



**Faculty of Engineering and Technology**  
**Master of Software Engineering (SWEN)**

Master Thesis

Assisting Android Lifecycle Development: A Multi-view Approach

المساعدة في تطوير دورة حياة Android: نهج متعدد وجهات الاستخدام

*By*

**Student Name** Timothawous Edward Ghanim

**Student Number** 1175474

**Supervised By** Dr. Samer Zein

*This thesis was submitted in partial fulfillment of the requirements for the Master's Degree in Software Engineering from the Faculty of Graduate Studies at Birzeit University.*

February 4<sup>th</sup>, 2021



---

Assisting Android Lifecycle Development: A Multi-view Approach

---

**Author**

*Timothawous Ghanim*

This thesis was prepared under the supervision of Dr. Samer Zain and has been approved by all members of the examination committee.

**Dr. Samer Zain, Birzeit University**

(Chairman of the Committee)

**Dr. Sobhi Ahmed, Birzeit University**

(Member)

**Dr. Mamoun Nawahdah, Birzeit University**

(Member)

Date Approved: January 30<sup>st</sup>, 2021

## **Declaration of Authorship**

I, Timothawous Ghanim, declare that this thesis titled, “Assisting Android Lifecycle Development: A Multi-view Approach” and the work presented in it are my own.

I confirm that:

- This work was done wholly or mainly while in candidature for a master’s degree at Birzeit University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

*Signed by:*

**Timothawous Ghanim**

*Timothy Ghanem*

---

*Date:*

21/2/2021

---

## **Acknowledgements**

I would like to give my highest acknowledgement and gratitude to God, the Almighty, for all the potentials of health and resources that He provided which helped me carry out this research from inception to publication.

I would also like to give my deepest gratitude to Dr. Samer Zein for all the help and guidance that he provided at every step of the way until this research reached its completion.

## Table of Contents

Declaration of Authorship .....	II
Acknowledgements .....	III
List of Figures.....	VII
List of Listings.....	VIII
List of Tables .....	IX
Acronyms.....	X
Abstract.....	XI
ملخص الدراسة .....	XII
Chapter 1: Introduction.....	1
1.1 Motivation .....	2
1.2 Research Aim .....	3
1.3 Contributions.....	4
1.4 Thesis Structure.....	5
Chapter 2: Background.....	6
Chapter 3: Literature Review .....	11
3.1 Literature Survey Process .....	11
3.1.1 Searched Databases.....	12
3.1.2 Inclusion/Exclusion Criteria .....	12
3.2 Model-based Android App Development .....	13
3.3. Android Activity Lifecycle Handling Analysis .....	18
3.3.1 Security Analysis .....	18
3.3.2 Android App Testing .....	20
3.3.3 Memory Leaks Detection .....	21
3.3.4 Other Approaches .....	21
Chapter 4: Methodology .....	24
4.1 Domain-Specific Languages .....	30
4.1.1 Meta-model for approach DSTL .....	30
4.1.2 Meta-model for approach DSVL .....	31
4.2 Evaluation, Data Collection, and Analysis .....	33

Chapter 5: Implementation .....	35
5.1 Overview .....	39
5.1.1 IntelliJ Plugin Metadata .....	39
5.1.2 IntelliJ Plugin Settings .....	41
5.1.3 IntelliJ PSI.....	43
5.1.4 IntelliJ Indexing Framework.....	44
5.2 Plugin Architecture .....	45
5.2.1 FileEditorManager .....	46
5.2.2 EventHandler .....	46
5.2.3 Activity View Service.....	47
5.2.4 Activity File Analyzer .....	48
5.2.5 Plugin Settings .....	48
5.2.6 Lifecycle Node Factory .....	49
5.2.7 Tool Window .....	49
5.2.8 Notification Service .....	49
Chapter 6: Evaluation .....	51
6.1 Case Study Setup.....	51
6.2 Participants .....	51
6.3 Case Study App.....	52
Chapter 7: Results.....	54
7.1 Demographic Survey.....	54
7.2 Post-Case-Study Questionnaire.....	58
Chapter 8: Discussion.....	62
8.1 Threats to Validity.....	64
8.1.1 Internal Validity.....	64
8.1.2 External Validity.....	64
Chapter 9: Conclusion .....	66
References .....	68
Appendices .....	74
Appendix A - Demographic Survey.....	74

Appendix B - Post- Case Study Questionnaire ..... 74

## List of Figures

Figure 1: Literature Review Method .....	12
Figure 2: Approach Architecture.....	25
Figure 3: Code Analysis Algorithm .....	26
Figure 4: DSTL to DSVL Converter .....	28
Figure 5: Domain-Specific Textual Language. ....	30
Figure 6: The DSVL implemented in Android Studio plugin.....	35
Figure 7: The presented Activity lifecycle view allows expanding and collapsing the tree nodes. ....	36
Figure 8: The full view of the Activity lifecycle when fully expanded. ....	38
Figure 9: The developer can navigate to an implemented callback method. ....	38
Figure 10: The developer can add unimplemented callback method.....	38
Figure 11: The developer can navigate to the line in the source allocation where the resource allocation is located.....	38
Figure 12: The developer can navigate to the line in the source code where the resource release is located. ....	38
Figure 13: IntelliJ Platform PSI Model. ....	44
Figure 14: IntelliJ Platform Plugin Architecture.....	46
Figure 15: Android Studio Plugin Settings. ....	49
Figure 16: The number of participants in each years of experience range that successfully completed the case study. ....	55
Figure 17: The number of participants per the number of projects contributed to that the case study included.....	55
Figure 18: The number of participants per their Activity lifecycle development familiarity that the case study included.....	56
Figure 19: Post-Case-Study Questionnaire Results.....	58



## List of Listings

Listing 1: Activity in AndroidManifest.xml.....	6
Listing 2: Sample Activity Class Implementation.....	7
Listing 3: IntelliJ Plugin XML File .....	40
Listing 4: Sample Configurable Implementation. ....	42
Listing 5: Sample PersistentStateComponent. ....	43
Listing 6: A snippet how the IntelliJ Platform Indexing Framework is used. ....	45
Listing 7: FileEditorManager Listener. ....	47

## List of Tables

Table 1: Building Blocks for the DSVL.....	32
Table 2: Connections between Nodes of the DSVL.....	33
Table 3: Participants Groups .....	57

## Acronyms

**AOCA** Activity-oriented context-aware applications

**APK** Android Package

**BPMN** Business Process Model and Notation

**DSL** Domain-Specific Language

**DSTL** Domain-Specific Textual Language

**DSVL** Domain-Specific Visual Language

**FSM** Finite State Machine

**GUI** Graphical User Interface

**IDE** Integrated Development Environment

**IFML** Interaction Flow Modeling Language

**JSON** JavaScript Object Notation

**MDD** Model-driven development

**MVC** Model-View-Controller

**POJO** Plain-Old Java Object

**UML** Unified Modeling Language

**XMI** XML Metadata Interchange

**XML** Extensible Markup Language

## **Abstract**

In this study, we present an approach and a framework that provides a visual view for the Activity lifecycle state transitions during implementation. The approach follows model-based development utilizing a DSL (Domain Specific Visual Language) and is implemented as a proof-of-concept Android Studio plugin. We evaluated our approach through a case study on real Android developers. The results show that our approach can be useful and effective in assisting Android developers.

## ملخص الدراسة

في هذه الدراسة، لقد قدمنا منهاج وإطار عمل يوفر لمطوري برامج الأندرويد عرضاً مرئياً غير النص البرمجي الذي يكتبونه لدورة حياة برنامج الأندرويد باستخدام لغة خاصة بتمثيل دورة الحياة للبرنامج. وأيضاً قمنا ببناء تطبيق تجريبي لهذا المنهاج وإطار العمل وتقييمه من خلال استخدامه من قبل مبرمجين أندرويد حقيقيين. النتائج أظهرت لنا أن هذا المنهاج مفيد وفعال لمساعدة مبرمجي الأندرويد.

## **Chapter 1: Introduction**

Two primary distinguishing features of Android over any other mobile development platform are that it is: (1) open source and hence free to download and use and (2) built using the Java programming language; one of the most popular and globally used programming languages. The fact that Android is open source has made it the primary choice for smartphone manufacturers and as a result has been adopted in a wide variety of smartphone devices. As a result, it became the primary focus for scientific research targeting the development of mobile applications which has contributed a lot to the identification of design issues and development of tools to assist developers who build applications that target the Android platform [1].

Handling of the Android Activity lifecycle state transitions is implemented in the Activity lifecycle callback methods. This is a very important aspect of any Android app development activity in order to avoid app crashes and consumption of valuable system resources. It has been observed that a large spread of memory leaks in the wild are tied to improper implementation of Activity lifecycle callback methods [2], and that the objects that most frequently contain memory leaks in Android apps were the Activity classes [3]. In addition, it has been observed that Android developers lack the necessary knowledge and awareness of the Android Activity lifecycle model [4]. Activity lifecycle implementation failures could be triggered as easily as a change in the orientation of the device. It has been observed that a change in orientation of the device after performing some interactions with Dropbox version 27.1.2 caused it to suddenly stop working [5].

To the best of our knowledge, none of the current state-of-the-art frameworks and tools provide a multi-view support to assist Android developers during Activity lifecycle implementation.

To that end and in order to aid Android app developers in implementing a better handling of the Activity lifecycle callback methods, we present in this research a model-driven multi-view approach, implemented as an Android Studio plugin which presents the developer with a view that reflects the Activity lifecycle callback methods implemented in each Activity class. The approach utilizes two domain specific languages (DSLs): textual (DSTL) and visual (DSVL). The view presented by our approach is an additional view to the default textual code view that the developer sees.

## **1.1 Motivation**

A user can perform many activities with a smartphone such as switching between apps, change device orientation...etc. As a result of these activities, mobile apps go through different state transitions. In Android, the affected component in the mobile app is called the Activity. An Activity is the main component with which an Android user interacts [6]. Typically, the operating system notifies the app's Activity of all those state transitions in order to allow app developers to maintain an acceptable user experience throughout the app lifecycle. Activity's lifecycle state transitions are typically implemented by using callback methods that are implemented inside the Activity class. Those callback methods are called by the underlying operating system whenever the Activity's state changes [7]. As a result, Android apps are categorized as event-driven applications [8]. A mobile app lifecycle is significantly different from that of a standard Web or Desktop application [9]. For instance, and in the Android case, when a user switches between apps, the foreground app Activity is transitioned to the paused state and the switched-to Activity is transitioned to the resumed state [10]. On the other hand, a desktop app remains a foreground process except that its windows may be unallocated from the screen and a Web application does not even have the paused state.

To analyze Android Activity lifecycle-related issues, there exists a large body of research that presents several categories of approaches such as security analysis [11]–[15], black-box testing [2], [5], lifecycle conformance checking [16]...etc. Such approaches attempt to

model the app's implementation of the Activity lifecycle in each Activity class in the app and then perform different types of analysis to verify different aspects in the Activity lifecycle implementation. Examples on these aspects are: making sure that no sensitive data is leaked from the app [17], checking if an app correctly releases the resources it acquired when it is no longer in the foreground and if the app doesn't purposefully alter its lifecycle for more than what the user expectations are [15].

Model-driven development is an approach for mobile app construction from different types of models. In general, these approaches provide an initial boost in development time and hence developer's productivity. Several types of model-driven development approaches are presented in the literature such as visual-driven [18]–[20], text-driven approaches [21], or a combination of both [22]. Visual-driven approaches present the developer - or app modeler - with a high-level view where the pages, flow and data are all represented at a high-level abstraction. The model typically conforms to a certain type of a DSVL. Then, through a series of model transformations, the final source code for the mobile app is produced. Similarly, text-driven approaches start the modeling process using a high-level DSTL where the developer (or modeler) describes the mobile app using a textual code that conforms to the DSTL. The code is then transformed into the final app source code. Model-driven development could also contain a combination of both the high-level models (i.e. visual (DSVL) and textual (DSTL)) where each model type allows specifying different aspects of the end mobile app.

None of the existing approaches and tools presented in the literature has a dedicated support to aid Android developers in writing robust and reliable Android Activity lifecycle handling.

## **1.2 Research Aim**

More formally, this research aims to answer the following questions:



- RQ1: How can model-based solutions be applied to assist Android Activity lifecycle development?
- RQ2: To what extent can a multi-view solution affect Android developers implementing Activity lifecycle callback methods?

It is worth noting that the first question is intended to investigate using an approach that utilizes models to represent an Android Activity lifecycle implementation where the output model can be processed by another logic. The other logic is covered by the second question which aims to investigate using a rendering engine that is capable of drawing Activity lifecycle models that conform to the model definition presented by discoveries of the first question. The second question goes even further into investigating the effect of such a rendering logic on the development experience of Android apps. As a result, an empirical evaluation is needed where Android app developers interact with the resulting view and report on their satisfaction. The two questions present a good foundation for a loosely coupled architecture where the rendering logic depends only on an instance of the Activity lifecycle visual model without taking into consideration how that model was generated. As a result, the model could be generated by one tool and rendered by another. Additionally, the question investigates the concept of a “multi-view” approach. The reason for calling it “multi-view” is because developers typically see one view of their Activity lifecycle implementation which is the code that they write. However, another model presented to the developer in a view gives rise to a second view and hence a “multi-view” experience is presented to the developer. Finally, the entire focus of the research questions is on the development of the Activity lifecycle handling and not any other aspect of Android app development.

### **1.3 Contributions**

In addition to the main contribution which is aiding Android app developers better implement the Android Activity lifecycle handling, this research contains the following contributions as well:

1. A Domain-Specific Textual Language (DSTL) that can be used to represent an implementation of the Android Activity lifecycle.
2. A visual domain-specific language (DSVL) to represent - in visual terms - an implementation of the Android Activity lifecycle.
3. A tool, implemented as an Android Studio plugin that implements an Android Activity class source code converter to DSTL code that conforms to the presented DSTL, a DSTL-to-DSVL converter that converts an instance of the DSTL code to another instance that conforms to the DSVL presented. The tool contains a rendering engine that is able to represent any instance of the DSVL.
4. Another contribution is that the implemented tool is designed with extensibility in mind such that it can be easily extended to allow for different customizations.
5. The research results of this thesis have been recognized by the 4<sup>th</sup> International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT 2020) and were published to the IEEE Xplore Digital Library [23].

## **1.4 Thesis Structure**

This report is structured as follows: the Background chapter addresses the necessary technical background to understand the technical side of a proper implementation of the Android Activity lifecycle callback methods, the Literature Review chapter discusses a survey of the current state-of-the-art approaches that are used to build Android apps and tools that test and analyze an Android app's lifecycle implementation. In the Methodology chapter we discuss the proposed research methodology that was used to answer the previous research questions. The Implementation chapter goes over the technical implementation of the presented approach as an Android Studio plugin. In the Evaluation chapter we discuss the evaluation of the presented approach and in the Results chapter we present the results that represent the level of satisfaction of Android developers who used the Android Studio plugin. The Discussion chapter discusses the presented results. Finally, in the Conclusion chapter we present a summary and potential future work to extend this work.

## Chapter 2: Background

Android applications are event-driven applications. This means that the app receives event notifications about changes to its state from the underlying operating system. The Activity class in Android is the primary thing with which the user interacts [7]. Each Activity class has its own lifecycle that is implemented by overriding and implementing different callback methods from the Activity class. Activities in Android are managed using stacks. When a new Activity is started, it is placed on the top of the stack and becomes the running Activity. The previous Activity remains below it and will not come to the foreground until the Activity on the top of the stack exits.

Activity classes in Android are registered in the **AndroidManifest.xml** file which is an XML file, with precisely this name, that exists at the root of the Android app project. Among other things, the manifest file contains the Activity classes that are in the Android project with the basic properties of each Activity including the fully-qualified name of the class where the Activity is implemented [24]. Listing 1 illustrates a sample portion of an **AndroidManifest.xml** file with an Activity class defined in it.

```
<manifest ... >
    <application ... >
        <activity android:name="com.myapp.MainActivity" ... >
            </activity>
        </application>
    </manifest>
```

Listing 1: Activity in AndroidManifest.xml

In the AndroidManifest.xml file, the Activity classes are registered using the **activity** XML tag and the fully qualified name of the class that contains the implementation of the Activity is defined inside an XML attribute called **name** from the **android** namespace. From Listing 1, it can be seen that looking up the class **com.myapp.MainActivity** in the Android project

will get us the implementation of this Activity. Listing 2 illustrates a sample implementation of an Activity class.

```
package com.myapp;

import android.os.Bundle;
import android.app.Activity;

public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Override
    protected void onStart() {
        super.onStart();
    }

    @Override
    protected void onResume() {
        super.onResume();
    }
}
```

Listing 2: Sample Activity Class Implementation

The methods defined in the class shown in Listing 2 are part of a collection of methods that the Activity class uses to listen to notifications from the underlying operating system about changes in its state. The following is the list of the methods that represent handlers to changes in the Activity state [25]:

1. **onCreate:** this method must be implemented by the Activity class which is used to notify the Activity class when the operating system creates the Activity. Invoking this method means that the Activity has entered the *Created* state.

