

Poznan University of Technology  
Faculty of Computer Science and Management  
Institute of Computer Science

Master's thesis

**AUTOMATED GUI TESTING OF MOBILE JAVA APPLICATIONS**

Marcin Zduniak

Supervisor  
dr inż. Bartosz Walter

Poznań, 2006–2007

Tutaj przychodzi karta pracy dyplomowej;  
oryginal wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                             | <b>1</b>  |
| 1.1      | Overview . . . . .                              | 1         |
| 1.1.1    | Mobile Application Development . . . . .        | 1         |
| 1.1.2    | Problems of GUI Testing . . . . .               | 3         |
| 1.2      | Motivation and Goals . . . . .                  | 5         |
| 1.3      | Typographical Conventions . . . . .             | 5         |
| 1.4      | Thesis Outline . . . . .                        | 5         |
| <b>2</b> | <b>Background</b>                               | <b>6</b>  |
| 2.1      | Approaches to Software Testing . . . . .        | 6         |
| 2.2      | Review of Existing Testing Frameworks . . . . . | 7         |
| 2.3      | J2ME Architecture . . . . .                     | 8         |
| <b>3</b> | <b>RobotME Framework Design</b>                 | <b>12</b> |
| 3.1      | High-level Architecture . . . . .               | 12        |
| 3.2      | Recorded Event Types . . . . .                  | 15        |
| 3.3      | Problems . . . . .                              | 16        |
| <b>4</b> | <b>RobotME: Implementation</b>                  | <b>21</b> |
| 4.1      | System Components . . . . .                     | 21        |
| 4.1.1    | Enhancer . . . . .                              | 21        |
| 4.1.2    | Recorder . . . . .                              | 23        |
| 4.1.3    | Replayer . . . . .                              | 23        |
| 4.1.4    | Log Server . . . . .                            | 24        |
| 4.1.5    | XML Processor . . . . .                         | 25        |
| 4.2      | Assertions and Failure Notification . . . . .   | 26        |
| <b>5</b> | <b>Experiments and Evaluation</b>               | <b>28</b> |
| 5.1      | Lab Tests . . . . .                             | 28        |
| 5.2      | Real-life Use Case . . . . .                    | 29        |
| 5.3      | Bytecode Size and Performance Aspects . . . . . | 31        |
| 5.4      | Obfuscation . . . . .                           | 32        |

|                                  |           |
|----------------------------------|-----------|
| <b>6 Summary and Conclusions</b> | <b>33</b> |
| 6.1 Summary . . . . .            | 33        |
| 6.2 Future Directions . . . . .  | 33        |
| <b>Bibliography</b>              | <b>35</b> |
| <b>Web Resources</b>             | <b>36</b> |

# Chapter 1

## Introduction

### 1.1 Overview

Applications written for mobile devices have become more and more complex and sophisticated, adjusting to the constantly improving computational power of hardware. With the growing application size comes the need for automated testing frameworks, particularly frameworks for automated testing of user interaction and graphical user interface. While such testing (also called *capture-replay*) has been thoroughly discussed in literature with respect to desktop applications, mobile development limits the possibilities significantly. To our best knowledge only a few solutions for creating automated tests of mobile applications exist and their functionality is very limited in general or constrained to only proprietary devices. In this paper will be demonstrated preliminary results of the attempt to design and implement a framework for capturing and replaying user interaction in applications written for the Java 2 Micro Edition environment. The prototype implementation of our project we called RobotME framework.

#### 1.1.1 Mobile Application Development

Java applications written for mobile devices (mobile phones in vast majority) are simpler and smaller compared to their desktop or server cousins. The environment provides a simple virtual machine (JVM) for executing the program's code and a set of generic application programming interfaces (API) for accessing hardware layer – the device's display, network or communication ports.

Similar to desktop and server applications development, also mobile applications development cycle is divided into a few separated steps, that are commonly understood in IT industry and only for clarity depicted on the figure. What is usually different between regular (desktop or server) application development and mobile application development is Developing Software stage and Testing stage, other stages like: Analysis, Requirements Gathering or Releasing are rather similar and not worth describing. First of this stages has additional task that does not exist while developing other kind of software: obfuscation. Obfuscation is the process of removing from the compiled application code these parts of the software

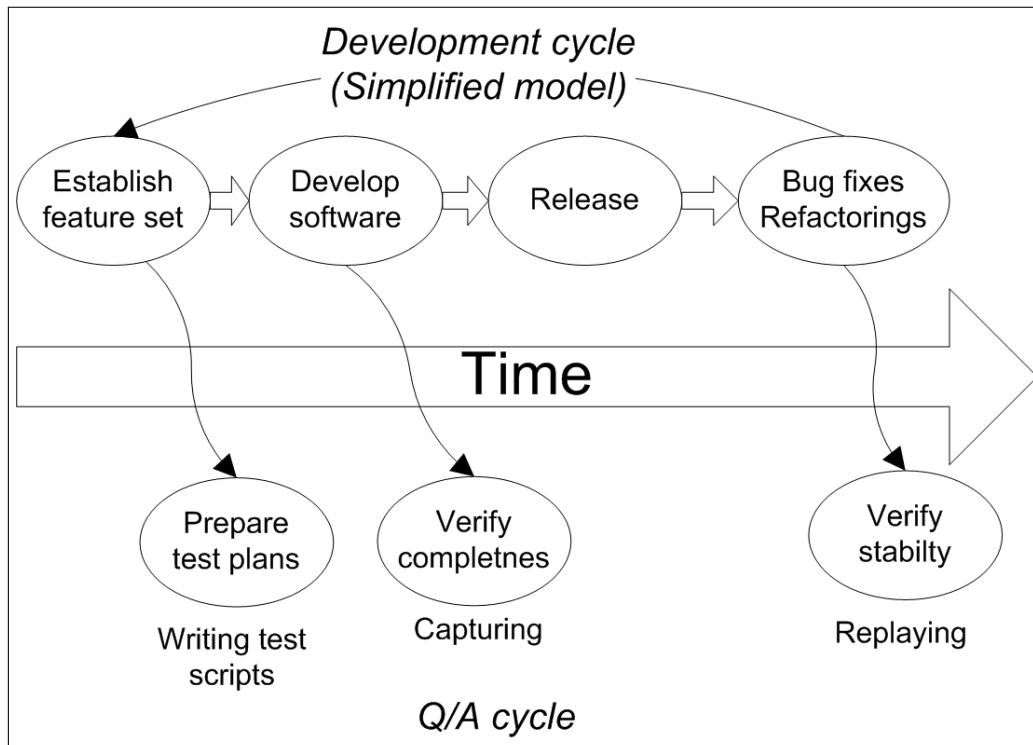


FIGURE 1.1: Capture-replay regression test scenario.

that are entirely not used in the code (cleaning up death code) and also changing remaining code so that it is very difficult to understand even after decompilation (reverse engineering). Obfuscation is necessary because thanks to it mobile application is smaller, behaves more performant and also is more difficult to steal by some unpleasant person (it is very hard to understand by human previously obfuscated code after decompilation).

Testing stage is different to some extent from analogous stage from regular applications development because this time software must be tested on the real device – mobile phone. It is the only option if the software developer would like to have its software working properly on possibly wide range of devices. Usually differences between devices are so widespread and considerable that application must be developed with this problems in mind just from the beginning, and even more often different applications must be developed for different devices. Sometimes differences are rather small and it is possible to rewrite only part of the application for one device so that it behaves correctly on other device. Such rewriting process in mobile industry is called 'porting', and good testing framework must also has solution for this process (usually the part of code that needs to be rewritten is connected only with graphic rendering and not with user interaction with the application, so it is now very harmful for testing framework).

During development stage programmer has additional ability to test their application continuously – thanks to simulators of real devices. One of such simulator, called Wireless

Toolkit provided by Sun Microsystems, is a simulator of generic J2ME device. It has build-in most of various APIs that might be used in mobile devices. Preliminary test might be performed using simulators, they have advantage comparing to performing tests on real devices – it is significantly faster to deploy and perform such test. RobotME Testing Framework we have built can be executed both on real devices and simulators. Unfortunately most of simulators simply do not simulate devices properly in every aspects, and there is no other option to testing mobile applications but to verify it on real devices when comes to the release stage.

### 1.1.2 Problems of GUI Testing

Both programming and particularly testing are much more difficult in such a constrained environment compared to writing programs for the desktop. Each mobile phone, for example, has a different hardware configuration: display size and capabilities (number of colors), size of memory and varying computational power. Application interfaces defined by the J2ME specification and considered a ‘standard’, are implemented by different vendors and often contain differences that must be taken into account, increasing the complexity of the program. The same application looks, but often also *behaves* a bit different depending on the target device it was installed on. The key differences between mobile and traditional software development was summarized in Table 1.1.

Because of the differences in hardware and software, software for mobile devices should be tested on each individual piece of equipment separately. Knowing that the deployment process takes some time, testing quickly becomes a tedious routine software developers grow to hate in no time. Writing a *capture-once, replay-on-all* testing framework seems like a natural answer addressing the problem, even if the experiences with this type of tests in desktop applications are not always rosy (contrary to the desktop, mobile applications are much simpler, so test scenarios should retain manageable size). Unfortunately, the J2ME environment does not offer any system-level support with respect to handling GUI events and any other events for that matter. In the following section we show how to substitute this required and missing functionality with automatic preprocessing of the binary code of the tested program (a process generally known as *bytecode-level instrumentation* or *code injection*).

While theoretically appealing, writing integration tests for applications with a rich graphical user interface (GUI) presents a generally complex technical problem. Since the human-computer interaction is quite unpredictable, GUI applications resist rigorous testing. A common solution is to *record* real scenarios of user interaction with the program (directly off the application’s screen) and then try to reproduce the same stimuli at the testing phase, validating program’s response accordingly. This kind of procedure is made possible with various GUI automation tools and programming interfaces; programs for recording GUI events are called *robots* and the technique is dubbed *capture-replay testing methodology*.

