RWTH Aachen

Department of Computer Science
Technical Report

Testing Life Cycle-related Properties of Mobile Applications

Dominik Franke

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2015-02

RWTH Aachen · Department of Computer Science · May 2015

Testing Life Cycle-related Properties of Mobile Applications

Testen von Lebenszykluseigenschaften mobiler Anwendungen

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Dominik Franke
aus Tichau (PL)

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr. Horst Lichter

Tag der mündlichen Prüfung: 05.11.2014

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

Dominik Franke
Lehrstuhl Informatik 11
franke@embedded.rwth-aachen.de

Aachener Informatik Bericht AIB-2015-02

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

With an increasing number of mobile devices like smartphones and tablets, their relevance to users and growing number of available applications, also their field of application widens. For the software quality of mobile applications, the application life cycle - the process-related states and state transitions - plays an important role. Today's mobile platforms, like Android, iOS and Windows Phone, have specific scheduling policies on application level to ensure the reactiveness of an application, targeting an improved responsiveness and a good user experience. Depending on the life cycle state of an application, it is allowed or restricted to access resources like RAM and CPU. Such policies can lead to data loss and unexpected behavior of the mobile application.

This work presents a conceptual approach for testing application properties which are related to life cycle state changes, so called *life cycle-related properties*. The first step consists of reverse engineering the life cycles of mobile applications. These life cycles are used as a basis for testing life cycle-related properties at state changes. The testing approach uses callback-mechanisms of the underlying mobile platforms to check assertions about life cycle-related properties. It handles application components with an own life cycle as units and tests each unit in a unit-based testing approach. In a case study, the conceptual approach is implemented for the mobile platform Android. One of the results of the case study is the AndroLIFT tool for testing life cycle-related properties of Android applications.

The evaluation of this work presents the capabilities and limitations of the conceptual approach. While the approach is well-suited for today's mobile platforms, extensible and scalable with respect to the type and number of life cycle-properties, it mainly depends on the callback-mechanism of the underlying mobile platform. The evaluation of the AndroLIFT tool in the context of a practical course with student participants confirms the value of the Android implementation of the presented approach to test life cycle-related properties of Android applications.

Zusammenfassung

Mit der steigenden Anzahl mobiler Endgeräte wie Smartphones oder Tablets, ihrer Relevanz heutzutage sowie einer stetig wachsenden Anzahl verfügbarer mobiler Apps, wächst auch ihr Anwendungsbereich. Für die Softwarequalität von Apps spielt ihr Lebenszyklus - bestehend aus prozessbezogenen Zuständen und Zustandsübergängen - eine wichtige Rolle. Aktuelle mobile Plattformen wie Android, iOS und Windows Phone haben Strategien für das Scheduling von Apps, welche sich von dem Scheduling auf Desktop- und Server-Betriebssystemen unterscheiden. Das Scheduling auf mobilen Plattformen soll insbesondere eine hohe Reaktivität von Apps sicherstellen, mit dem Ziel Reaktionszeiten möglichst kurz zu halten und eine hohe Benutzerfreundlichkeit zu gewährleisten. Abhängig von dem Zustand einer App kann ihr der Zugriff auf Ressourcen wie RAM und CPU gewährt oder verwehrt werden. Dieses Verhalten kann zu Datenverlust und unerwartetem App-Verhalten führen.

Diese Arbeit stellt einen konzeptuellen Ansatz zum Testen von App-Eigenschaften, die in Bezug zum Lebenszyklus der App stehen, vor. Im ersten Schritt wird ein Vorgehen vorgestellt, welches das Reverse Engineering von Lebenszyklus-bezogenen Eigenschaften erlaubt. Die resultierenden Lebenszyklen werden als Basis für das Testen Lebenszyklus-bezogener Eigenschaften verwendet. Der Ansatz nutzt Callback-Mechanismen der darunterliegenden Plattformen um Annahmen über Lebenszyklus-bezogene Eigenschaften zu überprüfen. Anwendungskomponenten mit eigenen Lebenszyklen werden von dem Ansatz als Module angesehen und im Rahmen von Modultests getestet. In einer Fallstudie wird der konzeptuelle Ansatz für die mobile Plattform Android implementiert. Aus dieser Fallstudie geht AndroLIFT hervor, eine Bibliothek zum Testen von Lebenszyklus-bezogenen Eigenschaften von Android Apps.

Die Evaluierung dieser Arbeit stellt die Potentiale und Grenzen des konzeptuellen Ansatzes vor. Der Ansatz ist leicht für die heutigen mobilen Plattformen zu implementieren, er skaliert mit einer zunehmenden Anzahl von Annahmen und ist leicht anpassbar und erweiterbar im Hinblick auf unterschiedliche Annahmen. Aber es zeigt sich auch eine Abhängigkeit von den für diesen Ansatz benötigten Callback-Mechanismen der darunterliegenden Plattformen. Die Evaluation von AndroLIFT im Rahmen eines studentischen Praktikums bestätigt den Nutzen der Android Implementierung des vorgestellten Ansatzes um Lebenszyklus-bezogene Eigenschaften zu überprüfen.

Acknowledgments

It would not have been possible to write my doctoral thesis without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

I would like to thank my supervisor Prof. Dr.-Ing. Stefan Kowalewski for his decision on my PhD application. Thank you for your confidence and trust on me. I had a great freedom to plan and execute my ideas in research without any pressure. This made me to identify my own strength and drawbacks, and particularly boosted my self-confidence.

My big thanks go to my thesis co-supervisor and promoter Dr. Carsten Weise. Your patience in explaining me various scientific topics in full width and intense depth is amazing. You taught me how important details, timing and politics are for research. Moreover, the Ideas you gave me as an input, whenever I had a hard time with my ongoing work, were of my moral support. I always ended up with confidence and full of energy after the discussions with you. I inspired from you about the true research and its value, which I feel at the end very important for budding researchers like me. You always understood any issues from my point of view, and gave me full freedom and your support for all the decisions I have made. I am lucky to be one of those who had an opportunity to work with you.

I am grateful to the dissertation committee members Prof. Dr. Wolfgang Thomas, Prof. Dr. Horst Lichter and Prof. Dr. Rumpe for their careful reading, valuable comments and suggestions on my thesis.

Special thanks to my ex colleagues with whom I have enjoyed my past four years at the Embedded Software Laboratory at RWTH Aachen University. In particular I want to thank John F. Schommer for always having some time left for new ideas, insights and critical evaluations as well as Dr. Hilal Diab and Ibtissem Ben Makhlouf for their mental and motivational backup.

A large part of my work has been done in cooperation with my amazing students. First of all, I want to thank Corinna Elsemann for laying the groundwork for my research. Second, I appreciate the long term assistance and constant input of Tobias Royé. Third, I enjoyed watching Norman Hansen contributing a large part to my research with ease. Further thankful contributions were made by my students Robert Mathes, David Thönnessen, Daniel Forster, Dani Baumeister, Stefan Hempel, Andreas Weigelt, Nath Prakobkosol, Dominik Schmithausen, Tim Lange and Marko Mijatovic. I also thank my students Dzenan Dzafic, Igor Kalkov, Christian Dernehl, Thomas Gerlitz and Daniel Losch, who today work their way through their own PhD topics.

I have made significant progress while writing down my thesis at Lero, the Irish Software Engineering Research Centre at the University of Limerick. I gratefully thank Prof. Dr. Mike Hinchey for inviting me to Ireland and giving me the opportunity to write down my PhD thesis with the great advisors Dr. Goetz Botterweck and Dr. Andreas Pleuss. Thank you for the caring integration, great degrees of freedom and an unforgettable time.

Last but not least I thank my parents Christina and Stephan Franke as well as my sister Violetta Franke and my brother Adam Franke for directing and guiding me from my very first days. I also thank Hannelore and Heinz Schwartz who had major influence on my educational career, which finally allowed me to write this thesis. My special thanks go to my girlfriend Elena Kirch, who accepted my excessive research journeys, took care of social commitments whenever they fell in my prioritization and enabled a wonderful life under demanding conditions.

Dominik Franke
Aachen, January 2015

Contents

List of Figures

1 Introduction

Mobile devices, like smartphones and tablets, are omnipresent today. They significantly change the traditional computer market by shifting the focus from PCs to smartphones and tablets. This leads to a continuous decrease of PC sales and increase of sales in the mobile device area in the recent past and expected in the future. For instance, worldwide PC sales are forecast to decrease by 7.6 percent from 2012 to 2013 and the sales of mobile devices (smartphones and tablets) to increase by 11.3 percent in the same time period [77]. Although the performance of mobile devices continuously improves with the booming market, they still are limited in their capabilities. Major limitations arise from the limited battery capacity, CPU, RAM and restricted possibilities to receive input from the user and present information to the user.

Nonetheless, the number of available mobile devices increases continuously. With the growing number of devices also the number of applications for these devices increases. For instance, Google counts 48 billion installations of Android applications and Apple 50 billion iOS installations (status May 2013) [56]. In such a growing and booming mobile application market, software quality is an important topic. Many quality issues on mobile devices arise from the fact that applications do not deal appropriately with the scarce resources, causing applications to halt, crash or any other unexpected application behavior [92]. This leads to a bad user experience and decreases the experienced quality of the application.

To prevent such an application behavior, today's mobile platforms, like Android, iOS and Windows Phone, have more or less similar policies regarding the access of applications to the device resources. Many mobile platforms allow only a very limited number of visible applications to be executed simultaneously. This number is often even limited to one single application. Since usually smartphones have a small display to present information to the user, the only applications being executed are the ones presenting information to the user. There may also be other services (not displaying data to the user through the user interface) and system tasks active in the background, but the number of visible applications is strictly limited. This ensures that the visible applications have sufficient resources available to stay reactive and execute user tasks in time. A good responsiveness is an important factor for a good user experience and high usability.

To ensure the policy of only few (or one) visible applications being executed at a time, today's mobile platforms manage life cycles on application level. These life cycles specify the application behavior with regard to the application's states and state transitions. An application can be in different states in which it has different levels of access to resources. For instance, an application in the state *running* can have access to all resources, while an application in the state *stopped* has no access to the CPU and a shut down application has not even access to the RAM. To ensure that only one visible application is being executed at a time, the mobile platform allows only one single application at a time to be in the state *running*. All other applications, which are usually not visible at that time, remain in other states with limited access to resources.

If only very few (or one) applications are presented to the smartphone or tablet user simultaneously, the user has to switch often between the different applications. For instance, if the user reads an e-mail, the e-mail application is open. When he clicks on a link in the e-mail, the Internet browser application might be opened. While using the Internet browser, an incoming call might occur, which would open the phone application. All these actions cause state changes in the life cycles of the involved mobile applications. With every state change an application might loose access to resources. For instance, if an application is shut down or killed by the system, it is usually also flushed out of RAM. Such state changes might cause data loss and unexpected application behavior, e.g. if the entered e-mail text is lost after resuming with the intention to continue to write the unfinished e-mail.

On today's mobile platforms it is the responsibility of the application developer to deal with life cycle state changes of his application on application level. For instance, when an incoming call interrupts the running application, a banking application should appropriately shut down all secure connections and an e-mail application should store the e-mail content. In the following, properties which go beyond single life cycle states of an application are called *life cycle-related properties*. There exists no conceptual approach to test life cycle-related properties with the specific issues of today's mobile applications.

1.1 Objectives

The objective is to implement a conceptual approach to test life cycle-related properties of mobile applications. This approach shall not depend on a specific platform, but be applicable to a wide range of current mobile platforms. It shall be sufficiently generic to not depend on a specific system or system architecture and it shall not limit the number and kind of life cycle-related properties that can be tested. The approach shall be easy to implement for a specific platform to test life cycle-related properties.

To achieve these goals, first the status quo of how application life cycles are provided and handled by today's mobile platforms has to be analyzed. Afterwards, we have to find a common basis between the mobile platforms to identify and apply established testing techniques with the purpose of testing life cycle-related properties. Third, we have to evaluate our approach by implementing it for a specific mobile platform.

1.2 Contributions

The main contributions supporting the objectives of our work are the following:

- We provide a concept for reverse engineering mobile application life cycles. Due to the lacking quality of many official documentations on application life cycles [65], this approach can be used by developers to reverse engineer the actual application life cycle of a mobile platform. This approach is platform-independent and can be applied to current mobile platforms. It is applicable to graphical applications as well as services of mobile platforms.

- We applied this concept to various mobile platforms for reverse engineering life cycles of applications and services on Android, iOS, Java ME and Windows Phone. The results consist of reverse engineered life cycles of service and application components of various platforms. The results also contain trigger catalogs which trigger different life cycle state change sequences of a mobile application on a specific platform. These catalogs can be used for testing life cycle-related properties of further platforms.

- We developed a platform-independent, conceptual approach for testing life cycle-related properties on today's mobile platforms. The approach can be easily implemented on a specific mobile platform. It is a unit-based testing approach which works in a White-box manner and uses assertions. The approach is based on life cycle callback mechanisms of the mobile platforms.

- We implemented this approach in a case study for the mobile platform Android. The implementation resulted in a library and plug-in for testing life cycle-related properties of Android applications. The tools can be easily extended to test application-specific life cycle-related properties. They help users to quickly and easily test properties in a structured and efficient way.

1.3 Outline

Chapter 2 presents preliminaries used in this thesis. This includes background about application life cycles, unit and White-box testing as well as basic information about the tools used in our case study. Chapter 3 introduces our approach on reverse engineering application life cycles. The approach is used to test life cycle-related properties of mobile applications. It also presents the reverse engineered application life cycles of different mobile platforms, including Android. The results from reverse engineering are used to test life cycle-related properties of Android applications in our case study. In Chapter 4 our conceptual approach of testing application life cycles is presented. This approach is applied in a case study to test life cycle-related properties of Android applications. The case study and the results are presented in Chapter 5. This chapter also introduces the tools developed during the case study. The experiences and results from our case study are evaluated in Chapter 6. Chapter 7 presents related work and Chapter 8 concludes this thesis and sketches future work.

1.4 Bibliographic Notes

Some parts of this thesis are based on previous work, published beforehand. We describe the approach of reverse engineering life cycles of mobile applications and corresponding case studies on this in [55, 65, 67, 84, 85]. The approach of testing life cycle-related properties and applications of the approach were reported in [55, 62, 68, 69, 71, 120]. Related tools, like the AndroLIFT library, plug-in and Higgs are presented in [55, 60, 70, 88, 102, 123, 137]. Some work done on specifying requirements on life cycle-related properties is summarized in [72, 88]. The conceptual approach was also considered while working on real-time systems, like the real-time capable Android, and presented in [61, 79, 93, 94, 126, 134]. Some ideas in this work arose from our work on a mobile Android client for a project which deals with energy-efficient navigation for electric wheelchairs [36, 51–54, 63, 64, 66]. Basic conceptual ideas and inspiration are taken from various other own work published in [32, 47, 48, 78, 97, 100, 107, 110, 125, 127].

2 Preliminaries

2.1 Application Life Cycles

In this work, the term *application life cycle* refers to the process-related states of an application during runtime and transitions between these states. Application life cycles play an important role for the quality of today's mobile applications. Mobile devices like smartphones and tablets have limited amounts of resources like CPU, RAM and battery. Additionally, they have restricted capabilities in terms of output information and user input. Therefore, today's mobile platforms like Android, iOS and Windows Phone schedule their applications in a specific way: At each moment in time, only a very limited number of visible applications (often only one single application, next to some background services) is allowed to be active. This application has access to all resources. While one application is active, the other applications remain in other states. Figure 2.1 sketches a simplified life cycle of a Java ME application [69]. Regarding the Java ME policies related to this life cycle, only one single visible application is allowed to be in the state *active*, while all other applications have to be in the states *paused* or *destroyed*. Depending on the current state of an application, the mobile platforms restrict the access of the application to resources. While the only active application has access to all resources, all paused applications usually have only access to RAM. This way, paused applications are started more quickly when being resumed. Applications which are in the state *destroyed*, are usually completely shut down and do not have access to any resources like CPU and RAM.

Such scheduling policies of today's mobile platforms lead to many application state changes. For instance, if one application is active and visible to a smartphone user and he switches to a different (e.g. destroyed) application, the following state changes are necessary (assuming the application life cycle of Fig. 2.1):

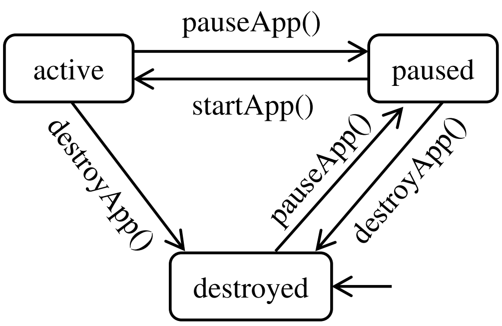

Figure 2.1: Simplified Life Cycle of a Java ME application

- The currently active application has to be paused. It may also be destroyed, depending on other platform-specific scheduling policies.
- The starting (from the state destroyed) application has first to be paused, too.
- Then the starting application has to switch its state from *paused* to *active*.

After these state changes, the second application is visible to the user. Depending on the platform-specific application life cycles, for an application switch often even more state changes are necessary like on Android, iOS and Windows Phone.

As the mobile applications may be forced to release resources when changing their states, the applications have to take care of data loss and consistency. For instance:

> A user types in a long text into an e-mail text field. Unexpectedly, he is interrupted by an incoming call. When resuming the e-mail application, he expects it to have stored the e-mail text. Therefore, the application needs to be notified by the underlying platform in case of an upcoming state change to be able to store the entered e-mail text.

Another example scenario:

> A smartphone user is using a gaming application. Due to the high power demand of the game, the low battery capacity shrinks quickly and the smartphone shuts down. The user expects his game to be paused where it was interrupted and his gaming score to be stored.

To notify mobile applications of upcoming state changes, today's mobile platforms use callback mechanisms. While on Android and iOS the platforms execute callback methods of the applications, Windows Phone notifies its applications sending a corresponding event. The transitions in Fig. 2.1 are labeled with the names of the corresponding callback methods of Java ME applications. So regarding the life cycle in Fig. 2.1, the callback method *startApp()* of a paused application is called when the application changes its state to *active*. The moment in time, when exactly the call is made, depends on the platform specification. The developer of a mobile application usually has the possibility to override the callback method in his application. He can insert data to store e-mail content persistently, pause games, shut down connections appropriately and so on. An application might have various callback methods, not only for life cycle actions, but also for notifications of actions like pushing a button. To clearly distinguish the callback methods called during state changes from other callback methods, we introduce the term *life cycle callback methods*. *Life cycle callback methods* are those callback methods of an application which are called as a consequence of a state change of the application. Further, we define the term *to implement an application life cycle*. An application *implements the application life cycle* if at least one of its life cycle callback methods is overridden. The application life cycles and life cycle callback methods play an important role when testing life cycle-related properties.

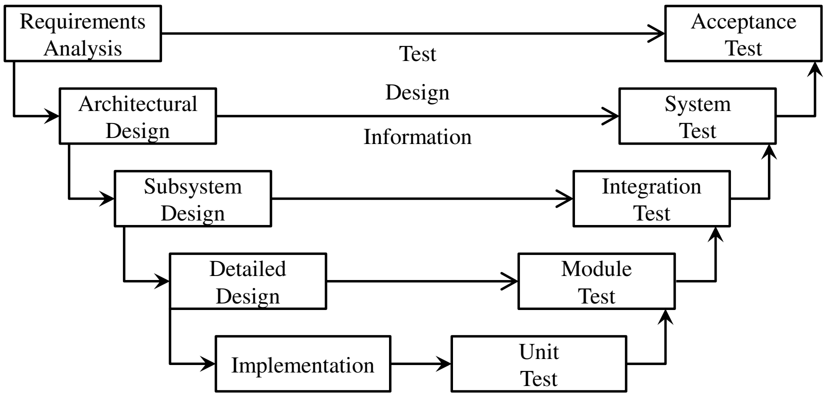

Figure 2.2: Unit Testing in the V-Model

2.2 Unit Testing

Testing of software is an extensive and multilevel task. Following the V-Model for software development, the different test tasks can be integrated into the V-Model as presented in Fig. 2.2 [26]. An important and widespread technique for testing on implementation level is unit testing. A unit is often referred to as a closed and smallest testable entity [99]. What a unit is in a specific case, depends on the programming language. For instance, it can be a function in C and C++, procedure or function in Ada, method in Java or subroutine in Fortran [26]. Sometimes classes, which can contain multiple methods, are called *modules*. In this case, a class in isolation is referred to as *module testing* (see Fig. 2.2) [26]. But in the context of object-oriented systems a whole class is also called unit. In object-oriented unit tests one unit might consist of one or multiple classes interacting with each other [99]. The major goal during unit testing is to create an isolated environment for the unit to test. The influences on the unit shall be in strict control of the tester. During test execution, the tester injects input triggers to a unit (e.g. method parameters of a certain value) and observes the reaction (e.g. output/return values) of the unit. The unit has to be isolated to make sure that the only reasons for the reaction of the unit are the input triggers. A unit can be tested with or without the internal knowledge of a unit [26]. For testing the unit in isolation, often test stubs are used [26]. These stubs are connected to the unit under test and inject the input triggers.

Unit testing is also an important application area for control flow testing [99]. In control flow testing the code structure is analyzed and the test progress can be monitored using coverage criteria. The IEC 61508-3 states that each software module has to provide a specified functionality. This standard uses the term *shall show*, which does not ask for a prove, but a demonstration [99]. It also hints that it is not necessary to cover all possible in- and output combinations, but, e.g., to work with formal methods and assertions. Units are not only separated for testing, but can also already be separated during development [42]. Different developers might separately develop single units. Then each unit can also be tested separately. When different units (or modules) are integrated during development, integration testing can be used for quality assurance (see Fig. 2.3).

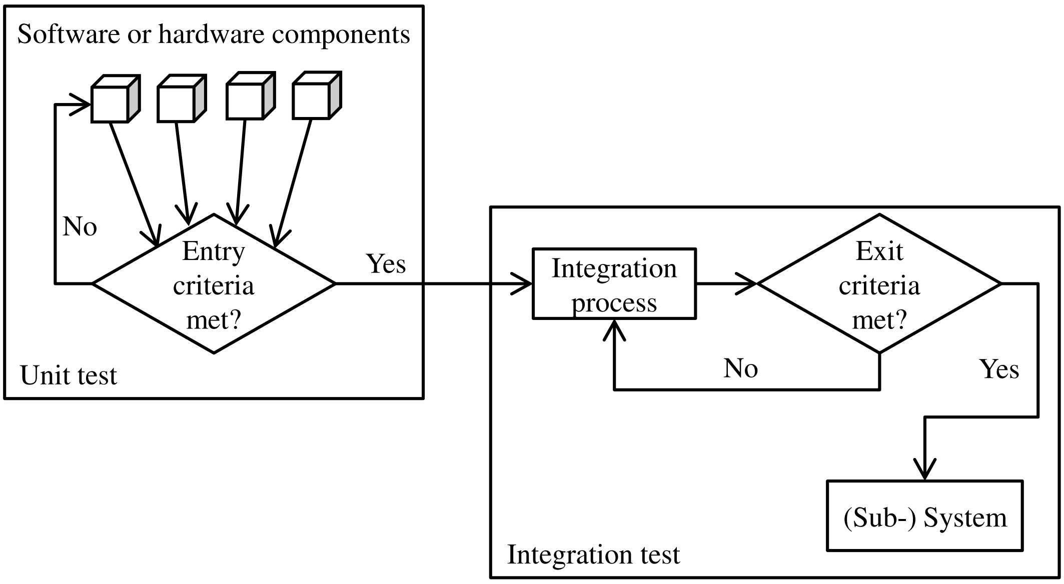

Figure 2.3: Unit and Integration Testing

Unit and integration tests are inherently finite, but cannot detect all defects, even if completely executed. Only a combination of the two approaches together with a risk-based test strategy can detect the important defects [42]. The testing concept presented in the following is most efficient and makes most sense if used together with other testing techniques to ensure the quality of the whole application, not only with respect to life cycle-related properties.

2.3 White-box Testing

A common categorization for test techniques is the distinction between White- and Black-box testing. Tests of both categories belong to the area of dynamic testing [99], where, e.g., a unit is executed and feeded with input [26]. The output is observed and analyzed. During White-box testing, the tester has knowledge about the internals of a unit [49]. For instance, he can see the control structures of the unit, branches, individual conditions and statements [26]. Thus, all structure-oriented test techniques are White-box tests. With knowledge about the internal structures of a unit, the tester can configure the input to a unit for a test case so that certain structures of the unit are entered and corresponding actions triggered (see Fig. 2.4 left) [99]. In contrast to White-box testing, during Black-box testing the tester has no insight into units. The input to the unit during testing is done without any knowledge of the internals of the unit (see Fig. 2.4 right). Many White-box techniques require more effort to learn the internals of a unit and implementing tests, but White-box tests also have a higher coverage effectiveness [35]. In both testing categories the test output of a unit is compared to the expected behavior, e.g., derived from the specification.

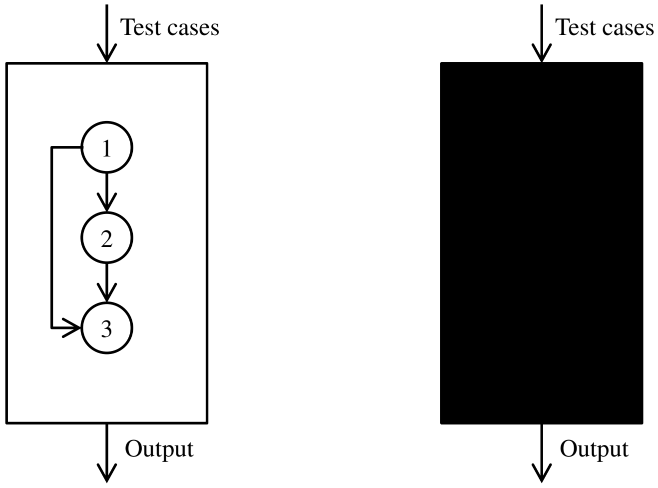

Figure 2.4: White- and Black-box Testing

2.4 Reverse Engineering

Referring to Chikofsky and Cross [44] *reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction.* During this process the reverse engineered software is not modified, which would be *reengineering*. There are various techniques for reverse engineering a software system. In our work, we reverse engineer the system by feeding it with triggers during runtime and logging its reactions. In a subsequent step, we analyze the log data to construct an abstract view (related to life cycles) on the system. It is common to use this procedure to, e.g., reconstruct missing or incomplete system documentation or various views on the same system [95]. Additionally, misbehavior of the system can be exposed if the reverse engineered behavior or design does not correspond to the specified one. Exposing system errors and completing documentations can improve the overall quality of the reverse engineered system.

2.5 Eclipse IDE and Android ADT

The Eclipse integrated development environment (IDE) is a platform for tool integration, available for many operating systems (Windows, OS X, Linux, ...). As part of the Eclipse software development kit (SDK), the Java Development Kit (JDK) can be used to develop Java applications within the Eclipse IDE. Other integrated tools are source code management (SCM) tools, like Concurrent Versions System (CVS) and Apache Subversion (SVN), task and life cycle management tools, like Mylyn[1] and many others, usually available as Eclipse plug-ins.

Eclipse provides a rich client platform (RCP) for the integration and extension of functionality [103]. This facilitates the development and integration of plug-ins. For

[1]See `http://www.eclipse.org/mylyn`.

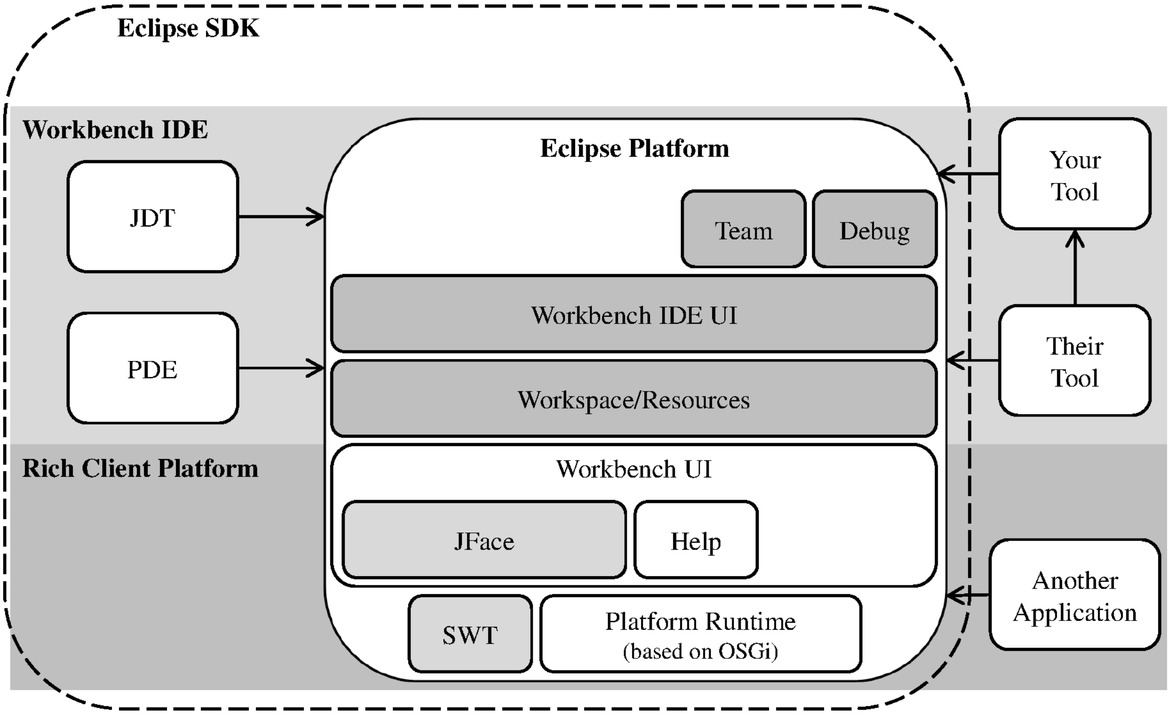

Figure 2.5: Eclipse Platform Architecture

instance, many of the available features in the Eclipse SDK, like JDT, are plug-ins, too [89]. Fig. 2.5 presents the Eclipse platform architecture [46]. The bottom layer unites the components building the RCP and the layer on top of the workbench IDE components. The RCP layer consists of components which allow the integration and execution of rich client applications. The Platform Runtime component, being itself an Open Services Gateway initiative (OSGi) module system, is the core of this layer. RCP provides as widget toolkit the Standard Widget Toolkit (SWT) and for drawing graphical user interfaces JFace [86]. The workbench IDE layer, on top of the RCP layer, contains various components for user interfaces, debug and team functionality. Additionally, the Eclipse SDK also contains plug-ins like JDT and the Plug-in Development Environment (PDE) [74].

One of the available Eclipse plug-ins is the Android Developer Tools (ADT) Eclipse plug-in [4]. On the official Android pages, ADT is the recommended way to develop software for Android [12]. The plug-in integrates various features to Eclipse, like *Extensible Markup Language (XML)* editors for viewing Android-specific XML files, views for previewing Android user interfaces and tools from the Android SDK [4]. Together with the Android SDK, the ADT Eclipse plug-in provides an integrated usage of ADT in Eclipse [4]:

- *Traceview:* Profiling of Android applications.
- *android:* Creation of Android projects, managing and starting Android emulators and much more.

- *Dalvik Debug Monitor Server (DDMS):* Debugging features, as thread and heap information, Logcat and screen capturing.
- *Android Debug Bridge (adb):* Access to devices and emulators for file transfer, shell commands, installing Android applications and so on.

Within the context of the case study, we extend the ADT Eclipse plug-in with the functionality to test life cycle-related properties. Therefore, we make use of the ADT, like adb and DDMS and integrate the modifications in Eclipse using Eclipse components like views and perspectives.

3 Reverse Engineering Application Life Cycles

Since application life cycles play an important role for the quality of today's mobile applications, it is important to implement the life cycle appropriately. To be able to do so, good documentations and representations of the life cycles are necessary. During our work with application life cycles of current mobile platforms like Android, iOS, Java ME and Windows Phone, we noticed that many of the official documents on life cycles are incorrect, incomplete and inconsistent. In this section, we present an approach to reverse engineer application life cycles in four major steps. We also present the results of applying this approach to the mobile platforms Android 2.2, Java ME with MIDP 2.0, iOS 4.0 and Windows Phone 7.5.

3.1 Issues with official Life Cycle Models

During our research on application life cycles, we faced a couple of different issues resulting from the official documentations and representations of life cycles. A complete list of the issues and details can be found in [55, 65, 84]. In the following, we present some interesting issues.

