
ProductsFragment -> ActionSelectDialogBuilder [label = "line: 61"+"  "+"file: ProductsFragment.java"]
ProductsFragment -> ProductType [label = "line: 63"+"  "+"file: ProductsFragment.java"]
ProductsFragment -> ViewHolder [label = "line: 96"+"  "+"file: ProductsFragment.java"]
ProductsFragment -> Product [label = "line: 113"+"  "+"file: ProductsFragment.java"]
ProductsFragment -> Project [label = "line: 129"+"  "+"file: ProductsFragment.java"]
ProductRecyclerViewAdapter -> ViewHolder [label = "line: 96"+"  "+"file: ProductsFragment.java"]
ProductRecyclerViewAdapter -> Product [label = "line: 113"+"  "+"file: ProductsFragment.java"]
ProductRecyclerViewAdapter -> Project [label = "line: 129"+"  "+"file: ProductsFragment.java"]
ProjectsFragment -> ActionSelectDialogBuilder [label = "line: 91"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> AppTycoonDialog [label = "line: 97"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> GameState [label = "line: 110"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ProjectSlot [label = "line: 111"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ProdDevProjectsRecyclerViewAdapter [label = "line: 113"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ViewHolder [label = "line: 149"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ProjectSlot [label = "line: 150"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ViewHolder [label = "line: 158"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ProjectSlot [label = "line: 181"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ActionSelectDialogBuilder [label = "line: 207"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> Project [label = "line: 211"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> Project [label = "line: 281"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ContractingProject [label = "line: 297"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ProductDevelopmentProject [label = "line: 325"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> Product [label = "line: 405"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ProductDevelopmentProject [label = "line: 406"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ViewHolder [label = "line: 419"+"  "+"file: ProjectsFragment.java"]
ProjectsFragment -> ProductDevelopmentProject [label = "line: 430"+"  "+"file: ProjectsFragment.java"]
ProjectsRecyclerViewAdapter -> ViewHolder [label = "line: 149"+"  "+"file: ProjectsFragment.java"]
ProjectsRecyclerViewAdapter -> ProjectSlot [label = "line: 150"+"  "+"file: ProjectsFragment.java"]
ProjectsRecyclerViewAdapter -> ViewHolder [label = "line: 158"+"  "+"file: ProjectsFragment.java"]
ProjectsRecyclerViewAdapter -> ProjectSlot [label = "line: 181"+"  "+"file: ProjectsFragment.java"]
ProjectsRecyclerViewAdapter -> ActionSelectDialogBuilder [label = "line: 207"+"  "+"file: ProjectsFragment.java"]
ProjectsRecyclerViewAdapter -> Project [label = "line: 211"+"  "+"file: ProjectsFragment.java"]
ProjectsRecyclerViewAdapter -> Project [label = "line: 281"+"  "+"file: ProjectsFragment.java"]
ProjectsRecyclerViewAdapter -> ContractingProject [label = "line: 297"+"  "+"file: ProjectsFragment.java"]
```

Figure A.14: Business Application’s Dependency Graph File Script 2

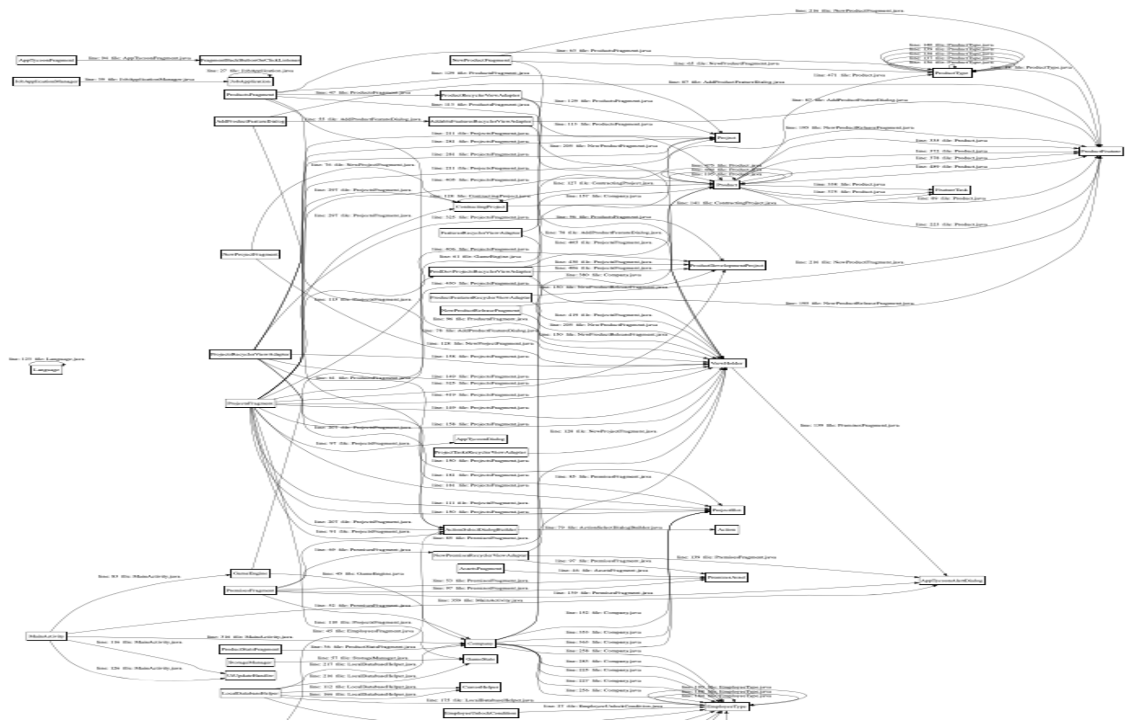


Figure A.15: Business Application’s Visual Dependency Graph 1

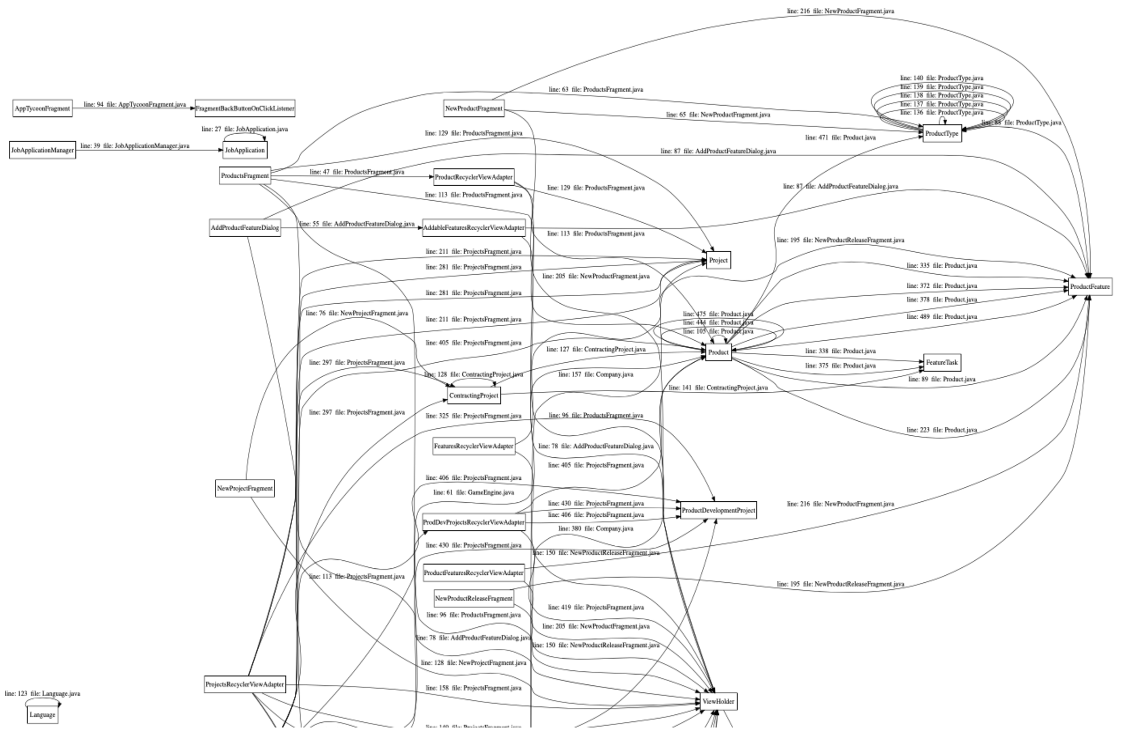


Figure A.16: Business Application's Visual Dependency Graph 2

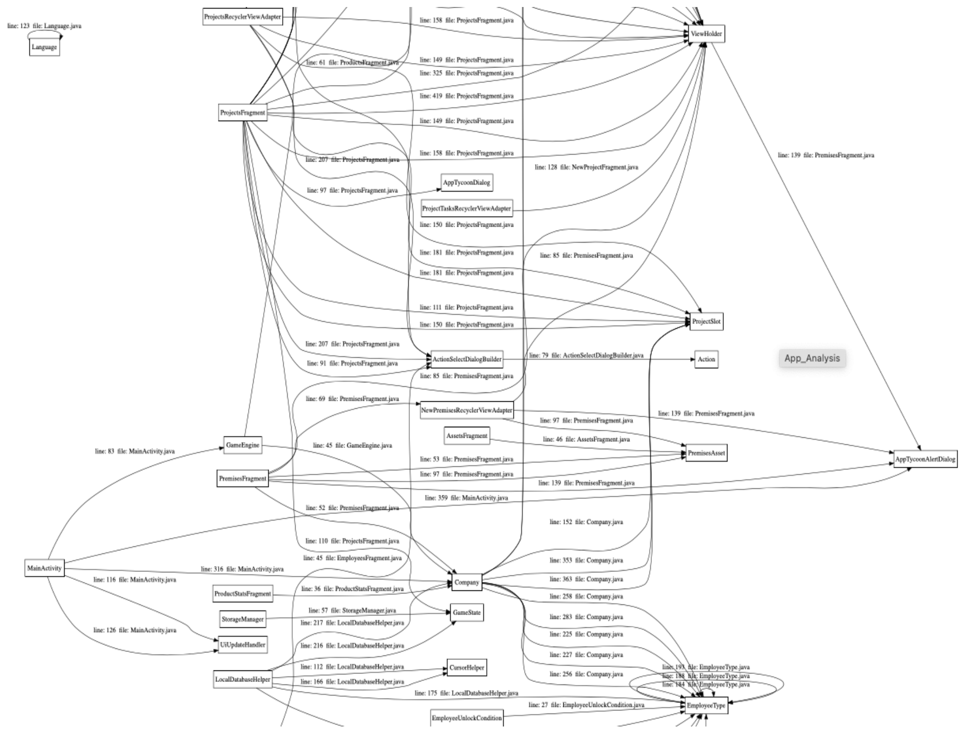


Figure A.17: Business Application's Visual Dependency Graph 3

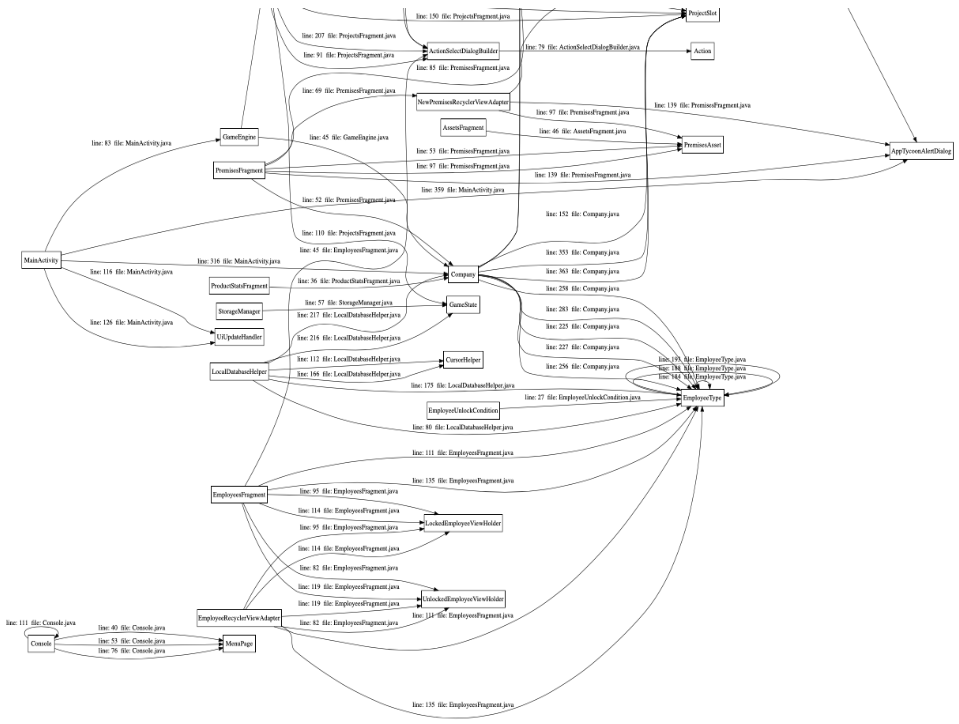
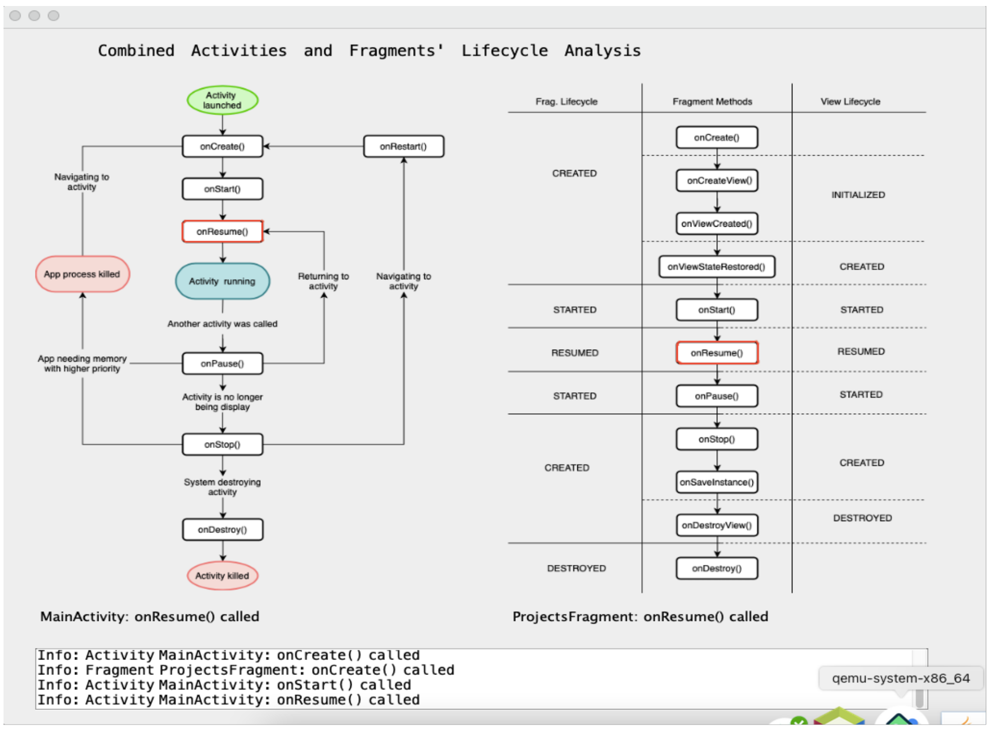


Figure A.18: Business Application’s Visual Dependency Graph 4

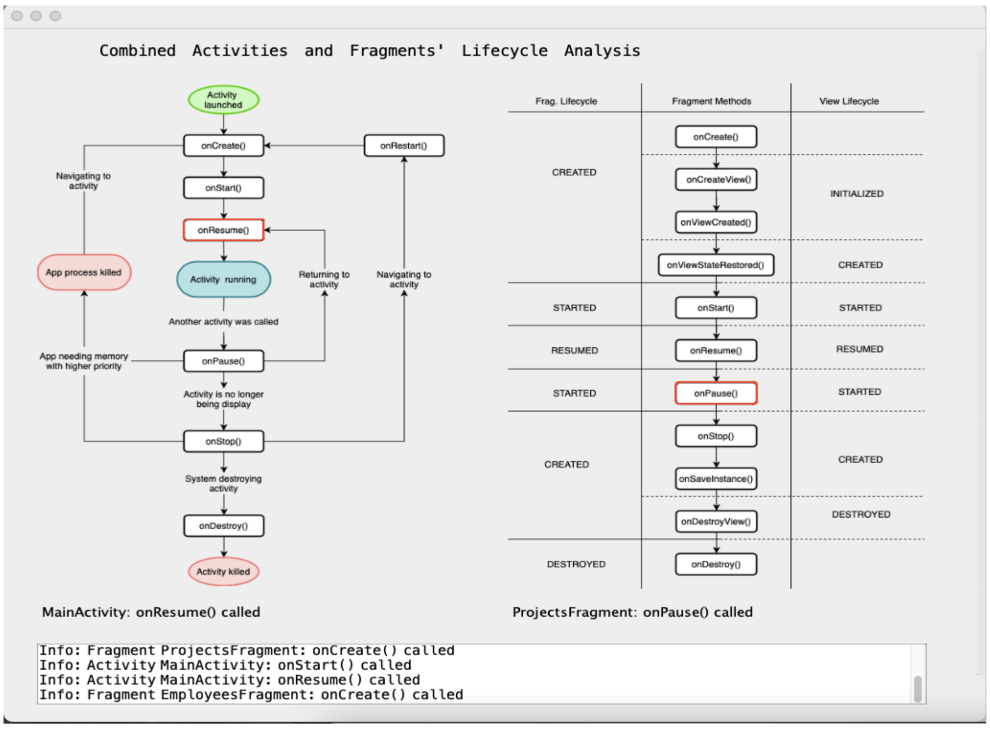

Combined Activities and Fragments' Lifecycle Analysis

MainActivity: onResume() called

ProjectsFragment: onResume() called