Java 2 Micro Edition environment (J2ME) lacks most of the above facilities for implementing GUI automation. All existing products (research or commercial) for testing mobile

TABLE 1.1: Differences between development and testing of mobile and traditional (desktop and server) applications.

| Element                      | Mobile  | Traditional   |
|------------------------------|---|---|
| Test recording               | Lack of programmatic access to recording GUI events. Emulation of user interaction impossible.  | Standard <code>java.awt.Robot</code> class for recording GUI events.  |
| Deployment automation        | Tedious (manual) routine of on-device deployment and testing.   | Deployment usually fully automatic. Testing and harvesting test results automatic and relatively easy.            |
| Test environment differences | Differences across devices (different virtual machines, varying memory and resource availability). Requirement to run tests on all possible configurations. | Virtually identical development and deployment/testing environment. In very rare cases operating-system specific. |
| Programming interfaces       | A number of non-standard APIs and proprietary solutions (playing sounds, access to external ports, access to the current display).                          | More mature and standard APIs, portable across JVMs from different vendors.                                       |

applications in J2ME are very simple and lack capture-replay testing support. This fact raises the following questions:

- In spite of technical difficulties, is it possible to devise a cross-platform architecture facilitating unit and integration testing of mobile applications? How much overhead (code, time) is required for running such a solution?
- Is there an industry need for integration tests aimed for mobile applications?
- How much time and resources can we save by implementing semi- or fully automatic integration tests in J2ME?

The first question is very technical in nature, but poses great technical difficulties because of the limited functionality available in the J2ME environment. We believe overcoming such major obstacles, although definitely with a technical in nature, qualifies as a research activity. In this thesis we demonstrate an architecture that allows capture and replay of GUI events in the J2ME environment by means of dynamic code injection. This is a significant improvement over all the products available in the literature and on the market. We also estimate the overhead of this solution in terms of space and time needed for its execution at runtime.

To answer the second question we present some preliminary results and feedback from the evaluation of our proposal in a leading commercial company developing mobile navigation systems in Java.

As for the last question, there seems to be no direct answer to how using regression tests translates into economic value. While we could try a controlled user-study to assess the time or effort savings gained from using regression tests, this kind of experiment is always subjective and lacks the real-life constraints of a commercial company's environment. This problem is actually omnipresent with respect to software testing in general – common sense suggests tests provide certain measurable value, but hard estimation of this value is very difficult.





## 1.2 Motivation and Goals

In reality, mobile application development stops at the level of unit tests because of lack of tools and techniques that would allow proper acceptance and regression tests. One of the reason why this is the case is that programming and testing in J2ME is more difficult compared to desktop programs. - each device has slightly different hardware configuration, - "standard" APIs from various vendors differ, - a number of non-standard APIs and proprietary solutions exists.

First, in spite of the API specification, there is a wide range of hardware and software vendors. This leads to various incompatibilities and quirks - most notably that a piece of software must be repeatedly tested over and over on all available device/ software configurations. Additionally, the standard mobile environment does not provide any support for intercepting system events (the standard Java edition has a `java.awt.Robot` class for this). These issues made us think if it is at all possible to automate capture-replay tests in the J2ME environment.

Having this issues in mind we set following goals: - to design an architecture to perform GUI testing in J2ME environment, - overcome J2ME deployment/ development cycle problems, - ease the task of developing and testing of J2ME applications, - implement the proposed solution, - and finally test and evaluate it in practice.

## 1.3 Typographical Conventions

## 1.4 Thesis Outline

Chapter 1: Introduction In this chapter we introduce reader to the mobile application development process, and outline goals we have in our work.

Chapter 2: Background In this chapter we present theoretical background to software testing in general and to GUI and mobile application testing in particular. We also describe some of existing solutions that try to solve problem of mobile applications testing.

Chapter 3: RobotME Framework Design In this chapter we present design and high level architecture of RobotME framework.

Chapter 4: RobotME: Implementation This chapter detaily describe the implementation of our concepts.

Chapter 5: Experiments and Evaluation Here we present some interesting results from lab experiments we performed.

Chapter 6: Summary and Conclusions At the end we summarize what we intended to do and what we finally achieved.

## Chapter 2

# Background

### 2.1 Approaches to Software Testing

Software testing is a process of verifying the quality of computer programs to make sure they are doing what was expected in a consistent, error-free manner. But software testing in practice depends on *how* and *when* it takes place in the development process. We can distinguish several types of tests [KFN99, Jef05]. *Unit testing* concentrates on low-level pieces of software, such as classes and methods. These tests are typically a responsibility of the programmer transforming the design into implementation. *Acceptance tests* occur at the end of the development process – when the software is confronted with its initial requirements specification and expectations of target users. *Integration tests* (also called *regression tests*), which we focus on in this paper, happen in between unit and acceptance tests and cover larger blocks of the program, often the entire product. By running integration tests frequently, we ensure all the modules work together as a whole and provide results consistent with previously released versions of the software. Regression tests are thus a quality control aspect – they allow early detection of the program's suspicious behavior caused by changes (refactorings or new features) introduced to the product. This kind of constant software testing in anticipation of potential errors is part of most modern software development methodologies and is called the *continuous integration* principle [FF].


Focusing only on GUI testing we may **distinguish** several techniques **specially** designed **to this** kind of **software** . One of more well known and often used **technique** is capture-replay which **require** some recording tool that will be able to record all important actions performed by the tester while using the application under test. The recording can then **by** replayed by the tool, thereby repeating test exactly as it was performed manually. Unfortunately most of the existing tools, due to the technical constraints and difficulties lack of the automatic verification module, leaving this aspect to the tester itself.

The other technique related to GUI testing (but not only) is called scripting. Recording a test case being performed manually results in one (usually long) linear script that can be used to replay the action performed by the manual tester. Doing the same for a number of test cases will result in one script for each test case. The script is really a form of computer

program, a set of instructions for the test tool to act upon. Having one such program for every test case is not efficient since many test cases share common actions (such as 'show client detail', 'go to help screen', etc). This will lead to higher maintenance costs than the equivalent manual testing effort because every copy of a set of instructions to perform a particular action will need changing when some aspect of that action changes.

## 2.2 Review of Existing Testing Frameworks



We can distinguish two different types of related works: research about GUI testing principles (theory) and programs allowing automated GUI tests in practice. The former topic has been broadly covered in research literature [KFN99, Jef05, Wil99] and **therefor we omit an in-depth background here in favor of surveying testing tools available at the moment.**  We reviewed the existing products (commercial and open source) that somehow tackle the problem of testing mobile applications in order to see to what extent they allow automated integration tests.

An open source project J2MEUNIT [B] can run simple unit tests. It does not allow testing application as a whole and the test cases are hard to maintain. It is also not possible to integrate J2MEUNIT into an automatic build process because results of performed tests must be verified by the programmer (which excludes its use for integration testing). Sony Ericsson's MOBILE JUNIT [C] is a more advanced framework, allowing unit testing on the device and collecting code coverage statistics automatically while running tests. MOBILE JUNIT is bound exclusively to the Microsoft Windows operating systems and on-device testing is limited to Sony Ericsson's telephones. Moreover, the tool's configuration and launching is quite complex and involves a pipeline of different tools which cannot be separated. Recently a few other toolsets similar to Sony's emerged: MOTOROLA GATLING [F] and CLDCUNIT [G] for example. The functionality they offer is close to that of Sony's.

So far we have only mentioned unit testing frameworks. One solution going beyond that point, towards GUI testing, is IBM's Rational Test RT, in short TESTRT [D]. TESTRT is a commercial package with a custom implementation of unit tests. The program allows GUI testing, but only on so-called *emulators* (software substitutes of real devices), not on the devices themselves. The simulation script knows nothing about the emulator or about the mobile environment – it merely replays the operating system's events such as keyboard actions or mouse clicks at certain positions over the emulator window. This implies that the product is testing a software emulation of a real device rather than the program running on that device. Unfortunately, TESTRT also lacks an automated test verification mechanism, the programmer is responsible for checking whether the replayed test passed or not.

A more sophisticated testing solution comes from Research In Motion and is bundled with development tools for this company's flagship device BlackBerry. The software emulator of a BlackBerry device (called FLEDGE [E]) is equipped with a controller tool that can interpret predefined event scripts. These scripts can contain events such as: starting and pausing the application, changing the readouts of GPS location API for devices supporting

GPS positioning, generating keypad and other input device events, generating various phone events such as remote phone calls or changing battery level. BlackBerry's controller has several limitations: it runs only with the simulator, not with real devices, it lacks an automated test verification mechanism (assertions) and, most of all, the developers are unable to record test scenarios – all scripts must be written by hand prior to testing.

The conclusion from the list above is that in spite of the evolving theory of GUI testing, practical implementations for testing mobile applications remain within the domain of the simplest unit and limited GUI tests.

## 2.3 J2ME Architecture

Sun Microsystems defines J2ME as 'a highly optimized Java run-time environment targeting a wide range of consumer products, including pagers, cellular phones, screen-phones, digital set-top boxes and car navigation systems'. Announced in June 1999 at the JavaOne Developer Conference, J2ME brings the cross-platform functionality of the Java language to smaller devices, allowing mobile wireless devices to share applications. With J2ME, Sun has adapted the Java platform for consumer products that incorporate or are based on small computing devices.

J2ME uses configurations and profiles to customize the Java Runtime Environment (JRE). As a complete JRE, J2ME is comprised of a configuration, which determines the JVM used, and a profile, which defines the application by adding domain-specific classes to the J2ME configuration to define certain uses for devices. The configuration defines the basic run-time environment as a set of core classes and a specific JVM that run on specific types of devices.

The following graphic depicts the relationship between the different virtual machines, configurations, and profiles. It also draws a parallel with the J2SE API and its Java virtual machine. While the J2SE virtual machine is generally referred to as a JVM, the J2ME virtual machines, KVM and CVM, are subsets of JVM. Both KVM and CVM can be thought of as a kind of Java virtual machine – it's just that they are shrunken versions of the J2SE JVM and are specific to J2ME.

The modular design of the J2ME architecture enables an application to be scaled based on constraints of a small computing device. J2ME architecture doesn't replace the operating system of a small computing device. Instead, J2ME architecture consists of layers located above the native operating system, collectively referred to as the Connected Limited Device Configuration (CLDC). The CLDC, which is installed on top of the operating system, forms the run-time environment for small computing devices.