2. **onStart:** this method is invoked by the system when the Activity enters the *Started* state where the app prepares the Activity to be pushed onto the foreground and become visible to the user and ready for interaction.
3. **onResume:** this method is invoked when the Activity enters the *Resumed* state. In this state, the Activity is visible to the user and the user interacts with it. The app and the Activity stay in this state until something causes the system to take the focus from the Activity to another Activity such as when the user receives a phone call. In such a case, the Activity enters the *Paused* state and the `onPaused` method is invoked.
4. **onPaused:** invoking this method means that focus has been taken away from the Activity as the system is switching to another Activity as per the user's interaction with the system or a disruptive event occurred that caused focus to be taken from the Activity to another one. In this state, the Activity may still be visible to the user but it's no longer in the foreground and the user can't interact with it until the user navigates back to it and puts it on the top of the stack. In other words, calling this method does not mean that the system is going to destroy the Activity.
5. **onStop:** this method is called to indicate to the Activity that it has entered the *Stopped* state where the Activity is no longer visible to the user and as a result the user cannot interact with it. In this state, the Activity is in a cross-roads, it could come back to be visible to the user and the user is able to interact with it or it finishes running and gets destroyed by the system. If the user navigates back to it, the activity is restarted and hence the `onRestart` method is called. Otherwise, the Activity is destroyed and hence the `onDestroy` method is called.
6. **onRestart:** this method is called when the Activity is navigated back to and becomes visible to the user and the user can interact with it. Generally, this method is used to indicate that the Activity is coming to the foreground. It is immediately followed by a call to the `onStart` method where the Activity enters the *Started* state.
7. **onDestroy:** this method is called when the Activity is being destroyed by the system. An Activity can be destroyed as a result of the user completely dismissing

it in which case this method will be the last invocation that the Activity class receives after which the system releases all the Activity resources. On the other hand, this method can be called as a result of a change in configuration such as a device rotation. After invoking this method, the system immediately calls the `onCreate` method to create a new instance of the Activity under the new configuration.

The Android Developer Guide [7] provides several guidelines on the handling of the Activity lifecycle callback methods with respect to resource allocation and releasing. It is recommended that all “global” state is set up in the `onCreate` method and released in the `onDestroy` method. During the paused state, while the Activity is still visible to the user, it is recommended to maintain the resources that are necessary to show what is needed to the user. In addition, it is highly recommended that any persistent data is written to the storage in the `onPause` method because it is the only method where the Activity is notified to lose the foreground state which is not killable by the system; the `onStop` callback is marked as killable meaning that the Activity may be killed at *any time* without another line of its code being executed after the method returns to the process hosting the Activity. In all cases, the system reserves its right to kill the application process at any time under extreme memory pressure.

Another aspect of Activity management in Android is the Activity Stack which is a stack data structure that is maintained by the Android system and contains the system Activities stored in the order they were opened. This stack is called the *back stack* [26]. The Activities in the back stack are never reordered, their position on the stack is the result of the Activities being popped and pushed onto the stack as a result of the user interactions or disruptive actions happening. When an Activity creates a new Activity, the new Activity is pushed to the top of the stack and becomes the Activity under focus. The previous Activity is stopped. When the user clicks on the back button, the Activity on the top of the stack is destroyed and the previous Activity on the stack is resumed.

Finally, it is worth noting that the Android operating system is built to run on top of the Linux operating system. Therefore, whenever an app's code needs to run, the Android operating system creates a new Linux process for it. The process will continue to run until it is no longer needed or the system needs to re-allocate its resources to another process. An Android process does not directly control its lifetime. The Android operating system decides whether to preserve a process based on the parts of the app that the system knows are running, how important they are to the user, and the overall memory availability in the system. The Activity objects inside an Android process are among other components that an Android app might contain and which contribute to the operating system's decision whether or not the system is going to kill the process. The other components are: *Service* and *Broadcast Receiver* [27].

Model-driven engineering is a process for bootstrapping the generation of mobile applications which starts with a model then through a series of transformation the final app is generated [28]. The models can be used to generate different types of artifacts including code, configurations. The code could be generated in different languages [29]. The models abstract the app development process and provides a certain level of precision and detail [30]. The process of model-driven engineering can be either forward [18]...etc. or backward [11], [12]. A meta-model is used to describe the model used in the model-driven engineering process. The meta-model is also used to derive the abstract syntax of a modeling language that includes the concepts and relationships in the meta-model [18]. The modeling language includes syntax that covers all the concerns that the meta-model covers and the concrete syntax and its semantics [31]. The modeling language is also called domain-specific language (or DSL). Domain-specific languages could be either visual (DSVL) or textual (DSTL) [22].

## **Chapter 3: Literature Review**

A critical literature review was performed where research papers were reviewed to survey the current state-of-the-art approaches for Android app development in general and Android Activity lifecycle in specific. The main finding of this literature review is that there is no approach or tool in the field of model-driven development that aims to help app developers properly handle the Android Activity lifecycle events and write a proper implementation in the lifecycle callback methods. The collected research papers were grouped into several categories. The next sections discuss each group individually and analytically.

### **3.1 Literature Survey Process**

The literature survey process depended on systematically searching for research papers reported in search results against the Google Scholar database. At least 10 pages in each search result were investigated. Additionally, a forward and backward snowballing approach was operated on the references inside each research paper to discover additional research papers. The 3.1.2 Inclusion/Exclusion Criteria discussed below was used to determine whether a research paper should be included in our final list. Figure 1 illustrates the literature review process employed in finding matching research papers.



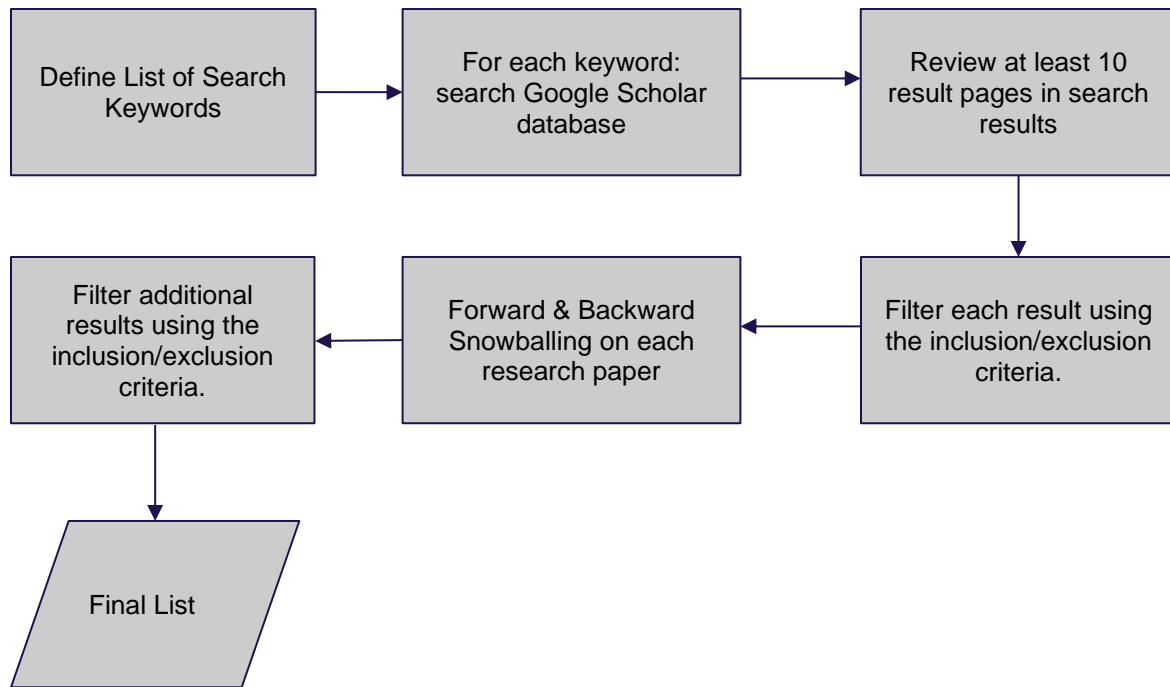


Figure 1: Literature Review Method

### 3.1.1 Searched Databases

Google Scholar was the primary database that was used to look up relevant research papers. The main keywords that were used: *model-based development*, *model-driven architecture*, *model-driven development*, *mdd*, *android*, *android lifecycle*, *android lifecycle model*, *android activity*, *activity lifecycle model*, *activity lifecycle*. Other databases such as IEEE Xplore, Scopus and ScienceDirect were also used to find research papers using the same keywords that were used against Google Scholar.

### 3.1.2 Inclusion/Exclusion Criteria

A special criteria of inclusion and exclusion was applied where a research paper was included if it is:

1. Published after the year 2016.
2. Attempted to address or analyze Android mobile app or Activity lifecycle implementation using model-driven techniques.
3. Included an empirical evaluation of the approach presented (if any).

4. 5 or more pages long.

This inclusion and exclusion criteria, when applied in the literature review method discussed above, have yielded a total of 22 research papers. It is worth mentioning that there was a considerable body of literature that was not included (i.e. excluded) because the research paper did not meet one or more of the above criteria. The works of Lachgar & Abdali [32], [33], Benouda et al. [34]–[36], Channonthawat et al. [37] on utilizing model-driven approaches to generate mobile applications were not considered because none of the presented approaches has a corresponding empirical evaluation to evaluate its validity. Also, the works of Sabraoui et al. [38], [39], Min et al. [40], de\_Almeida et al. [41], Lachgar & Abdali [42], Parada & De Brisolará [43], Le Goaer & Waltham [44], Madari et al. [45], Ko et al. [46], Mannadiar & Vangheluwe [47], Diep et al. [48], Heitkötter et al. [49], Kraemer [50], Son et al. [51] were excluded because they were published 5 or more years ago (as of 2020). Benouda et al. [34] was also excluded because the paper is less than 5 pages in length.

### **3.2 Model-based Android App Development**

Model-driven development or model-driven engineering is a category of software development where the app developer/modeler uses a high-level abstract model to describe the app. The high-level model allows specifying the structure and behavior of the user interface and the structure of the data collected from each page. The model could allow for more advanced scenarios such as backend service communication. Then, through a series of model transformations, the source code of the final app is produced. The developer can then apply final polishing to the code, compile and deploy it to the target device. The model can be either a visual or a textual model. Another synonym for the model is domain-specific language (or DSL).

Freitas et al. [18] identified the problem that existing model-driven engineering approaches do not generate complete model applications. To solve this problem, the authors presented

a graphical tool to model the business classes of an Android app and called it JBModel. The classes are modelled as UML classes including their relationships which are then transformed through a model-2-text transformer to POJO classes that target the JustBusiness framework which is then responsible to generate the UI, persistence code, and application resources necessary to compile and run the application. The developer is yet to implement the generated POJO classes manually. The authors claimed that an extra level of abstraction is added and therefore developers no longer need to deal with annotation and code details and can concentrate only on the modeling task. As a result, the development complexity is reduced.

A more advanced approach for model-driven development was presented by Vaupel et al. [19] where an app is modelled using a graphical tool which contains three different views for three different models. The data entities and their relationships are described using a *data model*. Data management, access to sensors, and use of other apps are described using a *process model*. The definition of the pages is described using a *GUI model*. At runtime, and to avoid redeployment of the app to reflect changes, another model, called the *provider model*, was introduced which contains instances of the data, process, and GUI models which serve as arguments to customize the behavior of the app at runtime. The authors aimed at providing an approach that can reduce the development cost and time that are associated with cross-platform app development. The authors claimed that the more standardizations that are made in the code and UI, time-to-market is considerably reduced.

An example of using pure textual model-driven development is presented in Thu et al., [21] where a rule-based transformation was presented which takes as input a textual model written in Umple and generates an entire app following the MVC pattern. The transformation rules are written in the Drools Rule language. The authors aimed at filling the gap that existing model-driven development approaches lack support to guarantee consistency and quality of the generated code. The authors argued that adoption of model-

driven development can simplify the development of mobile apps with significant reduction in technical complexity and development costs.

Rieger et al., [20] wanted a modeling language that did not require software engineering knowledge (technical complexity), can be understood by domain experts, and can be easily interpreted by code generators (avoid GUI oversimplification). They presented a graphical DSL, called MAML, to make this trade-off, accompanied with model transformers to allow generating native apps from the graphical model. The authors identified that existing general-purpose modeling notations such as BPMN are not detailed enough to cover mobile-specific aspects and are hardly interpreted by code generators from a technical point of view. Additionally, technical notations such as IFML are too complex to be understood by domain experts and require software engineering knowledge. Graphical editor components for the same notation differ significantly in modeling effort, learnability, and memory load for the user. The authors argued that DSLs are generally suited to cover a well-defined scope with sensible abstractions for domain concepts. DSLs increase programmers' productivity compared to general purpose languages. Textual DSLs provide minor benefits to non-technical users because they feel like programming.

An instance where multiple views are presented to model-driven developers exist in the work of Barnett et al. [22] where the authors presented a tool, called RAPPT, which leverages the two types of models (visual and textual) as two DSLs. The visual modeling language allows specifying high-level structures such as the number of screens, navigation flow between screens...etc. while the textual modeling language allows providing low-level details such as backend services, data schema, authentication...etc. To achieve this, the authors created a shared internal model which the two modeling languages target. The authors' approach aimed to fill the gap that although modeling techniques such as DSLs simplify app development by abstracting the details and hence improve developer productivity, their generated output is rigid and lacks the flexibility of specifying custom functionality. Traditional IDEs lack the communication transparency between UI designer

and source code behind the UI; it requires developers to navigate the code to understand the UI behavior. Traditional mobile app prototyping is discarded as soon as it is agreed upon and developers are required to replicate the prototype in code. In addition, non-visual aspects of the mobile app are rarely prototyped due to the cost.

Model-driven mobile app development was also found to be utilized in the generation of context-aware apps which can adapt to contextual parameters such as the user's computing infrastructure, location...etc. Pervasive computing embeds computing resources into the environment and provides services for users ubiquitously and transparently. Pervasive computing applications can sense the environment and react based on the environment. One kind of applications from the pervasive computing paradigm are activity-oriented context-aware applications (AOCA). Current development methods for pervasive computing applications do not consider the activity-oriented incremental development and thus cannot support the development and maintenance of AOCA applications in a flexible way. In addition, the development of AOCA applications using the existing programming framework and API still lacks enough guidance and developers need to spend a considerable amount of time learning how to use the API. Li et al. [52] presented a meta-model for context-aware activity-oriented applications with a platform-independent DSL, called AocML, to describe the application. The DSL is then converted, using a model-to-text transformation to Java code. The generated Java code targets the platform for activity-oriented context which is a Java-based platform for supporting the development and runtime of activity-oriented context-aware applications.

Various approaches exist to ease the task of designing and implementing applications that are context-aware and self-adaptive. At their core, these approaches build on reusable code. The context-awareness and context-inferring parts are separated from the functional logic by using context plug-ins which are individually deployable units that provide mechanisms for collecting and processing context data and inferring higher-level context information. A common drawback of these approaches is that developers are required to invest a

significant amount of time to develop customized components to collect, process, infer, store, query, and access context data. In addition, those components are harder to reuse and more error-prone which prevents cost-effective development. Paspallis et al. [53] wanted to facilitate the development of context-aware applications by presenting an approach for the design of reusable context plug-ins that can be used to monitor low-level context data and infer higher-level information about the users, their computing infrastructure and interaction. A UML profile was presented to be used as a modeling language. The UML model is converted to XMI which in turn is converted to XML/UML2 as expected by the underlying model transformer (MOFScript). MOFScript is then used to generate the actual source code for the context plug-in. The authors argued that component-based development with a component repository can greatly facilitate the development of such context-aware applications. As a result, the development of highly capable and robust context-aware applications is more affordable.

Yigitbas et al. [54] extended model-driven development to create mobile apps with user interfaces that are self-adaptive to contextual information. The authors presented two DSLs: ContextML and AdaptML. ContextML allows defining context properties and context providers that capture relevant context information. AdaptML allows modeling the UI adaptation rules. Several models are presented, Abstract UI model, Domain Model, Context Model, and Adaptation Model. The Abstract UI and Domain models are used to generate the final UI of the app. The Context Model is used to generate Context Services that monitor context information like accelerometer, GPS, brightness, or noise level. The Adaptation Model which references the Context Model allows defining constraints for triggering adaptation rules which reference the affected UI elements in the Abstract UI model. The Adaptation Model is used to generate an adaptation service which monitors information collected by the context service and adapts the final UI at runtime. The presented approach aimed to fill the gap that existing UI modeling approaches introduce additional complexity when modeling context-management and UI adaptation aspects due

to crosscutting concerns. The result is a tightly interwoven model that is hard to understand and maintain.

It can be observed from the previous survey of the current state-of-the-art approaches for model-driven development of mobile apps that there's no special attention given for the generation of proper handling of the app lifecycle. In addition, even while these approaches present a significant boost in development time and increased separation of concerns, yet they do not provide the necessary flexibility required by the developers and therefore, for apps with a long lifespan, the model-driven approach can only be used to bootstrap the initial app development. Any new features, enhancements, or bug fixes have to be performed manually without the model-driven approach, and where the app lifecycle handling is concerned, the proper implementation of the lifecycle callback methods is again left to the developer and the model-driven tool is useless in that case.