```
Info: Activity MainActivity: onCreate() called
Info: Fragment ProjectsFragment: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
```

Figure A.19: Business Application: Desktop App Displayed Method 1

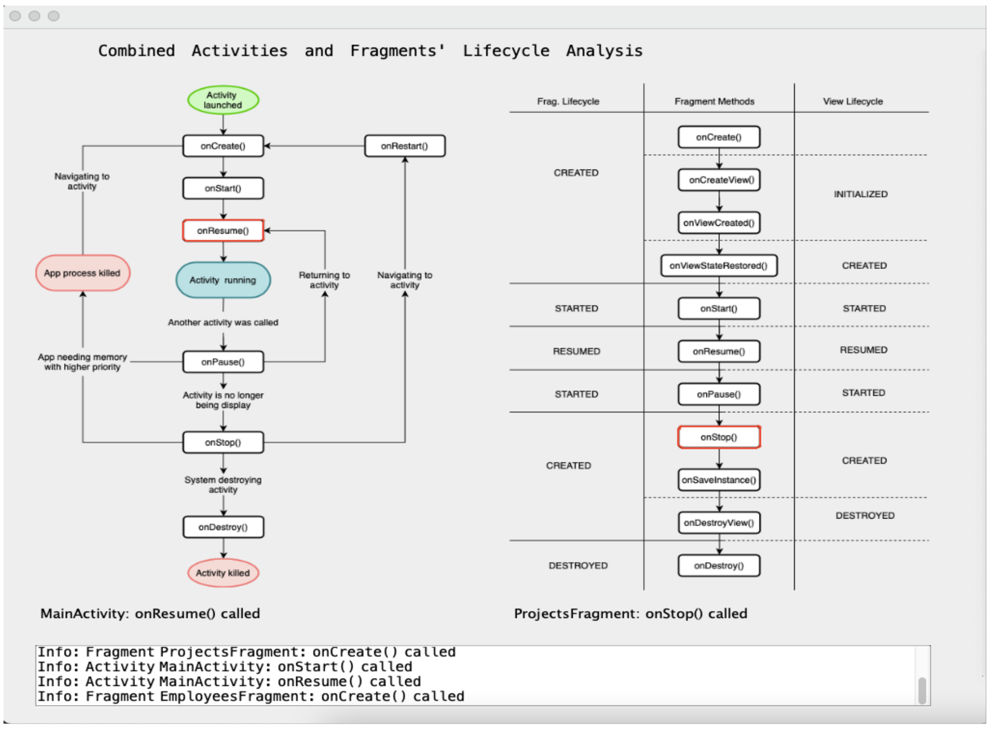


MainActivity: onResume() called

ProjectsFragment: onPause() called