The J2ME architecture comprises three software layers. The first layer is the configuration layer that includes the Java Virtual Machine (JVM), which directly interacts with the native operating system. The configuration layer also handles interactions between the profile and the JVM. The second layer is the profile layer, which consists of the minimum set of application programming interfaces (APIs) for the small computing device. The third layer

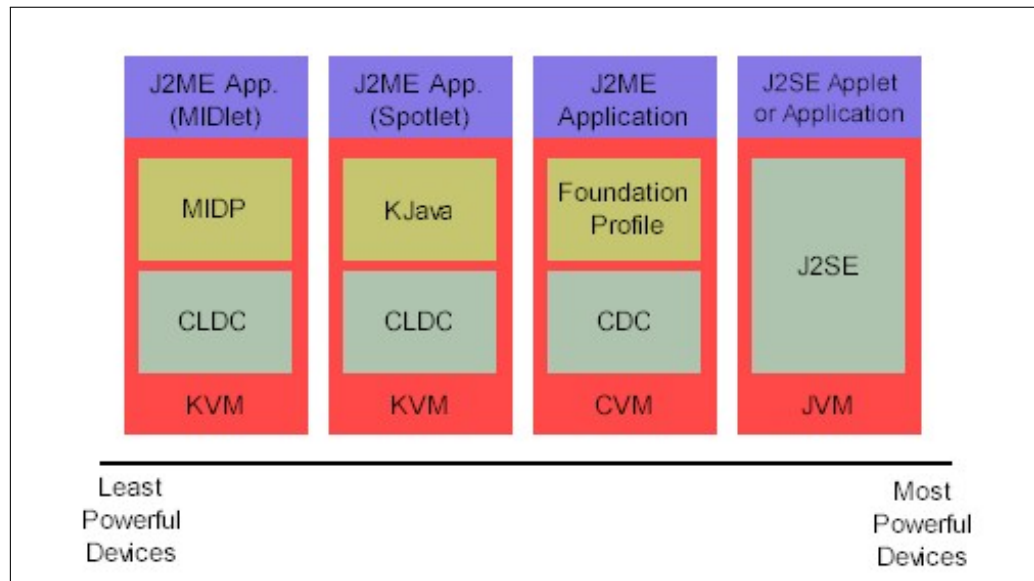


FIGURE 2.1: J2ME Architecture (from: "J2ME: Step by step" by IBM developerWorks).

is the Mobile Information Device Profile (MIDP). The MIDP layer contains Java APIs for user network connections, persistence storage, and the user interface. It also has access to CLDC libraries and MIDP libraries.

A MIDlet is a J2ME application designed to operate on an MIDP small computing device. A MIDlet is defined with at least a single class that is derived from the `javax.microedition.midlet.MIDlet` abstract class.

**Event Handling:** A MIDlet is an event-based application. All routines executed in the MIDlet are invoked in response to an event reported to the MIDlet by the application manager. The initial event that occurs is when the MIDlet is started and the application manager invokes the `startApp()` method.

The `startApp()` method in a typical MIDlet contains a statement that displays a screen of data and prompts the user to enter a selection from among one or more options. The nature and number of options is MIDlet and screen dependent.

A Command object is used to present a user with a selection of options to choose from when a screen is displayed. Each screen must have a `CommandListener`. A `CommandListener` monitors user events with a screen and causes the appropriate code to execute based on the current event.

**User Interfaces:** The design of a user interface for a MIDlet depends on the restrictions of a small computing device. Some small computing devices contain resources that provide a rich user interface, while other more resource-constrained devices offer a modest user interface. A rich user interface contains the following elements, and a device with a minimal user interface has some subset of these elements as determined by the profile used for the device.

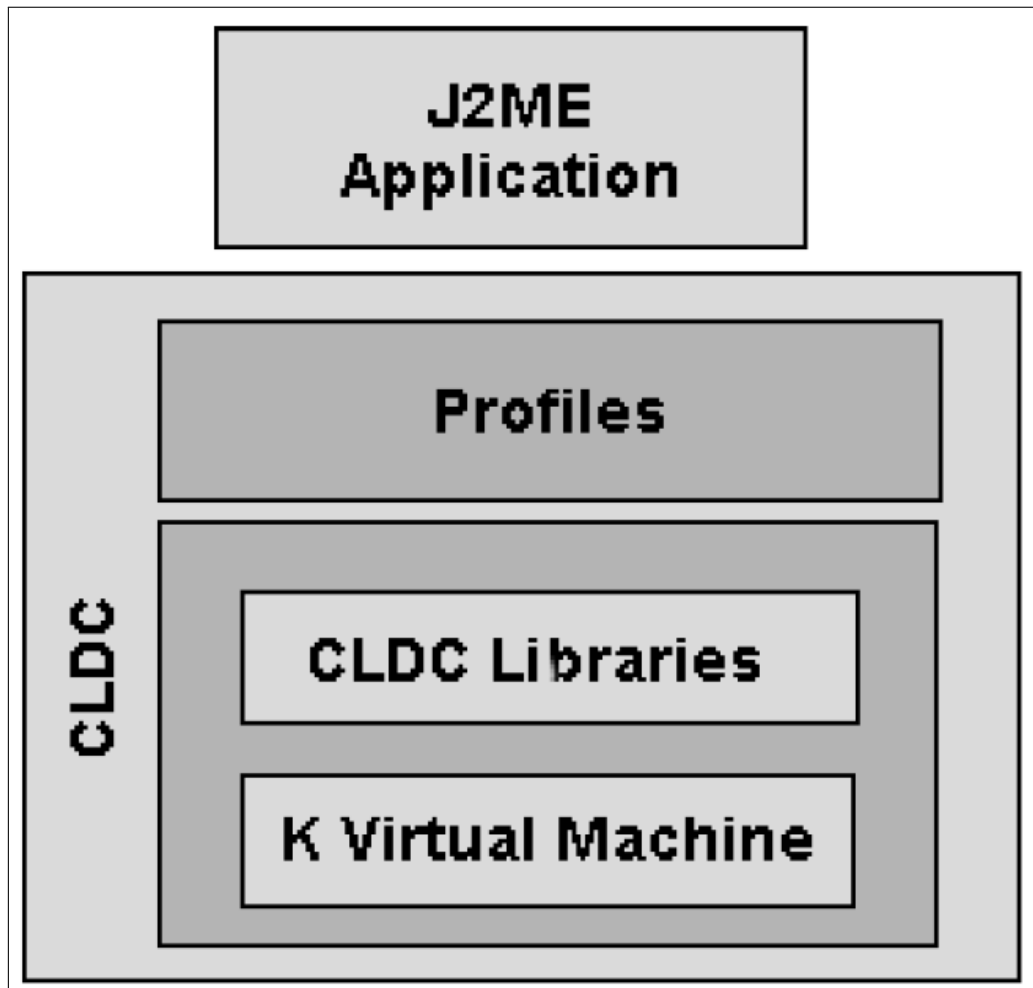


FIGURE 2.2: J2ME Architecture cd (from: "Wireless Programming with J2ME: Cracking the Code" by Dreamtech Software India, Inc., Team).

A Form is the most commonly invoked user interface element found in a MIDlet and is used to contain other user interface elements. Text is placed on a form as a `StringItem`, a `List`, a `ChoiceGroup`, and a `Ticker`.

A `StringItem` contains text that appears on a form that cannot be changed by the user. A `List` is an itemized options list from which the user can choose an option. A `ChoiceGroup` is a related itemized options list. And a `Ticker` is text that is scrollable.

A user enters information into a form by using the `Choice` element, `TextBox`, `TextField`, or `DateField` elements. The `Choice` element returns an option that the user selected. `TextBox` and `TextField` elements collect textual information from a user and enable the user to edit information that appears in these user interface elements. The `DateField` is similar to a `TextBox` and `TextField` except its contents are a date and time.

An `Alert` is a special Form that is used to alert the user that an error has occurred. An

Alert is usually limited to a `StringItem` user interface element that defines the nature of the error to the user.

To directly use the display and the low-level user interface API, the developer uses the `Graphics` and `Canvas` classes. The `Canvas` class provides the display surface, its dimensions, and callbacks used to handle key and pointer events and to paint the display when requested. The methods of this class must be overridden by the developer to respond to events and to paint the screen when requested. The `Graphics` class provides methods to paint lines, rectangles, arcs, text, and images to a `Canvas` or an `Image`. To have access to even more low-level events programmer can use `GameCanvas` class that is direct subclass of `Canvas`.

## Chapter 3

# RobotME Framework Design

We divided the problem into fairly independent goals. The first goal was to design a mechanism that would allow us to intercept and record the events resulting from user's interaction with the program (running on an emulator or a real device). This is called the *recording phase*. The second goal was to programmatically simulate the previously recorded events (user actions) – this is called the *replay phase*. Finally, we compare the initial recording with stimuli resulting from the replayed events; certain *assertions* are checked to ensure the program followed identical sequence of state transitions (this implies a correct outcome of the entire test). Note that states can be fairly low-level, such as action selection, but also high-level, perhaps even explicitly hardcoded in the program by the developer. We took extra care to facilitate future maintenance of the recorded scripts. Unlike with desktop applications, where an event is typically described by a mouse position or some obscure component identifier, our events are described with identifiers meaningful to the programmer (an action's label for example). Our goal is to make the recorded script comprehensible and comparable to a typical (unstructured) use-case scenario used in requirements engineering.

