

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



A Crash Reporting Library for Android

MASTER'S THESIS

Marek Osvald

Brno, Autumn 2017

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



A Crash Reporting Library for Android

MASTER'S THESIS

Marek Osvald

Brno, Autumn 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Marek Osvald

Advisor: prof. RNDr. Václav Matyáš M.Sc. Ph.D.

Acknowledgement

This work was carried out during the years 2016-2017 at AVG Technologies CZ and the Faculty of Informatics at the Masaryk University in Brno. It is a pleasure to use this page to sincerely thank all the people contributing or otherwise involved in the development of this thesis.

I owe my deepest gratitude to my thesis supervisor, professor Václav Matyáš. His continuous support helped me to write a thesis worthy of a prestigious institution such as the Masaryk University.

I am deeply grateful to my technical supervisor Petr Koudelka for his continuous optimism and guidance during the development. His advise helped me on countless occasions.

I would like to express my most sincere gratitude to Philip M. Gammon who answered my English-related questions and selflessly offered proofreading this text.

I am particularly grateful for support given by Katarína Varačková and Petr Buno during the initial prototype development. Their contribution helped to speed up the work on the prototype tremendously and it was truly a pleasure to work with both of them.

I want to express my gratitude to Marcel Kappler and Marting Šůs for their help integrating the solution with AVG's existing infrastructure. Their help was simply invaluable.

I am also in debt to Jan Švábenský who helped me to devise the testing strategy and for his insight during integration testing. His expertise helped me to learn a great lot about testing, QA automation and the underlying technology.

I would also like to thank Lucie Lénertová, Karel Ovesný, Libor Jancek, and Dominik Pokora for their comments during the GUI support tool development.

And last but definitely not least, I owe an endless debt of gratitude to my parents, Martina Osvaldová and Martin Osvald. They have tirelessly supported me throughout my entire academic career. Without their love, emotional and financial support this thesis would have never been possible. Thank you for being the way you are and helping me in every single imaginable way to achieve my goals and dreams. This text is dedicated to you.

Abstract

The goal of this master's thesis is to develop a universal and adaptable solution for crash reporting of Android applications. The thesis describes the entire development process of such solution, from its architecture concepts to the process of deployment including a test plan and means of its automation.

Keywords

Android, crash reporting, Google Play, JUnit, Mockito, PowerMock, Appium, TSD, autolib, Final-CI, Artifactory

Contents

1	Introduction	1
2	The Problem Defined	5
2.1	<i>The Nature of Failures on Android</i>	5
2.2	<i>Reporting Crashes</i>	5
3	Technologies Used	9
3.1	<i>Programming Languages</i>	9
3.2	<i>Tools</i>	11
3.3	<i>Libraries</i>	12
4	Implementation	15
4.1	<i>Analysis of Existing Solutions</i>	15
4.1.1	ACRA	15
4.1.2	HockeyApp	16
4.1.3	Crashlytics	17
4.2	<i>Architecture Concept</i>	21
4.3	<i>Handling Managed Code Crashes</i>	23
4.3.1	Collecting Metadata	26
4.4	<i>Handling Native Code Crashes</i>	28
4.4.1	Google Breakpad Integration	31
4.4.2	Initialising the Native Code Handler	34
4.4.3	Collecting Metadata	34
4.5	<i>Notifying the User</i>	35
4.6	<i>Providing Crash Data</i>	41
4.6.1	Default Synchronous Providers	42
4.6.2	Default Asynchronous Providers	44
4.7	<i>Configuration Management</i>	47
4.7.1	Application Identification Options	49
4.7.2	User Notification Options	50
4.7.3	Provider Options	51
4.7.4	Data Publishing Options	51
4.7.5	Connection Options	52
4.7.6	Logging Options	53
4.8	<i>Packaging of collected data</i>	53
4.8.1	The BinPacker File Format	53

4.8.2	BinPacker API	55
4.8.3	Support Tools	56
4.9	<i>Data Publishing</i>	57
4.9.1	CAP Crash Publisher	58
4.9.2	External Storage Publisher	60
4.9.3	Publishing Strategies	60
4.10	<i>Connection Handling</i>	61
4.10.1	Offline Publisher	63
4.10.2	Publishing Attempts	63
4.11	<i>Logging</i>	64
4.11.1	Logging API	65
4.12	<i>Testing</i>	66
4.12.1	Unit Testing	66
4.12.2	Integration Testing	68
4.12.3	Mock CAP Server	73
4.12.4	Automation	74
4.13	<i>Building</i>	75
4.13.1	Building the Library	75
4.13.2	Building the Simulator	76
4.13.3	Building the Mock CAP Server	78
4.13.4	Additional Gradle Tasks	79
4.14	<i>Publishing</i>	80
5	Conclusion	83
	Bibliography	85

List of Figures

- 2.1 Standard Android 7.1.2 Crash Dialogs. 6
- 4.1 Example of ACRA's Dialog Window. 17
- 4.2 HockeyApp's Dialog Window. 18
- 4.3 Example of HockeyApp Server Crash Reports. 19
- 4.4 Example of Fabric Crashlytics Dashboard. 20
- 4.5 Sequence Diagram: Handling Managed Code Crashes. 24
- 4.6 Sequence Diagram: Handling Native Code Crashes. 30
- 4.7 Google Breakpad Architecture. 32
- 4.8 Example of Error Activity Stylised as the Windows 10 BSoD. 37
- 4.9 Example of the *Crash Reporting Library* Dialog Window. 38
- 4.10 Example of the *Crash Reporting Library* Error Notification. 39
- 4.11 Example of the *Crash Reporting Library* Toast Message. 40
- 4.12 Sample Memory Heap Dump Processed in Eclipse MAT. 43
- 4.13 Class Diagram: Synchronous Providers. 44
- 4.14 Class Diagram: Asynchronous Providers. 45
- 4.15 Visualisation of the *BinPacker File Format*. 54
- 4.16 The BinUnPacker Batch Tool. 56
- 4.17 The BinUnPackerUI GUI Tool. 57
- 4.18 Class Diagram: Logging. 65
- 4.19 Example of a TSD Opened in TSD Debugger. 75
- 4.20 Example of a Test Run Configuration in Final-CI. 76
- 4.21 The *Crash Reporting Library* Artifactory Dossier. 80

1 Introduction

AVG Technologies is one of the leading security software companies targeting the global market. While traditionally being the most known for its PC products, AVG has launched its mobile application portfolio in 2010. In order to improve the quality of its products developed for mobile platforms, AVG has launched a series of research and development projects in order to improve the quality of its mobile application portfolio and to streamline and facilitate the development process. The goal of one of these projects was to research the means of reporting application errors and to provide a universal solution that would support all of AVG's products for the Android platform.

Since AVG had already owned an infrastructure for handling crash reports of desktop applications, the desired goal was to implement a solution utilising the aforementioned infrastructure but also, not to be entirely dependent on it in case of any changes in technology. The proposed solution was meant to also utilise existing build and automation framework available at AVG.

This thesis describes the entire development process of a custom crash reporting solution for Android applications within its five chapters. This chapter briefly describes the contents of this thesis.

Chapter 2 defines the problems this thesis aims to solve. Section 2.1 describes the nature of crashes on Android and how they manifest to the user. It also describes the differences between managed code and native code crashes. Section 2.2 describes the phases of the development cycle where crash reporting might be required and their specifics. The section also states the questions this thesis aims to answer.

Chapter 3 describes the technologies used during the development of the proposed solution. Section 3.1 describes the programming languages chosen for implementation and testing. Section 3.2 describes tools that aided the development process. And finally, Section 3.3 describes both open-source and proprietary libraries that are used by the proposed crash reporting solution.

Chapter 4 describes the actual implementation of the proposed solution. Section 4.1 starts the chapter with an analysis of crash reporting solutions available for Android applications. This section compares

their respective strengths and weaknesses and argues the point for developing a custom crash reporting solution.

Section 4.2 explains the basic architecture concept of the proposed solution, how the library is structured and its fundamental building blocks and terminology.

Section 4.3 describes the API for detecting and handling managed code crashes and the process of collecting crash report related metadata.

Section 4.4 explains the process of detection and handling native code crashes and the integration of the Google Breakpad library used for generating memory minidumps. The section also describes the differences between collecting metadata for managed code crashes and native code crashes.

Section 4.5 describes how the proposed solution replaces the default Android crash dialog window with a customisable user notification. The section provides illustrations of possible configurations and explains the difference in behaviour between different build configurations of the *Crash Reporting Library*.

Section 4.6 explains how the proposed solution retrieves data used for crash reporting and the difference between *synchronous* and *asynchronous data providers*.

Section 4.7 lists all of the configuration options available in the *Crash Reporting Library*, their respective default values and describes how its API can be utilised to override the default configuration.

Section 4.8 defines the custom light-weight *BinPacker File Format* that is used for crash reporting. The section also describes the capabilities of the support tools handling the custom file format and how to use them.

Section 4.9 explains how the *Crash Reporting Library* is integrated within AVG's existing infrastructure and how its API can be used in order to process crash reports in multiple ways. The Section also explains how the *Crash Reporting Library* can utilise different publishing strategies and when the crash report is considered successful or unsuccessful.

Section 4.10 describes how the *Crash Reporting Library* handles offline states and unsuitable connection types. The Section also introduces the Offline Publisher and explains its role within the *Crash Reporting Library*.

Section 4.11 describes the API that the *Crash Reporting Library* uses for logging and how this API can be used to integrate the *Crash Reporting Library* logging with the host application.

Section 4.12 describes the testing and automation strategy for the *Crash Reporting Library* and explains the process and environment of the integration testing.

Section 4.13 describes the build configurations of the *Crash Reporting Library* and the support applications. The section also lists Gradle tasks that can be used to produce one or more solution artefacts.

Section 4.14 defines the process of publishing and deployment the developed library into a repository owned and maintained by AVG.

And finally, Chapter 5 provides a brief summary of the achieved results, including author's personal contribution. The Chapter also lists the value provided to the industrial partner and highlights accomplishments.

2 The Problem Defined

This chapter defines the problem this thesis aims to solve and the technical parameters and constraints of any viable solution.

2.1 The Nature of Failures on Android

Programming is a human activity that is rather prone to errors. Even the best programmers in the world create faults which manifest themselves as failures throughout the different parts of the application life cycle.

The most severe types of failure in any environment are unrecoverable failures – simply called crashes. Probably the most common cause of such crashes on Android is the infamous `NullPointerException` [1], thrown each time a method is called on an uninitialised object variable. Such error immediately leads to an application crash and displaying of a dialog window (see Figure 2.1, left).

Starting with Android 6.0 (API level 23, codename Marshmallow), the operating system even contains a crash log that stores a list of crashed applications and the reasons for their respective crashes. When any application experiences multiple crashes, Android displays a slightly different dialog window (See Figure 2.1, right), which notifies the user the application is problematic and restarting it will not resolve the issue.

A completely different case is handling native code written in C or C++ and using Android NDK (see Section 3.2). Not only do crashes caused by native code not display any dialogs or other notifications of any kind, but Android does not provide any solution for handling native code crashes on its own.

2.2 Reporting Crashes

In order to detect and remedy a fault in the source code, a developer needs either a precise list of instructions as to how to induce the given failure or, provided that the failure manifests rather randomly, at least a set of data which would enable a comprehensive analysis. Such data

2. THE PROBLEM DEFINED

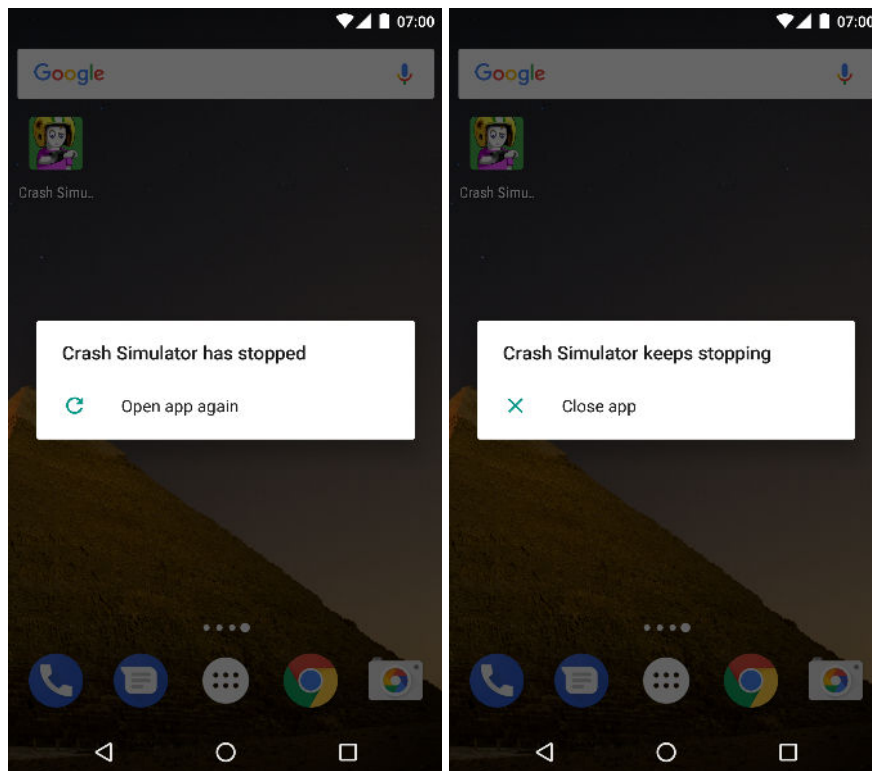


Figure 2.1: Standard Android 7.1.2 Crash Dialogs.

include application logs, detailed information concerning both the crashed application, its version and settings and detailed information concerning the device, since Android is particularly infamous for its fragmentation and vendor-specific quirks and bugs.

Android does provide its own crash reporting solution which, however, applies only to applications already published on the *Google Play* application marketplace. On top of that, each crash report requires an explicit user agreement via the click on a confirmation dialog window. This obviously limits the effectiveness of the solution and for many non-consumer facing applications (such as industrial, internal or technical demonstration applications, usually not published on *Google Play*) is simply not suitable.

Moreover, these crash reports cannot be personalised or customised in any way. This may prove unacceptable, as storing application data

on third party servers is unacceptable for many companies, including AVG. Another problem, besides the obvious vendor-locking and relying on a third party technology, is that Android's default solution is available only for Android applications. This would mean cumbersome crash management for companies and organisations maintaining a portfolio of applications available on multiple platforms.

This situation inevitably leads to the conclusion that any solution for the aforementioned challenges needs to scale both horizontally and vertically. Crashes encountered in different stages of the application life cycle require separate handling. On the one hand, production bugs need to be centrally analysed and categorised by their impact and severity.

On the other hand, pre-production versions of an application might require a different handling of the crash data and provide a quick and steady solution for sharing of the crash data within the development team.

This means that the best possible solution needs to be highly configurable for the different stages of the application life cycle, would not rely on any particular technology or back-end solution and respect the fundamental limitations of mobile devices such as rather lower computational power, battery power limitations and unstable network connection.

This thesis aims to answer the following questions:

1. What crash reporting solutions are currently available? What are their respective strengths and weaknesses? What potential do they have for an integration with other technologies?
2. What are the options for replacing the default crash notification dialog?
3. How can a modular architecture be utilised in order to achieve maximal configurability and further extensibility?
4. Which data formats and communication protocols and adapters are best suited for crash reporting of Android applications?
5. Which libraries, tools and APIs can be utilised for reporting crashes of both managed and native code?

3 Technologies Used

This chapter describes all of the programming languages, libraries, APIs, tools and other technologies used during the entire development of the Crash Reporting Library.

3.1 Programming Languages

Since all Android applications are essentially run in a virtual machine, all Android system APIs are provided for either Java or C/C++ native calls.

Nowadays there are several programming options for the development of Android applications such as Xamarin (C#) [2], Ruboto (Ruby) [3], NativeScript (JavaScript/TypeScript) [4] or Kotlin¹ [5]. However, these either rely on being compatible with Java, heavily utilise Android's WebView [6] component for rendering web pages or utilise cross-compilation to Java, Java bytecode and eventually Dalvik Executable Format.

In order to eliminate the dependency on third-party components, through an executive decision issued by the project supervisor, the developed library uses Java, C and C++ only, since these languages (along with Kotlin) are officially supported by the development kits for Android.

Java Standard Edition. Java is a language originally developed by Sun Microsystems and currently maintained by Oracle. Android's APIs were until recently based upon the open-source Apache Harmony [7] implementation. However, with the version 7.0 (API level 24, codename Nougat), Android has switched to Oracle's OpenJDK [8, 9] implementation.

The original Harmony-based implementation supported the Java SE API up to the version 6.0. In October 2013, Google added support for the Java 7.0 language features using Build Tools version 19 or newer, Android Studio version 0.3.2 or newer and Android Gradle Plug-in version 0.6.1 or newer [10].

1. Kotlin is a JVM-compatible language actively developed by JetBrains.

3. TECHNOLOGIES USED

Most features such as the diamond operator, multi-catch and strings in switches were available for all API levels, while try-with-resources is available for Android version 4.4 (API level 19, codename KitKat) or higher.

Android currently also supports a subset of the Java 8.0 features and APIs, using the new Jack toolchain and Android Studio version 2.1 or newer. Some features like the lambda expressions, method referencing and type annotations are available at all API levels. Other features such as default and static method implementations and repeatable annotations are available from API level 24 onwards [11].

In March 2017, Google announced that the Jack toolchain is about to be deprecated and the Java 8.0 language features are to be supported directly by the `javac` and `dx` compilers, which are included in the Android SDK [12] (see Section 3.2). The latest stable release of Android Studio supports the Java 8.0 language features using the updated compilers.

The Android Java APIs do not completely match standard Java APIs, most notably replacing the `awt` package with its own GUI libraries located in packages `android.view` and `android.widget` and removing the `rmi` package without a direct replacement. Conversely, Android Java APIs contain several interesting extensions such as security extensions and OpenGL API [13, p. 96-97].

Java is used as the main language of choice for the library, the demonstration application, the test/demonstration server implementation and the support tools.

Python [14] is a popular open-source scripting language originally developed by Guido van Rossum and first released in 1991. Python is used in many environments ranging from configuration management, server back-end development and testing automation.

Python currently has two major stable versions: 2.7.14 and 3.6.3. While Python 3 was intended as a replacement for Python 2, due to syntax incompatibility, many projects still utilise Python 2.7 because of its extensive library support. Python 2.7 is used as the language of choice in the component and integration test automation.

Bash (Bourne Again SHell) [15] is a scripting language that primarily serves as the shell for Unix-like operating systems. Bash was originally developed by Brian Fox for the GNU project as a replace-

ment for the Bourne shell, first released in 1989. Bash upholds the POSIX standard for OS shell and provides various extensions.

Bash is used in the automated build verification process, where it runs the unit tests for build verification and as a language of choice for Google Breakpad extension's build script.

3.2 Tools

Android provides two standard APIs for application development, the Android API for Java and Kotlin and the Android Native API for C and/or C++.

Android SDK (Android Software Development Kit) [16] is a collection of libraries, compilers, preprocessors and tools which allow the development of applications for Android using the standard Java Android APIs.

Android uses a different bytecode than Oracle-compatible VMs, with compiled classes usually denoted by the `.dex` file extension. The original (now called legacy) toolchain would compile Java source files into Java bytecode classes and then transcompile them using the `dx` tool.