One convenient and typical way to test mobile applications of today's mobile platforms is to use the provided simulators and emulators. To test life cycle-related properties of an application with a simulator or emulator, it is necessary that the simulator or emulator provides the required options to trigger life cycle state changes. For instance, the emulator must be capable of simulating an incoming call, a shutdown due to low battery, killing an application as a consequence of running out of memory, change the device orientation, lock the screen, press hardware buttons and much more. All these actions may result in state changes of the not shut down applications and may lead to different traces in the application's life cycle. Our experiences with the available simulators and emulators for the platforms Android 2.2, iOS 4.0, Java ME and Windows Phone 7.5 show that none of them fulfills the requirements completely [65]. For instance, it is not possible to simulate an incoming phone call and an empty battery with the iOS 4.0 simulator. Furthermore, the CPU and RAM capacities of the iOS 4.0 simulator running on a desktop operating system equal the capacities of the underlying operating system. So if the developer runs the simulator on a MacBook Pro with 6 GB of RAM, the simulator has access to the whole unused part of the 6 GB RAM. Thus, e.g., it is hard to simulate killing an application due to an out of memory situation.

Another major issue are several inaccuracies in official life cycle documentations and representations [65]. Usually, life cycles are taught to developers by a documentation with a corresponding life cycle model representation. For instance, the documentation of the Android 2.2 *Activity*, which is the main class for Android applications with a graphical user interface, does not define the state *shut down*. But this state is a part of the official life cycle model representation (see Fig. 3.1) [2, 55]. On the contrary, the states *paused* and *stopped* are not presented in the model representation, but defined in the textual documentation [2, 55]. The textual documentation mentions the possibility of calling the life cycle method *onStart()* immediately followed by *onStop()*. This sequence of life cycle callback methods is not possible in the corresponding model representation (see Fig. 3.1).

In contrast to the mostly informal life cycle model representation of Android 2.2, iOS 4.0 and Windows Phone 7.5, the official detailed representation of the Java ME MIDP 2.0 life cycle model is kept predominantly formal (see Fig. 3.2) [121]. Nonetheless even there are some issues, since not all transitions follow the same syntax [55]. For instance, transitions follow the notation *event [guard] / action* [55, 121]. But, e.g., the transition from the state *start* to *destroyed* is labeled */AMS: Create MIDlet() [RuntimeException thrown]*. This does not fit into the notation, as it is unclear if *[RuntimeException thrown]* is the guard and what *AMS: Create MIDlet()* is.

Due to unclear, ambiguous and incomplete official material on application life cycles, it is hard for developers to correctly implement and reliably test application life cycles. The goal of reverse engineering application life cycles is to obtain a more reliable, more complete and unambiguous model representation of the life cycle.

3.2 Conceptual Approach

The life cycle is usually implemented by the core application class of an application, whereas the whole application might consist of multiple classes. For instance, on Android the core application class may be an instance of the *Activity*-class and on iOS an instance of the *application*-class. The code of these classes is usually executed after startup of an application and they play an important role during runtime. On Android, which is Java-based, the thread executing the Activity-class has the privilege to be the only thread of the whole application to access the user interface. All user interface updates have to be done through this thread. Furthermore, the Activity class also has access to various platform resources, like the application context. As the core application classes are also able to implement the life cycle by overriding the life cycle callback methods or handling life cycle events, they are in the main focus when reverse engineering the application life cycles.

The approach basically sees the application core classes as black boxes. It feeds the black boxes with life cycle triggers, like starting an application and interrupting it with an incoming call. Additionally, it monitors and logs the behavior of the application life cycle, as a consequence of the received triggers. The recorded information are then evaluated to restore information about the actual life cycle. The approach consists of four major steps,

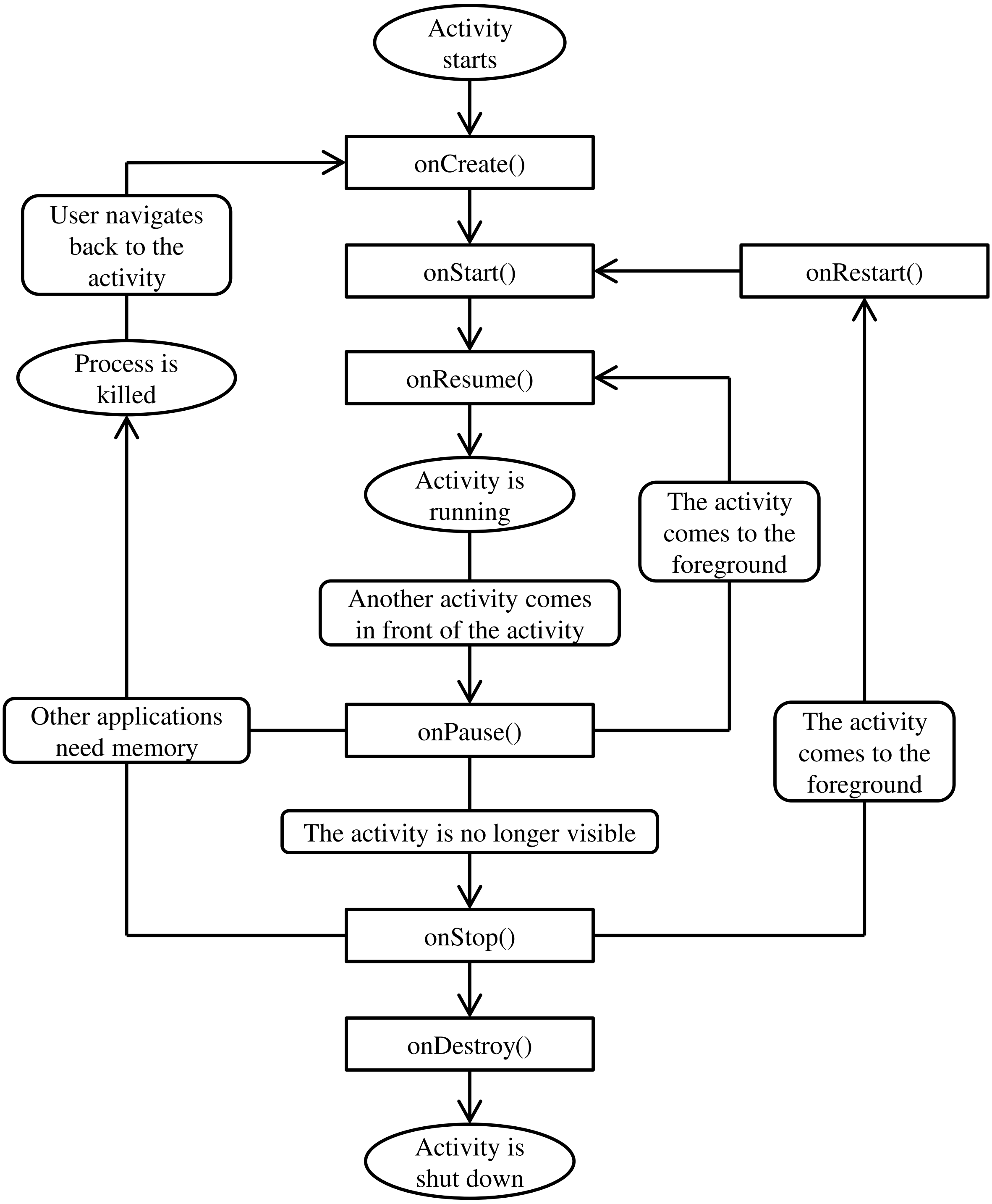

Figure 3.1: Official Life Cycle Model Representation of the Android 2.2 Activity

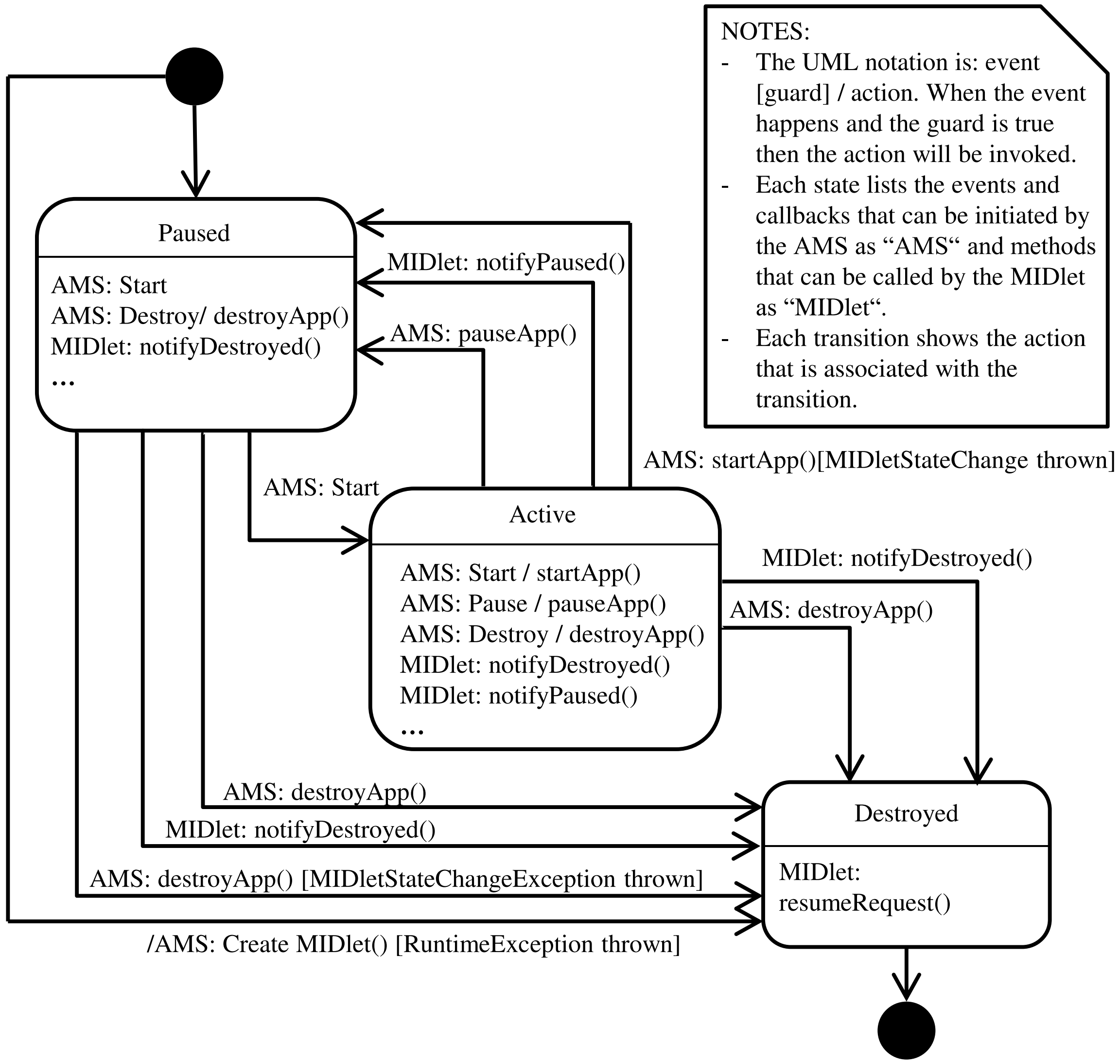

Figure 3.2: Official Life Cycle Model Representation of a Java ME MIDlet 2.0 Application

Time (ms)	Application	Message
...	...	...
10002	App1	pauseApp() called.
10039	App2	startApp() called.
10048	App1	destroyApp() called.
...	...	...

Figure 3.3: Logger Example with two logging Applications

summarized in the following section. More details on each of the steps can be found in [55, 65].

3.3 Extracting the Life Cycle in four Steps

The first step is the *full implementation of the application life cycle*. The developer has to implement all application life cycle callback methods or catch all life cycle relevant events. The exact procedure depends on the underlying platform, e.g., Android uses callback methods and Windows Phone events. To get a full list of the life cycle callback methods and events, the developer has to check the documentation of the platform, the life cycle model representation, if available, the source code of the core application class, if accessible, and developer guidelines. For instance, regarding only the application life cycle model representation in Fig. 2.1, the life cycle methods *pauseApp()*, *startApp()* and *destroyApp()* have to be overridden in the application core class. The resulting application can be a simple *Hello, world!* application with all life cycle methods overridden. It is important that the application itself does not affect the application life cycle, e.g., by invoking life cycle callback methods manually through method calls. So a small application, e.g., which displays in a text view *Hello, world!* to the developer, suits the requirements.

In the second step, *log injection*, the developer has to inject log functionality into each of the overridden life cycle callback methods. Each time one of the methods is called, the following information shall be logged: A current timestamp, a unique name to identify the application and an identifier to unambiguously identify the life cycle callback method. The timestamp is needed to detect the call sequence of different methods. The names of the application and callback method are necessary to unambiguously identify the life cycle callback method. This is especially important if the developer has multiple applications running, e.g., while examining the life cycle callback method sequences when opening one application out of another. A log output example, based on the life cycle model presented in Fig. 2.1, is given in Fig. 3.3 [65]. The example shows that first the life cycle callback method *pauseApp()* of the application *App1* is called, next the *startApp()* of *App2* and finally *destroyApp()* of *App1*. So after *App2* has already been started, further life cycle state changes are triggered in the background. This is also the case on, e.g., Android 2.2.

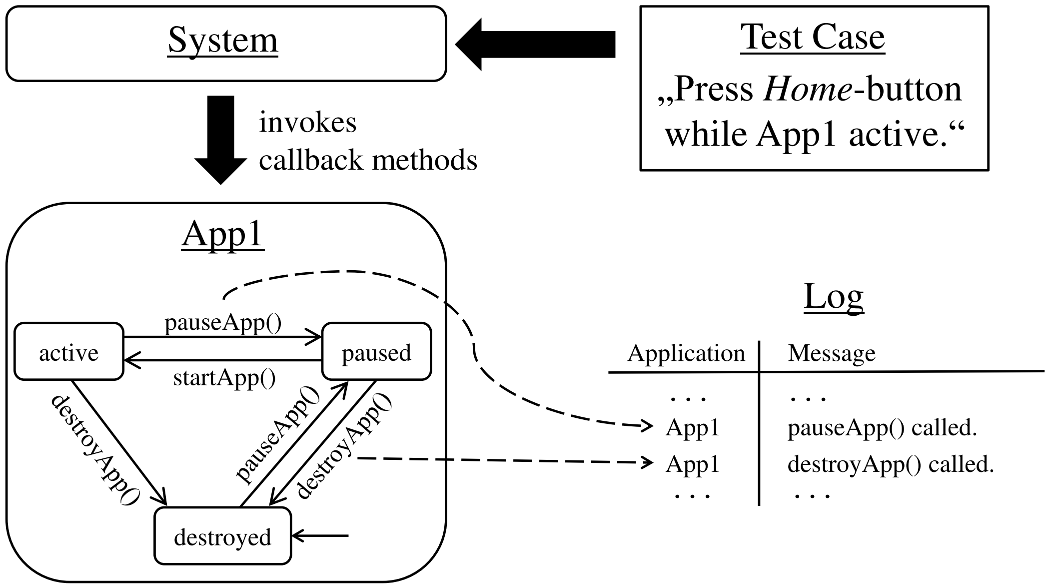

Figure 3.4: Transition-Trigger Detection using a Black-box Approach

Transition-trigger detection, as the third step, identifies triggers which cause state changes of applications. For each mobile platform, there is a limited amount of triggers which cause different sequences of life cycle state changes. For instance, on Android 2.2 locking the screen pauses a running application and accepting an incoming call stops it. The trigger catalog can be established using Black-box testing [114]. Figure 3.4 depicts the procedure. After the developer specifies a trigger for a test case, he executes it on the mobile device. The mobile platform receives the trigger, e.g. pressing the *Home*-button, and decides what to do next. If the *Home*-button is pressed, the platform intends to present the home screen to the user. So if an application is currently active, it has to be paused or even destroyed. When changing the states of the active application, the system invokes the corresponding life cycle callback methods. Due to the injected logging code in the callback methods, each invocation of the methods is logged. In Fig. 3.4 the two callback methods *pauseApp()* and *destroyApp()* are called as a result to the trigger *Press Home-button while App1 is active.*

Two questions remain: At which point did the developer specify a sufficient number of test cases and is the resulting trigger catalog complete? With this approach, there is no way to verify that the resulting trigger catalog is complete. First of all, the developer should try out the most obvious life cycle state change triggers, like incoming phone calls, accepting and declining these calls, receiving SMS messages and so on. Additionally, he has to check the official platform documentation, developer manuals and other available documents for triggers that might lead to a different sequence of state changes. Finally, he can also explore already available trigger catalogs from other platforms, like the ones presented in this work. At some point, the developer does not produce any new state change sequences, which are not already in the catalog. For instance, receiving an incoming call from the persons *A* and *B* on the Android platform leads to the same state changes as an interrupted running application. The same holds for being interrupted by an incoming SMS or e-mail. In the end, the developer has a catalog of single triggers (no combinations) and corresponding state change sequences, which all differ in their sequences. Table 3.1 presents a part of the catalog from reverse engineering the Android 2.2 Activity life cycle [65]. Since usually there are no life cycle callback methods called when the application is killed, the developer has to get this information from the system log. Android, for instance, which is running on a Linux kernel, reports the killing of an application by logging the process ID of the underlying killed Linux process. Our experiences with the platforms Android, iOS, Windows Phone and Java ME confirm that such a catalog is usually sufficiently complete to rebuild a reliable life cycle model representation.

In the fourth and last step, *rebuilding the application life cycle*, the developer is able to reconstruct a life cycle model representation from the collected state change information, represented by corresponding life cycle callback method sequences. Life cycle callback method sequences from the test results must also be possible in the model representation. Regarding Fig. 3.4, the sequence of *pauseApp()* followed by *destroyApp()* also has to be possible in the derived model representation, which is indeed the case in Fig. 2.1. On the contrary, the sequence of *destroyApp()* followed by *startApp()* could not be triggered by any triggers in the catalog. Thus, the sequence is also not possible in the life cycle model representation (see Fig. 2.1). The names of the states usually derive from the official model representations, as well as from the official documentation. But sometimes the reverse engineered model has more states than given in the official documents and models [65, 67, 84]. In this case, it is up to the developer to give the states reasonable names. Some reverse engineered life cycle model representations are presented in the following section.

3.4 Case Studies

We applied the approach on reverse engineering application life cycles to various mobile platform components, like the Android 2.2 service, Java ME MIDlet 2.0 application, iOS 4.0 service, Windows Phone 7.5 XNA applications and so on [55, 65, 84]. In this section, we present some interesting results of reverse engineering the Android 2.2 Activity, iOS 4.0 and Windows Phone 7.5 Silverlight application life cycles. Further information and more details on the process and results of reverse engineering these application life cycles are given in [65, 84].

3.4.1 Android

Figure 3.5 presents the outcome of reverse engineering the Android 2.2 Activity life cycle [65]. The life cycle model representation consists of four major states: *shut down*, *stopped*, *paused* and *running*. The states are named after the states mentioned in the official Android 2.2 documentation [2]. There is also a fifth state added to the model representation, in which the application never remains. This state is presented smaller

Table 3.1: Trigger Catalog for the Android 2.2 Activity

Trigger	**Reaction**
(1) start application by clicking on the application item on the home screen	`onCreate()`, `onStart()`, `onResume()`
(2) after (1) receive an incoming call	`onPause()`, `onStop()`
(3) after (2) decline the incoming call	`onRestart()`, `onStart()`, `onResume()`
(4) after (1) press *Back*-button	`onPause()`, `onStop()`, `onDestroy()`
(5) after (1) activate screen locking by shortly pressing the *On/Off*-button	`onPause()`
(6) after (5) press *Menu-/Home-* or *Call*-button	`onResume()`
(7) after (5) change device orientation (e.g. from vertical to horizontal)	`onStop()`, `onDestroy()`, `onCreate()`, `onStart()`, `onResume()`, `onPause()`
(8) after (1) change device orientation (e.g. from vertical to horizontal)	`onPause()`, `onStop()`, `onDestroy()`, `onCreate()`, `onStart()`, `onResume()`
(9) after (5) push the *On/Off*-button once and release it, then hold it and choose *Shut down* from the upcoming menu	`onResume()`, `onPause()`, kill
(10) after (1) push the *Home*-button and shut the device down	`onPause()`, `onStop()`, kill
...	...

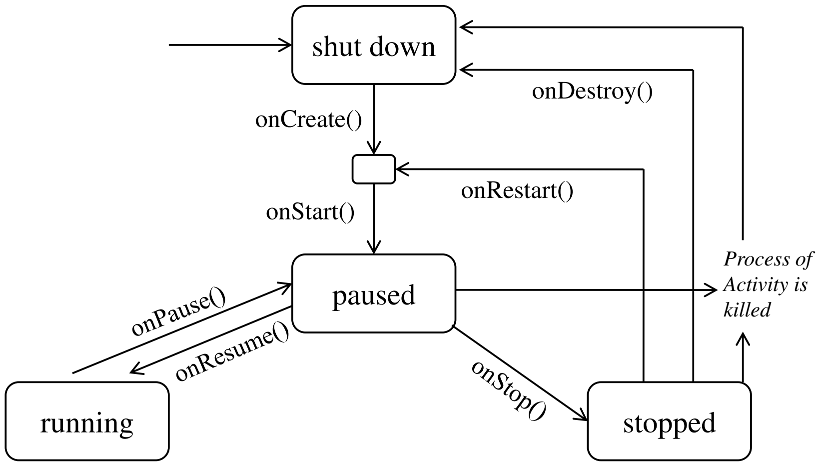

Figure 3.5: Reverse Engineered Android 2.2 Activity Life Cycle

and not labeled, as it is only added for syntactical reasons of the model representation. It is needed to stay conform with labeling each transition with only one life cycle callback method name. So if two transitions are in sequence, there has to be a state in between. This unlabeled state describes that the callback methods *onCreate()* and *onRestart()* are always immediately followed by the callback method *onStart()*. The transitions are labeled by seven different life cycle callback method names: *onCreate()*, *onStart()*, *onResume()*, *onPause()*, *onStop()*, *onRestart()* and *onDestroy()*. The unlabeled transition pointing from nowhere towards the state *shut down* marks this state as the initial state for each Android 2.2 Activity. The unlabeled transitions from the states *paused* and *stopped* towards *shut down* describe state changes in which the application is killed without invocation of any life cycle callback methods. Indeed, *running* is the only state in which an Activity cannot be killed in normal operation (e.g. not plunging the phone into water). So storing data from a running Activity persistently should be done in the life cycle callback method *onPause()*, since in all other states the application might get killed without invocation of any callback method. In case of getting killed, the application is not notified by any callback method, thus unable to execute any actions like storing data or safely closing any connections.

The model representation also reveals that calls to *onCreate()* and *onRestart()* are always immediately followed by a call to *onStart()*. If a developer wants to establish a server connection when starting his application, he should do it in the callback method *onStart()* instead of *onCreate()* or *onRestart()*. This way, the developer avoids redundant code and related problems, e.g., maintenance and evolution of redundant code. The call sequence of *onStart()* immediately being followed by *onStop()* shows that an application can be started and stopped without being running in between. This fact is important for application developers. For instance, an application developer who assumes that an application is always running after being started, might open up a required server connection in the *onResume()*-callback method and shut it down in *onStop()*. However, if the application is not started, but immediately stopped after being started, the application

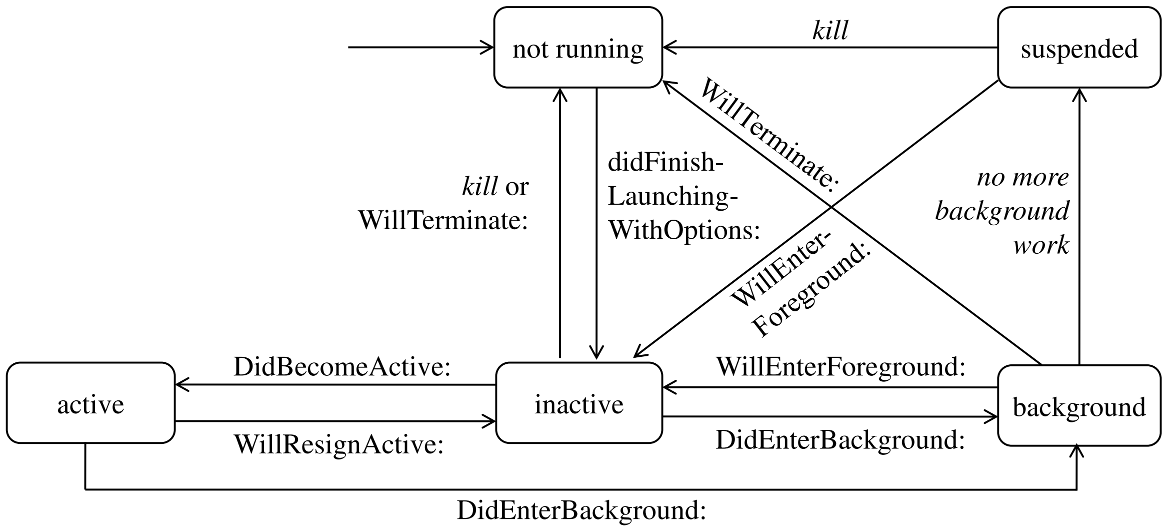

Figure 3.6: Reverse Engineered iOS 4.0 Application Life Cycle

tries to close a connection, which has not been established and probably has not even been defined. That might lead to various errors at runtime, which are hard to track. This state change sequence is not possible in the official life cycle model representation of the Android 2.2 Activity (see Fig. 3.1).

3.4.2 iOS

The rebuilt iOS 4.0 application life cycle is presented in Fig. 3.6 [65]. It consists of five states: *not running*, *suspended*, *background*, *inactive* and *active*. For reasons of clarity, we abbreviated all method names by skipping the prefix *application* before each method and *application:* before *didFinishLaunchingWithOptions:*. So not-abbreviated, the method names are *application:didFinishLaunchingWithOptions:*, *applicationDidBecomeActive:*, *applicationWillResignActive:* and so on. Like in case of the reverse engineered Android 2.2 Activity model, transitions labeled *kill* mark state changes where no callback method is executed and which leave the application in the state *not running*. The transition labeled *no more background work* depicts a state change which is also not accompanied by a callback method call. If an application is in the state *background* and executes some background actions, like downloading data, different state changes and invocations of callback methods are possible than if the application is in the background and inactive. So if an application is in the background and finishes its background tasks, it changes its state to *suspended*, without any life cycle callback methods being executed [55, 65]. The transition from the state *inactive* to *not running* can be taken if the application is killed or if it terminates regularly. In case of regular termination, the callback method *applicationWillTerminate:* is called.

All state labels are taken from the official documentation of the iOS 4.0 application. Further details about their meaning can be taken from the official documentation and our work on reverse engineering the life cycle [31, 65]. The official documentation on the iOS

4.0 application never tries to draw a complete life cycle model representation [55]. It draws parts of the life cycle and a few example scenarios of implementing the life cycle. Since the iOS 4.0 application life cycle is never drawn completely in the official documentation, it cannot immediately be compared to the reverse engineered life cycle model representation. But one interesting observation comparing the reverse engineered iOS 4.0 application to the Android 2.2 Activity life cycle is the fact that iOS 4.0 applications in the state *active* also cannot be killed without the invocation of at least one life cycle callback method. From other states the iOS 4.0 application can be killed without invoking any callback method. For instance, after finishing all background tasks in the state *background*, the application is transferred to the state *suspended* and from there it can be killed without the chance to react on the state change. So regarding the reverse engineered life cycle, important data of an active iOS 4.0 application has to be stored persistently in the callback methods *applicationWillResignActive:* or *applicationDidEnterBackground:*.

3.4.3 Windows Phone

Windows Phone is a recent mobile platform by Microsoft. Windows Phone 7.5 provides two different frameworks for application development: Silverlight and XNA [118]. While Silverlight usually covers applications with text fields, lists and buttons, XNA focuses more on game-like applications, e.g. with the game-typical update-draw loop, canvas and graphics functions. These two application types have different life cycles. In this section, we present the Windows Phone 7.5 Silverlight application life cycle. More details on the process of reverse engineering this life cycle, as well as reverse engineering the Windows Phone 7.5 XNA application life cycle, can be found in [84].

Silverlight applications are notified on Windows Phone 7.5 using events instead of callback methods. To react on such an event, the developer can implement a corresponding method which is executed on arrival of the event. Such methods are similar to life cycle callback methods on iOS and Android. Therefore, and to stay conform with the previous sections, we immediately refer to the methods catching the events instead of the events themselves. For instance, in the following the method *Launching()* is called when the event for launching the application arrives and is caught by a corresponding method handling this event. The same holds for all other methods in Fig. 3.7.

Figure 3.7 shows the result of reverse engineering the life cycle of a Windows Phone 7.5 Silverlight application [84]. On Windows Phone the life cycle is also often referred to as the *execution model* [115]. The life cycle contains six different states: *shut down*, *running*, *obscured*, *dormant obscured*, *dormant unobscured* and *tombstoned*. *Kill* and *tombstone* mark state changes during which no callback method is executed. These state changes are observable while logging with the debugger, since the debugger prints information about the termination of threads and processes. The state *shut down* is the initial state of each application. In this state the application holds no application-related data in RAM. In the state *running* the application is started and in the foreground. If the application becomes obscured, e.g. due to a ringing alarm clock, its state changes from *running* to *obscured*. In case of leaving the application without terminating it, e.g. by backwards-navigation, the application changes its state to one of the two *dormant*-states. If the application is in

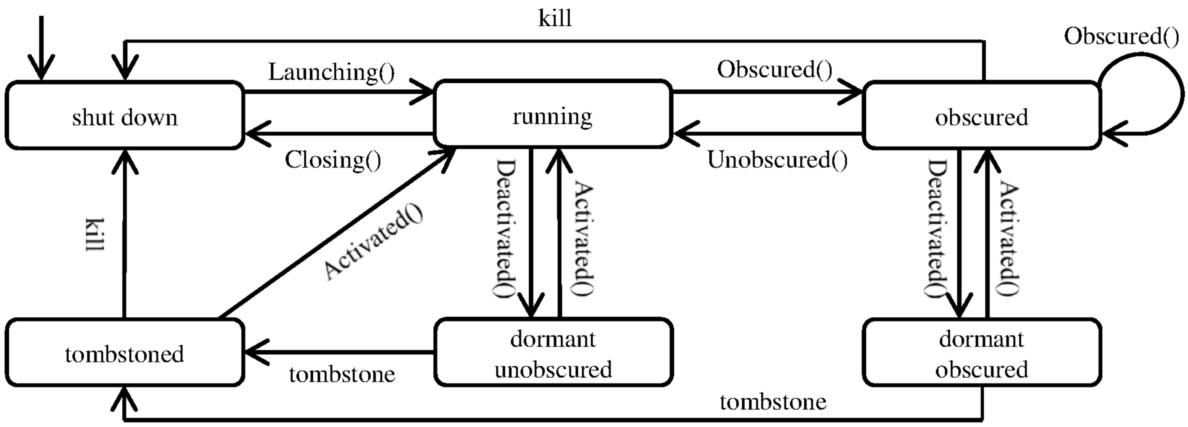

Figure 3.7: Reverse engineered Windows Phone 7.5 Application Life Cycle

the state *running* (not obscured) and switches to the state *dormant unobscured*, it may continue execution in the state *running*, e.g. if the user navigates back to the application. If the application is in the state *obscured* and switches to *dormant*, navigating back to the application leads again to the state *obscured*. In both *dormant*-states the execution of the application is stopped. In the state *tombstoned* the execution of the application is stopped and additionally its threads are terminated. But still a certain amount of data might be stored in RAM to reduce the time of resuming the tombstoned application. This information about the state changes and life cycle behavior of an application are not extractable from the official model [84, 116].

4 Testing Application Life Cycles

This section presents the approach for testing life cycle-related properties of mobile applications. First, it names the challenges which exist throughout current mobile platforms. Second, it sketches the existing approaches and corresponding effort to test such properties. Third, the reasons for choosing a unit- and assertion-based approach are given. Fourth and last, the testing concept is presented in three steps.

4.1 Challenges in Testing Life Cycles

As explained in Sec. 2.1, mobile platforms schedule applications in a specific way which might lead to killing of paused and stopped applications. This killing can be a result of lacking memory resources. Usually the currently visible and active application has the highest priority in scheduling. This ensures access to all resources, a high reactiveness and a good user experience. The applications which are currently not visible or not active often reside paused or stopped. The details vary from platform to platform. On Android, for instance, the platform uses an application stack to assign priorities to the stopped and paused applications [55]. Applications, which have been used by the user most recently, have a higher priority in the stack than applications used some time ago. According to this priority assignments on the stack, in out of memory situations the mobile platform shuts down or kills the stopped and paused applications with the lowest priority first.