### 3.1 High-level Architecture

When talking about high-level architecture of the created RobotME Testing Framework we must distinguish several different parts of the framework: - **Framework core** – it is the groups of classes that are used both during capturing and replaying phases of the application under test. Here is most of the code that perform user inputs capturing and simulating and forwarding events to the Log Server module. - **Enhancer** – it is a separate tool that gets as its input Java archive (JAR) file with the compiled Java mobile applications and returns 'enhanced' the same Java archive with added code related to the RobotME Testing Framework (specialized code is 'injected' into original JAR file). It works in two modes: one dedicated to recording/ capturing phase that creates enhanced JAR with injected code that is able to intercept user events; and second dedicated to replaying phase that creates enhanced JAR with injected code that is able to simulate previously captured events. - **Log Server** – this is yet another tool that is deployed on some PC machine and is constantly listening for in-



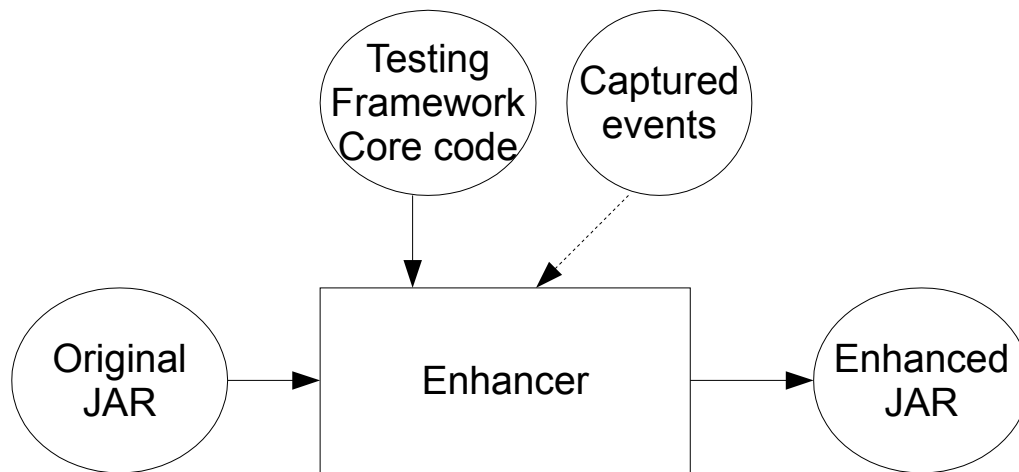


FIGURE 3.1: Enhancement process.

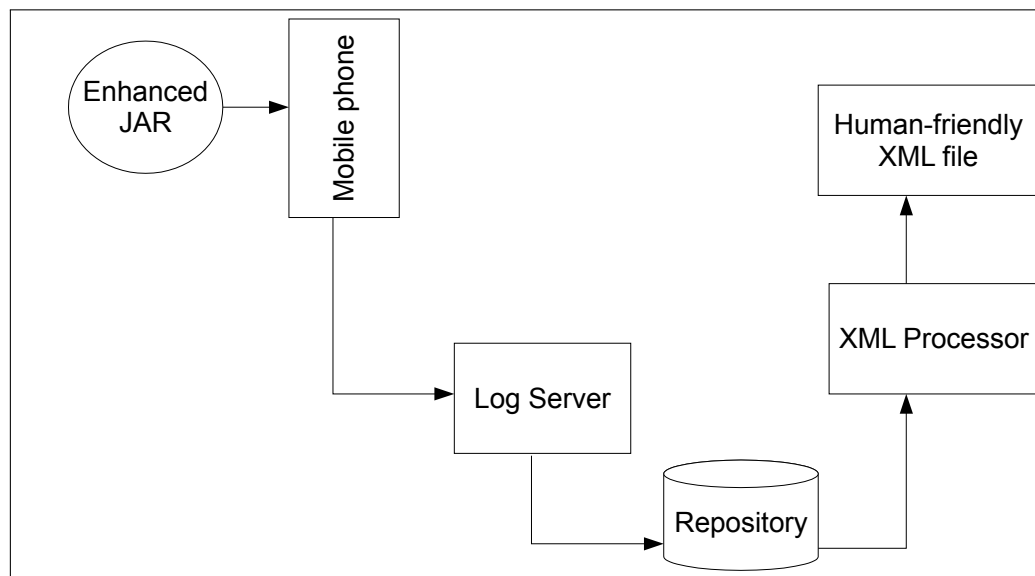


FIGURE 3.2: Typical capturing/ replaying process.

coming messages from the mobile phone under test. It stores received messages (events) from the mobile phone in the byte protocol format. - XML Processor – another tool that its responsibility is to decode byte protocol with stored events in it from mobile phone into human-friendly XML form and vice versa. - Application under test – it is really not part of the RobotME Testing Framework, but is important part of the testing scenario and forms an input to the RobotME Testing Framework therefore it is important to mention it.

Going into further details of RobotME core architecture (but still not going into low-level implementation detail) several most important classes must be described and interrelations

between them. All of the important classes are depicted on the figure below.

**RobotMERecorder** It is the central point of all of the framework activities and the only class that is directly invoked from the injected code. It preserves all important information about the running Midlet like: connactions between displaying screens and commands on them and what screen is currently displayed.

**RobotMEReplaying** This class derives from the RobotMERecorder class because from the capture-replay framework point of view replaying process is an extended version of recording process. It is because during replaying process to be able to watch what is going on in the application we must observe the application, therefor record each occurred events. Note that RobotMEReplaying is used only during replaying process, but RobotMERecorder is used both during capturing and replaying process. The important parts in which this class extend RobotMERecorder class is mostly by adding ability of starting and stopping replaying process.

**RobotMEReaplyingThread** it is usual Java thread (derives from standard Java Thread class). Responsibility of this class is to constantly reading remaining events from the repository (usually repository is plain file located in the same JAR as Midlet application) and simulating read events in appropriate time intervals.

**DisplayableState** it is common knowledge that all action usually has some reaction (visible or not). Thesame is with mobile application: all user or system-level actions (events) usually leads to some reaction (both visible on the screen but not always). DisplayableState abstract class is the way of storing all those reactions that are visible on the screen. There are many implementations of this class: AlertState, CanvasState, FormState, ListState and TextBoxState. All screen types existing in J2ME specification has its own DisplayableState implementation that knows how to obtain and preserve state of such a screen.

**AbstractWidgetState** one of the screen types could have very complex internal state, it is Form. This kind of screen has its own implementation of DisplayableState - it is FormState, but FormState implementation preserve only general information about Form, like its title and ticker text (ticker is kind of scroll bar that is constantly scrolling some text on top or on the bottom of mobile phone screen). Forms as defined in the J2ME specification could have so called Items in it. The examples of predefined items are: simple labels, text fields, data fields, choice groups (radio buttons or check-boxes), gauges, images and more, even custom items (prepared entirely by the programmer, in current prototype implementation our testing framework does not support this kind custom of Items). Each of this items has its own internal state, and taken together form state of their parent Form. AbstractWidgetState abstract class is the parent class for all of the implementations that store information about internal states of the items. Some of the implementations are: ChoiceGroupWidgetState, DateFieldWidgetState, ImageItemWidgetState, TextFieldWidgetState and others. Each of implementation of

`AbstractWidgetState` knows how to retrieve state of its `Item` and how to preserve it (serialize to the event code).

**LogHandler** is the handler responsible for logs, for forwarding serialized events to appropriate receiver. Examples of such receivers could be: remote PC with socket, bluetooth, infra-red or serial connection, or even local (on the phone) receiver that stores events in the local store (so called Record Store).

**LogEvent** is the base class for all kind of events possible to intercept during program execution. Detailed list of every event types is presented in the next section.

**Replayable** is a base interface for implementation of class that are able to simulate (replay) previously recorded events.

**Assertion** after each of simulated action state of the whole application is verified (display screen, internal memory etc). This kind of verification is called assertion, and if the verification process does not succeed Failure event is stored as the occurred event (it is kind of internal special purposes event).

### 3.2 Recorded Event Types

RobotME Testing Framework was created with the possibility of easily extending events set in mind. In the current prototype implementation we were concentrated on implementing interception of most occurred events, which are:

- intercepting of modification of every kind of screens, also with complex Canvas and Forms (excluding Custom Items inside Forms, it is possible to support such kind of Items, but needs to introduce direct dependency between application under test and RobotME Testing Framework, it is subject to future extends),
- intercepting invocation of Commands actions,
- intercepting of low-level key events,
- intercepting of low-level pointer events.

This set of interceptors allows to test most of the currently existing mobile applications. The other kind of interceptors that are planned to implement in the future:

- intercepting modifications of the local record store
- intercepting invocations of all phone calls
- intercepting incoming and outgoing SMS, MMS i CBS messages
- intercepting incoming and outgoing HTTP, socket, bluetooth and other kind of connections

- intercepting sound events
- intercepting every access to System.out and System.err streams

Most of mobile phone hardware vendors have their own API to fulfill special needs like: LED lighting, access to vibra or mobile phone contact book. We are not considering implementation of such proprietary events, but left our RobotME Testing Framework open and it will be possible to implement such events by the interested in them parties as a future extends.

### 3.3 Problems

Creating tool that operates on the level of other applications compiled Java byte code needs to resolve several problems that are specific to this kind of tools. We as a tool provider can't assume what programming techniques use each developer, everybody use slightly different attempt to solve common programatic problems: instantiating objects, creating constant values or modifying behaviour of existing classes. All of this and many other variations must be supported by the tool that want to be comonly used, and all of this variations must be properly differentiated by the tool during examining application byte code.

Lets provide some examples: Testing tool must intercept every attempts to add Command objects to existing screens (by invoking addCommand method). Programmer may add Commands by many ways, for example, by adding class-level static fields:

```

1 public class ClassUnderTest {
2
3     private static final Command MY_COMMAND = new Command("Cmd label", Command.OK, 1);
4
5     public ClassUnderTest() {
6         Form f = new Form("Title");
7         f.addCommand(MY_COMMAND);
8     }
9 }

```

On the Java bytecode level, invoking only addCommand() methods looks like this:

```

1 ALOAD 1
2 GETSTATIC ClassUnderTest.MY_COMMAND : Ljavax/microedition/lcd/Command;
3 INVOKEVIRTUAL javax/microedition/lcd/Displayable.addCommand(Ljavax/microedition/lcd/Command;)V

```

which means: load first local variable onto the stack (Form f), load static class field onto the stack, invoke addCommand method of the Displayable instance with the arguments from the stack.

The other possible attempt to do the same thing is using object-level fields:

```

1 public class ClassUnderTest {
2
3     private final Command MY_COMMAND = new Command("Cmd label", Command.OK, 1);
4
5     public ClassUnderTest() {
6         Form f = new Form("Title");
7         f.addCommand(MY_COMMAND);
8     }
9 }

```

```
9 }
```

On the Java bytecode level, invoking only `addCommand()` methods in this variant looks like this:

```
1  ALOAD 1
2  ALOAD 0
3  GETFIELD ClassUnderTest.MY_COMMAND : Ljavax/microedition/lcd/Command;
4  INVOKEVIRTUAL javax/microedition/lcd/Displayable.addCommand(Ljavax/microedition/lcd/Command;)V
```

which means: load first local variable onto the stack (Form `f`), load ‘this’ variable onto the stack, load object-level field onto the stack (and in the same step remove ‘this’ variable from the stack), invoke `addCommand` method of the `Displayable` instance with the arguments from the stack.