### **3.3. Android Activity Lifecycle Handling Analysis**

Android Activity lifecycle handling analysis starts by presenting a model of the Android Activity lifecycle implementation. Then, given the Android Package (APK) file or the app's source code, a tool will parse the app's source code, develop an instance of the presented model, then analyze the model instance to detect issues with the app's implementation of the Activity lifecycle callback methods.

#### ***3.3.1 Security Analysis***

A considerable portion of the literature which includes a modeling of the Android Activity lifecycle is concerned with security analysis. Junaid et al., [12] attempted to construct an Android lifecycle model that captures all the states and transitions implemented in the app. The model is then systematically used to derive lifecycle callback events which are then used by taint analysis techniques to detect malicious app behavior. The authors aimed to fill the gap that an Android-supplied lifecycle model does not capture all states and transitions implemented in Android and does not specify guard conditions that govern the

invocations of the callbacks. As a result, any analysis model may not detect attacks that exploit such omissions in the lifecycle model.

The analysis of malicious behavior with respect to data flow analysis was extended by Li et al., [13] to investigate the effect of the Android Fragment lifecycle on the Activity lifecycle. The authors introduced a control flow graph model for control flow transfers of Fragments' and Activities' lifecycles and used the model to perform information leakage detection to eliminate the false positives in the current state-of-the-art information leakage techniques which focus only on the Activity lifecycle. The authors identified that none of the existing approaches describe the effect of the Fragment's lifecycle on data flow analysis. If Fragments are not taken into consideration, then some data flows in the app will be missed leading to false negatives when analyzing data leakages. A malware can have the lifecycle of a Fragment to avoid detection methods that are based on data and control flow analysis. The authors concluded that a Fragment life cycle has an effect on the data leakage detection results.

A stealthy attack goes through multiple states, state transitions are caused by sequence of attack actions, and an attack action typically involves multiple Android APIs on different objects. Malware detection models are insufficient to capture sophisticated stealthy attacks that involve multiple states. To detect Android apps that execute additional actions to hide their malicious behaviors (i.e. stealthy attacks), Junaid et al. [14] presented a static analysis framework which implements a FSM model which includes a modeling of the Android lifecycle callbacks to depict actions and action-sequence based stealthy attacks.

Another category of apps that alter their behavior are diehard apps which either directly or indirectly alter their lifecycle to avoid being killed by the operating systems. While Android's permissive lifecycle controls give apps more flexibility to react to user interactions and system events timely enabling rich functionalities, it opens doors for apps to directly or indirectly alter their lifecycles; apps can easily abuse their entry points to



automatically start up in the background, requiring no user interaction and game the lifecycle management mechanism to evade being killed. The results are battery drain and device performance degradation. This behavior is called the diehard behavior and apps exhibiting it are called diehard apps. Existing Android features that were introduced to limit the background apps affect such apps to a certain extent but cannot fundamentally solve the problem and all have obvious limitations. Shao et al. [15] presented a runtime framework which is based on a system-wide app lifecycle model called Application Lifecycle Graph (ALG). The framework builds the ALG at runtime and uses it to realize fine-grained lifecycle control of apps. The authors argued that diehard behaviors violate the system's app lifecycle control and should be better managed. Apps should more gracefully achieve long-running and clearly indicate their background activities using Android recommended approaches instead of abusing or gaming the system life cycle management mechanism.

### ***3.3.2 Android App Testing***

Another purpose for modeling the Android app lifecycle was found to assist in the black-box testing of Android apps. Riccio et al. [5] identified that none of the existing Activity lifecycle conformance testing approaches address GUI failures that are a result of an unexpected GUI state. The authors presented a black-box testing approach that is able to detect crashes and GUI failures which are tied to the Activity lifecycle implementation. The authors argued that lifecycle event sequences are able to exercise the Activity lifecycle and expose failures. In specific, the *Double Orientation Change* event sequence is more effective than the *Background Foreground* and *Semi-Transparent Activity Intent* event sequences in revealing GUI failures and crashes. In addition, the faults causing Activity lifecycle failures were mostly located outside the code implementing the lifecycle callback methods.

### ***3.3.3 Memory Leaks Detection***

Some memory leaks detection approaches focus only on a subset of bad programming practices and may report a considerable number of false positives. Other approaches which depend on dynamic analysis techniques often require the source code to be available in order to be applicable. Yet another group of approaches require an app model to be able to generate test cases and when a model is not available, their usage is not straightforward. Amalfitano et al. [2] presented an approach to automatically detect memory leaks that are due to bad programming practices with focus on bad handling of system events that are tied to the Android Activity Lifecycle. The authors argued that there is a large spread of memory leaks tied to the Activity Lifecycle Events in real Android apps.

Finally, and in order to avoid context leaks in Android, contexts should not be made reachable from static fields or threads which are roots of non-garbage-collectable data. However, this rule is easily violated in practice because contexts are often contained in other objects such as views or fragments. Existing solutions to detect context leaks in Android applications are based on either dynamic or static analysis. However, both types of tools only recognize syntactic code patterns and do not perform semantic analysis. Toffalini et al.'s [3] attempted to detect context leakages in Android native apps by presenting a static code analysis tool which identifies static fields that reference contexts either directly or indirectly. Potential leaks are then systematically analyzed w.r.t severity.

### ***3.3.4 Other Approaches***

Other approaches in the literature exist which build a model of the Android app lifecycle to achieve other purposes.

App modeling is a better approach than direct app source code analysis because apps are event-driven and do not have a fixed program entry. The existing app modeling approaches, callback-directed and data-directed both suffer some drawbacks. While callback-directed app modeling is able to provide logical structure over the entire app program, it can hardly

provide concrete assistance for analysis in practical scenarios since they are closely combined with solid data input and output. On the other hand, data-directed app modeling has a heavy cost in constructing a generic model. In general, neither of the two approaches is able to conduct a callback model in a generic and fine-grained manner. In specific, existing approaches are not complete. While Activity is involved in the modeling, Service and BroadcastReceiver, where security compromises and logic bugs frequently exist, are ignored. Only lifecycle related callbacks and user interaction are taken into consideration and the generated callback model only considers the possible flow from a start node to an end node ignoring information such as how and when the flow is executed. Guo et al. [17] presented a tool that constructs a generic callback-related model to support path-sensitive analysis. The model can identify connections between different components, path-sensitive conditions, and the handling of the system-driven fine-grained lifecycle callbacks.

Perez et al. [8] identified that when analyzing Android apps, the current state-of-the-art tools fail to generate sequences of callbacks that match the runtime behavior of Android apps. There is not a representation of sequences of callbacks that integrates different sources of changes of control to be directly usable by analysis and testing tools. The authors presented a program representation that can generate sequences of Android lifecycle callbacks that match the runtime behavior.

Event-based races are concurrency errors due to the fact that the posted events by the Android framework are nondeterministic. Event-based race concurrency errors form the majority of Android race bugs and are 4-7 times more frequent than data races. Existing tools that are based on dynamic analysis are prone to false negatives and greatly depend on high-quality inputs to ensure good coverage. In addition, Android's concurrency model makes it difficult to establish happens-before relations. To detect race conditions in Android apps, Hu et al. [55] presented a static analysis tool that is able to model threads, messages, lifecycle Activities and GUI events. Then, static analysis is used to produce

candidate races where false positives are ruled out by path-sensitive, backward symbolic execution.

Finally, novice Android developers coming from different background technologies (e.g. Web and Desktop) face difficulties in developing high quality and reliable Android apps. In specific, novice Android app developers face a challenge building apps that conform to the lifecycle rules. The nature of mobile application development itself imposes challenges in terms of managing the app lifecycle events correctly to ensure app reliability. Little attention has been made to test conformance of the mobile app lifecycle. To detect non-conformant Android lifecycle handling, Zein et al. [16] presented a static code analysis tool that can verify that system resources, that are shared between different mobile apps such as Camera, GPS, Network connections...etc., have been correctly initiated and released inside Android apps.

Although given their varying accuracies in detecting Android lifecycle-related issues, it can be observed from this part of the literature that none of the existing tools and approaches presents an Android development-integrated support to assist developers in building robust and reliable Activity lifecycle handling.

The literature is full of tools and approaches that help Android developers bootstrap their apps, analyze existing apps for security vulnerabilities, memory leaks, possible malicious behavior...etc. However, it is observed that given this variety of tools and approaches, yet there exists no approach that directly attempts to help Android developers write a proper Android Activity lifecycle handling that conforms to the rules of the Android Activity lifecycle model.

## **Chapter 4: Methodology**

To answer RQ1 and to address the problem of improper Android Activity lifecycle handling, a model-driven, multi-view and generic approach is presented to assist Android developers get a high-level view of their implementation of the Activity lifecycle. The approach implements a generic architecture that utilizes two DSLs (a DSTL and a DSVL) and can be easily extended. The architecture is presented in Figure 2.

Our choice of a multi-view approach was because it has been found that using multi-view based approaches for mobile development is beneficial while addressing multiple and specific concerns during app development [22] and with an extra level of abstraction, developers no longer need to deal with annotation and code and can concentrate on the model which reduces development complexity [18]. In general, model-driven development was found to have a high potential for accelerating software development for multiple platforms. It also increases standardizations in code and UI and considerably reduces time-to-market [19]–[21]. Also, it has been found that visual models (or graphical DSLs) are generally more suited to cover a well-defined scope with sensible abstractions for domain concepts whereas textual domain-specific languages provide minor benefits to non-technical users because they feel like programming [20].

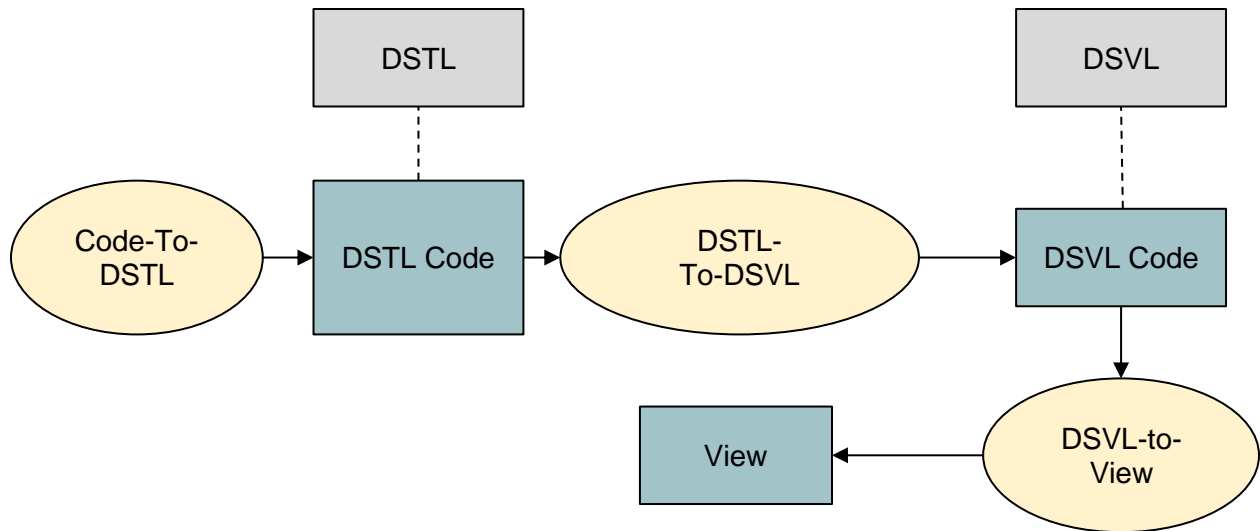


Figure 2: Approach Architecture.

The approach defines a DSTL that can be used to describe an Activity lifecycle implementation. A static code analyzer is triggered (Code-To-DSTL in Figure 2) when an Activity file is opened or selected. The code analyzer parses the Activity file source code and creates an internal model that conforms to the presented DSTL. The code analyzer procedure parses the Activity class looking for the Activity lifecycle callback methods that are implemented in it. Figure 3 illustrates the algorithm used to parse the Activity class.

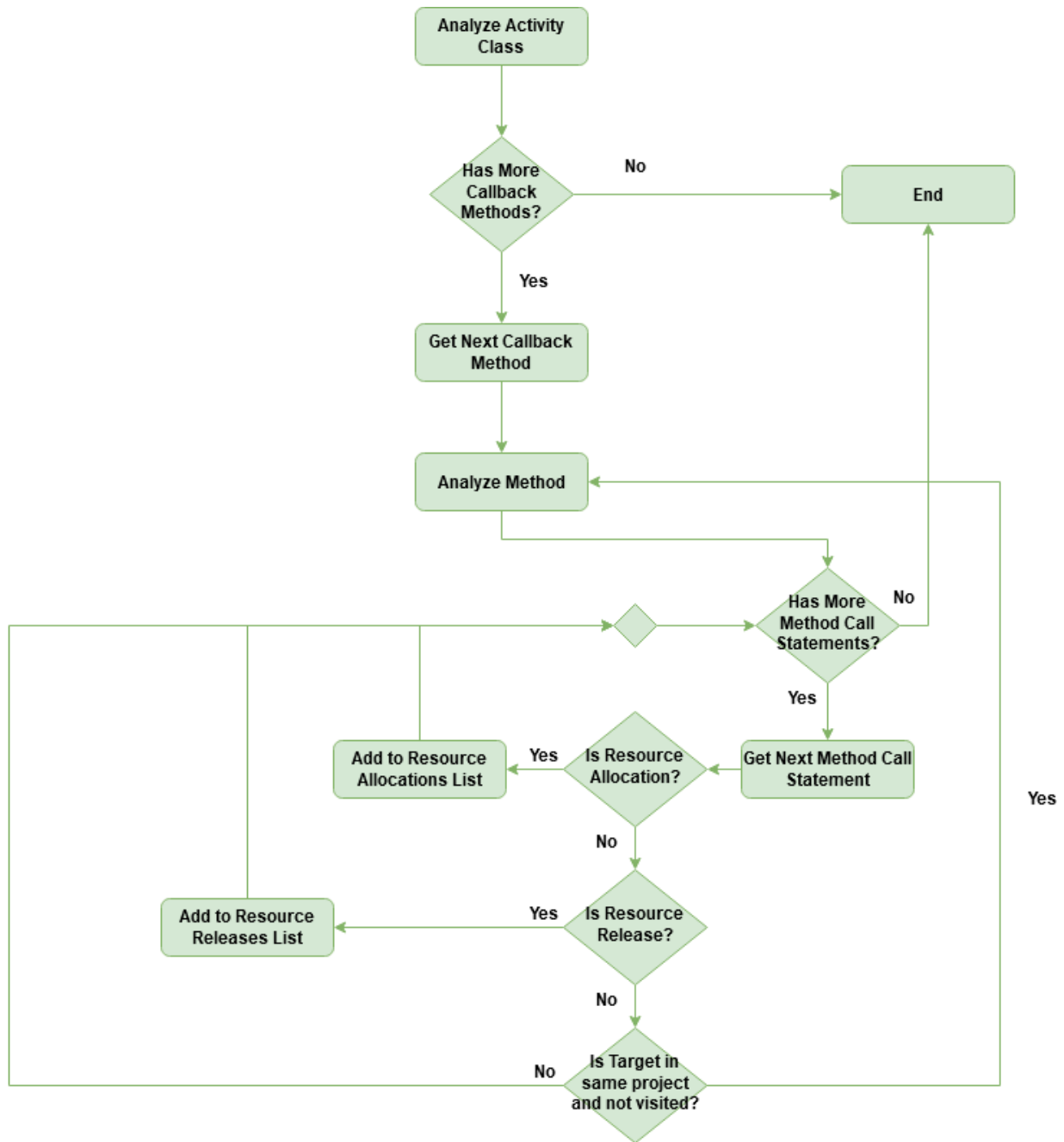


Figure 3: Code Analysis Algorithm

From Figure 3 it can be observed that only lifecycle callback methods are considered in the Activity class and when navigating the syntax tree of the callback method, the algorithm looks for method call statements only and depends on an external logic to determine whether a method call statement is recognized and is either a resource allocation or a resource release method call statement. One implementation of this external logic is used

in section 5.2.5 Plugin Settings in the implemented tool. The tool defines a configuration file that contains two sets of key-value pair entries: *Resource Allocations* and *Resource Releases*. The key in each entry refers to the name of the resource and the value refers to the fully-qualified method name which consists of the method name attached to the fully-qualified class name. Therefore, the analysis algorithm upon detecting a method call from any of the two sets will create the corresponding node from the DSTL. Additionally, the algorithm does not navigate beyond classes that are defined in the current Android project. Furthermore, the algorithm does not consider Activity class methods that are invoked within a specific Activity state. For example, a button handler that is invoked when the user clicks on a button rendered on the Activity window while the Activity is in the *Resumed* state is not detected. The reason why such a capability is necessary is because different resources could be allocated or released inside UI handler methods that are invoked as a result of user interaction with the Activity through its UI or the device sensors. Such a capability requires mapping the events that the Activity registers for and handles in the methods defined in its class or methods that are potentially defined outside the Activity class but are referenced from the Activity class or refer to the Activity class.