The newer Jack toolchain produces `.dex` files directly. A complementary library tool called Jill (Jack Intermediate Library Linker) can be used to transcompile standard Java libraries distributed in the `.jar` format [11]. The new Java 8.0 compatible compiler uses the same approach as the legacy compiler.

There are two virtual machines for Android: the original Dalvik Virtual Machine and its replacement ART (Android Runtime). Dalvik originally interpreted the bytecode however, Android version 2.2 (API level 8, codename Froyo) introduced the JIT (Just-in-Time) compilation for special cacheable parts of the code called traces [17].

ART was introduced as an experimental feature in Android 4.4 (API level 19, codename KitKat) and ultimately replaced Dalvik as the default virtual machine in Android 5.0 (API level 21, codename Lollipop). ART uses the AoT (Ahead of Time) compilation. The bytecode gets compiled during the application installation, thus completely eliminating Dalvik's interpretation and JIT trace-based compilation. This led to an optimisation during runtime, a better performance and

3. TECHNOLOGIES USED

a lower battery usage while prolonging the application installation time for obvious reasons.

Android NDK (Android Native Development Kit) [18] contains a gcc toolchain optimised for mobile processors, an implementation of the C++ Standard Library and header files for accessing Android native APIs.

TSD Debugger is a proprietary tool developed by AVG for launching, editing and debugging automated tests. TSD Debugger utilises a client-server architecture which allows for launching automated tests on both the development machine and remote virtual machines. TSD Debugger is implemented in C#/Mono and is available for Windows and macOS.

TSD Debugger utilises the special TSD (Test Definition) format as a domain-specific language for tests, usually using the `.tsd` file extension. TSDs do not use one-time scripts like other testing approaches do, but a list of reusable steps instead. The data format is based on the JSON data format [19].

Final-CI is a proprietary continuous integration system developed by AVG. Final-CI utilises Atlassian Bamboo [20] as its build server and provides a web-based interface for running build plans, unit tests (including JUnit tests for Java) and component/integration tests using AVG's proprietary TSD format (see above). Final-CI also supports various post-build triggers and hooks including automated deployment and release management.

Final-CI is the solution of choice for automating the build process and running automated build verifications.

3.3 Libraries

This section describes all of the libraries used in the development process of the Crash Reporting Library.

Android Support Library [21] is a backport of features introduced in newer versions of the Android SDK for the older ones. This allows the usage of newly developed components and APIs even on older devices. Android Support Library is used in the demonstration application.

JUnit [22] is one of the most popular frameworks for the development, running and configuration management of unit tests written in Java. The latest stable version is 4.12. A new overhauled version 5.0 targeted for and using new features of Java 8.0 was released in September 2017 [22]. All unit tests of the *Crash Reporting Library* and its support tools were written using JUnit 4.

Mockito [23] is an open-source testing framework for Java. Android does support local unit tests² using the `android.jar` library included in the Android SDK. However, all of the classes contained in the `android.jar` are mere stubs³.

In order to implement a locally run unit test, one does need to provide a custom test implementation of all of the dependency classes located in the Android SDK. This is usually achieved either through inheritance of classes and interfaces or, preferably by using a mocking library. Mockito is used in unit tests of the *Crash Reporting Library*.

PowerMock [25] is Java framework dedicated to solving common testing problems, obstacles and pitfalls. PowerMock provides two API extensions: one for Mockito (called PowerMockito) and one for EasyMock.

PowerMock allows for mocking and testing implementation even of `final` and `static` methods and classes by manipulating the code on the VM/bytecode level and hence simulating otherwise unreachable states required by unit testing. PowerMockito is used in unit tests of the *Crash Reporting Library*.

Google Breakpad [26] is an open-source library developed by Google used for handling native code crashes and generating memory dumps. Google Breakpad is usable on several platforms including Microsoft Windows, Linux, macOS and Android amongst others.

A successor library named Google Crashpad is currently under development. However, Crashpad does not support Android at the moment [27]. Google Breakpad is used in the native code crash reporting loop within the *Crash Reporting Library*.

Appium [28] is an open-source tool for test automation of native, web-based and hybrid applications for iOS and Android platforms.

2. Meaning the tests are run on a development machine not on the actual handset, tablet or other device with Android.

3. Blank implementations formally returning default values and throwing `RuntimeException` [24] upon each call of any method from the archive.

3. TECHNOLOGIES USED

Appium itself is based on Selenium [29], a suite of tools for automating web browsers.

Selenium and Appium utilise the client-server architecture using REST API calls. While the server is implemented in NodeJS, Appium provides multiple clients for various programming languages including Java, Ruby, Python and C# amongst others.

The *Crash Reporting Library* uses the Python client. Appium is used in the testing automation of the *Crash Reporting Library*.

Autolib is a proprietary library developed by AVG for testing automation of mobile applications. Autolib has two implementations – the Python one and the PowerShell one.

The *Crash Reporting Library* uses the Python implementation in cooperation with the Appium Python client. Autolib's source code is not disclosed in this thesis, in compliance with AVG's specific instruction.

NanoHTTPD [30] is a light-weight, embeddable implementation of a HTTP server written in and usable from Java. NanoHTTPD is used as both the demonstration crash reporting server implementation and in the testing automation.

4 Implementation

This chapter defines the process of implementation from the initial analysis of existing solutions to the testing and automation process.

4.1 Analysis of Existing Solutions

This section describes several existing solutions for crash reporting of Android applications and their respective strengths and shortcomings. The analysis focused on the following features of the existing solutions:

- comprehensive and easy configurability,
- forms of user notification,
- further extensibility,
- support for various back-end solutions and
- support for native code crashes.

4.1.1 ACRA

Application Crash Reports for Android (ACRA) is an open-source library developed by Kevin Gaudin, enabling Android applications to automatically post their crash reports to a report server [31]. In February 2016, ACRA was used in 2.68% of applications available on *Google Play*. The latest version is 4.10.0, released in June 2017. ACRA's source code was released under the Apache License 2.0 license.

ACRA requires the `android.permission.INTERNET` permission which is automatically granted on devices with Android 6.0 or newer and granted upon installation on devices with older versions of Android.

ACRA's architecture utilises a handler service running in its own process. This allows to terminate the crashing process and immediately send the crash report to a server.

The service is meant to be started within the `android.app.Application#attachBaseContext(Context)` method.

There are two ways how one can initialise and pass configuration to ACRA. The first one is by adding the `@ReportCrashes` annotation located in the `org.acra.annotation` package to the `Application` class and initialising ACRA via the `org.acra.ACRA.init(Application)` method call. The other way uses a configuration builder class and initialises the library by calling the `org.acra.ACRA.init(Application, ConfigurationBuilder)` method.

There are several back-end implementations for ACRA written in various languages such as Java, Ruby and Go amongst others [32]. ACRA prescribes a standard behaviour to which a server must comply and provides a referential implementation named `Acralyzer` [33].

On top of that, ACRA allows for custom publishing mechanisms to support custom and third party servers (see Subsection 4.1.2).

ACRA does not support crash reporting for native code crashes. A third party adapter called `Acra-breakpad` [34] exists, using a modified version of ACRA [35]. However, `Acra-breakpad` seems outdated and no longer actively developed as the last update was released on 15 March 2014.

ACRA can also suppress the default dialog window (see Figure 2.1). ACRA supports four different modes for user notification whenever an application crash is detected: `DIALOG` (displaying a customisable dialog window, see Figure 4.1), `NOTIFICATION` (displaying a system notification in the status bar), `TOAST` (displaying a Toast message [36]) and the so-called `SILENT` mode, which does not display any user notification.

4.1.2 HockeyApp

HockeyApp is a crash reporting solution originally developed by BitStadium. In December 2014, HockeyApp was acquired by Microsoft [37]. The source code for HockeyApp SDK was released under the Apache License 2.0 license.

HockeyApp is partially compatible with ACRA in a sense that the ACRA client can report to the HockeyApp server [38] using the provided `HockeySender` classes.

Customisation is possible by inheriting and extending the `CrashManagerListener` and `UpdateManagerListener` classes located in the `net.hockeyapp.android` package.

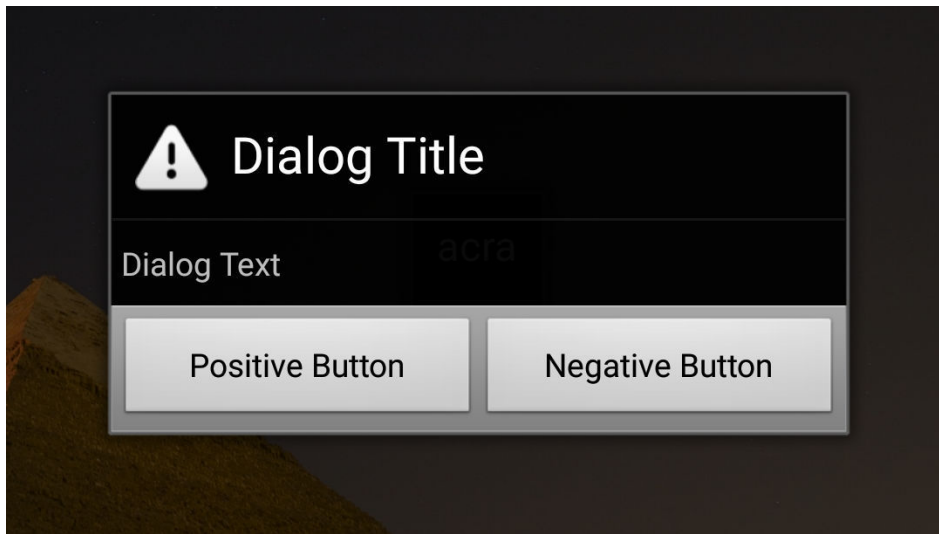


Figure 4.1: Example of ACRA's Dialog Window.

HockeyApp has a preliminary (early access) support for the Android NDK and crashes reported from the native code. HockeyApp uses Google Breakpad for generating native code memory dumps (see Sections 3.3 and 4.4).

HockeyApp uses its own proprietary server solution (see Figure 4.3). The client and server are synchronised using an API key stored in the host application manifest.

HockeyApp replaces the default dialog window with its own (see Figure 4.2). Instead of displaying the dialog and sending the crash report immediately after the crash occurrence, the dialog is displayed upon restarting the host application.

HockeyApp requires three system permissions. The `android.permission.ACCESS_NETWORK_STATE` and the `android.permission.INTERNET` are required for all devices, while the `android.permission.WRITE_EXTERNAL_STORAGE` is required only for devices with API level 18 (version number 4.1 – 4.3.1, codename Jelly Bean) or lower.

4.1.3 Crashlytics

Crashlytics is a proprietary SDK originally developed by a company of the same name. In January 2013, Twitter, Inc. acquired Crashlytics

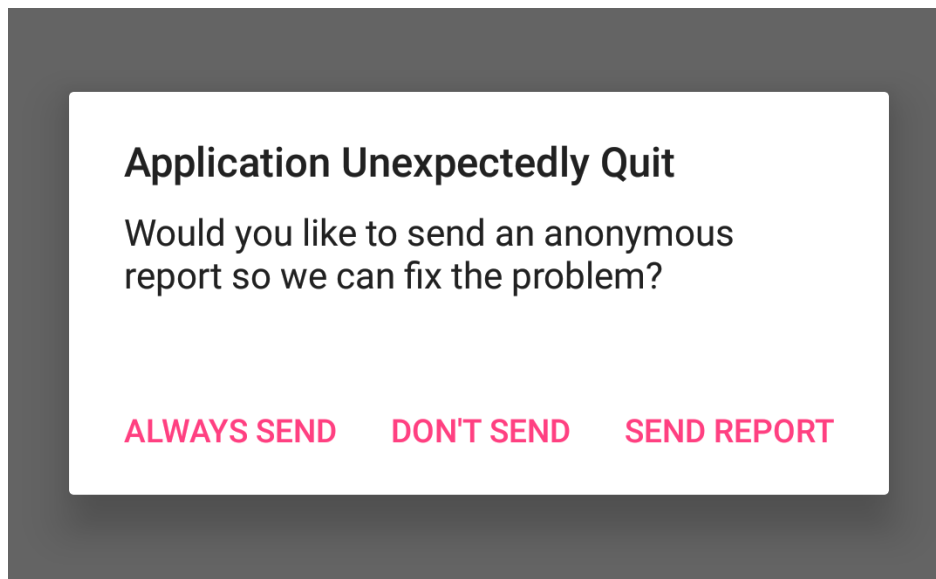


Figure 4.2: HockeyApp's Dialog Window.

[39] and maintained its development tools until January 2017, when it was acquired by Google who currently plan to merge Crashlytics with their Firebase initiative [40].

Crashlytics supports Android crash reporting since May 2013 [41]. In May 2015, a support for native code crashes was announced. In October 2014, Crashlytics became a part of Fabric – a larger portfolio of mobile application analytic, user identity and authentication tools [40].

Crashlytics supports Android 2.2 (API level 8, codename Froyo) or newer. The NDK crash reporting supports all NDK architectures. Crashlytics requires a single system permission, the `android.permission.INTERNET`.

Crashlytics supports Ant, Maven and Gradle build systems [42]. Gradle projects are required to use a specialised Gradle plug-in that processes the Crashlytics settings.

Similarly to HockeyApp, Crashlytics uses a proprietary back-end server solution accessible via an API key stored in the Android manifest. The crash reports can be viewed using the Fabric Crashlytics Dashboard (see Figure 4.4).

The screenshot shows the HockeyApp Server interface for crash reports. At the top, there are navigation tabs: App, Overview, Versions (6), Crashes (38), Events, Feedback (1), and Users (3). Below the tabs, there are buttons for 'Upload Crash' and 'Manage Crashes'. A search bar and filters (10, 25, 50) are also present. The main content area shows a list of crash groups with the following columns: Status, Count, Users, Description, Last Crash, and Version. The status of each crash group is 'open'. The description column contains the crash type and the specific line of code where the crash occurred. The last crash column shows the date and time of the most recent crash. The version column shows the version of the app that crashed.

Status	Count	Users	Description	Last Crash	Version
open	10	0	presentation.PresentationActivity.reloadItemViews line 289 java.lang.RuntimeException: Unable to start activity ComponentInfo{cor	13 Dec 2014, 10:03	1.2.3 (35)
open	6	0	forms.FormFragment.createItem line 1547 java.lang.NullPointerException	21 Dec 2014, 10:31	1.2.6 (36)
open	2	0	goods.CategoryFragment.loadCategories line 59 java.lang.NullPointerException	06 Jan 2015, 14:45	1.2.6 (36)
open	3	0	goods.ItemDetailFragment.onAddItem line 434 java.lang.NullPointerException	18 Nov 2014, 13:08	2 versions
open	2	0	OrderDetailFragment\$DownloadDetailTask.<init> line 57 java.lang.RuntimeException: Unable to start activity ComponentInfo{cor	11 Dec 2014, 13:22	1.2.3 (35)
open	2	0	android.widget.ListView.layoutChildren line 1555 java.lang.IllegalStateException: The content of the adapter has changed	07 Dec 2014, 13:55	1.2.3 (35)
open	1	0	forms.FormFragment.createItem line 1539 java.lang.NullPointerException	10 Dec 2014, 14:32	1.2.6 (36)
open	1	0	android.widget.ListView.layoutChildren line 1555 java.lang.IllegalStateException: The content of the adapter has changed	14 Dec 2014, 12:16	1.2.6 (36)
open	1	0	forms.FormFragment.validateTab line 539 java.lang.NullPointerException	20 Nov 2014, 14:29	1.2.3 (35)
open	2	0	android.view.MotionEvent.nativeGetAxisValue java.lang.IllegalArgumentException: pointerIndex out of range	21 Dec 2014, 19:34	2 versions

Figure 4.3: Example of HockeyApp Server Crash Reports.

Crashlytics does not override the default crash dialog window (see Figure 2.1).

Conclusion

All three of the aforementioned solutions lack the desired universality. While ACRA provides the desired level of configurability and technological independence of any particular back-end solution, it does not handle native code crashes. Conversely, HockeyApp and Crashlytics do support native code crash reporting but lack the configurability and require their respective proprietary servers. To change the default behaviour would mean significantly modifying the source code which cannot be done for Crashlytics since its source code is undisclosed.

HockeyApp's crash reporting after an application restart is an impractical solution for automated integration tests and maintaining

4. IMPLEMENTATION

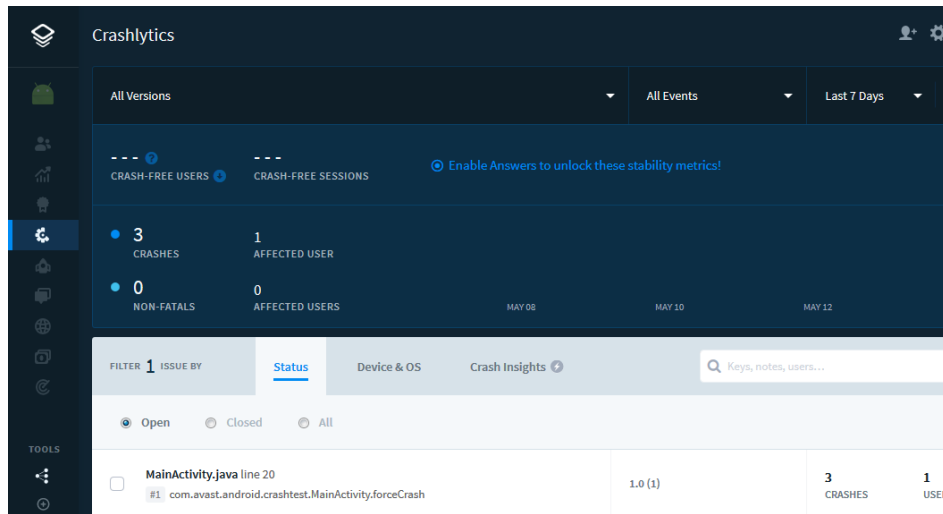


Figure 4.4: Example of Fabric Crashlytics Dashboard.

multiple callbacks in a designated Activity seems cumbersome and unwieldy.

Out of the three, the most viable concept to build upon and extend is ACRA. Unlike the other solutions, it lacks the support for Android NDK. However, ACRA's basic architecture seems to be very effective, robust and stable to extend.

ACRA, however, also prescribes its own communication protocol between the client and the reporting server. This is not acceptable for the proposed solution since AVG wanted to maintain full control over the communication protocol (due to its existing crash reporting server solution already utilised by AVG applications for Microsoft Windows and macOS). Attempting to modify ACRA would therefore mean to apply extensive changes to the source code. Moreover, since ACRA is still actively developed, maintaining both independently developed codebases would most likely be quite difficult, case in point being the abandoned development of Acra-breakpad.

Both HockeyApp and Crashlytics rely on their respective third-party servers. These servers are unsuitable for the proposed solution. Both also utilise the Google Breakpad library, therefore it has been chosen as a technology of choice for the native code crash reporting.

Because of the aforementioned reasons, the solution proposed in this thesis is a standalone library rather than a mere extension of any of the existing solutions, in order to best suit the demands of AVG and other parties who would choose to use the library.

The proposed solution is hereinafter referred to as the *Crash Reporting Library*.

4.2 Architecture Concept

Android API provides a way of developing multi-process applications in a form of a service running in its own process. Such services need to have their process name specified in the Android manifest using the `android:process` XML attribute [43]. Private processes restricted to the given application may be created by placing a leading colon character in its process name (i.e. `:service`).