The second example is only slightly different from the first one. But it is possible to do the same thing in completely different manner, using factory methods:

```
1 public class ClassUnderTest {
2
3     private static final Command getMyCommand() {
4         return new Command("Cmd label", Command.OK, 1);
5     }
6
7     public ClassUnderTest() {
8         Form f = new Form("Title");
9         f.addCommand(getMyCommand());
10    }
11
12 }
```

with Java bytecode:

```
1  ALOAD 1
2  INVOKESTATIC ClassUnderTest.getMyCommand()Ljavax/microedition/lcd/Command;
3  INVOKEVIRTUAL javax/microedition/lcd/Displayable.addCommand(Ljavax/microedition/lcd/Command;)V
```

or even providing `Command` objects while adding it to the `Form`:

```
1 public class ClassUnderTest {
2
3     private static final Command getMyCommand() {
4         return new Command("Cmd label", Command.OK, 1);
5     }
6
7     public ClassUnderTest() {
8         Form f = new Form("Title");
9         f.addCommand(new Command("Cmd label", Command.OK, 1));
10    }
11 }
```

with Java bytecode:

```
1  ALOAD 1
2  NEW javax/microedition/lcd/Command
3  DUP
4  LDC "Cmd label"
5  ICONST_4
6  ICONST_1
7  INVOGESPECIAL javax/microedition/lcd/Command.<init>(Ljava/lang/String;II)V
```

```
8 INVOKEVIRTUAL javax/microedition/lcd/Displayable.addCommand(Ljavax/microedition/lcd/Command;)V
```

Event different approach could be subclassing Command class (subclassing Command class has no real value, but is technically correct):

```
1 public class ClassUnderTest {
2
3     public ClassUnderTest() {
4         Form f = new Form("Title");
5         f.addCommand(new Command("Cmd label", Command.OK, 1) {
6             public String getLabel() {
7                 return "Other Cmd label";
8             }
9         });
10    }
11
12 }
```

with Java bytecode:

```
1 ALOAD 1
2 NEW ClassUnderTest$1
3 DUP
4 ALOAD 0
5 LDC "Cmd label"
6 ICONST_4
7 ICONST_1
8 INVOGESPECIAL ClassUnderTest$1.<init>(LClassUnderTest;Ljava/lang/String;II)V
9 INVOKEVIRTUAL javax/microedition/lcd/Displayable.addCommand(Ljavax/microedition/lcd/Command;)V
```

There are also cases that can't be resolved by simply investigating compiled byte code once, because of the complexity between interdependencies of standard API code and code created by the programmer. The only solution to that cases is having two-stages review of the compiled code: first review creates map of the interdependencies, and second review basing on the created map and currently investigating code looks for the appropriate injection points. Example of such complicated code could be creating Form screen by simply using standard API:

```
1 Form f = new Form("Title");
```

and creating object that its class derives from standard Form class:

```
1 public class MyOwnForm extends Form {
2
3     public MyOwnForm(String s) {
4         super(s);
5     }
6
7 }
8
9 MyOwnForm f = new MyOwnForm("Title");
```

Creating first object in byte code level looks like this:

```
1 NEW javax/microedition/lcd/Form
2 DUP
3 LDC "Title"
4 INVOGESPECIAL javax/microedition/lcd/Form.<init>(Ljava/lang/String;)V
5 ASTORE 1
```

and creating second object:

```
1  NEW MyOwnForm
2  DUP
3  LDC "Title"
4  INVOKESPECIAL MyOwnForm.<init>(Ljava/lang/String;)V
5  ASTORE 1
```

As you see, second example has no direct relation with the standard J2ME API, therefore it is impossible to differ only from this snippet of code what type of object `MyOwnForm` is. In the Java runtime we may use ‘instanceof’ statement or `getClass()` method to verify it, but it is impossible to do similar thing while statically investigating Java code.

An additional problem to solve in the recording phase was related to storage of the recorded events. Saving all events directly on the device was inconvenient because we could collide with the application’s data or exceed the device’s limited capacity (memory or persistent storage). Eventually we decided to transmit all events directly over the wire during the simulation and have an option to save them locally in case network protocol is unavailable on the device.

All of this and many other variants must be properly distinguished and resolved by the RobotME Framework.

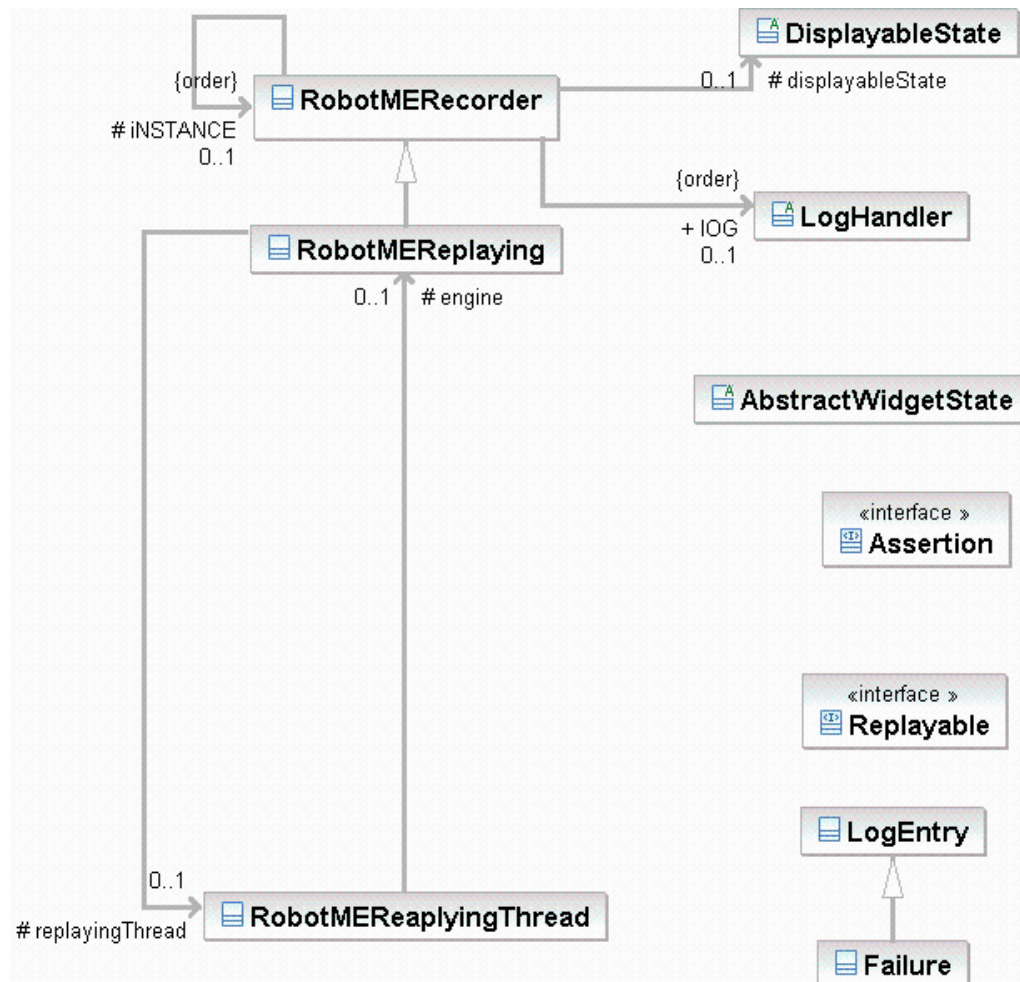


FIGURE 3.3: Most important classes from RobotME core.



## Chapter 4

# RobotME: Implementation

In the several following sections we will present in deep details each RobotME Framework components' implementation characteristics.

### 4.1 System Components

As was outlined previously in High-level Architecture section the main system components are: Framework core itself, Enhancer, Log Server, XML Processor. There do not exist separate components such as Recorder or Replayer (they are part of the Framework core being precise) but they are substantial parts of the RobotME Testing Framework and created by us source code therefore we will describe ideas behind this modules separately to be more accurate and relevant.

#### 4.1.1 Enhancer

As was said in previous chapter, this component is a separate tool that gets as its input Java archive (JAR) file with the compiled Java mobile applications and returns "enhanced" the same Java archive with the code related to the RobotME Testing Framework (specialized code is "injected" into original JAR file). It works in two modes: one dedicated to recording/capturing phase that creates enhanced JAR with injected code that is able to intercept user events; and second dedicated to replaying phase that creates enhanced JAR with injected code that is able to simulate previously captured events.

From the implementation point of view it is a separate Java J2SE program, implemented as an ANT specialized task (so it can work as a stand-alone tool from the console but also as a part of building process – note: continuous integration process – or even from inside IDE like Eclipse or NetBeans). The sole responsibility of this tool is to investigate compiled Java code of the input JAR file, intercept appropriate parts of the code (in AOP terms they are called join points) modify them in expected way and compress back to the output JAR file.