There are various scenarios in which the killing of an application or an unintended state change behavior of an application might lead to an undesired behavior of the application. In the following, we give some examples of requirements for different applications, which might be affected by life cycle state changes [69]:

1. An e-mail or any other application may have various input elements (text input, radio buttons, spinner elements, ...). The content of these elements is saved or processed if the user clicks on a corresponding button, like *Send E-mail* or *Save Form*. In case the user gets interrupted by an incoming call after entering some text into text input elements or changing the state of some other input elements, the developer may want to store the state of all input elements. When the application resumes, it shall restore the state of all input elements so that the user can proceed with the application where he was interrupted. The user shall not have to redo any input actions and previously entered information shall not be lost. Such an application behavior follows the user expectations and ensures a good user experience.

2. A *Voice over Internet Protocol (VoIP)* application is capable of establishing phone calls using the Internet connection of the device (like the Skype application[1]). The mobile platform users can usually install such applications next to native phone applications, which use, e.g., *Global System for Mobile Communication (GSM)* networks. If the user is currently having a GSM phone call and he concurrently receives an incoming phone call through the VoIP application, the developer of the VoIP application may intend to not disturb the phoning user and to ignore the incoming VoIP call. Or he even may want to notify the VoIP caller that the callee is currently busy with another phone call, e.g., by sending the caller a busy signal. The developer may also want to notify the phoning user of the incoming VoIP call by playing back a silent signal in the background of his current call to let him decide whether to accept the incoming VoIP call or not. There might be a different reaction desired if the user holds an active VoIP call and he additionally receives an incoming GSM phone call. If the developer holds the opinion that GSM phone calls are more important than VoIP calls, he might interrupt and hold the ongoing VoIP call automatically as soon as an incoming GSM phone call arrives. The VoIP call might also be hold for the duration of the GSM phone call and resumed afterwards. The desired behavior depends on the intentions of the developer and can be implemented by reacting appropriately on life cycle state changes.

3. Mobile game applications are highly demanded on today's platforms [111]. To ensure a good quality of a game application, the developer also has to take care of various sources of interference, like incoming phone calls, incoming messages, ringing alarm applications and shut downs due to low battery. Thus, a game developer may want to pause the game in each case of interruption and probably suppress some distracting system notifications, like sounds and overlay notifications of incoming messages. If the game is interrupted, the developer may not only intend to freeze the game but also to store the state of the game application. The state may include the current game level, the position of game elements, game settings and the current score.

4. *Systems on chips (SoC)* of today's mobile devices often contain many different peripheral hardware components, like *Global Positioning System (GPS)*, approximation and ambient light sensors, Bluetooth and a compass sensor. Some mobile applications access these sensors only if they are currently visible to the user. An application presenting a compass view to the user only needs to access the compass sensor if it is visible to the user (not paused or in background). If such a compass application is paused or stopped, it should release the compass sensor to make it accessible for other applications and to not unnecessarily put load on the battery. On the other hand, some applications require to use sensors and hardware modules even though they are not visible to the user. A GPS tracking application, which continuously tracks and stores the user location, may also have to work while the

[1]See `http://www.skype.com`.

screen is locked or the user is having phone conversations. Releasing resources (like GPS, Bluetooth modules, database connections, ...) and passing ongoing tasks and required resources to background services does often make sense during state changes.

5. Banking, corporate and some chat applications require secure server connections. Such connections have to be set up before usage of the application and shut down afterwards [82]. Depending on the purpose of the connection, developers may intend to keep the connection up even if the application is paused or stopped, or to close it in both cases. If the chat application shall receive messages in the background, even if the application is not visible to the user, the connection has to remain active. If a banking application requires user interaction, it would make sense to shut the connection down each time the application is paused or stopped.

Most parts of all these requirements can be covered by appropriate actions during life cycle state changes of the applications. These requirements, which describe properties and features of the resulting applications, can only be tested if the state of the application is changed. They cannot be tested by leaving the application in one single state. We refer to such requirements, describing properties which go beyond one single state of an application, as *life cycle-related properties*. Such properties are present on most scheduling computer systems, including desktop and server systems. But due to the often specific scheduling policies on today's mobile platforms, these properties gain an increased importance for high quality mobile applications (see Sec. 2.1) [62, 68]. This holds in particular with regard to Sec. 3.4, where we have learned that on many mobile platforms, such as Android, iOS and Windows Phone, applications are not always notified in case of an upcoming killing event. In these situations the mobile applications cannot react appropriately by, e.g., storing data persistently and shutting down connections appropriately before being terminated. If an application developer does not pay enough attention to life cycle-rleated properties, this might lead to an undesired application behavior, data loss, issues in security and bad usability. Referring to the examples above:

1. If the developer misses to store the contents of input elements, like an e-mail text or a selected radio button, and the user resumes the application, the information might be lost. The user may find the application in its initial state, with e-mail fields empty and default radio buttons chosen. Depending on the intention of the application developer, this data loss might not be desired. If this behavior does not correspond to the user expectations, it leads to unexpected application behavior and bad user experience.

2. If the developer of the VoIP application does not specify and implement the behavior of his application, e.g., in case of a concurrent GSM call, this might lead to problems during runtime. For instance, if the user has an ongoing GSM call while a VoIP call arrives, the uncontrolled ring tone for the incoming VoIP call might disturb the phoning user. The VoIP application might experience problems when trying to access the audio hardware, which is currently accessed by the GSM

phone application. Insufficient attention on these life cycle-related properties during specification and implementation might lead to various unintended and unexpected situations, decreasing the experienced quality of the application.

3. If a game developer forgets to freeze a game application or to save the score, it immediately affects user experience. For games it is often important that they do not start automatically when the game application is resumed. When the application resumes, the state needs to be restored (settings and scores set, same view and perspective as before, ...) and the game paused. Then the user can decide, when he is prepared to play on and push a corresponding button.

4. If an application, which is using a hardware module like GPS, does not turn off or release the module, it might lead to wastage of energy and runtime problems. Since energy resources on mobile devices are scarce, an efficient and energy-conscious scheduling of energy consuming hardware modules is necessary. In times of not only more powerful [129], but also more energy-consuming mobile devices [98], applications with a high demand on energy lead to fast discharging of the battery and in the end may be avoided by the user. A music playback application, which currently does not necessarily require the audio module but still holds a lock on it, leads to problems if another application like voice recording tries to access the same hardware module.

5. Applications communicating sensitive data, like banking or secure communication channels, require not only a proper connection setup but also a proper connection shut down to ensure the security of the connection and data [82]. If such an application misses to close a sensitive connection while shutting down, data security cannot be assured [132].

To ensure these life cycle-related properties, they do not only need to be appropriately specified in the specification of the application and implemented accordingly, but also checked during quality assurance. One way to ensure the proper implementation of the specified life cycle-related properties is testing. To be able to test these properties, the developer has to pay attention to further obstacles related to life cycles.

Another very important obstacle is to understand how life cycles behave and how notifications about state changes are communicated from the mobile platform to the applications. Figure 4.1 sketches the connection between incoming events and the subsequent propagation of corresponding life cycle actions. The mobile platform receives the incoming event, e.g., an incoming phone call and decides what to do next (see Chap. 3). Depending on its policy on scheduling applications, the platform may stop the currently active application, shut down a stopped application and kill another application due to low memory (see Sec. 2.1). Then it might start the application handling the incoming event, e.g. a phone application. This asynchronous triggering of life cycle actions and the scheduling policies of mobile platforms may lead to counterintuitive life cycle action sequences. Figure 4.2 presents a callback sequence revealed during reverse engineering the Android 2.2 Activity life cycle [69]. At the beginning of this scenario, application A

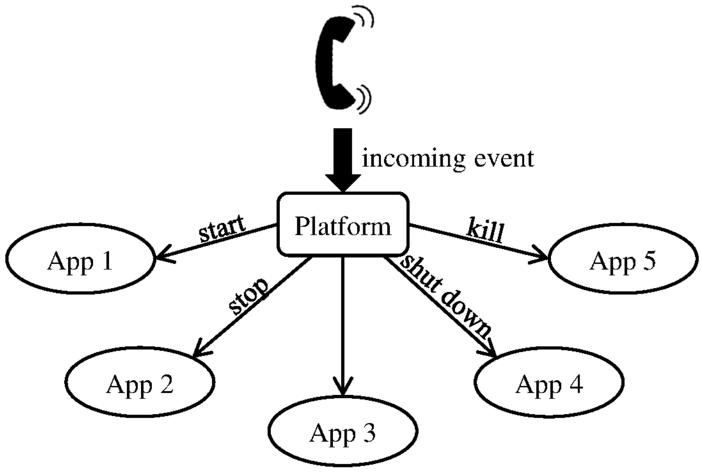

Figure 4.1: Asynchronous Life Cycle Triggering

Application A		Application B	
Callback Method	State	State	Callback Method
⋮	⋮	⋮	⋮
	(running)	(shut down)	
onPause()			
	paused		
			onCreate()
			onStart()
		paused	
			onResume()
		running	
onStop()			
	stopped		
⋮	⋮	⋮	⋮

Time

Figure 4.2: Example of an Android 2.2 Callback Sequence

is running and application B is shut down, not having yet been started. In the next step, application B is started from within application A. Application A could be an application displaying all contacts and the user chooses a contact to establish a phone call. Since the underlying platform knows that the phone application can establish phone calls, it starts the phone application and dials the chosen number. In this scenario application B would be the phone application. To switch the applications, the Android platform first pauses application A by calling *onPause()*. Second, it moves application B from *shut down* to *running* by calling *onCreate()*, *onStart()* and *onResume()*. After application B is already running, application A is stopped. This sequence is counterintuitive as the developer might expect that first application A is stopped completely and then application B is started. It might also lead to various problems. For instance, if application A and B need to access an exclusive resource, which only one application can use at a time, and the developer expects that first application A is stopped, before application B is started. Then he might release the resource in *onStop()* of application A and try to access it in *onCreate()*, *onStart()* or *onResume()* of application B. If the application scheduling from Fig. 4.2 applies to the underlying platform, his application switching scenario will fail. In his scenario application B tries to access a certain resource that application A releases in *onStop()*. Since all three life cycle callback methods *onCreate()*, *onStart()* and *onResume()* of application B are executed before *onStop()* of application A, application B cannot access the resource. Such errors are hard to track and difficult to debug, especially if the state change sequences are not clear and not taught to the developer, as in the given example. Unfortunately these scheduling policies and sequences are not given in the official documents, a fact which significantly complicates a correct implementation of life cycle-related properties and integration of manual testing processes [69].

There are also many other obstacles resulting from incomplete, incorrect and inconsistent official documents on application life cycles (see Sec. 3.1). A documentation on a life cycle not mentioning all available states, whose model representation is ambiguous and where the model is inconsistent with the textual description, does not help the developer to implement the life cycle requirements correctly, indeed it even distracts him [55, 65, 67, 69, 84]. Our personal experience with mobile applications is that many bugs, especially in freely available Android applications, result from life cycle-related issues. One famous example is the news application *Tagesschau* for Android from the German news station *Arbeitsgemeinschaft der öffentlich-rechtlichen Rundfunkanstalten der Bundesrepublik Deutschland (ARD)*[2]. It has been downloaded 1-5 million times[3], but still has an annoying error related to life cycle-related properties: If the user interrupts a running video playback of a news video by pushing the screen lock button, video playback stops as expected. Then the user pushes the button again to unlock the screen. By pushing the button on an Android device, usually the touch screen gets activated and displays a mechanism to unlock the screen. To unlock the device screen, the user then might have to type in a code, draw a gesture or simply move one item over the

[2] See `http://www.ard.de`.

[3] See `https://play.google.com/store/apps/details?id=de.cellular.tagesschau&feature=nav_result#?t=W251bGwsMSwyLDNd`, status on 4th May 2013.

touchscreen to a different place. But right after pushing the screen lock button and before being able to unlock the screen, the application, which was paused by locking the screen, gets active in the background. The news application immediately starts to play back the video, although the user still might be busy unlocking the screen, e.g., by drawing a gesture or typing in a code. But this is not the only life cycle-related issue in this moment: Additionally, each time the application is paused this way and resumed, the video playback starts from the beginning of the video and not where playback was interrupted.

Concluding, there are many problems related to life cycles, their documentation and representations, which make it difficult for developers to appropriately implement their desired life cycle behavior and test it accordingly. But ensuring quality related to life cycle-related properties is key for high quality mobile applications.

4.2 State of the Art

Testing is an important part of the development of mobile applications these days. In an Android market with over 800.000 mobile applications (status January 2013) in its application store *Google Play*[4] [124] and over 775.000 applications in the iOS *App Store*[5] [29], quality is an essential selection criterion for users. As explained in Sec. 4.1, application life cycle-related properties play an important role for the user experience, data persistence, functionality and usability.

To ensure the quality of today's mobile applications, many SDKs come with integrated debugging, simulation and testing tools (Android SDK[6], iOS SDK[7], Windows Phone SDK[8], ...). Further tools for ensuring quality are available through corresponding communities (Robotium[9], Testing with Frank[10], ...). A large group of all these tools focuses on user experience. All current mobile platforms provide a simulator or emulator which tries to execute and display the application similar to a real device. The user can also simulate user interaction with the simulator or emulator and often even execute more complicated gestures, e.g., pinching and spreading with two fingers. The simulators and emulators usually also have access to the Internet through the underlying platform, can simulate incoming calls, rotate the display, simulate GPS activity and so on [102].

Another large part of the integrated testing tools aims on functional testing, like unit testing. For instance, to test Android applications the developer can use JUnit 3 tests [131], enriched by Android-specific functionality [8]. To test iOS applications, the developer can use the integrated *Cocoa Touch Unit Testing Bundle* [30]. The open source Android community tool *Robotium* allows advanced functional testing on user experience level [81]. An excerpt from a functional test case with Robotium for an Android calculator

[4]See `http://play.google.com`.
[5]See `http://www.apple.com/iphone/from-the-app-store`.
[6]See `http://developer.android.com/sdk`.
[7]See `http://developer.apple.com/iphone`.
[8]See `http://developer.windowsphone.com`.
[9]See `http://code.google.com/p/robotium`.
[10]See `http://testingwithfrank.com`.

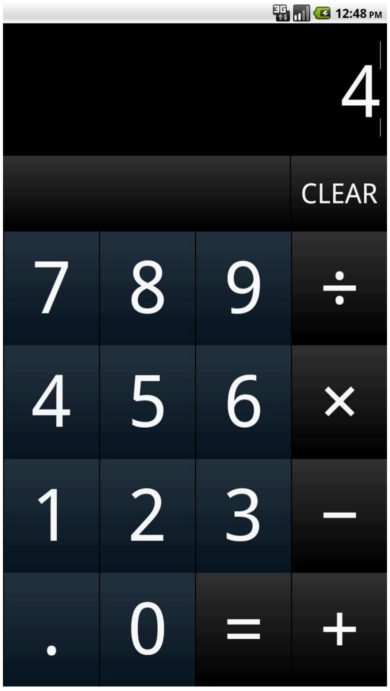

Figure 4.3: View on the Android Calculator Application

application[11] presented in Fig. 4.3 is given in Listing 4.1.

Listing 4.1: Excerpt from a Robotium Test Case for the Calculator Application

```
solo.clickOnText("2");
solo.clickOnText("+");
solo.clickOnText("2");
solo.clickOnText("=");
Assert.assertTrue(solo.searchText("4")); // should succeed

solo.clickOnText("CLEAR");
Assert.assertFalse(solo.searchText("4")); // should succeed
```

Robotium provides a quick way to write test cases by immediately addressing the visual elements the user sees. Without Robotium, the command *solo.clickOnText("2")* had to be hard coded in many lines of code. The visual element, displaying the number *2* to the user, had to be found by traversing through all visible elements and then the element had to be clicked. Robotium encapsulates all these steps and actions by using the various tools and possibilities which the Android platform provides for developers and allows to quickly write human readable test cases. Despite the availability and high quality of testing tools for applications of today's mobile platforms, none of them focuses on the specific needs of testing life cycle-related properties. For instance, the Android-extended JUnit 3 test cases cannot be used to assume that the content of a text field contains

[11] See `http://code.google.com/p/robotium/wiki/RobotiumTutorials`.

the same text when the application is resumed after being killed in between. The JUnit framework for the Android application is not targeted to keep and check assertions of an application which is stopped or killed in between.

Nonetheless, our experience with the available tools is that especially the testing and debugging tools for Android and iOS are mature and elaborated. Additionally, they all contribute features, which at least allow developers to test manually for life cycle-related properties. For instance, some allow simulation of incoming calls, manually shut down and resume applications, trigger life cycle state changes due to device rotations and so on. But if the developer wants to check a simple life cycle requirement, like if the content of a text input element remains persistent when stopping and resuming the application, the developer has to do it manually. Therefore, he can start the application, type text into the text input element, manually execute an action which triggers the desired life cycle behavior, like navigating to the home screen and resuming the application, and by looking on the text input element, decide whether the actual content meets the requirements or not. This would be the manual execution of a test case for one single life cycle trigger. But as we know from reverse engineering application life cycles, different life cycle triggers might lead to different life cycle state change sequences (see Chap. 3). So the developer would need to redo the test with a different life cycle trigger and to manually decide if in that specific case the outcome meets the requirement, by watching the outcome on the screen. During *beta testing*, this manual testing is sometimes also outsourced or extended to testing by potential users [58]. This user acceptance testing method is in the mobile area today especially driven by Apple [28].

Our experience is that life cycle-related properties are rarely tested during application development. The expected life cycle behavior is often not even specified in the specification of an application, which builds an obstacle to developers intending to test for life cycle-related properties. Other obstacles are the lack of clear guidelines for testing life cycle-related properties, ambiguous life cycle models and corresponding tool support, which makes manual testing for life cycle-related properties a time consuming task and keeps application developers from doing so [55, 84]. In the following, we present an approach which guides developers of mobile applications towards testing of life cycle-related properties and builds the groundwork for platform-specific life cycle testing tools.

4.3 Approach

Each testing approach needs a clearly defined context. The potential factors of influence during a test need to be identified and, if possible, prevented. The first part of this section defines the context and potential factors of influence for this testing approach and classifies the approach among other, well-established testing approaches. Thereby it also draws the limits and potentials of this approach on an abstract level. The second part of this section defines what a positive and negative testing result of this testing approach is and how the developer can specify the expected results. Therefore, we use, combine and extend in both parts well-known and widespread testing techniques: Unit testing and testing with assertions.

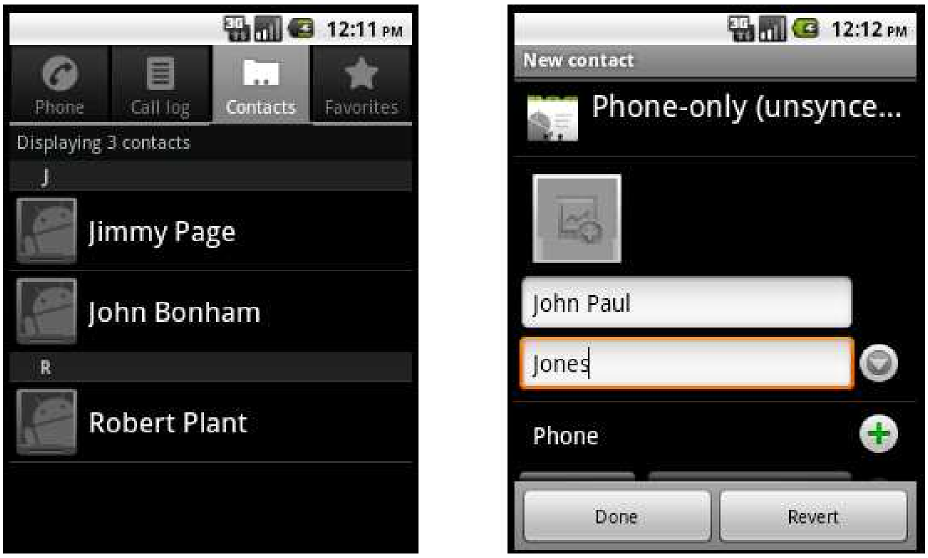

Figure 4.4: Two different Components of one Android 2.2 Application

4.3.1 Unit-based Approach

Applications for today's mobile platforms are built up as modules, like Activities on Android and *applications* on iOS [101, 105]. Each application has usually access to its own private memory area where it can store sensitive data without other, potentially harmful applications accessing it. On the current mobile platforms it is common style that one mobile application has its user interface and view components integrated, e.g., following the *model-view-controller (MVC)* architecture [104, 117]. Further, each application has often its own privileges which describe that the application requires access to the Internet, camera, GPS and so on. On Android this information is stored in the application-specific *manifest* file [45].

One application might also consist of multiple viewable and executable components. For instance, a Windows Phone application might contain multiple pages and an Android application might consist of multiple Activities. Such components can be stand-alone independent executables like Activities on Android, or belong to an executable superordinated component like views in Windows Phone Silverlight applications. Figure 4.4 presents an example of an application consisting of two stand-alone executable components. The example is taken from the Android platform and shows the Android 2.2 contacts application. The left screenshot presents a list of all contacts. In case the user intends to add a new contact, he clicks the menu button and selects the *New contact* option. As a result, the current contact list Activity starts another Activity, presented in the right screenshot of Fig. 4.4. The contact list Activity leaves the state *running*, the platform sets the next Activity as *running* and presents it to the user. The life cycle state changes in the background can be similar to the application switch presented in Fig. 4.2. In the second Activity the user can enter information about a new contact and navigate back to the previous Activity using the buttons *Done* and *Revert*. These two Activities are parts of the contacts application. But on platform level, they are two stand-alone

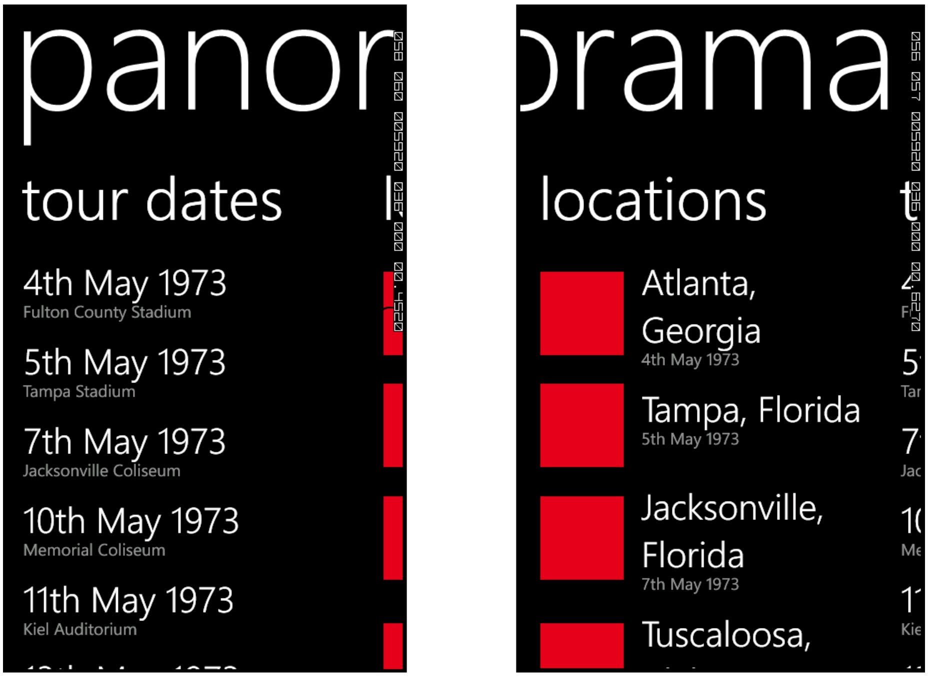

Figure 4.5: Two different Views on one Windows Phone 7.5 Application

executable Activities. Each has its own life cycle and its own state. While the current scroll bar position of the contacts list is part of the state of the contacts list Activity (left screenshot), the entered name and surname of a new contact is part of the state of the second Activity (right screenshot).

In such composed applications, the different single components, e.g. Activities on Android, usually have clear separate functions and goals. If one component intends to make use of a function of a different component, e.g. contact list application needs to add a new contact or requires the result of that function, e.g. get information about a new user and present it in the contacts list, the platforms provide techniques to pass data to a starting component, or receive data when the component shuts down. On Android, this is done by passing a data bundle when starting an Activity.

A different example of a composed application is presented in Fig. 4.5. On the left side it presents a list of different items and by sliding with a finger from right to left the user is able to view the information presented on the right side. In contrast to the example from the Android platform (see Fig. 4.4), these two components of Fig. 4.5 are not stand-alone executables, but they present different view components of one executable component. Both views share the life cycle of the executable superordinated component. Thus, the state of both components has to be managed by the superordinated executable component, which can for instance be a subclass of *PhoneApplicationPage*.

But independent of the existence of multiple components or one single executable component with different views, usually each active application has one component in the state running. The system controls the whole application by communicating, notifying and triggering life cycle state changes through this single active component [69].

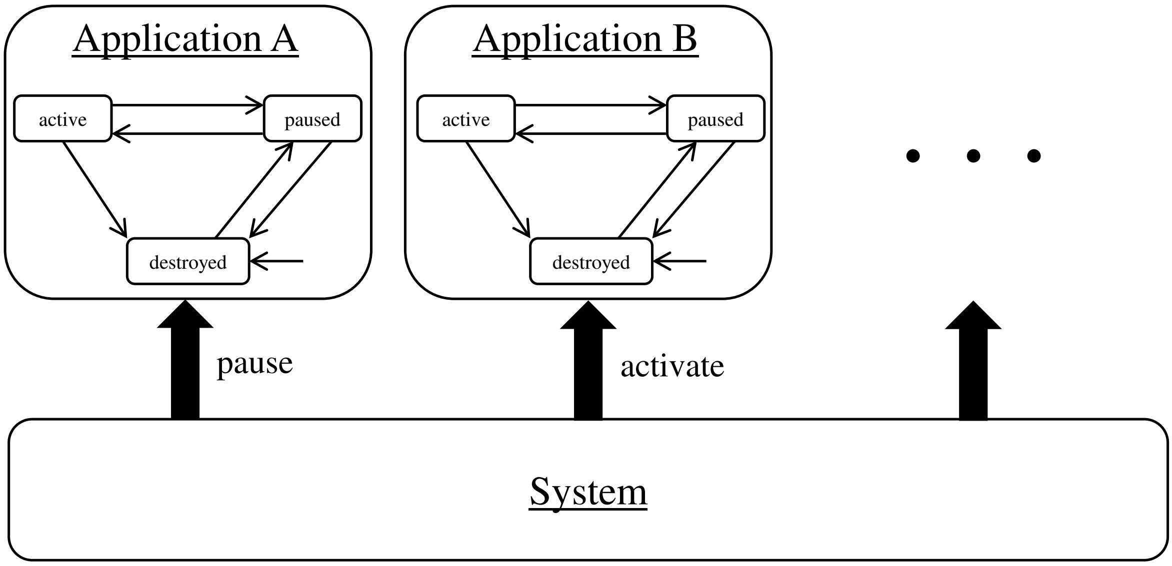

Figure 4.6: Asynchronous Triggering of Life Cycle Events

The mobile platform controls the life cycles of all executable components [69]. These life cycles are independent of each other in the sense that the life cycle actions of one component do not immediately influence life cycle actions of another component. Like on most operating systems, the platform has the overview of all applications and their states and controls the application scheduling [133]. Mobile platforms also trigger starting, stopping and all other state changes of their applications. This way, the mobile platform can ensure that all mobile applications are scheduled with respect to its scheduling policies. Figure 4.6 sketches the separate and asynchronous triggering of life cycle actions by the underlying platform [69]. The mobile platform decides for each running application with an own life cycle when to start, stop, shut down or kill it. The platform is usually the only component building a dependency between the life cycles of two different applications. For instance, application A could be an application to compose SMS in the state *active* and application B is a phone application in the state *destroyed* (see Fig. 4.6). If the user, currently entering SMS text, is interrupted by an incoming call (see Fig. 4.1), the system prompts the active SMS application to pause by calling the corresponding life cycle methods. After that, or concurrently, the system starts the phone application by triggering corresponding life cycle actions. If there is a scheduling policy on the mobile platform, which takes care that only one application can be in the state *active* at a time, the system has to trigger the life cycle changes accordingly. In that case, it would wait until the SMS application has left the state *active* to run the phone application.

There are various platform-specific possibilities for mobile applications to communicate with each other, e.g., through shared resources or asynchronous message passing [43]. Mobile platforms often communicate life cycle actions to applications using asynchronous event handling. Life cycle actions are not communicated between two different applications, but between the mobile platform and the corresponding application. This way, the mobile platform can ensure its scheduling policies, e.g., only one running application at a time,

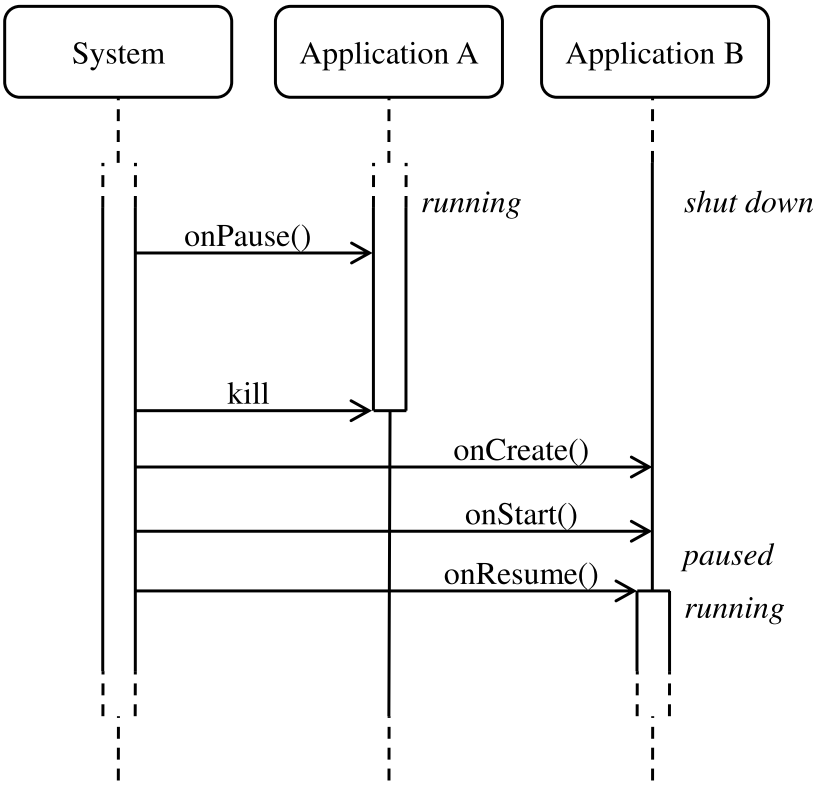

Figure 4.7: Timeout during Triggering of Life Cycle Actions

which application to start next and which one to be killed to free resources. If an application should need too much time to execute a triggered state change, e.g., if the execution of the code in the corresponding life cycle callback method takes too long, the mobile platform usually is able to execute further actions to terminate the application and proceed the state change process [55, 84]. Such actions might be forcing or killing of applications. For instance, if an iOS 4.0 application requires more than nine seconds to execute the code in the life cycle callback method *applicationDidEnterBackground:*, it either can request the system for more time or the application will be terminated [55]. The same holds for the life cycle method *applicationWillTerminate:*. To make this time-dependency between the platform and its applications more clear, the message sequence chart (MSC) in Fig. 4.7 sketches the application switch from Fig. 4.2, but with a timeout. The chart representation is simplified. The system tries to pause application A while application B is still shut down. Application A requires time t to execute its callback method *onPause()*. If t exceeds the maximum time an application is allowed to move into the *paused*-state, the application might be killed by the system. Depending on the configuration of the underlying system, all other remaining life cycle callback methods might not be called when the application is terminated by the system. Some mobile platforms allow to exceed the timeout or to receive requests from the application for more time [55]. During our tests the default timeout of the Android 2.2 platform for stopping an application was measured around 10 seconds.

If an application requires too much time, e.g., for pausing, it might influence the moment in time of executing life cycle methods of other applications (see Fig. 4.7). This time dependency is also handled through the mobile platform, which in the end decides with respect to its scheduling policies, when an application will be killed and the life cycle actions of a different application proceeded. Like the other mentioned dependencies, this does neither lead to direct dependencies and influences between two or more applications. The common denominator, regarding the life cycles of the applications, remains the underlying system.

For this reasons, we regard in our approach components with own life cycles (a whole application or executable application components) as units. We take advantage of the fact that the underlying platform is the main interface and factor of influence for each unit, regarding their life cycle actions. In our approach, we neglect that an application can timely influence the life cycle actions of another application. If this should be of importance for one or another life cycle-related property, the developer can check the time requirements by querying and recording of timestamps during testing.

This unit-based approach does not comply with unit testing in the original sense, i.e. where the smallest testable component or method is tested in isolation [69, 83]. In the following testing approach, a unit can be a whole application with an own life cycle. This is minimal in the sense that it has only one life cycle and the only source of influence, regarding the life cycle, is the underlying platform. To test such an application for life cycle-related properties, it might be necessary to shut down and restart the whole application during the execution of a test case. In Sec. 4.4, we explain how such units can be used for testing life cycle-related properties.