The crash detection has two entry points. Managed code crashes are handled by `CrashUncaughtExceptionHandler` registered as the default `UncaughtExceptionHandler` (described in more detail in Subsection 4.3). The other entry point is the Google Breakpad library.

The library operates in multiple processes, the processes of the host application (usually just a single one) and the dedicated process of the handler service (implemented in `com.avg.zaap.crashlibrary.CrashReportingService`). The `CrashReportingService` serves as the centrepiece for handling both managed code and native code crashes.

Upon a managed code crash, the library collects the process specific crash report data using the so-called *synchronous providers* (see Section 4.6). *Synchronous providers* are called serially, one at a time. Once the data collection has finished, the `CrashReportingService` is bound and a new managed code crash is reported.

Native code crashes are managed by Google Breakpad which creates a memory dump and consequently terminates the host process whenever a native code crash is detected. A dedicated class named `FileObserver` registered by the `CrashReportingService` automatically detects newly created files in the designated Google Breakpad dump directory. Upon detecting a new file, it binds to the `CrashReportingService` and reports a new native code crash. Native code crash reporting is described in more detail in Section 4.4.

Once a new (native or managed code) crash is reported, the `CrashReportingService` notifies the `CrashReporter` class which handles most of the application logic. The `CrashReporter` collects additional data that is not necessarily related to the crashed host applications process. Unlike the *synchronous providers*, the *asynchronous providers* are called in parallel and communicate asynchronously via interface callbacks. Both *synchronous* and *asynchronous providers* are described in more detail in Section 4.6.

The collected data (from *synchronous providers*, *asynchronous providers* and/or Google Breakpad) is compiled into a single file using a proprietary light-weight algorithm and data format. The compilation is handled by the `BinPacker` class. The data format and related tools are described in Section 4.8.

Finally, the crash report data file is processed using one or more *data publishers*. The available publishing strategies as well as the default publisher class is described in more detail in Section 4.9.

The crash report data publishing might also be postponed when the device does not have a suitable Internet connection (meaning either no connection or connection deemed unsuitable for crash reports). The handling of the connection and offline states is described in more detail in Section 4.10.

The *Crash Reporting Library* prescribes a single initialisation point within a descendant of the Android `android.App.Application` class. The library can be initialised within the `attachBaseContext(Context)` method or within the `onCreate()` method by using either the `init(Context)` or the `init(Context, CrashLibConfig)` method of the `com.avg.zaap.crashlibrary.CrashLib` class.

Upon initialisation, the *Crash Reporting Library* loads its configuration. There are two ways of passing configuration options to the library, either by using the `@CrashLibConfigAnnotations` annotation or the `CrashLibConfig.Builder` class. The configuration allows to add custom extensions and modify the default behaviour of the *Crash Reporting Library*. Configuration management is described in more detail in Section 4.7.

The *Crash Reporting Library* requires three system permissions: `android.permission.ACCESS_NETWORK_STATE`, `android.permission.INTERNET` and `android.permission.WRITE_EXTERNAL_STORAGE`.

4.3 Handling Managed Code Crashes

Unexpected behaviour of any program written in Java is handled by throwing an instance of an object implementing the `java.lang.Throwable` interface, commonly referred to as exceptions. Java SE API distinguishes three kinds of exceptions – checked exceptions (inheriting the `java.lang.Exception` class), runtime exceptions (inheriting the `java.lang.RuntimeException` class) and errors (inheriting the `java.lang.Error` class) [44].

Checked exceptions need to be explicitly handled by putting them into a `try-catch` block or by declaring the exception as thrown by the method, using the keyword `throws` in the method signature. In that case, the method needs to be handled by the caller method in the same way. The compliance to this rule is enforced by the syntax validator during compilation.

Errors indicate serious problems with an application that a reasonable application should not try to catch. Most of these errors are abnormal conditions that should never occur [45] and might signalise an issue within the virtual machine.

Java SE API defines a handler in the `Thread.UncaughtExceptionHandler` class located in the `java.lang` package. A dedicated thread handler can be assigned to each thread. In case an uncaught exception handler is not explicitly assigned, a default one is used. The default handler can be set using the `Thread#setDefaultUncaughtExceptionHandler` method.

As it is common among other GUI-based application frameworks, Android uses one dedicated thread for GUI operations, denoted in the VM as `main` or commonly referred to as the *UI thread*. All other threads are commonly called *background threads*.

When a *background thread* crashes, the thread gets terminated by the uncaught exception. Java's default implementation of the `UncaughtExceptionHandler` class terminates the host process and writes a stacktrace to the standard error output. However, terminating the application is not a must. When the default `UncaughtExceptionHandler` is replaced by a custom one, it is possible to terminate only the crashing thread without any effects to any other ones. The normal operation after the `uncaughtException(Thread,`

4. IMPLEMENTATION

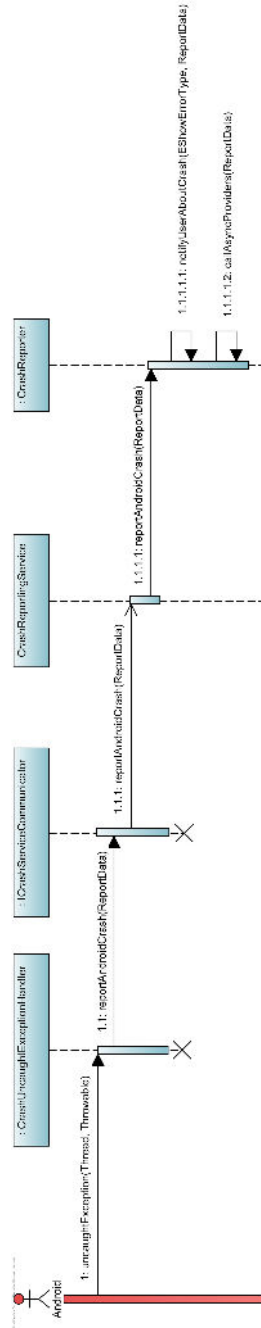


Figure 4.5: Sequence Diagram: Handling Managed Code Crashes.

Throwable) method call resumes and the application may continue its run.

Conversely, when the *UI thread* crashes, it is completely unrecoverable since the *UI thread* handling the GUI loop gets terminated. This leads to completely unresponsive UI and ultimately forces the developer to terminate the entire process using the `android.os.Process.killProcess(int)` method.

However, discarding a *background thread* silently might prove, in fact, counter-productive since Android has very strict rules regarding blocking the *UI thread* and therefore most time-consuming operations are required to be handled asynchronously through *background threads*.

Discarding a *background thread* without notifying the appropriate listeners, observers and other classes of similar semantics could possibly break important application logic or require significant editing of the application code (implementing fallback and time out mechanisms, etc.).

Therefore, a common approach for both the *main thread* and *background threads* was chosen. The *Crash Reporting Library* terminates the failing thread's process regardless of whether the crash happened within the *main thread* or any of the *background threads*.

The *Crash Reporting Library* implements a custom exception handler in the `CrashUncaughtExceptionHandler` class within the `com.avg.zaap.crashlibrary.core` package. The handler is set as the default exception handler for all of the host application processes except the `CrashReportingService`'s one. This avoids endless recursion in case the `CrashReportingService` itself would crash.

Whenever a managed code crash is detected in the host application, the `CrashUncaughtExceptionHandler` collects all the data from the *synchronous data providers* (including the stacktrace writers, see Section 4.6) and required metadata (see Subsection 4.3.1).

After data collecting, the `CrashUncaughtExceptionHandler` binds to the `CrashReportingService` that runs in its own dedicated process.

Android supports interprocess communication using the AIDL (Android Interface Definition Language). AIDL allows to message other processes running in the system and to pass data to them. Such data can be either primitive data types or objects that implement either the `java.io.Serializable` interface or the Android replacement for `Serializable`, the `android.os.Parcelable` interface.

The crash report data and metadata are stored in the `com.avg.zaap.crashlibrary.ReportData` class that implements the `android.os.Parcelable` interface. The class is mutable and serves further the data structure for other processing classes, such as `CrashReporter`, the *asynchronous providers* and publishers amongst others (see Sections 4.6, 4.9 and 4.10).

Finally, the host process gets terminated by calling the `android.os.Process.killProcess(int)` method and the crash report files are handled further by the `CrashReportingService`.

4.3.1 Collecting Metadata

Besides the crash report data collected by the configured *synchronous providers* (see Section 4.6), the `CrashUncaughtExceptionHandler` also collects metadata. The metadata can be divided into two distinctive categories:

- internal metadata and
- identification metadata.

The **internal metadata** are collected in order to identify identical crashes within the given device and to store human-readable name, description and identifier of the crash. Internal metadata are also used to keep track of crash report temporary files. Internal metadata are not meant to be published by any of the configured *publishers* (see Section 4.9) however, they might be shown to the user in the debug build of the *Crash Reporting Library*.

The **identification metadata** are collected in order to enable identification and processing for crash reports on a crash reporting server. Identification data are meant for the configured crash *publishers* in order to help identify, group and categorise similar or identical crashes.

Both categories of metadata are stored within the `com.avg.zaap.crashlibrary.ReportData` class. The `ReportData` class implements the `Parcelable` interface in order to allow passing metadata between application processes using the AIDL interface (see Section 4.2). Metadata are handled by the `CrashReportingService`.

Metadata of native code crashes are collected in a similar way by the `com.avg.zaap.crashlibrary.CrashReporter` class (see Subsection 4.4.3).

Internal Metadata

This subsection describes the internal metadata collected by the *Crash Reporting Library* and stored within the `ReportData` class.

Crash Code is an internal unique identifier of the detected crash. The value is calculated as a SHA-1 hash of the crashed thread's stack trace using the `generateHash(StackTraceElement[])` method of the `com.avg.zaap.crashlibrary.core.hash.HashGenerator` class. The Crash code value is used in order to determine the name of the compiled crash report file (see Sections 4.9 and 4.10).

Error Name stores the name of the uncaught exception. The value is retrieved by getting the simple class name of the `Throwable` instance using the `java.lang.Class#getSimpleName()` method.

Error Message stores the short error message provided by the uncaught exception. The value is retrieved using the `java.lang.Throwable#getMessage()` method. This value might be null as it is not provided by all instances of `Throwable` (i.e. `java.lang.NullPointerException`).

Error Details stores the detailed information about the crash. The value is retrieved using the `getErrorDetails(Throwable)` method of the `CrashUncaughtExceptionHandlerHelper` class located in the `com.avg.zaap.crashlibrary.core` package.

Files to Pack stores all of the files produced by the configured *synchronous data providers* that are to be compiled into the crash report file. Since the `ReportData` class is mutable, the collection is modified when the *asynchronous providers* are called to provide their respective data (see Section 4.6).

Identification Metadata

While the *Crash Reporting Library* aims to support multiple crash reporting server solutions, the main goal is to integrate it within AVG's existing crash reporting infrastructure.

The proprietary AVG back-end solution requires a particular set of data to analyse and process the crash. This subsection describes the metadata collected for identification, classification and grouping of crash reports.

Exception Code stores the unique crash identifier in the form of a 10 characters long hexadecimal string (i.e. 0xabcdef01). This value is extracted from the first 8 characters of the Crash Code (see above) and adding an 0x prefix.

Faulting Offset stores the source code line number of the top element in the stacktrace. The value is retrieved using the `java.lang.StackTraceElement#getLineNumber()` method.

Module Name stores the uncaught exception's cause in the following format: `<file name>:<class name>:<method name>`. In most cases the file name (omitting the file extension) is identical to the class name. However, since the cause of the crash might also be an ambiguously named inner class, the file name is always used as a qualifier. This data is collected using the `getFileName()`, `getClassName()` and `getMethodName()` methods of the `java.lang.StackTraceElement` class.

Module Version stores the application version set in the *Crash Reporting Library* configuration (see Section 4.7).

Process Name stores the current process name. The value is retrieved by calling the `getProcessName(Context)` method of the `ProcessHelper` class located in the `com.avg.zaap.crashlibrary.core.helper.process` package. This value is particularly useful in multiprocess applications. The default value for a single-process application is `main`.

Process Version stores the *Crash Reporting Library* version code available through the `com.avg.zaap.crashlibrary.BuildConfig.VERSION_NAME` static field automatically generated by Gradle (see Section 4.13).

GUID stores a UUID identifier randomly generated by the `java.util.UUID.randomUUID()` method. The UUID helps to identify each occurrence of a particular crash type.

4.4 Handling Native Code Crashes

Java provides an API for calling native code and passing data between native and managed code called JNI (Java Native Interface). Native code methods are declared without an implementation (similarly to

abstract or interface methods) and by using the native keyword in the method signature.

JNI requires the implementation to be contained within a C source file containing the JNI functions [46]. A developer can either write the functions manually or have them generated by Android Studio 2.2.0 or newer [47].

Native code can also be attached to project files using the shared libraries (commonly denoted by the `.so` file extension). Shared libraries can be loaded using either the `java.lang.System.loadLibrary(String)` method or the `java.lang.System.load(String)` method [48].

From a system perspective, the native code is treated in the same way as the managed code. A case in point being that native code still has to comply with the Android permissions system. Therefore, native code can only write and modify files if the application has been granted the appropriate permissions.

C++ has a very similar exception handling mechanism as Java. However, uncaught exceptions are translated to signals which under normal conditions lead to process termination. Native code written in C uses only signals, since the C syntax does not support exceptions.

Native code crashes cannot be handled in Java due to the nature of the JNI interface. When calling a native method, the control is given to the native code assembly and the managed code run is resumed once the native code successfully returns or ends.

The GNU C Library defines an API for handling signals in a dedicated header file named `signal.h` [49] which is also supported by Android. A developer can configure a custom signal action handler using the `sigaction` structure and `sigaction(int, const struct sigaction*, struct sigaction*)` function specifying a pointer to a function that gets called upon receiving a specified signal. Whilst theoretically, the signal handler can remedy the cause of the crash (by modifying the memory values causing the crash) and enabling the program to continue. However, this most likely is not a viable option for most real-world applications due to their complexity.

Since the native code cannot continue by processing the next machine code instruction, the process needs to be terminated. This limits the options of processing crashes by the *Crash Reporting Library* to low-level multi-process options. In UNIX-like systems these include

4. IMPLEMENTATION

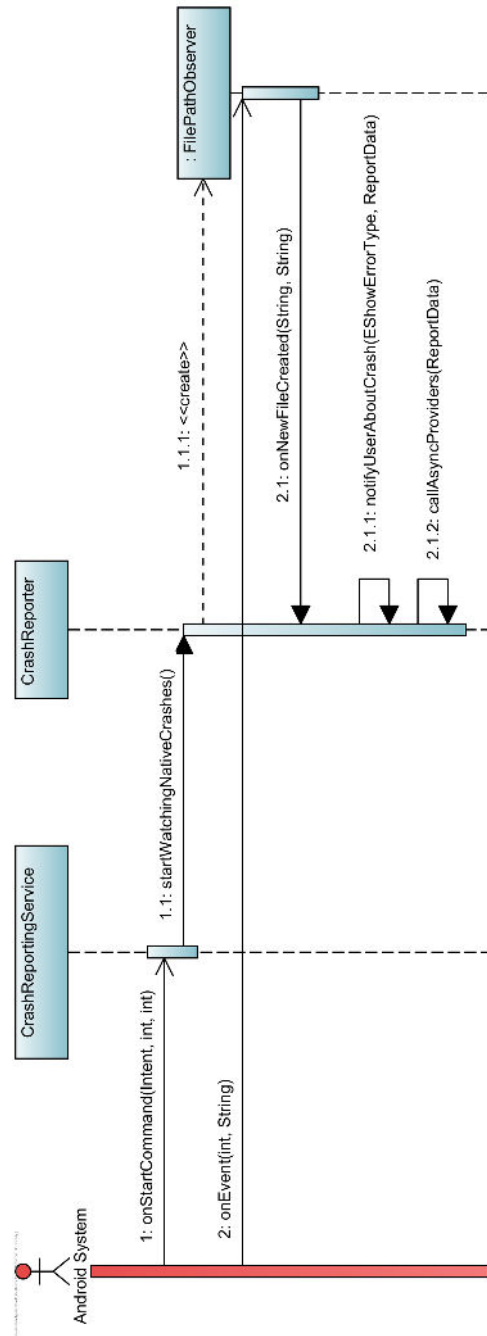


Figure 4.6: Sequence Diagram: Handling Native Code Crashes.

communication using a file, communication using the environment variables and communication using process names. In the case of crash reporting, the communication using a file is used, as it is the most suitable for the designated purpose.

Instead of declaring a custom signal handler, the *Crash Reporting Library* utilises the open-source Google Breakpad library (see Subsection 4.4.1).

The *Crash Reporting Library* targets the NDK Platform level 19 in order to support even devices with Android 4.4 (API level 19, codename KitKat) or lower [50].

4.4.1 Google Breakpad Integration

The source code of Google Breakpad is available for download at the Google Open Source repository [51] or its GitHub clone.

The Android version of Google Breakpad requires Linux system calls that are defined in the `linux_syscall_support.h` header file. The file itself is not provided in the project repository however, the Chromium project provides information about the header file, its differences from the standard C library and the ways of possible integration [53].

Google Breakpad registers a custom signal handler for detecting native code crashes. Upon detecting a crash, Google Breakpad creates a memory minidump file in the specified directory. The minidump format is similar to core files and was originally created by Microsoft. The format allows to traverse the dump and reconstruct a stacktrace of the crashed process. Each crash file is named using the UUID pattern and the `.dump` file extension. The contents of the file can be processed by providing an exported list of symbols of used libraries and processing both the list and the crash report by the Google Breakpad minidump processor (see Figure 4.7).

Breakpad provides a simple API that handles the signal receiver and attaches a minidump generator to the created signal handler. The library is initialised by calling a constructor of the `google_breakpad::ExceptionHandler` class (see Source code 1).

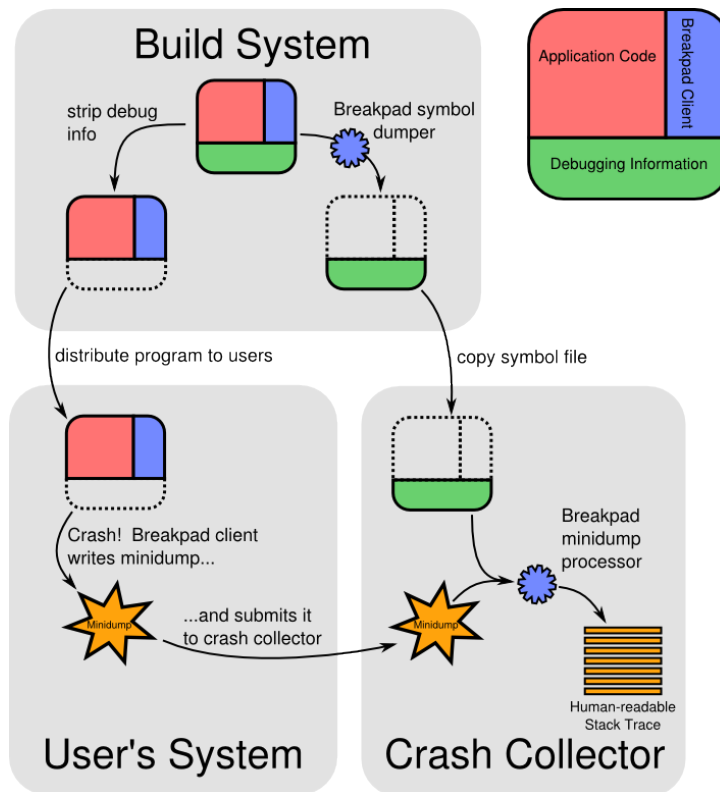


Figure 4.7: Google Breakpad Architecture.
[52]

Building Google Breakpad

Android NDK contains `gcc` toolchains specifically designed for compiling C/C++ source code for Android. NDK also allows to export a stand-alone version of the toolchain for selected architecture, NDK platform and ABI (Application Binary Interface) using the `make-standalone-toolchain.sh` included in the Android NDK (see Source code 2).