In addition, a DSVL is presented that can be used to describe instances of the DSTL for rendering purposes. After the intermediate DSTL model is generated, a DSTL-to-DSVL converter is used to transform the intermediate DSTL model into a DSVL model that conforms to the rules of the DSVL. Finally, the DSVL model is processed by a DSVL-to-View engine to render the visual model. The textual and visual DSLs allow the representation of information such as an Activity that is present in the app and the lifecycle callback methods implemented in it. Figure 4 illustrates the algorithm that is used to transform an instance of the DSTL to a corresponding DSVL model.



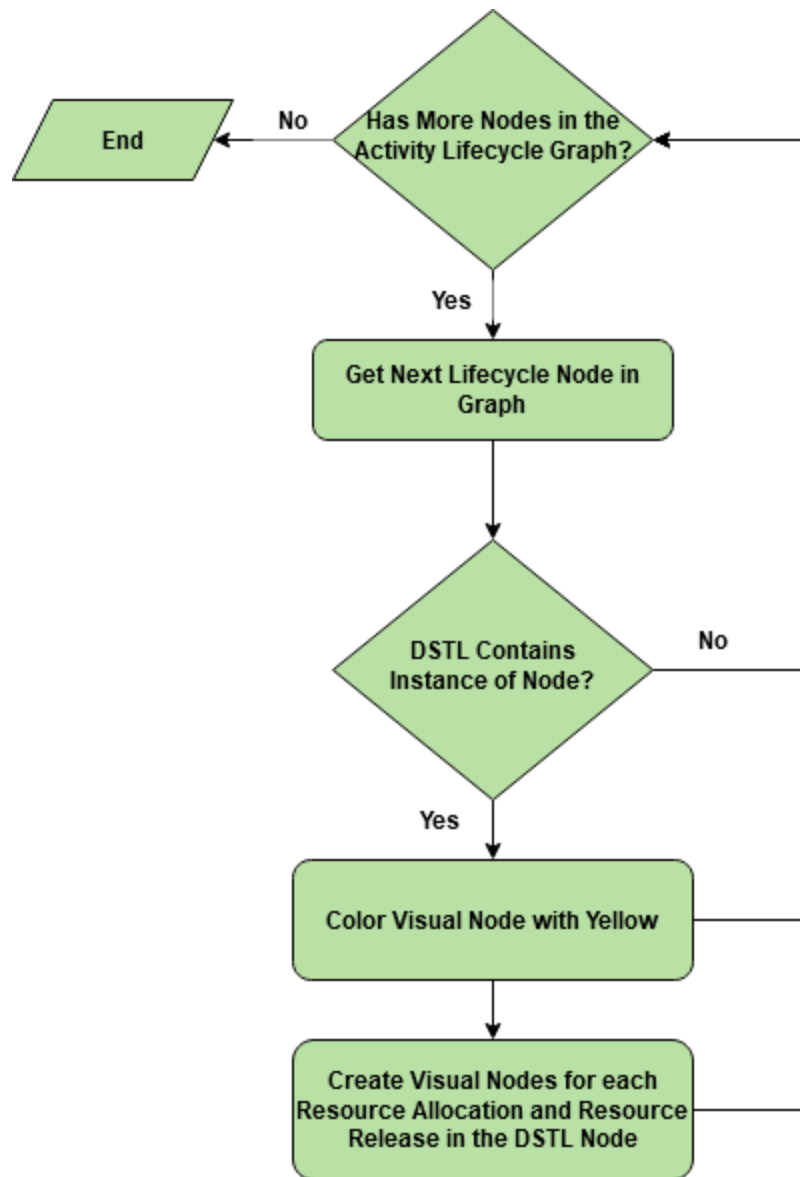


Figure 4: DSTL to DSVL Converter

The DSVL, described in the 4.1.2 section, defines for every Activity lifecycle node in the DSTL, a corresponding visual node. Therefore, any instance of the DSVL must contain all Activity lifecycle nodes. However, when the DSTL to DSVL converter algorithm generates an instance of the visual model, the converter changes the color of the visual node depending on whether there exists an underlying node in the DSTL model. If there exists an underlying DSTL node for the visual node, the converter colors the visual node with a Yellow background, otherwise the converter chooses the Gray color for the background.

The DSVL-To-View component in our architecture will typically have an implementation-specific algorithm which depends on the actual technology that is used to render the view. In our implementation (see section 5.2.7 Tool Window), the DSVL-To-View component is implemented by using IntelliJ's platform tabbed window component to render the view which is based on top of the Java Swing architecture.

It can be observed that the previous architecture is a pluggable architecture where each of the main components (Code-To-DSTL, DSTL-To-DSVL, and DSVL-To-View) can be replaced by a different implementation when needed. This has the potential of increasing the extensibility of any implementation that depends on this architecture. Also, the fact that each of the components is separate from the others and depends on a well-defined data format produces a loosely coupled architecture where each component can be developed and maintained independently of the other components. The result will be that any implementation that depends on this architecture will have a high level of cohesion and low coupling among the components.

The input of the previous architecture is the Activity class implementation (i.e. textual code) that is typically written in the Java or Kotlin programming language. The output of the previous architecture is a view. The input of the architecture is the first and default view that the developer sees. However, the output is an additional view. As a result, we call our approach a “multi-view” approach because the developer now has two views of his implementation of the Activity lifecycle. The intermediate messages that pass between the components: Code-To-DSTL and DSTL-To-DSVL, DSTL-To-DSVL and View depend on well-defined models and as a result we call our approach a “model-based” approach. Finally, the previous architecture is a generic approach that can be adapted to any kind of mobile app’s implementation. In order to answer RQ1, an Android specific version of both the textual and visual DSLs was defined and an implementation of the previous architecture, as an Android Studio plugin, was built to process implementations of the Android Activity lifecycle implementation. It should be noted that the previous architecture

can have analogous versions and implementations for other types of mobile application development technologies such as iOS, UWP and Xamarin.

## 4.1 Domain-Specific Languages

In our approach, two domain-specific languages are presented: a DSTL and a DSVL. The DSTL is defined using the UML modeling language and the DSVL is defined in terms of visual notations. The DSTL definition can have implementations in any modeling language such as the Extensible Markup Language (XML) or JavaScript Object Notation (JSON) while the DSVL can have implementations in any general purpose programming language such as Java and C#.

### 4.1.1 Meta-model for approach DSTL

The DSTL presented here allows representing Android Activity classes, the implemented lifecycle handling callback methods, and any resource allocation or release inside each of the lifecycle callback methods. The design of the domain-specific textual language is presented in Figure 5.

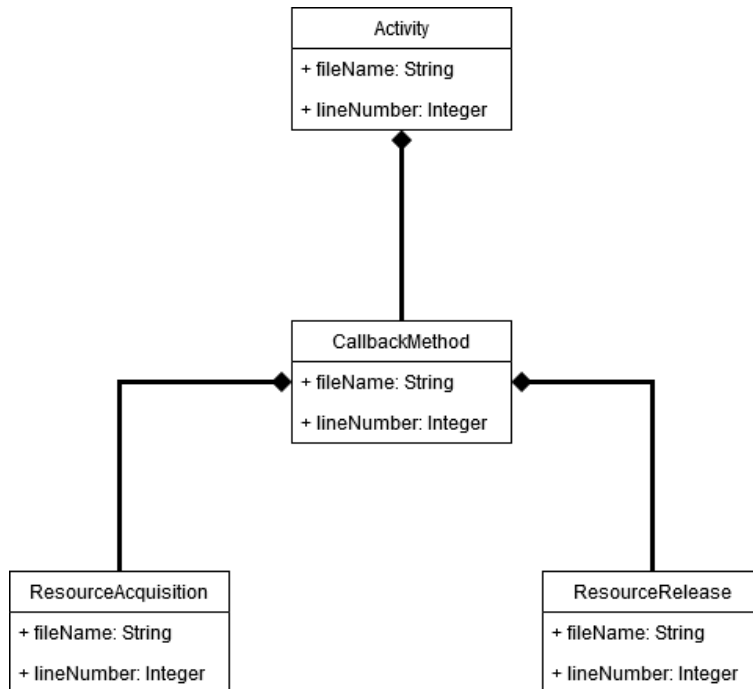


Figure 5: Domain-Specific Textual Language.



From Figure 5, it can be observed that the language defines the following entities:

1. **Activity:** this entity represents an Android Activity. This node is the root node for all the other entities in any instance of the DSTL model.
2. **CallbackMethod:** this entity represents a lifecycle callback method that is implemented in the Activity class. Specifically, they are onCreate, onStart, onResume, onPause, onStop, onRestart, and onDestroy.
3. **ResourceAcquisition:** this entity represents an attempt to acquire a resource inside the callback method or any other method in its subtree. An “attempt” to acquire a resource is a method call in the subtree of the callback method.
4. **ResourceRelease:** this entity represents an attempt to release a resource inside the callback method or any other method in its subtree. An “attempt” to release a resource is a method call in the subtree of the callback method.

It should be noted from the previous model that an Activity class can contain multiple CallbackMethod instances and a CallbackMethod instance can contain multiple ResourceAcquisition and ResourceRelease instances. In addition, each entity in the previous model contains a fileName and lineNumber attributes that point to the location in the app source code where the entity is defined or located.

#### 4.1.2 Meta-model for approach DSVL

The DSVL allows representing the Activity lifecycle states and the transitions between them. Table 1 illustrates the node types that the language defines.

Node	Description
	This node type represents a callback method that is implemented by the Activity class. The callback method represented by this node type handles the transition of the Activity’s state to the state represented by this node type.
	This node type represents a callback method that is not implemented by the Activity class. It means that the Activity class does not handle the transition to the state that is represented by











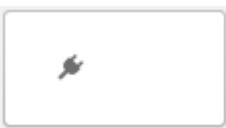
	this callback method.
	This node type represents a resource that has been allocated.
	This node type represents a resource that has been released.
	This node type represents a recursive callback method. This node type was introduced to eliminate the need to render the life cycle graph as a circular graph and allows representing it as a tree.

Table 1: Building Blocks for the DSVL

Table 2 explains the transition lines between each two of the previous node types:

Source Node	Target Node	Description
		The line between the source node and target node means that the Activity lifecycle transitions from the source state to the target state and only the source state transition is handled.
		The line between the source node and target node means that the Activity lifecycle transitions from the source state to the target state and both state transitions are handled.
		The line between the source node and target node means that the Activity lifecycle cycle transitions from the source state to the target state and only the target state transition is handled.
		The line between any of the source nodes and the target node means that the callback method represented by the source








		node acquires a resource that is represented by the target node.
		The line between any of the source nodes and the target node means that the callback method represented by the source node releases a resource that is represented by the target node.
		
		The line between any of the source nodes and the target node means that the Activity lifecycle transitions from the source state to a previous state that is represented by the target node.
		

Table 2: Connections between Nodes of the DSVL

## 4.2 Evaluation, Data Collection, and Analysis

In order to understand the extent by which the approach presented in this research can assist Android developers in implementing Activity lifecycle (answer RQ2), the approach presented in this research was evaluated. In order to evaluate the approach presented in this research, an implementation of this approach was first created. The implementation targeted the Android Activity lifecycle implementation. The evaluation of the presented approach followed the method of Barnett et al. [22].

A case study was conducted where the participants are Android mobile app developers with varying levels of experience. The participants were asked to fill a questionnaire where they answer questions about their experience using the presented implementation. The questionnaire consists of 7 questions with 5-point Likert scale ranging from Strongly Disagree to Strongly Agree and 2 open-ended questions that capture the participants'

experience of using the implementation. The questionnaire was designed by following the positive questionnaire design approach that is suggested by Sauro et al. [56].

The data was analyzed to understand the relationship between the developers' familiarity with Android Activity lifecycle and their satisfaction with the implementation. This analysis was based on the answers to the 5-point Likert scale questions. Also, analysis of the developers' experience was collected using the participants' answers to the open-ended questions included in the questionnaire. Furthermore, another analysis was performed to compare the results obtained from the close-ended questions and those results obtained from the open-ended questions.

## Chapter 5: Implementation

The multi-view approach presented in this research was implemented as an Android Studio plugin to allow developers to view their Android Activity lifecycle implementation. Figure 6, Figure 7, and Figure 8 illustrate the plugin as seen by the Android developer:

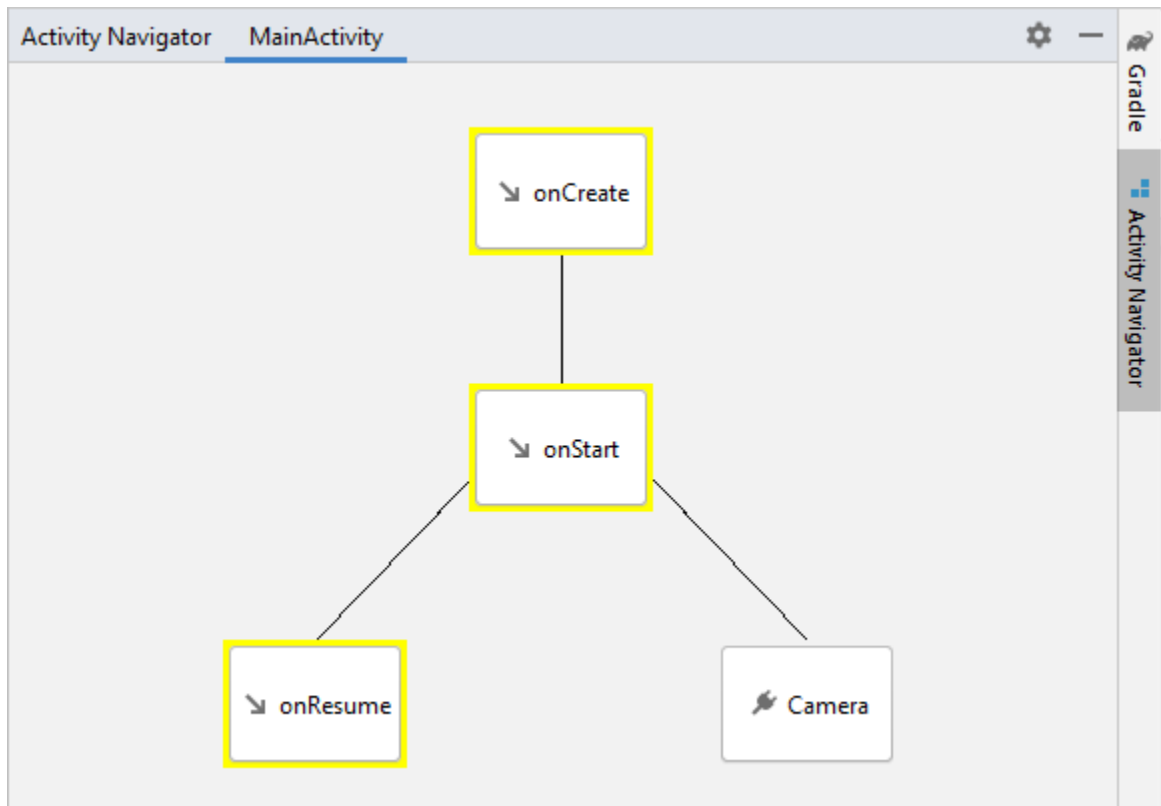


Figure 6: The DSVL implemented in Android Studio plugin.