```
Info: Fragment ProjectsFragment: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Fragment EmployeesFragment: onCreate() called
```

Figure A.20: Business Application: Desktop App Displayed Method 2

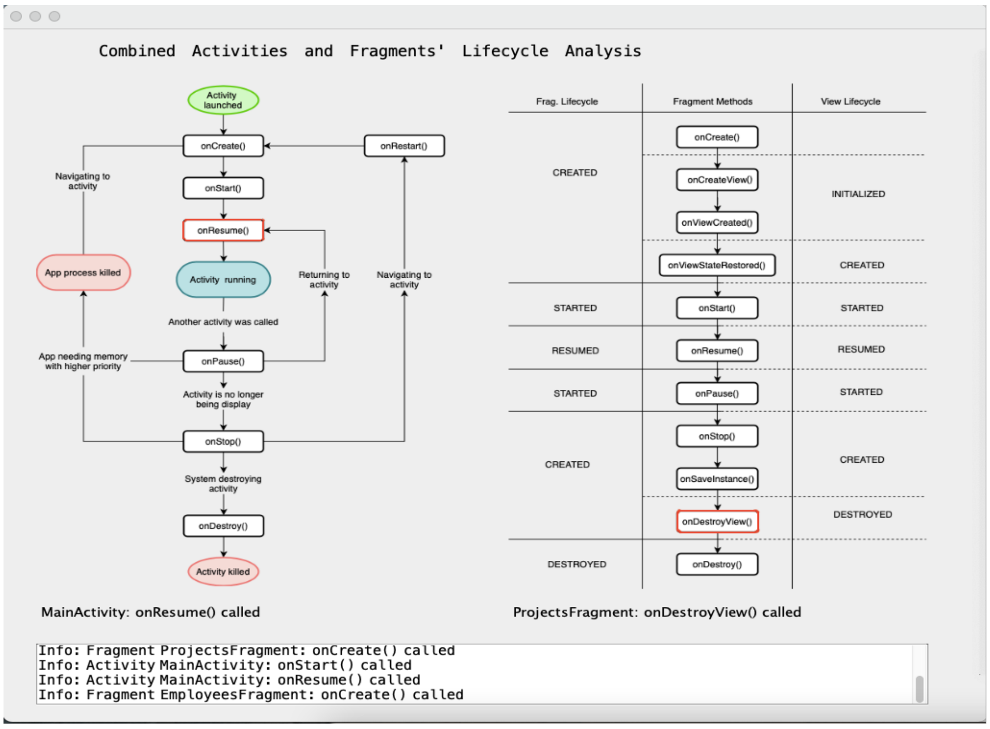


MainActivity: onResume() called

ProjectsFragment: onStop() called