The *Crash Reporting Library* supports three 32-bit ABIs: `armeabi`¹ (compatible with ARM and ARM64 CPUs), `armeabi-v7a` (compatible with ARM v7 and ARM64 CPUs) and `x86` (compatible with 32-bit and

1. The `armeabi` ABI is now considered deprecated and support for it will be removed in an upcoming release of the Android NDK.


```

#include "client/linux/handler/exception_handler.h"
#include "android_breakpad.h"
using namespace google_breakpad;

void initializeBreakpad(const char* path) {
    mMinidumpDescriptor = new MinidumpDescriptor(path);
    mExceptionHandler = new ExceptionHandler(
        *mMinidumpDescriptor, NULL, dumpCallback, NULL,
        true, -1);
}

```

Source code 1: Initialising Google Breakpad.

```

$NDK/build/tools/make-standalone-toolchain.sh --arch=arm
--abis=armeabi --platform=android-19 --ndk-dir=$NDK
--install-dir=$HOME/toolchains/armeabi-platform19

```

Source code 2: Exporting a stand-alone toolchain for ARM ABI.

64-bit Intel CPUs). Google Breakpad for each supported platform can be built using the respective exported stand-alone toolchain.

Google Breakpad client can be configured by using the configure script and compiled using the make script (see Source code 3). Google Breakpad client is distributed as static library named `libbreakpad_client.a`. The *Crash Reporting Library* encapsulates the static library into a shared library named `libandroid_breakpad.so`.

```

./configure --host=arm-linux-androideabi
--disable-processor --disable-tools
make -j4

```

Source code 3: Building Google Breakpad Client.

Since JNI requires a C-compatible API, the `libandroid_breakpad` library provides a single function declared in the `android_breakpad.h` header file. Google Breakpad client is initialised by calling the `initializeBreakpad(const char*)` function.

The function argument is a C-string specifying the file path of a directory specified for creating the memory dumps. The build process of all three Google Breakpad shared libraries has been automated using the provided `build.sh` shell script.

4.4.2 Initialising the Native Code Handler

The *Crash Reporting Library* uses a so-called *native module*. Gradle normally compiles the *native modules* using the NDK toolchain and produces a shared library using the module name, prefix `lib` and `.so` file extension (i.e. a native module named `example` produces shared library named `libexample.so`).

The *Crash Reporting Library* overrides this process and defines two custom Gradle tasks instead. The `ndkBuild` task calls the compiler script (`ndk-build.cmd` on Windows or `ndk-build` on UNIX-like operating systems). The `ndkClean` task removes intermediate object files located in the `src/main/obj` directory.

Using custom tasks allows the usage of a custom `Android.mk` makefile instead of the default one generated by the NDK plug-in for Gradle. The `Android.mk` file specifies the name of the output library, in the case of the *Crash Reporting Library*, the module is named `crash` and the generated output file is named `libcrash.so` and packaged within the library's `.aar` file.

During the static initialisation of the `com.avg.zaap.crashlibrary.CrashLib` class, the native module shared library is loaded and initialised by calling the `initializeNativeLibrary(String)` native method.

Finally, the native code dump directory path is set during the `init(Context)` or `init(Context, CrashLibConfig)` method call of the `com.avg.zaap.crashlibrary.CrashLib` class.

4.4.3 Collecting Metadata

This subsection describes the difference between collecting metadata for managed code crashes (see subsection 4.3.1) and native code crashes.

The memory dump output file path is observed by the `com.avg.zaap.crashlibrary.file.observer.FilePathObserver` class. When-

ever a new memory dump file is detected, the `FilePathObserver` calls a callback method `onNewFileCreated(String, String)` of the `IFilePathObserverCallback` interface located in the `com.avg.zaap.crashlibrary.file.observer` package. The callback is handled by the `CrashReporter` class.

The metadata are collected by the `CrashReporter` class similarly as the `UncaughtExceptionHandler` does.

Internal Metadata

Error Name, **Error Message** and **Error Details** values are unused and left empty.

Crash Code is equal to the GUID generated by Google Breakpad (omitting the `.dump` file extension).

Files to Pack value contains only the Google Breakpad memory dump file path, since Google Breakpad generates only one file.

Identification Metadata

Exception Code is derived from the GUID generated by Google Breakpad and adding adding a `0x` prefix (i.e. the Crash code for `9c95312e-bf5d-4547-8036-616e85124b9f.dump` is `0x9c95312e`).

Module Version, **Process Version** and **GUID** are collected the same way as for the managed code crashes (see Subsection 4.3.1).

Faulting Offset, **Module Name** and **Process Name** values are set to constants. Faulting offset is set to 0 and both **Module Name** and **Process Name** are set to `native`, denoting a native code crash.

4.5 Notifying the User

One of the key requirements of the *Crash Reporting Library* is to replace the default Android crash dialog window (see Figure 2.1) with a customisable solution.

The *Crash Reporting Library* allows setting one of five possible user notification modes:

- `ERROR_SCREEN` – displays an Activity,
- `DIALOG` – displays a customisable dialog window,

4. IMPLEMENTATION

- NOTIFICATION – displays a system notification in the status bar,
- TOAST – displays a system Toast message,
- NONE – does not notify the user about crash.

The mode can be selected by setting the `showError` configuration value (see Section 4.7). The configuration values are contained within the `com.avg.zaap.crashlibrary.config.EShowErrorType` enumeration.

ERROR_SCREEN

When the `ERROR_SCREEN` configuration value is set, whenever the *Crash Reporting Library* detects a crash, it displays the `com.avg.zaap.crashlibrary.show.error.activity.ErrorActivity` and puts it on top of the navigation stack.

Since XML resources are merged between library and application projects, the layout can be customised by overriding the `activity_error.xml` layout file located in the `src/res/layout` directory.

While it is not enforced, the provided layout should contain the following Views:

- a `TextView` with the `crashlib_crashCode` ID for displaying the unique **Crash Code**,
- a `TextView` with the `crashlib_exceptionCode` ID for displaying the unique **Exception Code**,
- a `TextView` with the `crashlib_errorMessage` ID for displaying the **Error Message** and
- a `TextView` with the `crashlib_errorDetails` ID for displaying the **Error Details**.

Whenever any of these `TextViews` is not present in the provided layout, the library logs an error and does not display the corresponding error information or specified message.

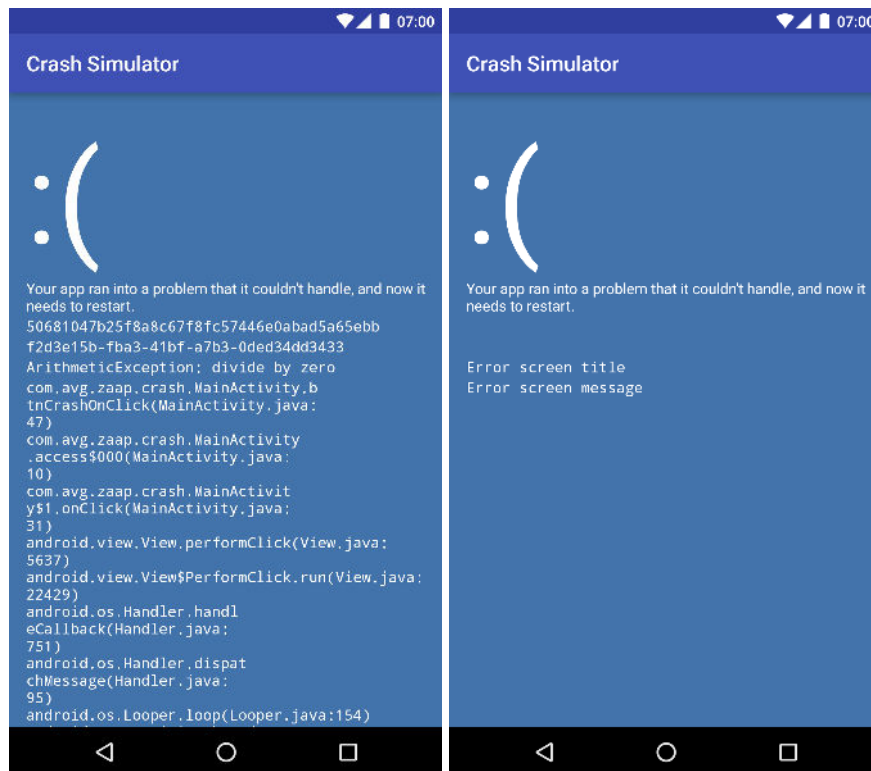


Figure 4.8: Example of Error Activity Stylised as the Windows 10 BSoD.

The `ERROR_SCREEN` configuration value behaves differently for debug and release build configurations (see Section 4.13). The unique crash code and the unique exception code are displayed only in the debug configuration option as they are meant for internal releases, release candidates, demonstration builds, testing builds and other internal releases.

In the release build configuration, the error screen displays the `errorMessage` and `errorDetails` configurations options (see Figure 4.8, right). The debug build configuration overrides those with the uncaught exception message and the crashed thread stacktrace (see Figure 4.8, left).

The layout of the Activity can be further enhanced by using custom Views that would embed application logic. This configuration value is deemed to be the preferred notification mode for most usages

4. IMPLEMENTATION

and is set as the default value in the *Crash Reporting Library* configuration management.

DIALOG

When the DIALOG configuration value is set, whenever the *Crash Reporting Library* detects a crash, it displays a customisable dialog window informing a user about the crash.

Android does allow displaying a system-level alert dialog window that overlaps other applications (including the launcher and other previously launched applications). Such dialog requires the `android.permission.SYSTEM_ALERT_WINDOW` permission. Usage of this permission is discouraged, as creating the system-level windows is meant to be reserved for system-level interaction with the user.

Moreover, since the Android 6.0 (API level 23, codename Marshmallow) permission overhaul, this permission is so-called *extra protected* which means the standard permission requesting dialog does not pop up for `SYSTEM_ALERT_WINDOW` [54]. Therefore, the user has to manually grant the permission to an application using the standard Android *Settings* application. In conclusion, this solution is not viable for applications targeting the API level 23 or higher for reliable crash reporting and user notification.

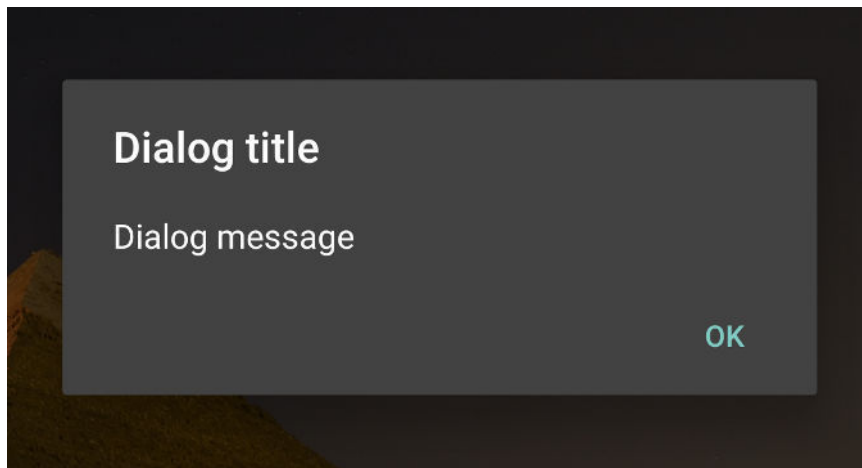


Figure 4.9: Example of the *Crash Reporting Library* Dialog Window.

Instead, the *Crash Reporting Library* simulates the system layer dialog windows by displaying a transparent Activity (implemented in the `com.avg.zaap.crashlibrary.show.error.dialog.DialogActivity` class) with disabled incoming and outgoing animations. Upon tapping anywhere in the activity, the activity dismisses the hosted dialog and itself.

The dialog window sets the `errorMessage` configuration option as the dialog title and the `errorDetails` configuration option as the dialog message. This configuration value is similar to ACRA's `DIALOG` reporting interaction mode.

NOTIFICATION

When the `NOTIFICATION` configuration value is set, whenever the *Crash Reporting Library* detects a crash, it displays a customisable system notification in the status bar (see Figure 4.10).

The notification sets the `errorMessage` configuration option as the notification title and the `errorDetails` configuration option as the notification message.

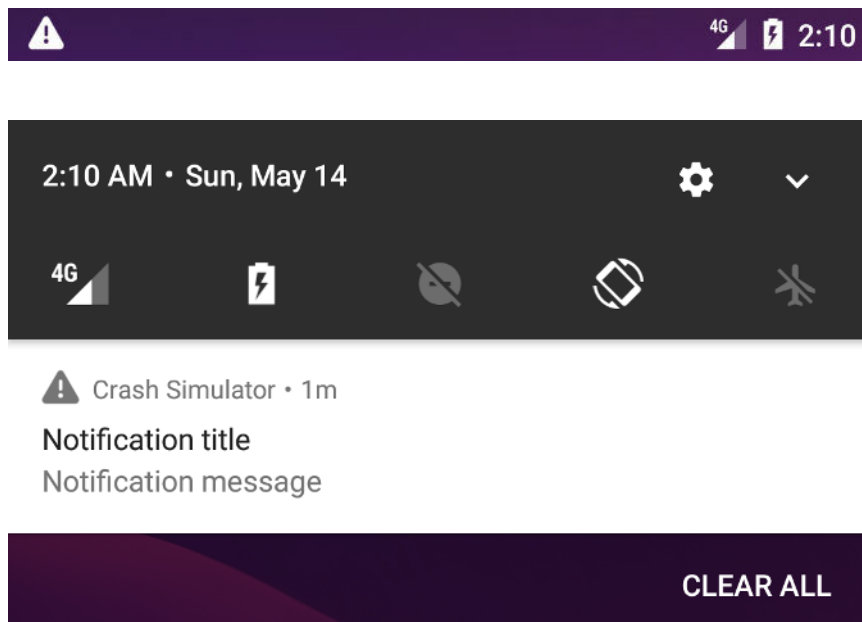


Figure 4.10: Example of the *Crash Reporting Library* Error Notification.

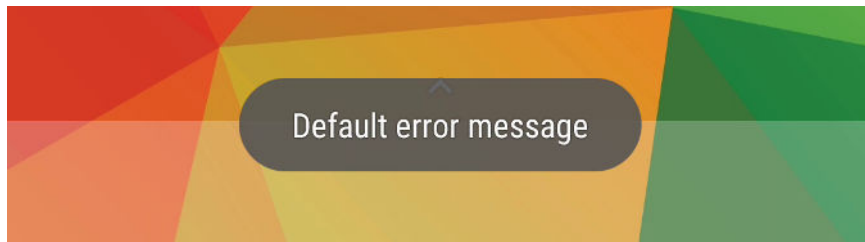


Figure 4.11: Example of the *Crash Reporting Library* Toast Message.

Furthermore, the notification icon can be configured by using the `notificationSmallIcon` configuration option. The default value is `android.R.drawable.ic_alert` (see Figure 4.10).

This configuration value is similar to ACRA's NOTIFICATION reporting interaction mode.

TOAST

When the TOAST configuration value is set, whenever the *Crash Reporting Library* detects a crash, it displays a Toast system message (see 4.11). The `errorDetails` configuration option is set as the Toast message text. The Toast is displayed for the `android.widget.Toast.LENGTH_SHORT` interval (2 seconds on standard devices).

This configuration value is similar to ACRA's TOAST reporting interaction mode.

NONE

When the NONE configuration value is set, whenever the *Crash Reporting Library* detects a crash, the user does not get notified in any way. This mode is predominantly meant for test builds, automation and unstable release candidates. Not notifying the user does not prevent the crash data from being collected and published (see Section 4.9).

This configuration value is equivalent to ACRA'S SILENT reporting interaction mode.

4.6 Providing Crash Data

As it was already established in Section 4.3, crashes mandatorily lead to process termination. As such, an effective data provision needs to be divided into two groups:

- data related to the current process and
- data which can be collected after the process termination.

The former group includes all of the data related to the memory allocated by the crashing process. Such data needs to be collected before the actual process termination – most notably the data describing a memory state, such as the stacktrace or the heap memory dump. The *Crash Reporting Library* calls the providers of such data *synchronous providers* as they actively block the `CrashUncaughtExceptionHandler`'s thread until all of requested data is collected. All *synchronous providers* implement the `com.avg.zaap.crashlibrary.provider.ICrashSyncProvider` interface.

The latter group contains data of either constant or persistent nature. This includes identifiers necessary for reproducing the crash, such as application build version, hardware ID, manufacturer and also persistently stored data such as database journals, logs and possibly other useful application diagnostic data. The *Crash Reporting Library* calls these data providers *asynchronous providers*. All *asynchronous providers* implement the `com.avg.zaap.crashlibrary.provider.ICrashAsyncProvider` interface.

Each *provider* defines a file name where it stores its result data. This value is mandatory and each *provider* is expected to store all of the collected information in the specified file for the data to be reported. The *Crash Reporting Library* stores these files in a directory named `crash` in the application internal storage space (`/data/data/<application package>/crash/` on standard devices). The *provider* files are meant to be temporary and are overwritten each time a new crash report is detected. The *Crash Reporting Library* does not restrict collision and is up to the application developer to provide non-colliding names. The *provider* files are further compiled into a single file during the crash report processing (see Section 4.8).

All *synchronous providers* implement the `provideData(String)` method returning full file path of the output file. Alternatively, *asynchronous providers* implement the `provideData(String, ICrashProviderCallback)` method. This method is asynchronous and provides the result file path via the `com.avg.zaap.crashlibrary.provider.ICrashProviderCallback` callback interface.

An application developer may define custom providers – both *synchronous* and *asynchronous*. *Synchronous providers* can be easily used to cover helpful non-persistent data (such as state of singleton classes and caches) and *asynchronous providers* can be used for reporting state and metadata of persistently stored information (i.e. logs, database data, images and tokens amongst others). Custom *providers* can be added using the *Crash Reporting Library* configuration management (see Section 4.7).

4.6.1 Default Synchronous Providers

This subsection defines the *synchronous providers* that are available within the *Crash Reporting Library* by default. All of these *synchronous providers* are enabled in the default configuration. If needed, these *providers* can be manually disabled in the configuration management (see Section 4.7).

Memory Heap Provider

The Memory Heap provider is a *synchronous data provider* generating the heap memory dump and implemented in the `com.avg.zaap.crashlibrary.provider.memory.MemoryHeapProvider` class.

The *provider* calls the `android.os.Debug.dumpHprofData(String)` method which is always executed on the *UI thread*. The memory dump generation can take quite some time since time to collect the HPROF data depends on the heap size set by the device manufacturer. Therefore, it is recommended to use the provider in development builds and internal releases and not in production.

Java uses the HPROF data format (usually denoted by the `.hprof` file extension) for its heap memory dumps. Since the Dalvik Executable format is not compatible with the standard Java bytecode, the memory dumps generated by Dalvik are not compatible with standard tools.