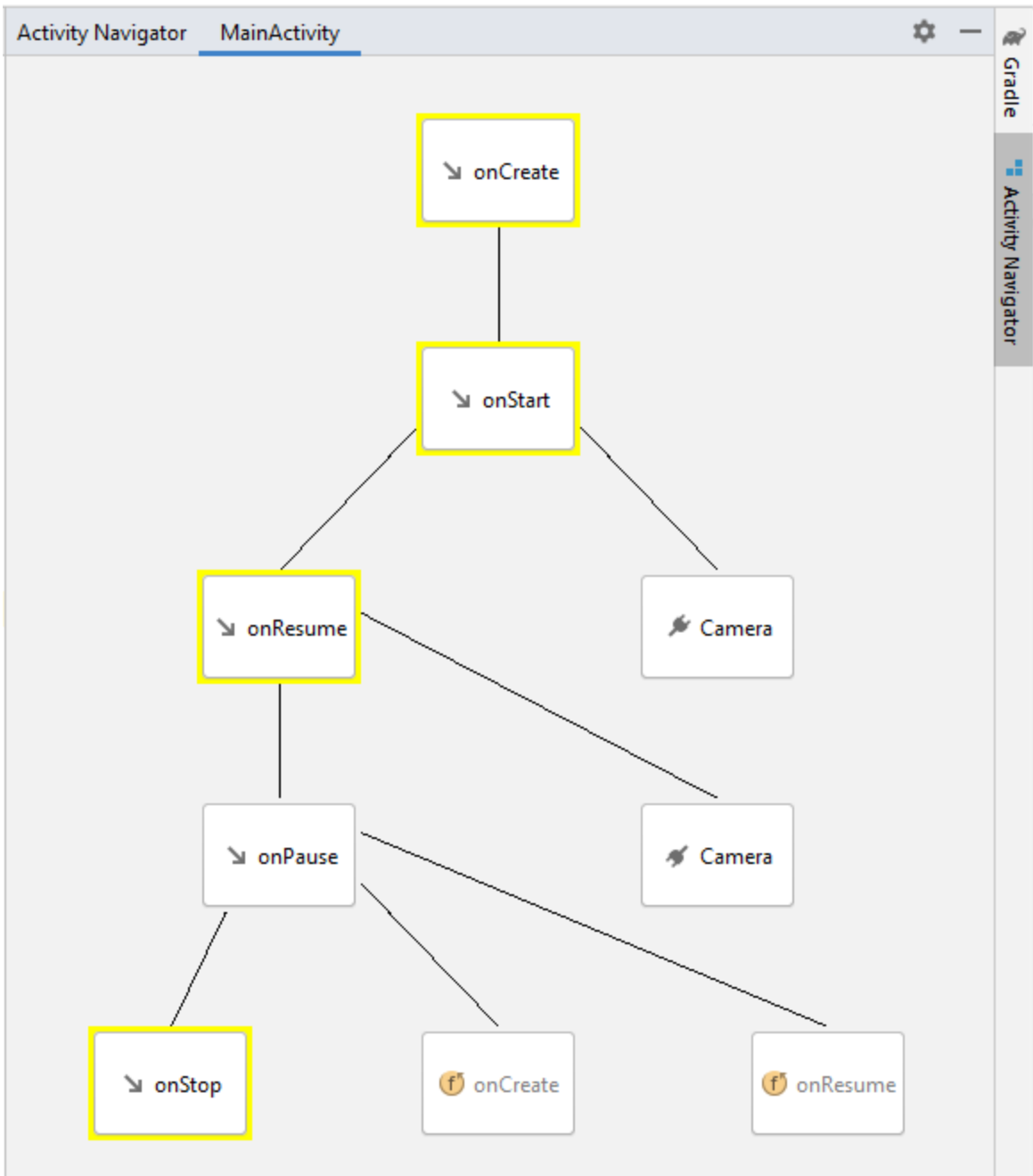


Figure 7: The presented Activity lifecycle view allows expanding and collapsing the tree nodes.



Figure 8: The full view of the Activity lifecycle when fully expanded.

From the previous figures, it can be observed that the view is progressive, and the developer has the option to hide/show a subtree of the lifecycle graph. The implementation includes additional features for the developers such as navigating to the implemented callback method, adding an unimplemented callback method, and navigating to the allocated or released resource.

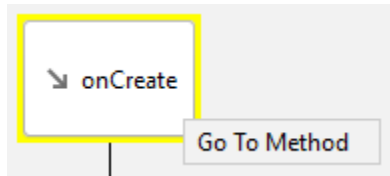


Figure 9: The developer can navigate to an implemented callback method.

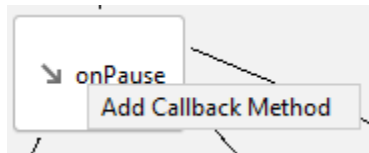


Figure 10: The developer can add unimplemented callback method.

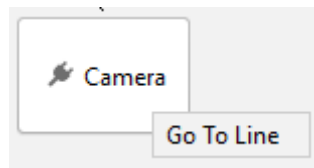


Figure 11: The developer can navigate to the line in the source allocation where the resource allocation is located.

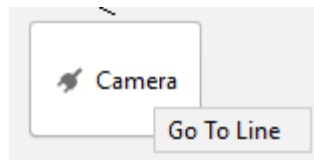


Figure 12: The developer can navigate to the line in the source code where the resource release is located.

## 5.1 Overview

The approach was implemented as an Android Studio plugin<sup>1</sup>. The plugin was developed as a plugin to the IntelliJ Platform which allows it to be run on virtually any IDE that is developed on top of the IntelliJ Platform (e.g. IntelliJ IDEA, Android Studio...etc.). When developing for the IntelliJ platform, IntelliJ IDEA was used to develop the plugin. Development for the IntelliJ platform depends on two fundamental design patterns: Service-Locator and Message Bus. IntelliJ platform-compatible plugins can define services that are referenced by both the platform itself and other components inside the plugin. The Message Bus design pattern allows components to listen to specific types of messages and to send messages of specific types to all listeners that are loaded by any instance of the IntelliJ platform. Both design patterns allow for loose coupling between all components that are loaded by the instance of the platform. An instance of the IntelliJ platform is an IDE such as IntelliJ IDEA and Android Studio. The following sections discuss the major aspects of developing an IntelliJ platform-compatible plugin with examples from our own implementation.

### 5.1.1 IntelliJ Plugin Metadata

The metadata of an IntelliJ platform-compatible plugin exists inside a file called **plugin.xml** inside the META-INF folder at the root of the plugin project hierarchy. The plugin.xml file includes metadata about the plugin. It also includes extensions that are used by the IntelliJ platform and other plugin components. The plugin metadata file is an important file because it is used by the IntelliJ platform to install the plugin, discover its dependencies, the version of the IntelliJ platform and it depends on...etc. Listing 3 illustrates a portion of the plugin.xml file used for the plugin.

```
<idea-plugin>
  <id>org.birzeit.swen.AndroidLifecycleAnalyzer</id>
  ...
```

---

<sup>1</sup> The source code can be found at <https://github.com/tghanem/android-lifecycle-visualizer>.

```

<depends>com.intellij.modules.java</depends>
<depends>com.intellij.modules.platform</depends>
<extensions defaultExtensionNs="com.intellij">
  <toolWindow
    anchor="bottom"
    factoryClass="impl.toolwindows.ActivityLifecycle..."
    icon="/tool_window_icon.png"
    id="Activity Navigator"
    secondary="true"/>

  <applicationService
    serviceInterface="interfaces.INotificationService"
    serviceImplementation="impl.services.Notifica..."/>

  <applicationConfigurable
    parentId="tools"
    instance="impl.settings.AppSettingsConfigurable"
    id="impl.settings.AppSettingsConfigurable"
    displayName="Activity Lifecycle Navigator" />
</extensions>
</idea-plugin>

```

Listing 3: IntelliJ Plugin XML File

From Listing 3, it can be observed that the plugin uses three types of extensions: *toolWindow*, *applicationService* and *applicationConfigurable*. A Tool Window is a child window of an IntelliJ IDE that is used to display information [57]. The tool window is a tabbed window that contains multiple tabs and each tab has its own panel that is used to render part of the window's UI. Tool Windows are automatically detected by the IntelliJ platform and an item is added for them in the View → Tool Windows menu and an instance of the window is created by the platform using the class referenced in the *factoryClass* attribute. The factory class is given an instance of the ToolWindow class and is expected to populate the content manager of the ToolWindow with the UI components. The content manager of the tool window contains support to add multiple content Panel objects to the tool window where each Panel object is displayed inside a separate tab in the window.

The other type of extensions that the plugin uses is called an *applicationService*. Application Services are one type of Plugin Services [58] that can be used inside an IntelliJ platform plugin. A service is a component that can be located by any other component in the plugin by calling the *ServiceManager.getService* method. A Plugin Service is defined in the plugin.xml by two items: the service interface, specified in the *serviceInterface* attribute and service implementation which is specified in the *serviceImplementation* attribute. The service interface defines the contract that the plugin service implementation must implement. It includes the methods that can be used to access the functions of the service; it is a Java or Kotlin interface. The service implementation provides the concrete class that implements the service interface. The *ServiceManager* class is responsible for loading and maintaining instances of these services when requested by other components in the plugin. This design pattern promotes loose coupling between plugin components.

It can also be observed from the plugin.xml file that the plugin depends only on the Java and IntelliJ platform SDKs. This has the potential of making the plugin run on virtually any IDE that is built on top of the IntelliJ platform such as Android Studio, IntelliJ IDEA...etc.

### 5.1.2 IntelliJ Plugin Settings

Another type of extensions in the IntelliJ platform that the plugin uses is plugin configurations. This is done by registering an *applicationConfigurable* in the plugin.xml file. An *applicationConfigurable* extension must implement the *Configurable* interface in order to be recognized by the IntelliJ platform. Listing 4 illustrates a sample class that implements the *Configurable* interface.

```
public class AppSettings implements Configurable {
    @Override
    public String getDisplayName() { ... }
}
```

```

@Nullable
@Override
public JComponent createComponent() { ... }

@Override
public boolean isModified() { ... }

@Override
public void apply() throws ConfigurationException { ... }

@Override
public void reset() { ... }

@Override
public void disposeUIResources() { ... }
}

```

Listing 4: Sample Configurable Implementation.

Listing 4 illustrates the major functionality that any Configurable implementation must provide. The *createComponent* is used to create the UI component that is going to be displayed to the IDE users in the Settings → Tools → <<tool name>> window. IDE users can use the created UI to read and change the plugin configuration. The <<tool name>> value in the UI control path is populated from the *applicationConfigurable* XML element's *displayName* attribute in the plugin.xml file. Any Configurable implementation is expected then to provide implementation for three other methods: *isModified*, *apply*, and *reset*. The UI component created by the *createComponent* method that is displayed to the IDE user to read/edit the plugin settings is displayed within a container window that provides the IDE user with two buttons: *Apply* and *Reset*. If the *isModified* method returns *True* then the two buttons are enabled. Otherwise, they will be disabled. When the user clicks on the *Apply* button, the *apply* method in the Configurable class is called. Similarly, if the user clicks on the *Reset* button, the *reset* method in the Configurable class is called.

In order to persist the plugin configuration object on a file, another class that implements the *PersistentStateComponent* interface must be defined and annotated with the *@State* annotation. Listing 5 contains a sample implementation of the *PersistentStateComponent* interface.

```
@State(  
    name = "impl.settings.Settings",  
    storages = {@Storage("Settings.xml")}  
)  
public class Settings implements  
PersistentStateComponent<Settings> {  
    @Nullable  
    @Override  
    public AppSettings getState() { ... }  
  
    @Override  
    public void loadState(@NotNull AppSettings state) { ... }  
}
```

Listing 5: Sample *PersistentStateComponent*.

The *@State* annotation contains two attributes that indicate to the IntelliJ platform the fully-qualified name of the class that will be serialized and persisted to the persistent storage and the file name on which the serialized class contents will be written.

### 5.1.3 IntelliJ PSI

The IntelliJ Platform exposes an important layer called the Program Structure Interface (PSI) [55] which is a comprehensive layer that abstracts the parsing of files and project model using interfaces. It contains abstractions to many of the aspects of the IDE including interfaces to navigate the project structure and the syntax tree of any code file. In addition, this layer is responsible for creating the syntactic and semantic models of code files. This layer delegates to the actual implementation of each of its interfaces. Figure 13 provides a simplified version of a part of the PSI model.



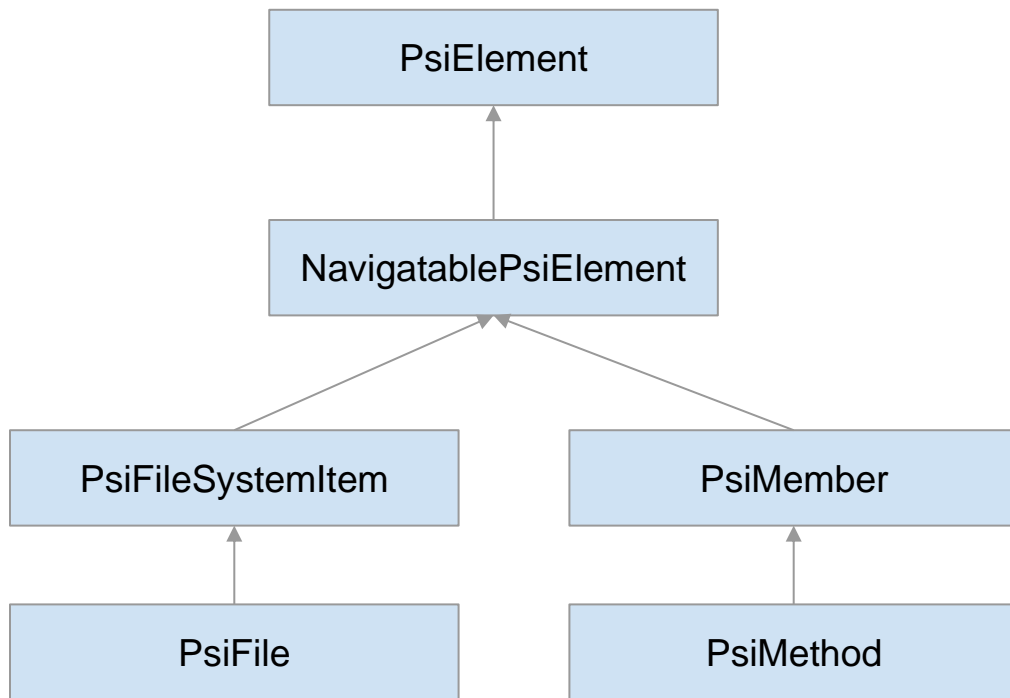


Figure 13: IntelliJ Platform PSI Model.

It can be observed from Figure 13 that the model is able to represent a wide variety of entities in any project such as files and methods. Although the PSI model provides direct access to the underlying implementation, it is highly advisable that any plugin built on top of the IntelliJ platform is developed against the PSI model as this will make the plugin IDE-neutral and cross-platform.

#### ***5.1.4 IntelliJ Indexing Framework***

Another important component of the IntelliJ platform is its indexing framework which provides quick access to specific elements in the project such as files that contain specific words or methods with a particular name [59]. Accessing built indexes has a significant performance gain when searching for files. The indexing framework creates two operating modes for the IntelliJ platform: Dump and Smart. The Dump Mode is when the IntelliJ platform has not yet evaluated all the indexes and as a result the platform features are restricted to the ones that don't depend on the indexes. The Smart Mode is active once all indexes are built and ready for use. The IntelliJ platform provides the DumpService class

to manage access to the indexing framework. Listing 6 illustrates a sample usage of the `DumbService` to access the indexing framework of the IntelliJ platform.

```
DumbService
    .getInstance(project)
    .runWhenSmart(() -> {
        ...
    });
```

Listing 6: A snippet how the IntelliJ Platform Indexing Framework is used.

The method passed as argument to the `runWhenSmart` will not be called until all platform indexes are built and ready for use.

## 5.2 Plugin Architecture

This section discusses the concrete architecture of our plugin in the context of the generic architecture presented previously and the design practices of the IntelliJ platform. Figure 14 illustrates the concrete plugin architecture that conforms to the IntelliJ platform design principles.

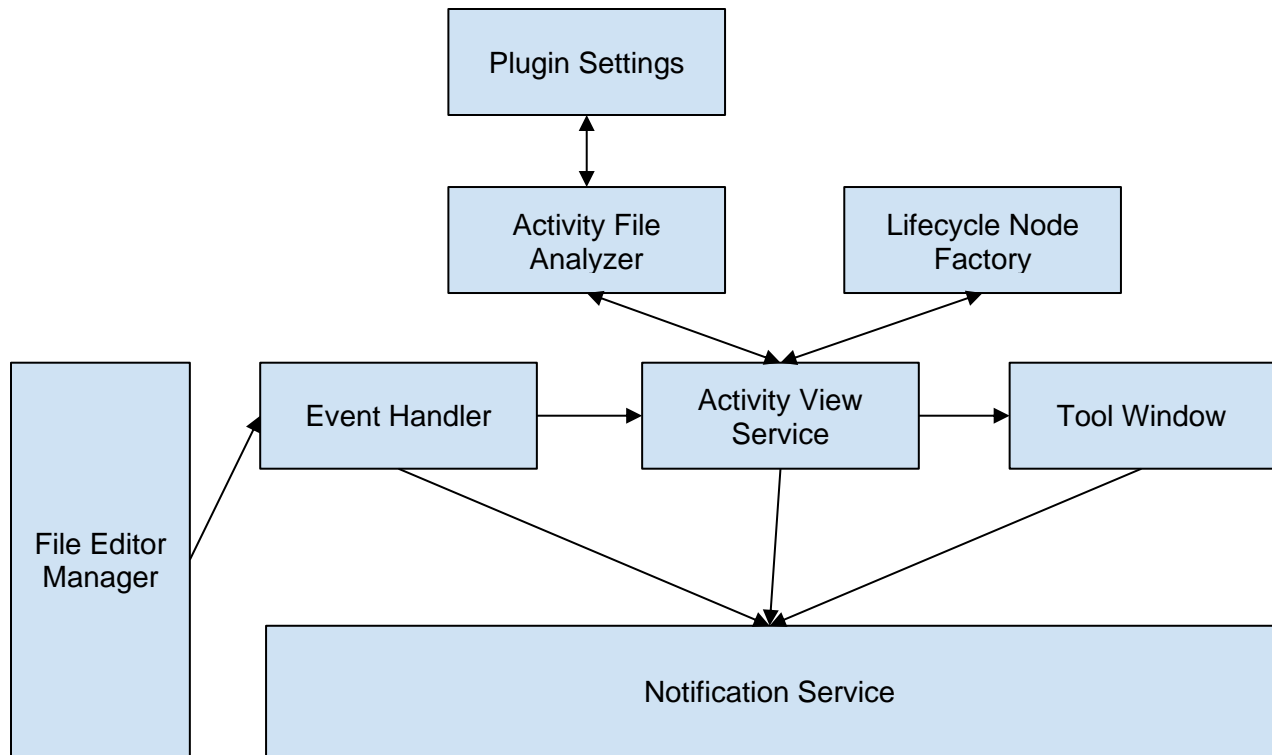


Figure 14: IntelliJ Platform Plugin Architecture.

