Evaluating Responsiveness of Android Applications based on Event-Listener Profiling

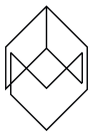
Jens-Marius Nergård Hansen

Master's Thesis in Software Engineering

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

June 2015



Abstract

As the use of mobile applications is growing and constantly being put to use in new contexts, the need for effective testing tools grows proportionally. Performing software testing on Android applications using tools such as Robotium and UI-Automator is common and widespread. These tools are intended to test functional properties of the applications. The work presented in this thesis address the need for testing properties related to user interface responsiveness in Android applications. By building on existing techniques for automatic code instrumentation of Java desktop applications in the context of Android applications, specific code segments in the application under test can be instrumented and analysed by running traditional black-box user interface test scenarios. The primary objective of this thesis was to develop an approach for evaluation of the responsiveness in Android applications. A prototype implementation called the Android Listener Profiler (ALP) was implemented to evaluate the usefulness of the approach. Evaluation was done by comparing ALP responsiveness evaluation reports with the experience of real users. This involved testing on a selection of open source Android applications. The ALP-tool was implemented utilizing Aspect Oriented Programming with the Aspectj framework to perform instrumentation of event listeners associated with View-elements. When the instrumentation has been added to the application under test, the event listeners are exercised by GUI-test scenarios. This allows event listener processing durations to be logged and persisted to a relational database, enabling post-analysis and use of descriptive statistics to improve evaluation accuracy. The evaluation results are presented in a HTML report summarising potential responsiveness issues. Testing of the Tomdroid, K-9 Mail, Text Warrior, and Sky Map applications by use of the ALP-tool has identified several event listeners that could benefit from optimization. This demonstrates empirically that the approach implemented in the ALP-tool is capable of evaluating responsiveness in Android applications.

Contents

Chapter 1

Introduction

As the use of mobile applications is growing and constantly being put to use in new contexts, the need for effective test tools grows proportionally. As the users of smart phones become accustomed and dependent on mobile applications in their daily lives, stability and performance is vital to the success of the applications. Android is one of the most popular operating systems for smart-phones and tablets as well as emerging platforms such as smart-watches and smart TV's. Having effective tools available to the developers implementing Android applications is therefore increasingly important.

Performing the necessary testing of an Android application is an expensive process as not only functional correctness needs to be considered. Performance aspects, such as responsiveness of the user interface, must also be evaluated in the process of ensuring user satisfaction. Slow responses can be caused by poor handling of network communication, file system access, processing of bitmaps, and other data-types [10, 12]. Android developers can utilize the *Strict Mode*-tool to be alerted of potentially long-running method calls being performed on the GUI-thread [10]. The worst case scenario would be that the application fails to respond within the given time-frame and the operating system terminates the application and prompt the user with a message indicating that the application has stopped or is not responding (Figure 1.1). However, much less extreme cases may also negatively impact user satisfaction. According to [6], applications must respond within 100 milliseconds to avoid being perceived as slow by the user. With easy access to alternative applications through application stores such as Google Play, users express low loyalty to their applications. According to a recent study by Localytics [7], one out of five mobile applications are only used once. This shows that developers have very little time to convince users to use their product.

One of the main ingredients of a successful mobile application is a fluid and

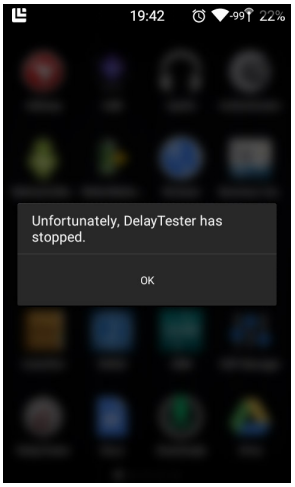


Figure 1.1: Application Not Responding (ANR).

quickly responding user interface. In order to achieve targeted performance, efficient tools are required. Being able to detect poorly performing components in an effective manner is essential when optimizing responsiveness. In [1], Shengqian Yang et al. use a combination of model-driven test generation and adding artificial delay to code segments in order to investigate the impact in user interface responsiveness during execution of the application. Work on enhancing traditional techniques of method-profiling to investigate event listener performance of Swing and AWT applications has been presented by Jovic and Hauswirth in [2]. The technique in [2], involves automatic instrumentation of listener calls as well as a manual instrumentation of the event dispatcher component in order to generate a representative performance profile.

1.1 Research Questions

The term responsiveness is used throughout this thesis to describe the amount of time it takes from a user action is received by the Android application until the required processing is completed and the result is presented to the user. In the Android operating system, the processing conducted by the AUT (Application Under Test) is initiated within event-listeners of *Activity-classes*. An Android activity-class represents a visible page within the application and usually contains buttons, menus, text-boxes, and input-fields. Activities and other Android application components are introduced later in this thesis.

Multiple frameworks for performing testing of the user interface in Android

applications exist. Use of Androids UI-Automator [4] and the Robotium framework [8] are common choices. These approaches require the developers to write a large amount of test code in order to extensively test the application through its user interface. Model-driven approaches has been developed to reduce the time spent writing tests. The use of GUI-models enables generation of test suites capable of testing a wide range of scenarios with much less effort from the developers [3, 5]. However, the tests still have an exclusive focus on functional correctness. Aside from being able to uncover Application Not Responding (ANR) messages, performance issues related to responsiveness has to be investigated separately by the developers. This can happen either through inspection of log files from test runs, or by letting users test the applications in action and provide feedback on the perceived performance.

The goal of the thesis work is to investigate how traditional approaches for functional GUI-testing of Android applications can be utilized and extended to evaluate the responsiveness of an application under test. The approach require utilities to track and measure the processing of user input, gather and store the resulting data and its associated environment information as well as the means to analyse the data and provide meaningful information to the developer using the tool.

1.2 Results

To address the introduced issues an approach involving automatic code instrumentation, test-scenario execution, and log analysis has been devised and a prototype, called the ALP, has been implemented. The prototype has allowed evaluation of the approach by applying the tool on the open source applications Tomdroid, K-9 Mail, Text Warrior, and Sky Map. The ALP-tool allow Android GUI-test cases to be used as a basis for evaluating responsiveness in Android applications. Automatic code instrumentation of the application under test is performed by use of Aspect Oriented Programming as supported by the Aspectj framework. The instrumented application under test produce log output (describing its behaviour) when being exercised by GUI test scenarios. The log messages obtained during test execution are persisted allowing post-analysis and the ability to compile the results into a human readable report.

The test results obtained from applying the ALP on open source applications has shown that the approach is capable of scaling to investigate applications of varying size and complexity, as well as being able to pinpoint poorly performing code segments within an application under test. Testing of the Tomdroid, K-9 Mail, Text Warrior, and Sky Map applications by use of

the ALP-tool has identified several event listeners that could benefit from optimization. This demonstrates the ability of the ALP-tool to evaluate responsiveness in Android applications.

1.3 Android Case Study Applications

During development of the prototype implementation, the Android Listener Profiler(ALP), a selection of open source Android applications were used to test and evaluate the approach. The selected applications span different categories of mobile applications and vary in size and complexity. Additionally, a small test application, the *Delay Tester*, was implemented to test certain aspects of the Android application platform. The applications are briefly described

Delay Tester is a minimal application consisting of only two Activities populated by a small number of View-elements consisting of buttons and text-boxes. The buttons are assigned different event listeners with delays of different lengths and types. The application has allowed for testing of static analysis techniques as well development of techniques for detecting delays with different characteristics. In chapter 7 on Tool Evaluation, the application is used for illustrating how event listeners with known behaviour is reflected in the responsiveness evaluation report produced by ALP.

Tomdroid is a note-taking application which allows saving and importing notes to the SD card as well as performing online synchronization. It is file-format compatible with the multi-platform note taking application Tomboy. Figure 1.2 shows the main user interface of the Tomdroid application, containing menu buttons for adding notes, refreshing the list, and opening existing notes. Tomdroid is moderate in both size and complexity, and has few external dependencies. This makes the application convenient to set up, build and perform tests on. Tomdroid was chosen to be used as a running example through out this thesis as well as a test target during development of ALP. The application has been download more than ten-thousand times from the Google Play Store.

K-9 Mail is a email client application with support for the IMAP, POP3 and Exchange email protocols. It has been developed since 2008 and has more than five-million downloads from the Google Play Store. The application has a high level of complexity with a code-base consisting of nearly seventy thousand lines of code. In Figure 1.3 we can see the start-up user interface when the application has been configured with a Gmail email account in advance.

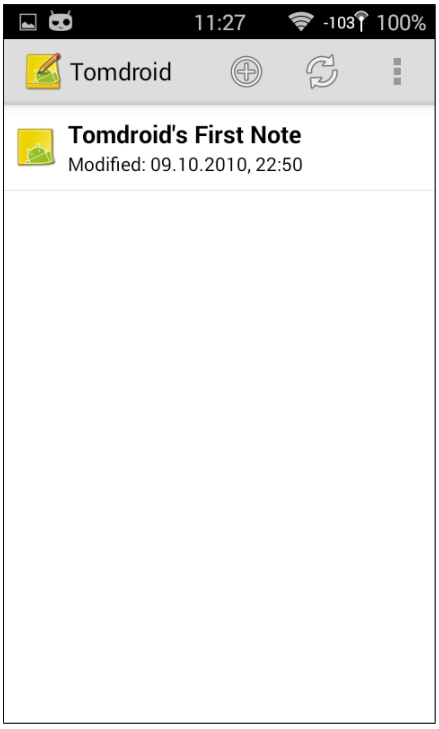


Figure 1.2: The start up screen of the Tomdroid note-taking application.

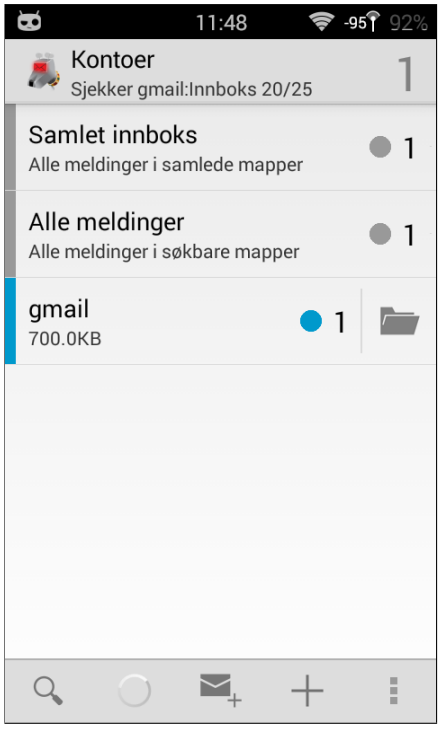


Figure 1.3: Start up screen of the K9-Email email application when signed into a Gmail account.

Text Warrior is a text editor that aims to make text editing on touch screens easier with features for fine-tuning text selection and cursor positions, syntax highlighting, clipboard selection, word wrap, and search and replace. The main screen of the application is the text view, seen in Figure 1.4. The application has been in development since 2013, and has more than fifty-thousand downloads from the Google Play application store.

Sky Map allows the user to explore the sky by pointing in the direction of interest to view the names of the planets, stars and constellations visible in that direction. In Figure 1.5 the sky map view can be seen displaying the constellation Hercules. The location of the user is obtained by use of GPS or manual setting which in combination with the compass sensor and time is used to lookup the positions of celestial bodies relative to the user. The application has been downloaded from the Google Play application store more than ten-million times.

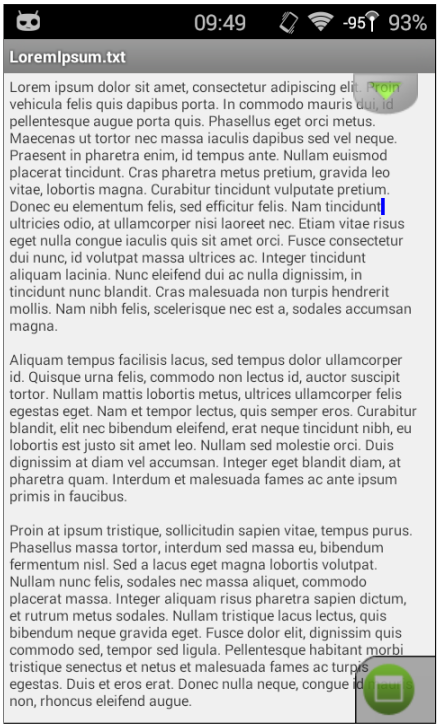


Figure 1.4: The text view screen of the TextWarrior application.

Figure 1.5: The Hercules constellation as seen in the Sky Map application.

The criteria for application selection has been the aim to cover applications with various size, complexity, and usage domains. Testing the tool on real applications as well as the Delay Tester validation application ensure that the tool can be applied in realistic environments. In addition, the selected applications consist largely of Java Source code for their user interface implementations. This allows for full instrumentation with the utilized software libraries. Figure 1.6 summarize the complexity of the case study applications in terms of lines of code and number of activities.

Application	**Activities**	**Lines of Code**
Delay Tester	2	178
K-9 Email	17	67 969
Tomdroid	14	10 079
Text Warrior	4	12 748
Sky Map	7	16 482

Figure 1.6: Complexity Comparison of Test Applications

Both the DelayTester and the Tomdroid applications are used as running

examples through out this thesis. The examples illustrate benefits and shortcomings of existing testing approaches as well as the principles used by the new ALP-tool.

1.4 Outline

The rest of the thesis is organized as follows:

Chapter 2: Android Application Development introduces Android programming and application model by providing an overview of application components and their life-cycle, the build process, and the project folder layout.

Chapter 3: Software Testing and Analysis describes software testing concepts and terminology such as functional, scenario and black-box testing as well as common tools and techniques used for analysing Android applications.

Chapter 4: Design of ALP introduces the components of the prototype implementation, focusing on the overall design and functionality. The chapter provides the reader with sufficient information to understand the purpose of each individual components and their purpose in the tool as a whole.

Chapter 5: ALP Implementation presents the inner workings of the ALP-tool. The chapter is not a complete reference of the implementation, but rather a documentation of the key aspects. The chapter is intended as a source of information in the event that a development team wish to modify the behaviour of ALP.

Chapter 6: Configuration and Usage explains how to install, configure and, use the ALP-tool with the Eclipse and Android Studio IDEs. The guide makes use of the Tomdroid application to illustrate step by step how to perform responsiveness evaluation of an application and interpret the generated report.

Chapter 7: Tool Evaluation makes an overall evaluation of the usefulness of the ALP-tool. The processing and resource overhead introduced by the tool is evaluated through a comparison with manual instrumentation and testing. In addition, the tools ability to detect responsiveness issues is investigated by applying the tool to a selection of open source applications and a comparing the results of the responsiveness evaluation report with real world user experience.

Chapter 8: Conclusions and Future Work makes a comparison of the approach with other Android performance optimization techniques and tools, before discussing possible extensions and improvements to the ALP-tool.

1.5 Readers Guide

Knowledge of the Java Programming Language or a similar object oriented programming language is assumed. Some knowledge of application testing using the Junit test framework [21] can be of advantage in understanding the testing techniques commonly used for Android application testing. Also, familiarity with modern smart-phones - performing tasks like web browsing, storing contacts, and installing applications from application stores is required to understand testing methodology used by the ALP-tool.

For Android application developers, chapter 2 can be considered easy reading and might be omitted. For Android test developers familiar with the Android SDK, chapter 3 will present familiar content. However, the terms, methodologies, and tools are used extensively throughout the thesis and should therefore be fresh in the mind of the reader.

For readers familiar with Aspect Oriented Programming and Aspectj, chapter 4.2.1 can be omitted, as only a minimal set of examples providing the required understanding of the concepts and constructs used in the implementation of ALP are included.

Figures are used throughout this thesis to illustrate and outline mechanisms and concepts. Figures containing source code, mark-up language, and log messages are often shortened with dots (*"..."*) to emphasize the aspects relevant for the current section, and should not be interpreted as complete segments unless explicitly stated as such. Inline code, e.g., the Java method signature `private void someMethod()` is highlighted using a dedicated font, while names and terms are written in *italic*.

Chapter 2

Android Application Development

The following chapter the Android programming model and associated terminology used throughout this thesis. The section might be well known for Android application developers, but has been included to make this thesis self contained. Many of the terms and concepts used in the following chapters presenting the design and implementation of the ALP-tool rely on the concepts introduced in this chapter.

2.1 Android Programming Model

Android applications are written in the Java Programming language, and compiled using the tools provided in the Android Software Development Kit (Android SDK). The Android compiler takes the source code of the application along with its resources in order to pack them into an Android package. Android packages are given the `.apk` file-name extension and can be installed on Android devices and emulators.

While it is possible to construct projects manually by use of the *android*-command line tool, it is recommended to use an Integrated Development Environment during development. This is due to the complex project structure requirements of Android applications. The Eclipse IDE with the Android Development Tools (ADT) plugin (Figure 2.1) is a common choice of development environment. More recently the Android Studio IDE (Figure 2.2), built upon IntelliJ IDEA, has emerged as an alternative.

When an application has been installed on a device it executes within a separate security perimeter. Android is a multi-user Linux-based operating

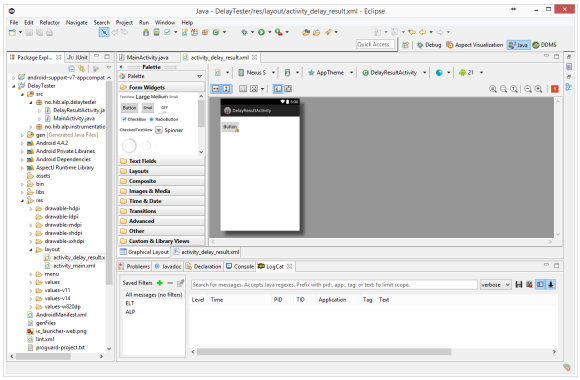


Figure 2.1: The Eclipse IDE with the Android Development Tools plugin.

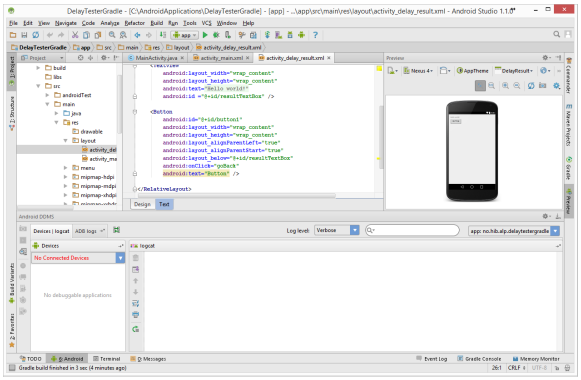


Figure 2.2: The Android Studio IDE built upon IntelliJ IDEA.

system. This allows every application to have a unique user ID, making sure that access to application files are restricted to the application itself. In addition, every application is running in its own virtual machine, ensuring that the applications cannot interfere with other applications. It is possible to arrange for two applications to have the same user ID, allowing them to access each others files and run in the same process.

When an application is opened, it starts executing on its own thread called the *main-thread* or the *GUI-thread*. The main-thread is responsible for instantiating system components, register and redirect user input events to the corresponding handlers as well as updating the screen with new frames. If the main-thread is blocked for extended periods of time, the user interface of the application becomes unresponsive. As such it is important that long duration processing is performed on other threads than the main-thread. Such threads are often referred to as *worker-threads* or *background-threads*.

Every Android application is required to have a manifest file (`AndroidManifest.xml`) located its root directory. The manifest file declare fundamental properties

of the application. This includes the ability to access device resources (like SD-Card, camera, and wireless connections), where the application should start processing when started, and which of its components that can be accessed by the system and other applications.

2.1.1 Application Components

Android applications are constructed in components that can be invoked both externally by other applications or internally by the use of `Intent`-objects. Intent-objects contain abstract descriptions of requested operations, and acts as a glue to connect components. The ability to receive intents can give an application multiple execution start locations. It can also provide a run-time coupling to functionality contained in external applications. As an example, an application may want to display a web page, but does not provide a built in web browser. Instead, an intent is constructed with a description to open a URL in an application capable of fulfilling that operation. If the device has multiple web browsers capable of handling the operation, and a default application has not been selected, the user will receive a query dialog allowing the user to select the preferred application (Figure 2.3). Intents can also initiate components that is not directly visible to the user like the *Service* and *Broadcast Receiver* components.

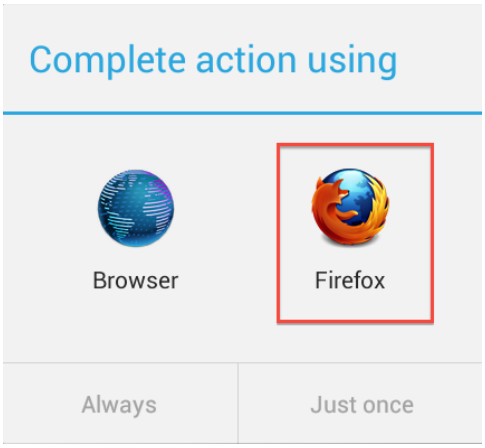


Figure 2.3: Selection of application to handle a web-page intent.

The different kinds of Android application components are discussed in detail below.

The **Activity**-component represents a single screen visible to the user. The user interface is constructed from a hierarchy of view-elements (instances of the `android.view.View`-class) that can display text, pictures or other graphics. The most common method of defining the view-element hierarchy

of an activity is through XML (Extensible Markup Language) layout files located in the resource directory of the Android application. View elements allow the user to provide input to the application through *event listeners* (introduced later in this chapter). While an XML file define the user interface layout, a Java class that extend the `android.app.Activity`-class describes the behaviour of the Activity.