In order to access the memory dump data, the memory dump has to be converted first by the `hprof-conv` tool to the standard HPROF format. Then the memory dump can be processed using standard tools such as Eclipse MAT (Memory Analysis Tool, see Figure 4.12) or Oracle's `jhat` which is a part of the standard JDK. Android Studio can open and process the Dalvik HPROF files on its own using the aforementioned SDK tools.

The Memory Heap Provider stores its data in the `dump.dalvik.hprof` output file.

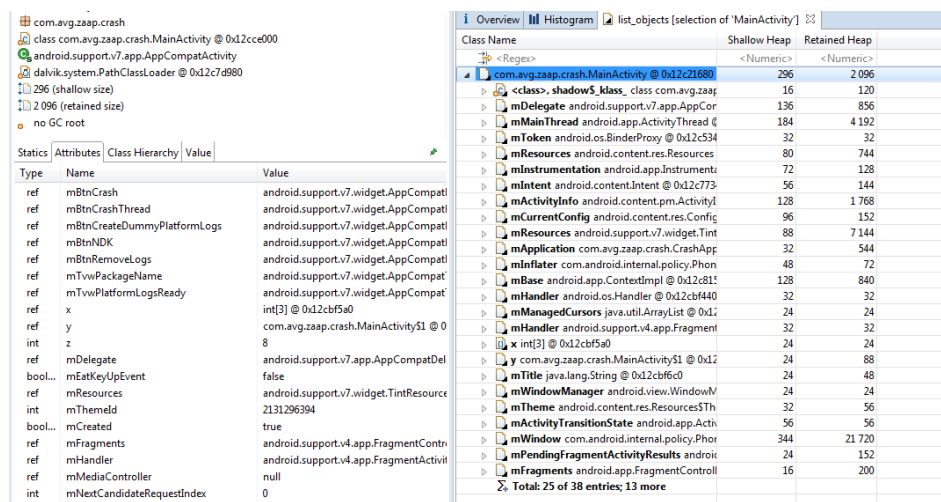


Figure 4.12: Sample Memory Heap Dump Processed in Eclipse MAT.

Crashing Stacktrace Writer and All Stacktrace Writer

There is a special class that, to be precise, is not a *synchronous provider* (since it does not implement the `ICrashSyncProvider` interface). The class is named `StackTraceWriter` and is located in the `com.avg.zaap.crashlibrary.provider.stacktrace` package. Instead of the `provide()` method defined in the `ICrashSyncProvider` interface, `StackTraceWriter` defines two special methods.

The first one is the `writeCrashingThreadStackTrace(String, Throwable)` method, used to serialise the crashing thread's stacktrace into a file. The output file name for such collected stacktrace is `StackTraceCrashingThread.txt`.

4. IMPLEMENTATION

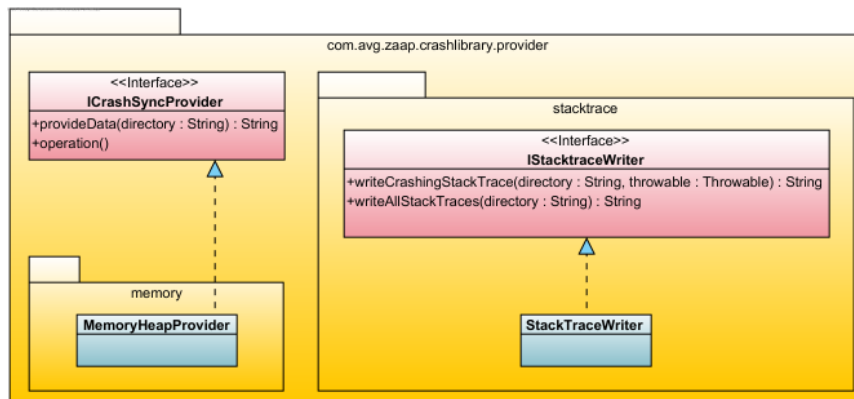


Figure 4.13: Class Diagram: Synchronous Providers.

The second one is the `writeAllStackTraces(String)` method, serialising stacktraces of all threads of the crashed process. The output file name for collected stacktraces is `StackTracesAllThreads.txt`.

The aforementioned methods are treated in the same way as the *synchronous providers*, meaning they are enabled by default and can be disabled in the *Crash Reporting Library* configuration management (see Section 4.7).

4.6.2 Default Asynchronous Providers

This subsection defines the *asynchronous providers* that are available within the *Crash Reporting Library* by default. All of these *asynchronous providers* are not enabled in the default configuration, one has to add them manually via the configuration management (see Section 4.7).

Device Info Provider

The Device Info Provider collects the data available in the `android.os.Build` class such as device brand, manufacturer, model, serial number and ABIs supported by the device's CPU amongst others. This data is meant to provide additional information in order to pinpoint any problems regarding a customised Android build, a particular vendor or a device.

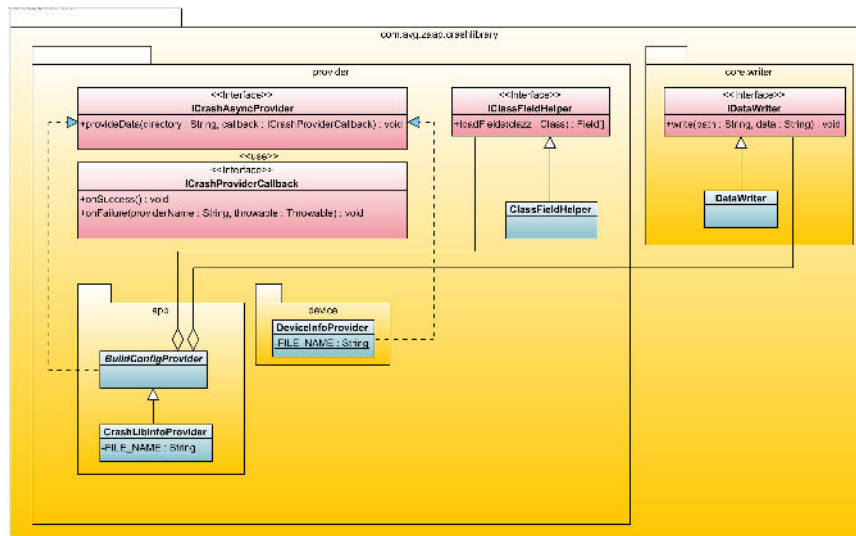


Figure 4.14: Class Diagram: Asynchronous Providers.

The output of the provider is stored in a key-value formatted text file, using colon as a separator (see Source code 4). The Device Info Provider is implemented in the `com.avg.zaap.crashlibrary.provider.device.DeviceInfoProvider` class and uses `DeviceInfo.txt` file name to temporarily store the collected data.

Crash Library Info Provider

The Crash Library Info Provider retrieves the *Crash Reporting Library* build configuration data. The application logic is delegated to the `BuildConfigProvider` class (see below).

The output of the provider is stored in a key-value formatted text file, using colon as a separator (see Source code 5). The Crash Library Info Provider is implemented in the `com.avg.zaap.crashlibrary.provider.app.CrashLibInfoProvider` class and uses `CrashLibInfo.txt` file name to temporarily store the collected data.

The BuildConfig Provider

Gradle plug-in for Android automatically creates a class named `BuildConfig` in the application (or library) package. This class is de-

4. IMPLEMENTATION

```
Board:          sailfish
Bootloader:    8996-012001-1709121620
Brand:         google
Device:        sailfish
Display:       OPR3.170623.013
HW:           sailfish
Host:          vpef8.mtv.corp.google.com
ID:           OPR3.170623.013
Manufacturer:  Google
Model:         Pixel
Product:       sailfish
Serial:        FA6AS0301099
SDK:          26
Tags:          release-keys
Type:          user
User:          android-build
ABIs:         arm64-v8a, armeabi-v7a, armeabi
```

Source code 4: Sample Device Info Generated for Google Pixel.

clared as `public final` and contains `public static` constants only. Gradle automatically adds mandatory fields from the `build.gradle` configuration file (such as package name, version name, version code, etc.) as these constants. A developer may also add additional constants into the `BuildConfig` class by using the `buildConfigField(String, String, String)` method specifying the constant type, name and value.

While there is no actual `BuildConfig Info` provider, one can write a custom *asynchronous provider* class inheriting the abstract `com.avg.zaap.crashlibrary.provider.app.BuildConfigProvider` class in order to collect the `BuildConfig` data.

The constructor `BuildConfigProvider(String, Class)` takes two formal arguments: the file name (including file extension) into which is the data temporarily stored and the class holding the `BuildConfig` data. The *provider* automatically extracts all accessible constants from the provided class.

```
APPLICATION_ID      : com.avg.zaap.crashlibrary
BUILD_TYPE         : debug
DEBUG              : true
FLAVOR             :
VERSION_CODE       : 3
VERSION_NAME       : 0.0.3
```

Source code 5: Sample Crash Library Info for Version 0.0.3.

The data is stored in the same format as the Crash Library Info Provider data (see Source code 5). In fact, the Crash Library Info Provider is implemented using the `BuildConfigProvider`.

4.7 Configuration Management

This sections defines the configuration options of the *Crash Reporting Library*, the means of passing a configuration to the library and default configuration options.

The *Crash Reporting Library*'s configuration management is heavily influenced by ACRA, providing a very similar interface for setting its configuration options. The *Crash Reporting Library* stores its configuration in an immutable class named `CrashLibConfig` located in the `com.avg.zaap.crashlibrary.config` package. There are two ways of creating a `CrashLibConfig` instance:

- using the `CrashLibConfig.Builder` builder class or
- by parsing the `@CrashLibConfigAnnotations` annotation.

The `CrashLibConfig.Builder` is a mutable class implementing the builder design pattern. The configuration options can be set by calling their respective setter methods. Each setter method returns the instance of the `CrashLibConfig.Builder` class, making it suitable for method chaining (see Source code 6). Finally, the `CrashLibConfig` instance can be retrieved by calling the `build()` method.

When using the builder class, the *Crash Reporting Library* needs to be initialised using `CrashLib.init(Context, CrashLibConfig)` method at the beginning of the application life cycle (in

4. IMPLEMENTATION

```
import android.app.Application;
import com.avg.zaap.crashlibrary.CrashLib;
import com.avg.zaap.crashlibrary.config.EShowErrorType;

public class CrashApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        CrashLibConfig config = new CrashLibConfig.Builder()
            .showErrorType(EShowErrorType.ERROR_SCREEN)
            .errorMessage("Sample error message")
            .errorDetails("Sample error details")
            .disableMemoryHeapProvider()
            .disableStackTraceCrashingThread()
            .disableStackTraceAllThreads()
            .maxAttempts(10)
            .delay(60 * 1000)
            .build();
    }
    CrashLib.init(this, config);
}
```

Source code 6: Initialising the *Crash Reporting Library* Via the Builder.

the body of either `Application#onAttachBaseContext()` or `Application#onCreate()`.

Alternatively, an application developer can use the `@CrashLibConfigAnnotations` annotation located in the `com.avg.zaap.crashlibrary.config` package.

Each configuration option can be set by assigning its respective annotation element (see Source code 7). Both Source code 6 and Source code 7 produce the same configuration.

The unspecified values are supplanted by the default configuration (see below). Unless specified otherwise (see Subsection 4.7.3), each configuration option can be set via the annotation element or via a setter method of the same name.

For example, the `applicationVersion` option can be configured by either using the `applicationVersion = <value>` annotation element or by calling `CrashLibConfig.Builder#applicationVersion(String)`. However, the number of formal arguments and their types may differ.

```
import android.app.Application;
import com.avg.zaap.crashlibrary.config.
    CrashLibConfigAnnotations;
import com.avg.zaap.crashlibrary.config.EShowErrorType;

@CrashLibConfigAnnotations(
    showErrorType = EShowErrorType.ERROR_SCREEN,
    errorMessage = "Sample error message",
    errorDetails = "Sample error details",
    disableMemoryHeapProvider = true,
    disableStackTraceCrashingThread = true,
    disableStackTraceAllThreads = true,
    maxAttempts = 10,
    delay = 60 * 1000
)
public class CrashApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        CrashLib.init(this);
    }
}
```

Source code 7: Initialising the *Crash Reporting Library* Via Annotation.

4.7.1 Application Identification Options

The *Crash Reporting Library* defines two options for identifying host applications: `applicationVersion` and `capProductID`.

The `applicationVersion` defines the host application version, most commonly either the build number or the release version using

the semantic versioning pattern [55]. The host application may specify any valid string as its `applicationVersion`. The default value is set to "unknown".

The `capProductID` (CAP stands for Crash Analysis Portal) configuration option specifies the host application product ID. The host application may specify any valid string as its `capProductID`. The default value is set to "ANDR".

4.7.2 User Notification Options

The *Crash Reporting Library* defines four options for configuring and customising user notification (see Section 4.5): `showErrorType`, `errorMessage`, `errorDetails` and `notificationSmallIcon`.

The `showErrorType` specifies the form of user notification. The configuration value must be one of the `com.avg.zaap.crashlibrary.config.EShowErrorType` enumeration values. The impact on the user is described in more detail in Section 4.5. The default value is set to `EShowErrorType.ERROR_SCREEN`.

The `errorMessage` allows user to specify a custom message that is displayed to used when the `showErrorType` is set to one of `ERROR_SCREEN`, `DIALOG`, `TOAST` or `NOTIFICATION`. The default value is set to "errorMessage is not configured".

The `errorDetails` allows user to specify a custom message that is displayed to used when the `showErrorType` is set to one of `ERROR_SCREEN`, `DIALOG`, or `NOTIFICATION`. The default value is set to "errorDetails is not configured".

The debug build of the *Crash Reporting Library* overrides both `errorMessage` and `errorDetails` when the `ERROR_SCREEN` `showErrorType` is set (see Section 4.5 and Figure 4.8, left).

The `notificationSmallIcon` specifies a custom small notification when using the `NOTIFICATION` `showErrorType`. The specified resource `int` needs to point to a drawable resource upholding the notification icon standards and requirements [56]. This value is ignored for other `showErrorType` values. The default value is set to `android.R.drawable.ic_dialog_alert`.

4.7.3 Provider Options

The *Crash Reporting Library* defines five options for configuring *synchronous* and *asynchronous providers*: `syncProviders`, `asyncProviders`, `disableMemoryHeapProvider`, `disableStacktraceCrashingThread` and `disableStackTraceAllThreads`.

The `syncProviders` configuration option allows a developer to add additional *synchronous providers*. The formal argument for the annotation element is of a `Class<? extends ICrashSyncProvider> []` type, whereas the setter method takes `List<ICrashSyncProvider>` as the formal argument. The default value is an empty `List`.

The `asyncProviders` configuration option allows a developer to add additional *asynchronous providers*. The formal argument for the annotation element is of a `Class<? extends ICrashAsyncProvider> []` type, whereas the setter method takes `List<ICrashAsyncProvider>` as the formal argument. The default value is an empty `List`.

The `disableMemoryHeapProvider` allows a developer to disable the default `MemoryHeapProvider` (see Subsection 4.6.1). When set to `true`, the `MemoryHeapProvider` is disabled and does not collect memory data. The default value of this configuration option is `false`.

The `disableStacktraceCrashingThread` allows a developer to disable the default stacktrace writer for the crashing thread (see Subsection 4.6.1). When set to `true`, the writer for the crashing thread is disabled and does not collect stacktrace data. The default value of this configuration option is `false`.

The `disableStackTraceCrashingThread` allows a developer to disable the default stacktrace writer for all threads (see Subsection 4.6.1). When set to `true`, the writer for all threads is disabled and does not collect stacktrace data. The default value of this configuration option is `false`.

The setter methods for disabling the default *providers* do not take any formal arguments. Since default *providers* can only be disabled, any arguments would be excessive.

4.7.4 Data Publishing Options

The *Crash Reporting Library* defines seven configuration options related to the crash report publishing: `crashReportURL`, `crashUploadURL`,

`crashPublishers`, `disableCapPublisher`, `strategy`, `maxAttempts` and finally `delay` (see Section 4.9).

The `crashReportURL` and `crashUploadURL` configuration options specify the endpoint URLs used in the default two-step protocol (see Section 4.9). The default *publisher* uses both of these URLs. A custom *publisher* may use one or both of these values. The default value of both configuration options is set to an empty string.

The `crashPublishers` configuration option allows user to add additional *publishers*. The formal argument for the annotation element is of a `Class<? extends ICrashPublisher>[]` type, whereas the setter method takes `List<ICrashPublisher>` as the formal argument. The default value is an empty `List`.

The `strategy` configuration option specifies the publishing strategy for publishing crash reports (see Subsection 4.9.3). The default value is set to `Strategy.BEST_EFFORT`.

The `disableCapPublisher` allows a developer to disable the default *publisher* used by the *Crash Reporting Library*. When set to `true`, the default publisher is disabled. The default value of this configuration option is `false`. The setter method for disabling the default *publisher* does not take any formal arguments.

The `maxAttempts` specifies the maximal amount of attempts to publish any single crash report. The default value is set to 3.

The `delay` configuration option specifies a minimal delay between publishing attempts in milliseconds. The default value is set to 3600000 (1 hour in milliseconds).

4.7.5 Connection Options

The *Crash Reporting Library* defines three configuration options related to suitable connection settings (see Section 4.10): `offlineSuitable`, `connectionLevelSuitable` and `roamingSuitable`.

The `offlineSuitable` configuration options specifies whether a working Internet connection is necessary for crash reporting. When set to `true`, the application developer has to disable the default *publisher* by setting the `disableCapPublisher` configuration option to `true` and add custom *publishers* that do not require an Internet connection. The default value is set to `false`.

The `connectionLevelSuitable` configuration options specifies the minimal level of connection to be considered for crash reporting (see Section 4.10). The default value is set to `Connection.WIFI`.

The `roamingSuitable` configuration options specifies whether roaming data connections are considered as suitable for crash reporting. When set to `true`, the *Crash Reporting Library* reports crashes even when the device uses a roaming connection. The default value is set to `false`.

4.7.6 Logging Options

The *Crash Reporting Library* defines a single configuration option for application logging named `logger`.

The `logger` configuration option allows an application developer to replace the default logger instance (see Section 4.11) with a custom logger. The default value is set to an instance of the `com.avg.zaap.crashlibrary.core.logger.DefaultLogger` class.

4.8 Packaging of collected data

This section defines the proprietary data format used within the *Crash Reporting Library* and support tools that can be used for reading and extracted the crash report data.

As it was established in Section 4.6, each *provider* creates its own temporary file containing its data. This approach allows to easily configure default *providers* within the *Crash Reporting Library* and virtually effortless implementation of custom application-related *providers*.

However, handling multiple files tends to be tedious, especially when they are to be transferred over a network. Network communication of mobile devices cannot be perceived as reliable due to their mobile nature. Therefore, transactional handling of crash reports is a must. Handling multiple files within a transaction either leads to multiple requests which are harder to process transactionally.

4.8.1 The BinPacker File Format

During the initial prototype development of the *Crash Reporting Library*, the idea was to use a commonly used compression algorithm for

4. IMPLEMENTATION

compiling *provider* files into a single one, in order to save the effort of developing custom file format and support tools. The initial prototype used the ZIP algorithm since Java already provides a standard API for handling standard ZIP and GZIP file formats within the standard Java API [57].

However, as it had turned out during the empiric observation, the implementation is too computationally demanding for mobile devices. This issue naturally affects older single-CPU devices the most. On older devices even when using the STORE compression level², compiling the collected data may take up to several minutes.