4.3.2 Using Assertions

A life cycle-related property is not satisfied or unsatisfied in general. This often depends on the context. For instance, if the user is interrupted during the usage of a web browser application by an alarm clock application, he might expect to see the same web page again after resuming the web browser. But if the user checks today's weather with a weather application, closes it and reopens it one week later, he may expect to see the weather of the actual day, not of the day last week. Even for one single application the expected life cycle behavior might differ, depending on the user actions. For instance, if an Android user quits the web browser application using the *Back*-button (which is often associated on Android devices as quitting an application) and resumes the application after some time, he might expect to see the welcome page of the browser and the cursor focused in the empty URL input text field. But if the user quits the browser application using the *Home*-button (which is often associated on Android devices as pausing the application) and resumes the application afterwards, he might expect to see the same browser page as when he left. The application behavior and user expectations are part of the specification of the application, against which they have to be checked for conformance during testing. One possibility to test such life cycle-specific requirements and properties is to use assertions. The developer specifies the expected behavior for each test case by defining corresponding assertions. These assertions are checked during runtime. This technique is used, e.g., in JUnit testing and widespread [96]. We use this technique to specify and test the different and specific life cycle-related properties.

4.4 Concept

This section presents a conceptual approach to test life cycle-related properties of mobile applications. It tries to be platform independent. The conceptual procedure is kept generic, to be easily extensible and applicable to different platforms, mobile or not. Extensions could result in new assertion categories, depending on the field of application. Other application areas of this approach could be desktop or server systems. Nonetheless, by choosing examples from the mobile application area, the concept tries to be sufficiently precise and concrete to be immediately applicable to mobile applications. An application of this conceptual approach is presented in Sec. 5.2.

4.4.1 When to test Life Cycle Properties?

Requirements on life cycle-related properties go along with state changes (see Sec. 4.1). The properties addressed by our approach do not cover requirements of an application in one single state, e.g.:

> In the state *running*, the content of the input text field has to be stored at some point in time.

Instead, they cover requirements which imply state changes, e.g.:

> When the application leaves the state *running*, the content of the input text field has to be stored.

On code level, state changes on today's mobile platforms can be identified by calls to corresponding application life cycle callback methods or incoming life cycle notification events. But some state changes do not lead to such notifications. This includes, e.g., the killing of a suspended iOS 4.0 application (see Fig. 3.6) and changing the state of a Windows Phone 7.5 application from the state *dormant obscured* immediately to *tombstoned* (see Fig. 3.7). But the most regular state changes, caused during normal operation and not, e.g., due to out of memory or low battery situations, lead to corresponding state change notifications by callback methods or events. We make use of this fact to inject the test code for life cycle-related properties into the corresponding life cycle callback methods of the application under test (AUT) [69].

Using callback methods to test life cycle-related properties complies with our unit-based testing approach (see Sec. 4.3.1). One test case, addressing a single unit, requires only attention to the callback methods of this unit. The test code is injected into these methods. It is the responsibility of the developer to control and reduce the impact of the test code injected into these callback methods of the AUT. For instance, the injected code should not have such a long execution time that the overall execution time of the callback method exceeds the permitted time and thus leads to killing the AUT (see Sec.

4.3.1). Regarding our measurements, this would be around ten seconds on Android and around nine seconds on iOS 4.0 [55]. The injected testing code should be minimal in the sense that, e.g., it looks for the required information like the content of a text input element and stores them. But it may not modify any application data, so not modifying the content of a text input element.

The callback methods, relevant to the current test case, can be extracted from the official documentation or by reverse engineering the application life cycle (see Chap. 3). In the following, we use the reverse engineered models, where possible. To make our procedure in the following more clear, we use the contacts application from the Android 2.2 platform as an example. We focus on the Activity responsible to add a new contact (see Fig. 4.4 right). A requirement for a life cycle-related property on this Activity can look as follows [69]:

> If the application is paused/stopped/shut down without interaction of the user (he did not click any of the two buttons) and resumed afterwards, the content of the input element shall be the same as when the application was left.

This requirement avoids data loss in the text input element. This part of the specification addresses directly the different state changes: pausing, stopping, shutting down and resuming. Thus, the corresponding callback methods, invoked during these state changes, are relevant for the test case. How this methods can be used to test life cycle-related properties is introduced in the next section.

4.4.2 How to test Life Cycle Properties?

In Sec. 4.3.2 we introduce the idea of using assertions to test life cycle-related properties. Assertions can be used to specify the expected behavior during a test case. Assuming, we have the same requirement as given above [69]:

> If the application is paused/stopped/shut down without interaction of the user (he did not click any of the two buttons) and resumed afterwards, the content of the input element shall be the same as when the application was left.

Aligned to the usage of assertions in JUnit testing, a corresponding assertion on code level can look as presented in Listing 4.2 [69]:

Listing 4.2: Example of an Assertion on Code Level

```
assertThat(oldTIEContent, equalTo(currentTIEContent))
```

The variables *oldTIEContent* contain the previous and *currentTIEContent* the current values of the same text input element. This line of code assumes that the recent content of the text input element is equal to the current content of the element. To be able to check this assertion, on one side the old value of the text input element and on the other side the current value must be known. These may be two different points in time.

This leads to the question, at which position in code assertions about life cycle-related properties can be defined and at which position they can be checked. We use the term *to define an assertion* for the process of defining and storing assertion data with the purpose of checking it later during runtime. For instance, the assertion above can be defined when the value of *oldTIEContent* and the position in code where it will be checked (later during runtime) is known. We use the term *to check an assertion* if at some position in the code after its definition, the assertion is checked since all necessary information (either stored or currently obtainable) are available. Given the example above, defining and checking an assertion might happen at significantly different points in time. For instance, if the application is shut down in between.

Positions of defining and checking assertions, mainly depend on the specification of the application [69]. If we assume for the AUT the life cycle given in Fig. 2.1 and the following requirement on its life cycle behavior [69]:

> If the application is paused/stopped/shut down without interaction of the user (he did not click any of the two buttons) and resumed afterwards, the content of the input element shall be the same as when the application was left.

Then we can identify certain life cycle callback methods as relevant for the corresponding test case. Regarding the life cycle model representation in Fig. 2.1, the active application can be paused and destroyed. In case of pausing the application, the callback method *pauseApp()* is called. In case of transferring the application from the state *active*, in which the user can interact with the application as addressed by the specification, immediately to *destroyed*, the callback method *destroyApp()* is executed. When resuming the application, i.e. changing back to the state *active*, in both cases *startApp()* is called. When the application leaves the state *active*, the content of the text input elements is available in the callback methods *pauseApp()* and *destroyApp()*. Referring to the assertion code snippet above, this would be the position in the code where the value of the variable *oldTIEContent* is available. The value of the variable *currentTIEContent* is available in the callback method *startApp()*.

In case of injecting code into life cycle callback methods, the developer has to be aware of the fact that many applications use these callback methods for restoring and saving the state of the application. For instance, *startApp()* can be used by the developer of the application to restore the content of the text input element. For this reason, code that defines and checks life cycle-related properties should be placed at the very end of the life cycle callback methods.

Concluding, the relevant life cycle callback methods to check the life cycle-related property above, are: *pauseApp()*, *destroyApp()* and *startApp()*. The position in code and time of execution of definitions and checks of assertions may diverge. In this case, the assertion has to be defined in the callback methods *pauseApp()* and *destroyApp()*, since at this two points the content of the input text field (*oldTIEContent*) is known. These methods are called during pausing/stopping/shutting down of the application, which is part of the requirement. During resume of the application, which might be significantly later in time, the assertion can be checked in *startApp()*, where the new content of the text input element (*currentTIEContent*) is known.

Since the moments of defining and checking assertions might diverge in time and code position, the developer has to ensure that data of a defined assertion remains available and consistent, until it is checked. Depending on the underlying platform it might be necessary to persistently store defined, but not yet checked assertions. As outlined in Sec. 2.1 and 4.1, it is possible that on some platforms all application data might be removed from RAM when the application enters a specific state (e.g. when it is destroyed, shut down or killed). In test cases which include passing of such a state, defined assertions stored in RAM might be lost, too. To avoid this, persistent storing of defined assertions is necessary.

As introduced in Sec. 4.3.1, on many mobile platforms each application has access to a dedicated, application-specific and private memory area (database, flash storage, ...). This is also the case for platforms keeping their applications for security reasons in a sandbox environment [50, 108]. If applications on today's mobile platforms are hold in sandboxes, they usually have access to a clearly defined private and, e.g., to some other shared memory areas [18]. The shared memory areas are often used to exchange data with other applications. Memory access to areas outside of the sandbox are usually prohibited by the platform. This way, the platform ensures that an application cannot execute harmful operations by modifying memory areas, e.g., of other applications. The private memory areas of applications can be used to store assertions persistently. This ensures that the assertions are available during runtime of the application, the application can access them and they are not modified by other applications. It is the responsibility of the developer that the usage of the private memory area of the AUT does not influence the regular execution of the application without the injected life cycle test code. If an influence cannot be completely excluded, it has to be kept minimal and under strict control. The developer has to be fully aware of the influence during testing. So what should not happen, e.g., is that the AUT runs out of private memory due to too much stored assertion data. Or a test case trying to store assertions outside of a sandbox without corresponding permission. If the AUT persistently stores its assertions before a shut down, it can restore them after resume. Referring to the example above: If the AUT persistently stores the value of *oldTIEContent* in *destroyApp()*, it can restore the value when resuming in *startApp()* and check the assertion by comparing the restored value *oldTIEContent* to *currentTIEContent*.

The overall process of how to test life cycle-related properties looks as follows [69]:

1. The developer needs to extract all relevant life cycle callback methods/events for a test case from the specification of the application. These methods are used for defining and checking assertions. If the callback methods relevant to the test case are obvious from the specification, like in *If resuming from an incoming call ...*, the developer can identify the corresponding method names in the reverse engineered models (see Chap. 3) or from the official documentation and guidelines of the platform. If it is not clear which life cycle methods are affected by the specification, like in *Entered text field data might never get lost ...*, the user has to analyze

the specification of the application with respect to the underlying application life cycle. The purpose of this analysis is to find out which life cycle state changes and corresponding callback methods might be affected by this part of the specification. In the example *Entered text field data might never get lost …*, it would be all transitions leaving the state in which the user may enter data into the text field.

2. In a second step, the developer needs to identify life cycle triggers which test the properties named in the specification. If this is obvious from the specification, like in *If resuming from an incoming call …*, he can execute the triggers immediately. In this case, it would be interrupting the running AUT by placing a call to the device and resuming the AUT afterwards. If it is not clear from the specification which triggers are necessary to test a requirement, the developer can make use of the trigger catalogs from reverse engineering the application life cycles [65, 67, 84]. As presented in Chapter 3, during the process of reverse engineering application life cycles a catalog with life cycle triggers, which lead to different sequences of life cycle state changes of an application, is composed. The number of triggers in the catalog is limited (see Chap. 3) [65, 67, 84]. The developer can execute all triggers from the catalog. Depending on the test case, it might also be necessary to execute different combinations and sequences of the triggers in the catalog.

3. After the definitions and checks of assertions are placed in the callback methods and the life cycle triggers relevant to this test case identified, the developer needs to execute the triggers, manually or automatically. During execution of the test case, the assertions are defined (as the values of the variables are known) and checked.

4. In the last step, the results of the test case, the outcome of checking the assertions, have to be collected and presented to the user. Therefore, each mobile platform provides different possibilities, as e.g. logging functionality or debugging streams.

The remaining question is what life cycle-related properties can be tested with this approach.

4.4.3 What can be tested?

In general, components and properties can be checked, whose state or value might have changed during (at least one) state change [69]. They can be checked if as a consequence of the state changes callback methods have been executed. The checks might include, e.g., if a Bluetooth module has been turned off in between and was not turned on during resuming or if the previous selection of radio buttons is still selected or restored.

Which properties are available on a specific mobile platform depends on the platform and the device itself. For instance, *near field communication (NFC)* is natively supported on the Android platform from version 2.3 on, so not on Android 2.2 and previous devices. The same holds for multiple camera-support for Android applications. Some devices with Android 2.3 still may have only one camera available. Additionally, it also depends on the underlying platform whether the available properties can be checked on application level or not. Android devices may support placing calls, but for security reasons an Android application cannot place a call without interaction[12] of the user [102].

In the following, we classify the different life cycle-related properties into three categories [69]. Our approach is not limited to these three categories, but can be extended depending on the device capabilities, platform access and the available properties. According to our experiences, the properties in these three categories are relevant for many mobile devices. It is up to the accessibility of the applications on the platform and the capabilities of the underlying device if these properties are checkable on a specific platform.

- *Data Persistence:* In the ever-changing environment of mobile devices, data persistence is important and can be challenging. For the user, this is often a natural quality of an application. Properties related to data persistence of mobile applications can be categorized into three different subcategories [69].
  - *Volatile data* contains all information not automatically stored by the platform persistently. Depending on the platform and state changes, this includes data from input text elements, spinner element selections and other input elements. This also includes values of variables which are kept, e.g., in RAM and have to be stored by the application itself to remain persistent.
  - *Persistent data* includes data from databases and permanent storages (hard drive, flash, memory card, ...). Data on persistent storages can often be modified during runtime by different applications and services. For instance, a service which updates stock data in a database table might run in background, although the application with the user interface is not visible. At an initial start of an application, corresponding databases and tables might have to be created and initialized. A different application might modify the folder structure on a flash storage so that important folders and files are moved. These changes might be relevant for resuming applications and probably have to be checked accordingly.
  - *Content providers* are often realized on mobile platforms as services [69]. They provide functionality and data for other applications, which goes along with the sandboxing approach (see Sec. 4.4.2). A content provider might, for instance, provide contact information of the device to its applications. An application which intends to access the data provided by a content provider might require a corresponding permission. With the permission it might be able to ask for a specific data set, e.g., all available forenames, surnames and e-mail addresses. Due to its function as a service, data of a content provider might often be manipulated by different applications. For a life cycle test it might be of interest, e.g., if the contact data of the current application is still up to date or if it has to ask the content provider for the actual data set, as the data might have changed since the last execution of the application.

[12]This might be possible after rooting Android devices.

- *Hardware Status:* Today's mobile devices might have various different hardware modules integrated on their SoCs: GPS module, front camera, back camera, proximity sensor, ambient light sensor, display, compass, *inertial measurement unit (IMU)* and so on. Depending on the mobile application, the accessibility and availability of these modules might be important. An application which requires actual GPS data, like a GPS tracker, might want to check at startup if a GPS module is available or switched on or off. For reasons of security, some platforms require user interaction to turn the GPS module on. In this case, it would be necessary to provide the user a corresponding notification at application startup.

- *Connection Status:* Depending on the available hardware of a mobile device, it might have different types of connections: Bluetooth, Wi-Fi, NFC, radio-frequency identification (RFID), connections over TCP/IP to servers and services and so on. During the time an application is not active, the status of connections might change. It might be necessary for an application to check the status or reconnect a connection when resuming the application. An online chat application with a connection to a corresponding chat-server might require to reconnect to the server each time the application becomes active.

As mentioned above, whether properties are available and checkable on a specific platform, depends on the platform itself. In Sec. 5.2.3 we give an overview of the checkable properties on the Android 2.2 platform.

5 Case Study on Testing Life Cycles

This section presents the application and the results of applying the approach to the mobile platform Android. Android is a mobile operating system, initially designed for mobile devices like smartphones and tablets. At the moment of writing this thesis, Android holds the largest market share in the area of mobile smartphone sales (see Fig. 5.1) [76]. There are multiple reasons for the popularity of Android and why it could be able to defend its current position on the smartphone market in the future. Just to name a few: Although Android is mainly managed by Google, its development and evolution is also influenced by the Open Handset Alliance, which is a consortium of software, hardware and telecommunication companies like T-Mobile, Wind River Systems, Broadcom Corporation, HTC, Samsung Electronics and many others [25]. In contrast to other strong competitors on the smartphone market, like Apple's iOS and BlackBerry OS, by BlackBerry Ltd., Android is not limited to devices of one single manufacturer, but is capable to run on various devices by different manufacturers, like Samsung, Motorola and HTC. Further, Android is also available in different release versions and configurations, which allow it to be deployed to low, as well as highly capable recent devices. This flexibility and adaptability allows Android to serve a large market. Another reason for the popularity of Android is the fact that it is available open source [9]. Developers and the Android community are able to modify, extend and port the operating system. Thus, Android is also available on other devices like watches and tv boxes [80, 119].

The fact that Android is open source, helps when reverse engineering life cycles of the Android platform. Confusing information about life cycles in the official documentations and guidelines can be cleared by inspecting the source code. We made use of this opportunity when reverse engineering the Android 2.2 Activity (see Chap. 3). Comparing our case studies with various platforms and different components like Android services [55, 67], iOS applications [55], iOS services [55], Windows Phone 7.5 Silverlight and XNA applications [84], it is our experience that Android provides the most access to the platform, hardware modules and applications. Thus, on Android we have been able to check the largest number of different life cycle-related properties [102].

Android's market share is growing continuously through the last few years [130]. Together with the fact that through the Open Handset Alliance various companies are constantly in the process of developing and improving Android, this leads to many new releases and short release cycles [1]. For instance, between May 2010 and May 2011 the Open Handset Alliance released six different versions of Android (2.2, 2.3, 2.3.3, 2.3.4, 3.0 and 3.1). Additionally, the SDKs and official tools have their own release cycles, which even may be shorter [14]. This leads to problems if developing tools with the intention to serve all available Android versions. As a result of this case study, we present the tool AndroLIFT. Initially we tried to keep up with the different releases of SDKs, tools

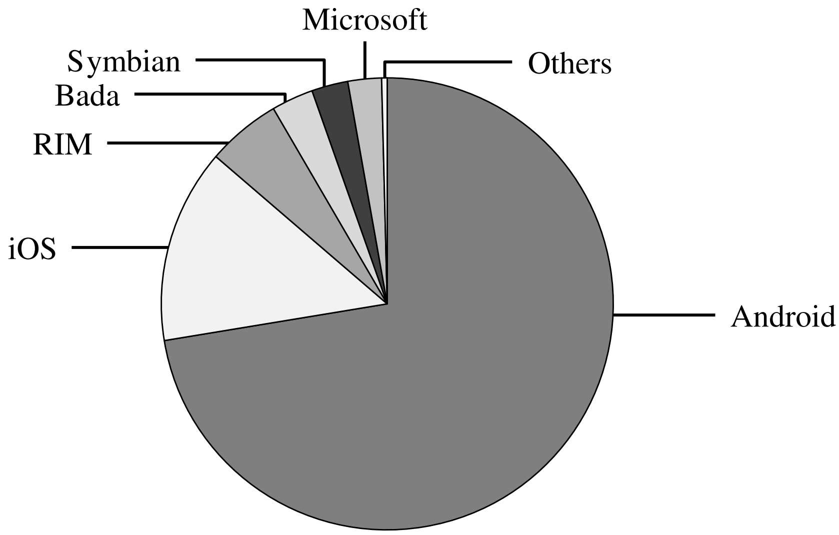

Figure 5.1: Worldwide Mobile Device Sales to End Users in Q3 2012

and Android versions. But finally we did not had enough manpower to continue that way and to proof the applicability of our testing approach in a case study, this is not necessary. Thus, the case study and the resulting tools are based on Android 2.2, but both is applicable and adaptable, with corresponding adjustments, to other versions, too.

At the time we started to work on the case study, Android 3.2 was already available. But nonetheless, Android 2.2 had still a large market share, regarding all available Android versions (see Fig. 5.2) [1]. Android 2.2 was the leading Android version for a long time. A reason might be that the subsequent Android versions were released soon after Android 2.2 and in short periods. They also did not introduce major new features, but, e.g., Android 2.3 provided support for NFC and Android 3.x was optimized for execution on tablets, whereas Android 2.2 worked on tablets, too [5, 6]. Due to the relevance and distribution of the Android 2.2 version, our case study focuses on this version. Probably the case study is also immediately applicable, or with small adjustments, to subsequent Android versions, as to our best knowledge these versions do not modify the Android Activity life cycle on application level.

This section is outlined as follows. First, it introduces the architectural and component-based setup of the Android platform. The focus lies on the Android 2.2 Activity, which is introduced in detail. In the second part, the conceptual approach of testing application life cycle-related properties is applied to the Android 2.2 Activity. The procedure is aligned to the conceptual approach presented in Sec. 4.4. The final part presents the tools resulting from this case study.

5.1 Android Platform

The Android platform consists of a number of different components, arranged in layers. This section first gives an overview of the overall system architecture of Android. It

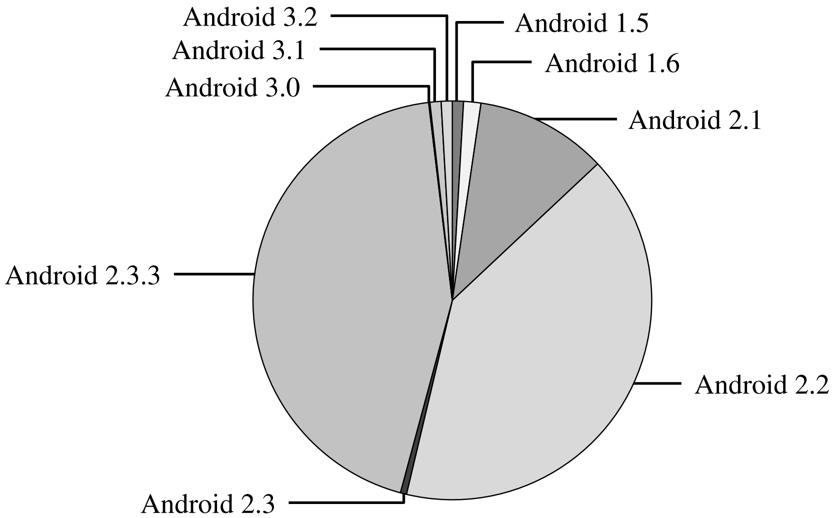

Figure 5.2: Distribution of Android Versions in October 2011

names and focuses on the relevant components for applying our testing approach. The core component in this case study, the Android 2.2 Activity, is introduced in detail in the second part of this section.

5.1.1 Overview

The Android platform consists of a layered architecture, presented in Fig. 5.3 [38]. Android 2.2 is based on a Linux 2.6 kernel. The kernel provides hardware-near functionalities to Android, as drivers, process management and network stack [16]. It also manages the execution of the *Dalvik Virtual Machine (VM)*, which is part of the *Android Runtime* layer. Android applications are predominantly written in Java and executed each in a separate Dalvik VM. Android Runtime also manages access to the core Java libraries [106]. Some C/C++ libraries which cover core system functionalities and hardware access, are part of the *Libraries* layer. These libraries cover graphics (*OpenGL|ES*), display (*Surface Manager*), audio, video (*Media Framework*) and database access (*SQLite*). The *Application Framework* layer makes use of the *Libraries* and *Android Runtime*. It bundles and encapsulates the functionalities for usage by the top application layer. For instance, it manages different layouts and views (*View System*) like buttons, text boxes and grids, provides access to resources (*Resource Manager*) like graphics, strings and audio files and handles the execution of components like *Content Providers* and Activities (*Activity Manager*). The topmost *Applications* layer contains all Android applications. These include applications installed by the user as well as core Android applications, like *Phone*, *Contacts* and *Browser* applications.

There are four executable components on Android with an own life cycle [7]:

- *Activities:* An Activity represents a component with a user interface. An application might consist of multiple Activities, each serving a different purpose. For instance,

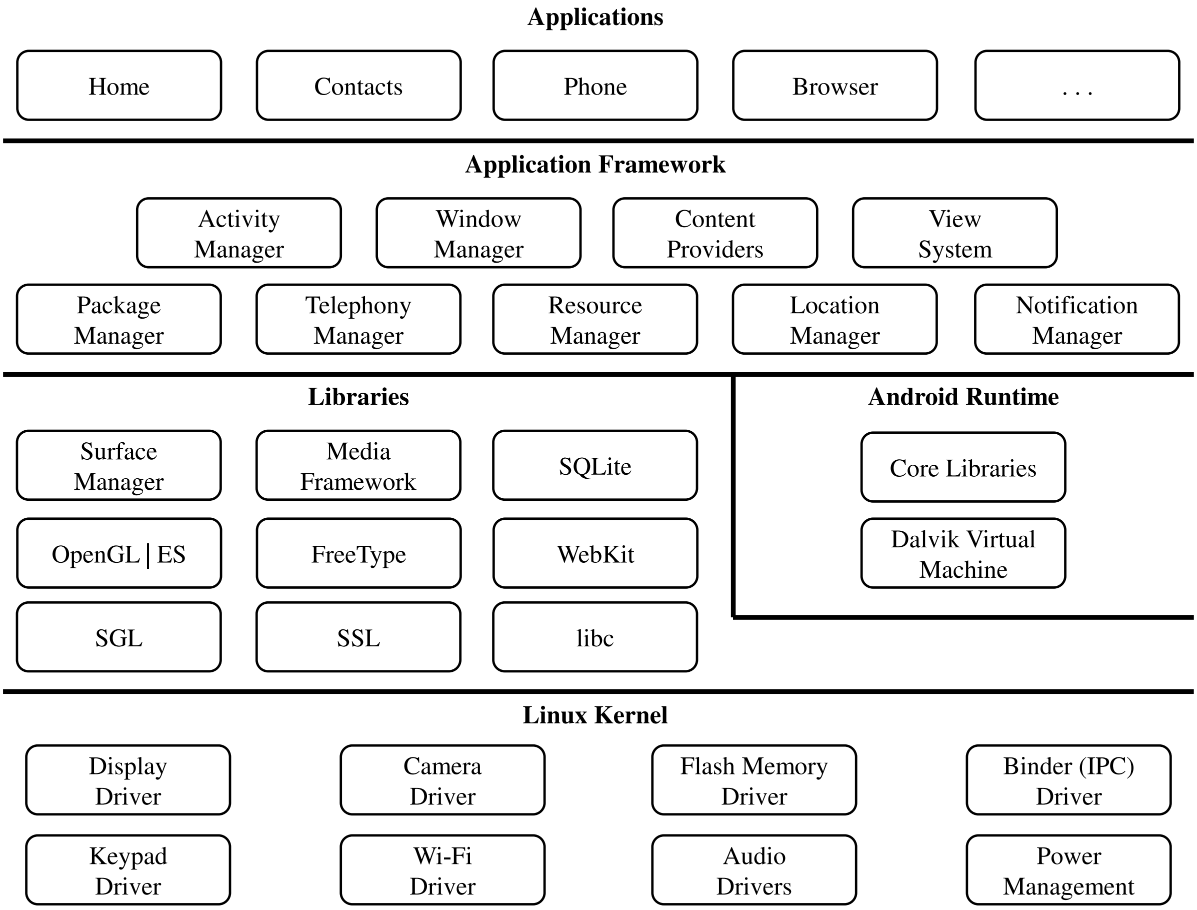

Figure 5.3: Layered Android Architecture

Figure 5.4: Android Status Bar

one Activity might present a list of all contacts and a second one might provide the functionality to edit or add a new user (see Fig. 4.4).

- *Services:* While Activities are mainly active when they are in the state *running* and visible to the user, services are capable of doing work in the background. Typical tasks of services are downloading content from the Internet, or handling an active stopwatch while the screen is locked and turned off. Services on Android might also be bound to an Activity. For instance, a music playback application might consist of an Activity and a service. The Activity displays the user interface of the application to the user. The service is responsible for playing the music. If the user starts playback of a song and navigates to a different application, so that the Activity of the music playback application is not visible any more, the song may still be played in the background by the service.

- *Content Providers:* On Android, content providers are a common way to share and reuse data. A content provider provides a well-defined interface for accessing and sometimes also manipulating a set of data. This data might be, e.g., the contacts information of the Android device or specific data of any other application. Due to the sandboxing approach on Android (see Sec. 4.4.2), one application usually cannot access data of another application. Content providers are one way to properly bypass this obstacle in a well-defined way. They also allow to limit the access and manipulation of data to specific applications with according permissions.

- *Broadcast Receivers:* Broadcast receivers are able to listen to system-wide notifications and react to specific ones. A reaction of a broadcast receiver to a notification of interest could be displaying a notification icon in the Android status bar (see Fig. 5.4) or notifying an application about a system-wide event.

Android applications might consist of an interplay of these different elements. But each Android application with a user interface usually contains at least one Activity component. Due to the importance of this component, we focus in the following on the Android Activity.

5.1.2 Android Activities

The main component and entry point of an Android application with a user interface, is usually an Activity. As presented in Sec. 5.1.1, an Android application might also consist of multiple Activities and even other components. Depending on the size and purpose of the application, a composed application might become complex. The following example highlights the importance and key role of Activities in interactive, composed Android applications. The example in Fig. 5.5 sketches the interplay of different components

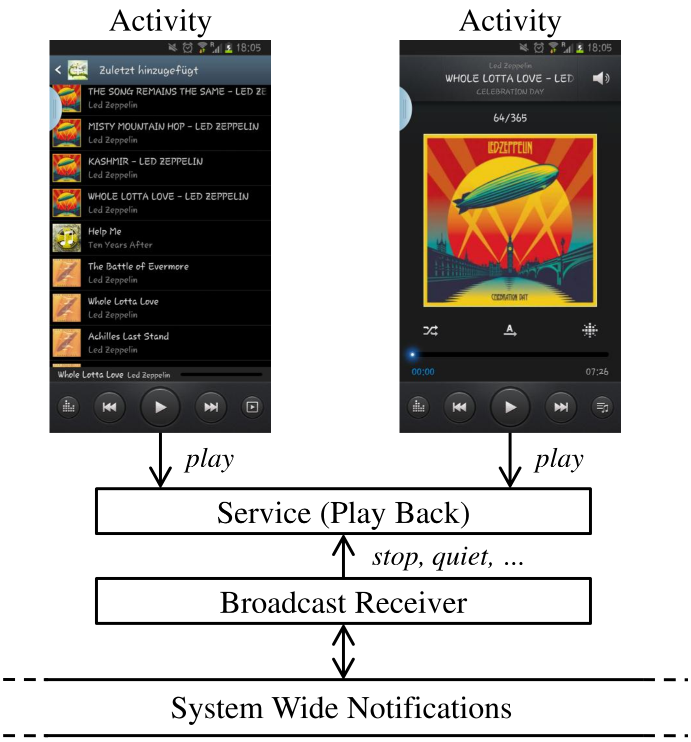

Figure 5.5: Role of Activities in Composed Applications

of one single audio playback application. For reasons of clarity, the presented example is simplified. The original application is the native music playback application called *MP3-Player* for the Samsung Galaxy Note 2[1]. The application consists of two Activities, one service and a broadcast receiver. One Activity, which could be the user entry point for this simplified example, displays a list of all available songs. By clicking on one of the songs in the list, the second Activity is opened. The second Activity presents details about the selected song and may also display lyrics during playback and further information, like release date of the song, name of the album and so on. In each of the two Activities the user has the possibility to control music playback, e.g., by starting or stopping it. If one of the buttons for playback control is clicked by the user, the corresponding command is forwarded to a single service in the background. The service handles playing songs, stopping, forwarding and so on. But since the service has no user interface, it cannot immediately receive commands from the user. All user commands to the service are done through one of the Activities. The role of the service usually remains invisible to the user. If one of the two Activities are shut down, change their states or the user switches between them, the service remains unaffected. If the user is about to receive an incoming call on his smartphone, it would be appropriate to stop music playback and play back a ring tone. This behavior can be realized by a corresponding broadcast receiver. The receiver can be configured to react in a defined manner on certain system notifications, e.g., other system actions which intend to concurrently access the audio playback module.

[1]See `www.samsung.com/galaxynoteII`.

Figure 5.6: Status Bar Music Player Notifications

In case the broadcast receiver receives such a message, he can notify the service, which again might stop music playback. Independent of the Activities, the service then can change the status bar icon from playing (see Fig. 5.6 top) to paused (see Fig. 5.6 bottom).

The different components, including Activities, are units in the sense of life cycle testing (see Sec. 4.3.1). Each Activity has its own life cycle, managed by the Activity Manager (see Sec. 5.1.1) [7]. During reverse engineering the Android 2.2 Activity life cycle, we identified the following callback methods, called in consequence of life cycle state changes [2, 55, 65]:

- *onCreate()*
- *onStart()*
- *onResume()*
- *onPause()*
- *onStop()*
- *onRestart()*
- *onDestroy()*
- *onSaveInstanceState()*
- *onRestoreInstanceState()*

The reverse engineered Activity life cycle is presented in Fig. 3.5. According to this life cycle, an Activity might be in one of four different states [55, 106, 112, 113]:

- *Running:* An Activity in this state is usually visible to the user and holds the focus. It is currently being executed and has access to all required resources, as CPU, RAM and so on. But an Activity might also be running if it is not currently visible to the user. For instance, if the status bar is pulled down or the screen is locked, but the display is active (see Sec. 4.1) [55].
- *Paused:* A paused Activity has lost the user focus but might still be visible to the user. Figure 5.7 presents an Activity which is partly overlapped by another Activity [55]. The Activity in the background is in the state *paused*, while the overlapping Activity, entitled *Subactivity*, is in the state *running*. The state *paused* is often entered by Activities for a short period of time, e.g, when changing state from *running* to *stopped* (see Sec. 3.4.1).