An application typically consist of multiple activities, e.g, the Tomdroid application consist of 14 activities. An activity can be opened from within the application, or from the outside via intents, if the application allows it. When used for video, images or web browsing, activities are often made full-screen, hiding the top menu drawer of the Android operating system. Activities can also be shown as floating windows, covering only a part of another activity. Such activities are often used to provide extra information or menu options.

Activities are managed trough an *activity-stack*. Upon initial start-up the activity is placed on top of the stack and become *active*. The previous activity will have the second place on the stack and cannot become active until the current activity has been stopped. An activity has four different states:

Active The activity is visible on the screen and on top of the activity stack.

Paused The activity is still visible on the screen, but is partially covered by another activity. State is maintained, but the system can kill the activity in cases of very low memory.

Stopped when the activity is completely covered by another activity. State is maintained, but the activity gets killed when memory is required elsewhere.

Uninitiated when the activity must be restarted and restored to a previous state before it is ready for use.

The activity life-cycle is defined by a set of methods that allow the activity to perform the required processing during state transitions. There are seven methods in total, with only the `onCreate()`-method being required to be overridden in activity implementations. In addition, the implemented life-cycle methods must call their super method before continuing additional processing. Each of the methods are described in Figure 2.4.

Method	Description
onCreate()	Called the first time the activity is created. Used to perform the initial set-up of the activity such as creating views and add data. The method also receive a *Bundle* instance that contain the prior state, allowing state restoration when available.
onRestart()	The method is called before the `onStart()`-method when transitioning from the stopped state.
onStart()	Called when the activity is becoming visible to the user.
onResume()	Called just before the user can start interacting with the activity.
onPause	Called when another activity comes to the foreground. The method is used to save states to persistent storage. The method should be fast, as another activity will not become active until this method has completed processing.
onStop()	Called when the activity is no longer visible to the user.
onDestroy()	The last call before the activity is destroyed.

Figure 2.4: Activity life-cycle methods.

Event listeners (ELs) in Android applications are most commonly associated with view-elements displayed in the activity components. They perform the processing required by the application in order to generate a response to user interactions such as editing text or clicking buttons. The processing often involve manipulation of data, web communication or switching between different activities. The processing is by default executed on the main-thread of the application. To ensure a smooth user experience, the application developers must be careful to avoid long duration processing in such methods. This makes the event listener handler methods the profiling target of the approach presented in this thesis.

Event listeners are regular Java methods that are associated with specific events of the view-elements. The associations can be defined either programmatically in the application source code, or through XML in layout files. The programmatic approach is achieved either by implementing Java interfaces directly in the activity class and overriding the desired methods, or by attaching anonymous inner classes to View elements through set-methods. The latter approach is illustrated in Figure 2.5, showing the declaration of an event listener that is executed when the cancel button of a dialog-box in the Tomdroid application is clicked.

```
1 progressDialog.setButton(ProgressDialog.BUTTON_NEGATIVE,
2   getString(R.string.cancel), new DialogInterface.OnClickListener() {
3     public void onClick(DialogInterface dialog, int which) {
4       progressDialog.cancel();
5     }
6   });
```

Figure 2.5: Assignment of listener method for progress dialog in the Tomdroid application.

Assigning event listeners methods using XML is achieved by adding the `android:onClick` attribute to the view-element holding the event as seen in Figure 2.6 line 11. The handler methods are loosely coupled to the view objects as the association is discovered at run time. This is achieved by use of the Java Reflection API after the XML file has been parsed and the view element has been instantiated. As such, the method must have a signature matching the one sought for in the view-class. This means the the method must be public with a return type of void and a single view-element as parameter. If the method cannot be found, a `java.lang.IllegalStateException`-exception is thrown.

```
1  <RelativeLayout
2          xmlns:android="http://schemas.android.com/apk/res/android"
3          xmlns:tools="http://schemas.android.com/tools"
4          tools:context="no.hib.alp.delaytester.MainActivity" >
5          <Button
6                  android:id="@+id/launch0Btn"
7                  android:layout_width="wrap_content"
8                  android:layout_height="wrap_content"
9                  android:layout_alignParentLeft="true"
10                 android:layout_alignParentTop="true"
11                 android:onClick="launchWith0Ms"
12                 android:text="0 ms"  />
13 </RelativeLayout>
```

Figure 2.6: A layout file in the DelayTester application assigning a single event listener.

Services are used to run tasks in the background and has no user interface. Typical tasks performed using a service are playing music or transferring files. Services are started by other application components and can continue to run after the user has switched to another application. A service does not run in its own thread or process by default, hence requested services are performed in the thread of the caller.

The life-cycle of a service is managed either as *bound* or *started*.

Bound A service can be bound by a component calling its `bindService()`-method. When bound, a server-client interface is activated allowing components to send requests and fetch results. Services can be bound across different processes and have multiple clients. When all clients are unbound, the service will be destroyed.

Started When the some application component calls the `startService()`-method, the Android system will start the service. The service can run indefinitely or it can stop itself by calling its own `stopSelf()`-method. It can also be stopped by other components calling the `stopService()`-method of the service.

Services must be declared in the manifest with a `<service>` element as a child of the `<application>` element. It is possible to add attributes to specify permissions or specific processes it should execute in. Once the application has been published, it is advisable not to change the service name, as it might break code depending on the service. It is also possible to make the service private to the application by setting the `android:exported` attribute to false. A service is implemented by extending the `android.app.Service`-class and implementing the required methods.

Content Providers are used to manage shared application data in persistent storage locations. Through a content provider, applications can access the data if they are permitted to do so by a declaration in their manifest file. Android by default supplies a set of content providers to manage audio, video, images, and contact information. In the Tomdroid application, a content provider manages storage of notes in the `org.tomdroid.NoteProvider`-class using an Sqlite database. Content providers are implemented by extending the `android.content.ContentProvider`-class.

Broadcast Receivers receives and handles system-wide messages. Broadcasts can originate from the Android system itself, e.g., announcements regarding changing screen-state or low battery. Applications may also send broadcasts. Typical usage is announcing that some file has been downloaded and is ready to be used. Broadcast receiver components do not have a user interface, but they can post notifications. Typically, broadcast receivers are used to initiate a background service if some event occur. Broadcast receivers are implemented by extending the `android.content.BroadcastReceiver`-class.

2.2 Building Android Applications

Android applications are compiled and built using the tools provided in the Android Software Development Kit. The build process is completely integrated into Android Studio and Eclipse IDEs, but can also be performed manually through command line operations. While the two build approaches have some differences they can both be configured and extended as needed by the ALP-tool. The folder structure of Android application projects depend on which build automation system is being utilized for the particular project. The ALP-tool is required to navigate the folder structure of the AUT in order

to analyse the relevant files and add instrumentation. In addition to this, ALP requires execution of an additional compile step during the application build process.

The build process utilize the source code of the application in combination with the attached resources and interfaces to produce an Android package that can be installed on Android devices and emulators. The process involves multiple steps and processes that can be automated using various techniques. In Android Studio, the build process is automatically managed through the *Gradle* build automation system. As seen in Figure 2.7, the AAPT (Android Assets Packaging Tool) use the application resource files to generate the *R.java* which combined with the Java interfaces from the AIDL-tool (Android Interface Definitions Language) and the application source files are compiled by the a Java compiler. The .class files are then converted into Dalvik byte code by the Dex-tool before it is converted to and Android package and signed either for debugging or release.

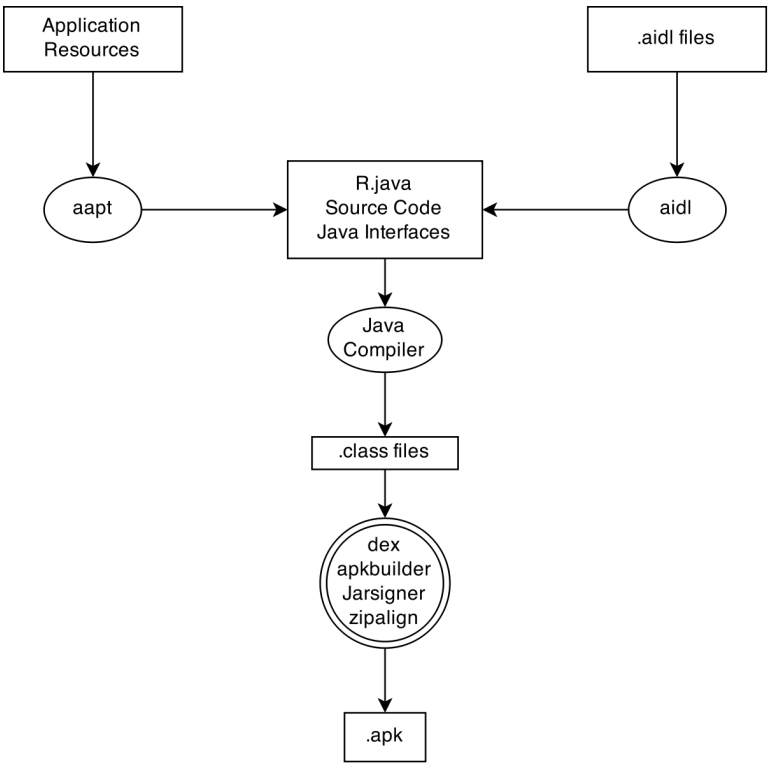


Figure 2.7: The simplified Android build process. Processes are drawn in ellipses with double lines for sets of processes while static resources are drawn in rectangles.

2.2.1 Project Folder Structures

The folder structure of the two build system utilize different application project folder structures, and as such ALP-tool has been implemented to to navigate both types. The following section describes both project layouts.

Figure 2.8 shows the folder structure of Android projects created using the Eclipse IDE. All project configuration files are located in the root folder of the application.

```
/ ...  Project top folder.
├── src ...  Java source files created by the application
│            developers.
├── gen ...  Java source files generated by aapt.  The files
│            should not be modified manually.
├── assets ...  Can contain any content that should be bundled
│               with the application.
├── bin ...  Folder for storing intermediate and final output
│            of the compile process.
├── libs ...  External libraries used by the application.
├── res ...  Contains a hierarchy of sub folder organizing
│   │        the various application resources such as images,
│   │        audio files, and XML files.
│   ├── values ...  XML files which define variables describing
│   │               themes, styles, button text and more.
│   ├── layout ...  Describes hierarchies of view elements
│   │               populating the user interface of Activities.
│   ├── menu ...  Contains menu descriptions.
│   ├── raw ...  Any kind of resource which can be looked up by
│   │            resource id.
│   └── drawable ...  Storage of bitmaps.  Often multiple versions of
│                     the bitmaps to accommodate different screens of
│                     different resolutions.
├── AndroidManifest.xml ...  Project manifest file that declare
│                             components and their permissions.
└── project.properties ...  Contains project information such as build
                            platform target (API level).
```

Figure 2.8: Eclipse Android project directory layout.

Applications using the Gradle build system as created by the Android Studio IDE has a project structure that can include test projects and sub-modules. The applications consist of the same elements as seen in the Eclipse folder structure except that the `.project`-file is replaced by Gradle build scripts. Figure 2.9 outlines the folder structure of Android projects created using the

Android Studio IDE. Only the parts relevant to the ALP-tool is included in the figure.

Android Studio projects consist of modules that encapsulate different types of source code and resources:

Android Application modules contain application source code, resources and configuration files. All contents are used when the Android package file is created.

Test modules contain test source files that are built into the test application running on the device during testing.

Library modules contain source code that can be shared between multiple applications. The files are included into the application .apk file at build time.

```
/ ...  Project top folder.
├─ build ...  Build output cache for all modules.
├─ gradle ...  Gradle-wrapper files.
├─ gradlew ...  Gradle startup script for Unix.
├─ gradlew.bat ...  Gradle startup script for Windows.
├─ build.gradle ...  Build configurations for all modules.
├─ gradle.properties ...  Project wide Gradle settings.
├─ settings.gradle ...  Listing of sub module to be built.
└─ app ...  Application module with source code, resources
   │        and configuration files.  Module folder names are
   │        selectable and specified in settings.gradle.
   ├─ build ...  Build output.
   ├─ src ...  Application files.
   │  ├─ androidTest ...  Android instrumentation tests.
   │  └─ main ...  Application source code, resources and
   │     │         AndroidManifest.xml.
   │     ├─ res ...  Contain sub folders for layout files, menu
   │     │           specifications, variables and others.
   │     └─ java ...  The java source code of the application.
   └─ libs ...  Libraries private to the module.
```

Figure 2.9: Android Studio project directory layout.

Chapter 3

Software Testing and Analysis

This chapter describe different concepts, terms and technologies that are used to put the ALP-tool into context of software testing and Android debugging techniques. For test developers the sections describing general terms should be well known, but has been included to make the thesis self contained.

3.1 Software Testing Concepts

Functional and Non-Functional Requirements specify different qualities and behaviours that the system should have. A functional requirement describes operations the system needs to perform. The operation can be described as a set of inputs that result in a specific behaviour and output. The set of functional requirements describe what the system should accomplish. Non-functional requirements support the functional requirements by adding quality constraints. A non-functional requirement is typically related to security, performance, or reliability. The purpose of the work presented in this thesis is to aid with testing of non-functional requirements regarding responsiveness of the user interface of Android applications.

Scenario Testing involve utilizing use-cases that describe ways the system might be used. Use-cases should represent realistic application usage in a way that allows the intended users to relate to them. Scenarios can often be found in design specifications and can be formalized into executable specifications which in turn can be executed on the actual application. Scenarios are ideal for testing event-listener execution times as they have the potential of providing realistic input for the application to process. When targeting the Android platform, scenarios can be implemented using the Robotium Framework or the UI-Automator API.

Black-box and White-box Testing are terms describing the type of knowledge used when developing tests. Black-box testing tests system components without using any knowledge of their internal structure. White-box testing on the other hand does use knowledge of the inner workings and internal structure of the source code as a basis for construction of test cases. Use of internal properties when choosing test input values, also allows the tester to exercise a precise execution path within the application, thus enabling detailed debugging of known faults. The approaches can be applied on any level of software-testing, from unit to GUI, and manual-testing. Black-box testing on the Android platform could be implemented in the form of GUI-tests where only the coordinates of clicks and swipes where specified in the test source code.

Regression Testing is used to investigate the consequences of changes made to any part of the system. The investigation can be the result of changes in configuration files, hardware, application versions or the underlying software platform. In contrast to other testing approaches, regression tests are not executed to confirm that the changes contributed the desired effects. The intent of regression testing is to make sure that the changes does not introduce new faults or performance issues in previously available functionality. This can be achieved by re-running previously approved and completed test-cases on the modified version of the system and comparing the results to identify changes in behaviour. When testing Android applications, regression testing is often performed by use of the Android Emulator (provided with the Android Development Kit) to execute functional test-cases on many versions of the Android operating system using different hardware set-ups.

Static analysis is the process of analysing a system without executing the implementation. It involves inspection of the application source code, reviewing design specifications, or inspection of system models. Static source code analysis is a commonly used approach to improve code quality and detect code-level problems. The process can to some extent be automated to reduce costs. Most automated static code analysis tools require no more than the source code as input and often integrate directly with the development environment. This means that it have the capability of providing developers with quick feedback on their work. While the tools are limited to catching the low-hanging fruits and lack the ability to detect issues related to complex business logic or subtle concurrency problems, static analysis cause very low drag in the development process and provide immediate benefit. Android Lint [11] is an example of a static analysis tool designed for the Android platform. It is capable of scanning applications for issues such as possible optimization improvements, correctness and security issues.

3.2 Android Test and Debug Utilities

Developing applications for the Android platform requires functionality for controlling either an Android emulator or a device for performing testing, debugging and execution. This is most often achieved by use of the tools supplied with the Android SDK which are introduced in this chapter.

3.2.1 Dalvik Debug and Monitor Server

The Dalvik Debug and Monitor Server is a tool provided with the Android SDK which allow convenient access to device information and resources such as logs, screen capture, incoming calls and SMS, location data, files system access, among others. The tool integrates with both the Eclipse and the Android Studio IDEs. Figure 3.1 shows DDMS within the Eclipse IDE, currently displaying information about the memory usage on a Samsung Galaxy S2 Smart phone.

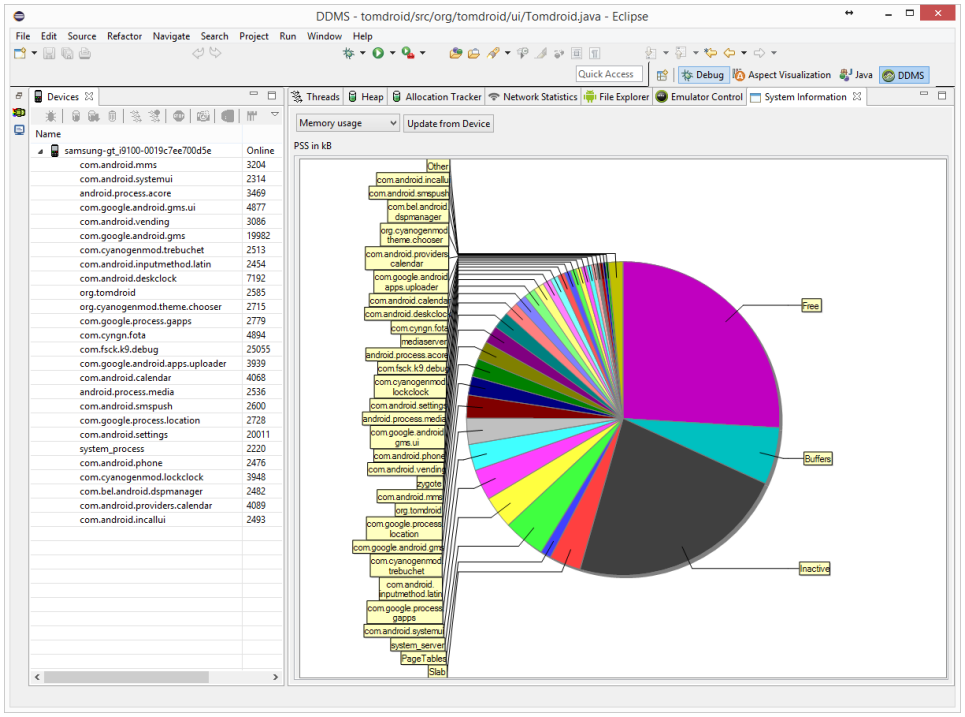


Figure 3.1: Memory usage of a smart phone visualized by Dalvik Debug and Monitor Server (DDMS)

3.2.2 Android Debug Bridge

The Android Debug Bridge (adb) provides the means to communicate with the Android device or emulator. It has a command line interface and consist of a client, a server, and a daemon as seen in Figure 3.2. The client and the server runs on the development machine, while the daemon runs on the Android device or emulator itself. The network may consist of multiple clients and daemons, but only one server. Every client is allowed to control and access every connected Android device and emulator. Clients can be used within IDEs to install applications or run tests on Devices and Emulators. An example of a client is the DDMS. When a client is initiated, it checks whether there is a server running and connects to it. If a server cannot be found or is unavailable, a new one is started. In order for the server to locate and connect to Android devices, USB debugging must be enabled on the device. It is also possible to start the adb daemon in *tcpip*-mode, allowing communication over a local area network.

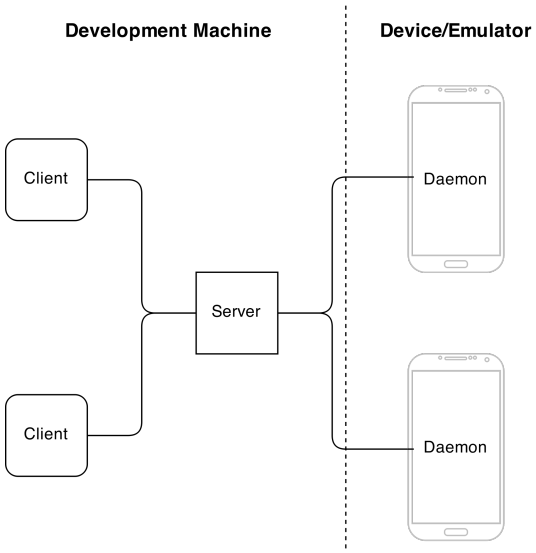


Figure 3.2: *Android Debug Bridge* components and architecture

3.2.3 Logcat

Logcat is a log reader tool built into the Android operating system. It provides a mechanism for collecting and grouping system and application messages. Messages are stored in circular buffers that can be read by use of `logcat-` from the command line, or through the Dalvik Debug Monitor Server (DDMS). Three buffers are used for storing messages from different device components:

Radio Contain messages related to radio and other mobile network communication.

Events Contain event related messages.

Main is the default buffer storing all other messages.

Log messages are printed from the Android application through calls to the `android.util.Log`-class. A message can have one of five severity levels: verbose, debug, information, warning, and error. Message tags are specified in the statement printing the log entry and are used for categorization. Figure 3.3 line 1 shows an example of writing an *informational* log message from the `org.tomdroid.ui.Tomdroid`-class of the Tomdroid application. The statement prints a message with the tag *ALP*, which can be seen in the corresponding Logcat output seen on line 2. The output message lists the fields: severity level (I), tag (ALP), process identification (8240) and the message payload separated by a colon. It is possible to modify the output format of the logcat command line tool by adding arguments to the command.

```
1 Log.i("ALP", getClass().getName()+ ": onCreate completed.");
2 I/ALP     ( 8240): org.tomdroid.ui.Tomdroid: onCreate completed.
```

Figure 3.3: Logging within an Android application (line 1) and the resulting log output collected using Logcat (line 2).