### 5.2.1 FileEditorManager

The FileEditorManager class in the IntelliJ platform is the class responsible for all UI-based file operations (e.g. open, select, close, edit, save) that the IDE user interacts with. The FileEditorManager class dispatches three types of file editing messages on the platform message bus: **fileOpened**, **fileClose**, and **selectionChanged**. The previous messages can be received by any handler that implements the **FileEditorManagerListener** interface and has been registered on the platform message bus to listen for messages sent by the FileEditorManager.

### 5.2.2 EventHandler

The **FileEditorManagerListener** listener interface was implemented in an anonymous class and registered on the message bus to listen for messages that are sent by the FileEditorManager. The anonymous class, upon receiving a message sent on the bus by the FileEditorManager, passes the message to a class called

**FileEditorManagerEventHandler** that is responsible for the following before passing the message to the inner components of the plugin:

1. Making sure that the file is in a valid state.
2. Making sure that the file opened by the user (i.e. developer) is an Activity file. This is done by locating the **AndroidManifest.xml** and searching for an Activity file registered with the same name as the file opened, selected, or closed.
3. Invokes the inner components of the plugin when the IDE is running in smart mode (i.e. when all the indexes that the IDE uses have been built).

Listing 7 illustrates the anonymous message bus listener that has been registered to listen on the IntelliJ platform message bus in order to process messages sent by the FileEditorManager.

```
project
    .getMessageBus()
    .connect()
    .subscribe(
        FileEditorManagerListener.FILE_EDITOR_MANAGER,
        new FileEditorManagerListener() {
            @Override
            public void fileOpened( ... ) { ... }

            @Override
            public void fileClosed( ... ) { ... }

            @Override
            public void selectionChanged( ... ) { ... }
        });
```

Listing 7: FileEditorManager Listener.

### 5.2.3 Activity View Service

The Activity View Service component is the main component responsible for maintaining the DSL model of the Activity file and handling all events that are fired by the nodes in

the visual model such as the event to expand a node, adding a callback method and navigating to a callback method. The Activity View Service depends on three components to perform its functions: Activity File Analyzer, Lifecycle Node Factory, and the Tool Window. The Activity View Service component uses the DSTL to DSVL Converter algorithm to convert the DSTL model to a DSVL instance that can be rendered in a tool window.

#### ***5.2.4 Activity File Analyzer***

The Activity File Analyzer is the component responsible for analyzing the Activity file, parsed as a PSI element (see 5.1.3 IntelliJ PSI), and producing an instance model that conforms to the DSTL. The current implementation of the plugin uses an implementation of the analyzer algorithm presented in Figure 3: Code Analysis Algorithm that visits the subtrees inside the Activity class that start at the callback methods. The analyzer visits the entire subtree of each of the callback methods looking for method calls that are identified as either a resource acquisition or resource release. For each method call in the subtree, whether it is a static method call or instance method call, the analyzer produces a string that consists of the fully qualified class name and the method name and matches the result to the resource allocations and releases that are defined in the plugin settings.

#### ***5.2.5 Plugin Settings***

The plugin settings contain two sections that define the method calls that are recognized as either a resource allocation or resource release. Each entry in the settings file is defined as a key-value pair where the key is the name of the resource and the value has the following format: <fully-qualified-class-name>.<method name>. Figure 15 captures the default settings for resource allocations and releases.

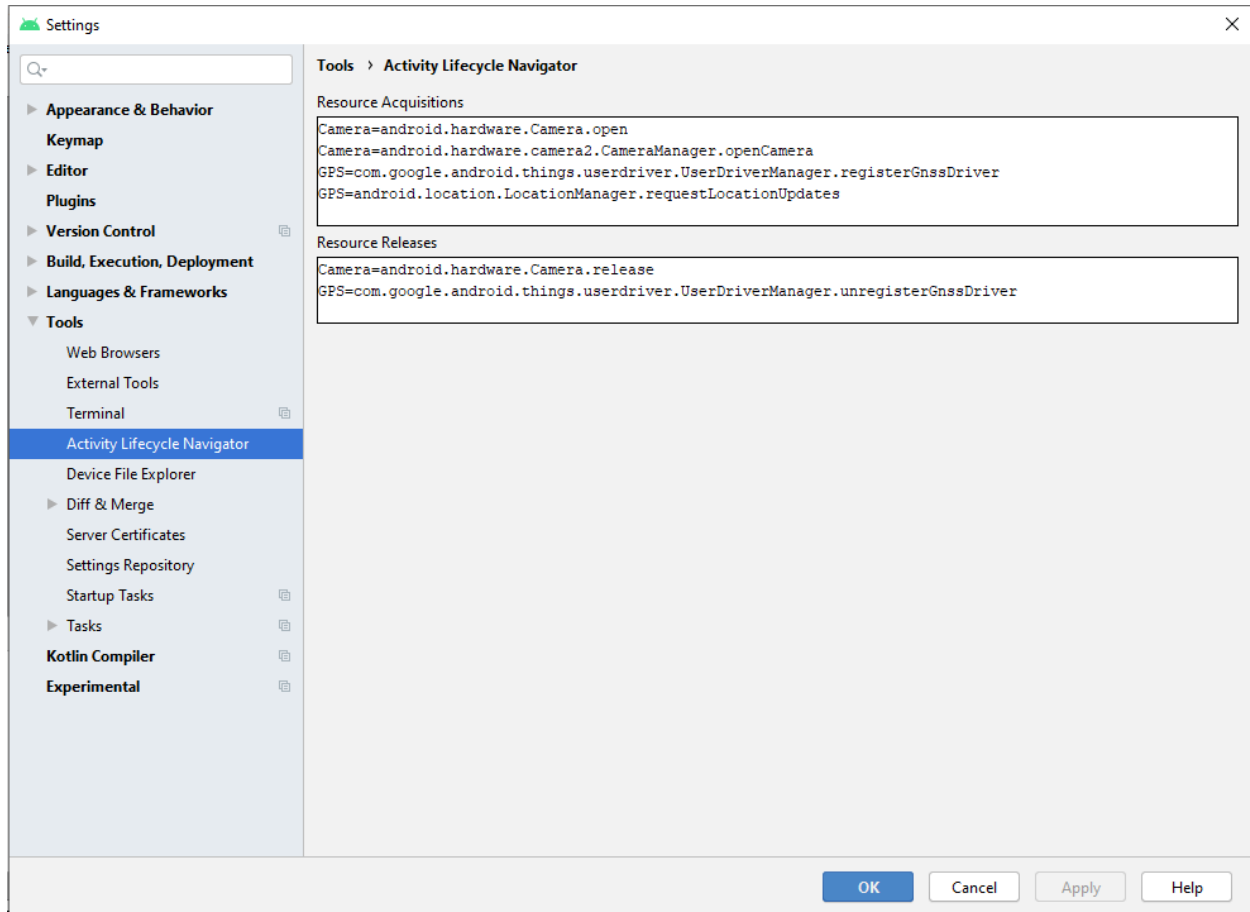


Figure 15: Android Studio Plugin Settings.

### 5.2.6 Lifecycle Node Factory

The LifecycleNodeFactory is the component responsible for creating the DSVL model nodes that are used to build the DSVL model instance from an instance of the DSTL model.

### 5.2.7 Tool Window

The implemented tool window in the plugin is used to render the DSVL model instance that represents the Activity lifecycle implementation. The tool window is a tabbed window where each tab represents an opened Activity file in the FileEditorManager.

### 5.2.8 Notification Service

The plugin, in this current version, follows the fail-fast principle which is whenever an exception or an error occurs, the current processing thread fails immediately and returns

back to the FileEditorManagerEventHandler anonymous class. Any failure in the current processing thread is reported as-is using a balloon notification dialog in the IDE. In addition, the failure is logged to the IDE event log.

## **Chapter 6: Evaluation**

In order to understand the extent by which the presented approach can assist Android developers implement the Activity lifecycle (answer RQ2), a user evaluation of the Android Studio plugin was conducted. The primary aim of this user evaluation is to capture the user (i.e. developer) satisfaction of having a second view of the Android Activity lifecycle implementation.

### **6.1 Case Study Setup**

Each participant was asked to do the following:

1. Fill a demographic survey which collects the participant's background experience on Android mobile app development.
2. Participate in a session to get introduced to the approach and the plugin that implements the approach.
3. Develop a simple Android application with the help of the plugin.
4. Fill a post-case-study survey to collect information and insights about the personal experience of using the plugin.

### **6.2 Participants**

For the user evaluation of the plugin, we targeted at least 6 Android app developers. To recruit participants for the case study, several approaches were used to contact Android app developers. First, the researchers used personal contacts to reach out directly and indirectly, through intermediate contacts, to Android app developers. Both regular phone and social media communications were used to communicate with the developers. A discussion group was created to share news and updates between all the developers. The developers who couldn't join the group were updated individually through social media. The discussion group was used to arrange for the best time an introductory session was to be held. Also, the group was used to answer any questions the developers had either about



the topic of the case study or about the arrangements that were being made for the case study. During the case study, the group was used to track the progress of developers. In addition to the discussion group, the developers were contacted individually to answer any specific questions that could not be discussed on the discussion group. The discussion group and individual communications were intended to reduce any bias introduced to the case study from the developers being unaware of the nature of the case study or any of its phases and rules.

In total, 10 Android app developers were communicated both directly and indirectly. Four developers did not fully complete the case study for several reasons: they didn't want to participate, could not guarantee to be able to commit to the case study, did not respond to the researchers message, and started to get introduced to the case study but did not continue in the later phases where an Android app was needed to be developed and post-case-study questionnaire was to be filled.

Only 6 participants participated in the discussion group, responded to emails, filled the demographic survey, developed the sample Android app, and filled the post-case-study questionnaire to report their satisfaction about the concept of a multi-view based approach to assist lifecycle development.

### **6.3 Case Study App**

The participants were asked to develop a simple Android application which consisted of a single Android Activity where the app user can press and hold on a screen button to capture a video clip. Once the user releases the button the app would then capture the current device position and upload the bundle that consists of the video clip and the geographical location to an online web service. The developers were asked to create a fake adapter for the online web service in their app code in order to reduce the complexity involved in developing the app. In addition, the developers were not asked to implement any special UI graphics for

the app. Further still, the participants were not required to complete the development of the app and produce a fully working application.

It can be observed that the app requirements were loosened too much in order to keep the participants motivated. The case study was focused on the development experience provided by the Android Studio plugin which provides a second view for the Android Activity.

## Chapter 7: Results

In order to answer RQ2 and understand the extent by which the presented approach affected the development of an Android mobile app and the handling of Activity lifecycle by the developers, the evaluation aimed to collect the following data: understandability, accuracy, usability, and satisfaction of the presented approach through a post-case-study questionnaire that was filled by each participant. The answers to the demographic survey and post-case-study questionnaire were as the following:

### 7.1 Demographic Survey

The demographic Survey was intended to obtain background information about the case-study participants. This data can be further used to extract potential relationships between different demographic data and satisfaction data reported in the Post-Case-Study Questionnaire.

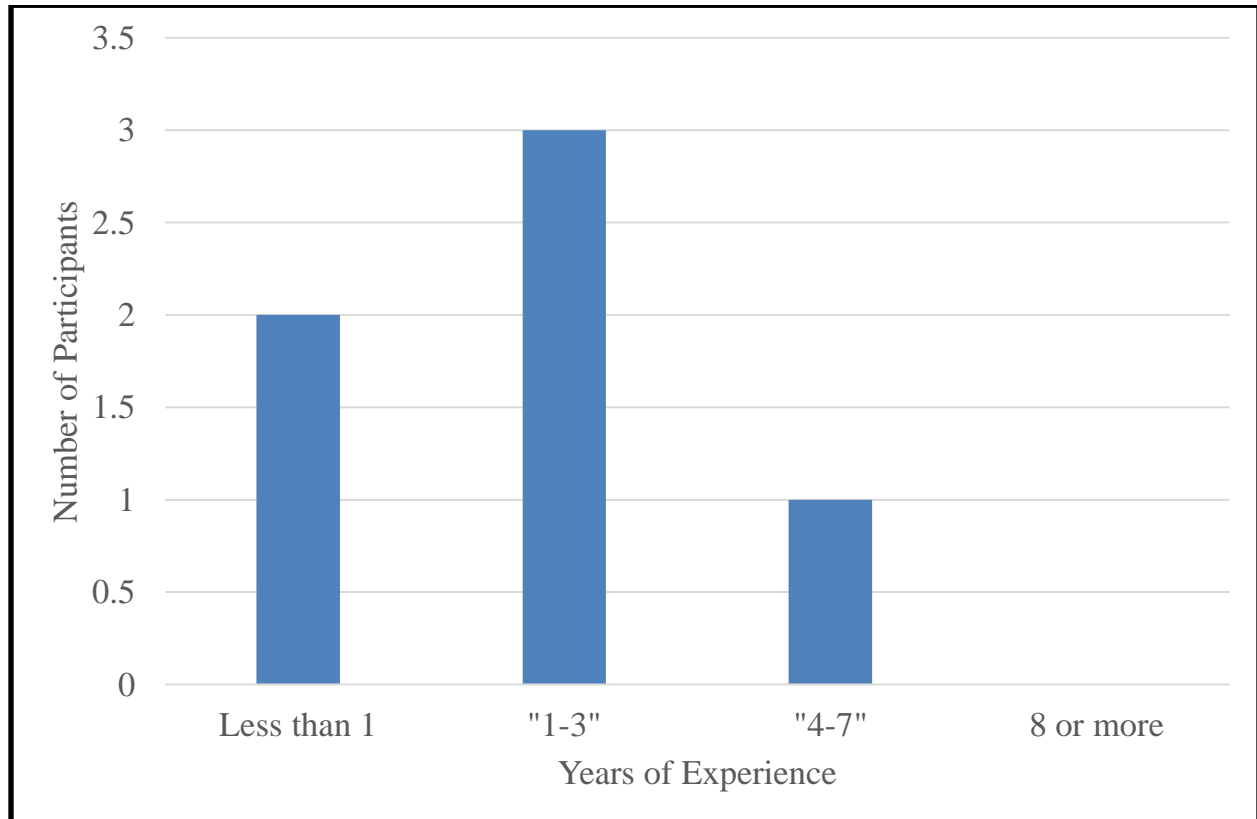


Figure 16: The number of participants in each years of experience range that successfully completed the case study.

From Figure 16, it can be observed that the majority of the participants of the case-study had 1-3 years of experience in mobile app development. 2 developers have less than one years of experience and 1 developer has 4-7 years of experience.

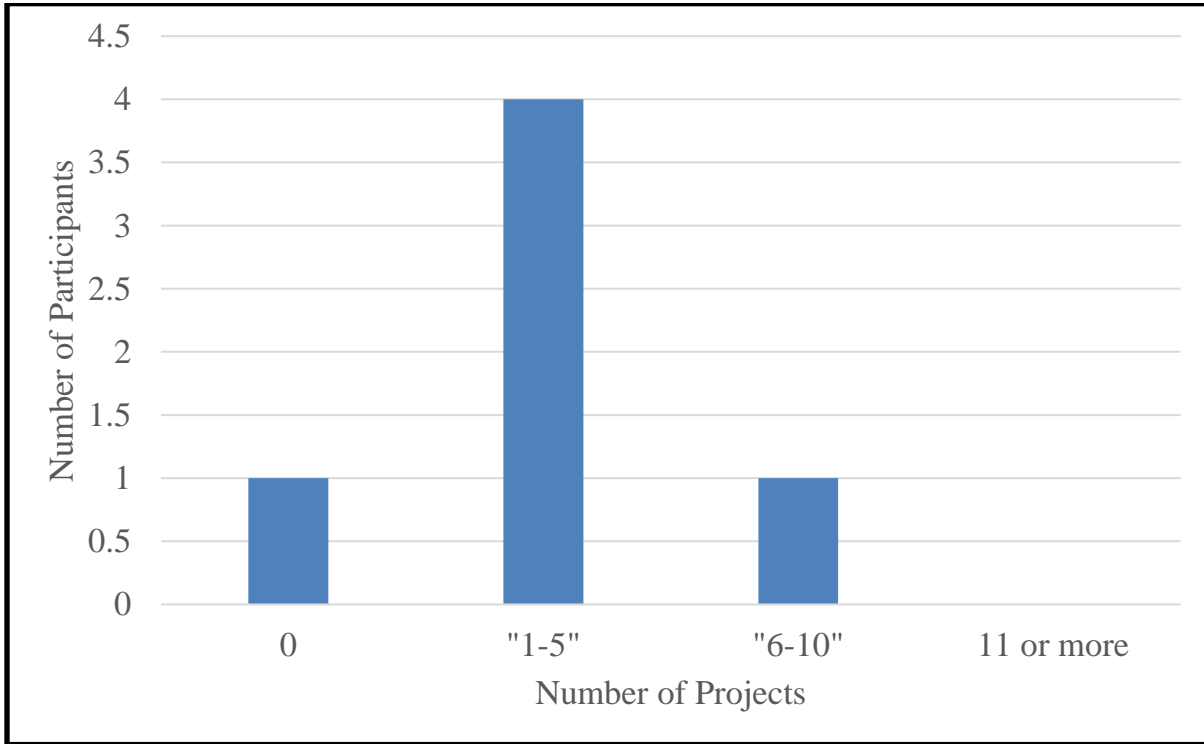


Figure 17: The number of participants per the number of projects contributed to that the case study included.