Since the main goal of the *Crash Reporting Library* is to reach maximal possible universality, even these older devices are meant to be supported. The *Crash Reporting Library* therefore uses a proprietary light-weight file format named *BinPacker File Format*.

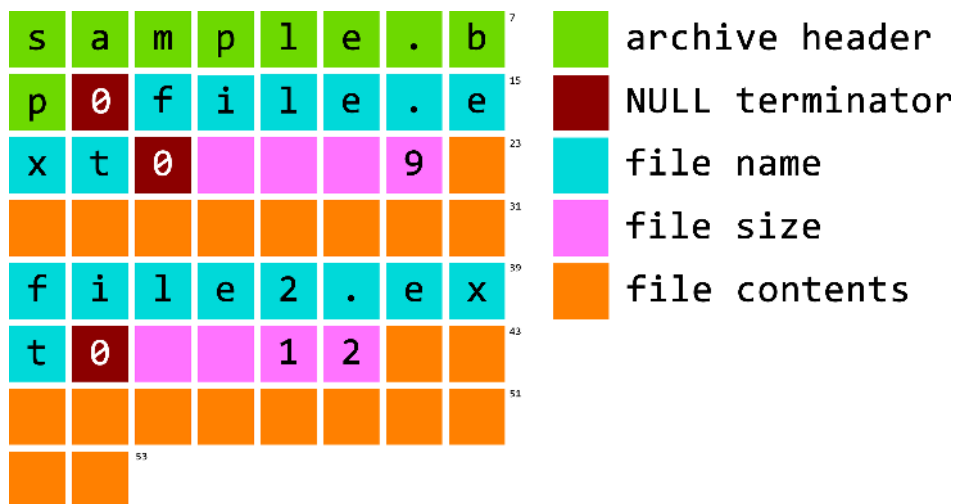


Figure 4.15: Visualisation of the *BinPacker File Format*.

The *BinPacker File Format* does not provide file compression and stores only the minimal amount of metadata. Unlike ZIP or GZIP, the *BinPacker File Format* does not support hierarchical directory structure. However, a root directory can be specified by using a directory header (see below). A *BinPacker File Format* file can be identified by the .bp

2. Meaning the ZIP archive is not compressed and actually takes more disk space than the sum of the original files.

file extension. Files compiled into a *BinPacker File Format* file are from hereinafter referred to as *binpacked*.

Each *BinPacker File Format* archive consists of two parts – the *archive header* and the *archive content*. The *archive header* is a null-terminated UTF-8 string (similar to strings in the C programming language). If the archive header is equal to the archive file name, the archive does not define a root directory. Otherwise the archive header specifies a root directory that is created upon the archive extraction.

The *archive content* consists of one or more file records. A record contained within a BinPacker archive has exactly three parts:

1. **file name** – a null-terminated UTF-8 string³,
2. a **32-bit unsigned integer** – specifying the file size in bytes⁴ and
3. **file content** – the actual file content (binary).

Figure 4.15 visualises a sample *BinPacker File Format* archive named `sample.bp` which contains two files: `file.ext` (9 bytes) and `file2.ext` (12 bytes).

4.8.2 BinPacker API

The API for both compiling and extracting *BinPacker File Format* archives is defined in two interfaces: `IBinPacker` and `IBinUnPacker` located in the `com.avg.zaap.crashlibrary.publish.packer` package. The actual implementations are the `BinPacker` and `BinUnPacker` classes located in the aforementioned package.

The `BinPacker` API is meant for Android only since it uses Android Support Annotations. Conversely, the `BinUnPacker` API can be used as a part of an Android project or as a standalone executable version usable with standard Java Runtime Environment (see Subsection 4.8.3).

3. Usage of non-English characters is supported, however, discouraged due to their larger size and possible incompatibilities among various file systems. All of the default *synchronous* and *asynchronous providers* use ASCII file names only.

4. Therefore the maximal theoretical file size of a single file within the archive is about 4 GB.

4. IMPLEMENTATION

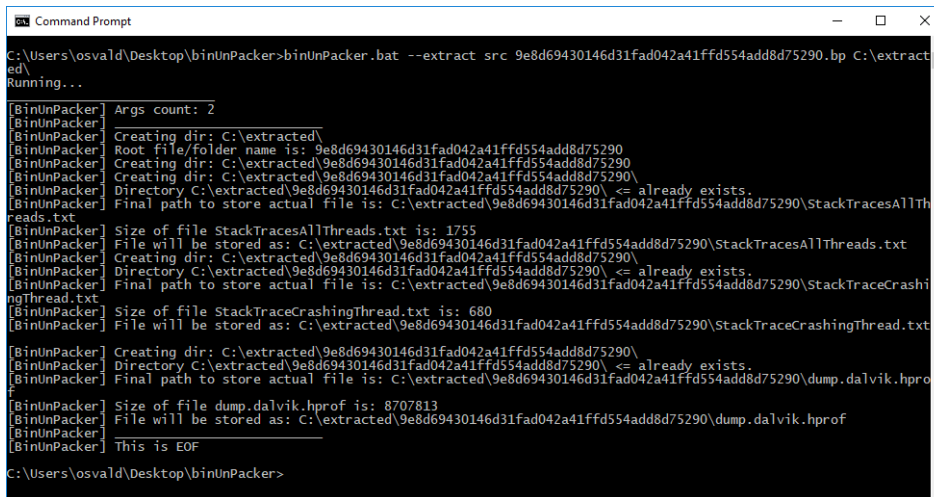
4.8.3 Support Tools

A *BinPacker File Format* archive can be processed using one the two available tools: the Windows batch file and or the GUI application written in Java. Both of these tools use the BinUnPacker Java API and therefore require Java Runtime Environment installed on the host machine.

The BinPacker batch tool allows to compile a standalone version of the BinPacker API from sources and use the compiled version. The host machine must have both Java Development Kit installed and its PATH environment variable needs to contain its bin directory. The tool allows to list the archive content and extract the binpacked files into a specific location (see Source code 8).

```
C:\>binUnPacker.bat --extract src
      9e8d69430146d31fad042a41ffd554add8d75290.bp
      C:\extracted\
```

Source code 8: Extracting a file using the BinUnPacker Batch Tool.



```
Command Prompt
C:\Users\osvald\Desktop\binUnPacker>binUnPacker.bat --extract src 9e8d69430146d31fad042a41ffd554add8d75290.bp C:\extract
ed\
Running...

[BinUnPacker] Args count: 2
[BinUnPacker]
[BinUnPacker] Creating dir: C:\extracted\
[BinUnPacker] Root file/folder name is: 9e8d69430146d31fad042a41ffd554add8d75290
[BinUnPacker] Creating dir: C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290
[BinUnPacker] Creating dir: C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\
[BinUnPacker] Directory C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\ <= already exists.
[BinUnPacker] Final path to store actual file is: C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\StackTracesAllTh
reads.txt
[BinUnPacker] Size of file StackTracesAllThreads.txt is: 1755
[BinUnPacker] File will be stored as: C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\StackTracesAllThreads.txt
[BinUnPacker] Creating dir: C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\
[BinUnPacker] Directory C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\ <= already exists.
[BinUnPacker] Final path to store actual file is: C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\StackTraceCrashi
ngThread.txt
[BinUnPacker] Size of file StackTraceCrashingThread.txt is: 680
[BinUnPacker] File will be stored as: C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\StackTraceCrashingThread.txt
[BinUnPacker]
[BinUnPacker] Creating dir: C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\
[BinUnPacker] Directory C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\ <= already exists.
[BinUnPacker] Final path to store actual file is: C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\dump.dalvik.hprof
[BinUnPacker] Size of file dump.dalvik.hprof is: 8707813
[BinUnPacker] File will be stored as: C:\extracted\9e8d69430146d31fad042a41ffd554add8d75290\dump.dalvik.hprof
[BinUnPacker]
[BinUnPacker] This is EOF
C:\Users\osvald\Desktop\binUnPacker>
```

Figure 4.16: The BinUnPacker Batch Tool.

The BinUnPackerUI is a GUI tool implemented in Java using the Swing UI library. The tool allows to set the source archive and the

destination path using standard file dialogs and to set the default extraction directory. The BinPackerUI tool also allows to specify automatic overwrite and/or safe extract strategy (see Figure 4.17).

The BinPackerUI tool is distributed as an executable .jar archive.

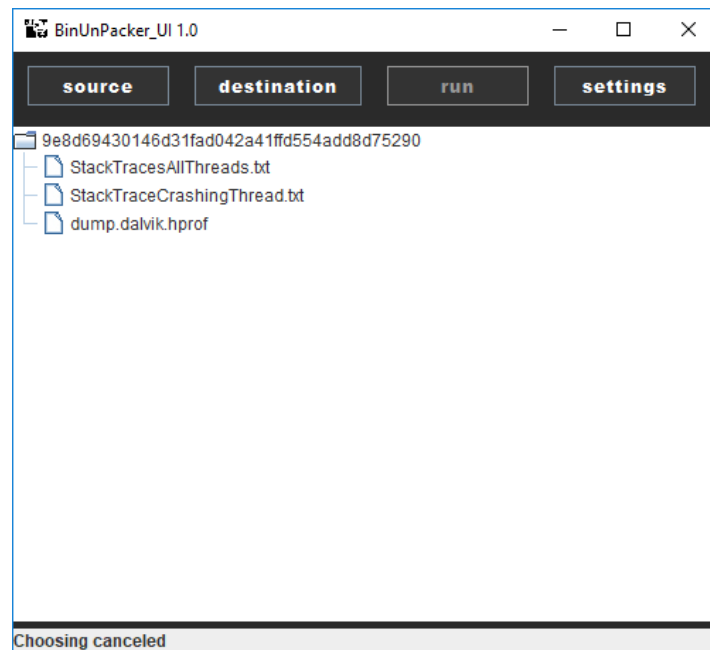


Figure 4.17: The BinUnPackerUI GUI Tool.

4.9 Data Publishing

This section describes the process of publishing crash reports and default *publishers* available within the *Crash Reporting Library*.

Once the crash report is compiled into a single BinPacker file, assuming there is a suitable Internet connection (see Section 4.10), the *Crash Reporting Library* attempts to publish the crash report using the configured *publishers* and publishing strategy (see Subsection 4.9.3).

A *publisher* is any class implementing the `com.avg.zaap.crashlibrary.publish.ICrashPublisher` interface. The interface defines a single method named `publish(Context, ReportData, String, ICrashPublisherCallback)`. The *publishers* communicate

asynchronously using the `ICrashPublisherCallback` callback interface.

The *Crash Reporting Library* library contains two *publishers*: the CAP Crash Publisher and the External Storage Publisher. An application developer may add additional *publishers* using the configuration management (see Subsection 4.7.4).

4.9.1 CAP Crash Publisher

The CAP Crash Publisher is the default *publisher* for the *Crash Reporting Library*. The *publisher* defines a two-step protocol compatible with the crash reporting infrastructure used by AVG. This *publisher* uses the two HTTP endpoints defined in the *Crash Reporting Library* configuration: `crashReportURL` and `crashUploadURL` (see Subsection 4.7.4).

The CAP Crash Publisher first sends just the metadata about the crash report. According to the received response, either the full crash report is uploaded on the server or the publisher returns success, marking the file as successfully published.

The crash report request is a HTTP POST `multipart/form-data` with following header containing the identification metadata (see Subsection 4.3.1):

- **EXC** – **Exception Code**,
- **FSO** – **Faulting Offset**,
- **MDN** – **Module Name**,
- **MDV** – **Module Version**,
- **PFN** – **Process Name**,
- **PFV** – **Process Version** and
- **BCK** – **Crash Code**.

The report request contains following string body parts:

- `guid` – randomly generated **GUID**,
- `prd` – **Product ID**,

- `prv` – **Application Version** and
- `protver` – **Product ID** (CAP Crash Publisher uses protocol version 2).

The response of the report request is a text/plain that contains one of the following result states (defined in the `com.avg.zaap.crashlibrary.publish.cap.http.CapResponseCode` class).

-1 ERROR

Signalises either communication breakdown or server error.

0 NO_DUMP

No more crash reports of given type are needed, CAP has already collected sufficient amount of samples.

1 MINI_DUMP

Minidump upload requested.

2 FULL_DUMP

Full dump upload requested.

6 KEEP_DUMPS

No more crash reports of given type needed, the client is advised to keep recorded dumps for further investigation.

8 DIRECT_CFG

Forces to reload the configuration of the crash reporter (*not implemented*).

The CAP Crash Publisher treats both `-1 ERROR` and connection error (time-out or failure to resolve the endpoint address) as a failure. The `0 NO_DUMP` response is treated as a success and the BinPacker archive is not uploaded. All other responses lead to the BinPacker file upload using the upload request.

The upload request is a HTTP POST `multipart/form-data` with following string body parts:

- `guid` – randomly generated **GUID** and
- `prd` – **Product ID**.

The crash reporting server pairs the report request and upload request using the **Product ID** and randomly generated **GUID**. Both of these values are the same for both requests. The *Crash Reporting Library* solution also contains a mock implementation of the CAP server for integration testing purposes (see Section 4.12).

The CAP Publisher is enabled by default and can be disabled using the configuration management (see Subsection 4.7.4).

4.9.2 External Storage Publisher

The External Storage Publisher is a *publisher* that copies the Bin-Packer archive to the external storage. The provider is implemented in the `ExternalStorageCrashPublisher` located in the `com.avg.zaap.crashlibrary.publish.externalstorage` package.

Unlike internal storage space that restrict access to their respective applications, the Android external storage may be used by all applications that are granted either `android.permission.READ_EXTERNAL_STORAGE` (for read access) or `android.permission.WRITE_EXTERNAL_STORAGE` (for read/write access) and is even accessible to the user. Historically, the external storage was implemented using an SD card that was mounted upon inserting into the device. Although many contemporary handsets no longer use SD cards, they emulate the external storage state API and provide a symbolic link `/sdcard/` that points to the emulated external storage.

The *Crash Reporting Library* stores its data in the dedicated directory for the host application (`/sdcard/Android/data/<package name>/files/` on standard devices). The External Storage Directory does not require a working Internet connection and returns failure only when the file cannot be written (i.e. when the SD card is unmounted or when it does not have enough free space left). The provider is not enabled by default but can be added using the configuration management (see Subsection 4.7.4).

4.9.3 Publishing Strategies

Since the *Crash Reporting Library* supports usage of multiple *publishers*, the question arises as to how to handle potential publishing failures.

Since *publishers* are independent from each other, one *publisher* may return success, while other one reports its result as a failure.

Therefore, the *Crash Reporting Library* provides three possible publishing strategies:

- ALL,
- ANY and
- BEST Effort.

Publishing strategies are defined as constants in the `com.avg.zaap.crashlibrary.offline.Strategy` enumeration.

The ALL configuration treats a publishing attempt as a success whenever all configured *publishers* return success as their result. This configuration is primarily intended for production builds and other builds where consistency is valued over reporting every single crash report.

The ANY configuration treats a publishing attempt as a success whenever at least one *publisher* returns success as its result. For example, when using both `CapCrashPublisher` and `ExternalStoragePublisher`, a situation when the network is unavailable but the crash report is successfully stored in the external storage directory is treated as a success.

The BEST Effort strategy treats every publishing attempt as a success no matter what the results of the configured *publishers* are. This configuration is primarily intended for controlled environments (such as QA automation or internal releases) where application bugs can be easily reproduced.

When the *Crash Reporting Library* is configured to use just a single *publisher*, the ALL and ANY publishing strategies behave identically. Once a crash report file is successfully published, the BinPacker archive is permanently deleted.

4.10 Connection Handling

This subsection describes the way the *Crash Reporting Library* handles different means of Internet connection or the lack of it.

4. IMPLEMENTATION

Mobile devices running Android often have to face unstable connection due to nature of their network access. Even when a network connection is available, the connection might be metered or have a small data limit. In such cases, it might be unacceptable to publish crash reports since some *publisher* files can easily take up to several dozens of megabytes.

Therefore, the *Crash Reporting Library* defines the following four network levels:

1. NONE,
2. OTHER,
3. MOBILE and
4. WIFI

The NONE configuration means that Internet connection is not required for reporting crashes. This means the crash report is handled locally and not meant to be published over a network.

The OTHER configuration represents less frequently used means of connection, such as Ethernet connection on devices like Android TV or handsets and tablets with Bluetooth tethering.

The MOBILE configuration represents connection using the mobile data network. This type of connection is often limited or metered and therefore mostly suitable for smaller crash reports, if any.

The WIFI configuration represents a Wi-Fi connection. This type of connection is often unlimited and unmetered and the *Crash Reporting Library* treats it as the most stable and suitable for crash reporting.

The minimal required connection level can be set using the configuration management (see Subsection 4.7.5). If the detected connection is of a equal or of a higher level than the configured level, the crash report is reported. Otherwise it is stored and the *Crash Reporting Library* waits until a suitable connection is detected. For example when setting the suitable connection to MOBILE, a crash report will be reported when the device is connected via either Wi-Fi or mobile data network but, not when using tethering or when the device is offline altogether.

4.10.1 Offline Publisher

The *Crash Reporting Library* defines a special pseudo-*publisher* named Offline Publisher that is implemented in the `com.avg.zaap.crashlibrary.offline.publisher.OfflinePublisher` class. Unlike the regular *publishers*, the Offline Publisher does not implement the `ICrashPublisher` interface. Instead, it implements the `com.avg.zaap.crashlibrary.offline.publisher.IOfflinePublisher` interface that is virtually identical to `ICrashPublisher`. This prevents an application developer from adding the Offline Publisher as a configured *publisher* while keeping a unified API at the same time.

Unlike regular *publishers*, the purpose of the Offline Publisher is not to publish the findings of a crash report but to rather store metadata about the report. The Offline publisher moves the crash report to the `offline` directory located within the application internal storage space.

The Offline Publisher also serialises the instance of the `ReportData` class containing the crash report metadata into a file named after the BinPacker archive with an additional `.metadata` extension (i.e. for a BinPacker archive named `example.bp`, the Offline Publisher would write the metadata into a file named `example.bp.metadata`). The application logic for serialising and deserialising the crash report metadata is defined in the `CrashMetadataWriter` and `CrashMetadataReader` classes respectively. Both classes are located in the `com.avg.zaap.crashlibrary.offline.publisher.metadata` package.

Whenever the *Crash Reporting Library* detects a connectivity change, the `CrashReportingService` checks whether the configured minimal delay has passed and if so, it starts another publishing attempt. Each time the Offline Publisher processes a crash report, it also increments its publishing attempts count.

4.10.2 Publishing Attempts

Each crash report has a maximum amount of crash report publishing attempts and a minimal time delay between two consequent attempts. When a publishing strategy is unsuccessful (see Subsection 4.9.3), the *Crash Reporting Library* waits for at least the specified delay before starting another publishing attempt.

When the maximal amount of crash reports is reached and none of them have been successful, the crash report and its metadata are permanently deleted and no more publishing attempts are performed.

4.11 Logging

This section defines the logging API used in the *Crash Reporting Library* and the means of possible extension.

Android replaces the Java Logging API defined in the `java.util.logging` package with its own. The standard Android Logging API for Java is implemented in the `android.util.Log` class [58]. The Android logging API defines the following severity levels:

1. VERBOSE (lowest severity),
2. DEBUG,
3. INFO,
4. WARNING,
5. ERROR and
6. ASSERT (highest severity).

```
#include <android/log.h>
__android_log_print(ANDROID_LOG_DEBUG, "Tag", "Message");
```

Source code 9: Logging a DEBUG Message Using Android NDK.