3.2.4 Android Performance Analysis Tools

The Android SDK provides two utilities that are specifically intended for investigating application performance and hence linked to the objective of this thesis. The tools Traceview and Systrace are both capable of providing detailed performance metrics but has some important drawbacks explained in the following sections.

Traceview

Traceview [18] use a log file (trace-file) as input and visualizes it, allowing developers to locate long-duration method-calls. There are two different ways to create Trace-files, i.e., initiate method-tracing. Tracing can be controlled from within the AUT by use of the `android.os.Debug`-class directly through the methods `startMethodTracing()` and `stopMethodTracing()` or by use of the method profiling feature of the *Dalvik Debug Monitor Server* (DDMS). While tracing through DDMS is difficult to control precisely, it allows for tracing without modification of the AUT source code.

The trace-file can only reflect performance as percent-wise changes when comparing traces from executions of different versions of the same application and code segments [9]. This claim is reinforced by comparing measurements obtained using the `startMethodTracing()`-method to that of the ALP-tool. By tracing the `onResume()`-method of the *Tomdroid*-class and visualizing the trace file using Traceview (Figure 3.4), the total execution time can be read on the top level row from the column named *Incl Real Time*. The *Incl Real Time*-column display the wall clock measurement of the method tree. The execution time for the complete method tracing reads 18.994 milliseconds, while the ALP-reading of the same method reported 4 milliseconds. The difference is caused by the recursive method profiling technique of Traceview. Where ALP only perform a single measurement (from the start of the method to its end), Traceview measure every single method executed during the trace, including those of every child method call. This contribute to the observed processing overhead.

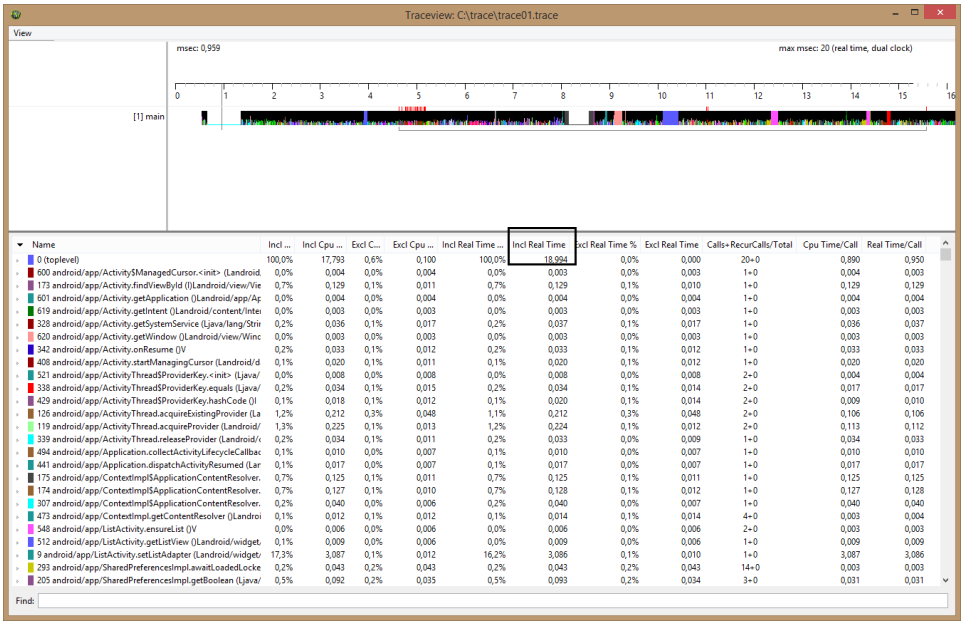


Figure 3.4: Method tracing with Traceview in `Tomdroid.onResume()`

Traceview is a powerful tool for in-depth investigation of poorly performing code segments, but it is not useful for investigation of real world application performance. It should be noted that when applied repeatedly, the results of the method tracing are comparable to each other and can be used to reflect percent-wise changes in performance. However, Traceview cannot indicate whether the execution time of an event-listener has reached an acceptable level.

Systrace

Systrace [19] allows for detailed inspection of application performance by combining data from the Android kernel to construct a report that provide an overall picture of the processing on the device. Systrace can be initiated either through DDMS or from the command line both allowing selection of specific system resources to be traced. The tracing results are presented using HTML documents. To improve readability of the report, it is possible to add markers to the log-file through instrumentation in the AUT using the methods `beginSection()` and `endSection()` of the `android.os.Trace`-class.

Figure 3.5 show 1.4 seconds of a trace conducted on the Tomdroid application while opening a note from the Tomdroid activitiy which results in an activity switch to the ViewNote-activity. *Graphics* and *View System* was the only resources tracked during the trace. The `org.tomdroid`-rows display processing within the Tomdroid application. The horizontal length of the bars represent processing time within system and application classes.

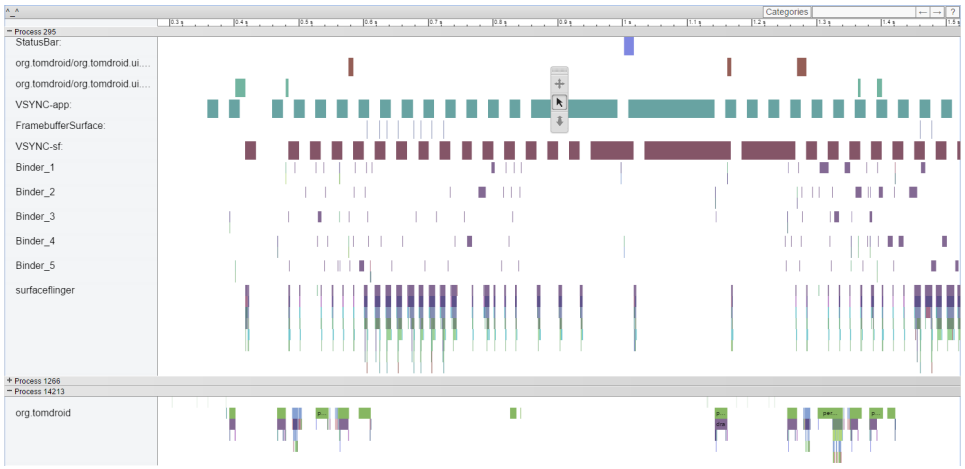


Figure 3.5: Tracing with Systrace in the Tomdroid application.

As Systrace has very low overhead it is capable of reflecting real world application performance. However, due to the huge amount of information produced during tracing it is a time consuming process for a reviewer to determine whether or not slowness exist within an application. A more realistic use is performing in-depth investigation of known performance issues to determine their causes.

3.2.5 Android Application Testing

With the Android Test Framework [20], test methods are organized into classes, called *test cases* or *test suites*. Each test is an isolated test of an individual module in the AUT. The classes are containers for related test and helper methods. The Android build tools use the test source files to create a test package that is loaded by the test tools and executed by the test runner classes. The test cases are again organized into test projects that contain the test source code and manifest file. The Android SDK provides means for generating the test project with the required files and folder structure structure. The test project can be located anywhere in the file system but is commonly located in a sub folder of the AUT, as described in section 2.2.1 on project directory layouts.

The Android framework includes a set of utilities that allows for different types of testing as well as extensions for testing of the different Android Application components. While the regular Junit test cases are sufficient to test classes that does not use the Android API, testing of classes that depend on the Android API must be tested by extending the `AndroidTestCase`-class and be executed using an Android specific test-runner.

Test cases testing Android Application components utilize the Android Instrumentation Framework, which allows for independent control over specific parts of the AUT. This allows the test cases to call Android component life cycle methods directly, and proceed through their life cycles step by step. The instrumentation framework runs the test project and the AUT in the same process (see Figure 3.6), allowing the test case to examine and modify fields in AUT components directly. The *InstrumentationTestRunner* executes the test cases from the test package on the AUT. The InstrumentationTestRunner class is the main Android test runner class. It extends the Junit runner framework and is capable of executing any test case class provided with Android.

The Android test framework provide test case classes that extends the Junit `TestCase`-class with Android specific functionality. The classes each specialize in testing specific components using different testing approaches. The `AndroidTestCase`-class provides the standard Junit methods `setUp()` and `tearDown()` as well as the assert-methods. Additionally, functionality to test permissions and to protect against memory leaks by clearing out class references is included. The component specific test case classes help address the different needs when testing the specific components as well as methods for setting up mock objects.

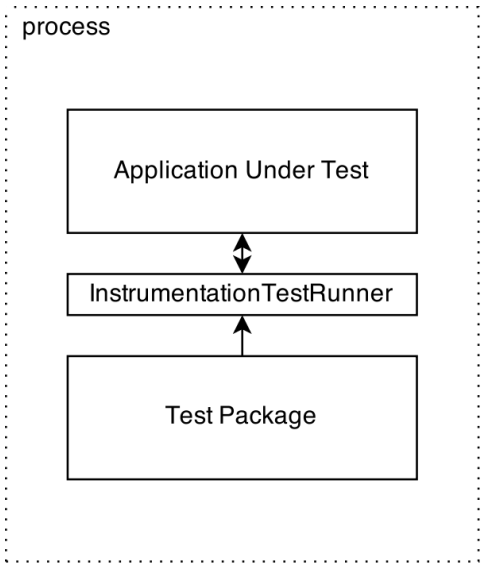


Figure 3.6: The Android Test Framework principle.

Activity Testing

Invocations to activity life cycle methods can only be performed by the Android system itself or through the Instrumentation framework. The only means of providing simulated user interaction to the application is through the Instrumentation framework and the classes introduced below.

`ActivityInstrumentationTestCase2` is used to perform functional UI-testing and is capable of testing one or more activities running within the complete application context.

`ActivityUnitTestCase` is used to test a single activity isolated by running within a mock Context or application.

`SingleLaunchActivityTestCase` allows for testing a single activity within a context that do not change between tests as the `setUp()` and `tearDown()`-methods are executed only once for the entire class instead of for each test method.

3.3 Execution of Test Scenarios

During development of the ALP-tool, the Robotium framework [8] was chosen to implement test scenarios. The Robotium framework extends the Android testing framework, making tests easier to write by providing a more streamlined API.

Obtaining a set of test cases that extensively test all major aspects of the AUT is important when evaluating perceived responsiveness. Implementing a sufficiently large set of tests like the one to be presented in section 3.3.1 through manual coding is a laborious task. Not only must the test scenarios be mapped to Robotium API-calls, the test developer must also locate id-tags to identify view elements as targets for input events. In order to speed up the process of obtaining test-cases alternative methods has been developed. Tools like Testdroid and Robotium Recorder [16, 17] is capable of registering normal user interactions and generate Robotium test cases based on these.

During the thesis work, the Testdroid recorder was used to construct Robotium test cases. Testdroid recorder integrates with the Eclipse IDE allowing quick rerunning of, and modification to, the generated test scenarios.

3.3.1 Robotium Testing

Robotium test classes extend the `ActivityInstrumentationTestCase2`-class. As with Junit, test-classes implements `setUp()` and `tearDown()` methods to provide life cycle management. Test cases are implemented using methods prefixed with the *test*-keyword. Figure 3.7 shows a test-case which creates a new note by clicking the new note button (line 2) and entering a title (line 3) and some note content (line 4) before saving the note (line 5). Line 6 evaluate if the test should pass by looking for the new note in the final list of notes that is opened after the save operation. Figure 3.8 shows the Note list activity before and after running the test. The icons seen in the top menu-bar allows for adding new notes, refreshing the notes list and opening the options menu.

```
1 public void testAddANote() {
2   solo.clickOnView(solo.getView("@id/menuNew"));
3   solo.enterText((EditText) solo.getView("@id/title"),"example note");
4   solo.enterText((EditText) solo.getView("@id/content"),"Note");
5   solo.clickOnView(solo.getView("@id/edit_note_save"));
6   assertTrue(solo.searchText("example note"));
7 }
```

Figure 3.7: A Robotium test-scenario for creating and saving a new note in the Tomdroid application.

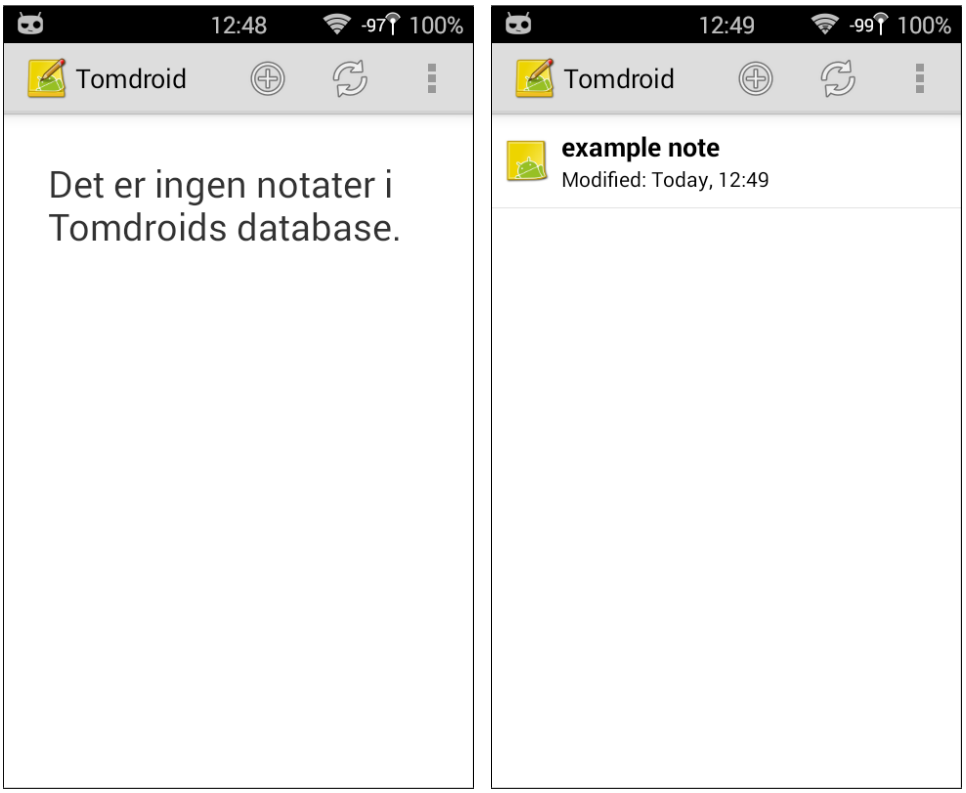


Figure 3.8: Tomdroid main activity before (left and after (right) execution of test-scenario seen in Figure 3.7.

Chapter 4

Design of the Android Listener Profiler (ALP)

The aim is to provide the developer with convenient means of evaluate the perceived responsiveness of an Android application. This involves providing the ability to perform regression testing to track improvements in responsiveness across application versions as well as software and hardware platforms.

The tool is required to target specific code segments in the AUT in order to minimize its memory and processing overhead to avoid introducing inaccuracies to processing duration measurements. Specific instrumentation targets helps the tool overcome the previously discussed drawbacks of Traceview and Systrace, with a lower processing overhead and less amounts of data produced in total. It is also desirable that the tool is easy to configure. Keeping the tool from modifying existing AUT source code ensures that enabling and disabling the ALP instrumentation logic is quick and free from side effects.

The output of the ALP-tool is a HTML report with a summary that highlights poorly performing event listeners while providing sufficiently detailed information on each individual event listener measurement (event listener trace) to let the developers understand the circumstances of the reading aiding them in a decision of an optimization strategy.

4.1 Overview

The proposed approach consist of a three step process (see Figure 4.1), where each step must be completed to enable execution of the next step. The first step involves static analysis and instrumentation of the AUT. The second step is to execute the test scenarios on a real android device while collecting event listener trace data, allowing the third step to construct a human readable report from the collected data.

The tasks performed in each step require different degrees of access to the AUT. The instrumentation step requires complete access to the inner program structure to perform static source code analysis and generation of instrumentation code. The second step only utilize functional information contained in GUI test-cases. The third step utilize the information resulting from the previous steps and have no direct interaction with the AUT. This makes the ALP-tool a hybrid between the white and black-box test approaches.

The individual steps performed by the ALP-tool can be summarized as follows:

Step 1: Analysis and Instrumentation A static analysis of the application source code is conducted in order to locate event listeners and add instrumentation.

Step 2: Test Execution executes test scenarios on a real Android device and collects the data generated by the instrumented AUT.

Step 3: Log Analysis Utilize the data collected in the previous step to perform an analysis to present an evaluation of the AUT in a easy to read HTML report.

4.2 Analysis and Instrumentation Step

In order to evaluate the responsiveness of the AUT, measurements of the execution time of event listeners located in the Activity-classes are performed. The act of instrumenting an event listener involves adding statements to start and stop a stopwatch-timer at the very beginning and end of the corresponding Java-method. The result of the measurement is stored to enable post-analysis in the third step.

For performing analysis and instrumentation of the source code, the Aspectj framework was chosen. The following sections introduce the techniques and terminology required to understand the operations performed by the Aspectj framework. The aim is to provide the insight required to understand how

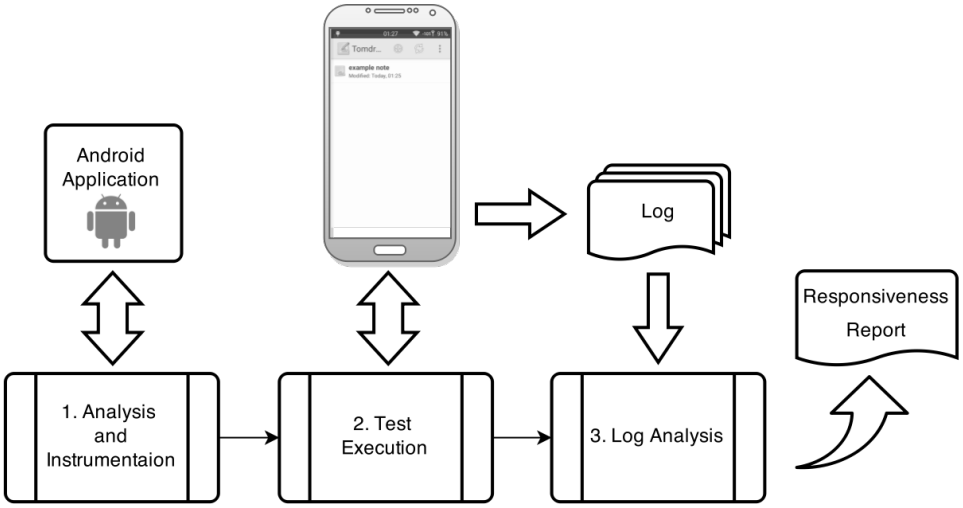


Figure 4.1: Tool design diagram.

it is applied in the implementation of the ALP-tool. The implementation specific details regarding the ALP-tool is described in chapter 5.

4.2.1 Aspect Oriented Programming with Aspectj

Aspect Oriented Programming (AOP) is a programming paradigm constructed to increase modularity by allowing separation of cross-cutting concerns. Aspectj extends the Java programming language with additional syntax for defining AOP constructs. The container for defining AOP operations is the *Aspect-container*. Aspects contains predicates for *join-points* (Java methods) that identify the locations of the cross-cutting concern, as well as the *advice* (additional code) that addresses it. An advice can be executed before, after or around join-points.

Logging is an example of a cross-cutting concern as it applies to every logged part of the system. For developers of the Tomdroid application, it is conceivable that debugging of the note presentation functionality required logging of all methods in the `ViewNote`-class. A possible approach would be to add the logging functionality directly by inserting the required code to the beginning of every method of the class. The approach is laborious and introduce a lot of duplicate code. With Aspectj the same functionality can be implemented without any manual modification to the `ViewNote`-class itself since all logging logic can be specified in a aspect located in a separate file. The aspect in Figure 4.2 define the complete logging mechanism suggested above. The coupling to the `ViewNote`-class is specified by the point-cut predicate on line 2, while the logging logic itself is defined in the before-advice on the lines

3 to 5. The *execution* type point-cut encapsulate the method body of the specified join-point.

```
1 public aspect ViewNoteTrace {
2     pointcut log() : execution(* org.tomdroid.ui.ViewNote.*(..));
3     before(): log() {
4         //Logging code
5         }
6 }
```

Figure 4.2: Aspect for logging of method invocations in the `ViewNote`-class of the Tomdroid application.

4.3 Test Execution Step

The Testrun Manager require a real Android device in order to simulate normal usage of the AUT. When the simulation is running (executing test-cases), the process is monitored through the Event Listener instrumentation added in the previous step. The data is persisted in order to allow post analysis in the third step as described in section 4.4.

A cause for inaccurate timing of Event Listeners is background tasks being performed by the Android operating system concurrently with test-case execution. To avoid this, it is recommended that the Android device is as free from other applications as possible during testing. Another cause for error is failure of ensuring similar application and device state between different test-runs. In order to ensure similar state at the beginning of all test runs, the AUT and the related projects are re-installed on the Android device before every test execution.

Even when care is taken to provide similar device state in between test runs, the non-deterministic nature of the Android task scheduling mechanism will in practice result in some variance in processing durations. To overcome this uncertainty, the test scenarios are executed multiple times in order to provide a statistical basis for the evaluation.