The participants' years of experience in mobile app development aligned closely with the number of Android apps that each participant have developed. It can be seen that 4 participants have contributed to 1-5 number of Android apps while 1 developer has contributed to none and another contributed to 6-10 apps.

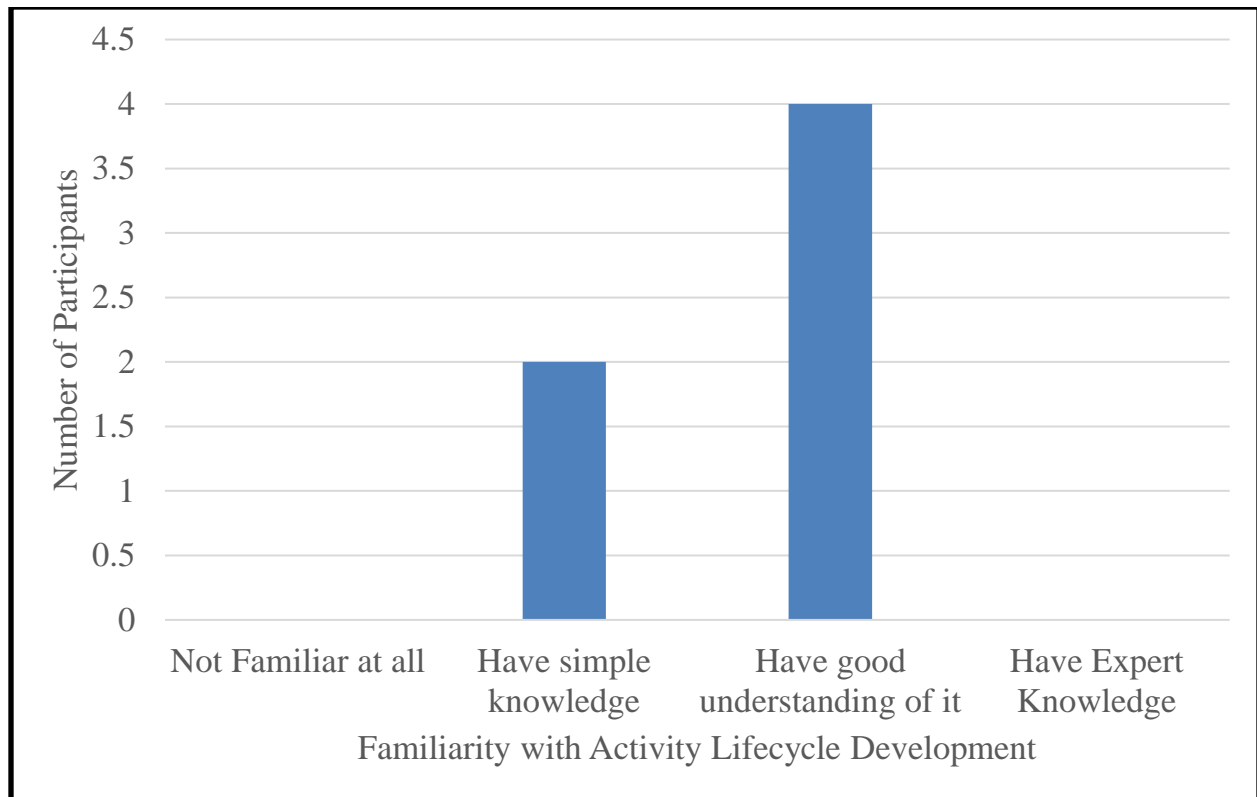


Figure 18: The number of participants per their Activity lifecycle development familiarity that the case study included.

All the participants of the case-study knew about the Activity lifecycle callback methods with 2 developers having simple knowledge about it and 4 developers possessing a good understanding. Having simple knowledge means that the developer knows what the callback methods are and what they are being used for. Having a good understanding of the Activity lifecycle means that the developer understands the basic rules about when the state of the Activity changes, what callback methods get called and as a result what should be implemented in each callback method.

It can be observed from the above survey results that the case-study included a good range of mobile app development experience. However, the case-study lacked experienced developers who are experts in Android app lifecycle development, have 8 or more years or experience in mobile app development, and contributed to 11 or more Android apps in their years of experience.

Additionally, further insight could be obtained by comparing the answers to the survey questions. Most of the participants who answered that they have 1-3 years of experience in mobile app development also reported that they have contributed to 1-5 Android apps in those years of experience and that they have a good understanding of handling the Activity lifecycle callback methods. One participant who answered that he had 4-7 years of experience and contributed to 6-10 Android apps also considered that he had a good understanding of handling the Activity lifecycle callbacks. Finally, the two participants who had Less than 1 years of experience, both had simple knowledge of the Activity lifecycle handling; one contributed to 1-5 and the other to 0 Android apps in his years of experience. Table 3 summarizes the previous insight into groups to be used in the analysis of the results.

<b>Group ID</b>	<b>Years of Experience</b>	<b>Project Contribution Count</b>	<b>Activity Lifecycle Familiarity</b>	<b>Number of Participants</b>
A	1-3	1-5	Good Understanding	3
B	4-7	6-10	Good Understanding	1
C	Less than 1	0	Simple Knowledge	1
D	Less than 1	1-5	Simple Knowledge	1

Table 3: Participants Groups

It can be observed that there is a relationship between the number of experience years and the number of Android apps contributed to and the overall understanding of the Android Activity lifecycle handling.

## 7.2 Post-Case-Study Questionnaire

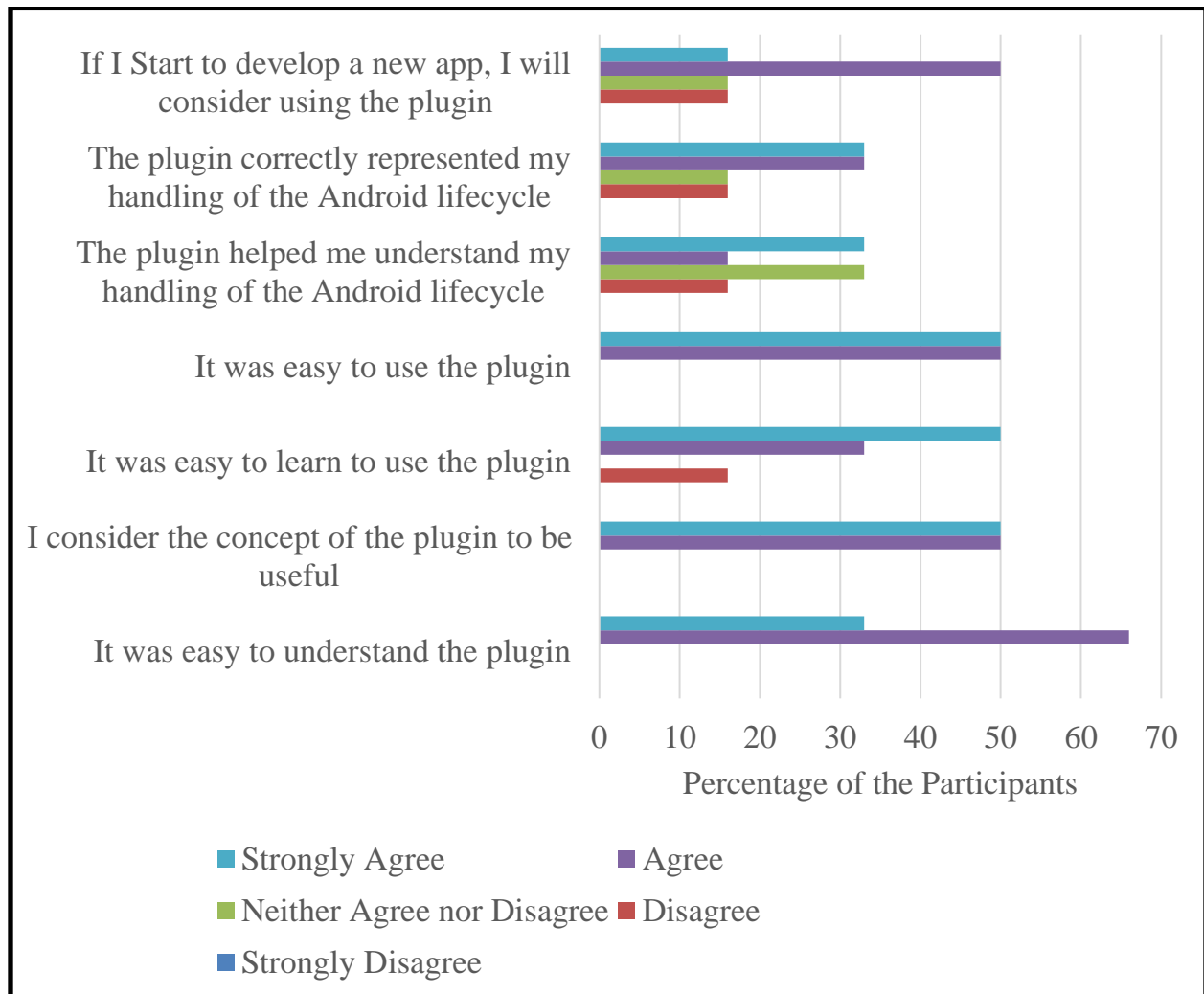


Figure 19: Post-Case-Study Questionnaire Results.

When asked about whether they will consider using the plugin in future apps, 50% of the participants agreed that they will. Around 33% of the participants considered that the plugin correctly represented their handling of the Activity lifecycle. The participants didn't report conclusive results when asked about whether the plugin helped them understand their handling of the Android lifecycle; 33% of the participants Strongly Agreed while 33% Neither Agree nor Disagree and one participant disagreed. The participants agreed that learning, understanding, and using the plugin were easy tasks. Only one participant disagreed that it was easy to use the plugin. Finally, the participants agreed that the concept of the plugin was a useful concept.

Additional insights can be obtained by comparing the answers to the questionnaire with the answers to the survey. Using the groups from Table 3, it is observed that Group A agreed with varying strength that it was easy to learn, understand and use the plugin and that the plugin correctly represented their handling of the Activity lifecycle and helped them understand it. Furthermore, this group considered the concept of the plugin to be useful and it will be considered when they start developing a new Android app. When asked about the issues they had while using the plugin, the answers from this group were as follows. One of the participants reported: *“It (the plugin) crashed a couple of times, I had to restart the activity or restart android studio”*, *“Some methods used in various lifecycle methods were not shown in the plugin.”*, *“Plugin doesn’t refresh immediately after implementing a new lifecycle method.”*, *“Recursive nodes are a bit confusing, in my opinion, I’d like to see the recursion instead of showing a new node that is faded.”*. The other participant didn’t report any issue when he used the plugin. When asked to suggest changes to be made to the plugin, the first participant in this group stated *“Double click to go to method implementation. This makes navigation between methods faster.”*, *“Add an option named ‘break here’ where it runs the application in debug mode and hit a breaking point in the specified state.”*, *“During debugging, highlight the current state of the Activity. For example, if a breakpoint is hit inside a method that is called from any of the callback methods, highlight the corresponding node in the lifecycle view. This allows seeing the current state of the application without needing to review the stack trace.”*, *“Add an option to expand all the nodes in the graph in one click. Navigating through multiple unopened nodes is counterproductive and distracting.”*, *“I sometimes like to write To Dos inside a specific lifecycle method. It would be nice if the tool allows me to see the To Do comments and maybe click them for navigation.”*, *“Global state is needed in an Activity sometimes. It is helpful if I have the option to see this global state and how it is changed throughout the lifecycle of my Activity.”*, *“Add an option to implement all callback methods in the Activity class.”* The other participant reported *“Support for Fragment lifecycle.”* and *“Use colors to show issues in the life cycle.”* A third participant in this group was inconclusive on



whether the plugin correctly represented his handling of the Activity lifecycle and consequently helped him understand it. This participant was not sure that he will consider using the plugin when developing future Android apps. When asked about the issues he had while using the plugin, this participant reported that *“It does not show correct behavior when resource allocations/releases function calls are nested inside other methods.”* When asked to suggest changes to the plugin, he reported *“Warning of memory leaks, unclosed stream, and freeing of mobile resources.”*

Group B of the participants agreed with a varying strength that it was easy to learn, understand and use the plugin, the plugin correctly represented their handling of the Activity lifecycle and that they will consider using it in future apps. However, this group was inconclusive about whether the plugin helped them understand their implementation of the Activity lifecycle callback methods. One participant in this group had voluntarily provided additional comments about the questions in the questionnaire and reported that *“The concept behind the plugin is useful for complex activities, (but) for simple activities, it seems unnecessary. I do not think an advanced developer will look for this diagram unless either the activity is too complex, or more features to be introduced.”* and *“For advanced developers, the plugin is not very useful in helping them understand their handling of the Android lifecycle because they already understand the lifecycle. However, it will help beginner and intermediate developers.”* When asked about the issues he had while using the plugin, this participant reported *“I had to re-open the Activity class every time I do a change to see the change reflect on the diagram.”* *“Otherwise all good”*, he added. When asked to suggest changes to the plugin, he reported *“Alert on undisposed resources”*, *“Keep track of resources within other classes, packages.”*, *“Xml configuration to define resources”*, *“Support onRestoreInstanceState and onSaveInstanceState.”*

Group C considered that the concept of the plugin was useful and it was easy to understand and use the plugin. However, for the remaining questions in the questionnaire, a participant from this group reported his answers depending on the “version” of the plugin. By

“version”, the participant meant the number of features and options that are available in the plugin as a result of the number of years that the plugin was being developed and maintained. For the “current” version of the plugin, which was a proof-of-concept, the participant disagreed that it was easy to learn the plugin and that the plugin correctly represented his lifecycle implementation and helped him understand it. As a result, the participant considered that he disagrees to use the “current version” of the plugin in any future Android app development. On the other hand, this participant reported that his “disagree” answers will be “agree” if he was using a “future version” of the plugin. By “future version”, the participant was referring to an imaginary version of the plugin where many features and options exist and the plugin provides so much information and insights that he didn’t know about his Activity lifecycle handling. When asked about the issues he had while using the plugin, he reported *“Did not see warning when used camera from button handler.”* When asked to suggest changes that should be made to the plugin, he reported *“Issue warning when using the camera and all the possible resources. Warnings should be complete to show the correct location of the resource allocation/release”* and *“Add support to correct the incorrect usage from the view.”*

Group D answered with “Strongly Agree” on all of the questionnaire questions except for the question about whether it was easy to understand the plugin where he answered with “Agree”. When asked about the issues they had while using the plugin, one participant reported *“I had to understand the activity life cycle of the android app to use it”*. And when asked to suggest changes to the plugin, he reported *“The plugin should give more information about the Activity lifecycle; why we should handle every method and what are the cases that the app goes into mapped to those methods.”*

## Chapter 8: Discussion

From the collected information in the post-case-study questionnaire, the following points can be seen.

1. The participants' satisfaction ranged from Disagree to Strongly Agree.
2. The majority of the participants considered the concept of a “multi-view” approach for an Activity lifecycle implementation to be useful and that the plugin that implemented the concept was easy to understand, learn and use. Only one participant reported that learning the plugin was not an easy task.

The reason that some of the participants didn't consider or couldn't decide whether the plugin helped them understand their handling of the Activity lifecycle, correctly represented their handling of the Activity lifecycle, or will consider using it in future applications can be inferred from their answers to the open-ended questions:

1. From the participants' answers to the first open-ended question which is about the issues they had while using the plugin, the participants' responses were either pointing to a bug, missing features, or unexpected behavior. In specific, the participants indicated that the plugin crashed a couple of times where they had to restart Android Studio, resource allocation/releases that occurred in nested methods (inside the callback methods) are not shown, and the recursive nodes in the view were confusing and should rather be replaced by a line back to the target node.
2. From the participants' answers to the second open-ended question where they had to suggest changes to be made to the plugin, it can be observed that developers expected the plugin to contain so many features: from warning of memory leaks, context information about each callback method, including TODO comments in each life cycle method, tracking resources in other classes, packages...etc.

It can be inferred then from the answers to the two open-ended questions that Android developers expect that any plugin or tool that they use be fully-featured and bug-free in order to be useful for them. The voluntary comments provided by one of the participants and the other participant who had different answers depending on the “version” of the plugin both confirm the conclusions that were drawn from analyzing the responses to the open-ended questions and comparing them with the previous questions.