Figure 5.7: Example of Overlapping Activities

- *Stopped:* A stopped Activity is not visible to the user. It also does not have access to the CPU in the sense that the main Activity thread, which also updates its user interface, is not being executed. An Activity is transferred to the state *stopped*, e.g., if another Activity has been started and fills the whole screen. Stopped Activities might still hold data in RAM.
- *Shut down:* An Activity in the state *shut down* does not hold any application data in RAM and is not visible. This applies, e.g., to Activities which have not yet been started or already killed.

In the following, we apply the conceptual approach of testing application life cycles to the Android 2.2 Activity.

5.2 Testing the Android Activity Life Cycle

This section presents an application of the testing concept to the Android Activity. For reasons of clarity, the structure of this section is aligned to the structure of the conceptual approach presented in Sec. 4.4.

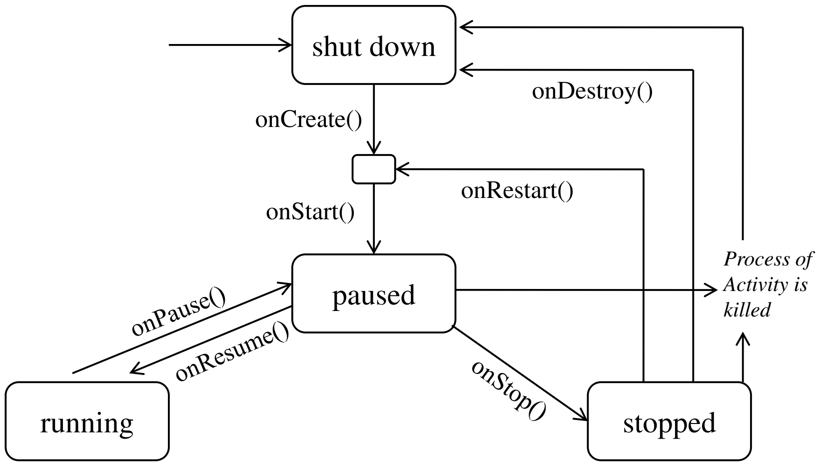

Figure 5.8: Reverse Engineered Android 2.2 Activity Life Cycle

5.2.1 When to Test

Following our approach, the first step is to identify the callback methods which are called as a consequence of life cycle state changes (see Sec. 4.4.1). On Android, we used the following sources to find corresponding callback methods:

- The official Android guidelines [7, 19].
- The official life cycle model representation [2].
- The documentation of the source code [17].
- The Android source code [13].

As explained in Sec. 3.1, we experienced some inconsistencies with the official documents. Thus, we also used the outcome of reverse engineering the Android 2.2 Activity life cycle (see Chap. 3) as well as the corresponding model (see Fig. 3.5). Due to the importance of the model in this case study and to facilitate inspection, we present the model representation once again in Fig. 5.8 [65].

We identified the following life cycle callback methods for the Android 2.2 Activity [55, 65]:

- *onCreate()*
- *onStart()*
- *onResume()*
- *onPause()*
- *onStop()*
- *onRestart()*

- *onDestroy()*

We also identified two further methods, which are called during life cycle state changes [55, 65]: *onSaveInstanceState()* and *onRestoreInstanceState()* (see Sec. 5.1.2). But these methods are only called under certain conditions, e.g., at state changes due to locking the screen or low battery [10, 38]. They provide a convenient way to the developers to store the states of user interface view elements [38, 43]. If they are called, this happens in addition to the life cycle callback methods listed above [55, 65]. For instance, *onSaveInstanceState()* is called before *onPause()* and *onRestoreInstanceState()* after *onStart()* [55, 106]. Due to their redundancy regarding the application life cycle and due to the fact that they do not lead to new states [55, 65], we do not add these methods to the Android 2.2 Activity life cycle callback methods. *onCreate()* is called when an Activity is started from the state *shut down* (see Fig. 5.8). This is always followed by a call of *onStart()*, after which the Activity is in the state *paused*. From *paused* the Activity can move to the states *running*, *stopped* and *shut down*. When moving to *running*, *onResume()* is called. From *running* the Activity always moves back to *paused* [65]. An Activity in the state *running* cannot be stopped or shut down without at least a call to *onPause()* (see Fig. 5.8) [55]. If the Activity changes its state from *paused* to *stopped*, *onStop()* is called. If it changes its state from *paused* to *shut down*, e.g., by being terminated by the system due to low battery, no callback method is being invoked. From the state *stopped* an Activity might change to the states *shut down* or *paused*. The state change to *shut down* might happen in two different ways. If the Activity is killed by the system, it changes its state to *shut down* and no callback method is executed. The developer has no chance to react on this state change [65]. If the Activity is terminated properly by the system, the callback method *onDestroy()* is called. In this case the developer can use the callback method to react to the shut down of the Activity. When switching states from *stopped* to *paused*, first *onRestart()* is called followed by *onStart()*. *onStart()* is called, when the Activity changes its state either from *shut down* or *stopped* to *paused*.

Since all Activity instances on Android inherit from the Activity class, the life cycle callback methods can simply be overridden. An Activity which overrides all these life cycle callback methods is given in Listing 5.1.

Listing 5.1: Activity Overriding Life Cycle Callback Methods

```java
public class NewActivity extends Activity {

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

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

```java
    // ...
  }

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

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

  @Override
  protected void onStop() {
    super.onStop();
    // ...
  }

  @Override
  protected void onRestart() {
    super.onRestart();
    // ...
  }

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

5.2.2 How to Test

According to Sec. 5.1.2, Android Activities are units in the sense of our unit-based testing approach (see Sec. 4.3.1). Each Activity has its own life cycle, which is managed by the system (see Sec. 5.1.2). Since in general there is no correct or incorrect behavior of applications during life cycle state changes, as this depends on the specification of the application, we decided to use assertions (see Sec. 4.3.2). Assertions can be defined in one life cycle callback method and checked in another life cycle method, later in time. According to Sec. 4.4.2, assertions have to be stored persistently, as an AUT might be shut down during test execution and its data (also the assertions) removed from volatile memory (e.g. RAM). To store assertion data persistently on Android, we can make use of the different storage options for Android applications [69]. There are different options available [23]:

- *Shared preferences:* This memory option can be used to store persistently primitive data (ints, longs, strings, ...) in key-value pairs.
- *Internal and external storage:* Data can be stored on internal and external storage devices (e.g. flash card).
- *SQLite Databases:* Android supports SQLite databases.

All these storage options are usually available to the AUT. Depending on the type of the data to store, corresponding storage options can be chosen. For instance, if an image shall be stored for later comparison, the internal/external storage can be used. For storing few single strings, shared preferences might be sufficient.

One design decision, which we made very early during this case study on Android, is to encapsulate the functionality of testing application life cycle-related properties of Activities into a library and provide an appropriate application programming interface (API). There are multiple reasons for this decision [69]. Libraries are a common way to share functionality and can usually be integrated with low effort into projects. They often provide dedicated functionality, as for our purpose of testing life cycle-related properties. If the API is designed and implemented well, the libraries are easy to understand and use. By encapsulating a specific functionality, libraries stay modular and reusable. The code, encapsulated in the library, remains in the library and is not injected into the AUT, where it otherwise had to be removed after testing. At the time of this decision, we also started planning the AndroLIFT tool, which is presented in Sec. 5.3. The library is also used as a core component of the AndroLIFT tool. One important goal when designing such a library, was to keep a high degree of independence regarding the Android versions and the ADT versions. This is necessary to avoid problems resulting from the short release cycles of the Android platform and ADT (see Chap. 5).

The core functionality of the library is to create, manage and check assertions. The effort for a developer to define and check assertions shall be minimized by the library. In a first step, an *AndroLIFT* object has to be instantiated in the AUT. This object is the link to the library and takes care of the test management for this AUT. The initialization of this object is presented in Listing 5.2.

Listing 5.2: Instantiating a Link to the Library

```
public class AUT extends Activity {
  // ...

  // link to the library
  AndroLIFT androLift;

```

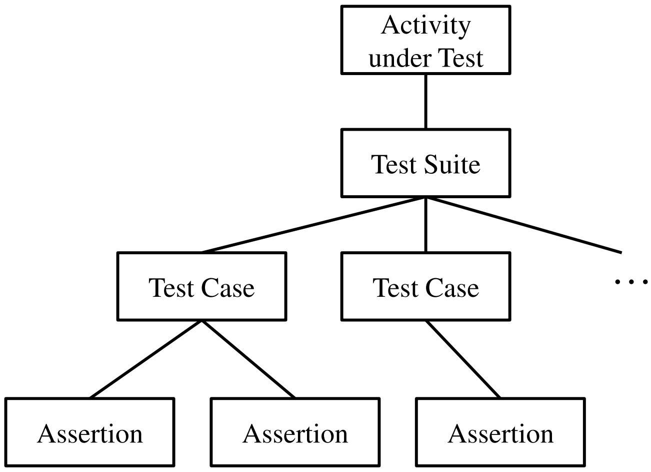

Figure 5.9: Relations between Activities, Test Suites, Test Cases and Assertions

```
7     @Override
8     public void onCreate(...) {
9       // ...
10
11      // initiate the library
12      androLift = new AndroLIFT(this);
13
14      // set a test suite to be executed during runtime
15      androLift.setTestSuite(new TestSuite1(this));
16    }
17
18    // ...
19  }
```

During initialization, the developer passes a reference to the Activity under test to the library (see Line 12 in Listing 5.2). This is required by the library to link the assertions to this Activity and, e.g., to inform the developer about the results of checking the assertions. Additionally, the developer has to specify the test suite containing test cases to be executed during runtime. Every Activity under test can reference one test suite. Each test suite can contain multiple test cases. Each test case can contain multiple assertions. To keep test cases small and results traceable and reproducible, usually each test case has only a small amount of assertions. The relations are sketched in Fig. 5.9. This design is aligned to the well-established JUnit testing [131].

Next, we have to make sure that the library managing the assertions gets notified in case of state changes. This information is required, e.g., if an assertion shall be checked at a specific state change. To keep the influence of the library transparent to the user, he has to actively notify the library at each state change. On one side, this means additional effort to the developer, but on the other side he remains aware of the code injected to the AUT and this approach favors the independence of the library regarding the Android version. The additional effort for the developer consists of adding one line of code at the end of each life cycle callback method. The line of code, which has to be placed at the very end of each method (see Sec. 4.4.2), is a call to a method within the library. The method in the library has the same name as the corresponding life cycle callback method of the AUT. When the library gets notified by calls to these methods, it can initiate managing actions of the assertions, e.g., checking assertions scheduled for the current life cycle state change. Listing 5.3 presents all life cycle callback methods with the injected calls to the library.

Listing 5.3: Notifying the Library of State Changes

```
public class AUT extends Activity {
  // ...

  // link to the library
  AndroLIFT androLift;

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

    // initiate the library
    androLift = new AndroLIFT(this);

    // set a test suite to be executed during runtime
    androLift.setTestSuite(new TestSuite1(this));

    // notify library
    androLift.onCreate();
  }

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

    // notify library
    androLift.onStart();
  }

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

```

    // notify library
    androLift.onResume();
  }

  @Override
  protected void onPause() {
    super.onPause();
    // ...

    // notify library
    androLift.onPause();
  }

  @Override
  protected void onStop() {
    super.onStop();
    // ...

    // notify library
    androLift.onStop();
  }

  @Override
  protected void onRestart() {
    super.onRestart();
    // ...

    // notify library
    androLift.onRestart();
  }

  @Override
  protected void onDestroy() {
    super.onDestroy();
    // ...

    // notify library
    androLift.onDestroy();
  }

  // ...
}
```

A test suite, as defined in the *onCreate()*-method, can contain multiple test cases. An example is given in Listing 5.4.

Listing 5.4: Example of a Test Suite

```
public class TestSuite1 extends TestSuite<AUT>{

  public TestSuite1 (AUT activity) {

    super(activity);
    addTestCase(new TestCaseA());
    addTestCase(new TestCaseB());
    addTestCase(new TestCaseC());
    // ...

  }

}
```

A test suite class extends the *TestSuite* superclass of the AndroLIFT library, typed with the activity under test (AUT). This way, the test suite has access to all assertion functionality of the superclass and Activity-specific values and methods of the given Activity type. The same holds for test case classes (see Listing 5.5), which extend the AndroLIFT class *TestCase*. The developer can define assertions in the corresponding methods in the test case class. If his intention is to define an assertion in the *onPause()* life cycle method, he can override the corresponding method in the test case class and define the assertion. Assertions are defined using the factory pattern [73]. The developer does not define any assertion objects by hand, but passes the corresponding arguments to create an assertion to a factory. The method *createDataAssertion(...)* in Listing 5.5 calls the factory to create a data assertion. Next, the developer needs to pass the required parameters. In Listing 5.5 the developer defines an assertion to check the content of an Android *EditText* element, which is a text input element. The first parameter contains a custom description of the assertion, which is also printed in the test results. It may contain debug information for the developer. The second argument specifies the callback method at which this assertion has to be checked. In this example, the assertion will be checked in the method *onResume()*, identified as a global variable *ON_RESUME* in the AndroLIFT library. The third argument specifies the type of data assertion. In Listing 5.5 the global variable *DATA_EDITTEXT_STRING* defines the assertion as a string comparison in an *EditText* graphical input element. The different global variables are available through the AndroLIFT library and documented in the AndroLIFT documentation. The fourth parameter is a reference to the object of interest, which is in this case the *EditText* input element, referenced by the global variable *mText* in the Activity under test. The four parameters are sufficient if the view element, in this case *mText*, is created using XML in the Android project. If this is the case, the given reference *mText* to the *EditText*-object can be used by the library to access the object, even when the application has been killed in between. If the view-object has not been defined using XML, but in Java source code, the library needs to use reflections to access the object [41]. For using reflections, the library needs to have the name of the object as a string. Thus, the fifth parameter is necessary, which is the name of the global variable in the Activity under test, holding a reference to the view object. The developer can add further assertions to this method by extending the assertions array. He can also add assertions defined in other methods by overriding the corresponding methods in the test case class.

Listing 5.5: Defining a Test Case with one Assertion

```
public class TestCaseA extends TestCase<AUT>{
  // ...

  @Override
  public Assertion[] onPause() {

    return new Assertion[]{
      createDataAssertion("Test␣case␣A:␣checking␣if␣...",
                ON_RESUME, DATA_EDITTEXT_STRING,
                getActivity().mText, "mText")
    };

  }

}
```

After defining the assertions, the test case has to be executed. On Android, this is possible with the emulator, provided with the ADT or a real device. Using an Android emulator has the benefit of being able to simulate incoming calls and SMS by a click of a button, via DDMS (see Sec. 2.5) [20]. Our experience is that emulators and simulators often do not react similar to the corresponding real devices [55, 69, 102]. Thus, we recommend to use real devices where possible. While the actions specified in the test case specification are executed, e.g., interrupt the running AUT by an incoming call, the corresponding life cycle callback methods are invoked. Thereby, the assertions are defined, checked and results printed to the developer [69].

The assertion library keeps the defined assertions in lists. For each life cycle callback method the library keeps a list with defined assertions, which have to be checked, e.g., by comparing two values. When the callback method is invoked, the library is notified of the state change by forwarding the call to the library (see Line 19 in Listing 5.3). At that point, the library starts to process the list of assertions attached to this callback method and checks them. The results of checking the assertions are presented to the developer.

Listing 5.6: Logging Data with Logcat

```
import android.util.Log;
// ...
  Log.i("Some␣tag", "And␣a␣message.");
```

Level	Time	PID	TID	Application	Tag	Text
I	02-22 12:44:55.952	280	280	com.example.test	Some tag	And a message.

Figure 5.10: Logcat Output Example

To present the results to the developer, the library uses *Logcat*. Logcat is a logging tool and comes with ADT [21]. It does not require any further packages. This goes along with our approach to keep the library independent of the Android and ADT versions. An exemplary usage of the Logcat tool can be found in Listing 5.6. In this example the developer writes an information log message (*Log.i(...)*). There are also other types of log messages available, like debug, warning and error [21]. A screenshot of the output with the ADT Logcat view is presented in Fig. 5.10. The developer can see the log data level (here: *I* for *information*), timestamp, process (*PID*) and thread IDs (*TID*), package name of the application, tag and text message. These information are sufficient to notify the developer about results from assertion checks. The library uses the tag entry to identify the Activity and the text message to present the results of checking the assertion, often appended by debugging or test result details.

Further details on the library, its architecture and implementation are given in Sec. 5.3. The library is a core component of the AndroLIFT tool.

5.2.3 What can be Tested

Fig. 5.11 presents a package overview of the *assertion* package of the library. The assertion package of the library contains all available assertions for the Android platform. For reasons of clarity, the package representation is simplified. Some classes and the superior package context have been left out. A more detailed architecture and information about the relations between the classes are given in Sec. 5.3.

The *assertion* package contains multiple classes and three other packages: *data*, *hardware* and *connection*. The classes inside the *assertion* package provide functionality which is important for all assertion classes. They also contain information which all assertions have in common, e.g., log messages in case of a success or failure of the assertion. The class *AssertionFactory* encapsulates the functionality to create new assertions. Creating new assertions is done using the factory pattern [75]. The class *Assertion* holds information and functionality relevant for all assertions. This is the superclass of all other assertions in this package. *CustomAssertion* allows the developer to create customized assertions, e.g., if the required assertion is not part of the available assertions. This might for instance be the case if the developer uses a self-created graphical user interface (GUI) element or has connected a specific hardware module. By extending *CustomAssertion*, the library provides the developer a basis to define customized assertions, which are also automatically managed, checked and the results printed by the library.

The *data* package contains assertions related to data persistence. This covers also the content of text input elements, text labels and so on. One user scenario could be to check the content of an *EditText* element, which is the base class for text input user

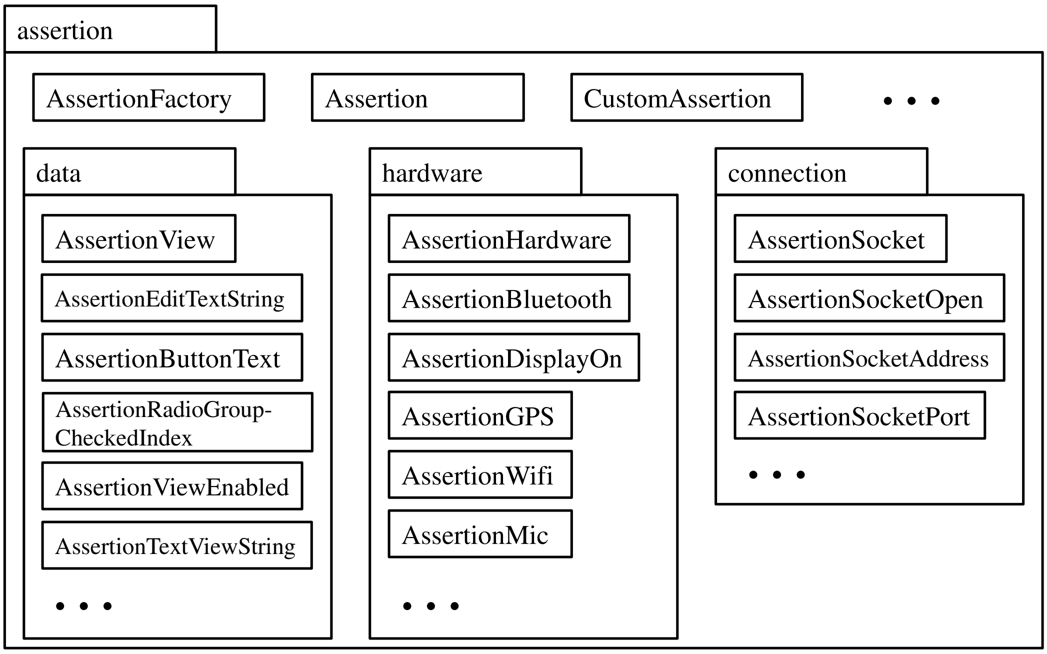

Figure 5.11: Simplified *assertion* Package Overview

Figure 5.12: Radio Group Example

interface elements. The check could be a comparison of a previous content value to the actual content. Another test could check for an empty search input field after starting an Activity managing a contacts list. The class *AssertionRadioGroupCheckedIndex* checks the state of radio groups. Fig. 5.12 presents an example of a radio group on Android. For instance, the developer can check if the states of the radio group buttons have changed or not. The class *AssertionView* extends *Assertion* and is a superclass of, e.g., *AssertionEditTextString*. It bundles the functionality to check assertions related to views, like text input elements and labels. The developer can extend this class, e.g., for checking customized or extended view elements.

The *hardware* package contains assertions for checking hardware-related properties. This might be checking if the Bluetooth or GPS module is on or off, if the Wi-Fi module is connected or if a microphone is switched on. Android also allows to check for the level of display brightness. This can be relevant, e.g., for a torch light application.

AssertionHardware provides all necessary functionality to allow the developer to create custom hardware assertions which are fully managed by the library.

The *connection* package combines all functionality related to connection assertions. For instance, the *AssertionSocketOpen* class can be used to check if a socket is opened or not. This is a prerequisite for an active socket connection. Since Bluetooth, Wi-Fi, UMTS and other connections are established on Android using socket connections, this class already covers a large variety of possible connections. *AssertionSocket*, which is the superclass of *AssertionSocketOpen*, can serve as a basis for other assertion classes related to socket connections.

The assertions in the *assertion* package do not cover all possible assertions for Android. As explained in Sec. 4.3.2, assertions depend on the corresponding test case. For instance, a developer might want to check if a value which is present in a text input element before stopping the application, is available in the database after resuming the application. Therefore, a corresponding assertion might be of interest. This case is not immediately covered by the assertions in the *assertion* package. We designed the library to be flexible and extensible. With the given library architecture the developer is able to implement the required assertions with low effort, e.g., by extending the existing corresponding classes or extending *CustomAssertion*. By modifying the Android source code or rooting Android devices, even more life cycle-related properties can be checked as more hardware modules and software components may become accessible [102]. Further details on the architecture of the library and its classes are available in Sec. 5.3.

5.2.4 Testing Google's Notepad Application

In this section, we apply the implementation of the concept for testing life cycle-related properties on the Android platform to Google's notepad application. The notepad application is one of many tutorial applications that Google provides on the Android webpages [15]. This application was already available for early Android versions and is still being adjusted and refined to cover recent Android implementation and design concepts, too. The application presents the handling of interface elements, user interactions and usage of databases [15]. Since it is available on the official Android tutorial pages, we assume that it is widespread in the Android community and often used for learning Android. Since it is open source, we can apply our approach to test life cycle-related properties of the application. In this case study, we use the notepad version released with Android 2.2.

The application consists of three Activities (see Fig. 5.13). One Activity presents all available notes in a list (see Fig. 5.13 left). If the user clicks on one of the listed notes, the corresponding note is presented in a second Activity (see Fig. 5.13 center). In this Activity, the content of the note can be edited. By using the menu button in this second Activity, the user can choose to edit the title of the currently opened note. Choosing this option leads the user to a third Activity, which allows to edit the title of the note (see Fig. 5.13 right). By pressing the button labeled *OK*, the user can save the changed title and resume to the opened note. For reasons of clarity, we name the three Activities as follows:

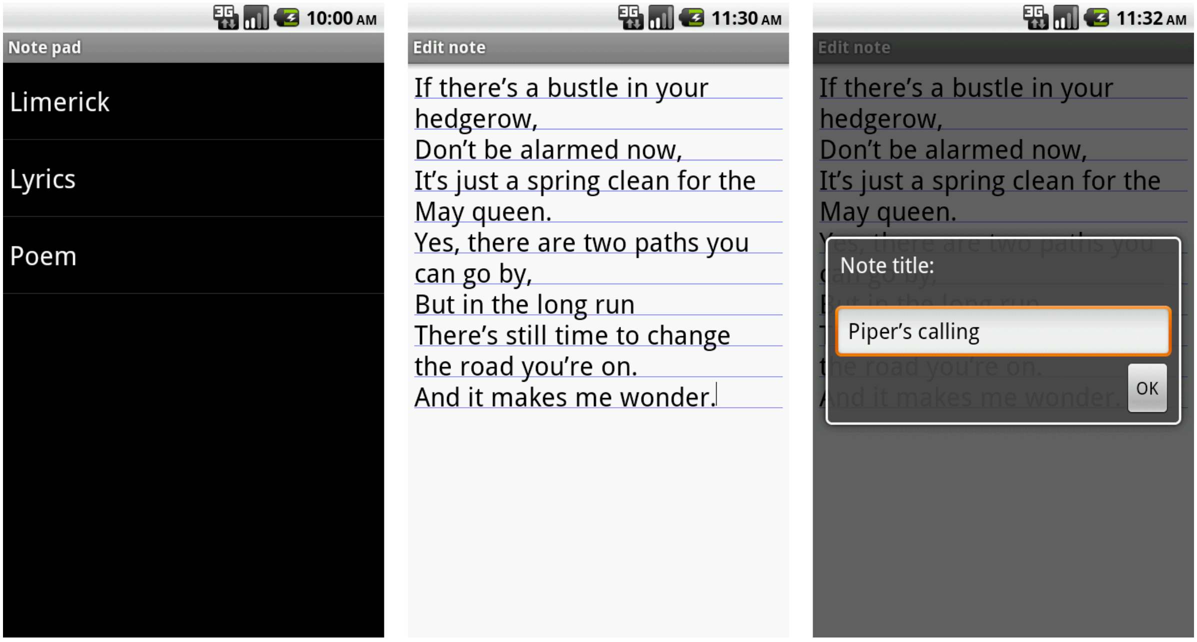

Figure 5.13: Google's Notepad Application

1. *list Activity*: Activity presenting the list with all notes
2. *note Activity*: Activity presenting the content of a single note in an editor
3. *title Activity*: Activity presenting the title of the note and allowing to edit it

Since we do not have the official specification of Google's notepad application, we specify the following requirement which describes life cycle-related properties and which we want to check in the following [69]:

> User input to the note Activity shall not get lost, neither by user interaction like pushing buttons or switching to other Activities, nor by external events like incoming calls or low battery.

This requirement shall ensure data persistence of the note Activity with respect to user input. From this requirement and the possible user actions in the note Activity, we derive the following three test cases [69]:

1. The user opens a note, edits it and then presses the *Home*-button to close the application. After restarting the application, the entered text shall still be available.
2. The user opens a note, edits it, opens the menu and clicks on the button *Edit title* to edit the title of the note. After resuming the note Activity, the entered text shall still be available.
3. The user opens a note, edits it and then receives a call. After declining the call and resuming the note Activity, the entered text shall still be available.

a)
b)
c)

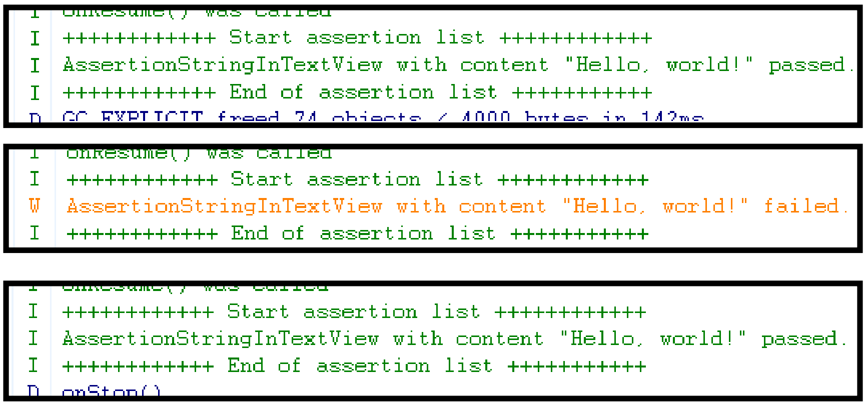

Figure 5.14: Test Results as Log Output

From [65], we know which life cycle callback methods are invoked during the actions described in the test cases [69]: Test case 1 corresponds to the 11th scenario in [65], test case 2 to the 23rd, where the small alarm clock dialog is similar to the edit title dialog, and test case 3 to the 4th scenario. For this reason, we focus during the three test cases on the transitions between the states *running* and *paused*. We define *AssertionEditTextString*-Assertions in the *onPause()* callback method and check them in the *onResume()* method.

All three test cases are executed on a real device, connected during test execution to a computer to print test results. The device is a Motorola Milestone 2 with Android 2.2. The setup before each test execution is a new and clean installation of the notepad application [69]. The only active applications on the device, next to notepad, are the standard Android applications and services which are started automatically at device boot. For each test case, the application is started using the application icon on the home screen.

The test results are presented as Logcat output in Fig. 5.14 [69]. The results presented in a) show that the assertion from the first test case was confirmed. If the author stops the note Activity with unsaved changes by pressing the *Home*-button, the unsaved text is still available after resume. The test results of the second test case show that the assertion could not be confirmed. This test results expose the following behavior of the note Activity [69]: The user opens an existing note with the text content, e.g., *Hello* (see Fig. 5.15 left). He edits the note by extending the note content to *Hello, world!* (see Fig. 5.15 middle). As given by the test case, he does not actively save the note but chooses from the menu to edit its title (see Fig. 5.15 right). Then he stores the title changes by pressing the *OK*-button, which brings the user back to the note Activity. When he resumes the note Activity, its content, which he changed before entering the title Activity from *Hello* to *Hello, world!*, has changed to *Hello* again (see Fig. 5.15 left). This behavior does not correspond to our expectations, since the unsaved modifications are lost. This is not necessarily a bug, since we do not have the official specification of Google's Notepad application and thus do not know if this behavior is intended or not. But obviously data is lost, which decreases the quality of data persistence, user experience and usability [69]. The assertion in the third test case was confirmed, too (see Fig. 5.14 c)). If the user is

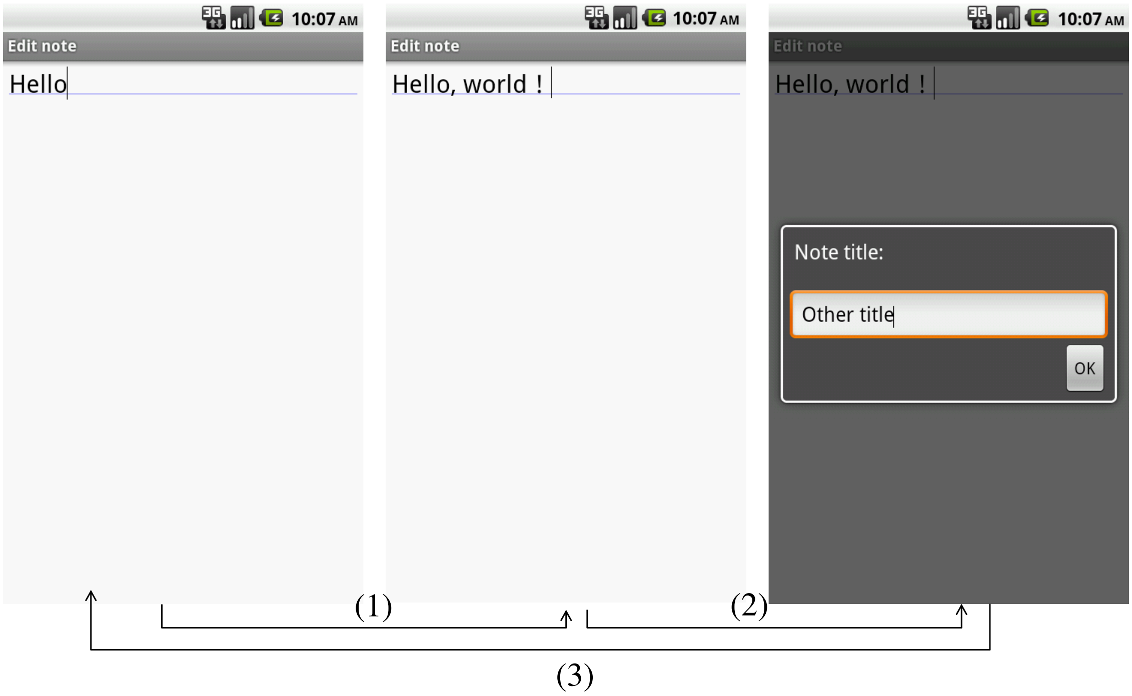

Figure 5.15: Note Activity of Google's Notepad Application

interrupted by an incoming call while composing a note, the unsaved changes are still available after resuming.

5.3 AndroLIFT

One of the results of the case study on Android is a tool for testing life cycle-related properties of Android Activities. The name of the tool is *AndroLIFT* and its core consists of the AndroLIFT library. The functionality and limitations of the library are presented in Sec. 6.3. Section 5.3.1 gives details about the library. The focus remains on design-level. The AndroLIFT plug-in, which is presented in Sec. 5.3.2, integrates the functionality of the library into the Eclipse IDE. The plug-in also provides extended functionality which facilitates testing of life cycle-related properties to the developer.

5.3.1 Library

The AndroLIFT library provides the core functionality to test life cycle-related properties of Android Activities. The main tasks of the library are:

- Hold an extensible catalog of assertions.
- Provide an easy way to define assertions.
- Manage assertions during runtime.

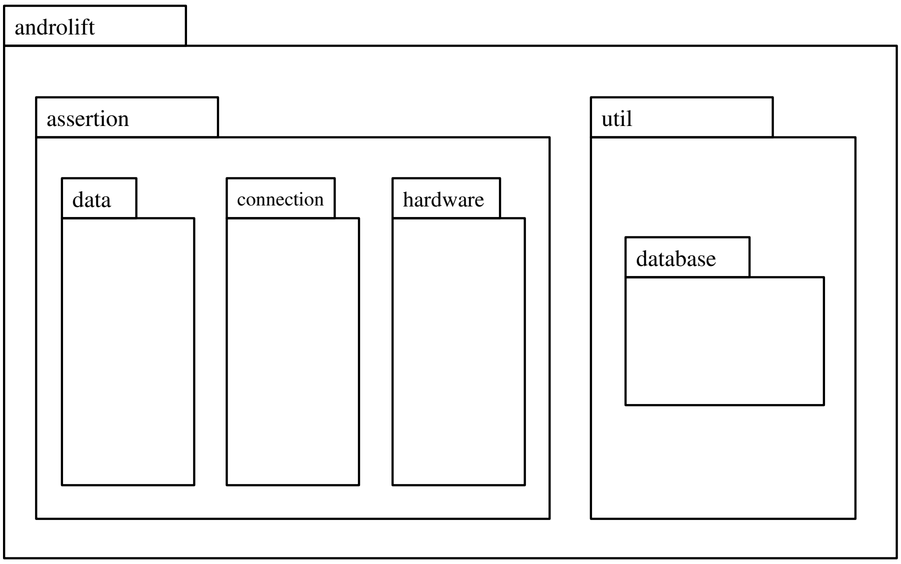

Figure 5.16: AndroLIFT Library Package Overview

- Take care of assertion persistence over the life time of the application.
- Check the assertions as specified.
- Report results and errors.

This functionality is splitted into different packages and provided to the developer through a corresponding API. Figure 5.16 presents an overview of the different packages. The *androlift* package holds the functionality on topmost level. This includes, e.g., the *AndroLIFT* class, which allows to subscribe new assertions and notify AndroLIFT of state changes (see Sec. 4.4.2). *androlift* also provides functionality to create test cases and bind them to test suites. This structure is aligned to the well known and widespread JUnit testing [135]. The package *assertion* holds the functionality and base classes for all assertions. The different requirements and implementations of assertion classes are bundled in the packages *data*, *connection* and *hardware* (see Fig. 5.11). The *util* package holds functionality which does not immediately contribute to testing life cycle-related properties, but is necessary for the functioning of the whole system. This includes different reflection steps which are necessary, e.g., to reconstruct links to objects from given IDs after the AUT was shut down and restarted. The *database* package holds functionality to persistently store and retrieve assertion data over the life cycle of an Activity.

Figure 5.17 presents a class diagram containing some important classes of the library. Each *AndroLIFT* instance takes care of one Activity. It manages the assertions of this Activity, takes care of code injections and uses the Activity-specific sandbox for persistent storage of the Activity's assertions. Code injections also contain, e.g., notifying the AndroLIFT instance of state changes of its Activity under test. The automatic code injection is a feature of the AndroLIFT Eclipse plug-in. The AndroLIFT instance also allows to define assertions. These are assigned to test cases, which are part of a test suite.

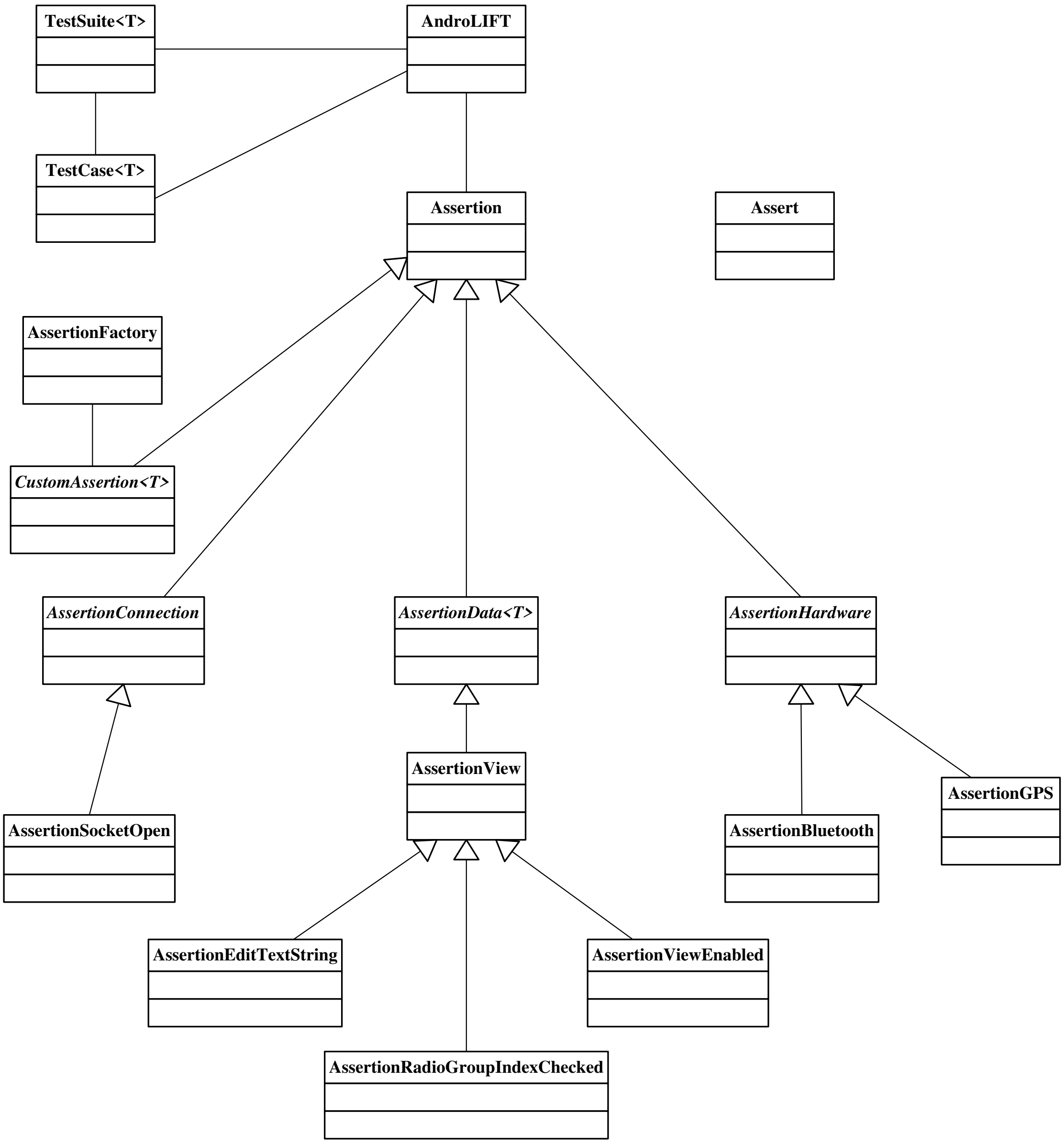

Figure 5.17: AndroLIFT Library Class Overview

One test suite might consist of multiple test cases, which again might contain several assertions (see Sec. 5.2.2).

The abstract class *Assertion* provides functionality required by all assertion types. This includes, e.g., the name of the assertion, the callback method in which the assertion shall be checked as well as abstract method definitions for checking and defining the assertion. The abstract methods are overridden in the concrete assertion classes with assertion-specific actions. *AssertionConnection*, *AssertionData*, *AssertionHardware* and *CustomAssertion* provide more concrete functionality required by the different assertion types. For instance, *AssertionData* provides methods to reconstruct links to view element data after shutting down or killing and restarting the application. One level below, the concrete classes, as *AssertionEditTextString* and *AssertionGPS* can be used to check concrete life cycle-related properties. The class *AssertionEditTextString* checks the content of an *EditText* view element. *AssertionGPS* can check whether the GPS module is on or off. Details about each concrete assertion type can be found in the corresponding documentation files for the classes. With the abstract class *CustomAssertion*, we provide the developer a basis for implementing own assertion classes which do not require any of the functionality provided by *AssertionConnection*, *AssertionData* and *AssertionHardware*. For instance, a developer can use this class to define application- or device-specific assertions, e.g., if a dedicated hardware module is connected to the Android device. The library is an important part of the AndroLIFT Eclipse plug-in.

5.3.2 Eclipse Plug-in

The AndroLIFT Eclipse plug-in provides the functionality of the AndroLIFT library plus additional functionality to the developer. This is accessible to the user through two different Eclipse views: Life cycle view and AndroLIFT view.

The life cycle view provides three features. It presents a view on the life cycle model of an Android 2.2 Activity (see Fig. 5.18). The model presents all states of an Activity and the state transitions, based on the outcome of reverse engineering the Android 2.2 Activity (see Chap. 3). The transitions are labeled with names of the corresponding callback methods or *kill*, in case a transition might be taken without invocation of any callback method (see Sec. 3.3). The developer can immediately see which states are available and which life cycle callback methods are called during state changes. He also can see in which states the Activity might be killed without invocation of any callback methods.

Another feature of the life cycle view is to highlight the current state of an Activity and the state transitions in case of state changes. Therefore, the developer can choose one Activity of which the life cycle view presents the state and state changes during runtime. The current state and invoked transitions are marked by a thick dotted line. In Fig. 5.18 the Activity is currently in the state *shut down*. Since state changes of an Activity might happen quickly, each action is presented to the developer in the model with a duration of 1000ms. For instance, if an application changes its state during startup from *shut down* to *running*, first the state *shut down* is marked for 1000ms, next the transition between *shut down* and *paused* for 1000ms, then the state *paused* for 1000ms and so on. This

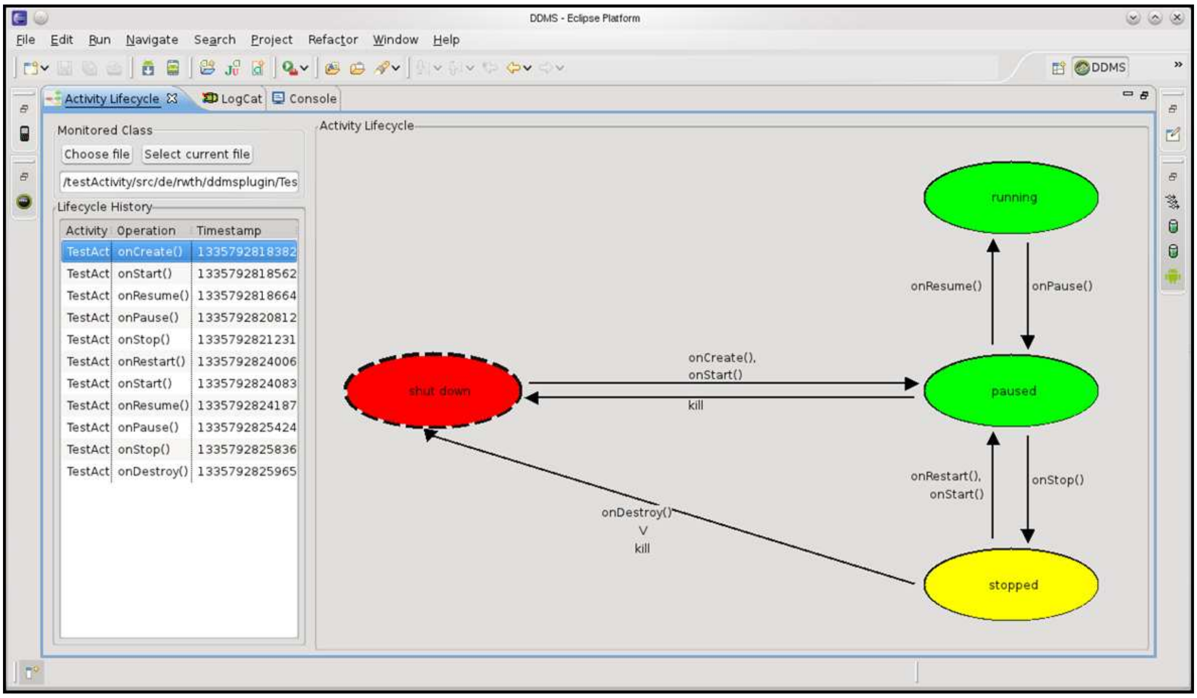

Figure 5.18: Life Cycle Model Representation of the AndroLIFT Life Cycle View

helps the developer to comprehend and learn the life cycle behavior of his application. He also learns about the state changes a certain trigger, e.g. an incoming call, leads to. This helps to implement requirements on the life cycle and to debug unexpected life cycle behavior. Additionally, all state changes are logged in the *Life Cycle History* (see Fig. 5.18 left). The developer can use this log to retrace the life cycle actions and analyze the temporal behavior of the life cycle.

The third feature of the life cycle view helps developers to implement life cycle-related features. Therefore, the developer can click on a method name in the life cycle view to get immediately to the corresponding life cycle callback method implementation in the code (see Fig. 5.19). If the method is not implemented yet, the life cycle view will automatically inject the code for overriding the clicked life cycle method and place the cursor accordingly (see Listing 5.7). The developer can immediately start implementing the life cycle actions.

Listing 5.7: Injected Life Cycle View Code