To provide input to the AUT, the Testrun Manager require a pre-configured set of test scenarios. Test scenarios should be as close to normal application usage as possible - both in size and complexity. The test scenario implementation technique itself is transparent to the instrumentation, and the log analysis steps. Test scenarios can be implemented using any test framework or test-API which support construction of user interface test-scenarios that can be executed repeatedly. The need for re-running of the test scenarios rule out test-frameworks that only produce random input scenarios to the

application. An example of such a test tool is the *UI/Application Exerciser Monkey* [15] included with the Android SDK.

4.4 Log Analysis Step

The raw log produced by the instrumented AUT typically contain large amounts of data, making manual analysis difficult and time consuming. The ALP analyser aims to summarize the log produced during the test execution step in order to state the results clearly - in a easy to read manner. The report describe expected duration for each event listener in every test-case by use of descriptive statistics. The information is presented through data tables and interval bar-graphs. Each bar-graph provide the tester with insight to what event listeners was executed and their order for each test scenario.

Not all event listeners show up in the summary. Event listeners are displayed in the summary when the average value plus two Standard Deviations exceed 80 milliseconds. The 80 milliseconds threshold limit has been found to be a practical filter through iterative testing using the case study applications introduced in section 1.3. The 80 milliseconds limit has been derived from the 100 milliseconds GUI responsiveness guidelines [6], giving a conserved estimate for poor responsiveness. If the limit is deemed to be incorrect by the test developers, it can be quickly adjusted with only the third step required to be re-executed.

In addition to evaluating the AUT based solely on timings of individual event listeners, the ALP-tool also detects method calls to the *startActivity(Intent)*-method (and similar methods) of the `Activity`-class, registering *activity transitions*. Activity transitions occur when the current activity (on top of the activity-stack) is being replaced by another activity. It is possible to determine how long time has elapsed from an input event trigger the transition to the new activity has finished loading and is visible to the user. To achieve this it is necessary to also instrument the life cycle methods of the `Activity`-class. If the `onStart()` method is implemented in the target activity it is the last of the life cycle methods called before the activity is displayed to the user. If `onStart()` is not implemented, the `onResume()` or `onCreate()`-methods are used in its place. It is difficult to set a limit on how long processing duration is acceptable for activity transitions. While the sub 100 milliseconds range is ideal here as well, it might also be acceptable with three of four times longer durations due to transition animations. What is acceptable depends on where it is located in the application and if it is part of a regular usage pattern. If the transition targets a preferences menu that typically is only used on rare occasions, longer delays might be acceptable. As such, all activity transitions are shown in the log without any filtering.

Chapter 5

Implementation of ALP

The ALP prototype has been implemented as two stand-alone Java applications. The *Instrumentation Generator* implements the *analysis and instrumentation* step. The second application is called the *Testrun Manager* and implement the functionality corresponding to the *test execution* and *log analysis* steps. When applying the tool, the applications must be executed in sequence starting with the Instrumentation Generator. The applications can be utilized with Android application project set-ups created by Android Studio and Eclipse ADT.

5.1 Instrumentation Generator

To evaluate application responsiveness, the source code of the AUT needs to be instrumented with statements to control stopwatch functionality at the beginning and end of each event listener method. The statements themselves make use of Androids built-in logging system to transmit messages and make them available through Logcat. The log entries contain a number of JSON encoded data fields with information such as timestamp class and method-names. See Figure 5.1 for a complete specification of the parameters included in the log-entries.

The timestamps as well as the processing durations are obtained by use of the `System.currentTimeMillis()`-method. This means that the time measured is the wall-clock execution time of the event listener. This provides the desired type of processing duration timing as it most closely represent actual system response time the way it is experienced by the user. As this introduce some variance in the processing durations caused by interference from other applications executing simultaneously, the tests are executed several times to obtain a statistical basis for the evaluation.

Parameter Name	Description
class	Name of class inhabited by the event listener.
method	Name of event listener method.
duration	Event listener processing duration in milliseconds.
start	Processing start timestamp.
end	Processing completion timestamp.
arguments	Event listener arguments list. Not present for empty parameter sets.
parameterType	Parameter type.
description	Argument description.
returnType	Event listener return type. Not present for void signatures.
returnValue	Description of the event listener return value. Not present for void signatures.
cat	Cross Activity Trace - Present for event listeners that open activities. Contains the fully qualified Java class name of the target activity.

Figure 5.1: Parameters logged for each event listener trace.

In Figure 5.2 the event listener `onCreateOptionsMenu()` of the `Tomdroid`-class has been measured. It can be seen that the log message itself is of type debug with the tag *ELT*. In the message field of the log entry, the payload data is encoded using JSON.

```
D/ELT     ( 4807): {
"duration":1,
"arguments":{"MenuBuilder":"com.android.internal.view.menu.MenuBuilder@41f05730"},
"start":1430287416543,
"returnValue":true,
"package":"org.tomdroid.ui",
"class":"ViewNote",
"method":"onCreateOptionsMenu",
"returnType":"Boolean",
"end":1430287416544
}
```

Figure 5.2: Logcat output from Aspectj instrumentation: Menu item selected in the *Tomdroid*-Activity.

5.1.1 Instrumenting Application Source Code

The ALP-tool utilize compile time *weaving* of the Aspectj advice code to perform AUT instrumentation. This has the advantage that only a minimal amount of processing is performed at runtime. The Aspectj runtime library is

used by the advise-code to gather join-point information and must therefore be imported to the AUT project. All sets of point-cuts and advice bodies are contained in a single Aspect-container called `Trace`.

The instrumentation runtime libraries and the `Trace`-aspect is added to the AUT automatically by the ALP Instrumentation Generator. The component require an initial configuration as described in chapter 6 on Configuration and Usage. After execution of the Instrumentation Generator, the added source files can be found in the directory path corresponding to the Java package `no.hib.alp.instrumentation` in the source folder of the AUT.

In Android applications, event listeners can be assigned using various techniques, as described in chapter 2 on Event Listeners. In order to instrument all kinds of event listeners, the ALP-tool utilize three separate sets of point-cuts and advice bodies. Two of them are always present in instrumented applications, while the third one is only present when the application utilize XML assigned event listeners.

The `eLTrace` point-cut address the concern for instrumentation of programmatically assigned event listeners while the `xmlELTrace` point-cut instrument event listeners assigned using XML Layout files. `XMLELTrace` is only required to be present when XML assigned event listeners are used in the AUT. The logging logic in the advice code is the same for both the `xmlELTrace` and the `elTrace` point-cuts and is therefore described in a separate section, and only outlined in the sections describing the point-cuts.

The `crossActivityTrace`-point cut has an advice body differing from the other two, as it does not target event listeners but is used to track activity transitions. When a join-point targeted by the `crossActivityTrace`-point-cut is executed within an event listener trace its operation is reflected in the trace log output.

The Trace-aspect contain a static boolean variable which is used by all three point-cuts to determine if a trace is currently being performed. The event listener trace point-cuts use it to avoid performing overlapping traces, as this could lead to faulty readings if an event listener trace advice was processed during a trace. The `crossActivityTrace` point-cut also make use of the variable to make sure that the `startActivity`-method calls are only noted during an event listener trace. The variable is set to false by default and managed by the event listener trace advice code during test execution.

Figure 5.3 summarize the three different aspects used by the ALP-tool. The following sections provide detailed descriptions of the point-cuts and the advice bodies.

Concern	Identifier	Description
EL Tracing	elTrace	Targets programmatically assigned event listeners.
EL Tracing	xmlELTrace	Targets event listeners assigned using XML layout files.
Activity Transition Tracing	crossActivityTrace	Targets *startActivity()*-method calls in order to track activity transitions.

Figure 5.3: The different concerns addressed in the Trace-aspect used by the ALP-tool.

Advice Code of Event Listener Trace Concerns

The *around*-type advice is used to encapsulate the targeted code with advice both ahead of and after the event listener processing. This is specified by use of the special method call `proceed()`, as seen in Figure 5.4 line 11. The code within the finally clause, starting at line 13, is executed after the event listener method has completed its processing. Line 2 shows the static boolean variable which is used to determine if a trace is currently running.

The objects `thisJoinPoint` and `thisJoinPointStaticPart` is part of the Aspectj runtime library. They provide information such as method signature, class name, and references to method arguments about the currently advised join-point.

The logging itself and the JSON data conversion logic is factored to the static method `writeToLogcat()`-located in the `TraceLogger`-class. This allows for reuse of the logging logic in both aspects.

In Figure 5.5 the `TraceLogger`-class is outlined, illustrating the process of constructing the JSON formatted log entries. The `writeToLogcat()`-method is called at the end of the event listener trace and has all the information available to complete the logging. The line 11 to 12 adds information regarding activity transitions. As the string variable `catTarget`, declared on line 2, is modified by the `crossActivityTrace`-advice code, it will no longer reference null and its value is included in the log message.

```
private static boolean traceIsRunning = false;

Object around(): .. {
   traceIsRunning = true;
   long start = System.currentTimeMillis();
   Object returnValue = null;
   try {
      returnValue = proceed();
      return returnValue;
   } finally {
      long end = System.currentTimeMillis();
      TraceLogger.writeToLogcat(thisJoinPoint.getArgs(),
         thisJoinPointStaticPart.getSignature(),
         returnValue, start, end);
      traceIsRunning = false;
   }
}
```

Figure 5.4: The advice-body used by both the `elTrace`, and the `xmlELTrace`-point-cuts.

```
public class TraceLogger {
   public static String catTarget = null;
   public static void writeToLogcat(..){
      JSONObject logData = new JSONObject();
      JSONObject arguments = new JSONObject();

      logData.put("method", s.getName());
      logData.put("duration", (end - start));
      .. // Set JSON parameters
      if(catTarget != null){
         logData.put("cat", catTarget);
         catTarget = null;
      }
      Log.d("ELT", logData.toString());
   }
}
```

Figure 5.5: Outline of the *TraceLogger*-class illustrating cross activity tracing.

Event Listeners Assigned Programmatically

Figure 5.6 shows the definition of the the `elTrace` point-cut and an associated *around*-advice for instrumentation of programmatically assigned event listeners. The *elTrace* point-cut is composed of the *execution*, *within* and *if* type of point-cuts.

The *execution*-type point-cut encapsulate the target code allowing capture of all event listener calls. This includes calls from sources outside the AUT such as the Android system and other applications. The method-pattern `* *.on*(..)` targets *on*-prefixed methods, regardless of method signature, class, or package throughout the AUT. Note that the method-pattern includes not only event listeners, but also the on-prefixed life cycle methods of the Activity class. This allows tracing of Activity transitions, as described in section 5.1.1 on the `crossActivityTrace`-point-cut and advice body.

The *within* point-cut is used to exclude classes that do not extend `android.app.Activity` or a sub-type. This excludes *on*-methods found in other application components such as Services and Broadcast Receivers.

The *if*-type point-cut is a runtime check which is utilized to avoid overlapping traces and make sure that all processing within traces belong to the AUT and is not caused by the ALP-tool.

```
1 public aspect Trace {
2    private static boolean traceIsRunning = false;
3    pointcut eLTrace() : execution(* *.on*(..))
4       && within(android.app.Activity+)
5       && if(!traceIsRunning);
6    Object around(): eLTrace() {
7    ..
8    }
9 }
```

Figure 5.6: Aspect providing instrumentation of event listeners and life-cycle methods of Activity classes.

Event Listeners Assigned using XML Layout-Files

Event listeners assigned using XML mark-up in layout files can have any method name desired by the application developers and may not follow the convention of *on*-prefixes. The only requirement for XML assigned event lis-

teners is the method signature `public void methodName(View)`. The chosen approach for detecting such event listeners is to analyse the XML files.

In XML layout files, the method name of an event listener is specified in the `android:onClick`-parameter of the parent view element. This means that the method name can be fetched directly. The class name of the activity must also be discovered. In Android applications this does not need to be specified within the XML-layout files, as the layout of the activity is declared programmatically within the Activity classes by use of the `setContentView(View)`-method. In Figure 5.7, the `tool:context` parameter (line 4) points to the activity class that make use of the XML-layout file. The parameter is used by IDE tools to determine the styling theme used by the Activity to allow for generation of accurate preview images. The Instrumentation Generator utilize this value to determine the class that contain the event listener method.

```
1  <RelativeLayout
2          xmlns:android="http://schemas.android.com/apk/res/android"
3          xmlns:tools="http://schemas.android.com/tools"
4          tools:context="no.hib.alp.delaytester.MainActivity" >
5          <Button
6                  android:id="@+id/button5"
7                  android:layout_width="wrap_content"
8                  android:layout_height="wrap_content"
9                  android:layout_alignParentLeft="true"
10                 android:layout_alignParentTop="true"
11                 android:onClick="lowVarianceDelay"
12                 android:text="Low Variance"  />
13 </RelativeLayout>
```

Figure 5.7: A layout file in the DelayTester Application assigning a single Event Listener in the `MainActivity`-class.

The Instrumentation Generator makes a pass of the `res`-folder of the AUT to locate `.xml`-files with view-hierarchies. When a view element with an attached `android:onClick`-parameter is discovered, a dedicated execution-type point-cut is added to the `xmlELTrace`-pointcut through composition. This can be seen in Figure 5.8, where the `xmlELTrace`-pointcut is responsible for targeting two event listeners. The first one is the `goBack()`-method of `no.hib.alp.delaytester.DelayResultActivity`. The second one is the `lowVarianceDelay` that was declared in Figure 5.7.

```
1 private static boolean traceIsRunning = false;
2 pointcut xmlELTrace() :
3 (execution(* no.hib.alp.delaytester.DelayResultActivity.goBack(..))
4  || execution(* no.hib.alp.delaytester.MainActivity.lowVarianceDelay(..)))
5 && if(!traceIsRunning));
6
7 Object around(): xmlLTrace() {
8   ..
9 }
```

Figure 5.8: Aspect providing instrumentation of XML-assigned event listeners.

Cross Activity Tracing

Figure 5.9 shows the definition of the pointcut and advice which is addressing the activity transition concern. The pointcut is composed of a *call* and an *if* point-cut. The if-point-cut is a runtime check that make sure that the advice body is only executed when an event listener trace is being performed. The call-type point cut encapsulate every method-call that matches the method-pattern. The method pattern, `* android.app.Activity.startActivity*(..)`, matches every method in the `Activity`-class who's name starts with *startActivity*, regardless of return values or parameters. This includes the methods `startActivityForResult()`, `startActivityForResultAsUser()` and `startActivityForResultAsCaller()`. The *startActivity*-methods can be used with both *explicit* or *implicit* intents. Implicit intents are used to let the Android system determine the correct activity to open, while the explicit type specify the target activity. The ALP-tool is only concerned with explicit Intents which target activity-classes within the AUT. Line 8 to 14 fetch the class name of the target activity from the intent object and stores it in the static `catTarget` string object of the `TraceLogger`-class. When the currently traced event listener has finished processing and the log entry is being constructed by the event listener trace advice, the target activity class identifier is included before the `catTarget`-String reference is reset to null.

```
1   pointcut crossActivityTrace():
2      call(* android.app.Activity.startActivity*(..))
3      && if(traceIsRunning);
4
5   before(): crossActivityTrace() {
6      Object[] args = thisJoinPoint.getArgs();
7      Intent I = null;
8      for (Object arg: args){
9         if (arg instanceof Intent) {
10           I = (Intent) arg;
11           TraceLogger.catTarget = I.getComponent()
12              .getClassName();
13           break;
14        }
15     }
16  };
```

Figure 5.9: The point-cut and advice definition addressing the activity transition concern.

Event-Listeners with parameters

Some event listeners has parameters that dictate their operation and lead to different execution times. Some of these event listener methods are listed in Figure 5.10. When developers review the responsiveness evaluation report, information regarding the circumstances of the traces can be helpful to optimize the AUT. The report includes a description of the arguments and return variable for each event listener trace. The argument descriptions originate from the log output produced during test execution on the instrumented application.

The argument descriptions are created in the `TraceLogger`-class of the instrumented AUT and originate from the `VariableDescriptionService`-class. The description is output alongside the other trace information in the event listener argument list for the method parameters and in the `returnValue`-parameter for the return value. Both the arguments list and the return parameters are only present in the log output for relevant event listeners.

Arguments of different types must be handled differently in order to provide meaningful descriptions. Java primitives are automatically wrapped through auto-boxing and can therefore be described by use of their `toString()`-method. Other argument types typically require a more specialized approach to provide meaningful information.

```
1 onOptionsItemSelected(MenuItem item)
2 onMenuItemSelected(int featureId, MenuItem item)
3 onMenuOpened(int featureId, Menu menu)
4 onKeyUp(int keyCode, KeyEvent event)
5 onKeyMultiple(int keyCode, int repeatCount, KeyEvent event)
6 onPrepareNavigateUpTaskStack(TaskStackBuilder builder)
7 onPrepareOptionsMenu(Menu menu)
8 onPreparePanel(int featureId, View view, Menu menu)
9 onNavigateUpFromChild(Activity child)
```

Figure 5.10: Common event listeners with contextual parameters.

As Android event listeners has a wide range of parameter types and can be used differently from application to application, the description generation system is required to be expandable. To achieve this, the concept of *type-describers* was created. Figure 5.11 lists the variable types for the type-describers supplied with ALP-tool.

Argument Type	**Description Content**
android.view.MenuItem	The item ID as provided by the `getItemId()`-method.
android.view.KeyEvent	The value `getCharacters()`-method.
android.view.Menu	The number of menu items as provided by the `size()`-method.
android.view.View	The ID as provided by the `getId()`-method.
com.android.internal.view.menu.ActionMenuItem	The item ID as provided by the `getItemId()`-method and its title from the `getTitle()`-method.

Figure 5.11: The type-describers included with ALP.

Use of specialized variable describers, *type-describers*, is achieved through use of the Java Reflection API as seen in Figure 5.12. A type-describer is class that implements the `VariableDescriber`-interface. The interface specifies a single method with the signature `public String getDescriptor(Object)`-method. The name of the type-describer must follow a precise naming scheme in order to be recognized by the ALP-tool. The name must correspond to the fully qualified class name of the variable type it describes with the navigation marks removed and the first letter of class and package names capitalized in addition to a *"Describer"* post-fix. As an example, the `android.view.View`-describer is named `AndroidViewViewDescriber`. The implementation of this is outlined in Figure 5.12. In the lines 1 to 7, the class

name of the argument is determined for use in line 8 to 10 where the Java Reflection API instantiate the type-describer object, allowing the description to be produced at line 11. This allows application testers to customize the information provided for each variable type to suit their particular needs. How to implement a type-describer is described further in chapter 6 on *Configuration and Usage*.

```
1  String argumentType = o.getClass().getName();
2  StringBuilder argTypeC = new StringBuilder();
3  String[] argTypes = argumentType.split("\\.");
4  for (String s : argTypes) {
5     argTypeC.append(Character.toUpperCase(s.charAt(0)))
6        .append(s.substring(1));
7  }
8  Class<?> d = Class.forName("no.hib.alp.instrumentation."
9   + argTypeC.toString() + "Describer");
10 ArgumentDescriber describer = (ArgumentDescriber) d.newInstance();
11 return describer.getDescriptor(o);
```

Figure 5.12: The *VariableDescriptionService*-class make use of the Java Reflection API to instantiate type-describers.

If an argument type without a specialized type-describer is encountered during a test-run, the default action is to use the `toString()`-method to provide a description. If the utilized `toString()`-method is the default of Java *Object*, the description will contain a hash code. This code will be different in every single trace, even for objects that are functionally similar when utilized by the event listener. This can make the resulting evaluation report harder to read and the test developers should consider implementing a type-describer for the variable type.

5.2 Testrun Manager

The *Testrun Manager* application manages execution of test-scenarios, configuration of the physical Android device, as well as parsing and storage of event listener traces. The event listener traces are sent from the instrumented AUT through the Android logging system and fetched by the Logcat utility through the Android Debug Bridge. When the trace log messages are fetched and parsed they are persisted using an embedded *Apache Derby* relational database.

Figure 5.13 shows the control flow of the Testrun Manager. The first execution step is loading the settings from the configuration file and perform a validity check. The validity check include availability of the apk-files, the Android SDK tools, the Android device, and test classes. If the configuration is valid, test execution can commence. Each test-execution cycle consist of a re-installation of the AUT and the associated test project, execution of all test cases of every test class, and persisting trace data from the device. When the number of test execution cycles (as specified in the configuration file) has completed, the report is generated and the application terminate.