The `Log` class defines one character long methods for each severity except `ASSERT`. Therefore, `VERBOSE` messages can be logged using `Log#v`, `DEBUG` messages using `Log#d` and so on. The `Log` class also defines a jokingly named method `Log#wtf` (literally *What a Terrible Failure*; reports a condition that should never happen) that logs with severity `ASSERT`.

The Android logging API is also available from native code upon including the `log.h` header file (see Source code 9).

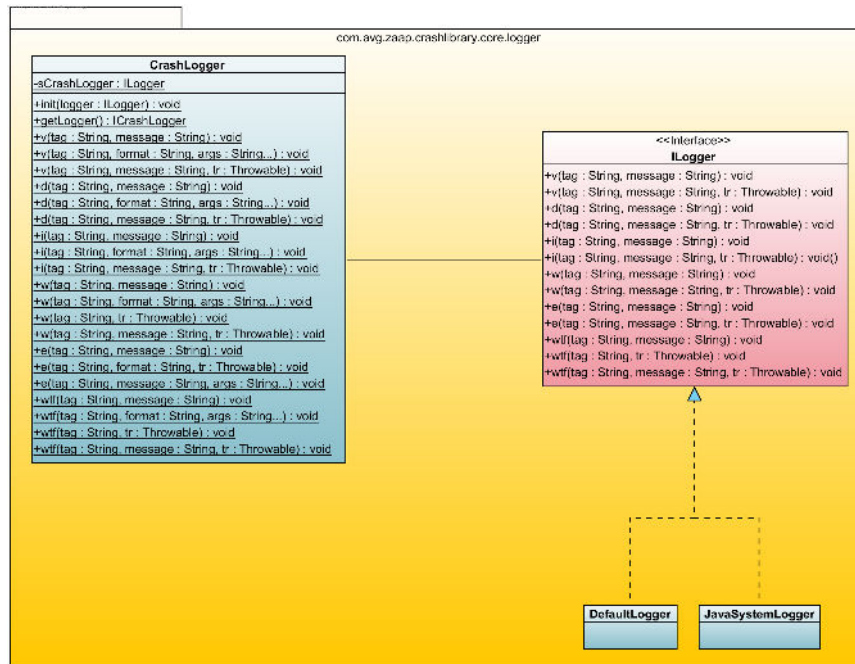


Figure 4.18: Class Diagram: Logging.

4.11.1 Logging API

The *Crash Reporting Library* uses the `CrashLogger` static class located in the `com.avg.zaap.crashlibrary.core.logger` package as an entry point for all of its logging functionality. The `CrashLogger` class is initialised during the *Crash Reporting Library* initialisation within the `CrashLib#init` method call. `CrashLogger` automatically loads the logger instance configured in the library configuration (see Subsection 4.7.6).

The `CrashLogger` API mirrors the API defined in the Android `Log` class, using the same one character long method naming pattern. Additionally, its methods also support string formatting using optional `varargs` arguments. The main advantage of such approach is the effortless compatibility with overwhelming majority of Android applications.

The default logger for the *Crash Reporting Library* is implemented in the `com.avg.zaap.crashlibrary.core.logger.DefaultLogger` class.

The default logger logs messages only when used in the debug build of the *Crash Reporting Library* (see Section 4.13). When using the release build of the *Crash Reporting Library*, the default logger silently discards all logging method calls.

The standalone BinUnPacker API uses a custom logger implemented in the `JavaSystemLogger` located in the `com.avg.zaap.crashlibrary.core.logger` package. The `JavaSystemLogger` prints log messages with severity `ERROR` or `ASSERT` to the standard error output. Messages with lower severity are printed to the standard output.

An application developer may also define its own logger by implementing the `com.avg.zaap.crashlibrary.core.logger.ILogger` interface and setting the implementation class in the configuration management (see Subsection 4.7.6).

4.12 Testing

This section defines the testing strategy used to verify the *Crash Reporting Library* builds.

The testing strategy for the *Crash Reporting Library* involves unit testing, integration testing and integration with the existing build and automation infrastructure at AVG.

4.12.1 Unit Testing

This subsection describes the unit testing strategy for the *Crash Reporting Library* and the patterns used to support testability.

The *Crash Reporting Library* uses Java, the JUnit testing framework, Mockito and PowerMock as its unit testing technologies of choice. The unit test asserts are written using Java implementation of the Hamcrest matchers [59]. Hamcrest provides a syntax more resembling a natural language than the default `junit.framework.Assert` class does.

The *Crash Reporting Library* does not use any dependency injection framework in order to limit the third party dependencies the *Crash Reporting Library* relies on. Instead, each class implements the dependency factory design pattern.

The dependency factory design pattern prescribes that each class also defines a factory interface and its default implementation (i.e.

```
package com.example;
import android.content.Context;

public class Example {

    private Context mContext;
    private Dependency mDependency;

    public Example(Context context) {
        this(context, new ExampleFactory());
    }

    public Example(Context context,
        IExampleFactory factory) {
        mContext = context;
        mDependency = factory.getDependency();
    }

    public boolean example(String value) {
        return value.isEmpty();
    }
}
```

Source code 10: Dependency Factory Pattern Example

for a class named `Writer`, there also exists an interface named `IWriterFactory` and an implementation named `WriterFactory` located in the same package, see Source code 10).

Each constructor that is meant for public API, calls another constructor of the same class with an additional formal argument of the dependency factory interface type. All dependencies are to be retrieved using getter methods of the dependency factory. Simply put, the only classes that use the `new` keyword are factory implementations.

This way, all of the internal dependencies are easily mockable using anonymous factory implementations (see Source code 11) and

Mockito. Most of Android SDK dependencies can be mocked using this approach as well.

The *Crash Reporting Library* intentionally avoids `static` classes and methods as much as possible. This is due to `static` and `final` classes and methods (including certain Android API calls) need to be tested using PowerMockito and the PowerMock JUnit runner. Additionally, even when using these, not every such construct is testable.

In case an implementation differs between debug and release builds, both implementations are tested within the same test method. In these cases, the test method contains a control structure that reads the `DEBUG` field of the `com.avg.zaap.crashlibrary.BuildConfig` class and adapts accordingly to the retrieved value.

Unit test methods use a different naming convention than regular Java methods. The method name consists of three parts separated by underscores: `<tested method>_<input>_<expected output>`.

This naming convention allows easier detection of the cause of a failed test. The individual parts use the camel case notation (see Source code 11).

The unit tests for a selected build type can be executed either using Android Studio or by running a Gradle task (see Subsection 4.13.4).

4.12.2 Integration Testing

This subsection defines the integration testing strategy used to verify builds of the *Crash Reporting Library*.

Android provides a UI testing framework named UIAutomator [60] and a GUI tool for scanning and analysing UI named UI Automator Viewer within the standard Android SDK.

AVG has developed a proprietary library named `autolib` for test automation of desktop and mobile applications. The `autolib` library provides two APIs: a PowerShell one for desktop applications and a Python one for mobile applications. The Python API supports both Android and iOS.

Unlike many other automation frameworks, tests written using the `autolib` are not monolithic. Instead, they define a series of reusable so-called *steps*. *Steps* are primarily meant to be independent of each other although, in some cases one *step* may depend on the result of another *step*.

```
package com.example;

import static org.hamcrest.core.Is.is;
import static org.mockito.Mockito.mock;

import android.content.Context;

public class ExampleTests {
    private Context mContext;
    private Dependency mDependency;
    private Example mExample;

    @Before
    public void setUp() {
        mContext = mock(Context.class);
        mDependency = mock(Dependency.class);
        mExample = new Example(mContext, new
            IExampleFactory {
                @Override
                public void Dependency getDependency() {
                    return mDependency;
                }
            });
    }

    @Test
    public void example_emptyString_returnsTrue() {
        assertThat(mExample.example(""), is(true));
    }

    @Test
    public void example_nonEmptyString_returnsFalse() {
        assertThat(mExample.example("test"), is(false));
    }
}
```

Source code 11: Example of a Unit Test Structure.

4. IMPLEMENTATION

The test scenarios are written TSD domain-specific language based on the JSON file format. TSDs support two-level hierarchy. Each TSD contains one or more *test cases*. Each *test case* contains one or more *steps*.

The test definitions are registered using the `steps.new_step()` function call of the `lib_python` module. Each test definition has to provide a human-readable *step* name and a pointer to its implementation function. *Steps* can be customised by adding formal arguments in its human-readable name using brackets. The `autolib` library automatically parses the *step* name and puts the parsed values into a dictionary. The implementation function is called with the dictionary containing the arguments as its formal argument.

Source code 12 demonstrates *step* customisation. Adding a *step* named "Example step: hello world" to a TSD would print [hello, world] to the standard output upon its execution.

Each step should call one of the following functions as its last statement:

- `execution.passed()` (signals success),
- `execution.failed()` (signals failure, the test case may continue) or
- `execution.error()` (signals critical failure, the test case execution should be terminated).

```
from lib_python import steps, execution, logger

def example_step_function(params):
    logger.debug("[{0}, {1}"].format(params["first"],
        params["second"])
    execution.passed()

steps.new_step("Example step: {first} {second}",
    example_step_function)
```

Source code 12: Example Step Implementation in Python.

These function calls allow the execution engine to determine the *step* result and display it accordingly. Any uncaught exception is implicitly treated as an `execution.error()` call.

In summary, the integration testing solution consists of these four parts:

- `autolib`,
- the **crash automation library**,
- **test definitions** and
- **TSDs**.

The **crash automation library** adds additional functionality that is specific for the *Crash Reporting Library* and is not covered by `autolib`. This includes handling the mock CAP Server (see Subsection 4.12.3), manipulation with the emulator using Telnet and managing simulated connection on an emulator. The library is located in the `automation/libcrashtest` directory.

The **test definitions** are contained within a single Python module that contains *step* implementations. Test definitions are located in the `automation/testdefinitions` directory.

The **TSDs** are JSON files containing the test scenarios to verify a *Crash Reporting Library* build. The *Crash Reporting Library* implements the TSDs listed below. All of these TSDs are located in the `automation/tsd` directory.

`CrashDialogDebug.tsd`

This TSD tests the `DIALOG` configuration value of the `showErrorType` configuration option for the debug build of the *Crash Reporting Library*.

`CrashDialogRelease.tsd`

This TSD tests the `DIALOG` configuration value of the `showErrorType` configuration option for the release build of the *Crash Reporting Library*.

`CrashErrorScreenDebug.tsd`

This TSD tests the `ERROR_SCREEN` configuration value of the

4. IMPLEMENTATION

showErrorType configuration option for the debug build of the *Crash Reporting Library*.

CrashErrorScreenRelease.tsd

This TSD tests the ERROR_SCREEN configuration value of the showErrorType configuration option for the release build of the *Crash Reporting Library*.

CrashNoneDebug.tsd

This TSD tests the NONE configuration value of the showErrorType configuration option for the debug build of the *Crash Reporting Library*.

CrashNoneRelease.tsd

This TSD tests the NONE configuration value of the showErrorType configuration option for the release build of the *Crash Reporting Library*.

CrashNotificationDebug.tsd

This TSD tests the NOTIFICATION configuration value of the showErrorType configuration option for the debug build of the *Crash Reporting Library*.

CrashNotificationRelease.tsd

This TSD tests the NOTIFICATION configuration value of the showErrorType configuration option for the release build of the *Crash Reporting Library*.

CrashToastDebug.tsd

This TSD tests the TOAST configuration value of the showErrorType configuration option for the debug build of the *Crash Reporting Library*.

CrashToastRelease.tsd

This TSD tests the TOAST configuration value of the showErrorType configuration option for the release build of the *Crash Reporting Library*.

CrashReportAll.tsd

This TSD tests the ALL publishing strategy using both CAP

Crash Publisher (with mock CAP Server) and External Storage Publisher.

CrashReportAny.tsd

This TSD tests the ANY publishing strategy using both CAP Crash Publisher (with mock CAP Server) and External Storage Publisher.

CrashReportBestEffort.tsd

This TSD tests the BEST_EFFORT publishing strategy using both CAP Crash Publisher (with mock CAP Server) and External Storage Publisher.

4.12.3 Mock CAP Server

This subsection describes the mock CAP Server implementation and the API used during the testing of the *Crash Reporting Library*.

The mock CAP server provides the same API as the real implementation. Unlike the real implementation, the mock one does not process the crash reports in any way and simply stores them in the file system instead. The mock CAP server also provides additional APIs for testing purposes. When started, the mock CAP server listens on the port 8080.

The mock CAP server supports the URLs listed below. Unless specified otherwise, the MIME type of each response is `text/plain`.

/report (POST)

This URL serves as an endpoint for reporting a new crash occurrence.

/upload (POST)

This URL serves as an endpoint for uploading a new crash report.

/files (GET)

Lists all files uploaded to the server thus far. The list uses the CRLF control characters as separators.

/file?guid=value (GET)

Returns the crash report file uploaded with given GUID *value*. The MIME type of the response is `application/octet-stream`.

4. IMPLEMENTATION

`/delete?guid=value` (GET)

Deletes the crash report file previously uploaded with the given GUID *value*.

`/disable-server` (GET)

Disables the mock server. When disabled, the mock CAP Server always return `-1` error as its response.

`/enable-server` (GET)

Enables the mock server if the mock server has been previously disabled.

`/uploading` (GET)

Returns whether there is a file currently being uploaded to the mock CAP Server.

4.12.4 Automation

This subsection defines the means of automation of the *Crash Reporting Library* integration tests.

The integration tests can be started either manually using a tool named TSD Debugger or using Final-CI. TSD Debugger is a proprietary tool developed by AVG for the development and execution of TSDs (see Figure 4.19). TSD Debugger implements a server-client architecture that allows running *steps* on real devices, local emulators (emulators running on the host machine) and remote emulators (emulators running on a remote or virtual machine). Unfortunately, certain *steps* (i.e. emulating a roaming data connection) can only be performed on an emulator.

Final-CI is a proprietary continuous integration framework build upon Atlassian Bamboo. Final-CI provides custom adapter that allow to process both JUnit and TSD test results. When running TSD tests, Final-CI acquires a new instance of a virtual machine from a dedicated pool. This virtual machine executes the Android emulator and starts performing TSD *steps* one at a time.

Further details about the automation environment are undisclosed, in compliance with AVG's specific instruction.

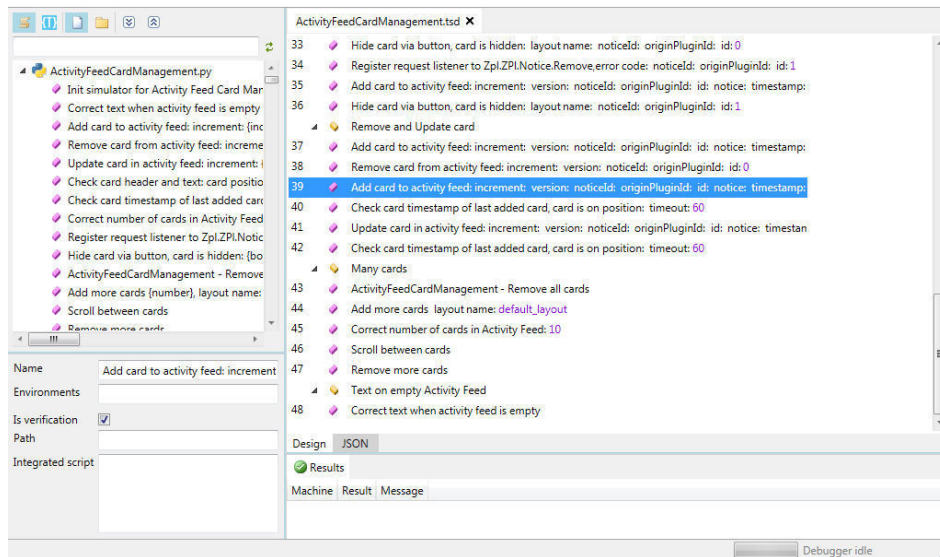


Figure 4.19: Example of a TSD Opened in TSD Debugger.

4.13 Building

This subsection describes the building process of the *Crash Reporting Library* and its respective variants.

The *Crash Reporting Library* solutions is divided into the following three Gradle submodules:

- `lib` (the actual *Crash Reporting Library*),
- `simulator` (the Simulator application used for testing) and
- `capserver` (the mock CAP Server implementation).

4.13.1 Building the Library

This subsection describes the build configurations of the *Crash Reporting Library*.

Android Studio automatically creates two build types for each application or library project. These build types are named `debug` and `release`. The *Crash Reporting Library* uses these two default build types generated by Android Studio.

4. IMPLEMENTATION

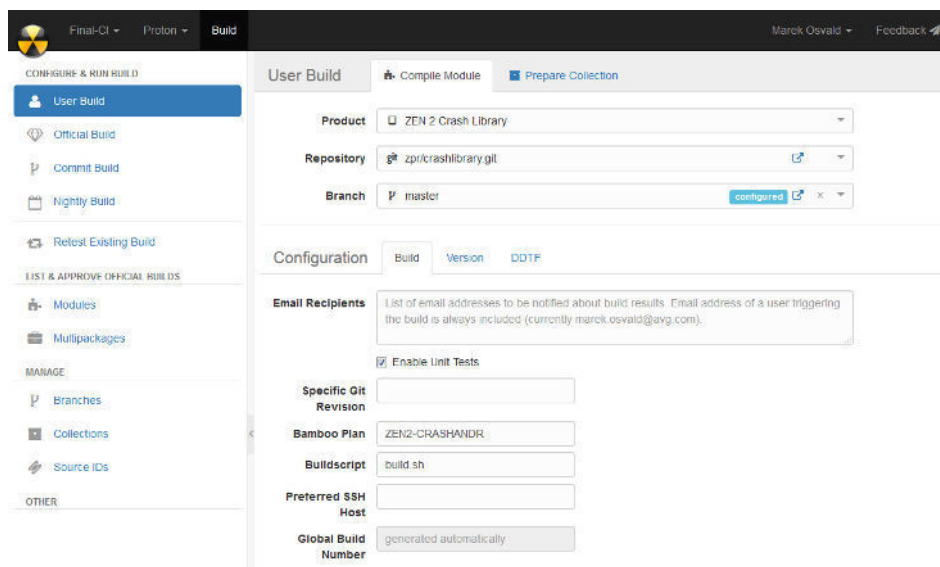


Figure 4.20: Example of a Test Run Configuration in Final-UI.

The differences between debug and release build configurations are described in Sections 4.5 and 4.11.

4.13.2 Building the Simulator

This subsection describes the build configurations of the Simulator application.

Gradle supports so-called flavours. A flavour allows to divide the codebase into a reusable core and flavour-related bits of code. The code shared by all flavours is located in the `src/main` directory, while the `src/<flavour name>` directory contains flavour-specific implementations of Java classes and/or flavour specific Android XML resources.