```
...
// injected code start
@Override
protected void onStart() {
  super.onStart();

  // cursor position
}
```

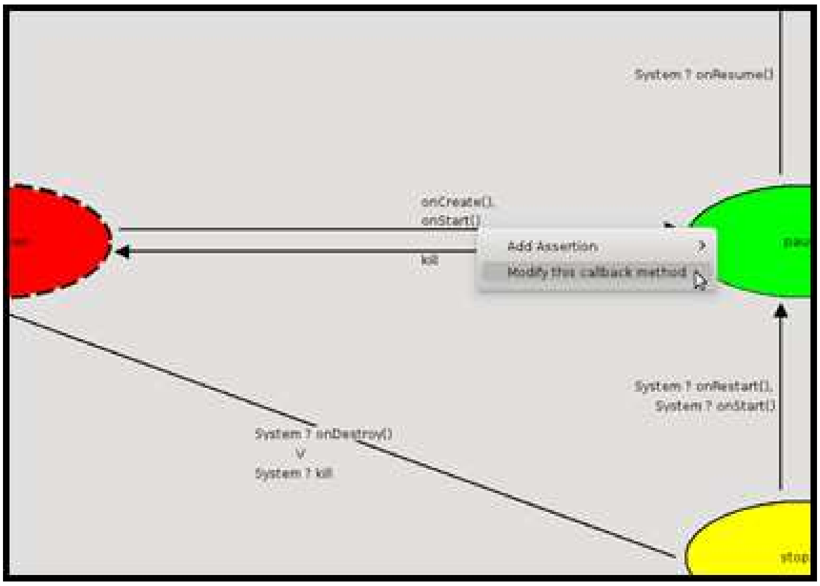

Figure 5.19: Link between Life Cycle View and Callback Method Implementation