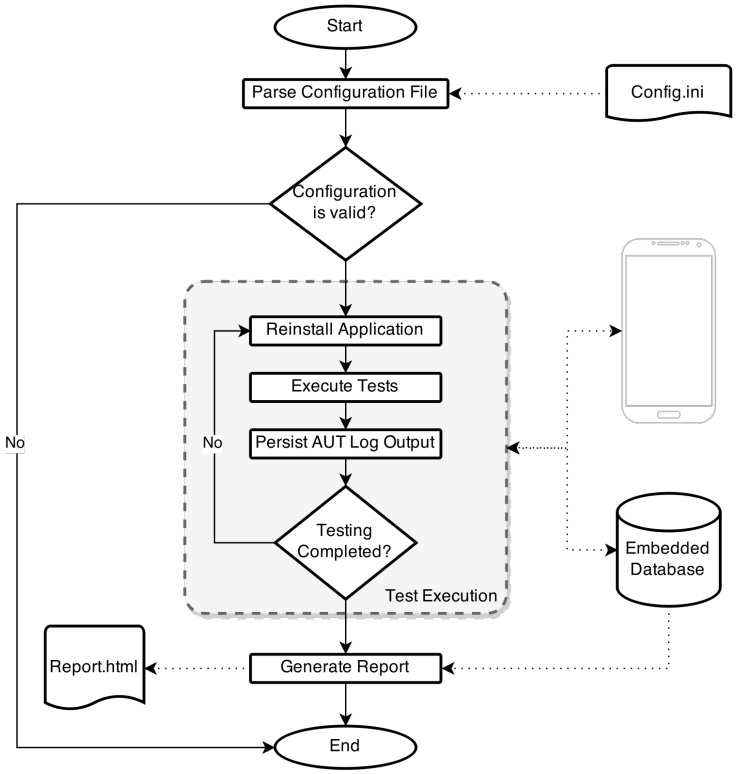


Figure 5.13: Testrun Manager control flow.

During the test execution cycles, the Testrun Manager utilize a set of command-line tools to send instructions to the Android device. All command-line communication with the android device is performed through the Android Debug Bridge. The relevant commands are listed in Figure 5.14.

Command-line Arguments	Description
`adb shell am instrument -w -e <className> <methodName> <testPackage> /android.test.InstrumentationTestRunner`	Run test-case in test-suite.
`adb uninstall <package>`	Uninstall Android application from device.
`adb -d install <pathToBin>.apk`	Install Android application on the device.
`adb logcat -c`	Clear the main log buffer. Used to avoid duplicate entries.
`adb logcat -d -s ELT`	Get the content of the main log buffer. The *s*-flag specifies filtering of entries with the tag *ELT*.
`adb device`	Get a list of devices currently connected to the adb server.

Figure 5.14: Various commands used by the Testrun Manager.

As seen in Figure 5.15, the Android Debug bridge commands are executed by use of the `java.lang.ProcessBuilder`-class. The code segment empty the log buffer on the default device by use of the command `adb logcat -c` (line 1). After the process is started (line 2), the output streams, standard output and standard error of the process is established in line 4 to 7. The while-loops in the lines 10 to 15 parse and interpret the output of the Android Debug Bridge client process. When the process has terminated its output streams are closed and the process can be destroyed (line 16) and the input streams closed (line 17 to 18).

5.3 Analysing Event Listener Traces

The HTML report is sectioned into three parts; *Summary Cross Activity Traces*, and a *Dataset*-section. All three parts make use of descriptive statistics to provide quick insight into the data of the associated dataset. The Summary section lists event listeners that are suspected of performing poorly while the Cross Activity Traces section lists every registered activity transition. The listings in the Dataset section contain the complete set of readings of all event listeners with all details included.

```
1    ProcessBuilder pb = new ProcessBuilder(ADB, LOGCAT, C_FLAG);
2    Process process = pb.start();
3
4    BufferedReader stdInputBufferedReader = new BufferedReader(
5      new InputStreamReader(process.getInputStream()));
6    BufferedReader errInputBufferedReader = new BufferedReader(
7      new InputStreamReader(process.getErrorStream()));
8
9    String line = "";
10   while ((line = stdInputBufferedReader.readLine()) != null) {
11      ..
12   }
13   while ((line = errInputBufferedReader.readLine()) != null) {
14      ..
15   }
16   process.destroy();
17   stdInputBufferedReader.close();
18   errInputBufferedReader.close();
```

Figure 5.15: Android device control by use of Android Debug Bridge through the `java.lang.ProcessBuilder`.

5.3.1 Summary Section

The summary section lists the event listeners which have passed through the filter, as explained in chapter 4. The event listeners are ordered by test-case as seen in Figure 5.16. Each event listener trace is described by average, median, Standard Deviation, and minimum and maximum value.

In addition to descriptive statistics, test cases that lead to discovery of poorly performing event listeners are visualized by use of range bar-graphs, as seen in Figure 5.17. The graph visualize the first execution of the test scenario that included one or multiple poorly performing event listeners. Due to the potentially huge differences in processing duration of event listeners, the readings in the visualization does not necessarily present slow readings, but still aid the tester with an understanding of the circumstance of the readings. This can be seen in Figure 5.17, where the event listener `onOptionsItemSelected(..)` of the `Tomdroid`-class has an average processing time of 81 milliseconds, while in the visualized test iteration, the duration was only 58 milliseconds. The processing duration of an event listener drawn in a graph can be seen on the right side of its method signature.

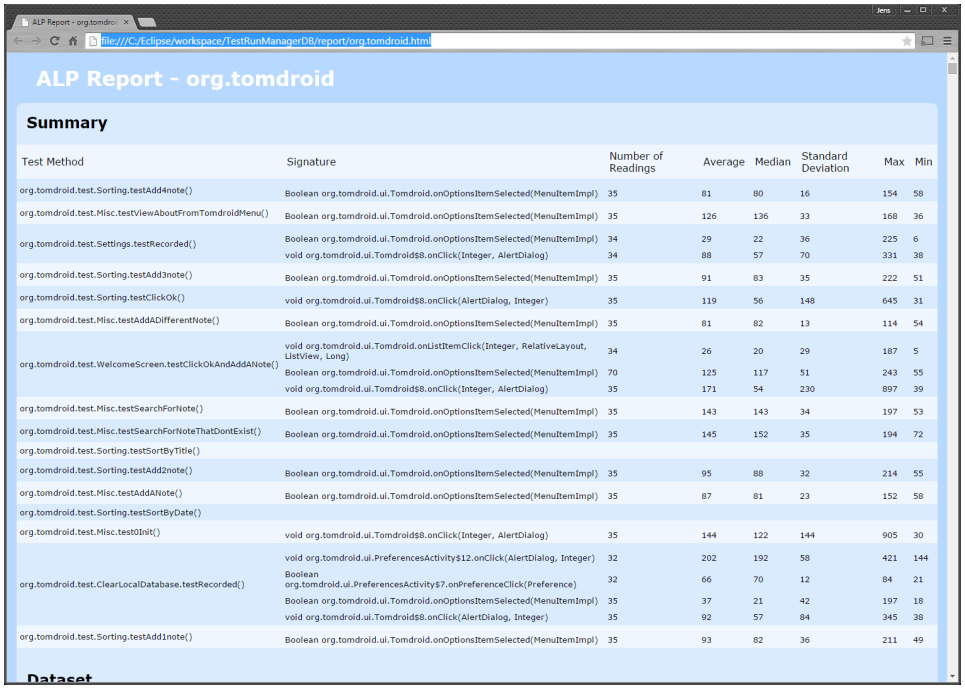


Figure 5.16: Summary section by example from the Tomdroid application testing.

5.3.2 Cross Activity Traces

The Cross Activity Traces section lists all the intra-application Activity transitions that was detected during the test execution cycles. The section, as can bee seen in Figure 5.18, is located below the Summary, and above the Dataset sections. Each transition is described with the columns *Test Method*, *Event Listener*, *Target Activity* in addition to descriptive statistics that summarize the set of readings on that specific transition.

5.3.3 Dataset Section

The dataset section contains a detailed view of all the data collected during test execution. The section is grouped by test-case and event listener. Each event listener is described with descriptive statistics, processing duration and variable descriptions. The leftmost column, *Test Run Identifier*, display which test run the reading originate from. Figure 5.19 shows part of the data collected on the `onOptionsItemSelected()`-event listener during the `Sorting.testAdd4Note()`-test case.

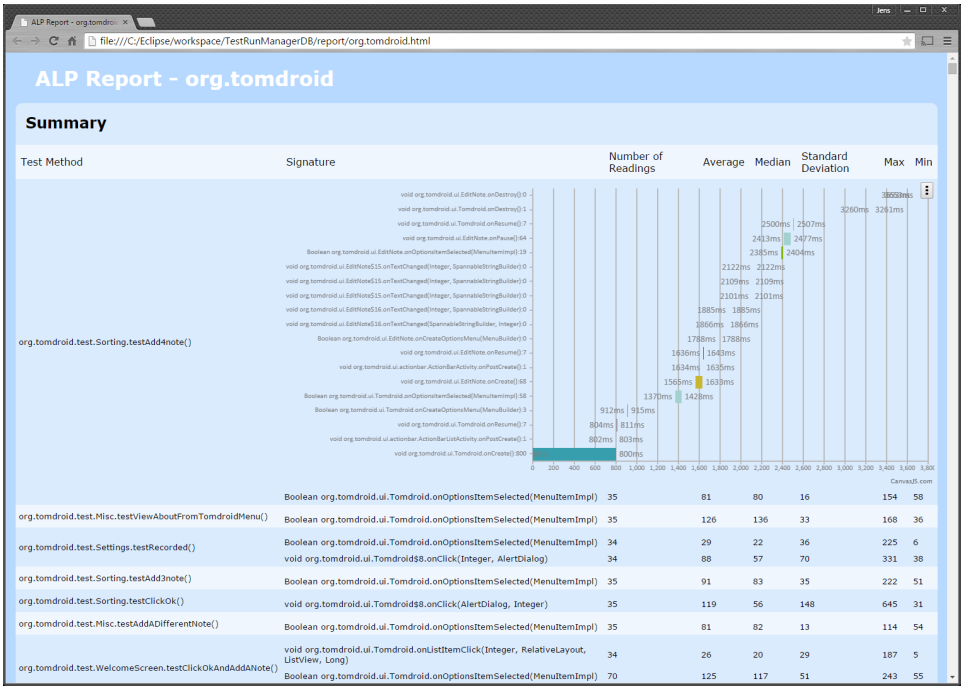


Figure 5.17: Test Scenario visualization using bar-graphs.

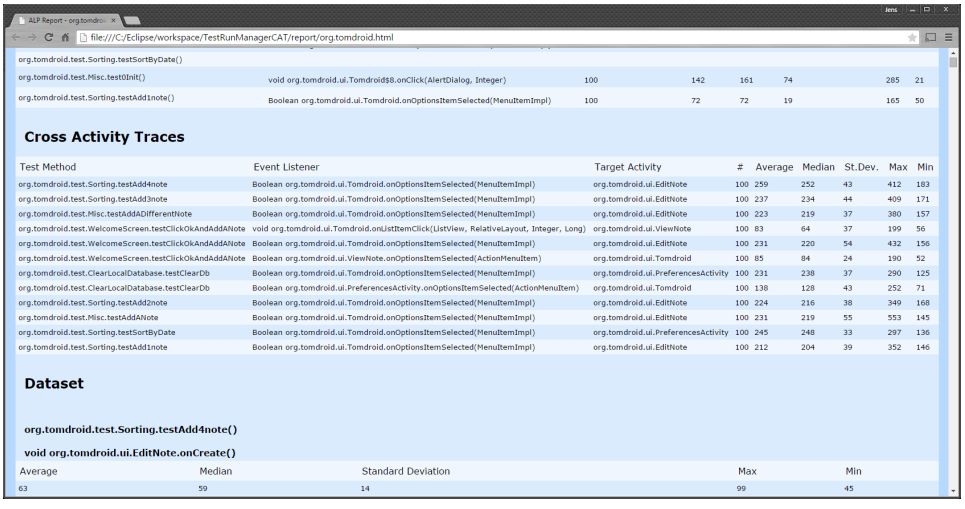


Figure 5.18: The Cross Activity Traces section by example from Tomdroid.

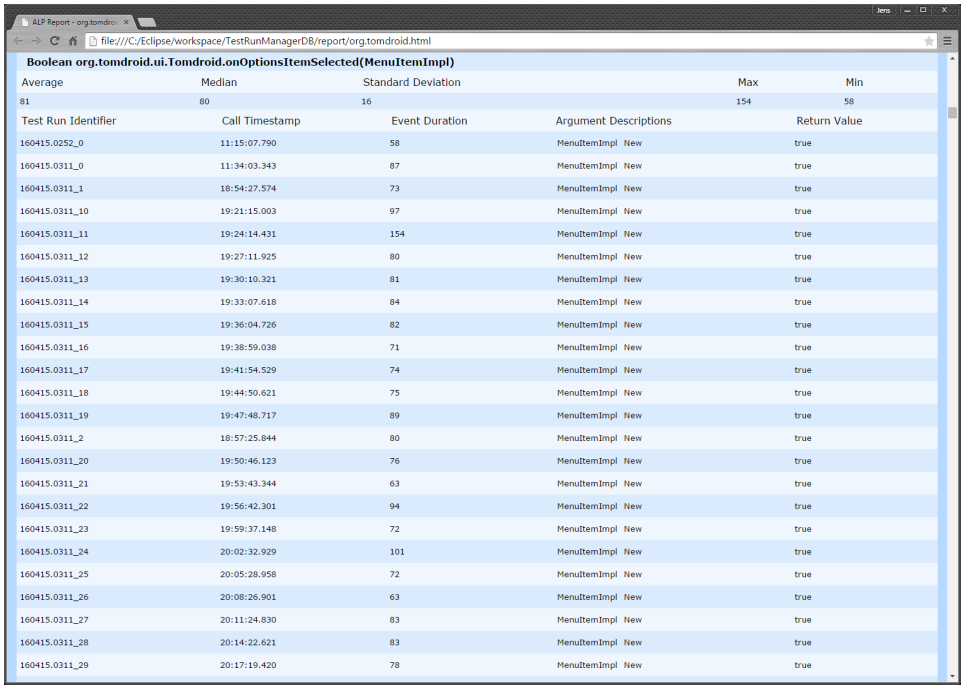


Figure 5.19: Dataset section about the `onOptionsItemSelected(..)`-event listener of the `Tomdroid`-class.

Chapter 6

Configuration and Usage of the ALP-Tool

The ALP-tool can be used with projects created by the Eclipse and the Android Studio IDE. The following chapter provide the information required to use the ALP-tool with both IDEs. The ALP-tool can be downloaded from [22]. A video demonstration is also available at [23], showing ALP configuration, instrumentation of the AUT as well as test execution.

6.1 Instrumenting the AUT

The instrumentation process involves insertion of the logging logic into the source folders of the AUT, before building the application with the Aspectj compiler included in the build process. The aspect files and the associated Java source files are generated using the *Instrumenation Generator* component of the ALP -tool. Before executing the Instrumentation Generator, the *AUTDirectory*-parameter must be set in the *conf.ini* configuration file located in the root folder of the application. The parameter must point to the root project directory of the AUT for Eclipse projects or the application module folder for Android Studio projects. Figure 6.1 shows the configuration file used during testing of the Tomdroid application. The *buildSystem*-parameter is used to determine the folder structure of the AUT. For Android applications using Android Studio and Gradle, the parameter value should be set to *gradle*.

```
[setup]
buildSystem = Eclipse-ADT
autDirectory = C:\Eclipse\workspace\tomdroid
```

Figure 6.1: Instrumentation Generator configuration file by example from the Tomdroid application testing.

6.1.1 Definition and Customization of Type-Describers

Type-describers are used by the ALP-tool to describe the parameters and return values of event listeners. The descriptions can be seen in the responsiveness evaluation report. Having accurate descriptions of the arguments and the return values allows the developers to better understand the execution path that led to long duration processing. Due to the huge amount of parameters and return types of Android event listeners, not to mention the many different aspects of each parameter type that might be of interest to the application developers, the system has been made expandable by use of type-describers.

A type-describer is class that implement the `VariableDescriber`-interface. The interface specify a single method, the `public String getDescriptor (Object)`-method. The name of the VariableDescriber implementations must follow a naming scheme in order to be recognized by the ALP-tool. The name must be the fully qualified class name of the variable type it describes with the navigation marks removed and follow the Java class naming convention of capitalizing the first character in each word, in addition to a *Describer* post-fix. As an example, the `android.view.View`-describer is named `AndroidViewViewDescriber`.

The VariableDescriber implementation (the *type-describer*) is used by the run-time library of the Android application instrumentation. The Java file must be copied to the `resources/instrumentation`-folder of the Instrumentation Generator application before it is executed.

Figure 6.2 shows a complete implementation of a type-describer. Line 7 to 10 implement the *getDescriptor*-method used by the ALP-tool to fetch the description used in the responsiveness evaluation report.

```java
package no.hib.alp.instrumentation;

import android.view.MenuItem;

public class AndroidViewMenuItemDescriber
      implements VariableDescriber {
   @Override
   public String getDescriptor(Object o) {
       return String.valueOf(((MenuItem) o).getItemId());
   }
}
```

Figure 6.2: A type-describer for the MenuItem-class.

6.1.2 Building the AUT with Aspectj Weaving

The process of building the AUT with the instrumentation depends on which build system is being used. Aspectj Weaving can be integrated into projects created using both the Eclipse and the Android Studio IDEs.

If the AUT uses Eclipse project set-up and build system the *Aspectj Development Tools* (AJDT) plugin can be installed to Eclipse by following the instructions available at [14]. By use of AJDT in Eclipse, the project can be converted to an Aspectj project by right clicking the project in the *Package Explorer* and selecting *Convert to AspectJ Project* from the *Configure* section of the context menu, as seen in Figure 6.3. This will import the required Aspectj Runtime Library and replace the `org.eclipse.jdt.core.javabuilder` with the `org.eclipse.ajdt.core.ajbuilder` as well as insert the `org.eclipse.ajdt.ajnature` in the `.project` file of the project. The project can now be built as normal and the aspect weaving will be performed automatically. Instrumentation can be disabled by right clicking the project and selecting *Remove AspectJ Compatibility* from the Aspectj section of the context menu and deleting the the instrumentation source files from the project.

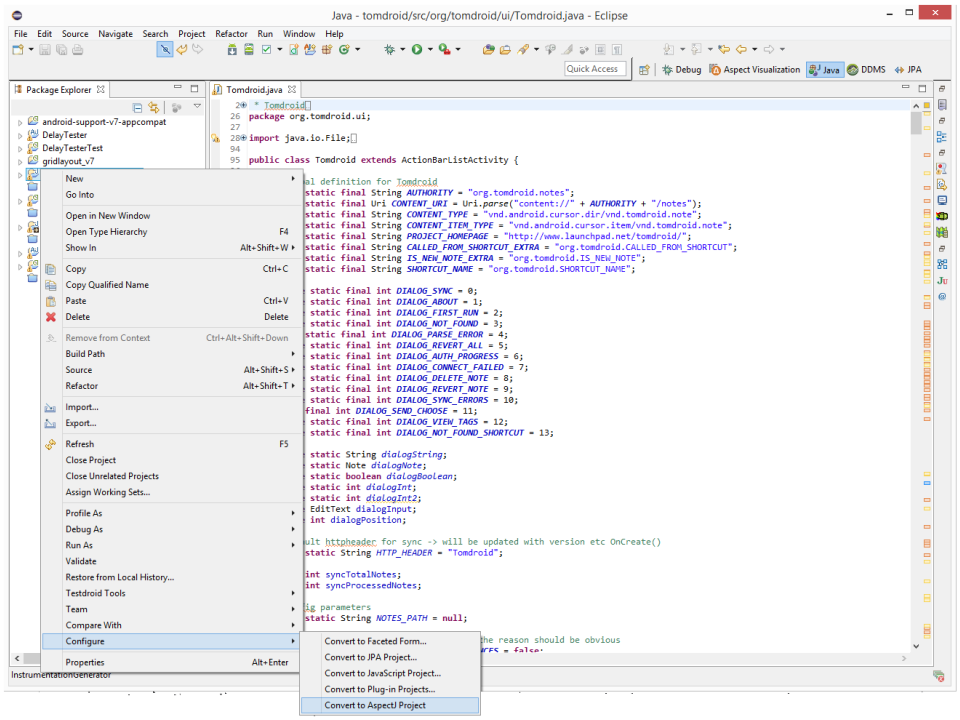


Figure 6.3: Enabling Aspectj using the Eclipse IDE with The Aspectj Development Tools.

For applications using the Gradle Build system as is the case with Android Studio, the Aspectj Gradle plugin available at [13] can be used. The plugin is added by inserting a classpath statement to the build scripts dependencies section as shown in Figure 6.4 line 1. The plugin is enabled in the respective modules by a statement in the Gradle Module file as seen in Figure 6.4 line 2. With the plug-in installed, the AUT can be built as normal with aspect weaving enabled. To disable the instrumentation, the additions to the build scripts can be removed and the instrumentation source files deleted from the project without any side effects.

```
1 classpath 'com.uphyca.gradle:gradle-android-aspectj-plugin:0.9.+'
2 apply plugin: 'android-aspectj'
```

Figure 6.4: Gradle Android Aspectj configuration.

6.2 Testrun Manager Configurations

The Testrun Manager is configured through the configurations file `conf.ini` located in the root folder. The configuration file specifies all the required information to test the Android application. Figure 6.5 show the configuration file used during testing of the Tomdroid application. The configuration files is divided into at least three clauses with the *misc*, *aut* and the required amount of *testClass* clauses. All parameters except the *device*-parameter is required. If the *device*-parameter is not present, the default device will be used.

The test classes are added with *testClass*-clauses, one for each test class. When adding multiple test classes the clauses must be named uniquely and be prefixed with *testClass*. The *tests*-parameter specifies the test-cases that should be executed, while the *package*-parameter specifies the Java package name of the test class. Multiple *tests*-parameters can be used to enhance readability of the configuration listing. Note that the order of the *testClass* and the *tests*-parameter listings represent the test execution order.