```
Info: Fragment ProjectsFragment: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Fragment EmployeesFragment: onCreate() called
```

Figure A.21: Business Application: Desktop App Displayed Method 3

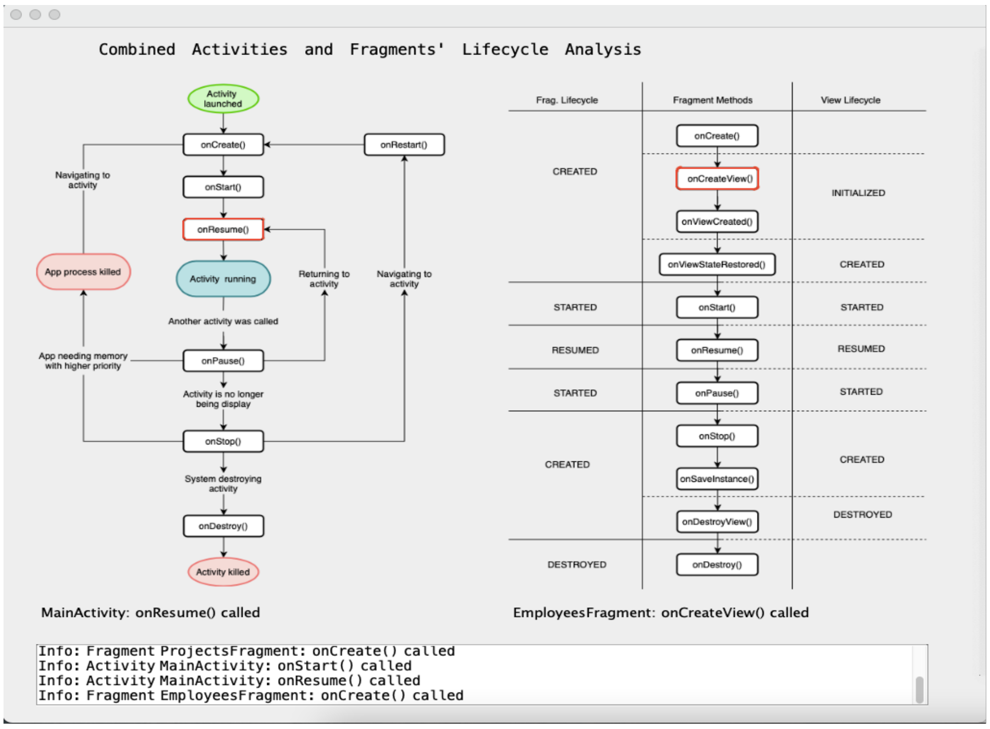

Combined Activities and Fragments' Lifecycle Analysis

MainActivity: onResume() called

ProjectsFragment: onDestroyView() called

```
Info: Fragment ProjectsFragment: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Fragment EmployeesFragment: onCreate() called
```

Figure A.22: Business Application: Desktop App Displayed Method 4

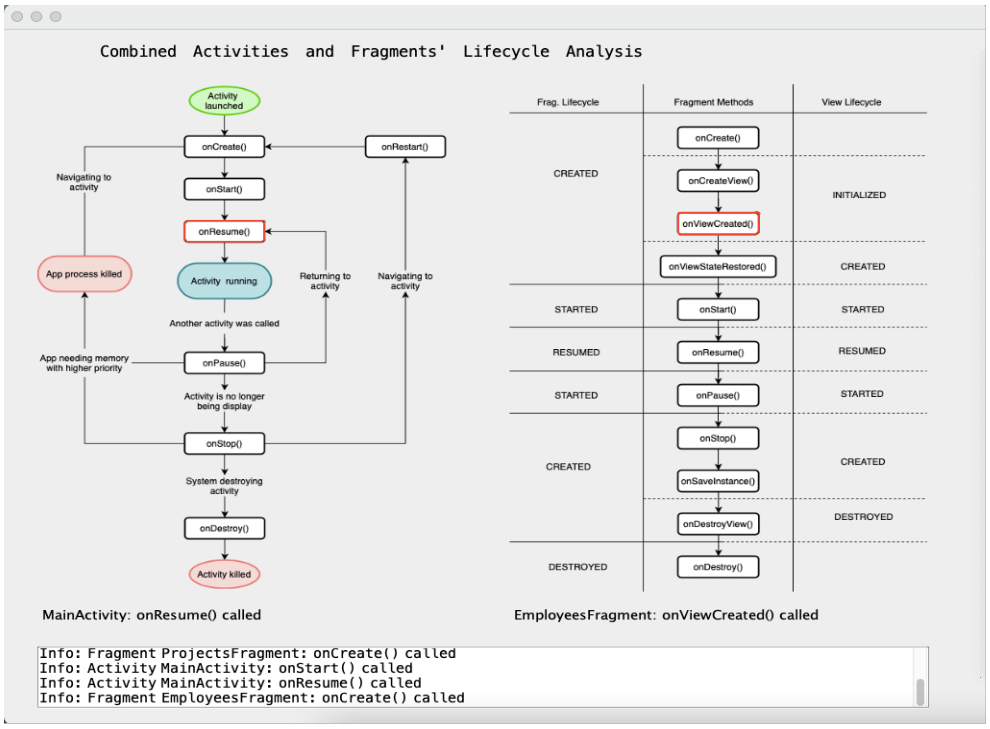

Combined Activities and Fragments' Lifecycle Analysis

MainActivity: onResume() called

EmployeesFragment: onCreateView() called

```
Info: Fragment ProjectsFragment: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Fragment EmployeesFragment: onCreate() called
```

Figure A.23: Business Application: Desktop App Displayed Method 5

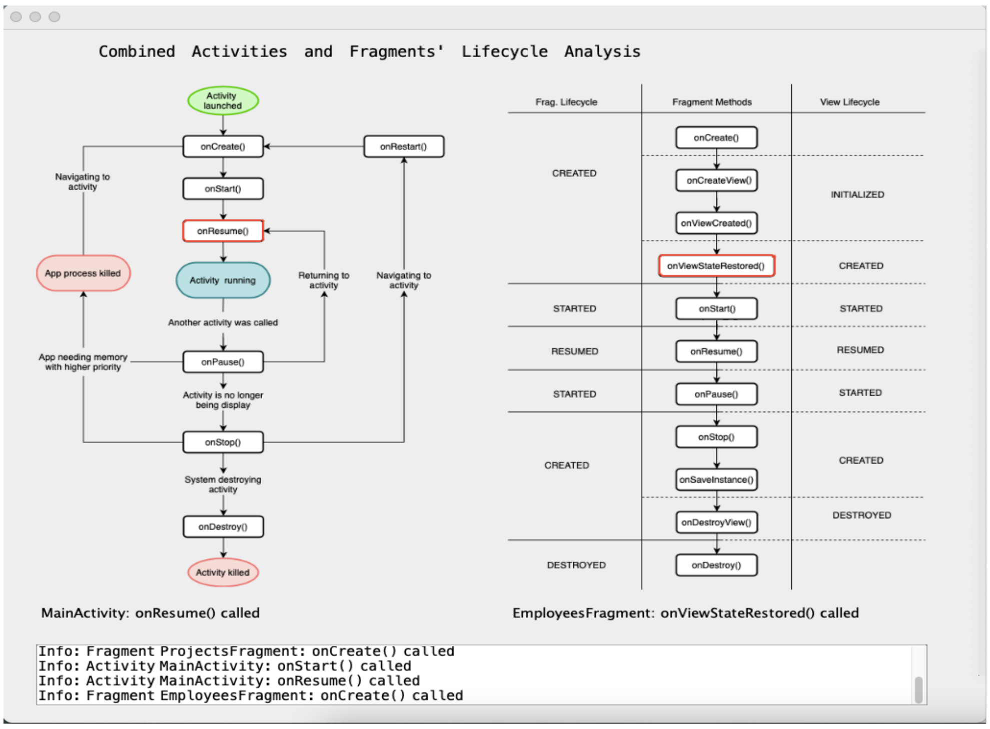


MainActivity: onResume() called

EmployeesFragment: onViewCreated() called