Additionally, it can be concluded from the results that there is an opposite relationship between the level of experience for developers and their tendency to depend on tools and plugins that assist them implement an Android Activity lifecycle. It can be observed that the participants from Groups A and C (with levels of experience of 1-3, and less than 1, respectively) positively agreed with all the criteria about the plugin. However, the participants from Group B, although positively agreed with the approach, yet provided a feedback that indicates that this approach may not be useful for experienced developers. From these observations and feedback, we conclude that the more experienced an Android developer, the less dependent they are on any approach to assist them in implementing an Android Activity lifecycle.

Finally, and going back to the research questions presented at the beginning of this research, we conclude that the concept of presenting a second view to Android developers about their Activity lifecycle handling, in addition to the code view (RQ1), is a useful idea. We also conclude that any implementation that implements the concept of a second view of the Activity lifecycle handling should contain many features and options and be bug-free in order to be endorsed and adopted by Android developers (R2). Furthermore, the more experienced an Android developer, the less dependent they are on any tool or plugin to assist them implement an Android Activity lifecycle.

## **8.1 Threats to Validity**

### ***8.1.1 Internal Validity***

The participants recruited for the case study have been recruited from software engineers at Zeva International<sup>2</sup>. Their participation may have been biased in their responses because they knew the researchers which may have affected their responses. In order to remediate this threat, the researchers indicated to the participants that they should be as honest as possible in their responses to the questionnaire and that their responses whether positive or negative will be considered as empirical data and that the researchers have no interest in a specific type of response.

The majority of the participants have less than 7 years of experience, contributed to less than 10 Android apps in their experience and have a good understanding of the Activity lifecycle model. The case study didn't include participants who developed more than 8 apps in their career, contributed to more than 11 or are experts in handling the Activity lifecycle. This has the effect of making the results of the case study biased towards novice and intermediate level Android developers.

### ***8.1.2 External Validity***

The number of participants that were involved in the case was 6 which was sufficient to draw the conclusions that we needed but not sufficient to draw reliable results. The participants were not required to develop a complete Android app and the software requirements were loosened as much as possible to keep the participants motivated to finish the case study. This has the effect of not reflecting the real nature of Android app development where developers are required to build complete and bug-free apps and typically for a long period of time. If the plugin had been evaluated in a real environment that involved a large number of Android developers for an extended period of time and

---

<sup>2</sup> Zeva International is a Palestinian software development company located in Bethlehem, Palestine. <https://www.zevainc.com/>

where Android apps contain many Activity classes that are potentially complex, the results of the case study would have been more statistically reliable.

## **Chapter 9: Conclusion**

This research presented an approach that aims at aiding Android app developers visualize their Activity lifecycle handling. This additional view of the Activity lifecycle handling provides the developers with additional insight on their implementation and aids them to discover potentially unhandled state transitions that may cause issues and crashes with their app. The approach at its core depends on domain-specific languages to represent and visualize the Activity lifecycle implementation. Additionally, the approach presents an architectural design that can be recycled for any mobile app development technology, including Android. In order to evaluate the approach, an Android specific implementation was developed as an Android Studio plugin and a case-study was conducted wherein Android developers were asked to develop an Android app with the plugin at hand. The developers' satisfaction with the approach was collected using a questionnaire. It has been observed that a "multi-view" approach was a useful idea. However, in order for such an approach to be fully endorsed and adopted by Android developers, it has to provide so much information and many options and be as bug-free as possible.

The results obtained in this research provide an invaluable insight on the use of multi-view-based approaches to assist Android Activity lifecycle development. However, these results were collected from a relatively small sample of Android developers such that they cannot be used for statistical reliability tests. Therefore one of the possible future works that extend this research is to perform the same case study but on a larger sample of Android developers. Another extension of this work could be to try to fix the issues and implement the suggested features reported in this research and perform the case study again using a different sample of Android developers and evaluate whether the reported issues and suggested features increase developer satisfaction and consequent adoption of this approach. A more advanced future work might involve implementing the evaluation method in a real Android app development environment where real apps are being

developed within real deadlines and strict requirements. This work could also be extended to target other mobile app development technologies such as iOS, UWP and Xamarin and investigate whether similar results are reported by mobile app developers that develop for other platforms.



## References

- [1] H. Tufail, F. Azam, M. W. Anwar, and I. Qasim, "Model-driven development of mobile applications: A systematic literature review," in *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2018, pp. 1165–1171.
- [2] D. Amalfitano, V. Riccio, P. Tramontana, and A. R. Fasolino, "Do Memories Haunt You? An Automated Black Box Testing Approach for Detecting Memory Leaks in Android Apps," *IEEE Access*, vol. 8, pp. 12217–12231, 2020.
- [3] F. Toffalini, J. Sun, and M. Ochoa, "Practical static analysis of context leaks in Android applications," *Softw. Pract. Exp.*, vol. 49, no. 2, pp. 233–251, 2019.
- [4] N. Hoshieah, S. Zein, N. Salleh, and J. Grundy, "A static analysis of android source code for lifecycle development usage patterns," *J. Comput. Sci.*, vol. 15, no. 1, pp. 92–107, 2019.
- [5] V. Riccio, D. Amalfitano, and A. R. Fasolino, "Is this the lifecycle we really want? an automated black-box testing approach for Android activities," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018, pp. 68–77.
- [6] A. D. Reference, "Activity." 2020, [Online]. Available: <https://developer.android.com/reference/android/app/Activity>.
- [7] A. D. Reference, "Activity." 2020.
- [8] D. D. Perez and W. Le, "Specifying callback control flow of mobile apps using Finite Automata," *IEEE Trans. Softw. Eng.*, 2019.
- [9] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *2012 7th International Workshop on Automation of Software Test (AST)*, 2012, pp. 29–35.
- [10] A. D. Reference, "Activity Lifecycle." 2020, [Online]. Available: <https://developer.android.com/guide/components/activities/activitylifecycle>.
- [11] A. Nirumand, B. Zamani, and B. Tork Ladani, "VAnDroid: A framework for

- vulnerability analysis of Android applications using a model-driven reverse engineering technique,” *Softw. Pract. Exp.*, vol. 49, no. 1, pp. 70–99, 2019.
- [12] M. Junaid, D. Liu, and D. Kung, “Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models,” *Comput. Secur.*, vol. 59, pp. 92–117, 2016.
- [13] Y. Li, J. Ouyang, S. Guo, and B. Mao, “Data Flow Analysis on Android Platform with Fragment Lifecycle Modeling,” in *International Conference on Security and Privacy in Communication Systems*, 2016, pp. 637–654.
- [14] M. Junaid, J. Ming, and D. Kung, “StateDroid: Stateful Detection of Stealthy Attacks in Android Apps via Horn-Clause Verification,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 198–209.
- [15] Y. Shao, R. Wang, X. Chen, A. M. Azab, and Z. M. Mao, “A Lightweight Framework for Fine-Grained Lifecycle Control of Android Applications,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–14.
- [16] S. Zein, N. Salleh, and J. Grundy, “Static analysis of android apps for lifecycle conformance,” in *2017 8th International Conference on Information Technology (ICIT)*, 2017, pp. 102–109.
- [17] C. Guo, Q. Ye, N. Dong, G. Bai, J. S. Dong, and J. Xu, “Automatic construction of callback model for Android Application,” in *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2016, pp. 231–234.
- [18] F. Freitas and P. H. M. Maia, “Justmodeling: An mde approach to develop android business applications,” in *2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2016, pp. 48–55.
- [19] S. Vaupel, G. Taentzer, R. Gerlach, and M. Guckert, “Model-driven development of mobile applications for Android and iOS supporting role-based app variability,” *Softw. Syst. Model.*, vol. 17, no. 1, pp. 35–63, 2018.
- [20] C. Rieger and H. Kuchen, “A process-oriented modeling approach for graphical development of mobile business apps,” *Comput. Lang. Syst. Struct.*, vol. 53, pp. 43–58, 2018.

- [21] E. E. Thu and N. Nwe, "Model driven development of mobile applications using drools knowledge-based rule," in *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*, 2017, pp. 179–185.
- [22] S. Barnett, I. Avazpour, R. Vasa, and J. Grundy, "Supporting multi-view development for mobile applications," *J. Comput. Lang.*, vol. 51, pp. 88–96, 2019.
- [23] T. Ghanem and S. Zein, "A Model-based approach to assist Android Activity Lifecycle Development," in *2020 4th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, Oct. 2020, pp. 1–12, doi: 10.1109/ISMSIT50672.2020.9254687.
- [24] A. D. Reference, "App Manifest Overview." 2020, [Online]. Available: <https://developer.android.com/guide/topics/manifest/manifest-intro>.
- [25] A. D. Reference, "Activity Lifecycle." 2020.
- [26] Android, "Understanding Tasks and Back Stack." 2020, [Online]. Available: <https://developer.android.com/guide/components/activities/tasks-and-back-stack>.
- [27] Android, "Processes and Application Lifecycle." 2020, [Online]. Available: <https://developer.android.com/guide/components/activities/process-lifecycle>.
- [28] D. C. Schmidt, "Model-driven engineering," *Comput. Comput. Soc.*, vol. 39, no. 2, p. 25, 2006.
- [29] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Future of Software Engineering (FOSE'07)*, 2007, pp. 37–54.
- [30] H. Gomaa, *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, 2011.
- [31] S. Biffli, E. Mätzler, M. Wimmer, A. Lüder, and N. Schmidt, "Linking and versioning support for AutomationML: A model-driven engineering perspective," in *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, 2015, pp. 499–506.
- [32] M. Lachgar and A. Abdali, "Modeling and generating native code for cross-platform mobile applications using DSL," *Intell. Autom. Soft Comput.*, vol. 23, no. 3, pp. 445–

- 458, 2017.
- [33] M. Lachgar and A. Abdali, "Rapid Mobile Development: Build Rich, Sensor-Based Applications using a MDA approach," *IJCSNS*, vol. 17, no. 4, p. 274, 2017.
  - [34] H. Benouda, M. Azizi, M. Moussaoui, and R. Esbai, "Automatic code generation within MDA approach for cross-platform mobiles apps," in *2017 First International Conference on Embedded & Distributed Systems (EDiS)*, 2017, pp. 1–5.
  - [35] H. Benouda, M. Azizi, R. Esbai, and M. Moussaoui, "Code generation approach for mobile application using acceleo," *Int. Rev. Comput. Softw.*, vol. 11, no. 2, pp. 160–166, 2016.
  - [36] H. Benouda, M. Azizi, R. Esbai, and M. Moussaoui, "MDA approach to automate code generation for mobile applications," in *Mobile and Wireless Technologies 2016*, Springer, 2016, pp. 241–250.
  - [37] T. Channonthawat and Y. Limpiyakorn, "Model Driven Development of Android Application Prototypes from Windows Navigation Diagrams," in *2016 International Conference on Software Networking (ICSN)*, 2016, pp. 1–4.
  - [38] A. Sabraoui, M. El Koutbi, and I. Khriss, "GUI code generation for Android applications using a MDA approach," in *2012 IEEE International Conference on Complex Systems (ICCS)*, 2012, pp. 1–6.
  - [39] A. Sabraoui, M. El Koutbi, and I. Khriss, "A MDA-based model-driven approach to generate GUI for mobile applications," in *International Review on Computers and Software Journal (IRECOS)*, 2013, pp. 845–852.
  - [40] B.-K. Min, M. Ko, Y. Seo, S. Kuk, and H. S. Kim, "A UML metamodel for smart device application modeling based on Windows Phone 7 platform," in *TENCON 2011-2011 IEEE Region 10 Conference*, 2011, pp. 201–205.
  - [41] J. de\_Almeida Monte-Mor, E. O. Ferreira, H. F. Campos, A. M. da Cunha, and L. A. V. Dias, "Applying MDA approach to create graphical user interfaces," in *2011 Eighth International Conference on Information Technology: New Generations*, 2011, pp. 766–771.
  - [42] M. Lachgar and A. Abdali, "Generating Android graphical user interfaces using an

- MDA approach,” in *2014 Third IEEE International Colloquium in Information Science and Technology (CIST)*, 2014, pp. 80–85.
- [43] A. G. Parada and L. B. De Brisolará, “A model driven approach for android applications development,” in *2012 Brazilian Symposium on Computing System Engineering*, 2012, pp. 192–197.
- [44] O. Le Goaer and S. Waltham, “Yet another DSL for cross-platforms mobile development,” in *Proceedings of the First Workshop on the globalization of domain specific languages*, 2013, pp. 28–33.
- [45] I. Madari, L. Lengyel, and T. Levendovszky, “Modeling the user interface of mobile devices with DSLs,” in *8th International Symposium of Hungarian Researchers on Computational Intelligence and Informatics, Budapest, Hungary*, 2007, pp. 583–589.
- [46] M. Ko, Y.-J. Seo, B.-K. Min, S. Kuk, and H. S. Kim, “Extending UML meta-model for android application,” in *2012 IEEE/ACIS 11th International Conference on Computer and Information Science*, 2012, pp. 669–674.
- [47] R. Mannadiar and H. Vangheluwe, “Modular synthesis of mobile device applications from domain-specific models,” in *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2010, pp. 21–28.
- [48] C.-K. Diep, Q.-N. Tran, and M.-T. Tran, “Online model-driven IDE to design GUIs for cross-platform mobile applications,” in *Proceedings of the Fourth Symposium on Information and Communication Technology*, 2013, pp. 294–300.
- [49] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, “Cross-platform model-driven development of mobile applications with md2,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 526–533.
- [50] F. A. Kraemer, “Engineering android applications based on uml activities,” in *International Conference on Model Driven Engineering Languages and Systems*, 2011, pp. 183–197.
- [51] H. S. Son, W. Y. Kim, and R. Y. C. Kim, “MOF based code generation method for

- android platform,” in *International Journal of Software Engineering and Its Applications* 7.3, 2013, pp. 415–426.
- [52] X.-S. Li, X.-P. Tao, W. Song, and K. Dong, “Aocml: A domain-specific language for model-driven development of activity-oriented context-aware applications,” *J. Comput. Sci. Technol.*, vol. 33, no. 5, pp. 900–917, 2018.
- [53] N. Paspallis, “An MDD-based method for building context-aware applications with high reusability,” *J. Softw. Evol. Process*, vol. 31, no. 11, p. e2200, 2019.
- [54] E. Yigitbas, I. Jovanovikj, K. Biermeier, S. Sauer, and G. Engels, “Integrated model-driven development of self-adaptive user interfaces,” *Softw. Syst. Model.*, pp. 1–25, 2020.
- [55] Y. Hu and I. Neamtiu, “Static detection of event-based races in android apps,” *ACM SIGPLAN Not.*, vol. 53, no. 2, pp. 257–270, 2018.
- [56] J. Sauro and J. R. Lewis, “When designing usability questionnaires, does it hurt to be positive?,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011, pp. 2215–2224.
- [57] JetBrains, “Tool Windows.” 2020, [Online]. Available: [https://jetbrains.org/intellij/sdk/docs/user\\_interface\\_components/tool\\_windows.html](https://jetbrains.org/intellij/sdk/docs/user_interface_components/tool_windows.html).
- [58] JetBrains, “Plugin Services.” 2020, [Online]. Available: [https://jetbrains.org/intellij/sdk/docs/basics/plugin\\_structure/plugin\\_services.html](https://jetbrains.org/intellij/sdk/docs/basics/plugin_structure/plugin_services.html).
- [59] JetBrains, “Indexing and PSI Stubs.” 2020, [Online]. Available: [https://jetbrains.org/intellij/sdk/docs/basics/indexing\\_and\\_psi\\_stubs.html](https://jetbrains.org/intellij/sdk/docs/basics/indexing_and_psi_stubs.html).

## **Appendices**

### **Appendix A - Demographic Survey**

Before starting the case study, each of the participants will fill a survey that contains the following questions:

1. How many years of experience do you have in mobile app development?
  - a. Less than 1
  - b. 1 - 3
  - c. 4 - 7
  - d. 8 or more
2. How many Android apps have you contributed to in your years of experience?
  - a. 0
  - b. 1 - 5
  - c. 6 - 10
  - d. 11 or more
3. How familiar are you with the handling of the Android app lifecycle?
  - a. Not familiar at all
  - b. Have simple knowledge
  - c. Have good understanding of it
  - d. Have Expert knowledge

### **Appendix B - Post- Case Study Questionnaire**

The questionnaire is composed of the following questions each with a 5-point Likert scale ranging from 1 (Strongly Disagree) to 5 (Strongly Agree) and also includes open-ended questions to get extra information from the participants.

		<b>Strongly Disagree</b>	<b>Disagree</b>	<b>Neither Agree nor Disagree</b>	<b>Agree</b>	<b>Strongly Agree</b>
1	It was easy to understand the plugin					
2	I consider the concept of the plugin to be useful					
3	It was easy to learn to use the plugin					
4	It was easy to use the plugin					
5	The plugin helped me understand my handling of the Android lifecycle					
6	The plugin correctly represented my handling of the Android lifecycle					
7	If I start to develop a new app, I will consider using the plugin					
<b>Open-ended questions</b>						
1	What issues did you have when using the plugin?					



2	What changes do you suggest should be made to the plugin?	
---	---	--