The *aut*-clause contain information regarding the application under test. The *autPackage* and the *testPackage* parameters are the package names as seen in the AUT and test project AndroidManifest.xml file.

```
[misc]
testCycles = 5
summaryELThreshold = 80
;device = <device ID>

[aut]
autPackage = org.tomdroid
testPackage = org.tomdroid.test
autApkPath = C:\workspace\tomdroid\bin\tomdroid.apk
testApkPath = C:\workspace\tomdroidTest\bin\tomdroidTest.apk

[testClass]
package = org.tomdroid.test
class = TestCase1
tests = testInit,testAddANote,testSearchForNote
tests = testAddADifferentNote,testSearchForNoteThatDontExist
tests = testViewAboutFromTomdroidMenu
```

Figure 6.5: Testrun Manager configuration by example from the Tomdroid application testing.

In addition to the steps described above, the *platform-tools* folder of the Android SDK must be added to the environment path variable. This will allow the Testrun Manager to locate the Android Debug Bridge.

6.3 Reading the Report

The report is generated automatically after the Testrun Manager has completed the configured amount of test execution cycles. The report itself consist of a regular HTML-file with an attached css-stylesheet and a few javascript files. As such, the report can be viewed using any modern web browser. It is located in the `report` folder of the Testrun Manager. The name of the HTML file correspond to the package name of the AUT.

Chapter 7

Case Studies and Evaluation of ALP

In order to evaluate the effectiveness of the approach, a number of open source applications were selected for testing. The applications themselves were already introduced in section 1.3. Each of the test applications has its own section with a description the test scenarios and a discussion of the findings. A summary of the application testing can be found in section 7.6. Section 7.7 evaluate processing overhead introduced into the AUT by the ALP-tool and the Android test framework.

All performance results presented in the following sections were obtained using a Samsung Galaxy S2 (Model GT-I9100) smart-phone running the Cyanogen Mod 11 operating system based on Android Version 4.4.4. In order to obtain as accurate readings as possible, the CPU frequency of the smart-phone was locked at its highest possible setting of 1200 Mhz as this reduced the processing duration variance significantly.

7.1 Delay Tester

The Delay Tester application was specifically constructed for use during development of the ALP-tool and contains event listeners assigned using both XML and Java code. the application is also helpful when investigating the relationship of how different application behaviours are perceived by the user and reflected in the evaluation report. In order to simulate different types of processing, artificial delays was added to the event listeners using the `Thread.sleep()`-method. Random values are generated using `Math.random()`. The delays are added in each event listener before the `startActivity()`-method is called to transition to the `ResultActivity`-

activity. The intention of the testing is to demonstrate the tools ability to filter and highlight poorly performing event listeners. The resulting user experience as well as the expected behaviour is for each delay type described below.

Slow performance by applying a constant delay of 90 milliseconds. The event listener should be listed in summary section. The delay produce noticeably slower system response than the *Good*-performance delay.

Fast event listener processing duration using a constant delay of 10 milliseconds. The event listener should not be listed in the summary section. The delay is not noticeable and the system response seems immediate.

Random duration delay in the range 0 to 200 milliseconds. The event listener has an expected duration of 100 milliseconds and as such is slow most of the time and should be listed in summary section. The system response is occasionally slow.

High Variance delay with a 50 milliseconds constant in addition to a random value with a maximum of 120 milliseconds. The event listener is expected to be too slow most of the time and should be found in the summary section due to the expected average value of 110 milliseconds. The system response seems slow and should not be present in a common application usage patterns.

Low Variance delay with a 50 milliseconds constant in addition to a random value with a maximum of 40 milliseconds. The event listener is on the limit of being slow, but perform decent in most cases. Should be listed in summary section.

7.1.1 Test Scenarios

The test scenarios exercise the buttons of the `MainActivity`-class, shown in Figure. 7.1. Each of button is exercised by a dedicated test class. This ensure similar application state for each execution as ALP reinstalls the application between execution of different test classes.

A click on a button within the `MainActivity`-activity trigger execution of the event listener with the corresponding delay. The event listeners perform the delay before making a call to the startActivity-method, which takes an intent-object as argument, specifying the `ResultActivity` (shown in Figure 7.2) as the target activity. The `ResultActivity` features only a back-button and a text-view. The text-view shows the duration of the activity transition that opened the activity.

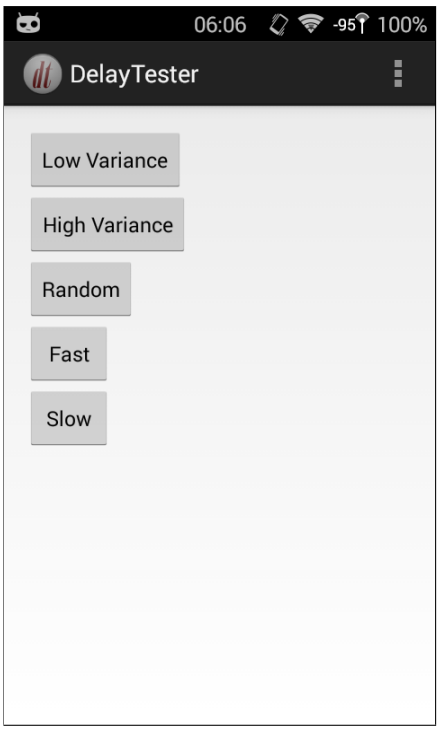


Figure 7.1: The Main Activity of the DelayTester application.

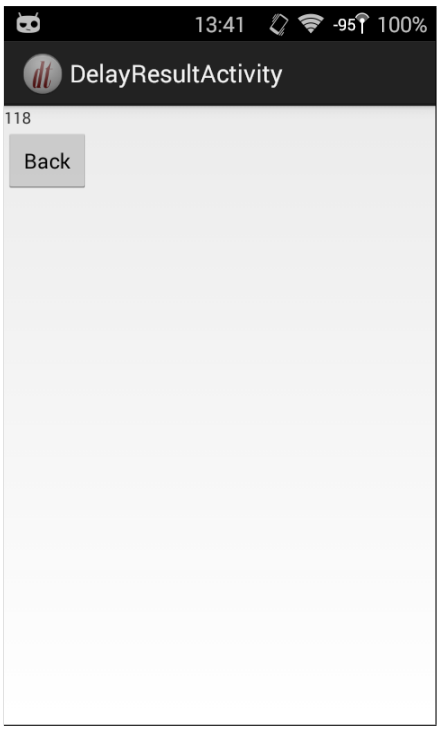


Figure 7.2: The Result Activity of the DelayTester application.

7.1.2 Results

The test-scenarios were executed 150 times each allowing for precise characterization of each event listener. The result of the individual event listener traces are much as expected based on the different delay times.

Figure. 7.3 shows the contents of the summary section of the evaluation report. As expected the event listener of the *Fast* test scenario is not present in the summary as it falls below the threshold value. The others are listed with values close to their expected values.

The Cross Activity Trace section contains measurements of how long time it takes from the beginning of an event listener to a new activity has been loaded and is visible to the user. As each event listener in the Delay Tester application contain a call to the start activity method for launching the result activity, all of them are listed in the Cross Activity Traces section of the report, as shown in Figure 7.4. It can be seen that the activity transitions have some variance in processing duration. This is caused both by the event listeners themselves and by the Android task scheduling mechanism. This is particularly visible for the *Slow* and *Fast* event listeners, as

their processing durations still vary with a standard deviation of 6 and 7 milliseconds respectively, even though they feature constant delays. As such it is not completely within the scope of application developers to improve and optimize activity transition performance. However, it can be seen that the event listeners exercised during the *High Variance*, *Slow* and *Random* scenarios are performing significantly worse with considerably higher values for average and standard deviation. Thus, keeping the event listener performance on an acceptable level is of high importance for activity transitions, particularly if the transitions are part of common usage patterns.

In the dataset section of the report it can be seen that the fast event listener has an average of 15 milliseconds and a standard deviation of 3 milliseconds. This is on average 5 milliseconds more than its delay would suggest. The additional 5 milliseconds can be attributed to the instantiation of the intent object, and the processing of the start activity method.

Test Method	Signature	Average	Median	St.Dev.	Max	Min
High Variance	void highVarianceDelay(Button)	120	121	36	174	54
Low Variance	void lowVarianceDelay(Button)	74	73	13	150	54
Slow	void slowDelay(Button)	95	95	1	102	93
Random	void randomDelay(Button)	94	91	56	201	7

Figure 7.3: Summary section for testing on the DelayTester application.

Test Method	Source Event Listener	Target Activity	Average	Median	St.Dev.	Max	Min
HighVariance	void highVarianceDelay(Button)	DelayResultActivity	174	174	38	244	106
LowVariance	void lowVarianceDelay(Button)	DelayResultActivity	125	124	14	184	95
Slow	void slowDelay(Button)	DelayResultActivity	147	147	6	175	127
Fast	void fastDelay(Button)	DelayResultActivity	68	68	7	89	48
Random	void randomDelay(Button)	DelayResultActivity	146	141	56	261	42

Figure 7.4: Cross Activity Traces in the DelayTester application.

7.2 Tomdroid

The Tomdroid application is a open source note taking application. All of its event listeners are assigned programmatically. The test scenarios used for testing were recorded using the Testdroid Recorder. The test scenarios was executed 100 times, where each cycle took approximately 4 minutes to complete.

7.2.1 Test Scenarios

Clear Local Database navigates from the main notes list view to the preferences menu to click the *Clear Notes* list item. When asked to confirm the action, the *Yes*-button from the dialog-box is clicked. A short delay can be noticed when confirming the clear database action.

Clear Search History makes two searches before navigating to and clicking the *Clear Search History* item in the preferences activity. After performing the button click in the preferences menu nothing happens the first moment, before a `Toast`- message shows the message *Search history has been cleared.*

Delete Note deletes the default welcome note. No responsiveness issues was detected during manual execution of the test scenario.

Add Note adds a note with a title and some contents. No responsiveness issues was detected during manual execution of the test scenario.

Search For Note adds a note and searches for its title. No responsiveness issues was detected during manual execution of the test scenario.

View About From Tomdroid Menu selects *About* from the menu, checks the message and click *OK*. No responsiveness issues was detected during manual execution of the test scenario.

Delete First Note deletes the default note. No responsiveness issues was detected during manual execution of the test scenario.

Sorting adds three notes and toggle between using *Date* and *Title* as sorting criteria. No responsiveness issues was detected during manual execution of the test scenario.

7.2.2 Results

As each of the test scenarios was executed 100 times, a broader view of the application performance was obtained than what was achieved by manually executing the test scenarios. This is reflected in the summary, shown in Figure 7.7. It contains event listeners that was not noticed to be slow during manual testing. This is understandable when taking into account that many of the listed event listeners have descent average processing duration, but suffer from occasional slowdowns as seen from their standard deviation values. An example of this is the `Boolean Tomdroid.onOptionsItemSelected (MenuItemImpl)`-event listener which is registered during execution of seven different test classes where four of them (`addNnote()`-test cases) execute the same operation.

The `ClearLocalDatabase.testClearDb()`-test case also revealed some long duration processing within the `PrefrerencesActivity`-activity occurring when confirming the deletion of the local database from the `AlertDialog`-dialog. The processing is performed in the `void PreferencesActivity$12. onClick(AlertDialog, Integer)`-event listener. The event listener is called when the user confirms the action to delete the database. Upon inspection of the source code, it seems that this operation performs the delete operation, an insertion of the default note, and the creation of a `Toast`-message box on the UI-thread. The operations could be moved to a worker thread to save some time on the UI-Thread.

The `void Tomdroid$8.onClick(AlertDialog, Integer)` event listener is registered during execution of four different test cases. Inspection of the scenario executions, using the bar-graph (see Figure 7.5), revealed that the readings are a result of the same application functionality being exercised by all four test scenarios. The event listener is executed when clicking the *OK*-button of the welcome message dialog-box (shown in Figure 7.6), as it is displayed on the first launch of the application and must be clicked before normal usage can begin. This was not noted during manual testing as it is expected by the user that the application does some extra processing on the initial start up.

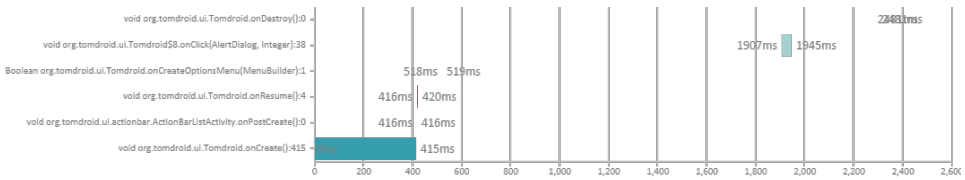


Figure 7.5: Bar-graph showing the initial start-up of the Tomdroid application.

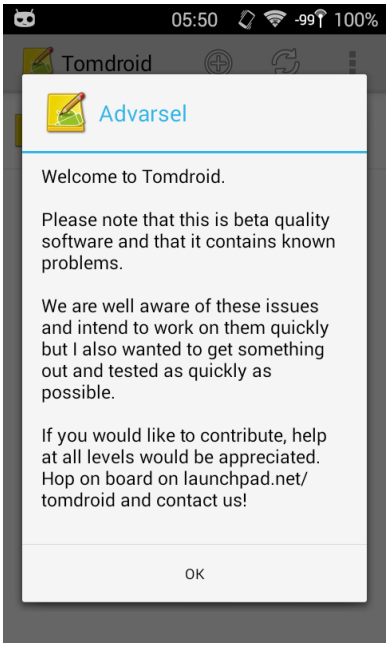


Figure 7.6: Tomdroid Welcome Message shown the first time the application is started.

During manual execution of the scenario involving clearing of the local notes data-base, opening the preferences menu seemed to take a bit longer than other similar operations. This was not picked up by the automatic testing, as transitioning from the `Tomdroid`-activity to the `PreferencesActivty`-activity do not show particularly higher values than transitions to other activities. However, it can be seen that activity transitions tend to have high variance. As such, manual testing can lead to false positives when the tester is unfortunate with the sample.

Test Method	**Signature**	**Avg.**	**Med.**	**St.D.**	**Max**	**Min**
Sorting.testAdd4note	Boolean Tomdroid.onOptionsItemSelected(MenuItemImpl)	52	48	15	184	42
Sorting.testAdd2note	Boolean Tomdroid.onOptionsItemSelected(MenuItemImpl)	58	54	12	100	44
Sorting.testAdd1note	Boolean Tomdroid.onOptionsItemSelected(MenuItemImpl)	51	48	12	139	40
Sorting.testClickOk	void Tomdroid$8.onClick(AlertDialog, Integer)	28	23	21	161	17
Misc.test0Init	void Tomdroid$8.onClick(AlertDialog, Integer)	28	22	23	152	17
Misc.testAddANote	Boolean Tomdroid.onOptionsItemSelected(MenuItemImpl)	54	47	13	104	39
Misc.testAddADifferentNote	Boolean Tomdroid.onOptionsItemSelected(MenuItemImpl)	54	50	17	196	41
WelcomeScreen.testClickOkAndAddANote	Boolean Tomdroid.onOptionsItemSelected(MenuItemImpl)	49	46	12	102	37
	void Tomdroid$8.onClick(Integer, AlertDialog)	30	24	24	177	18
ClearLocalDatabase.testClearDb	void PreferencesActivity$12.onClick(AlertDialog, Integer)	145	138	25	291	117
	void Tomdroid$8.onClick(AlertDialog, Integer)	30	23	24	152	16
Settings.testClearSearchHistory	Boolean PreferencesActivity$4.onPreferenceClick(Preference)	191	172	55	528	142

Figure 7.7: Summary section for testing on the Tomdroid application.

Test Method	Source Event Listener	Target Activity	Avg.	Med.	St.D.	Max	Min
Sorting.testAdd4note	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.EditNote	196	194	16	329	174
Settings.testClearSearchHistory	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.PreferencesActivity	103	103	3	113	97
Sorting.testAdd3note	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.EditNote	184	185	8	210	169
Misc.testAddADifferentNote	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.EditNote	179	177	17	322	156
WelcomeScreen.testClickOkAndAddANote	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.EditNote	172	168	14	231	147
WelcomeScreen.testClickOkAndAddANote	Boolean ui.ViewNote.onOptionsItemSelected(ActionMenuItem)	ui.Tomdroid	64	63	5	93	54
ClearLocalDatabase.testClearDb	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.PreferencesActivity	102	101	6	132	95
ClearLocalDatabase.testClearDb	Boolean ui.PreferencesActivity.onOptionsItemSelected(ActionMenuItem)	ui.Tomdroid	70	69	11	108	55
Settings.testSettings	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.PreferencesActivity	101	100	4	114	96
Settings.testSettings	Boolean ui.PreferencesActivity.onOptionsItemSelected(ActionMenuItem)	ui.Tomdroid	62	62	5	77	55
Sorting.testAdd2note	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.EditNote	181	179	12	232	163
Misc.testAddANote	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.EditNote	167	164	19	300	142
Sorting.testSortByDate	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.PreferencesActivity	109	108	6	131	100
Sorting.testAdd1note	Boolean ui.Tomdroid.onOptionsItemSelected(MenuItemImpl)	ui.EditNote	163	159	14	256	143

Figure 7.8: Cross Activity Traces in the Tomdroid application.

7.3 K9-Mail

The K9-Mail application is a huge and complex Android application with 17 activities and 67 969 lines of code. The test scenarios used for testing covers common uses of the application. A Gmail account was used to send and receive email during testing. The device was connected to the Internet through a Wifi broadband connection and as such is not guaranteed to discover issues related to network communication. The test scenarios was executed 100 times, where each cycle took approximately 5 minutes to complete.

7.3.1 Test Scenarios

Account Set-up makes use of the configuration wizard which is starting automatically on the fist launch of the application. The scenarios scrolls through the welcome message, completes the wizard making the required configurations to use the Gmail-email account, before it reaches the `Accounts`-activity, shown in Figure 7.9. There was some waiting involved during the procedure. However, non that was not expected, as information regarding long duration processing such as as network communication was shown to the user before initiation as well as through frequent progress reports.

Check Email start out on the Accounts activity before selecting the Gmail-account to enter the `MessageList`-activity. The activity features a list of emails located in the in-box mail folder, as seen in Figure 7.10. The synchronize button is clicked to refresh the list. No slowdown was noticed during manual testing. It seems that the email synchronization is performed correctly using background threads. Immediate feedback is also provided by changing the icon of the synchronization button to an animated version, informing that synchronization has been initiated.

Send Email selects the Gmail account from the `Accounts`-activity resulting in transitioning to the `MessageList`-activity where the new email button is clicked. This opens the `MessageCompose`-activity (see Figure 7.11) where the receiver email address, message title, and message body, is entered to their respective text-views before the *send message*-button is clicked. When the message is sent, the application navigates back to the `MessageList`-activity. No responsiveness issues was detected during manual execution of the test scenario.

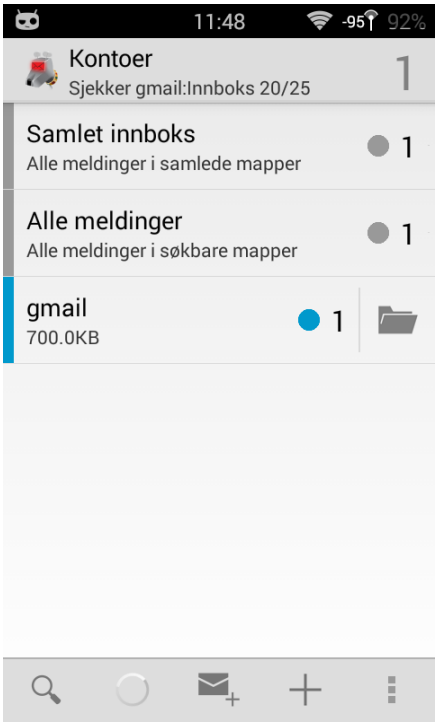


Figure 7.9: Start up screen of the K9-Mail email application. Signed in to a Gmail account.

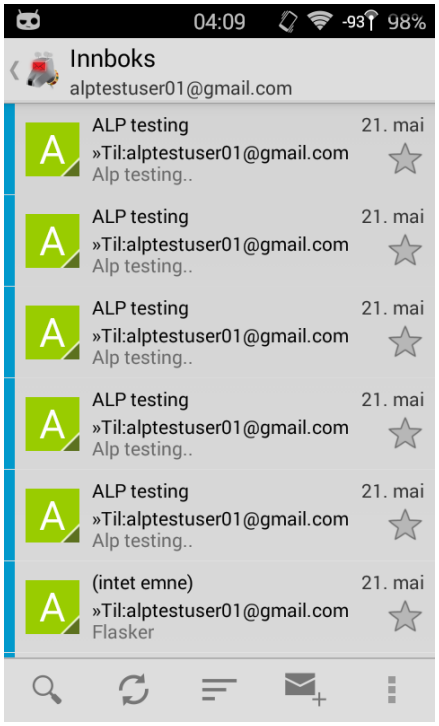


Figure 7.10: The `MessageList`-activity of the K9-Email application showing the inbox-folder.

Add Draft starts out at the `Accounts`-activity and navigate to the `MessageList`-activity before composing an email. When the email is composed, it is stored as a draft by clicking the *Save Draft* menu-item. After the draft has been saved the `MessageList`-activity is reopened. No responsiveness issues was detected during manual execution of the test scenario.