```
Info: Fragment ProjectsFragment: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Fragment EmployeesFragment: onCreate() called
```

Figure A.24: Business Application: Desktop App Displayed Method 6

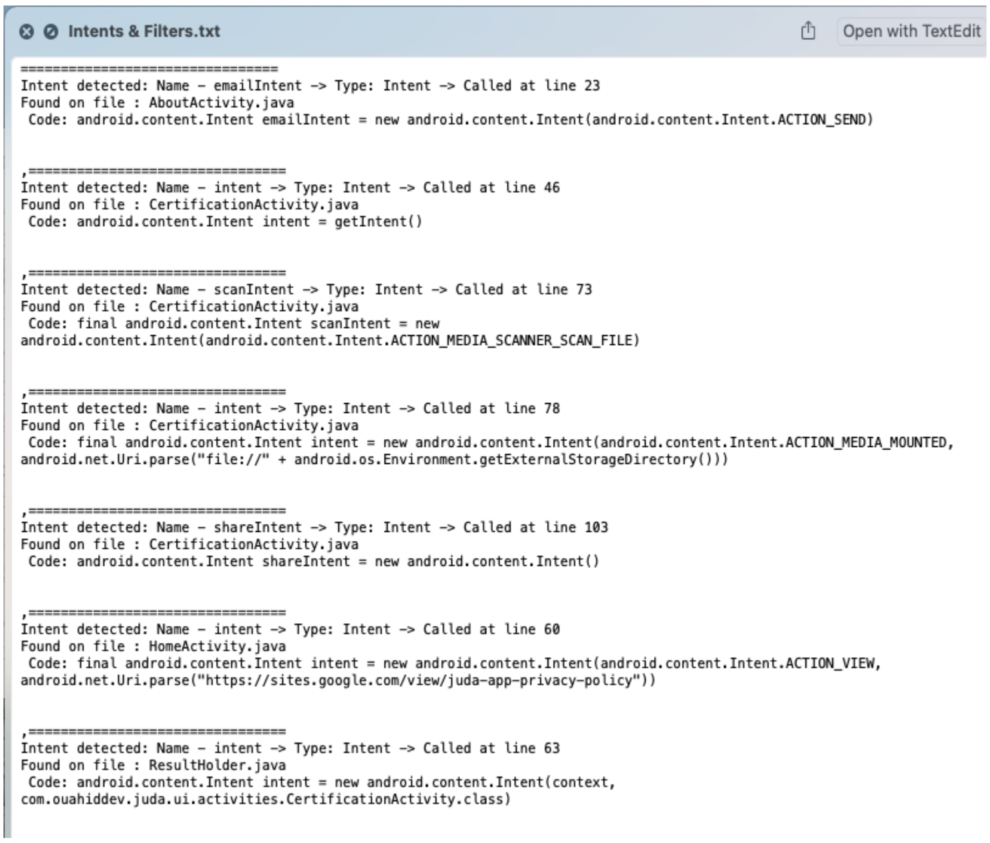


MainActivity: onResume() called

EmployeesFragment: onViewStateRestored() called

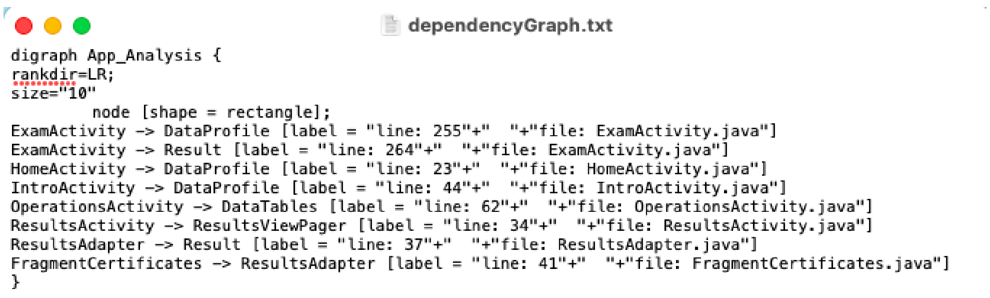
```
Info: Fragment ProjectsFragment: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Fragment EmployeesFragment: onCreate() called
```

Figure A.25: Business Application: Desktop App Displayed Method 7



```
================================
Intent detected: Name - emailIntent -> Type: Intent -> Called at line 23
Found on file : AboutActivity.java
 Code: android.content.Intent emailIntent = new android.content.Intent(android.content.Intent.ACTION_SEND)


,================================
Intent detected: Name - intent -> Type: Intent -> Called at line 46
Found on file : CertificationActivity.java
 Code: android.content.Intent intent = getIntent()


,================================
Intent detected: Name - scanIntent -> Type: Intent -> Called at line 73
Found on file : CertificationActivity.java
 Code: final android.content.Intent scanIntent = new
android.content.Intent(android.content.Intent.ACTION_MEDIA_SCANNER_SCAN_FILE)


,================================
Intent detected: Name - intent -> Type: Intent -> Called at line 78
Found on file : CertificationActivity.java
 Code: final android.content.Intent intent = new android.content.Intent(android.content.Intent.ACTION_MEDIA_MOUNTED,
android.net.Uri.parse("file://" + android.os.Environment.getExternalStorageDirectory()))


,================================
Intent detected: Name - shareIntent -> Type: Intent -> Called at line 103
Found on file : CertificationActivity.java
 Code: android.content.Intent shareIntent = new android.content.Intent()


,================================
Intent detected: Name - intent -> Type: Intent -> Called at line 60
Found on file : HomeActivity.java
 Code: final android.content.Intent intent = new android.content.Intent(android.content.Intent.ACTION_VIEW,
android.net.Uri.parse("https://sites.google.com/view/juda-app-privacy-policy"))


,================================
Intent detected: Name - intent -> Type: Intent -> Called at line 63
Found on file : ResultHolder.java
 Code: android.content.Intent intent = new android.content.Intent(context,
com.ouahiddev.juda.ui.activities.CertificationActivity.class)
```

Figure A.26: Educational Application's Intent File



```
digraph App_Analysis {
rankdir=LR;
size="10"
         node [shape = rectangle];
ExamActivity -> DataProfile [label = "line: 255"+"  "+"file: ExamActivity.java"]
ExamActivity -> Result [label = "line: 264"+"  "+"file: ExamActivity.java"]
HomeActivity -> DataProfile [label = "line: 23"+"  "+"file: HomeActivity.java"]
IntroActivity -> DataProfile [label = "line: 44"+"  "+"file: IntroActivity.java"]
OperationsActivity -> DataTables [label = "line: 62"+"  "+"file: OperationsActivity.java"]
ResultsActivity -> ResultsViewPager [label = "line: 34"+"  "+"file: ResultsActivity.java"]
ResultsAdapter -> Result [label = "line: 37"+"  "+"file: ResultsAdapter.java"]
FragmentCertificates -> ResultsAdapter [label = "line: 41"+"  "+"file: FragmentCertificates.java"]
}
```

Figure A.27: Educational Application's Dependency Graph File Script

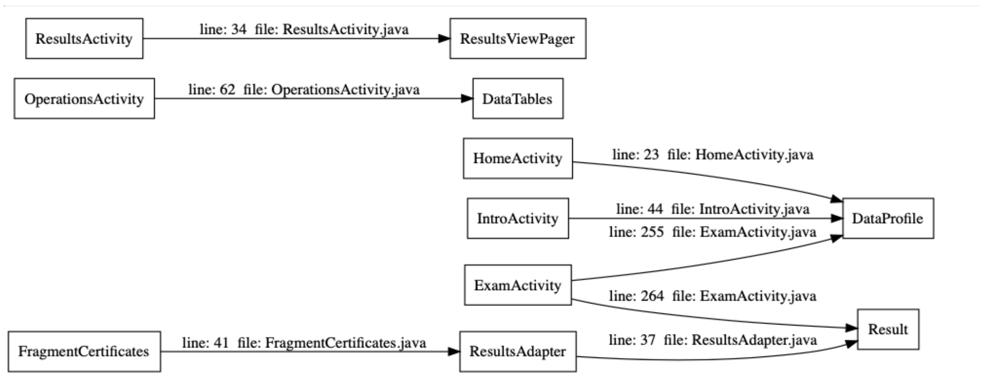
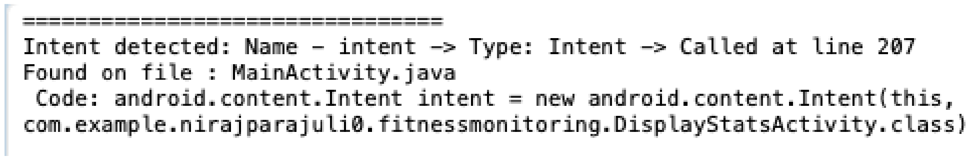
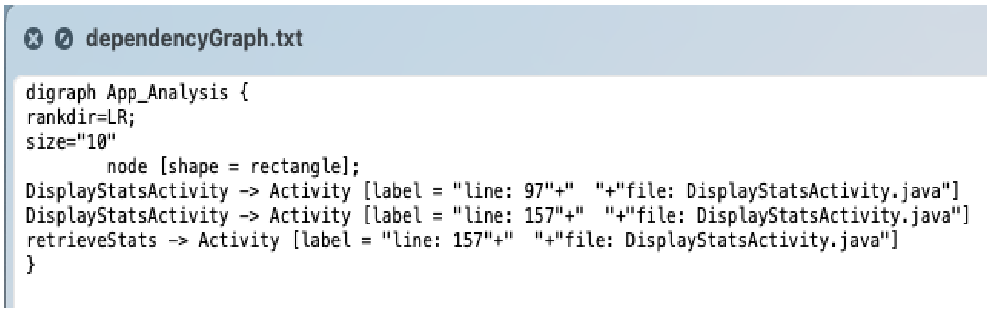

ResultsActivity
line: 34 file: ResultsActivity.java
ResultsViewPager
OperationsActivity
line: 62 file: OperationsActivity.java
DataTables
HomeActivity
line: 23 file: HomeActivity.java
IntroActivity
line: 44 file: IntroActivity.java
line: 255 file: ExamActivity.java
DataProfile
ExamActivity
line: 264 file: ExamActivity.java
Result
FragmentCertificates
line: 41 file: FragmentCertificates.java
ResultsAdapter
line: 37 file: ResultsAdapter.java

Figure A.28: Educational Application's Dependency Graph Visualisation