Java compiled bytecode is kind of assembler language with low-level op codes (mnemonics) and old-style jumps to the appropriate offsets in the code. It is very difficult to investi-

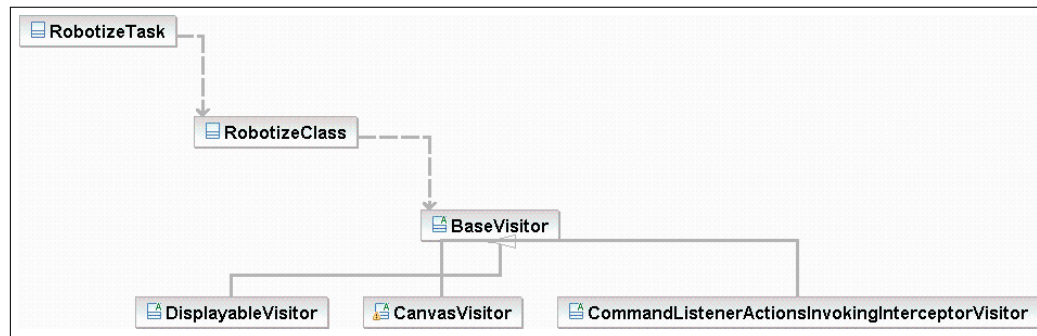


FIGURE 4.1: Enhancer implementation.

gate such code and look for appropriate places to intercept. Programmer that want to do this must feel comfortable with great amount of different mnemonics Java has. The approach to simplify techniques of Java bytecode investigations are various aspect-oriented frameworks like AspectJ, AspectWerkz or even Spring Framework build-in ASM facility. Unfortunately this frameworks offer too little comparing to our needs. There is finite set of transformations that AOP (aspect-oriented programming) techniques can do (for example Spring Framework AOP mechanism works only by creating proxies for classes it transforms. AspectJ can include various Java code inside Java methods, but it also includes code strictly dependent to own API which is not acceptable in our case – mobile applications must be as little as is possible). Due to all of this problems we decided not to use AOP frameworks at all, and concentrate on the Java bytecode modification frameworks that offer more low-level access to the modifying bytecode but also offer some help with transformations techniques. To our knowledge there are two most often used open-source frameworks that fulfill such requirements: ASM and Apache BCEL. Both of this frameworks were thoroughly tested and verified by us, we chose ASM as a foundation for our Enhancer for several reasons:

- the company that stands behind it, France Telecom R&D Center, guarantee good quality of the product
- it has substantially smaller memory and computation overhead
- it is easier to use, and cover both tree-like investigation of the Java bytecode as well as event-like (simillary to XML code parsing where this terms are called in order: DOM and SAX techniques).

Having chosen bytecode modification framework we created base Enhancer architecture around the ASM framework and its processing scheme based on visitor design pattern (event-like). The internal Enhancer's architecture is depicted on the following class diagram.

Responsibilities of presented classes are:

RobotizeTask – it as ANT task and entry point to the Enhancer tool. It posess reference to the RobotizeClass.

RobotizeClass – main class responsible for delegating enhancing task to the appropriate utility components including: decompressing JAR file, iterating over Java class files, writing transformed bytecode to the Java class file, compressing back transformed code to the output JAR file.

BaseVisitor – each of transformation type is implemented in its own separate Java class and BaseVisitor forms a base implementation for all of transformations and offer most frequently used methods like obtaining Java bytecode and managing lifecycle of the transformation process. It has many specific implementations some of which are depicted on the diagram as classes that derives from it.

#### 4.1.2 Recorder

Recorder starts working when the mobile application (Midlet) starts running inside JVM. Thanks to injected into the original mobile application code delegates that invoke RobotME Framework internal classes in appropriate moments of mobile application lifecycle the Framework is able to monitor changes in mobile application internals. The main delegate that takes role of main entrance to the RobotME Testing Framework core is RobotMERecorder. It is singleton class and only one instance of this class exists per one running mobile application. It is able to differentiate between low-level events like: key events, pointer events and high-level events like commands invoked by the user and pass appropriate data about the intercepted event to the cooperating objects like: implementations of DisplayableState class and LogHandler and further to Log Server component. There are cases when event occurred in the mobile application does not lead to any changes in the internal state of the mobile applications. It is also RobotMERecorder responsibility to spot that situations and behave accordingly (log an intercepted event but with the information that it is not candidate for the assertion verification because it leads to no visible internal state changes).

#### 4.1.3 Replayer

Replayer is an extended version of Recorder component (reason to such a case was mentioned in the High-Level architecture chapter). From the Framework's internals point of view Replayer are really two cooperating things: Java thread that constantly reads events from some kind of repository (in current prototype implementation only files bundled into JAR file are supported), basing on read event (LogEntry) it creates corresponding Replayable object that knows how to replay given LogEntry. The second thing is RobotMEReplayable class. Responsibility of this class is simialar to RobotMERecorder – to intercept every event interesting from the RobotME Testing Framework point of view and perform verification of the internal mobile application state. If state of the mobile application violates those that was stored in the application usage scenario assertion is triggered (assertions are further described below in the Assertions and Failure Notification section). If state of the mobile application is as expected simulator runs without stopping itself.

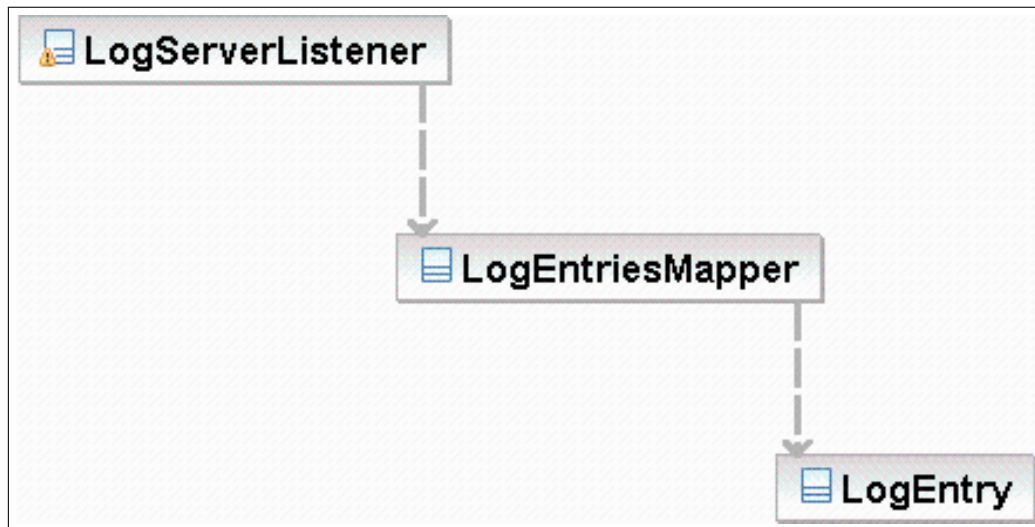


FIGURE 4.2: Log Server implementation.

#### 4.1.4 Log Server

Log Server is another separate Java J2SE tool (similarly to Enhancer component). It works on PC machine and its responsibility is to constantly listening for the incoming events from the mobile programs under test and store each received event (not interpreting them). Log Server is internally dependent on the Framework Core component and uses some of its classes, mainly as events encapsulations.

Log Server architecture is rather straightforward. It consists of one main class: `LogServerListener`, that internally creates separate thread for each incoming connections from the mobile application under test, therefore is able to receive separate events from separate mobile applications, each connection not interfere with any other.

`LogServerListener` is build in the way that it can be abstracted from the nature of the underlying connection. It does not matter if the connection is performed as a bluetooth or infra-red transfer, GPRS internet transfer or even simply serial port connection (but what is usually important for the users is a price, some of above connections are free of charge while the others might not be). `LogServerListener` treat them in the same manner – as a form of events transfer.

`LogEntriesMapper` – is a class that knows how to create `LogEntry` class instance from the received by `LogServerListener` event. Each `LogEntries` and also each events are distinguishable by its unique ID number. To allow future implementations of 3rd party special events (for example Nokia phones specific events) each ID has its prefix and every internal events implemented as a part of the core RobotME Testing Framework has its own reserved prefix. There is also reserved prefix for custom, 3rd parties event IDs.

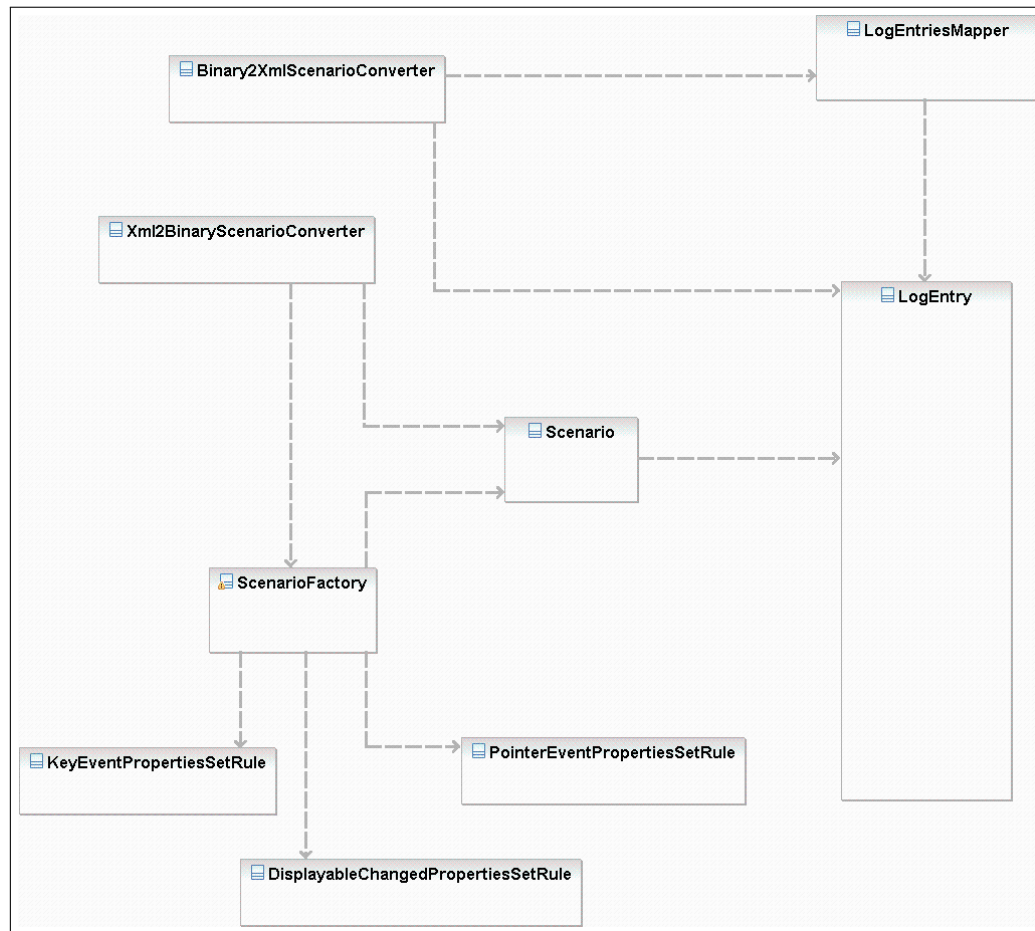


FIGURE 4.3: XML Processor implementation.