Delete Draft The scenario deletes all drafts by navigating from the `Accounts`-activity to the `FolderList`-activity before navigating to the `MessageList`-activity with the drafts folder. All elements are selected by clicking *Select All* from the options menu, before the elements are deleted by clicking the delete button. The delete operation is seen to spend some time to processing. This is however done without locking up the user interface while the user is frequently updated on the progress through regular updates of the text-view which can be seen highlighted in Figure 7.12. *Activity Browsing*-scenario

sweeps through various settings activities and changes the application color theme. No responsiveness issues were detected through manual execution of the scenario. Activity transitions have noticeable processing durations, but are kept at a minimum and outside of the most common application navigation.

No repeatable responsiveness issues were detected by manually executing the above test scenarios, however the application occasionally show some uneven frame-rates during activity transition animations and view element switch animations.

Figure 7.11: Composing a new Email in the K9-Email application using the `MessageCompose`-activity.

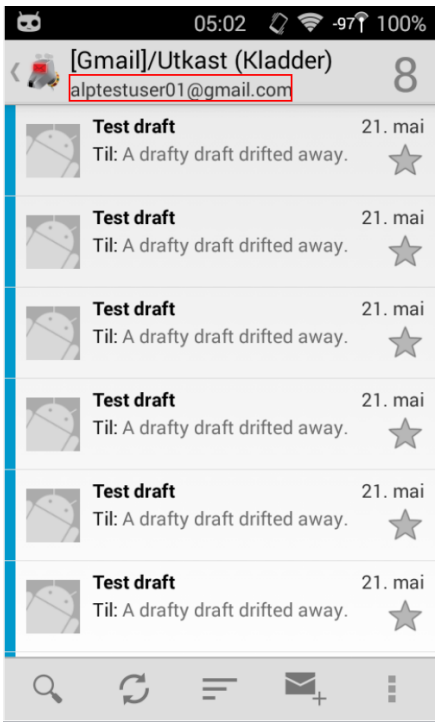


Figure 7.12: Drafts viewed in the `MessageList`-Activity. Progress of background tasks are shown in the highlighted text-view.

7.3.2 Results

K9-Email is a mature and optimized email application, and this can be seen from the short report summary, shown in Figure 7.13. The application make frequent use of background processing through the `AsyncTask`-class which allows for convenient update of the user interface through callback methods. This allows the user to navigate freely while waiting for tasks to complete and avoid frustration if tasks take longer time than usual. In addition, the number of transitions to new activities are held at a minimum by instead switching out parts of the view hierarchy to enable different functionality. An example of this is the `MessageList`-activity, which is used to show both the messages list and complete email details through a message view.

The *Account Setup*-test scenario has four different event listeners listed in the summary section of the evaluation report, non of which was noticed during manual execution of the scenario. The slowest of the event listeners was the `activity.setup.AccountSetupNames.onClick(Button)`-method, with an average execution time of 211 and a standard deviation of 68 milliseconds. The event listener is executed when the user has entered an optional name for the email account and the required name of the email account user. The operation include a validation of the text input fields before the information is saved by use of the `Account.save()`-method and the `Accounts`-activity is started with the accounts list visible.

The inner class `UpgradeDatabaseBroadcastReceiver`, of the `activity .UpgradeDatabases`-activity has the event listener `void onReceive(Intent, Context)`. This processing is not a direct response to user input, as is executed upon reception of broadcast messages from the `DatabaseUpgradeService`-service.

The `activity.setup.AccountSetupBasics.onClick(Button)`-event listener checks the input values on the account set-up screen. This includes a check of the entered email address, if the service provider details (server addresses, email protocols, etc) is known by the application or must be entered by the user. The processing duration of the event listener is on the limit of what is visible in the summary report, and as such is not likely to be noticed by the user.

Processing of the `activity.setup.AccountSetupBasics.onActivityResult (Integer)`-method is not a direct result of user input, as it is executed upon transition from the `AccountSetupCheckSettings`-activity. As such, the processing duration is not very noticeable to the user, as the activity transition animation is shown just prior to its execution and the user do require some time to review the new screen before providing the next input.

The *Check Email*-scenarios shows `Boolean activity .FolderList`

`.onOptionsItemSelected (ActionMenuItem)`-event listener being suspected of slow operation, with the 109 milliseconds average and 31 millisecond standard deviation. The method handles the processing associated with the selection of menu items. The scenario execute the method with the *Compose*-menu item, selecting the option to compose a new email, triggering a transition to the `MessageCompose`-activity. As such, the lack of responsiveness during the processing is not directly noticeable to the user. If we look at the transitioning in total, using the cross activity trace, we can see that total processing duration, as experienced by the user, is 251 milliseconds on average with a standard deviation of 16 milliseconds.

Execution of the *Browse Settings* scenario resulted in a single event listener being listed in the results summary. The `FolderList.onOptionsItemSelected (MenuItemImpl)`-event listener reads an average of 95 milliseconds and a standard deviation of 41 milliseconds. The bar-graph reveal that the event listener is exercised six times through the scenario in association with opening the `activity.setup.Accounts`-activity. This is picked up in the cross activity traces section as well, where the transition has an average duration of 552 milliseconds.

Test Method	**Signature**	**Avg.**	**Med.**	**St.D.**	**Max**	**Min**
testActivityBrowsing	Boolean activity.FolderList.onOptionsItemSelected(MenuItemImpl)	95	111	41	147	0
accountSetup	void activity.UpgradeDatabases$UpgradeDatabaseBroadcastReceiver.onReceive(Intent, K9)	152	62	171	835	0
	void activity.setup.AccountSetupBasics.onActivityResult(Integer)	92	109	92	420	2
	void activity.setup.AccountSetupBasics.onClick(Button)	77	76	11	116	62
	void activity.setup.AccountSetupNames.onClick(Button)	210	209	68	465	89
testCheckEmail	Boolean activity.FolderList.onOptionsItemSelected(ActionMenuItem)	109	111	31	232	35
	Boolean activity.MessageList.onKeyUp(KeyEvent, Integer)	65	34	38	109	27

Figure 7.13: Summary section from testing on the K9-Email application.

Test Method	Event Listener	Target Activity	Average	Median	St.Dev.	Max	Min
testDeleteDrafts	void activity.Accounts.onCreate()	activity.MessageList	437	380	164	1119	290
testDeleteDrafts	Boolean activity.MessageList.onOptionsItemSelected(ActionMenuItem)	activity.FolderList	101	99	10	144	81
testActivityBrowsing	void activity.Accounts.onCreate()	activity.MessageList	383	327	167	1052	265
testActivityBrowsing	Boolean activity.FolderList.onOptionsItemSelected(ActionMenuItem)	activity.Accounts	552	502	172	938	305
testActivityBrowsing	Boolean activity.MessageList.onOptionsItemSelected(ActionMenuItem)	activity.FolderList	103	101	10	130	86
testActivityBrowsing	Boolean activity.FolderList.onOptionsItemSelected(MenuItemImpl)	activity.setup.AccountSettings	217	215	6	229	213
testActivityBrowsing	Boolean activity.MessageList.onOptionsItemSelected(MenuItemImpl)	activity.setup.FolderSettings	108	100	23	179	91
testActivityBrowsing	Boolean activity.MessageList.onOptionsItemSelected(MenuItemImpl)	activity.setup.Prefs	129	126	15	217	112
testAddADraft	void activity.Accounts.onCreate()	activity.MessageList	429	364	174	1095	293
testAddADraft	Boolean activity.MessageList.onOptionsItemSelected(MenuItemImpl)	activity.MessageCompose	266	258	30	362	223
accountSetup	void activity.setup.AccountSetupBasics.onActivityResult(Integer)	activity.setup.AccountSetupCheckSettings	84	83	11	131	60
accountSetup	void activity.setup.AccountSetupBasics.onActivityResult(Integer)	activity.setup.AccountSetupNames	269	259	65	484	167
accountSetup	void activity.setup.AccountSetupNames.onClick(Button)	activity.Accounts	1607	1586	226	2170	1151
accountSetup	void activity.setup.WelcomeMessage.onClick(Button)	activity.setup.AccountSetupBasics	110	108	8	139	98
accountSetup	void activity.setup.AccountSetupBasics.onClick(Button)	activity.setup.AccountSetupCheckSettings	125	123	11	162	105
accountSetup	void activity.Accounts.onCreate()	activity.UpgradeDatabases	169	86	189	814	46
accountSetup	void activity.Accounts.onCreate()	activity.setup.WelcomeMessage	148	118	90	559	85
testCheckEmail	void activity.Accounts.onCreate()	activity.MessageList	450	408	120	876	314
testCheckEmail	Boolean activity.FolderList.onOptionsItemSelected(ActionMenuItem)	activity.Accounts	512	466	136	907	302
testCheckEmail	Boolean activity.MessageList.onOptionsItemSelected(ActionMenuItem)	activity.FolderList	102	100	14	164	83
testSendEmail	void activity.Accounts.onCreate()	activity.MessageList	377	347	126	1131	272
testSendEmail	Boolean activity.MessageList.onOptionsItemSelected(MenuItemImpl)	activity.MessageCompose	251	251	16	306	200

Figure 7.14: Cross Activity Traces in the K9-Mail application.

7.4 TextWarrior

The Text Warrior application was tested with scenarios involving tasks like opening a text file from the SD-card, scrolling up and down the text view, writing, and performing search and replace operations. The test scenarios was executed 100 times, where each test cycle took approximately 4 minutes to complete.

7.4.1 Test Scenarios

Open text-file opens a text file from the SD-card. This involves navigating to the file-picker activity before browsing the folder structure to locate the *Download*-folder where the text file is located. The text file is opened and the application transitions to the text view activity where the text is displayed. A few modifications is made to the file before it is saved by clicking the save menu button. No responsiveness issues was detected during manual execution of the test scenario.

Create File A file is created and a few sentences are written before the file is saved by clicking the *Save* menu item resulting in a activity transition to the file picker activity where the file is given a name and saved in the *download* folder on the SD-card. No responsiveness issues was detected during manual execution of the test scenario.

View Statistics opens a file and the *View More* menu option is clicked to reveal the *Wordcount and Properties*-item. The item is clicked, resulting in the opening of a dialog-box which shows information about the text file. The *OK*-button of the dialog is clicked and the scenario is completed. No responsiveness issues was detected during manual execution of the test scenario.

Change Color Theme start by clicking the *More* menu item, allowing navigation to the settings-activity where the *Color Theme*-item is clicked. This results in opening of a radio-button dialog-box with a set of selectable color themes. The *Solarized Dark* theme is selected, closing the dialog-box. The Back button is clicked to navigate back to the text activity. A slight delay is noticed when opening the settings activity. No further issues was detected during manual execution of the test scenario.

Open Java File navigates to the file picker activity to open a file containing Java source code. With the file opened in the text view activity, the scenario navigates to the settings activity to enable syntax highlighting. A slight delay is noticed when opening the settings activity. No further issues was detected during manual execution of the test scenario.

FindAndReplace opens a file containing Java source code before using the

replace-all feature to replace all occurrences of the *line*-word with *Line*. A slight delay can be noticed when the search view overlay is activated. This issue is only noticeable the first time the view is opened. No further issues was detected during manual execution of the test scenario.

View Help opens the help activity from the menu before scrolling through it. The back-button is clicked for navigating back to the text view activity. A slight delay can be noticed when transitioning to the help activity, however the help text content loads immediately when the activity becomes visible. No further issues was detected during manual execution of the test scenario.

View About open the about-message dialog from the *More* section of the menu. Finishes by closing the dialog-box by clicking the *OK* to return to the text view activity. No issues was detected during manual execution of the test scenario.

7.4.2 Results

Only a single event listener, the `onSaveInstanceState(Bundle)` of the `TextWarriorApplication` activity was listed in the summary section of the report, as can be seen in Figure. 7.15. This was not expected, as manual testing revealed multiple short delays while performing different tasks. The explanation lies in the fact that all but one of the delays was noticed during transitioning to other activities from the `TextWarriorApplication` activity.

The `onSaveInstanceState(Bundle)` method is called before the activity is placed in the background and is used by the application to save the text file when auto backup is enabled. Auto backup is enabled default and is therefore also enabled during test execution. Saving the text file to the file system is an unpredictable operation. This is noticed by the ALP-tool as the method has an average of 79 milliseconds and a 38 milliseconds standard deviation. It can also be seen that the majority of the traces is has descent processing durations while some occurrences last longer with a maximum of 277 milliseconds. The method must be completed before another activity can be moved to the top of the activity stack, and as such can have an unfortunate impact on application performance.

The issues noticed when transitioning to other activities from the `TextWarriorApplication` activity is picked up by the ALP in the Cross Activity Trace section of the report, seen in Figure 7.16. The scenario involving opening of a Java source code file shows a delay with a significantly higher average value than similar transitions when opening the `TextWarriorSettings` activity. Inspecting the bar-graph of the test scenario reveal that the onCreate method of the target activity uses especially long time. From the dataset section it seen that the onCreate method has an average processing time of 1330 milliseconds and a standard deviation of 45 milliseconds.

Test Method	**Signature**	**Avg.**	**Med.**	**St.D.**	**Max**	**Min**
testCreateAFile	void textwarrior.android.TextWarriorApplication.onSaveInstanceState(Bundle)	79	65	38	277	51

Figure 7.15: Summary section from testing on the Text Warrior application.

Test Method	Source Event Listener	Target Activity	Avg.	Med.	St.D.	Max	Min
testViewStatistics	Boolean TextWarriorApplication.onOptionsItemSelected(MenuItemImpl)	FilePicker	243	234	29	381	211
testOpenJavaFile	Boolean TextWarriorApplication.onOptionsItemSelected(MenuItemImpl)	FilePicker	236	230	21	295	201
testOpenJavaFile	Boolean TextWarriorApplication.onOptionsItemSelected(MenuItemImpl)	TextWarriorSettings	1564	1572	56	1679	1419
OpenFile	Boolean TextWarriorApplication.onOptionsItemSelected(MenuItemImpl)	FilePicker	249	240	41	413	199
testCreateAFile	Boolean TextWarriorApplication.onOptionsItemSelected(MenuItemImpl)	FilePicker	373	357	60	553	298
testFindAndReplace	Boolean TextWarriorApplication.onOptionsItemSelected(MenuItemImpl)	FilePicker	198	194	24	315	170
testViewHelp	Boolean TextWarriorApplication.onOptionsItemSelected(MenuItemImpl)	TextWarriorHelp	327	330	26	401	247
testChangeColorScheme	Boolean TextWarriorApplication.onOptionsItemSelected(MenuItemImpl)	TextWarriorSettings	258	252	31	397	204

Figure 7.16: Cross Activity Traces in the TextWarrior application.

7.5 Sky Map

The Sky Map application was tested by browsing the different menus, making searches for celestial bodies and and looking at maps from different times and locations. Sensor navigation is a major part of the application and was not used during testing. As such, event listeners associated with sensor input have not been properly exercised. The test scenarios was executed 70 times, where each test cycle took approximately 5 minutes to complete.

7.5.1 Test Scenarios

Accept Licence accepts the license agreement which is prompted to the user the first time the application is started. The *Accept*-button is clicked and the `DynamicStartMapActivity`-activity is shown. The licence agreement text takes a moment to load, as the dialog-box can be seen drawn in half size before the text is visible. When the licence is accepted the sky map is shown immediately without any delay.

Set Sensor Sensitivity adjusts the sensitivity of the motion sensor to reduce the impact of shaky hands when viewing the sky. The settings is located in the `EditSettingsActivity`-activity available from the menu in the `DynamicStartMapActivity`-activity. The *Slow*-setting is selected and the back-button is pressed to navigate back to the star map. The menu take a moment before it is visible and a short delay is noticeable when opening and returning from the preferences activity. No further responsiveness issues was noticed during manual test execution.

Set Location sets the location of the user. The location must be known to the to determine which part of the sky is visible. The application support automatic detection of position through GPS and manual setting. The scenario sets the location manually to ensure test stability. Settings the location require navigation from the `DynamicStartMapActivity`-activity to the `EditSettingsActivity`-activity where the location can be set through a dialog-box with a text-box for location search, or input of coordinates. The search is used, and the city Bergen is selected. A small delay can be noticed when opening the menu in addition to a small delay when navigating to and returning from the preferences activity. No further responsiveness issues was noticed during manual test execution.

Toggle Celestial Bodies deselects and selects which celestial bodies are visible on the star map. This is done by use of the check-boxes located in the `EditSettingsActivity`-activity. The scenario first un-checks display of all bodies, navigates back to the (now completely empty) star map, before re-enabling drawing of the bodies. When opening the menu a small delay was

noticed, in addition to a delay when navigating to and returning from the preferences activity. No further responsiveness issues was noticed during manual test execution.

Find Mars locates the planet Mars on the sky map using a search function. When the planet has been found an arrow points in its direction allowing the user to locate it in the sky. The search function is available from the menu of the `DynamicStarMapActivity`-activity. When opening the menu a slight delay can be noticed. No further responsiveness issues was noticed during manual test execution.

Time Travel utilize the Time Travel functionality to view the star map at a different point in time. The function is available from the menu of the `DynamicStarMapActivity`-activity. Clicking the menu item opens a dialog-box with a drop-down menu and some buttons. How long to skip into the past or future is selected either through setting a date or by choosing from a set of alternatives. The scenario looks up the map for Bergen city the next morning. When opening the menu and the dialog-box a slight delay can be noticed. No further responsiveness issues was noticed during manual test execution.

View Terms views the terms of service by opening the the menu of the `DynamicStarMapActivity`-activity and clicking the *View Terms of Service* item. The terms are displayed in a dialog-box containing a back button to navigate back to the sky map. A slight delay is noticed when opening the menu and when selecting the menu item.

View Help views the help page which can be opened from the menu of the `DynamicStarMapActivity`-activity. The help message is shown in a dialog-box with an *OK*-button. The scenario scrolls through the help text before clicking the button. When opening the menu a slight delay can be noticed. In addition, when clicking the help menu item the the application stops for a moment before the help dialog-box becomes visible. This only occur the first time it is opened, which often indicate some file-system or network operation is being performed and the result cached in memory for later use.

7.5.2 Results

As with K-9 Mail, Sky Map is an established and well performing application. As such, it would be surprising if a huge amount of responsiveness issues was detected. The summary section of the evaluation report, as shown in Figure 7.18, do not contain a high number of event listeners, with only three different methods noted. The `onKeyDown(Integer)` method of the `DynamicStarMapActivity` activity was present in all of the test scenarios, while the `onOptionsItemSelected (MenuItemImpl)` method of the `DynamicStarMapActivity` was present in five out of eight scenarios. The

third event listener noted in the summary was the `onPrefrenceChange(..)`-method of the `EditSettings`-activity.

The `onKeyDown(Integer)` event listener of the `DynamicStartMapActivity` has the responsibility of handling input from hardware button clicks. The method contains a switch statement to decide the appropriate response to the hardware key presses *DPAD_ LEFT*, *DPAD_ RIGHT*, and *BACK*, which are passed to the event listener as an integer parameter. The *dpad*-keys correspond to directional hardware navigation buttons, while the *back*-key represent the more common application back-navigation button. The dpad buttons result in the sky map view direction being rotated about its current center point. The back button removes the search bar visibility if it is currently present on the screen, otherwise the default action is performed. Other key events are handled as default, by being passed to the super method of the activity class. The event listener has an average varying from 36 to 90 milliseconds in the different test scenarios, which suggests that the different execution paths caused by the parameter is a key factor in determining its processing duration. This is particularly evident, as the minimum processing duration in all but one of the scenarios is 0 milliseconds.

Inspection of the dataset section of the report reveal that two key-codes have been used during test execution, where each result in a different processing duration. The key-code corresponding to the back button results in the slowest of the observed behaviours, while the quickest of the two handles the menu button. While the back button is clicked at the end of every scenario to finish its execution, it is not very noticeable, as it is expected that some processing is required to switch to the previous application on the activity-stack or to the operating system home screen.

The processing duration present when opening the menu in the `DynamicStarMapActivity` activity is noticeable to the user. However, the delay is only present at the very first invocation after the activity has loaded. This is achieved by retaining the inflated menu in system memory and to improve the response time in later invocations. This is not well reflected in the test results from the ALP-tool, as the application (and the activity) is restarted nearly every time the menu is accessed. This is an artefact of the test scenarios not providing sufficient complexity.

The `onOptionsItemSelected (MenuItemImpl)` method found in the `DynamicStarMapActivity` activity is responsible for handling menu item is clicks. As such, it is natural that the durations vary in the different scenarios. This can be seen, as the lowest average is 114 milliseconds while the highest is 253 milliseconds.

In the View terms scenario, the *Terms of Service* menu item is selected resulting in an average delay of 233 milliseconds produced by the onOption-

sItemSelected event listener. This was noticed during manual testing. From inspection of the source code, it can be seen that the action is logged using the Goolge Analytics API before the dialog-box is opened. A delay of 111 milliseconds is detected when opening the time travel dialog box in the Time Travel scenario. This was also detected during manual testing. For the View Help scenario, a slightly longer delay compared to other similar operations was noticed. This is reflected in the report, with a 254 milliseconds average processing duration. By inspecting the source code, it is seen that the operation is similar to opening the terms of service dialog, with some additions processing for displaying HTML formatted text.