```
================================
Intent detected: Name - intent -> Type: Intent -> Called at line 207
Found on file : MainActivity.java
 Code: android.content.Intent intent = new android.content.Intent(this,
com.example.nirajparajuli0.fitnessmonitoring.DisplayStatsActivity.class)
```

Figure A.29: Health Application's Intent File



```
digraph App_Analysis {
rankdir=LR;
size="10"
        node [shape = rectangle];
DisplayStatsActivity -> Activity [label = "line: 97"+"  "+"file: DisplayStatsActivity.java"]
DisplayStatsActivity -> Activity [label = "line: 157"+"  "+"file: DisplayStatsActivity.java"]
retrieveStats -> Activity [label = "line: 157"+"  "+"file: DisplayStatsActivity.java"]
}
```

Figure A.30: Health Application's Dependency Graph File Script

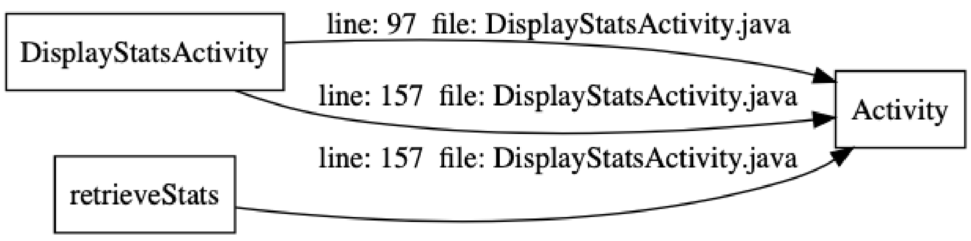
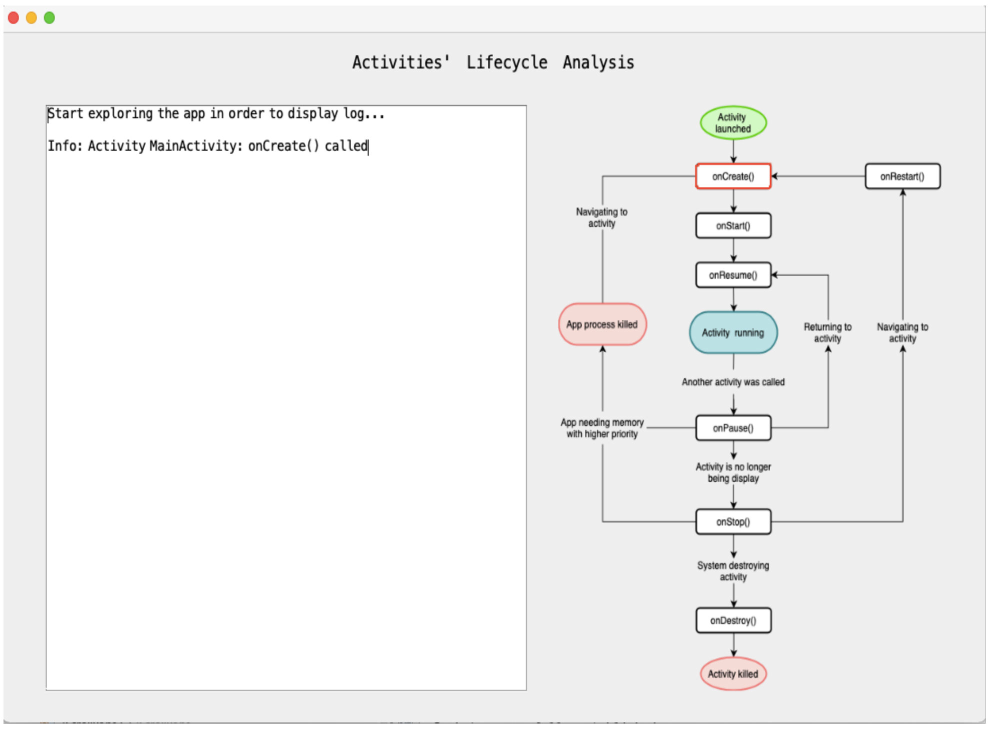

DisplayStatsActivity
line: 97 file: DisplayStatsActivity.java
line: 157 file: DisplayStatsActivity.java
Activity
retrieveStats
line: 157 file: DisplayStatsActivity.java

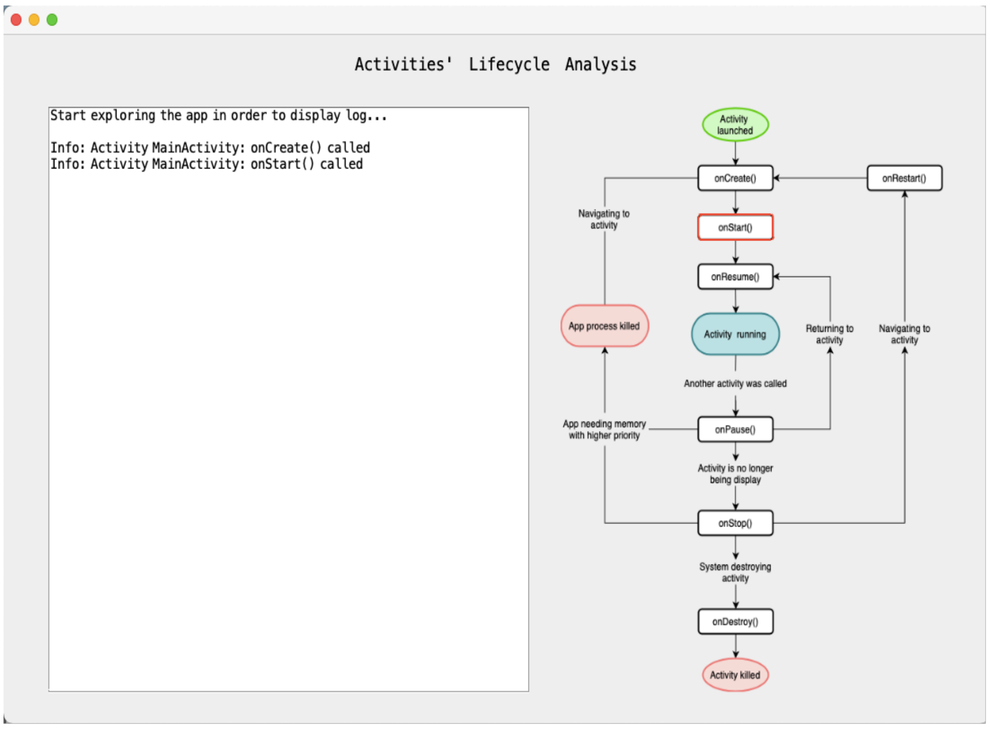
Figure A.31: Health Application's Dependency Graph Visualisation



Activities' Lifecycle Analysis