#### 4.1.5 XML Processor

XML Processor is yet another separate Java J2SE tool, that has important tasks to provide for whole RobotME Testing Framework. It is responsible for converting stream of events (stored as a compact bytecode protocol) to the human-friendly XML form of events. Its other task is also to convert events back from human-friendly XML form into compact bytecode protocol ready to replay on the mobile application.

Classes that take part in this processes are depicted on the following diagram.

Scenario – is a class that stores list of LogEntries that as a whole form a usage scenario of an associated mobile application. In its internal nature Scenario is very simple class, but fulfill important requirements. Having scenario object with LogEntries in appropriate order in it you know how mobile application is intended to use, therefore you can simulate it.

ScenarioFactory – is a factory method design pattern class that as an input takes two things: XML file with the payload of occurred events in the mobile application and a set of rules that describe how to transform XML constructions into list of LogEntry objects that

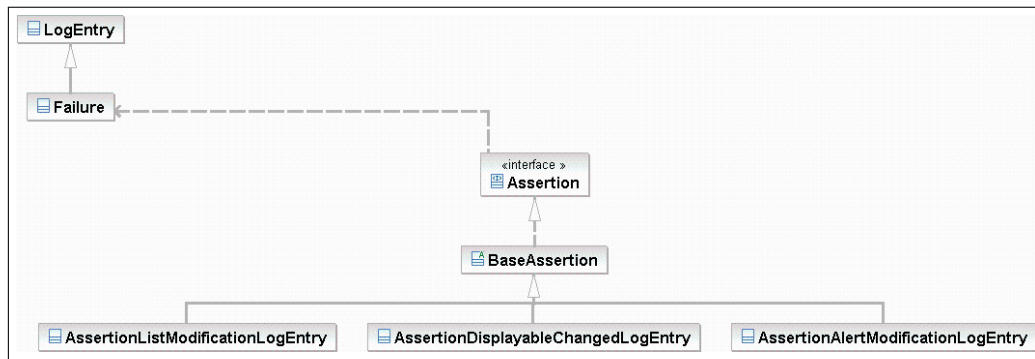


FIGURE 4.4: Assertions implementation.

will form Scenario object. Currently association of ScenarioFactory and a set of rules is hard-coded in the ScenarioFactory class, but it is done in very straightforward manner and it will be easy to move this association description to some configuration file. Thanks to it it will be simple for the third party companies that will want to add they own events in the RobotME Testing Framework in the future to do so just by modifying configuration file.

Xml2BinaryScenarioConverter and Binary2XmlScenarioConverter are two complementary converters that performs conversions from XML scenario format into compact bytecode format and vice versa. First of them realize its task by delegating internals of the conversion to ScenarioFactory class while the second uses LogEntriesMapper capabilities of creating LogEntry objects out of the bytecode protocol and LogEntry capability to convert itself into XML format event.

## 4.2 Assertions and Failure Notification

Each time Replayer component spots the difference between what was expected internal mobile application state and what was the real state assertion is triggered. Assertion is a base interface which must be implemented by each class that knows how to verify internal mobile application state (in some part). Because some of the verifying assertions tasks are common to most of the assertions implementation we decided to introduce BaseAssertions abstract class that has these common code. There are also several concrete implementations of this class, each verifies internal mobile application state for the part it is responsible for (for example some verified only List displayed on the screen, some verified other displayable objects, but there may exist implementations that verifies for example midlet record store contents, state of connections to the remote servers, or even time of the local timer if it might be important).

Every time when the assertion is violated Failure object is created (which is itself special implementation of LogEntry object therefor it can be serialized to the bytecode protocol and transferred to the Log Server component over a network connection, for further investigation of the assertion violation reasons) and passed to the Replayer component which decide

what do to next with it. In current prototype implementation there is only one option to do with Failures: terminate the whole mobile application and send every intercepted events to the Log Server both with Failure object itself. Than the real cause of the assertion violations might be further investigated by the tester. We are considering extending assertions so that there will be two types of them: - strong assertions – the same as in current implementation, - weak assertion – violation of such assertion will not cause the whole mobile application to terminate, but only appropriate Failer object will be transfered to the Log Server component. Sometimes this kind of assertions might be helpful with finding more errors in program execution than will be discovered only with strong assertions. Using this type of assertion we must be sure that after violation of weak assertion program will always be in consistent state, not causing further errors because of the weak assertion violation.

## Chapter 5

# Experiments and Evaluation

### 5.1 Lab Tests

Parallel to building prototype RobotME Framework we were also building mobile application that can be used to verify our framework capabilities. This application use each kind of visual component with each possible features that the testing framework supports.

Most of our tests were performed on emulators of real devices, usually using generic emulator provided by Sun Microsystem called WTK (Wireless Toolkit). It was because it is the fastest way of deploying mobile application and than executing it, and during constant development there is simply no time for time consuming deployment on real devices. After the prototype was created we performed several tests on real devices, including: SonyEricsson P900i (that supports CLDC 1.1 standard) and SonyEricsson K610i (that on the other hand supports CLDC 1.0 standard). Our framework supports all mobile devices that conforms to at least stanrds: MIDP 2.0 and CLDC 1.0 so theoretically these two devices should be fully supported by RobotME Framework.

Testing devices were connected to the Log Server through GPRS connection. For each of visual component our framework supports dedicated scenarios were recorded. Then we tried to replay previously recorded scenarios. When replaying scenario finishes with success (no assertion was violated) then we modified scenario so that it should no longer be correct and hoping that during next replaying process assertions will be violated and tests will not pass. It was indeed in our case. All of tests passed when it intened to, and all assertions was violated when we manually modified scenarios.

The example of one event that could be modified in the scenario so that whole scenario will not pass could be as follows. It presents snapshot of states items on the form that is currently shown on the device's screen.

Before modification:

```
1 <event level="INTERNAL" timestamp="1166970731531"
2   replayable="false" assertion="true">
3   <form-modification title="Form Title 1">
4     <items>
5       <choice-group label="Choice group label">
6         <selectedFlags>
```



```

7      <value>true</value>
8      <value>false</value>
9      <value>true</value>
10     </selectedFlags>
11     <strings>
12       <value>Option 1</value>
13       <value>Option 2</value>
14       <value>Option 3</value>
15     </strings>
16   </choice-group>
17   <date-field label="Date 1" date="1166970731529" />
18   <gauge label="Gauge 1" value="6" />
19   <image label="Img 1" altText="Alternative text 1" />
20   <spacer />
21   <string-item label="String item 1" text="Text 1" />
22   <text-field label="Text field 1" string="String 1" />
23 </items>
24 </form-modification>
25 </event>

```

After modification (form title was changed):

```

1 <event level="INTERNAL" timestamp="1166970731531"
2   replayable="false" assertion="true">
3   <form-modification title="Form Title 2">
4     <items>
5       <choice-group label="Choice group label">
6         <selectedFlags>
7           <value>true</value>
8           <value>false</value>
9           <value>true</value>
10        </selectedFlags>
11        <strings>
12          <value>Option 1</value>
13          <value>Option 2</value>
14          <value>Option 3</value>
15        </strings>
16      </choice-group>
17      <date-field label="Date 1" date="1166970731529" />
18      <gauge label="Gauge 1" value="6" />
19      <image label="Img 1" altText="Alternative text 1" />
20      <spacer />
21      <string-item label="String item 1" text="Text 1" />
22      <text-field label="Text field 1" string="String 1" />
23    </items>
24  </form-modification>
25 </event>

```

Please note that the same event XML definition could be used as an assertion (if appropriate attributes are set to: `replayable="false" assertion="true"`) but also could be used to simulate user interaction with the mobile application (if these attributes are set to: `replayable="true" assertion="false"`).

## 5.2 Real-life Use Case

As we were starting our work on this framework we were not sure if our idea is even possible to realize and implement, there are no similar frameworks for mobile platforms created

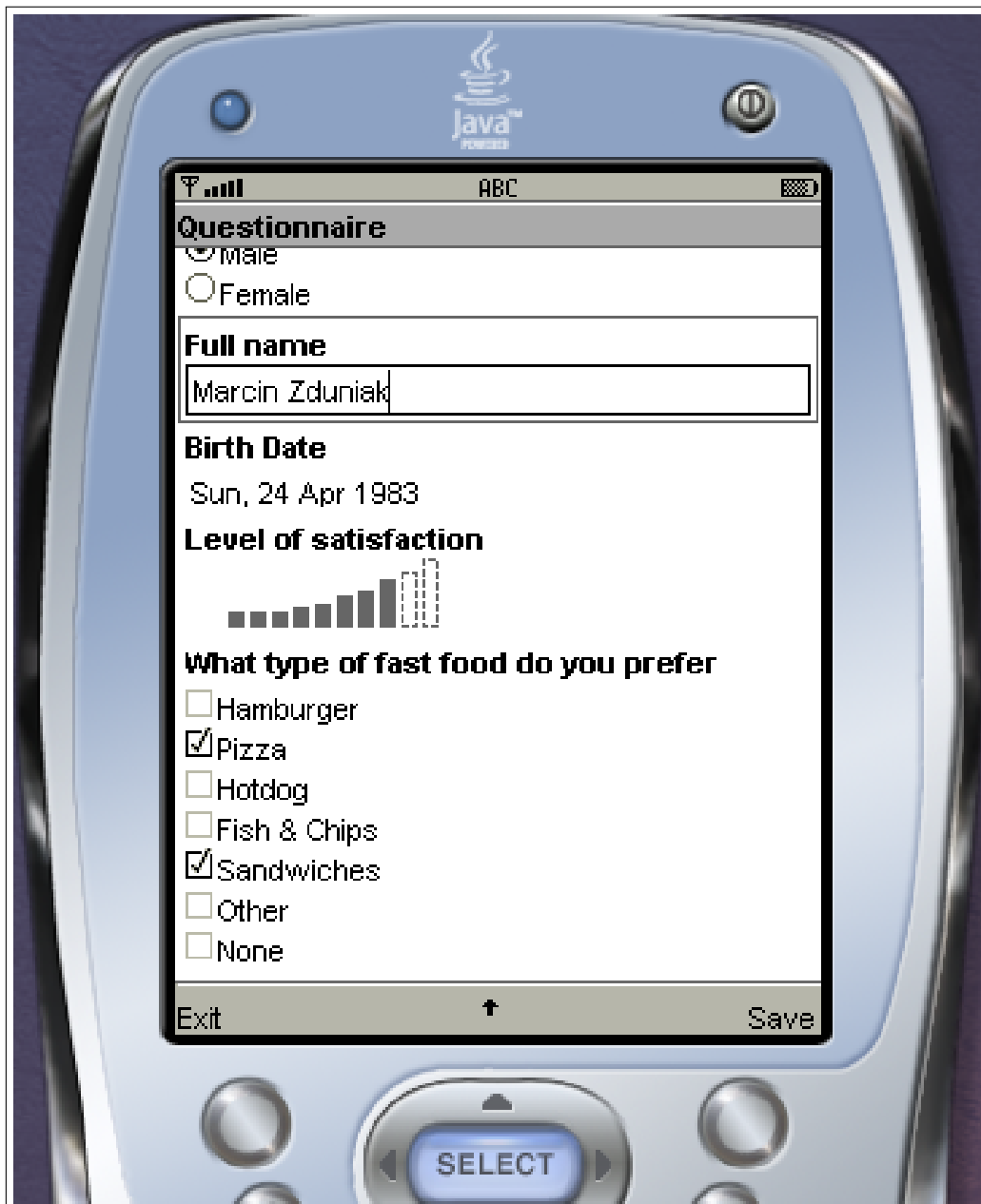


FIGURE 5.1: Emulator window with example mobile application under test.

before. We started from creating code that is able to intercept events from the mobile phone as well as correctly simulate them. Our prototype demonstrated it is possible.