For the Time Travel and Find Mars scenarios the processing duration of the onOptionsItemSelected event listener was much shorter than in the Terms of Service scenario, with averages of only 111 and 114 milliseconds. The Find Mars scenario launches the search view, which loads on top of the view hierarchy of the (current) sky map activity, while the Time Travel scenario use a dialog box to provide the user with time travel preferences. It was surprising to find that the time travel dialog box has half the processing duration compared to the other similar operations. Determining the exact cause with certainty would require further method profiling. However, a possible cause is the loading of the text resources for display in the text views, as the time travel dialog does not use such operations.

The `onPreferenceChange(EditTextPreference, String)` event listener located in the `EditSettingsActivity`, is only present in the Set Location scenario. It has an average processing duration of 232 milliseconds with a standard deviation of 104 milliseconds. Inspection of the dataset section reveal that the values of the arguments received by event listener are the same values which was selected in the edit (location) settings activity, which was opened from the preferences activity. Figure 7.17 shows the descriptions from the dataset section. The method make use of the `android.location.Geocoder` class to determine the coordinates of the user provided location. The user is kept updated through popup messages, and it is of benefit for the user to wait for the operation to complete (to be aware of the currently set location). However, the operations are shown to have a huge variance in processing duration, with the highest observed duration of 644 milliseconds. As such, the operations should be moved to a background thread to free up time on the UI-thread.

Argument Type	**Description**
EditTextPreference	Place name e.g. 1400 West Mars Hill Road, Flagstaff, AZ.
String	Bergen

Figure 7.17: Arguments of the `onPreferenceChange(..)` event listener of the EditSettingsActivity activity.

Test Method	**Signature**	**Avg.**	**Med.**	**St.D.**	**Max**	**Min**
testViewHelp	Boolean stardroid.activities.DynamicStarMapActivity.onKeyDown(Integer)	116	112	82	272	0
	Boolean stardroid.activities.DynamicStarMapActivity.onOptionsItemSelected(MenuItemImpl)	253	253	21	282	219
testToggleCelestialBodies	Boolean stardroid.activities.DynamicStarMapActivity.onKeyDown(Integer)	94	80	62	229	0
AcceptLicence	Boolean stardroid.activities.DynamicStarMapActivity.onKeyDown(Integer)	81	66	33	162	45
testSetSensorSpeedSlow	Boolean stardroid.activities.DynamicStarMapActivity.onKeyDown(Integer)	59	22	75	267	0
testViewTerms	Boolean stardroid.activities.DynamicStarMapActivity.onKeyDown(Integer)	91	86	74	321	0
	Boolean stardroid.activities.DynamicStarMapActivity.onOptionsItemSelected(MenuItemImpl)	233	214	51	374	193
testSetLocationToBergen	Boolean stardroid.activities.DynamicStarMapActivity.onKeyDown(Integer)	41	51	41	126	0
	Boolean stardroid.activities.EditSettingsActivity$1.onPreferenceChange(EditTextPreference, String)	232	189	104	644	139
testTimeTravel	Boolean stardroid.activities.DynamicStarMapActivity.onKeyDown(Integer)	108	95	89	281	0
	Boolean stardroid.activities.DynamicStarMapActivity.onOptionsItemSelected(MenuItemImpl)	111	98	35	175	69
testFindMars	Boolean stardroid.activities.DynamicStarMapActivity.onKeyDown(Integer)	64	36	76	365	0
	Boolean stardroid.activities.DynamicStarMapActivity.onOptionsItemSelected(MenuItemImpl)	114	108	25	235	93

Figure 7.18: Summary section from testing on the Sky Map application.

Test Method	Source Event Listener	Target Activity	Avg.	Med.	St.D.	Max	Min
testViewHelp	void SplashScreenActivity$1.onAnimationEnd(AnimationSet)	DynamicStarMapActivity	622	588	131	1093	396
testToggleCelestialBodies	void SplashScreenActivity$1.onAnimationEnd(AnimationSet)	DynamicStarMapActivity	639	615	111	1090	442
testToggleCelestialBodies	Boolean DynamicStarMapActivity.onOptionsItemSelected(MenuItemImpl)	EditSettingsActivity	190	190	34	227	148
AcceptLicence	void SplashScreenActivity$1.onAnimationEnd(AnimationSet)	DynamicStarMapActivity	773	772	34	869	685
testSetSensorSpeedSlow	void SplashScreenActivity$1.onAnimationEnd(AnimationSet)	DynamicStarMapActivity	627	598	108	940	440
testSetSensorSpeedSlow	void DynamicStarMapActivity.onCreate()	DynamicStarMapActivity	164	164	0	164	164
testSetSensorSpeedSlow	Boolean DynamicStarMapActivity.onOptionsItemSelected(MenuItemImpl)	EditSettingsActivity	234	224	36	412	205
testViewTerms	void SplashScreenActivity$1.onAnimationEnd(AnimationSet)	DynamicStarMapActivity	633	590	125	1026	413
testViewTerms	Boolean DynamicStarMapActivity.onOptionsItemSelected(MenuItemImpl)	DynamicStarMapActivity	483	483	0	483	483
testSetLocationToBergen	void SplashScreenActivity$1.onAnimationEnd(AnimationSet)	DynamicStarMapActivity	440	440	19	475	401
testSetLocationToBergen	Boolean DynamicStarMapActivity.onOptionsItemSelected(MenuItemImpl)	EditSettingsActivity	231	226	18	303	206
testSetLocationToBergen	void DynamicStarMapActivity.onResume()	DynamicStarMapActivity	868	676	909	5406	619
testTimeTravel	void SplashScreenActivity$1.onAnimationEnd(AnimationSet)	DynamicStarMapActivity	634	610	124	1036	428
testFindMars	void SplashScreenActivity$1.onAnimationEnd(AnimationSet)	DynamicStarMapActivity	602	576	92	932	421

Figure 7.19: Cross Activity Traces in the Sky Map application.

7.6 Case Studies Results Summary

The four open source applications evaluated by ALP all resulted in reports containing some signs of responsiveness issues. It is likely that a larger set of test cases, spanning more application functionality, would have revealed more responsiveness issues. However, it should be noted that the tested applications are mature and likely to have gone through multiple iterations of debugging and optimization.

The results of the testing is summarized in Figure 7.20. The Delay Tester is not a real Android application, as it does not serve any purpose besides testing and development of the ALP-tool. However, it was included in the summary table in the interest of completeness. The first and second row displays how many test scenarios was used during testing and how many event listeners was exercised during their execution. The third row shows how many of the traced event listeners that was suspected by the ALP-tool of introducing responsiveness issues to the application. The fourth row lists the number of issues that was detected during manual execution of the test scenarios, while the fifth row shows how many activity transitions was detected.

	Tomdroid	K9-Mail	TextWarrior	Sky Map	Delay Tester
Test Scenarios	8	6	8	8	5
Event Listeners Traced	15	23	15	10	5
Slow Event Listeners Detected	4	6	1	13	4
Manually Noticed Issues Detected	2	0	4	11	4
Cross Activity Traces	14	22	8	14	5

Figure 7.20: Test result summary of the 5 applications.

In total, many of the issues detected through manual execution of the test scenarios was also picked up by the ALP-tool. In all cases but the TextWarrior and the Delay Tester, more issues was detected by ALP than through manual testing. In the Text Warrior application, only a single event listener was listed in the summary section of the evaluation report. The main cause of this was that the noticed delays occurred when transitioning to other activities. The issue could be pinpointed to activity life cycle methods in the target activities by inspection of the the cross activity trace section and the bar-graph.

7.7 Evaluation and Overhead Profiling

It is conceivable that the instrumentation added by the ALP-tool introduce some processing overhead to the AUT and as a result, the readings do not reflect real world application performance accurately. To investigate the processing overhead introduced by the tool, tests was constructed for comparing performance of regular application usage with that of instrumented applications. The tests evaluate both the instrumentation code itself, but also the impact of the GUI-Testing framework utilized for executing test scenarios.

7.7.1 Manual versus ALP Performance Comparison

The DelayTester application was used to obtain activity transition traces through manual execution of test-scenarios. The results from the manual testing was then compared to the traces obtained from the ALP-tool. Manual test-scenario execution involved normal usage of the Android device by pointing the finger at the screen to interact with the application.

For manual test execution, a version of the DelayTester application without Aspectj and Android Test framework functionality enabled was created. The DelayTester-application has been implemented with the required functionality to manually investigate cross activity processing durations from the beginning. Figure 7.21 shows the Java method used for launching the `DelayResultActivity`-activity after a given delay. The event listener processing start time is stored and passed to the new activity by use of the `putExtra()`-method of the `Intent`-class. In the `DelayResultActivity`-activity, the duration is calculated and displayed in a text-view to allow manual readout.

For the manual testing, the *Fast* event listener was used. The event listener has a constant delay of 10 milliseconds and as such, the variance in activity transition duration is entirely caused by the Android task scheduler. A total of 110 iterations of manual and automatic testing was conducted. In Figure 7.22 we can see a comparison of the numbers obtained from both test techniques.

7.7.2 Investigating Advice-code Processing Durations

While most of the instrumentation logic is performed outside of the measurements of the targeted areas in the AUT, the work performed by the activity transition tracing advice, the `crossActivityTrace`-advice, is located within the event listeners that call the `starActivity()`-method. In addition, the

```java
private void startActivityWithDelay(long delay) {
   long time = System.currentTimeMillis();
   try {
      Thread.sleep(delay);
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
   Intent intent = new Intent(this, DelayResultActivity.class);
   intent.putExtra("t_mark", time);
   intent.putExtra("test_id", delay + "ms");
   startActivity(intent);
}
```

Figure 7.21: Method in the DelayTester application for transitioning to the result activity.

	Average	**Median**	**St.Dev.**	**Max**	**Min**	**Num.**
Manual	50	50	4	58	41	110
ALP	55	55	6	72	38	110

Figure 7.22: Comparison of Manual and ALP measurements of activity transitions in the DelayTester application.

ALP-tool use the `onStart()` method as the end-point in the activity transition duration when it is available. This is the case with the DelayTester application. The instrumentation of the `onCreate()`-method in the target activity is processed before the `onStart()` method, leading to a processing overhead in the trace. In order to interpret the traces produced by ALP, these delays must either be negligible or properly quantified. It is important to note that these issues are only present in activity transition traces, as described in section 5.1.1 Tracing of individual event listeners cannot overlap and cause interference with each other.

From Figure 7.22 it can be seen that the values for the two testing techniques are fairly similar. The ALP spends on average 5 millisecond longer than what was observed during manual testing. This could be the result of random variance or it could be caused by overhead in the ALP-instrumentation or the Robotium test framework.

To quantify the processing delay introduced by ALP by trace-advices, a test involving execution of the instrumentation code separated from any Android application code, was implemented. In Figure 7.23 we can see the `onAlpTesting()`-method being called from the `elTraceTest()`-method in a

loop on the lines 3 to 5. The processing duration of the loop is measured and logged, as seen on the lines 2,6 and 7. The body of the `onAlpTesting()`-method is empty, except for the advice-code added by the elTrace-pointcut at compile time. This ensure that only processing caused by ALP is measured. The parameters of an event listener can effect its logging time. As the `onCreate()`-method is the primary concern, a single parameter, the `View` object of the `elTraceTest`-event listener is used. To ensure that the test would not produce an Application Not Responding error, the number of loop cycles was limited to ten-thousand iterations.

```
1   public void elTraceTest(View view) {
2      long start = System.currentTimeMillis();
3      for (int i = 0; i < 10000; i++) {
4         onAlpTesting(view);
5      }
6      long duration = System.currentTimeMillis() - start;
7      Log.d("ALP_Test", duration+"");
8      }
9    public void onAlpTesting(View view) {
10   }
```

Figure 7.23: Overhead evaluation test for ALP- logging logic.

The test was executed ten times and the averages of each loop iteration was calculated with the results as shown in Figure 7.24. The total average of all runs was less than two milliseconds. This means that the instrumentation added in the `onCreate()`-method is only partly responsible for the differences in processing duration seen between manual and ALP cross activity tracing.

Iterations	1,523	1,5288	1,5137	1,5067	1,4871
	1,5672	1,5018	1,4908	1,4738	1,5373
Average	1,51302				

Figure 7.24: Result of ALP instrumentation overhead testing.

To investigate the processing time of the `crossActivityTrace`-advice another test was devised. The `crossActivityTrace`-advice is executed before calls to the `startActivity`-method of the activity class. When it is encountered during an event listener trace, the `crossActivityTrace`-advice locate the Intent-object and utilize the `intent.getComponent().getClassName()`-methods to find the fully qualified Java class name of the target Activity-class. The target Activity is stored as a static String-object that is available

to the logging logic of the event listener trace advices. To perform this testing, a dummy-startActivity method was required. This method can be seen in Figure 7.25, with the signature `public void startActivityDummy(Intent)`. The method call is instrumented in a similar manner as the `android.app.Activity.startActiviy()`-methods by a copy of the `crossActivityTrace`-advice and point-cut modified to target the dummy method. When the `onCrossActivityTraceTest()`-method, seen in Figure 7.25 line 1 is executed, the `crossActivityTrace`-advice code is executed ten-thousand times.

```
1  public void onCrossActivityTraceTest(View view) {
2      Intent intent = new Intent(this, DelayResultActivity.class);
3      long start = System.currentTimeMillis();
4      for (int i = 0; i < 10000; i++) {
5          startActivityDummy(intent);
6      }
7      Log.i("ALP_Test", (System.currentTimeMillis() - start)+"");
8  }
9  public void startActivityDummy(Intent intent) {
10 }
```

Figure 7.25: Test method for evaluation of `crossActivityTrace`-advice processing overhead.

As the test revealed some variance in the processing duration, the test was executed 110 times. Figure 7.26 shows the results of the testing in milliseconds for ten-thousand executions of the `crossActivityTrace`-advice code. This means that the average processing duration of the advice code is about 11 microseconds. This is not enough to effect event listener evaluation by any significant amount.

Average	Median	Standard Deviation	Minimum	Maximum
111	142	45	32	169

Figure 7.26: Results of `crossActivityTrace`-advice overhead evaluation testing.

The probing of the advice code goes only half-way in explaining differences in processing duration between the ALP-tool and the manual testing. The differences must therefore lie either in the Robotium Test Framework or simply be a result of the randomness introduced by the Android task scheduling mechanism. Regardless of cause, a few milliseconds of additional processing is not enough to invalidate the readings obtained through use of the ALP-tool, as a few milliseconds error margin is not noticeable to the user of the application.

Chapter 8

Conclusions and Future Work

The goal of the thesis was to investigate how traditional approaches for functional GUI-testing of Android applications can be utilized and extended to evaluate the responsiveness of an application under test. The approach required utilities to track and measure processing of user input, gather and store the resulting data as well as the means to analyse the data and provide meaningful information to the developer using the tool. An approach for evaluating the responsiveness of Android applications has been developed, implemented and evaluated.

8.1 Conclusions

The approach has been shown to be able to evaluate responsiveness in Android applications through the case studies presented in chapter 7. However, some challenges has also been discovered.

When testing the K9-Mail application, some occasional performance artefacts in the form of short breaks in animations and activity transitions was noticed. The issues could not be pinpointed during manual testing, and the results obtained from the ALP-tool did not provide any more insight to their causes. This would suggest that the operations causing the artefacts originate from outside the main thread of the application. This cannot be detected using the event listener instrumentation approach presented in this thesis.

Another limitation of the approach is that the quality of the evaluation report is always limited by the quality of the test scenarios. An example of this was encountered during testing of the Sky Map application, where opening the menu from the sky map activity was registered as a slow event listener by the ALP-tool. This was not noticed during manual testing. Upon

inspection, the cause of the inconsistency was discovered. When opening the menu for the first time, it is loaded into system memory, allowing it to be opened much quicker on later invocations. Opening the menu multiple times was not included in the test scenarios, and as such only the slow invocation carried weight in the evaluation report.

8.2 Related Tools and Approaches

The review of existing tools for evaluating responsiveness related issues in Android applications has revealed that there are many available approaches. Systrace and Traceview was found to be very effective tools for investigating performance issues that was known and roughly located within the AUT in advance. However, both tools had drawbacks making them unsuitable and challenging to use for detecting new responsiveness issues. Strict Mode can be used to avoid known performance issues like performing file system or network operations on the GUI-thread, but it cannot be used to detect general cases of long duration processing. The responsiveness analysis tool proposed by T. Ongkosit et.al. perform static analysis to detect blocking operations on the GUI-thread in order to produce a report where the operations are ranked by severity. Shengqian Yang et.al. propose an approach using insertion of additional delay to code segments adhering to specific criteria in order to monitor the effect it has on the responsiveness of the application during runtime. Figure 8.1 compares key aspects of ALP with similar tools.

	Android Listener Profiler	Systrace [19]	Traceview [18]	Strict Mode [10]	T. Ongkosit et.al. [12]	S. Yang et.al. [1]
Reflect Real World Performance	Yes	Yes	No	No	No	No
Manual Modification of Source Code	No	No	No	Yes	No	No
Dynamic Analysis	Yes	Yes	Yes	Yes	No	Yes
Static Analysis	Yes	No	No	No	Yes	Yes
Method Level Accuracy	Yes	No	Yes	Yes	Yes	Yes
Detect Responsiveness issues	Yes	Yes	No	Yes	Yes	Yes

Figure 8.1: Comparison of different Android application test tools.

8.3 Future Work

The evaluation report produced by the ALP-tool can be improved to increase its readability. The Dataset section should have an index allowing quick navigation to the areas of interest using hyper links and anchors. Currently navigation is performed using the scrollbar, which can be time consuming for large reports. In addition to the descriptive statistics, a histogram visualizing the datasets of each event listener, would allow quicker and more accurate interpretation.

Additional filtering of the event listener traces based on the values of the event listener arguments could be implemented. This has the potential of improving detection of issues like the one encountered during the Sky Map testing regarding the`onKeyDown(Integer)` event listener. This issues could easily have been left out of the report summary if the share of fast executions had out weighted the slower once.

The ALP implementation can be made easier to use by integrating the Instrumentation Generator and the Testrun Manager with an IDE such as Android Studio. This would allow for easy tool configuration through automatic detection of test classes, Android packages, and Android the SDK tools. The build process including aspect weaving could also be managed automatically. It would also provide a more natural work flow for application developers, as GUI-test scenarios are most commonly executed using IDE integrated tools already.

List of Figures

Bibliography

[1] *Testing for Poor Responsivness in Android Applications*
Shengqian Yang, Dacong Yan, Atanas Rountev Department of Computer Science and Engineering Ohio State University

[2] *Measuring the Performance of Interactive Applications with Listener Latency Profiling*
Milan Jovic, Matthias Mauswirth Faculty of Informatics, University of Lugano Lugano, Switzerland

[3] *Automating GUI Testing for Android Applications*
Cuixiong Hu, Lulian Neamtiu Department of Computer Science and Engineering University of California, Riverside, CA, USA

[4] *The Android Testing Support Library*
https://developer.android.com/tools/testing-support-library/index.html
(last visited 07.05.2015)

[5] *A Toolset for GUI Testing of Android Applications*
Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, Gennaro Imparato Dipartimento di Informatica e Sistemistica, Universistà di Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy

[6] *Writing Zippy Android Apps*
Brad Fitzpatrick Google I/O Developers Conference 20.06.2010 dl.google.com/googleio/2010/android-writing-zippy-android-apps.pdf
(last visited 07.05.2015)

[7] *Improving App Retention*
http://info.localytics.com/blog/app-retention-improves
(last visited 07.05.2015)

[8] *Robotium Test Automation Framework*
https://code.google.com/p/robotium/ (last visited 07.05.2015)

[9] *Profiling with Traceview and dmtracedump*
http://developer.android.com/tools/debugging/debugging-tracing.html
(last visited 07.05.2015)

[10] *Android StrictMode*
http://developer.android.com/reference/android/os/StrictMode.html
(last visited 07.05.2015)

[11] *Android Lint*
http://developer.android.com/tools/help/lint.html
(last visited 27.05.15)

[12] *Responsiveness Analysis Tool for Android Applications*
Thanaporn Ongkosit, Shingo Takada - Keio University Yokohama, Japan

[13] *Gradle Android Aspectj Plugin*
https://github.com/uPhyca/gradle-android-aspectj-plugin
(last visited 14.04.2015)

[14] *Eclipse Aspectj Plugin*
https://eclipse.org/ajdt/ (last visited 14.04.2015)

[15] *UI-Exerciser Monkey*
https://developer.android.com/tools/help/monkey.html
(last visited 28.04.2015)

[16] *Testdroid*
http://testdroid.com/
(last visited 28.04.2015)

[17] *Robotium Recorder*
http://robotium.com/products/robotium-recorder
(last visited 28.04.2015)

[18] *Traceview*
http://developer.android.com/tools/help/traceview.html
(last visited 14.05.2015)

[19] *Systrace*
http://developer.android.com/tools/help/systrace.html
(last visited 16.05.2015)

[20] *Android Testing Fundamentals*
http://developer.android.com/tools/testing/testing_android.html
(last visited 15.05.2015)

[21] *Junit Test framework*
http://junit.org
(last visited 15.05.2015)

[22] *ALP-Tool* download location.
https://drive.google.com/folderview?id=0Bw43574TMwbQTFVycmNrTW81NE0&usp=sharing
(last visited 31.05.2015)

[23] *ALP Video Demonstration*
https://drive.google.com/file/d/0Bw43574TMwbQMHFISEFJcUh0OVE/view
(last visited 31.05.2015)