```
// injected code end
```

The second view, called AndroLIFT view, provides the developer additional functionality to test life cycle-related properties. It allows developers to define assertions, specify test cases and test suites through a GUI (see Fig. 5.20). All assertions which are available in the AndroLIFT library can be defined using the corresponding GUI. Figure 5.21 presents the GUI while defining a Bluetooth assertion. The benefits of defining assertions through the GUI is that the developer does not require knowledge of the AndroLIFT API. He does neither need to use the AndroLIFT library, nor to work on code level. When assertions are defined through the AndroLIFT GUI, the corresponding steps on code level are done automatically:

- Resolve AndroLIFT dependencies, e.g., import the AndroLIFT library.
- Override all life cycle callback methods.
- Inject code required by AndroLIFT, e.g., notify the AndroLIFT library of state changes (see Sec. 5.2.2).
- Place assertion definitions with all required information, e.g., when to check the assertion.

During and after test execution, the results of the life cycle-related assertions are printed in a JUnit-like view to the user. Figure 5.22 presents the states of assertions during execution of a test case. The assertions that succeeded are marked green and the ones that failed red. The assertions which already have been defined, but not yet checked, are marked yellow. The developer also gets information on the name of each assertion, the expected value and in which callback method the assertion is checked.

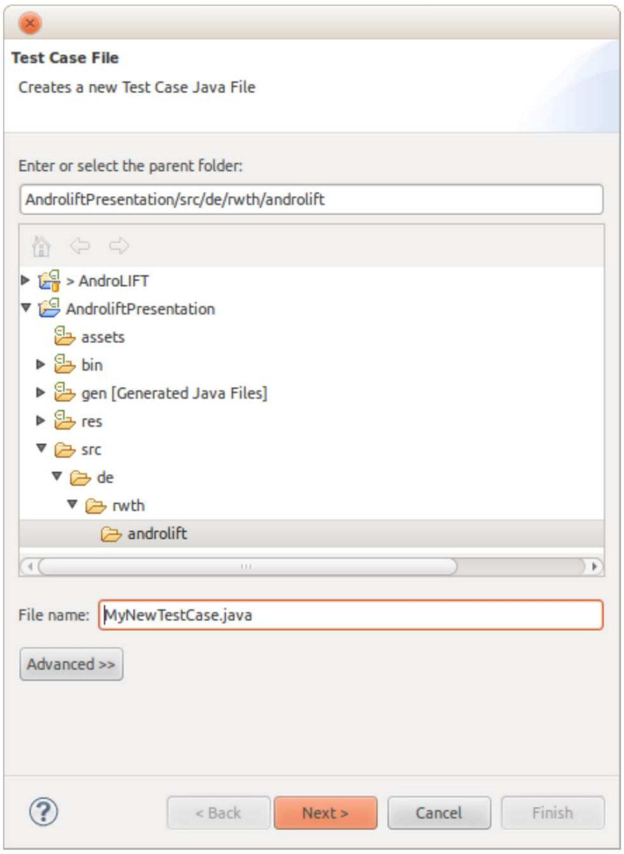

Figure 5.20: AndroLIFT View for defining Assertions and Test Cases

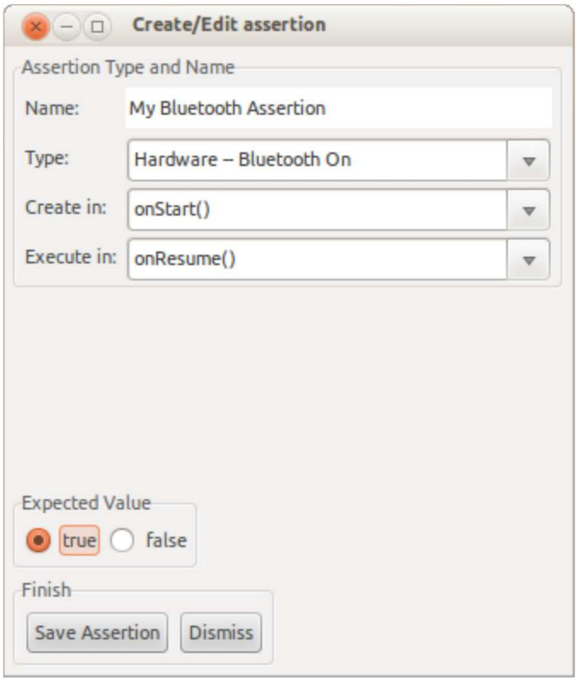

Figure 5.21: Defining a Bluetooth Assertion through the AndroLIFT View

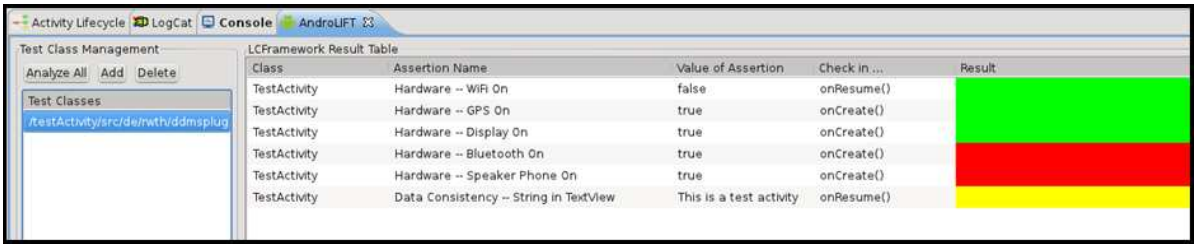

Figure 5.22: AndroLIFT View presenting the States of Assertions

The AndroLIFT Eclipse plug-in is implemented as an extension to the ADT. Thereby it has access to all required modules, like DDMS and Logcat.

The next section evaluates our conceptual approach regarding our experiences on applying it to Android 2.2 Activities and services, iOS 4 applications and services, Windows Phone 7.5 and Java ME MIDP 2.1 applications [55, 84]. Additionally, it presents the results of evaluating the AndroLIFT library in a practical course for students.

6 Evaluation

Next to the case study (see Chap. 5), where the presented approach is evaluated by implementing it for the Android platform, the first two subsections evaluate the concept with regard to capabilities and limitations. The third subsection presents an evaluation of the Android implementation of our concept during a practical course for students.

6.1 Functionality

This section evaluates which functionality is covered by the presented testing concept with respect to testing life cycle-related properties and beyond. Basically, the focus of this concept is on requirements and test cases which include state changes. For instance, a property, setting, state or value of a component can be checked at state changes. The values can also be compared to corresponding values from preceding state changes. Each test case can contain multiple state changes. The AUT can be shut down or killed during test execution. Since the concept insists on storing assertions persistently, it is up to each implementation of the concept to satisfy this requirement.

One important requirement for implementing this concept is the existence of mechanisms that notify the AUT about state changes and allow it to react on each state change. Such reactions might be defining and checking of assertions. On the current mobile platforms, which include Android, iOS and Windows Phone, such notification mechanisms are available. The AUT might be notified by events, to which it can react accordingly, or callback methods like on Android 2.2 (see Chap. 5).

Mainly, everything the AUT has access to can be checked with the presented approach. This includes hardware modules, connection status and memory content like RAM, flash memory and databases. If the concept is implemented to act as part of the AUT, as presented in the case study with the Android Activity, the test components often immediately have access to the same components, hardware modules and memory areas as the AUT [55, 84].

The presented concept is not limited in its functional range to applications from the mobile area. It can also be applied to other areas where state changes play an important role.

6.2 Limitations

While the previous section highlights the chances and applicability of the presented concept, this section hints to its limits. For instance, the presented concept is not designed to check assertions referring to states (instead of state changes). Thus, assertions which, e.g., require user interaction while the application is active, cannot be checked with the presented concept without modifications. For instance, a requirement like *If the user resumes the application and clicks on the Restore-button, he shall see the same text he entered before leaving the application*, cannot be checked. The presented concept focuses on properties which play a role at state changes. For checking requirements which refer to a behavior or function within one single state, other tools can be used, like unit testing, Robotium or Frank [81, 96, 136]. For instance, Robotium can check if a certain output is visible after the user has executed some actions (e.g. entering text to the GUI and clicking buttons).

One requirement of the concept to be applicable is that the AUT gets notified of state changes and is given the chance to react on them. If this should not be the case, the concept is not immediately applicable without any modifications. One solution would be, depending on the architecture and options of the underlying platform, to introduce one further layer between the AUT and the platform. This layer notifies the AUT of state changes. To be able to propagate state change information to the AUT, the additional layer itself requires access to this information on platform level. Another solution could be to modify the AUT so that at startup and shutdown of the application certain code (within the application) is executed, e.g. defining and checking of assertions. This would be raw code injections into the AUT, which have to be traced, whose influence on the application behavior has to be measured and which have to be removed after testing.

There are certain life cycle-related properties which cannot be checked with the presented testing concept. For instance, this holds for test cases going beyond state changes: An assertion which has to be checked 30 seconds after the AUT has resumed, is not covered by the concept. The presented concept uses events and methods at state changes to define and check assertions. It is possible to extend this concept to cover further assertions and test cases, to check and define assertions in arbitrary methods and probably also cover time triggered assertions.

If the concept is implemented as part of the AUT, e.g., as in the case study (see Chap. 5), the test code usually has the same options and access as the AUT. Depending on the test case it is also possible that the AUT requires different permissions during test execution than during normal mode of operation. Due to existing security concepts, as sandboxing (see Sec. 4.4.2), some adjustments might be necessary. This can be the case, e.g., if the developer wants to use the AndroLIFT library (see Sec. 5.3) to test life cycle-related properties of an application which does not have access to the database, since in regular operation mode it does not make use of the database functionality. The presented version of AndroLIFT uses databases to store and manage assertions persistently, thus the AUT had to request corresponding permissions for testing life cycle-related properties. The same might hold for hardware modules and memory areas, depending on the AUT, the underlying platform and the test case.

Depending on the test case it might also be necessary that assertions are defined in one component with an own life cycle and checked in a different component with its own life cycle. Such components might be two Android Activities which are part of one application. A corresponding test case can consist of one component which handles the creation of a new list element (e.g. name, address, ...) and another component which displays the list with all elements. In this test case the user might define an assertion when leaving the component after creating a new list element and check the assertion when opening the list displaying all elements. The assertion could check if the entered name of the list element appears in the list with all elements. The presented concept sees components with an own life cycle as units (see Sec. 4.3.1) and sticks assertions strictly to one unit like with JUnit testing [131]. If assertions pass the border of one single unit, other factors of influence, e.g. the moment of switching to a different unit, might have an impact on the test case. The concept had to be extended to be able to share, manage and hold assertions consistently throughout different units.

Defining and checking assertions costs computing time and memory. This might influence the execution of the AUT. Thus, implementations of this concept have to take care to keep the influence of the test code under control and the tester has to be aware of the impact of the test code. This holds especially if the AUT has strict timing or real-time requirements or operates on scarce memory. Depending on the implementation of our concept, the AUT might require a different amount of time and memory during test execution than during normal mode of operation.

6.3 AndroLIFT

This section evaluates the implementation of our approach for the Android platform (see Chap. 5). The evaluation took place within the context of a practical course at the RWTH Aachen University with seven student participants. Six of the participants were studying for master's degree in computer science and one student for master's degree in Software Systems Engineering. The course introduced the students to the importance of life cycle-related properties in the area of today's mobile devices, the relevance of the properties for the quality of mobile applications and how these properties can be tested using our approach. The students were taught during multiple lectures and by presenting them our approach on testing life cycle-related properties of mobile applications [69]. But the students were not introduced to our implementation of the approach for the Android platform, which is the AndroLIFT library. They did not had access or any details about AndroLIFT, except some general information presented in [69]. During the practical course, all students had to implement Google's Notepad application (see Sec.5.2.4) by following the corresponding tutorial [24]. Google's Notepad application was in the focus of this evaluation. To make the application more interesting for testing life cycle-related properties, we modified the *title Activity* (see Sec. 5.2.4) of the Notepad application by adding more graphical components. The three different Activities of the modified Notepad application are presented in Fig. 6.1. The modified Activity is presented on the right and is in the following referred to as *properties Activity*. It has different view elements with which the user can specify the properties of a note. For instance, he can specify the title, a category (*private* or *work*) and a *due date*. To enable the due date view element, the checkbox above has to be checked first. After the students implemented Google's Notepad application, they also got the source code of the modified Notepad

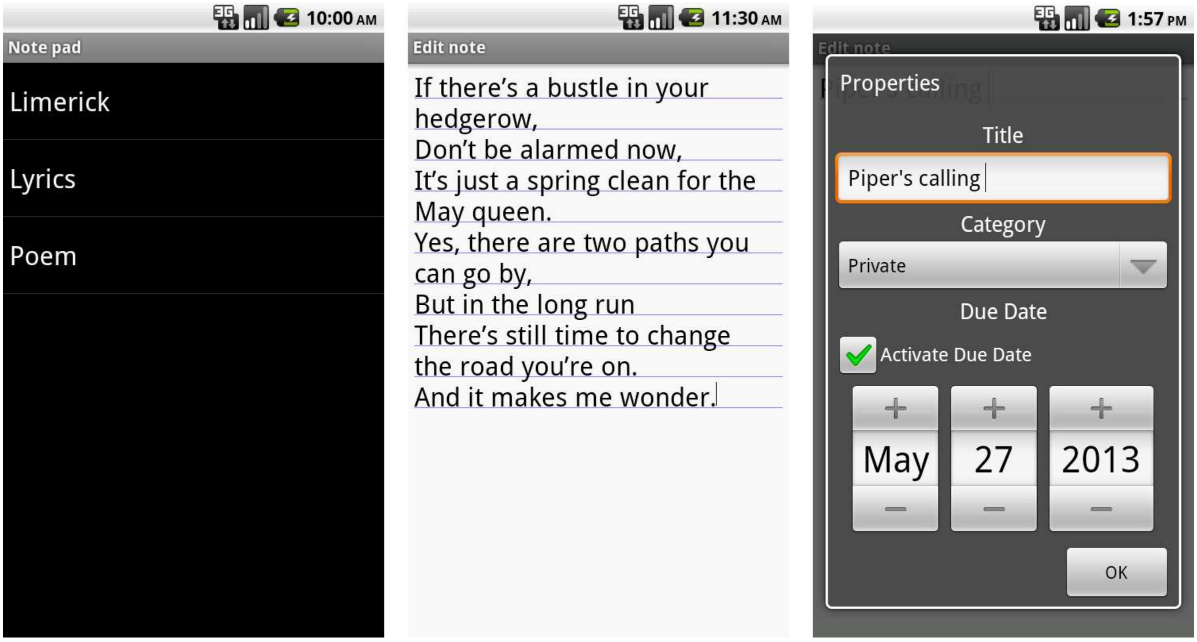

Figure 6.1: Modified Notepad Application

application (with the properties Activity) and studied this, too. Each student had to prepare a presentation about the Notepad application and was asked questions on design and implementation level to make sure each student got a good understanding of the application. In a second step, each student had to derive at least eight different test cases for life cycle-related properties of the modified Notepad application. One test case is sketched in the following [37]:

Test Case ID:
7

Test Object:
Note Activity

Test Environment:
Ubuntu 12.04, Eclipse 4.2, Android SDK 22.0.1, Android 2.2 emulator, Notepad application installed

Test Setup:
Start Notepad application. Open the options menu and click on *Add note*. Enter the text *Text1*. Leave the note using the *Back*-button.

Test Execution:
Open the note entitled *Text1* and append the text *extension* to the content of the note. Press the *Menu*-button and choose *Properties* from the menu. Leave the properties Activity using the *Back*-button.

Test Result (expected):
The content of the open note shall not have changed. It shall still be *Text1 extension*.

In the field *Test Result (expected)* the students were asked to specify the result they expect after test execution. For each two different students, who independently of each other specified the same test case, the expected results were always the same. This indicates that the expectations on the behavior of the modified Notepad application were clear. After merging the test cases of all seven students, the resulting test case catalog contained 15 different test cases. Two for the list Activity (see Fig. 6.1 left), six for the note Activity (see Fig. 6.1 center) and seven for the properties Activity (see Fig. 6.1 right). Each student got the test case catalog and was asked to study and execute each test case on an Android 2.2 emulator, running the modified Notepad application.

At this point each student had

- basic knowledge about life cycle-related properties and how to test them with our approach (on conceptual, not implementation level).
- good knowledge about the modified Notepad application, on design and implementation level.
- good knowledge about all 15 test cases related to life cycle-related properties.

With this level of knowledge, each student was invited to separately participate in an evaluation session. Basically each session consisted of two parts: In the first part, the student was asked to solve a task with any approach and any available tools (no limitations). In the second part, the student was asked to solve the same task with our approach and our AndroLIFT library. The evaluation was mainly based on questionnaires which the students filled out, enriched by observations and rarely interview sessions (e.g. if the intention of a student during a session was not clear to us). Each session, attended by one student participant, one recorder (a student assistant) and one observer (which is the supervisor), was structured as follows (in this sequential order):

1. The student was given a first questionnaire to determine the programming skills of the participant by self-evaluation. The student was asked for overall (programming language-independent), Java-specific and mobile-specific programming skills. He also was asked to give details about the mobile operating systems and devices with which he already got in touch with. The last question aimed on experiences with application life cycles before participating in this practical course.
2. After filling out the questionnaire, the student was shortly introduced to the system: Eclipse 4.2, running on Ubuntu 12.04 with Android SDK 22.0.1, a running Android 2.2 emulator and the familiar, modified Notepad application already loaded as an Eclipse project and ready to execute. The system was executed in a virtual machine running on a Windows desktop host, so we are able to reset the virtual machine to exactly the same starting point for each participant. This system filled one of two 24 inch monitors. On the other monitor, the participant had access to a web browser. In the web browser, the online Android documentation was already loaded, so he could access it immediately. Additionally, the participant had unrestricted access to the Internet during the session. He could also ask the observer questions.

The observer may answer questions which result, e.g., from lacking experience with Android programming or basic Java programming knowledge. Next to the well known, modified Notepad application, each student also got the full test case catalog, containing the 15 test cases, also well known to the student.

After introducing the student, he had two hours to implement the test cases from the catalog in such a way, that after execution of each test case the information, whether the expected results were met or not, were printed automatically by the system (e.g. in log or console). The students were free to choose any approach they like (even ours) and could make use of anything they needed (e.g. additional libraries, JUnit, ...). But they were not given the AndroLIFT library, yet. At this point, the student was in the role of an Android application developer, who knows the application very well, who also knows the test cases very well and now wants to test the 15 life cycle-related properties automatically. To keep the task simple and comprehensive, test execution should still be done manually (e.g. by tapping on the emulator), just the result of each test case should be printed after execution (e.g. *Test Case 1 passed.*). Such an automatic checking and notifying of test case results can help the developer to reuse the test code when testing a more recent release of the same application. If the students have not been finished with all 15 test cases after two hours, they were interrupted.

3. The student was given a second questionnaire to determine a self-estimation of the approach chosen by the students. Some questions were related to the conceptual and other to the implementation level. The idea was to get an indication, which approach application developers would choose to check life cycle-related properties and if the absolutely freely chosen approaches were probably even better than our approach.

4. Although the student was able to eat, drink, visit the restrooms etc. during the first part, he had the option to have a break at this point.

5. The student got the same task as in step 2, but he had to solve it using our library. So he had to execute all 15 test cases and print for each test case if the expected test case results were met or not. Therefore, he first got access to the AndroLIFT *how-to*, which is one wiki page with an introduction to the concept, library and a first code example. Much of the information given in the wiki were also part of our paper about testing life cycle-related properties [69], which the students already got in advance and were allowed to take with them to this session. Next, the student also got access to the documentation of the AndroLIFT library (generated with Javadoc [128]). Once again, the student had two hours to solve this task.

6. A third questionnaire was given to the student determining his experiences with our approach, on conceptual as well as implementation level. Additionally, we asked about the experienced quality of the how-to and documentation of the library.

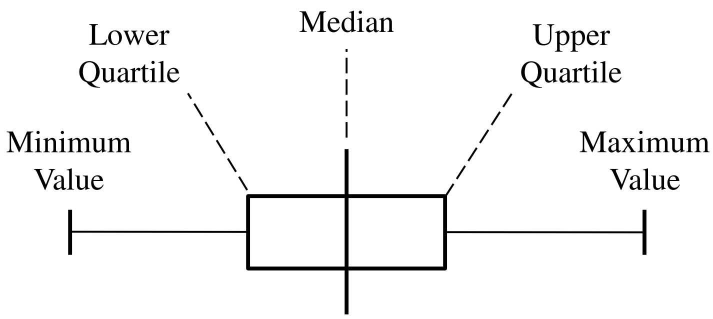

Figure 6.2: Boxplot Semantics

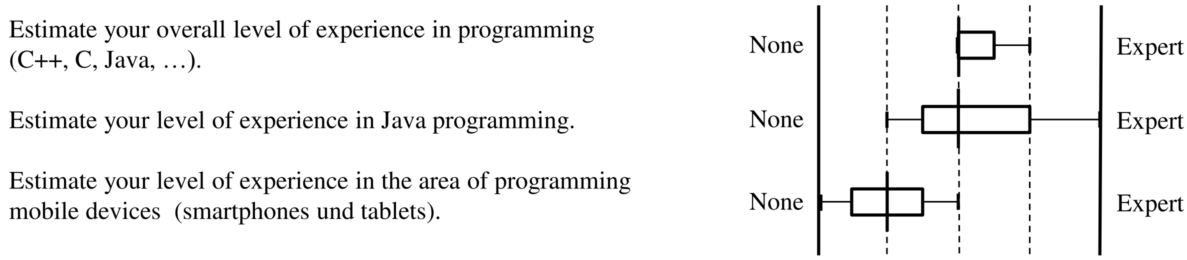

Figure 6.3: Questionnaire about Programming Skills

7. In the fourth questionnaire, we asked the student to compare his approach to our approach regarding different indicators, as efficiency, simplicity, effort and so on.

The recorder kept track of the actions of each student (e.g. the approach the student chose, taking timestamps, ...) and the observer answered specific questions. The observer did not answer questions which immediately would have helped to solve the task (e.g. he did not provide an approach to the student). But he helped with general questions, e.g. if students with only few experience with Android did not know how view adapters or database cursors on Android work. The questions were written down by the recorder.

In the following, we present the results of this evaluation in the same order as given above. The questionnaire results are printed as boxplots with the semantics given in Fig. 6.2. Fig. 6.3 presents the results from the first questionnaire, referring to the programming skills of the participants. Basically, all participants had knowledge in programming. For most participants, Java was their favorite and main programming language. One participant (out of seven) even estimated his skill to expert in Java, which was confirmed by the observer watching the student programming. Only few of the participants have developed applications for smartphones and tablets before attending the practical course. During the practical course, all participants got in touch with Android applications like the Notepad tutorial and the modified Notepad application, which was also the only application of interest during this evaluation session. Before the practical course, some participants already tried out some other Android tutorials, some also have developed applications for Android and one even had an Android application in Google's Play Store. Most students did not consciously come in touch with application life cycles.

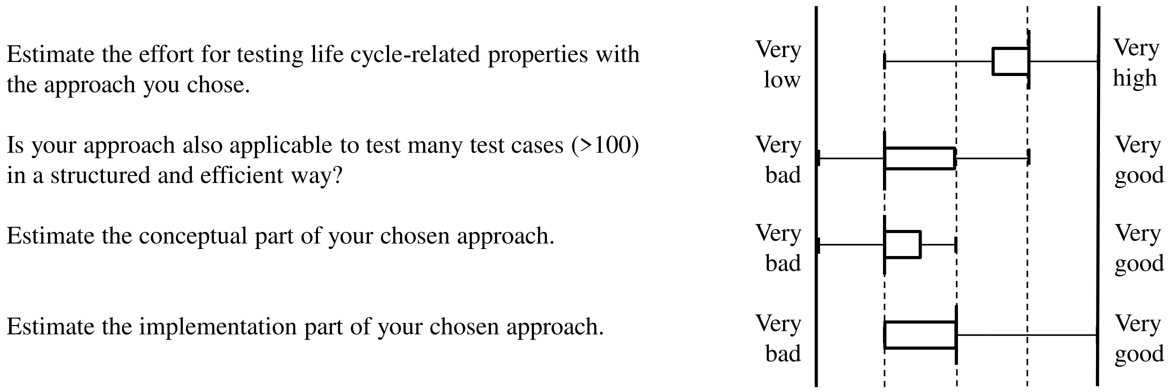

Figure 6.4: Questionnaire about the Student's own Approach

Two participants got in touch with the Android application life cycle during the lecture *Introduction to Embedded Systems*, held at the RWTH Aachen University. The lecture gives a basic understanding of Android's Activity life cyle and how to manage life cycle state changes on application level.

Fig. 6.4 summarizes the answers from the second questionnaire, dealing with an estimation of the approach chosen by the student. In general, the students tried very different ways to solve this task. Some students immediately started implementing one test case after the other. One student tried to solve this task using JUnit. Another student separated the testing code from the AUT code by using different packages and classes. Most approaches involved life cycle callback methods, which might result from the fact that the students did not work with application life cycles before this practical course and we introduced our testing approach (on conceptual level) before the evaluation session. So for many students it seemed clear to go this way, too. Nonetheless, regarding the results in Fig. 6.4, most students do not seem to be happy with their results. The effort for testing life cycle-related properties with their approach seems to be high, their approaches do not seem to scale with an increasing number of test cases (except for the student who separated the test code in separate test packages), the conceptual approach seems weak and the implementation does not seem to be satisfying. The student, who tried to use JUnit for testing life cycle-related test cases, had a good feeling about the scalability of his approach. The student separating test code from AUT code, estimated that it would only take a low effort to create further test cases and that the technical implementation of his approach was very good. In the end, the student trying out the JUnit approach did not manage to get one single test case running in time and even when he was finished, pure JUnit would not be able to deal with any of the test cases, as they refer to running whole Android applications and not static code, e.g. single classes or methods. The approach of separating test code from AUT code hints to our own approach, whereas the student's implementation of this approach was different. The student created for every single test case custom methods in his test code, which are called by the corresponding life cycle callback method in the AUT. So for each test case, the AUT has to be adjusted. Another issue of this approach, which was an overall issue during most sessions, is the fact that data is stored as global or even static variables. The global variables are cleared by the system as soon as the application, e.g., is destroyed during life cycle actions. This happened to a large part of the participants. The static variables are cleared by the system when the application (including the Dalvik Virtual Machine) is shut down, e.g. after being killed by the system. Another major issue during most sessions, was that the results from the different test cases were not being printed separately, but got more and more as the number of implemented test cases increased. It was hard for the students to distinguish the different test results in Android's Logcat. This issue is also related to another common problem: Many students added the test code into the AUT. They used the same objects, references and variables as the AUT and injected their code between code lines of the AUT. It was hard for them to trace the test code and take care of any interferences between test and AUT code. Although the students were allowed to use every approach and method they could think of, not one of the students was able to implement all 15 test cases during the first two hours of the session.

After the second questionnaire, the students got time for a break. Before starting to solve the same task of implementing all 15 test cases, but this time with our library, the students had to read through a how-to about the library. We measured the time it took the students to read and understand the how-to. We also answered questions related to the how-to and wrote down the student's questions and issues for the purpose of improving the quality of our how-to. On the average, it took the students 19.85 minutes to read and understand the whole how-to. The minimal duration was measured 11 minutes and the maximal duration was 37 minutes. The students with short reading times for the how-to were mainly the students who had better programming skills (regarding their answers from the first questionnaire). We did not find significant issues with the how-to. There was one definition of a global variable missing, which was corrected after the session. Very few students rushed so quickly through the how-to that they missed some details, which were required when implementing the test cases afterwards. Since the students had access to the how-to, AndroLIFT library and the Internet throughout the whole second part of the evaluation session, they did look up the details while implementing the test cases afterwards. This behavior seemed to be the usual behavior of the individual student reading through how-tos. Some students even are used to read step by step through a how-to and immediately implement the different steps. We kindly asked the students to first read, understand and ask questions (if any) related to the how-to and do all implementation steps afterwards.

When the students finished reading and understood the how-to, they had to execute the same task as during the first half of the session, but this time using our approach, implemented in the AntroLIFT library. They had to implement all 15 test cases. After that they were asked to fill out a third questionnaire. This questionnaire contained the same questions as the second questionnaire, but referred to our approach, instead of the students approach. It also contained two more questions related to the quality of the AndroLIFT documentation and how-to. The results of the third questionnaire are summarized in Fig. 6.5. Most students experienced a very low effort for testing life cycle-related properties with our approach. Many students used copy, paste and adjust

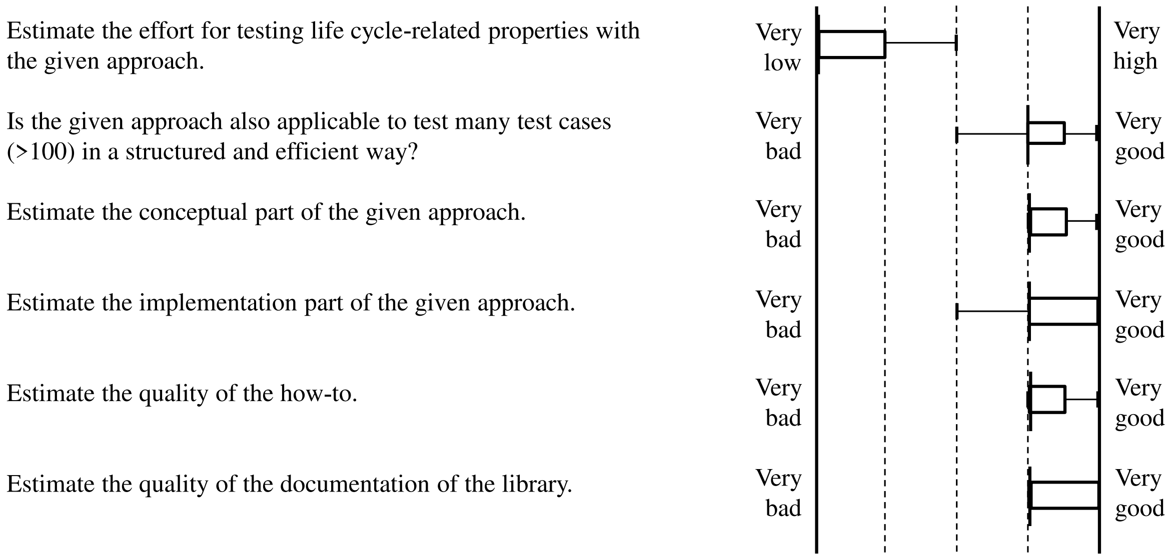

Figure 6.5: Questionnaire about our Approach

to create new test cases. This made test case generation very easy and quick. However, few students did not do all necessary adjustments after copying and pasting, which led to issues during testing. For one student, the effort to copy and paste was too much. He suggested to write a code generator with which the user can graphically specify test cases. The students also felt that our approach and our implementation scales well with an increasing number of test cases. One student, not being much familiar with testing, did not like the idea of having each test case in a separate file. He would prefer to arrange them in a different, *more clear* way (despite the fact that separate test cases can be grouped with our approach in test suites). Most students liked our conceptual approach as well as our implementation of the approach. They especially liked the separation of test and application code, the reusability of test cases, scalability and how much work the library did for them. Further, the approach and library seemed to be quite clear to them, as all students immediately started to implement the first test case after reading through the how-to. The students did also like the quality of the how-to as well as the AndroLIFT documentation. Asking for possible issues of our approach, some students had comments on the technical part of our implementation:

- If an assertion about a view element is defined, the developer needs to pass a reference to the object of interest (e.g. text input element) to the library. The reference to the object of interest can be passed as a Java object reference, e.g. a global variable. If the view element is defined in advance in xml - as it is standard procedure on Android [11], the library can access the object using the given reference. But if the view element is defined programmatically (during runtime), the library cannot access the object through this reference after the application has been killed and flushed out of RAM. Therefore, the library also has to ask for the name of the variable of the object of interest as a string, e.g. *mTextView* (see Sec. 5.2). With this information the library is capable to access the object of interest, even if it was defined during runtime and in between deleted from RAM. The string is required as the library uses *Java Reflections* for relocating the object of interest [59]. Some students did not like the idea of passing two information for each object of interest to the library. One student stated that he does not like reflections in Java at all.

- To notify the library about state changes of the Activity under test, the same code (independent of the test cases) has to be injected into each Activity. The code is injected by the developer during testing, so that he has full control of what he injects into his AUT. Some students did not like the fact that any code has to be injected into the AUT at all.

- If the user inserted new test cases into test suites and executed the suites, the old test cases (not deleted from the test suite) have still been executed. One student did not like this fact and proposed that the test cases also become aware of the test actions (e.g. an application gets paused due to an incoming call and not because the screen is being locked). This way, the developer would not have to specify a certain test case for execution, but the test cases would be executed only when the exactly specified actions (e.g. incoming call, not screen lock) are triggered.

- To make clear to the developer which objects of the AUT are accessed by our library, he has to declare the corresponding objects of interest (e.g. test input element) in the Activity under test as *public*. One student did not like this idea for security reasons.

- One student had the feeling that using the library is easier if the developer has knowledge of JUnit testing. Due to the unit-based setup and JUnit-like test organization, we agree.

- One student did not like to manually add test cases to test suites. He suggested to use Java annotations for this purpose [90].

While none of the students was able to implement all 15 test cases with an arbitrary approach within two hours, all students were able to implement all test cases with our approach (which they never used before) even quicker than in two hours. Fig. 6.6 presents the progress during test case implementation for each student, with and without our library. The different sections of each bar mark different test cases. The students are enumerated S1, S2, ..., S7. For instance, student S3 was able to solve six test cases within two hours without our library and all 15 test cases within 74 minutes with our library. This figure also highlights that without the library only one student was capable to implement more than half of the 15 test cases within two hours. Four students even only implemented two or less test cases.

Fig. 6.7 presents the times each student needed to implement every test case using AndroLIFT. One line corresponds to one student. The test cases are enumerated TC1, TC2, ... , TC15. The time was measured from the moment a student started to work on a test case until the correct test case result was printed in the Logcat after test execution.

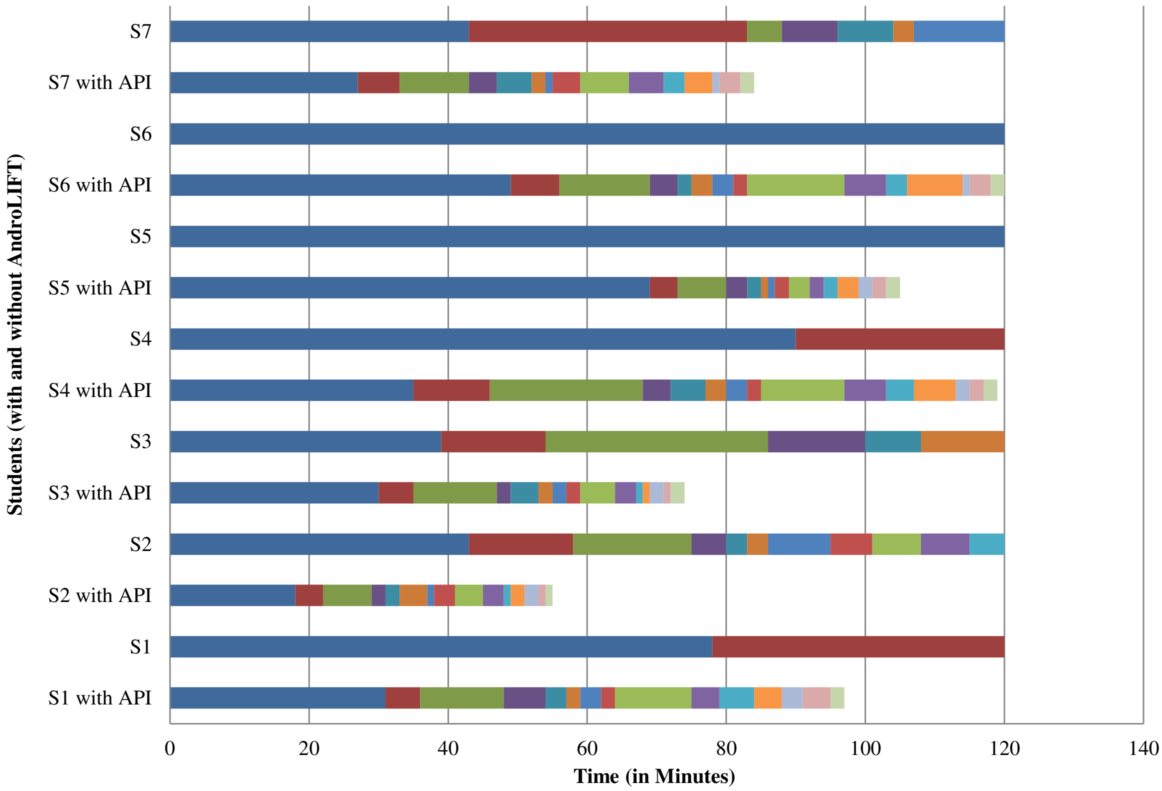

Figure 6.6: Progress Comparison of Implementing Test Cases with and without the AndroLIFT Library

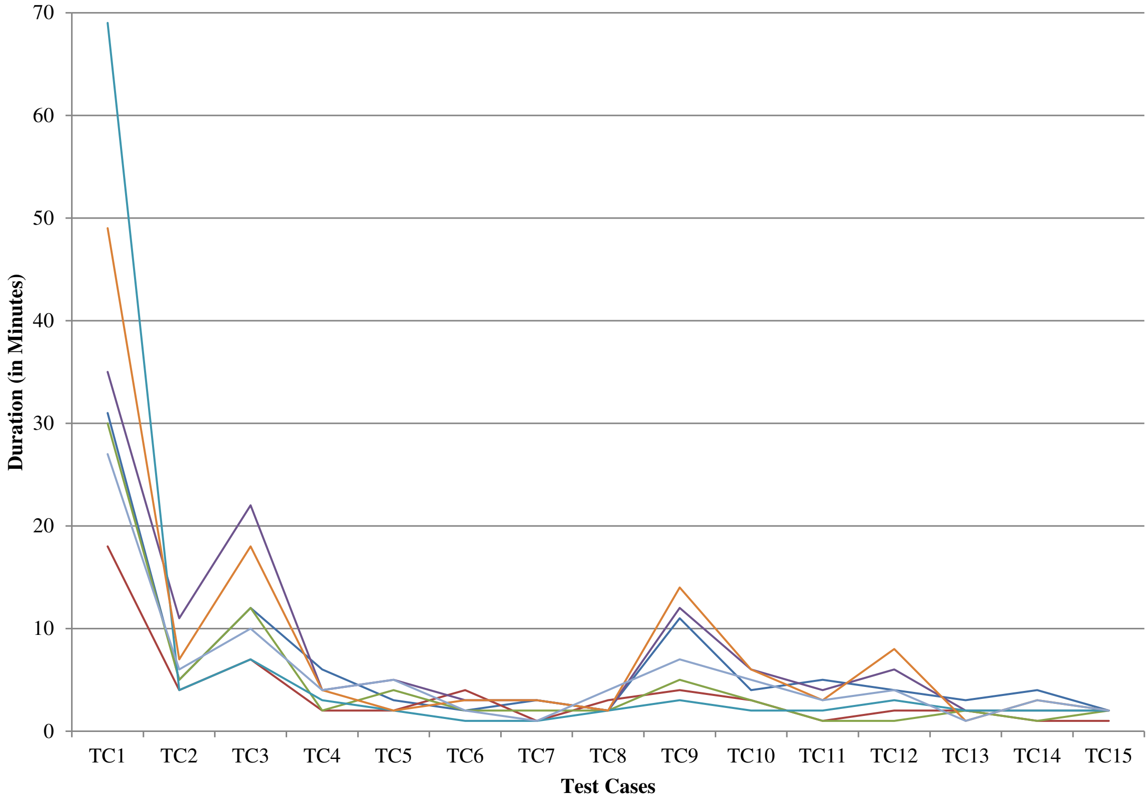

Figure 6.7: Time of each Student for Implementing every Test Case with the AndroLIFT Library

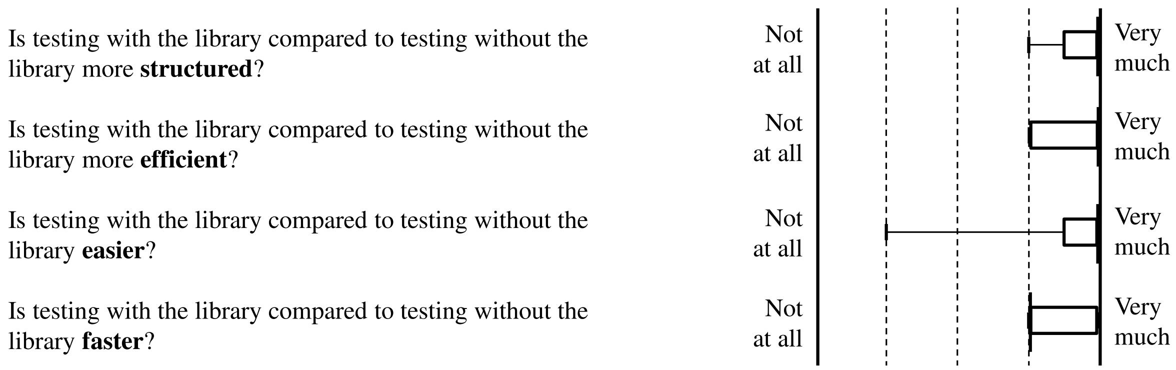

Figure 6.8: Questionnaire comparing both Approaches

All students needed most time for the first test case. The fastest student for the first test case was done with the test case after 18 minutes and the slowest student after 69 minutes. We were able to observe that the overall times of implementing test cases correlated with the programming skills of the students. The student who needed 69 minutes until he got the result from the first test case, prepared during that time all other Activities which were not required by the first test case, but by later test cases. This is also the reason why that student required only very few time during the test cases 3 and 9. During these two test cases most students required more time. The reason is that test cases 1 and 2 deal with the list Activity, test cases 3-8 deal with the note Activity and test cases 9-15 with the properties Activity. Since the students implemented the test cases in the order TC1, TC2, ..., TC15, they all had to prepare the Activity under test only once, before executing the first test case for this Activity. This explains the time peaks at test cases 1, 3 and 9, which are the first test cases in this sequence where a new Activity has to be prepared. The time peak at test case 12 results from low experience of the students with Android programming. The test case asks for the property of a view being *enabled*. The students expected this property to be specific to the view. But this property is a general property which holds for many different view elements. Thus, some of them (not all) first had to check the AndroLIFT documentation in more detail and were soon able to proceed.

In the fourth questionnaire the students have been asked to compare their own approach to the given approach. The results of the fourth questionnaire are given in Fig. 6.8. The answers depend on the self-estimation of the student's approach. For instance, one student found his own approach very easy, thus he did not value our implementation much more easier than his. But most of the other students found it even much more easier than their own approach. Overall, the students also agreed that testing with our approach is more structured, more efficient and faster than their approach. Asking the students which approach they would choose to test life cycle-related properties in the future, all students agreed to use our approach.

Although comparing in the last questionnaire our elaborated and established approach with the approach the students set up in 2 hours has a limited significance, one interesting observation can be summarized: At the beginning of the evaluation session, each student was in the role of an application developer who knows the application and all test cases very well. In this session the application is quite simple, so are corresponding test cases, too. Nonetheless it takes each developer a huge effort to implement a life cycle test case, even if he is allowed to use any approach available. The reasons for this result from the challenges when testing life cycle-related properties (see Sec.4.1). The implementation of our approach for the Android 2.2 platform provides the developer an easy way to learn, apply and efficiently implement test cases for life cycle-related properties. After a short introduction into the AndroLIFT library, all students have been able to implement all test cases in a quick and structured manner.

7 Related Work

The work by Mirzaei et al. deals with systematic generation of test cases [109]. The authors put the focus on Android as a strictly event-driven system. The events might be system-events (e.g. life cycle state changes), incoming triggers like calls and user interaction. In their approach for automated testing techniques the authors use symbolic execution to generate test cases for Android applications. Therefore, they use the tool *Symbolic PathFinder (SPF)* [122]. This tool is able to generate test cases out of Java byte-code. Since Android applications are executed on a Dalvik VM instead of a Java VM, the authors transform the Dalvik byte-code to Java byte-code. Therefore, Android-specific parts of each application are replaced by models. These models are responsible for simulating the behavior of the replaced parts, which also includes the callback behavior of the platform. This stubs are supplemented by so called *mock classes* which are responsible to simulate further behavior of the Android platform, like composed applications consisting of more than one Activity (e.g. multiple Activitites and services). With the knowledge of each application, the authors derive a Context Free Grammar to generate test drivers. A driver in this sense is a *sequence of events that simulate user's interaction with the app* [109]. These drivers are used to run test cases for single Activities as well as composed Android applications. With this setup the authors are able to automatically derive test cases from the source code of Android applications. The test cases cover system-, user- and environmental events. The events also include life cycle events like starting or pausing an Activity. To model the life cycle behavior of the Android platform, the authors use own models. The model for the life cycle behavior is given in [109]. It does not correspond to our reverse engineered Android life cycle, e.g., since after creating an Activity, it is immediately in the *running*-state instead of a *paused*-state. But as we also presented in [65], an Activity can also be paused and stopped after being created. It does not necessarily first have to be running before being stopped. If the mock classes, stubs and models for simulating the system behavior are not representing the real system behavior, test results may not be reliable. As the authors focus on automated test generation from the source code of the applications, they do not present any approach to test life cycle-related properties. Their framework can be used as a test executor for testing life cycle-related properties with the AndroLIFT library.

The work by Hu and Neamtiu describes a way of testing Android applications by using JUnit tests [91]. The focus is on GUI testing of Android applications which might consist of multiple Activities. The authors inject GUI events by using the Android tool Monkey [22] and log system information of the reacting application. The analysis of the resulting log information provides the authors also insight into life cycle-related issues of the application. Regarding the log information, the authors can track the state changes of the application. These state changes are then compared to a *state machine of an*

Android Activity, which presents the Activity life cycle as a state machine. If the life cycle state sequence from the log information is not retraceable in the state machine, the authors mark this as a potential bug. A subsequent manual analysis of the test case proves if this is a real bug. With this method the authors can reveal life cycle-related bugs if an application does not follow the regular state machine, e.g., if it is killed during a certain test case. But the authors do not focus and cover life cycle-related properties like connection or hardware status, even if the test case includes killing and starting an application again. They do not check states of components and hardware modules before and after changing a state. They do not provide a system-independent concept but focus on Android.