The next thing that is still in front of us is verifying our framework against real mobile application, that is not only a set of realted screens created by us to only demonstrate some Frameworks capabilities, but that that have some usefull logic behind this screens. Natural choice for candidate of such application is NaviExpert, application created by group of

```
[java] Received logEntryId: 1
[java] class: org.robotme.core.log.entries.LogEntry; id: 1; level: 1; timestamp: 1166
e; msg: MIDlet set to: org.example.midlet.TestTextBoxMIDlet@d590dbc; ex:
[java] Received logEntryId: 1
[java] class: org.robotme.core.log.entries.LogEntry; id: 1; level: 1; timestamp: 1166
e; msg: Command added to displayable: javax.microedition.lcdui.Command@c1a4cfaa; ex:
[java] Received logEntryId: 1
[java] class: org.robotme.core.log.entries.LogEntry; id: 1; level: 1; timestamp: 1166
e; msg: Command added to displayable: javax.microedition.lcdui.Command@c1b37664; ex:
[java] Received logEntryId: 3
[java] class: org.robotme.core.log.entries.DisplayableChangedLogEntry; id: 3; level: 1
lse; assertion: true; msg: Displayable set to: javax.microedition.lcdui.TextBox@c828ed68;
[java] Received logEntryId: 4
[java] class: org.robotme.core.log.entries.TextBoxModificationLogEntry; id: 4; level: 1
rue; assertion: false; msg: ; ex: ; string: I like testing applications...
[java] Received logEntryId: 2
[java] class: org.robotme.core.log.entries.CommandLogEntry; id: 2; level: 127; timest
on: false; msg: Command invoked: COMMAND; ex: ; displayableTitle: Label; cmdLabel: COMMAND
[java] Received logEntryId: 2
[java] class: org.robotme.core.log.entries.CommandLogEntry; id: 2; level: 127; timest
on: false; msg: Command invoked: COMMAND; ex: ; displayableTitle: Label; cmdLabel: COMMAND
[java] Received logEntryId: 2
[java] class: org.robotme.core.log.entries.CommandLogEntry; id: 2; level: 127; timest
on: false; msg: Command invoked: COMMAND; ex: ; displayableTitle: Label; cmdLabel: COMMAND
[java] Received logEntryId: 2
[java] class: org.robotme.core.log.entries.CommandLogEntry; id: 2; level: 127; timest
on: false; msg: Command invoked: COMMAND; ex: ; displayableTitle: Label; cmdLabel: COMMAND
[java] Received logEntryId: 2
```

FIGURE 5.2: Server console – Log Server.

researcher and students from Poznan University of Technology intended to navigate simple mobile phone users from one point to the other using back-end server and GPS connected to the phone or embedded into mobile device. At this moment our prototype solution is not ready to capture all important events in NaviExpert and properly replay them. The points that must be addressed before being able to test that application are:

- more kind of assertions need to be implemented, especially those related to Record Store, sound events and various means of data tranfering over the air like: bluetooth or GPRS connections.
- tool that could visualise XML files with captured events so that editing it will be much easier and less error prone. Because in NaviExpert there are lot of events, even simple scenario is very long, difficult to understand and modify accordingly.

### 5.3 Bytecode Size and Performance Aspects

After enhancing process input JAR is substantially increased in size. The testing JAR size with sample application is 34 000 bytes. Enhancing process increases JAR in two ways: one is an outcome of injecting code into existing classes and the other is an outcome of adding RobotME Framework core classes to an existing application. Testing JAR file after enhancement process that turns input mobile application into 'capturing' version has size: 120 434 (71,7 % increase), and 'replaying' version: 121 166 (71,9 %). Summarizing only the code that was injected (without RobotME Framework core classes) the numbers was as follow:

- 'capturing' version: 35 853 (5,1 %)
- 'replaying' version: 36 585 (7,0 %)

The differences (small but noticable) leads from the fact that 'replaying' version of the application does exactly thesame what 'capturing' version of the application is doing plus some other extra tasks.

The testing framework has different performance overhead depending if it is in the capture or in replay mode. In the capture mode the overhead is mostly bound to network traffic (sending events over the wire), which can be easily neglected by using some engineering tricks (asynchronous queue of events waiting to be sent). In the replay mode the overhead is connected to the background thread reading and stimulating events. We found this overhead negligible as well.

## 5.4 Obfuscation

Both injected code and RobotME Framework API obfuscate to some extent. The testing application input JAR after obfuscation takes 26 210 bytes, enhanced 'capturing' version of the application has 93 237 (71,8 %) bytes in size, and 'replaying' version has: 93 861 (72 %). Summarizing only the code that was injected (without RobotME Framework core classes) the numbers was as follow:

- 'capturing' version: 27 473 (4,6 %)
- 'replaying' version: 28 097 (6,7 %)

## Chapter 6

# Summary and Conclusions

### 6.1 Summary

We have presented a proof of concept demonstrating that fully-fledged capturereplay testing framework is feasible in Java 2 Micro Edition environment. The prototype implementation has been well received and deployed in testing environment in a commercial software house. Our framework might be used both in fully automated testing environments exploiting mobile devices simulators as well as in real environments with real mobile devices.

Our intention was to verify whether our idea and concept is feasible – it was achieved. What has not been currently achieved is creating production ready tool, that developers and testing people could use with little knowledge of internal mechanism in RobotME framework.

### 6.2 Future Directions

Our biggest challenge at the moment is to provide some objective means to assess the value gained from using the framework. What common sense states as obvious is quite difficult to express with hard numbers. We considered a controlled user experiment where participants would be given the same application and a set of tasks to implement (refactorings and new features). Half of the group would have access to the results of integration tests, the other half would just work with the code. We hoped this could demonstrate certain gains (number of early detected bugs, for example) that eventually translate into economic value for a company. Unfortunately, this kind of experiment is quite difficult to perform and its results are always disputable (i.e., due to ranging skills between programmers), so we decided to temporarily postpone it. Other possible research and technical directions are:

- Design a flexible architecture adding support for events that are outside the scope of the J2ME specification, but are commonly used in mobile development. These events include, for example, vendor-specific APIs such as vibration or backlight by Nokia.
- Implement alternative event serialization channels – through serial cables or Bluetooth connections.

- Consider evaluation schemes for the presented solution. A real feedback from developers translates into a proof of utilitarian value of the concept – does the testing framework help? How much time/ work does it save? What is the ratio of time spent on recording/ correcting test scripts compared to running them manually? We should emphasize that these questions are just as important as they are difficult to answer in a real production environment.
- Integrate the framework with popular integrated development environments. This goal is very important because developers must be comfortable with the tool to use it and must feel the benefits it provides. Instant hands-on testing toolkit would certainly assimilate faster in the community than an obscure tool (such as Sony's).
- We also think about extending the concepts presented in this paper to other Java-based platforms for building mobile applications, such as NTT DoCoMo Java, BlackBerry RIM API or Qualcomm Brew. They may not be as popular as J2ME, but the concepts we have presented should be applicable in their case as well.

# Bibliography

- [FF] Martin Fowler and Matthew Foemmel. Continuous integration. Available on-line (2007): <http://martinfowler.com/articles/continuousIntegration.html>.
- [Jef05] Jeff Tian. *Software Quality Engineering. Testing, Quality Assurance, and Quantifiable Improvement*. John Wiley & Sons, Inc., 2005.
- [KFN99] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 1999.
- [Wil99] William E. Perry. *Effective Methods of Software Testing (second edition)*. John Wiley & Sons, Inc., 1999.

# List of Web Resources

- [A] The Unicode standard.  
<http://www.unicode.org>
- [B] J2MEUnit  
<http://j2meunit.sourceforge.net>
- [C] Mobile JUnit, Sony Ericsson  
<http://developer.sonyericsson.com/getDocument.do?docId=87520>
- [D] TestRT (Rational Test RealTime), IBM  
<http://www.ibm.com/software/awdtools/test/realtime/>
- [E] Fledge (Java Development Environment and BlackBerry Fledge simulator), Research in Motion  
<http://na.blackberry.com/eng/developers/downloads/jde.jsp>
- [F] Gatling, Motorola  
<https://opensource.motorola.com/sf/sfmain/do/viewProject/projects.gatling>
- [G] CLDCUnit, Pyx4me  
<http://pyx4me.com/snapshot/pyx4me/pyx4me-cldcunit/>





© 2006–2007 Marcin Zduniak

Poznan University of Technology  
Faculty of Computer Science and Management  
Institute of Computer Science

Typeset using  $\LaTeX$  in Computer Modern.

Bib $\TeX$ :

```
@mastersthesis{ key,  
  author = "Marcin Zduniak",  
  title = "{Automated GUI Testing of Mobile Java Applications}",  
  school = "Poznan University of Technology",  
  address = "Pozna{\n}, Poland",  
  year = "2006-2007",  
}
```