```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
```

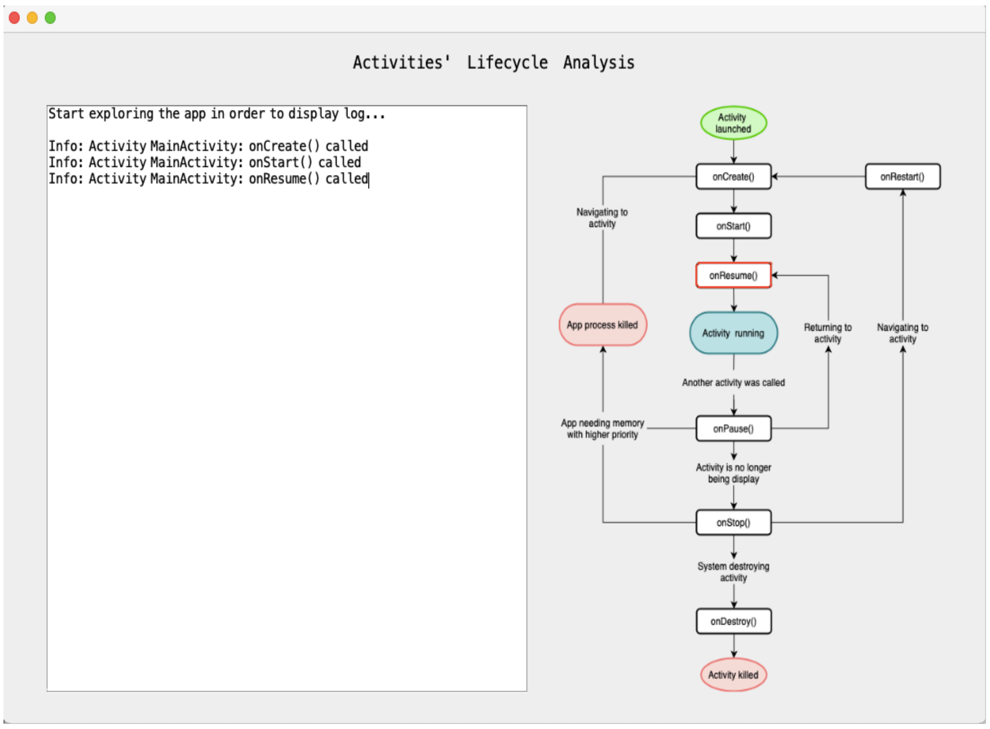
Figure A.32: Health Application's : Desktop App Displayed Method 1



Activities' Lifecycle Analysis

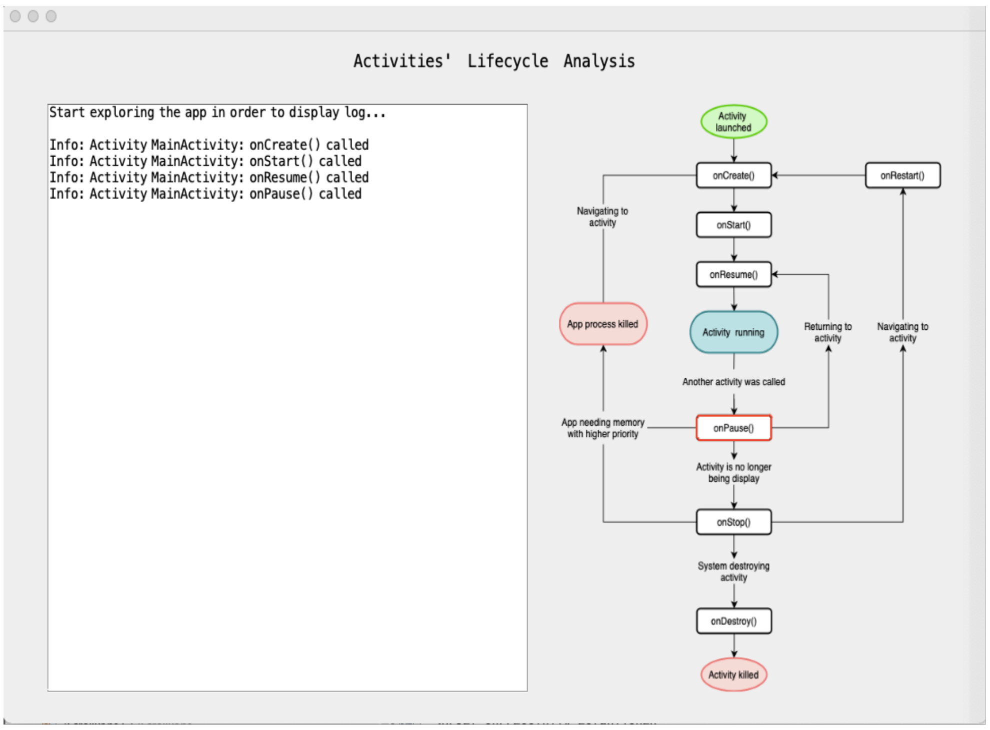
```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Activity MainActivity: onStart() called
```

Figure A.33: Health Application's : Desktop App Displayed Method 2


Activities' Lifecycle Analysis

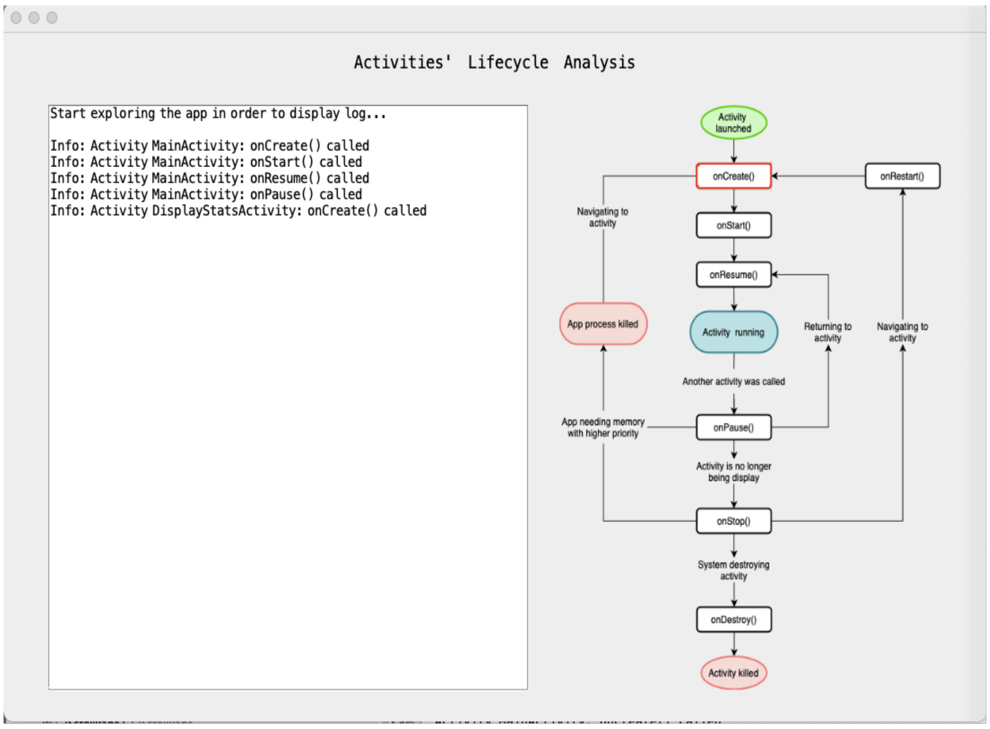
```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
```

Figure A.34: Health Application's : Desktop App Displayed Method 3


Activities' Lifecycle Analysis