The Simulator application uses flavours to declare multiple configurations of the *Crash Reporting Library* within a single application project. The simulator offers 16 build configurations listed below.

activityDebug

This configuration uses the debug version of the *Crash Reporting Library* and configures the `ERROR_SCREEN` as its `showError` configuration value.

activityRelease

This configuration uses the release version of the *Crash Reporting Library* and configures the `ERROR_SCREEN` as its `showError` configuration value.

dialogDebug

This configuration uses the debug version of the *Crash Reporting Library* and configures the `DIALOG` as its `showError` configuration value.

dialogRelease

This configuration uses the release version of the *Crash Reporting Library* and configures the `DIALOG` as its `showError` configuration value.

noneDebug

This configuration uses the debug version of the *Crash Reporting Library* and configures the `NONE` as its `showError` configuration value.

noneRelease

This configuration uses the release version of the *Crash Reporting Library* and configures the `NONE` as its `showError` configuration value.

notificationDebug

This configuration uses the debug version of the *Crash Reporting Library* and configures the `NOTIFICATION` as its `showError` configuration value.

notificationRelease

This configuration uses the release version of the *Crash Reporting Library* and configures the `NOTIFICATION` as its `showError` configuration value.

toastDebug

This configuration uses the debug version of the *Crash Reporting Library* and configures the `TOAST` as its `showError` configuration value.

toastRelease

This configuration uses the release version of the *Crash Reporting Library* and configures the TOAST as its `showError` configuration value.

reportAnyBrokenSDDebug

This configuration uses the debug version of the *Crash Reporting Library*. It configures the library to use the ANY publishing strategy and adds the `BrokenSDCardPublisher` in the configuration management.

reportAnyBrokenSDRelease

This configuration uses the release version of the *Crash Reporting Library*. It configures the library to use the ANY publishing strategy in the configuration management.

reportAnyDebug

This configuration uses the debug version of the *Crash Reporting Library*. It configures the library to use the ANY publishing strategy in the configuration management.

reportAnyRelease

This configuration uses the release version of the *Crash Reporting Library*. It configures the library to use the ANY publishing strategy in the configuration management.

reportBestEffortDebug

This configuration uses the debug version of the *Crash Reporting Library*. It configures the library to use the `BEST Effort` publishing strategy in the configuration management.

reportBestEffortRelease

This configuration uses the release version of the *Crash Reporting Library*. It configures the library to use the `BEST Effort` publishing strategy in the configuration management.

4.13.3 Building the Mock CAP Server

This subsection describes the build configurations of the mock CAP Server.

The mock CAP server offers the two build configurations automatically created by Android Studio: debug and release. Both of these configurations are identical.

4.13.4 Additional Gradle Tasks

This subsection describes the additional Gradle tasks defined in the `build.gradle` file.

The *Crash Reporting Library* project also defines several Gradle tasks that simplify building multiple builds of different types at once.

buildCrashLibraryDebug

Builds the debug version of the *Crash Reporting Library*, all debug configurations of the Simulator application and the mock CAP server.

buildCrashLibraryRelease

Builds the release version of the *Crash Reporting Library*, all release configurations of the Simulator application and the mock CAP server.

buildCrashLibrary

Builds both versions of the *Crash Reporting Library*, all build configurations of the Simulator application and the mock CAP server. Depends on `buildCrashLibraryDebug` and `buildCrashLibraryRelease`. This task is executed by `Final-CI`.

testCrashLibraryDebug

Runs JUnit tests for the debug version of the *Crash Reporting Library*. Depends on `buildCrashLibraryDebug`.

testCrashLibraryRelease

Runs JUnit tests for the release version of the *Crash Reporting Library*. Depends on `buildCrashLibraryRelease`.

testCrashLibrary

Runs JUnit tests for the all versions of the *Crash Reporting Library*. Depends on `buildCrashLibrary`.

`copyTestResults`

Copies the JUnit tests results into the output folder observed by Final-CI. Depends on `buildCrashLibrary`. This task is executed by Final-CI.

4.14 Publishing

This section describes the artefact publishing process of the *Crash Reporting Library*.

The common way of distributing Android libraries is the Android Archive format, usually denoted by the `.aar` file extension. The format itself is identical to standard Java `.jar` archive however, the `.aar` file extension signals that the library contains compiled classes in the Dalvik Executable Format and is therefore incompatible with standard Java Virtual Machine.

Therefore, Android libraries can be distributed in the same way as their Java counterparts, using the same tools. Gradle provides compatibility with Maven and its POM (Project Object Model) [61].

AVG uses JFrog's Artifactory [62], a universal repository manager that supports Java, JavaScript, PHP, Python and Ruby libraries amongst others. An artefact can be published using either the web interface (see Figure 4.21) or using the Gradle Plug-in.

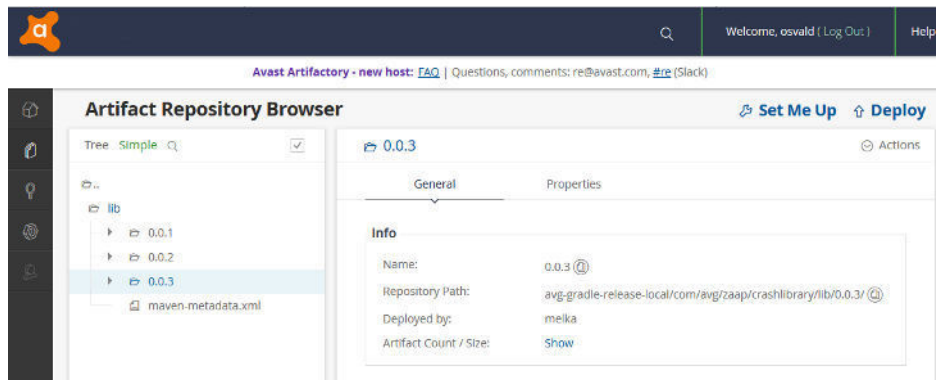


Figure 4.21: The *Crash Reporting Library* Artifactory Dossier.

The *Crash Reporting Library* uses the JFrog Artifactory Plug-in (dependency `com.jfrog.artifactory`) for direct publishing into Artifac-

tory. Gradle optionally uses Java Properties File Format for configuration. Gradle automatically loads the `gradle.properties` files located in home Gradle directory (`%USERPROFILE%\` `.gradle` on Windows or `~/` `.gradle` on UNIX-based systems), in the main project directory and in the submodule directory. This allows storing the Artifactory credentials outside the actual project.

The *Crash Reporting Library* defines a custom Gradle task named `artifactoryPublish` for publishing the library. The `artifactoryPublish` task requires the following Gradle properties to be configured in one of the `gradle.properties` files:

- `artifactory_contextUrl`,
- `artifactory_user` and
- `artifactory_password`

The values specifying the Artifactory instance URL and the user credentials are not disclosed for obvious reasons. The `artifactoryPublish` task publishes both debug and release versions of the *Crash Reporting Library*.

5 Conclusion

This thesis successfully fulfilled its outlined goals. A comparative research and analysis of the crash reporting solutions available for Android was performed. Since none of the considered solutions provided the desired level of universality and configurability, a custom solution named *Crash Reporting Library* was implemented.

The implemented solution offers crash reporting of both managed code and native code crashes, as requested. It does so using a unified API. The solution collects the metadata required by the existing infrastructure and allows processing crashes of Android applications in the same way as AVG's portfolio of existing desktop applications does.

The developed solution offers extensive and easily maintainable configuration management and due to its modular architecture, it offers multiple ways of notifying the user amongst others. The configuration management of the implemented solution offers various extension points in order to allow easy implementation of application-specific extensions.

The implemented architecture was documented using the Unified Modelling Language. The source code was thoroughly documented using the JavaDoc documenting syntax.

The *Crash Reporting Library* comes with a basic predefined sets of *providers* and *publishers*, shortening the time necessary for setting up a commonly used configuration.

A custom light-weight file format was developed in order to lower the computation power needed for collecting and publishing crash reports. A set of support tools for this file format was developed in cooperation with the Quality Assurance Team in order to best suit their needs.

Based on my proposal, the implemented solution offers advanced connection handling, eliminating undesired data transfers on limited, unstable and metered networks.

A detailed and comprehensive test strategy including unit tests, integration tests and automation was designed and implemented using the standard tools used by AVG. The build and verification process is fully integrated into AVG's infrastructure.

5. CONCLUSION

The implemented library was published in AVG's standard library repository, making it available for both existing and newly developed Android applications.

The *Crash Reporting Library* is currently used by one application developed by AVG and is ready for further development and extensions.

Personally, working on the *Crash Reporting Library* was a tremendous experience. It provided me with an opportunity to use multiple programming languages and to get familiar with new and exciting technologies. It was particularly interesting to be present during all phases of application development and to see an enterprise-level project (although a relatively small one) rise from the ground up.

I am absolutely positive that I will apply the knowledge gained during the development of the *Crash Reporting Library* during my future career.

Bibliography

1. ORACLE CORPORATION. *Java Platform SE 7: NullPointerException*. 2016. Available online [last revision 12 December 2017]
<https://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html>.
2. XAMARIN, INC. *Xamarin: Xamarin.Android*. 2017. Available online [last revision 12 December 2017]
<https://developer.xamarin.com/api/root/MonoAndroid-lib/>.
3. MEYER, K. *Ruboto*. 2017. Available online [last revision 12 December 2017]
<https://web.archive.org/web/20161119184120/http://ruboto.org:80/>.
4. PROGRESS SOFTWARE CORPORATION. *NativeScript: Home*. 2017. Available online [last revision 12 December 2017]
<https://www.nativescript.org/>.
5. JETBRAINS. *Kotlin Programming Language*. 2017. Available online [last revision 12 December 2017]
<https://kotlinlang.org/>.
6. GOOGLE. *Android Developers: WebView*. 2017. Available online [last revision 12 December 2017]
<https://developer.android.com/reference/android/webkit/WebView.html>.
7. THE APACHE SOFTWARE FOUNDATION. *Apache Harmony: Open Source Java Platform*. 2017. Available online [last revision 12 December 2017]
<https://harmony.apache.org/>.
8. ORACLE CORPORATION. *OpenJDK*. 2017. Available online [last revision 12 December 2017]
<http://openjdk.java.net/>.
9. MURPHY, M. L. *The CommonsBlog: Musings on Android and the OpenJDK*. 2016. Available online [last revision 12 December 2017]
<https://commonsware.com/blog/2016/01/07/musings-android-openjdk.html>.

BIBLIOGRAPHY

10. NORBYE, T. *Android Studio Project Site: Android Studio 0.3.2 Released*. 2013. Available online [last revision 12 December 2017]
<http://tools.android.com/recent/androidstudio032released>.
11. GOOGLE. *Android Developers: Use Java 8 Language Features*. 2017. Available online [last revision 12 December 2017]
<https://developer.android.com/guide/platform/j8-jack.html>.
12. LAU, J. *Android Developers: Future of Java 8 Language Feature*. 2017. Available online [last revision 12 December 2017]
<https://android-developers.googleblog.com/2017/03/future-of-java-8-language-feature.html>.
13. MEDNIEKS, Z.; DORNIN, L.; MEIKE, G. B.; NAKAMURA, M. *Programming Android, Second Edition*. Sebastopol, California: O'Reilly Media, 2012.
14. PYTHON SOFTWARE FOUNDATION. *Python.org: About Python*. 2017. Available online [last revision 12 December 2017]
<https://www.python.org/about/>.
15. FREE SOFTWARE FOUNDATION. *GNU Project: Free Software Foundation: Bash*. 2017. Available online [last revision 12 December 2017]
<https://www.gnu.org/software/bash/>.
16. GOOGLE. *Android Studio: The Official IDE for Android*. 2017. Available online [last revision 12 December 2017]
<https://developer.android.com/studio/index.html>.
17. CHENG, B.; BUZBEE, B. *A JIT Compiler for Android's Dalvik VM*. 2010. Available online [last revision 12 December 2017]
<https://dl.google.com/googleio/2010/android-jit-compiler-androids-dalvik-vm.pdf>.
18. GOOGLE. *Android Developers: Android NDK*. 2017. Available online [last revision 12 December 2017]
<https://developer.android.com/ndk/index.html>.
19. ECMA INTERNATIONAL. *Introducing JSON*. 2017. Available online [last revision 12 December 2017]
<http://www.json.org/>.

BIBLIOGRAPHY

20. ATlassian. *Atlassian: Bamboo*. 2017. Available online [last revision 12 December 2017]
<https://www.atlassian.com/software/bamboo>.
21. GOOGLE. *Android Developers: Support Library*. 2017. Available online [last revision 12 December 2017]
<https://developer.android.com/topic/libraries/support-library/index.html>.
22. JUNIT. *JUnit: About*. 2017. Available online [last revision 12 December 2017]
<http://junit.org/>.
23. FABER, S. *Mockito framework site*. 2017. Available online [last revision 12 December 2017]
<http://site.mockito.org/>.
24. ORACLE CORPORATION. *Java Platform SE 7: RuntimeException*. 2016. Available online [last revision 12 December 2017]
<https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html>.
25. HALEBY, J. *Mockito framework site*. 2017. Available online [last revision 12 December 2017]
<https://github.com/powermock/powermock>.
26. GOOGLE. *Google Git: Breakpad*. 2017. Available online [last revision 12 December 2017]
<https://chromium.googlesource.com/breakpad/breakpad>.
27. GOOGLE. *Crashpad: Project Status*. 2017. Available online [last revision 12 December 2017]
<https://chromium.googlesource.com/crashpad/crashpad/+//HEAD/doc/status.md>.
28. CUELLAR, D. *Appium: Mobile App Automation Made Awesome*. 2017. Available online [last revision 12 December 2017]
<http://appium.io/>.
29. GOUCHER, A. *SeleniumHQ Browser Automation: About Selenium*. 2017. Available online [last revision 12 December 2017]
<http://www.seleniumhq.org/about/>.

BIBLIOGRAPHY

30. HAWKE, P. *GitHub: NanoHTTPD – Tiny, easily embeddable HTTP server*. 2017. Available online [last revision 12 December 2017]
<https://github.com/NanoHttpd/nanohttpd>.
31. GAUDIN, K. *ACRA: Know your bugs*. 2013. Available online [last revision 12 December 2017]
<http://www.acra.ch/>.
32. GAUDIN, K. *GitHub: Backends*. 2017. Available online [last revision 12 December 2017]
<https://github.com/ACRA/acra/wiki/Backends>.
33. GAUDIN, K. *GitHub: Acralyzer*. 2015. Available online [last revision 12 December 2017]
<https://github.com/ACRA/acralyzer>.
34. SMIRNOV, A. *GitHub: Acra-breakpad*. 2014. Available online [last revision 12 December 2017]
<https://github.com/4ntoine/Acra-breakpad>.
35. SMIRNOV, A. *GitHub: 4ntoine/acra*. 2014. Available online [last revision 12 December 2017]
<https://github.com/4ntoine/acra>.
36. GOOGLE. *Android Developers: Toasts*. 2017. Available online [last revision 12 December 2017]
<https://developer.android.com/guide/topics/ui/notifiers/toasts.html>.
37. MICROSOFT. *HockeyApp Joins Microsoft*. 2014. Available online [last revision 12 December 2017]
<https://www.hockeyapp.net/blog/2014/12/11/hockeyapp-joins-microsoft.html>.
38. MICROSOFT. *HockeyApp Support: How to use ACRA with HockeyApp*. 2016. Available online [last revision 12 December 2017]
<https://support.hockeyapp.net/kb/client-integration-android/how-to-use-acra-with-hockeyapp>.
39. OWEN, T. *Twitter Is Buying A Startup, Crashlytics, And Not Killing It Off For A Change*. 2013. Available online [last revision 12 December 2017]
<http://www.businessinsider.com/twitter-acquires-crashlytics-2013-1>.

40. PARET, R. *Fabric is Joining Google*. 2017. Available online [last revision 12 December 2017]
<https://fabric.io/blog/fabric-joins-google>.
41. PEREZ, S. *Twitter's Mobile Crash Reporting Tool Crashlytics Arrives On Android*. 2013. Available online [last revision 12 December 2017]
<https://techcrunch.com/2013/05/30/twitters-mobile-crash-reporting-tool-crashlytics-arrives-on-android/>.
42. FABRIC. *Fabric for Android: Build Tools*. 2017. Available online [last revision 12 December 2017]
<https://docs.fabric.io/android/crashlytics/build-tools.html>.
43. GOOGLE. *Android Developers: Service*. 2014. Available online [last revision 12 December 2017]
<https://developer.android.com/guide/topics/manifest/service-element.html>.
44. ORACLE CORPORATION. *Java Platform SE 7: Throwable*. 2016. Available online [last revision 12 December 2017]
<https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html>.
45. ORACLE CORPORATION. *Java Platform SE 7: Error*. 2016. Available online [last revision 12 December 2017]
<https://docs.oracle.com/javase/7/docs/api/java/lang/Error.html>.
46. ORACLE CORPORATION. *Java SE Documentation: JNI Functions*. 2014. Available online [last revision 12 December 2017]
<http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html>.
47. GOOGLE. *Create Hello-CMake with Android Studio*. 2017. Available online [last revision 12 December 2017]
<https://codelabs.developers.google.com/codelabs/android-studio-cmake/#0>.
48. ORACLE CORPORATION. *Java Platform SE 7: System*. 2017. Available online [last revision 12 December 2017]
<https://docs.oracle.com/javase/7/docs/api/java/lang/System.html>.

BIBLIOGRAPHY

49. FREE SOFTWARE FOUNDATION. *The GNU C Library*. 2014. Available online [last revision 12 December 2017]
https://www.gnu.org/software/libc/manual/html_mono/libc.html#Signal-and-Sigaction.
50. GOOGLE. *Android Developers: Android NDK Native APIs*. 2017. Available online [last revision 12 December 2017]
https://developer.android.com/ndk/guides/stable_apis.html.
51. GOOGLE. *Git at Google: Breakpad*. 2017. Available online [last revision 12 December 2017]
<https://chromium.googlesource.com/breakpad/breakpad/>.
52. GOOGLE. *Getting Started with Breakpad*. 2017. Available online [last revision 12 December 2017]
https://github.com/google/breakpad/blob/master/docs/getting_started_with_breakpad.md.
53. GOOGLE. *Linux Syscall Support (LSS)*. 2017. Available online [last revision 12 December 2017]
<https://chromium.googlesource.com/linux-syscall-support/>.
54. GOOGLE. *Android Developers: Manifest.permission*. 2017. Available online [last revision 12 December 2017]
https://developer.android.com/reference/android/Manifest.permission.html#SYSTEM_ALERT_WINDOW.
55. PRESTON-WERNER, T. *Semantic Versioning 2.0.0*. 2017. Available online [last revision 12 December 2017]
<https://semver.org/>.
56. GOOGLE. *Android Developers: Status Bar Icons*. 2017. Available online [last revision 12 December 2017]
https://developer.android.com/guide/practices/ui_guidelines/icon_design_status_bar.html.
57. ORACLE CORPORATION. *Java Platform SE 7: Package java.util.zip*. 2017. Available online [last revision 12 December 2017]
<https://docs.oracle.com/javase/7/docs/api/java/util/zip/package-summary.html>.

BIBLIOGRAPHY

58. GOOGLE. *Android Developers: Log*. 2017. Available online [last revision 12 December 2017]
<https://developer.android.com/reference/android/util/Log.html>.
59. HAMCREST.ORG. *Java Hamcrest*. 2012. Available online [last revision 12 December 2017]
<http://hamcrest.org/JavaHamcrest/>.
60. GOOGLE. *Android Developers: UI Automator*. 2017. Available online [last revision 12 December 2017]
<https://developer.android.com/training/testing/ui-automator.html>.
61. THE APACHE SOFTWARE FOUNDATION. *Maven: POM Reference*. 2017. Available online [last revision 12 December 2017]
<https://maven.apache.org/pom.html>.
62. JFROG. *Android Developers: UI Automator*. 2017. Available online [last revision 12 December 2017]
<https://www.jfrog.com/artifactory/>.