The proposals [34, 57, 87] use runtime verification to analyze the correctness of applications. During runtime they check the behavior of an application against models describing the desired behavior. Observer components monitor certain properties and notify the developer in case of violations of the specified properties. Especially the work by Espada et al. focuses on mobile applications and applies the approach to the Android Activity life cycle [57]. In contrast to our testing approach, runtime verification requires a setup of models describing the desired behavior. Additionally, if the model with the desired behavior is not correct (e.g. since the Android life cycle might have changed in the current version and the models have not been adjusted), the test results might not be reliable, neither. In our approach, we do not externally implement the desired behavior, but connect it tightly to the AUT (e.g. using a library or code injections). Thus, the life cycle during test execution is always the one of the real AUT and not being compared to a model or external instance.

Anand et al. present a concolic approach for automated testing of smartphone applications [27]. In their evaluation they focus on Android and GUI testing. Their approach produces efficiently and automatically event input sequences for Android applications. These event input cover touch screen events, but also can cover other events like keyboard input. The authors do not refer to life cycle events or properties. If their approach and implementation is capable of covering life cycle events, too, it could be used to automatically generate and execute event input sequences, including user input as well as life cycle state triggers. Our library would provide the functionality of checking life cycle-related properties during test execution.

In their recent article about *State Management* tests, Google describes how to use the testing tools from the Android SDK to test state-related properties [3]. This approach is similar to manually testing life cycle-related properties (see Sec. 4.2), but allows to simulate life cycle actions manually by corresponding commands. For instance, the tester can call *instrumentation.callActivityOnResume(someActivity)* to directly invoke a call to the Activity's *onResume()*-method. But this call does not really resume the application [3]. This might lead to a different behavior when directly calling these actions (without really resuming) and when they are called in the context of a real state change implied by the system. Additionally, the direct calls can be made by the developer in arbitrary order and repeatedly. This may not correspond to the possible actions during a real execution of the application. Our approach uses the real execution context of the AUT and provides a way to manage and handle assertions related to life cycle-properties. Our conceptual approach is not only applicable to the Android platform, but is basically platform-independent (see Sec. 4.4).

Bickford and Cáceres present an approach to preserve the states of mobile applications throughout different mobile devices [40]. Both data and computation state are preserved. The user can switch the mobile device while using a mobile application and proceed using the same application in the same state on a different device. The authors focus on reducing any latencies that may occur when switching the devices. The approach uses incremental live synchronization of whole virtual machines that execute the mobile applications. By synchronizing in the background during application execution, the amount of data left to be synchronized when the user decides to switch to a different device is reduced, avoiding extensive delays. The authors do not clearly define the life cycle states of the applications that are synchronized. By synchronizing the data of whole virtual machines, the included applications can be in any life cycle state. The authors do not provide information about how the state of an applications is validated or checked for consistency after synchronization. Neither do they refer to life cycle-related issues of porting virtual machines.

In the work of Benli et al. different unit testing approaches for mobile applications are compared [39]. The authors focus on Android and state that unit testing on mobile applications differs reasonable from unit testing on desktop and server platforms. Referring to the authors, Android unit tests need to reference the Android element ID, have a longer execution time and so on. After seeding few errors into an application, the authors execute White- and Black-box tests to detect the seeded errors. An interesting outcome is that testing Android Activities should be done using White-box testing as state changes do not provide much information to the outside of the Activity. Mainly Android internal state information as well as Activity internal information change. This confirms our decision to use White-box testing for life cycle-related properties. The authors do not go deeper into unit testing of life cycle-related properties.

Kaasila et al. focus on testing mobile applications across different handsets of a variety of manufacturers [92]. The work results in *Testdroid*, a tool for automated remote user interface testing on Android. Testdroid users are able to specify user action scripts for their application (push button A, check content of text field C, ...) and automatically execute these scripts on a large number of devices from different manufacturers. The test results are presented to the users and the authors report about the most common errors. The most common error is failing to install the application on a device, e.g., due to the wrong target Android version, missing permissions or unsupported screen resolutions. The authors do not state how to test life cycle-related properties. But one clear way would be to write user stories triggering life cycle actions and to execute them automatically using Testdroid. User-centric tests, like expected content of text fields could be checked with Testdroid. But as it is focusing on user interface, life cycle-related properties which are not immediately visible in the user interface (e.g. *Is the Bluetooth module on?* or *Is the e-mail content stored in the database?*) cannot be checked with Testdroid.

The work of Azim and Neamtiu presents explorations of Android applications for systematic testing [33]. As a result of an evaluation, the authors state that the average Android user only uses around 30% of the applicaion screens and only less than 7% of the application methods. The evaluation was done by seven users testing 25 popular Android applications including BBC News, Amazon and YouTube. Based on these results, the authors present two different approaches to significantly increase the screen and method coverage. The authors do not report about life cycle-related states or methods. As the visible screens on Android are bound to life cycle-related methods, there is a correlation. But the authors do not give any further information. Increasing the application screen and method coverage may also increase the coverage of life cycle callback methods. But a more complete coverage is reached by using the test catalogs from reverse engineering the application life cycles (see Chap. 3).

8 Conclusion

This work presents a conceptual approach for testing life cycle-related properties of mobile applications. In a first step, a concept for reverse engineering life cycles of applications is introduced. This concept is independent of the mobile platform. It consists of four major steps which result in more accurate life cycle models. In three case studies this concept is used to reverse engineer the life cycles of Android 2.2 Activities, iOS 4.0 and Windows 7.5 applications. The concept faces the issues resulting from inaccurate, incomplete and inconsistent official documentations on application life cycles. The resulting unambiguous life cycle models with a clear syntax and semantics improve the understanding of application life cycles for developers and serve as a basis for testing life cycle-related properties.

The test approach is based on unit testing. It identifies and separates the key components for testing life cycle-related properties as units. This allows structured testing of applications composed of multiple units. The conceptual approach is platform independent and uses assertions to define life cycle-related properties. Thereby it does not limit the number or type of testable properties.

In a case study we implement the conceptual approach for the mobile platform Android. Compared to other current mobile platforms, Android allows much access to the system and applications. One result of the case study is the AndroLIFT library, which allows to test life cycle-related properties of Android 2.2 Activities. This includes connection, data as well as hardware assertions. The library is easily extensible for further assertions and assertion types. Additionally, the library serves as the core of the AndroLIFT Eclipse plug-in. The plug-in provides the functionality of the library in a graphical user interface. Testers can use the plug-in to specify test cases without having knowledge on code level.

An evaluation reports about the capabilities and limitations of the presented conceptual approach. The potential of the approach is often tightly coupled with the access permissions of the application under test. By using the callback methods of the application under test, everything the application has access to, can be accessed during testing, too. The callback mechanisms allow to be implicitly notified in case of state changes and to react immediately. But this also makes existing callback mechanisms, which notify the application of state changes, an important requirement for this approach to be applicable. By focusing on state changes to check life cycle-related properties, requirements which hold within one single state, cannot be checked. Therefore a combination of the presented approach with already existing other approaches is necessary.

In the context of a case study with students from a practical course, the approach and the AndroLIFT library were evaluated. The students felt the approach was intuitive, it scaled well with a growing number of assertions and was easy to understand and use. After a short time of introduction into the approach and the library, the students were all able to implement and execute 15 test cases using the AndroLIFT library. They also provided useful information about how to improve the Android 2.2 implementation of the concept. One point of improvement of the usability was to consider not to use reflections any more. Another interesting point was to use annotations to define assertions.

The presented approach provides a structured way to test life cycle-related properties. It can be implemented with low effort for different mobile platforms. Many of the presented artifacts like the reverse engineered life cycle models can be reused for future implementations.

8.1 Future Work

The ongoing boom in the mobile market indicates that the number and importance of mobile applications will continue to grow in the future. While the number of PC sales decreases, the number of mobile device sales and app downloads increases continuously. Android and iOS are also forecast to further lead the mobile market so that the presented solutions in this work will keep their relevance. With the growing number of mobile applications and solutions, one of the important decision criteria for users is quality. Thus, it is expected that testing in the mobile application area remains an active and innovative field.

During our work on life cycles, we noticed that specifying requirements on life cycle properties in a clear and testable way seems to be an obstacle to developers. Requirements are often mentioned implicitly in other specification documents. A structured approach for specifying requirements on life cycle-related properties can help to lower this obstacle. We already did some corresponding groundwork in [72, 88].

Another active field in the area of testing mobile applications is test automation. An interesting challenge related to life cycles could be to automatically check for life cycle-related properties. Combining the presented approach with other approaches from the field of automated testing, e.g., automated test case generation or automated test execution, is possible. We already did some preliminary work towards this field with our Higgs tool [60, 102]. Higgs allows automated test execution of test cases including different devices and platforms (Android, iOS). It also has been tested using life cycle-related properties. But a full implementation of the life cycle testing approach for the Higgs tool was not done.

As proposed by some students, certain adjustments of the AndroLIFT library may be appropriate. This includes avoidance of assertions and introduction of annotations for test specifications.

With the increasing performance, larger displays and higher resolutions, mobile device manufacturers and platforms are about to increase the number of concurrently running and visible applications. In this case not only one single application remains in the state running, but multiple applications. Whether the presented concept is applicable without modification to such scenarios, has to be checked by analyzing the corresponding platforms and techniques.

Bibliography

[1] Open Handset Alliance. Dashboards, October 2011. URL http://developer.android.com/about/dashboards/index.html.

[2] Open Handset Alliance. Activity, April 2011. URL http://developer.android.com/reference/android/app/Activity.html.

[3] Open Handset Alliance. Activity Testing Tutorial, May 2013. URL http://developer.android.com/tools/testing/activity_test.html.

[4] Open Handset Alliance. Android Developer Tools, April 2013. URL http://developer.android.com/tools/help/adt.html.

[5] Open Handset Alliance. Gingerbread, May 2013. URL http://developer.android.com/about/versions/android-2.3-highlights.html.

[6] Open Handset Alliance. Honeycomb, May 2013. URL http://developer.android.com/about/versions/android-3.0-highlights.html.

[7] Open Handset Alliance. Application Fundamentals, May 2013. URL http://developer.android.com/guide/components/fundamentals.html.

[8] Open Handset Alliance. Testing Fundamentals, May 2013. URL http://developer.android.com/tools/testing/testing_android.html#JUnit.

[9] Open Handset Alliance. Android, May 2013. URL http://www.openhandsetalliance.com/android_overview.html.

[10] Open Handset Alliance. Android Reference, May 2013. URL http://developer.android.com/reference/packages.html.

[11] Open Handset Alliance. Layouts, June 2013. URL http://developer.android.com/guide/topics/ui/declaring-layout.html.

[12] Open Handset Alliance. Android SDK, April 2013. URL http://developer.android.com/sdk/index.html.

[13] Open Handset Alliance. Android Open Source Project, May 2013. URL http://source.android.com.

[14] Open Handset Alliance. SDK Tools, May 2013. URL http://developer.android.com/tools/sdk/tools-notes.html.

[15] Open Handset Alliance. Notepad Tutorial, May 2013. URL http://developer.android.com/training/notepad/index.html.

[16] Open Handset Alliance. App Framework, May 2013. URL http://developer.android.com/about/versions/index.html.

[17] Open Handset Alliance. API Guides, May 2013. URL http://developer.android.com/guide/components/index.html.

[18] Open Handset Alliance. Security Tips, May 2013. URL http://developer.android.com/training/articles/security-tips.html.

[19] Open Handset Alliance. Training, May 2013. URL http://developer.android.com/training/index.html.

[20] Open Handset Alliance. Using DDMS, May 2013. URL http://developer.android.com/tools/debugging/ddms.html.

[21] Open Handset Alliance. Logcat, May 2013. URL http://developer.android.com/tools/help/logcat.html.

[22] Open Handset Alliance. UI/Application Exerciser Monkey, May 2013. URL http://developer.android.com/tools/help/monkey.html.

[23] Open Handset Alliance. Storage Options, May 2013. URL http://developer.android.com/guide/topics/data/data-storage.html.

[24] Open Handset Alliance. Notepad Tutorial, June 2013. URL http://developer.android.com/training/notepad/index.html.

[25] Open Handset Alliance. Open Handset Alliance Members, May 2013. URL http://www.openhandsetalliance.com/oha_members.html.

[26] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 1st edition, 2008. ISBN 978-0-5218-8038-1.

[27] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated Concolic Testing of Smartphone Apps. In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[28] Apple. Beta Testing Your iOS App, May 2013. URL http://developer.apple.com/library/ios/#documentation/IDEs/Conceptual/AppDistributionGuide/TestingYouriOSApp/TestingYouriOSApp.html.

[29] Apple. Apple Updates iOS to 6.1, May 2013. URL http://www.apple.com/pr/library/2013/01/28Apple-Updates-iOS-to-6-1.html.

[30] Apple. Setting Up Unit-Testing in a Project, May 2013. URL `http://developer.apple.com/library/mac/#documentation/developertools/Conceptual/UnitTesting/02-Setting_Up_Unit_Tests_in_a_Project/setting_up.html`.

[31] Apple. iOS Developer Library, April 2013. URL `https://developer.apple.com/library/ios/navigation`.

[32] Ashraf Armoush, Dominik Franke, Igor Kalkov, and Stefan Kowalewski. An Approach for Using Mobile Devices In Industrial Safety-Critical Embedded Systems. In *5th International Conference on Mobile Computing, Applications and Services (MobiCASE)*. Springer, 2013. To appear.

[33] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 641–660. ACM, 2013. ISBN 978-1-4503-2374-1.

[34] Howard Barringer, Klaus Havelund, Elif Kurklu, and Robert Morris. Checking Flight Rules with TRACECONTRACT: Application of a Scala DSL for trace analysis. In *Scala Days 2011*, 2011.

[35] Graham Bath and Judy McKay. *The Software Test Engineer's Handbook*. Rocky Nook, Santa Barbara, CA, USA, 1st edition, 2008. ISBN 978-1-9339-5224-6.

[36] Danni Baumeister. *Integration of the eNav Routing Algorithm into an OSM Server*. Bachelor thesis, RWTH Aachen University, 2012.

[37] Manfred Baumgartner, Richard Seidl, and Harry M. Sneed. *Der Systemtest*. Carl Hanser Verlag, München, Germany, 3rd edition, 2011. ISBN 978-3-4464-2692-4.

[38] Arno Becker and Marcus Pant. *Android 2: Grundlagen und Programmierung*. dpunkt.verlag, Heidelberg, Germany, 2nd edition, 2010. ISBN 978-3-8986-4677-2.

[39] Selin Benli, Anthony Habash, Andy Herrmann, Tyler Loftis, and Devon Simmonds. A comparative evaluation of unit testing techniques on a mobile platform. In *Proceedings of the 9th International Conference on Information Technology: New Generations (ITNG)*, pages 263 – 268. IEEE Computer Society, 2012. ISBN 978-1-4673-0798-7.

[40] Jeffrey Bickford and Ramón Cáceres. Towards synchronization of live virtual machines among mobile devices. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 13:1–13:6. ACM, 2013. ISBN 978-1-4503-1421-3.

[41] Joshua Bloch. *Effective Java*. Addison-Wesley, Indianapolis, IN, USA, 2nd edition, 2008. ISBN 978-0-3213-5668-0.

[42] Bart Broekman and Edwin Notenboom. *Testing Embedded Software*. Addison-Wesley Professional, Indianapolis, IN, USA, 1st edition, 2002. ISBN 978-0-3211-5986-1.

[43] Ed Burnette. *Hello, Android: Introducing Google's Mobile Development Platform*. Pragmatic Bookshelf, Frisco, TX, USA, 3rd edition, 2010. ISBN 978-1-9343-5656-2.

[44] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990. ISSN 0740-7459. doi: 10.1109/52.43044. URL `http://dx.doi.org/10.1109/52.43044`.

[45] Onur Cinar. *Android Apps with Eclipse*. Apress, New York, NY, USA, 1st edition, 2012. ISBN 978-1-4302-4434-9.

[46] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley Professional, Indianapolis, IN, USA, 2nd edition, 2004. ISBN 978-0-3213-0502-2.

[47] Christian Dernehl. *Development of an Autonomous Flight Control System for Micro Aerial Vehicles*. Master thesis, RWTH Aachen University, 2011.

[48] Christian Dernehl, Dominik Franke, Hilal Diab, and Stefan Kowalewski. An Architecture with Integrated Image Processing for Autonomous Micro Aerial Vehicles. In *International Micro Air Vehicle Conference (IMAV)*, pages 138 – 145. IMAV, 2011.

[49] Elfriede Dustin, Jeff Rashka, and John Paul. *Software automatisch testen: Verfahren, Handhabung und Leistung*. Springer, Heidelberg, Germany, 1st edition, 2001. ISBN 978-3-5406-7639-3.

[50] Himanshu Dwivedi, Chris Clark, and David Thiel. *Mobile Application Security*. McGraw-Hill Osborne Media, New York, NY, USA, 1st edition, 2010. ISBN 978-0-0716-3356-7.

[51] Dzenan Dzafic. *Development and Evaluation of a Navigation System for Electric Vehicles*. Bachelor thesis, RWTH Aachen University, 2011.

[52] Dzenan Dzafic. *Integrating Surface Information into eNav*. Master thesis, RWTH Aachen University, 2014.

[53] Dzenan Dzafic and Dominik Franke. Entwicklung und Evaluation eines Navigationssystems für Elektrorollstühle. In *Gesellschaft für Informatik Seminars, Informatiktage*, pages 185–188. Gesellschaft für Informatik e.V., 2013. ISBN 978-3-88579-446-2.

[54] Dzenan Dzafic, Dominik Franke, Danni Baumeister, and Stefan Kowalewski. Modifikation des A*-Algorithmus für energieeffizientes 3D-Routing. In *Angewandte Geoinformatik 2013 - Beiträge zum 25. AGIT-Symposium (AGIT)*, pages 414–423. Wichmann Verlag, 2013. ISBN 978-3-87907-533-1.

[55] Corinna Elsemann. *Analysis of Mobile Application Life Cycles*. Diploma thesis, RWTH Aachen University, 2011.

[56] engadget German. Google IO: 900 Millionen Android-Aktivierungen, 48 Milliarden App-Installationen bisher, June 2013. URL `http://de.engadget.com/2013/05/15/google-io-900-millionen-android-aktivierungen-48-milliarden-ap`.

[57] Ana-Rosario Espada, María-del-Mar Gallardo, and Damián Adalid. A Runtime Verification Framework for Android Applications. In *XXI Jornadas de Concurrencia y Sistemas Distribuidos (JCSD)*. Distributed Systems Group (UPV/EHU).

[58] Michael R. Fine. *Beta Testing for Better Software*. John Wiley & Sons, New York, NY, USA, 1st edition, 2002. ISBN 978-0-4712-5037-1.

[59] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications, Shelter Island, NY, USA, 1st edition, 2004. ISBN 978-1-9323-9418-4.

[60] Daniel Forster. *iOS-Extension to the Higgs Framework*. Bachelor thesis, RWTH Aachen University, 2012.

[61] Dominik Franke and Stefan Kowalewski. Verifikation der Java-Echtzeitfähigkeit für den Einsatz in zeitkritischen Systemen. In *Gesellschaft für Informatik Seminars, Informatiktage*, pages 165–168. Gesellschaft für Informatik e.V., 2010. ISBN 978-3-88579-443-1.

[62] Dominik Franke and Carsten Weise. Providing a Software Quality Framework for Testing of Mobile Applications. In *4th International Conference on Software Testing Verification and Validation (ICST)*, pages 431–434. IEEE Computer Society, 2011. ISBN 978-1-61284-174-8.

[63] Dominik Franke, Dzenan Dzafic, Carsten Weise, and Stefan Kowalewski. Entwicklung eines mobilen Navigationssystems für Elektrofahrzeuge auf Basis von OpenStreetMap-Daten. In *Konferenz für Freie und Open Source Software für Geoinformationssysteme (FOSSGIS)*, pages 92 – 99. FOSSGIS e.V., 2011. ISBN 978-3-00-034124-3.

[64] Dominik Franke, Dzenan Dzafic, Carsten Weise, and Stefan Kowalewski. Konzept eines Mobilen OSM-Navigationssystems für Elektrofahrzeuge. In *Angewandte Geoinformatik 2011 - Beiträge zum 23. AGIT-Symposium (AGIT)*, pages 148 – 157. Wichmann Verlag, 2011. ISBN 978-3-87907-508-9.

[65] Dominik Franke, Corinna Elsemann, Carsten Weise, and Stefan Kowalewski. Reverse Engineering of Mobile Application Lifecycles. In *18th Working Conference on Reverse Engineering (WCRE)*, pages 283 – 292. IEEE Computer Society, 2011. ISBN 978-1-4577-1948-6.

[66] Dominik Franke, Dzenan Dzafic, Danni Baumeister, and Stefan Kowalewski. Energieeffizientes Routing für Elektrorollstühle. In *13. Aachener Kolloquium Mobilität und Stadt (AMUS/ACMOTE)*, pages 65 – 68. RWTH Aachen, 2012. ISBN 978-3-88354-164-8.

[67] Dominik Franke, Corinna Elsemann, and Stefan Kowalewski. Reverse Engineering and Testing Service Life Cycles of Mobile Platforms. In *2nd DEXA Workshop on Information Systems for Situation Awareness and Situation Management (ISSASiM)*, pages 16 – 20. IEEE Computer Society, 2012. ISBN 978-0-7695-4801-2.

[68] Dominik Franke, Stefan Kowalewski, and Carsten Weise. A Mobile Software Quality Model. In *12th International Conference on Quality Software (QSIC)*, pages 154 – 157. IEEE Computer Society, 2012. ISBN 978-1-4673-2857-9.

[69] Dominik Franke, Stefan Kowalewski, Carsten Weise, and Nath Prakobkosol. Testing Conformance of Lifecycle-Dependent Properties of Mobile Applications. In *5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 241–250. IEEE Computer Society, 2012. ISBN 978-0-7695-4670-4.

[70] Dominik Franke, Tobias Royé, and Stefan Kowalewski. AndroLIFT: A Tool for Android Application Life Cycles. In *4th International Conference on Advances in System Testing and Validation Lifecycle (VALID)*, pages 28 – 33. Xpert, 2012. ISBN 978-1-61208-233-2.

[71] Dominik Franke, John Schommer, and Stefan Kowalewski. Softwarequalität in der Mobilen Welt: Testen von Lebenszyklen Mobiler Applikationen. In *Embedded Software Engineering Kongress (ESE)*, pages 478 – 482. ELEKTRONIKPRAXIS, 2012. ISBN 978-3-8343-2407-8.

[72] Dominik Franke, Stefan Hempel, and Stefan Kowalewski. Specifying Life Cycle Requirements in a Natural-like Language and ptLTL. In *8th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*, pages 120–121. IEEE Computer Society, 2013. ISBN 978-1-4799-1324-4.

[73] Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra, and Elisabeth Robson. *Head First Design Patterns*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2004. ISBN 978-0-5960-0712-6.

[74] David Gallardo, Ed Burnette, and Robert McGovern. *Eclipse in Action: A Guide for the Java Developer*. Manning Publications, Shelter Island, NY, USA, 7th edition, 2003. ISBN 978-1-9301-1096-0.

[75] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Indianapolis, IN, USA, 1st edition, 2004. ISBN 978-0-2016-3361-0.

[76] Gartner. Gartner Says Worldwide Sales of Mobile Phones Declined 3 Percent in Third Quarter of 2012; Smartphone Sales Increased 47 Percent, May 2013. URL `http://www.gartner.com/newsroom/id/2237315`.

[77] Gartner. Gartner Says Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013, June 2013. URL `http://www.gartner.com/newsroom/id/2408515`.

[78] Felix Gathmann, Christian Dernehl, Dominik Franke, and Stefan Kowalewski. An integrated vision aided GPS/INS Navigation System for ultra-low-cost MAVs. In *International Micro Air Vehicle Conference (IMAV)*, pages 1 – 8. IMAV, 2012.

[79] Thomas Gerlitz. *Test and Stabilization of RTAndroid*. Master thesis, RWTH Aachen University, 2012.

[80] Google. Google TV, May 2013. URL `http://www.google.com/tv`.

[81] Google Code Group. robotium - The world's leading Android test automation framework, May 2013. URL `http://code.google.com/p/robotium`.

[82] Sheran Gunasekera. *Android Apps Security*. Apress, New York, NY, USA, 1st edition, 2012. ISBN 978-1-4302-4062-4.

[83] Paul Hamill. *Unit Test Frameworks*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2004. ISBN 978-0-5960-0689-1.

[84] Norman Hansen. *Reverse Engineering of the Windows Phone Process Execution Model*. Bachelor thesis, RWTH Aachen University, 2012.

[85] Norman Hansen and Dominik Franke. Reverse Engineering des Windows Phone Process Execution Models. In *Gesellschaft für Informatik Seminars, Informatiktage*, pages 83 – 86. Gesellschaft für Informatik e.V., 2013. ISBN 978-3-88579-446-2.

[86] Robert Harris and Rob Warner. *The Definitive Guide to SWT and JFACE*. Apress, New York, NY, USA, 1st edition, 2004. ISBN 978-1-5905-9325-7.

[87] Klaus Havelund. Implementing Runtime Monitors (Invited Presentation). In *2nd Time ORiented Reliable Embedded NeTworked Systems Workshop (TORRENTS)*, 2011.

[88] Stefan Hempel. *Specifying Life Cycle Requirements for Mobile Applications*. Diploma thesis, RWTH Aachen University, 2012.

[89] Steve Holzner. *Eclipse*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2004. ISBN 978-0-5960-0641-9.

[90] Cay S. Horstmann and Gary Cornell. *Core Java, Volume II - Advanced Features*. Prentice Hall, Upper Saddle River, NJ, USA, 9th edition, 2013. ISBN 978-0-1370-8160-8.

[91] Cuixiong Hu and Iulian Neamtiu. Automating GUI Testing for Android Applications. In *6th International Workshop on Automation of Software Test (AST)*, pages 77–83. ACM, 2011. ISBN 978-1-4503-0592-1.

[92] Jouko Kaasila, Denzil Ferreira, Vassilis Kostakos, and Timo Ojala. Testdroid: Automated remote ui testing on android. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia (MUM)*, pages 28:1–28:4. ACM, 2012. ISBN 978-1-4503-1815-0.

[93] Igor Kalkov. *Design and Integration of Real-Time into the Android Platform*. Master thesis, RWTH Aachen University, 2011.

[94] Igor Kalkov, Dominik Franke, John F. Schommer, and Stefan Kowalewski. A Real-time Extension to the Android Platform. In *10th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, pages 105 – 114. ACM, 2012. ISBN 978-1-4503-1688-0.

[95] Rene Klösch and Harald Gall. *Objektorientiertes Reverse Engineering: Von klassischer zu objektorientierter Software*. Springer, Heidelberg, Germany, 1st edition, 1995. ISBN 978-3-5405-8374-5.

[96] Lasse Koskela. *Effective Unit Testing: A guide for Java Developers*. Manning Publications, Shelter Island, NY, USA, 1st edition, 2013. ISBN 978-1-9351-8257-3.

[97] Tim Lange. *Entwicklung eines Implementierungsentwurfs für den Prototypen einer Softwaresimulation*. Bachelor thesis, RWTH Aachen University, 2011.

[98] lifehacker. How Much Energy a Smartphone Uses in a Year (And What it Means for Your Budget), May 2013. URL `http://lifehacker.com/5948075/how-much-energy-a-smartphone-uses-in-a-year-and-what-it-means-for-your-budget`.

[99] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, Heidelberg, Germany, 2nd edition, 2009. ISBN 978-3-8274-2056-5.

[100] Daniel Losch. *Characterization and Usage of Onboard Time Devices*. Diploma thesis, RWTH Aachen University, 2011.

[101] David Mark, Jack Nutting, Jeff LaMarche, and Fredrik Olsson. *Beginning iOS 6 Development: Exploring the iOS SDK*. Apress, New York, NY, USA, 1st edition, 2013. ISBN 978-1-4302-4512-4.

[102] Robert Mathes. *Concept and Prototype of a Mobile Cross-Device Test Framework*. Diploma thesis, RWTH Aachen University, 2012.

[103] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform.* Addison-Wesley Professional, Indianapolis, IN, USA, 2nd edition, 2010. ISBN 978-0-3216-0378-4.

[104] Kevin J. McNeish, Greg Lee, Benjamin J. Miller, and Sharlene M. McNeish. *Diving In - iOS App Development for Non-Programmers Series: The Series on How to Create iPhone & iPad Apps.* Oak Leaf Enterprises, Troy, VA, USA, 1st edition, 2012. ISBN 978-0-9882-3274-7.

[105] Zigurd Mednieks, Laird Dornin, G. Blake Meike, and Masumi Nakamura. *Programming Android: Java Programming for the New Generation of Mobile Devices.* O'Reilly Media, 2nd edition, 2012. ISBN 978-1-4493-1664-8.

[106] Reto Meier. *Professional Android 4 Application Development.* Wrox, Indianapolis, IN, USA, 3rd edition, 2012. ISBN 978-1-1181-0227-5.

[107] Marko Mijatovic. *Analysis of the Practicability of Timed Hierarchies in Signalling.* Diploma thesis, RWTH Aachen University, 2011.

[108] Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philipp Weinmann. *iOS Hacker's Handbook.* John Wiley & Sons, New York, NY, USA, 1st edition, 2012. ISBN 978-1-1182-0412-2.

[109] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android Apps through Symbolic Execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, November 2012. ISSN 0163-5948. doi: 10.1145/2382756.2382798. URL `http://doi.acm.org/10.1145/2382756.2382798`.

[110] Ralf Mitsching, Carsten Weise, Dominik Franke, Thomas Gerlitz, and Stefan Kowalewski. Coping with Complexity of Testing Models for Real-Time Embedded Systems. In *3rd Workshop on Model-Based Verification and Validation (MVV)*, pages 128 – 135. IEEE Press, 2011. ISBN 978-1-4577-0781-0.

[111] Insider Monkey. Apple Inc. (AAPL): The Top 15 Best-Selling Apps, May 2013. URL `http://www.insidermonkey.com/blog/draft-top-15-best-selling-apple-inc-aapl-apps-107811/?singlepage=1`.

[112] Heiko Mosemann and Matthias Kose. *Android: Anwendungen für das Handy-Betriebssystem erfolgreich programmieren.* Carl Hanser Verlag, München, Germany, 1st edition, 2009. ISBN 978-3-4464-1728-1.

[113] Mark Murphy. *Beginning Android 2.* Apress, New York, NY, USA, 1st edition, 2010. ISBN 978-1-4302-2629-1.

[114] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, Hoboken, NJ, USA, 2nd edition, 2004. ISBN 978-0-4714-6912-4.

[115] Microsoft Developer Network. Execution Model Overview for Windows Phone, September 2012. URL `http://msdn.microsoft.com/en-us/library/ff817008(v=VS.92).aspx`.

[116] Microsoft Developer Network. How to: Combine Silverlight and the XNA Framework in a Windows Phone Application, August 2012. URL `http://msdn.microsoft.com/en-us/library/hh202938.aspx`.

[117] Greg Nudelman. *Android Design Patterns: Interaction Design Solutions for Developers*. John Wiley & Sons, New York, NY, USA, 1st edition, 2013. ISBN 978-1-1183-9415-1.

[118] Charles Petzold. *Programming Windows Phone 7*. Microsoft Press, Redmond, WA, USA, 1st edition, 2010. ISBN 978-0-7356-4335-2.

[119] Redmond Pie. Google's Android Team Reportedly Working On A Smart Watch, Too, May 2013. URL `http://www.redmondpie.com/google-android-team-reportedly-working-on-a-smart-watch-too`.

[120] Nath Prakobkosol. *Testing Lifecycle Implementations of Mobile Applications*. Master thesis, RWTH Aachen University, 2011.

[121] Sun Java Community Process. JSR-000118 Mobile Information Device Profile 2.1 Maintenance Release, September 2010. URL `http://download.oracle.com/otndocs/jcp/midp-2.1-mrel-oth-JSpec`.

[122] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 179–180, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0116-9.

[123] Tobias Royé. *Editor-Erweiterung und Integration von Modultests in das Android Life Cycle Plug-in*. Bachelor thesis, RWTH Aachen University, 2012.

[124] RssPhone.com. Google Play Store: 800.000 apps and overtake Apple AppStore!, May 2013. URL `http://www.rssphone.com/google-play-store-800000-apps-and-overtake-apple-appstore`.

[125] Dominik Schmithausen. *Evaluierung des Ressourcen-Scheduling in zeit-heterogenen Systemen*. Bachelor thesis, RWTH Aachen University, 2011.

[126] John F. Schommer, Dominik Franke, Stefan Kowalewski, and Carsten Weise. Evaluation of the Real-Time Java Runtime Environment for Deployment in Time-Critical Systems. In *7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, pages 51–60. ACM, 2009. ISBN 978-1-60558-732-5.

[127] John F. Schommer, Dominik Franke, Tim Lange, and Stefan Kowalewski. Load Balancing for Cross Layer Communication. In *36th Computer Software and Applications Conference Workshops (COMPSACW)*, pages 476–481. IEEE Computer Society, 2012. ISBN 978-1-4673-2714-5.

[128] John Ferguson Smart. *Java Power Tools*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2008. ISBN 978-0-5965-2793-8.

[129] Softpedia. 15% Multi-Core Smartphones This Year, 45% in 2015, May 2013. URL `http://news.softpedia.com/newsPDF/15-Multi-Core-Smartphones-This-Year-45-in-2015-179657.pdf`.

[130] Statista. Global smartphone sales to end users from 1st quarter 2009 to 4th quarter 2012, by operating system, May 2013. URL `http://www.statista.com/statistics/74592/quarterly-worldwide-smartphone-sales-by-operating-system-since-2009`.

[131] Petar Tahchiev, Felipe Leme, and Vincent Massol. *JUnit in Action*. Manning Publications, Shelter Island, NY, USA, 2nd edition, 2010. ISBN 978-1-9351-8202-3.

[132] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education, Upper Saddle River, NJ, USA, 4th edition, 2003. ISBN 978-8-1775-8165-2.

[133] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition, 2007. ISBN 978-0-1360-0663-3.

[134] David Thönnessen. *Steuerung einer Fertigungsanlage mit RTAndroid*. Bachelor thesis, RWTH Aachen University, 2012.

[135] Diego Torres Milano. *Android Application Testing Guide*. Packt Publishing, Birmingham, UK, 1st edition, 2011. ISBN 978-1-8495-1350-0.

[136] ToughtWorks. Testing with Frank, May 2013. URL `http://testingwithfrank.com`.

[137] Andreas Weigelt. *Development of a Test Framework for Life Cycle Properties of Mobile Applications*. Diploma thesis, RWTH Aachen University, 2013.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,

Email: `biblio@informatik.rwth-aachen.de`

2011-01 *	Fachgruppe Informatik: Jahresbericht 2011
2011-02	Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting
2011-03	Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems
2011-04	Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars
2011-06	Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++
2011-07	Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV
2011-08	Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog
2011-09	Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing
2011-10	Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
2011-11	Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains
2011-12	Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13	Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14	Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16	Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-17	Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis

2011-18 Kamal Barakat: Introducing Timers to pi-Calculus
2011-19 Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode
2011-24 Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations
2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26 Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata
2012-01 Fachgruppe Informatik: Annual Report 2012
2012-02 Thomas Heer: Controlling Development Processes
2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems
2012-04 Marcus Gelderie: Strategy Machines and their Complexity
2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting
2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data
2012-07 André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms
2012-08 Hongfei Fu: Computing Game Metrics on Markov Decision Processes
2012-09 Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains
2012-10 Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations
2012-12 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs
2012-15 Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations

2012-16 Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets

2012-17 Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods

2013-01 * Fachgruppe Informatik: Annual Report 2013

2013-02 Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen

2013-03 Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM

2013-04 Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries

2013-05 Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

2013-06 Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation

2013-07 André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung

2013-08 Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers

2013-10 Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata

2013-12 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs

2013-13 Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators

2013-14 Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures

2013-19 Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol

2013-20 Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models

2014-01 * Fachgruppe Informatik: Annual Report 2014

2014-02 Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software

2014-03 Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide

2014-04 Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06 Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07 Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations
2014-08 Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs
2014-09 Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014
2014-14 Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks
2014-15 Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code
2014-16 Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results
2015-01 * Fachgruppe Informatik: Annual Report 2015
2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
2015-06 Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference Frontiers of Formal Methods
2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon
2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization
2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.