```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Activity MainActivity: onPause() called
```

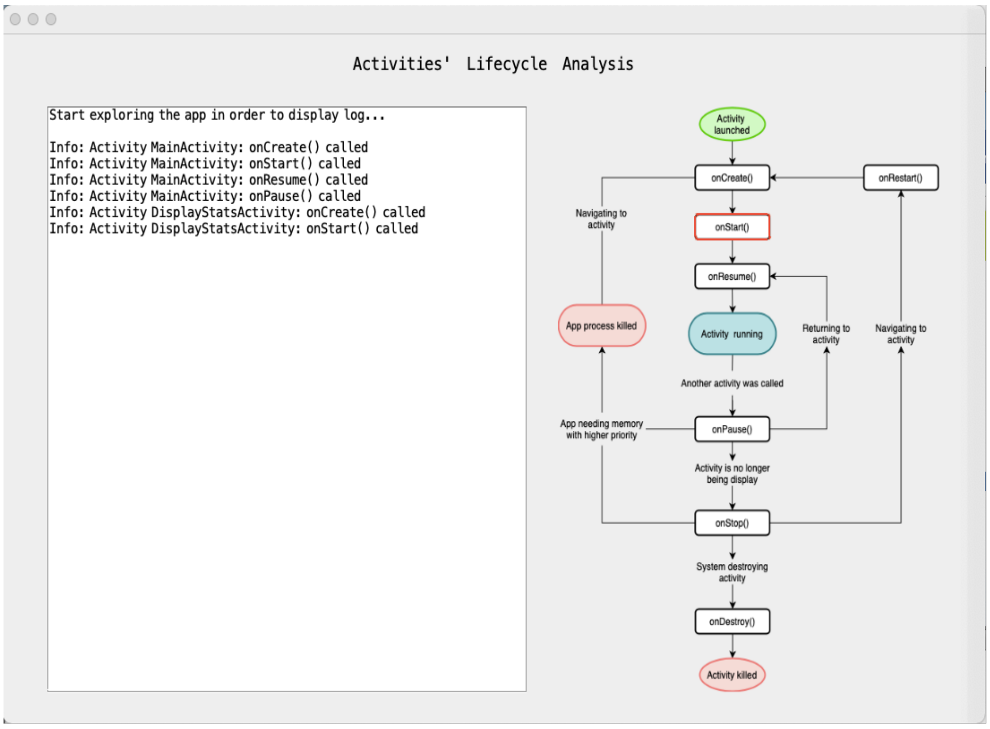
Figure A.35: Health Application's : Desktop App Displayed Method 4



Activities' Lifecycle Analysis

```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Activity MainActivity: onPause() called
Info: Activity DisplayStatsActivity: onCreate() called
```

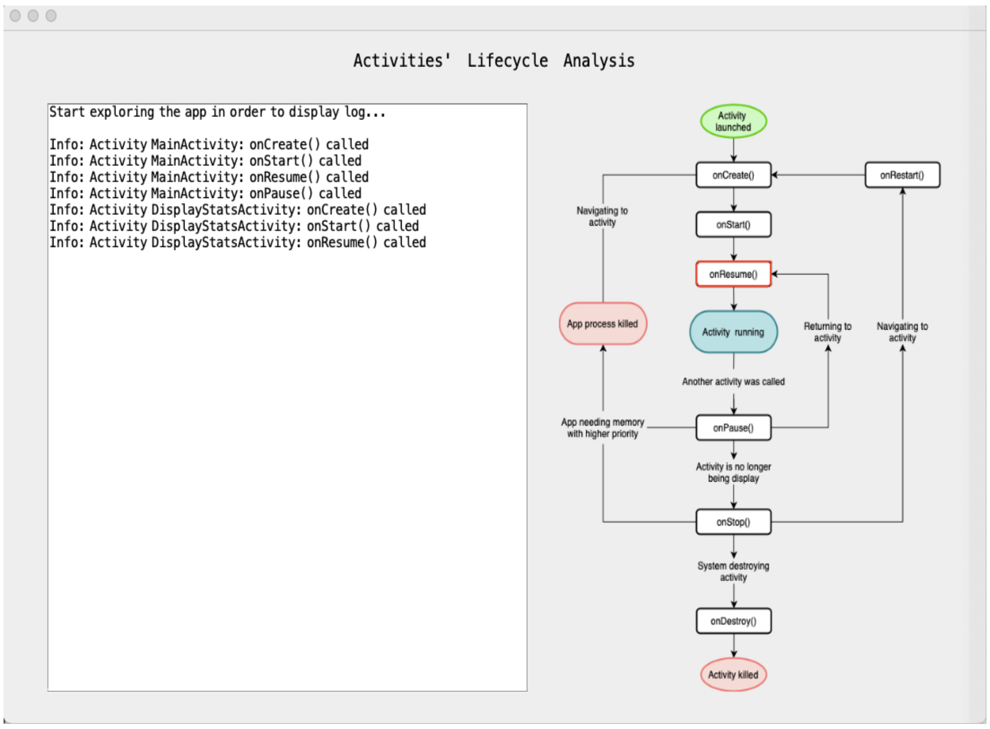
Figure A.36: Health Application's : Desktop App Displayed Method 5



Activities' Lifecycle Analysis

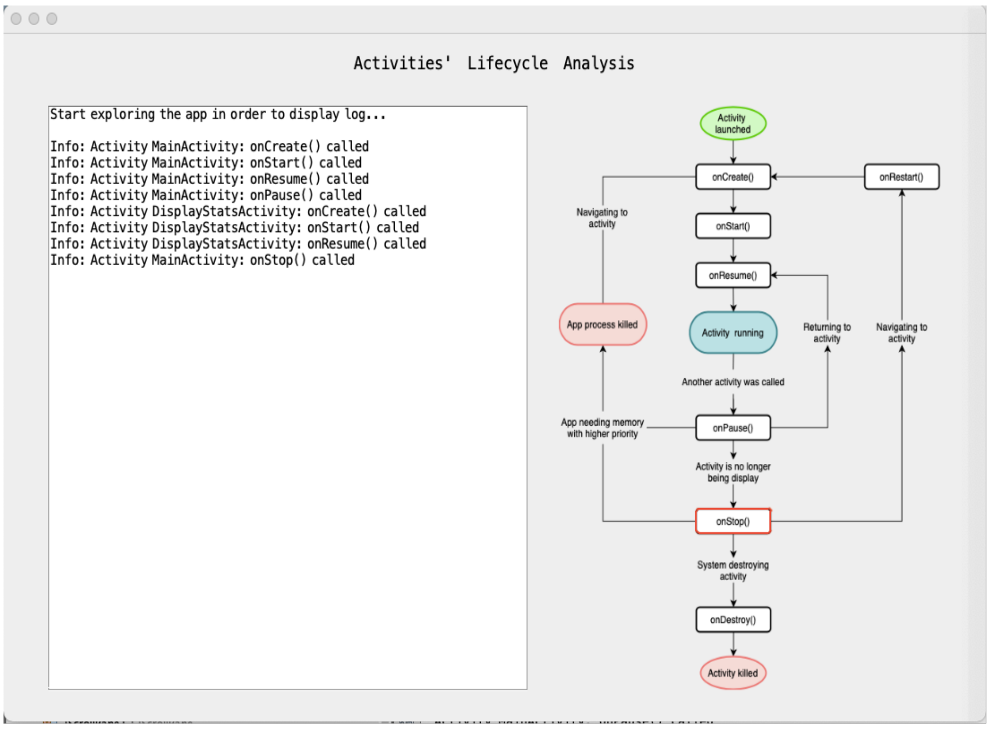
```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Activity MainActivity: onPause() called
Info: Activity DisplayStatsActivity: onCreate() called
Info: Activity DisplayStatsActivity: onStart() called
```

Figure A.37: Health Application's : Desktop App Displayed Method 6


Activities' Lifecycle Analysis

```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Activity MainActivity: onPause() called
Info: Activity DisplayStatsActivity: onCreate() called
Info: Activity DisplayStatsActivity: onStart() called
Info: Activity DisplayStatsActivity: onResume() called
```

Figure A.38: Health Application’s : Desktop App Displayed Method 7


Activities' Lifecycle Analysis

```
Start exploring the app in order to display log...

Info: Activity MainActivity: onCreate() called
Info: Activity MainActivity: onStart() called
Info: Activity MainActivity: onResume() called
Info: Activity MainActivity: onPause() called
Info: Activity DisplayStatsActivity: onCreate() called
Info: Activity DisplayStatsActivity: onStart() called
Info: Activity DisplayStatsActivity: onResume() called
Info: Activity MainActivity: onStop() called
```

Figure A.39: Health Application’s : Desktop App Displayed Method 8