Western University
Scholarship@Western

Digitized Theses | Digitized Special Collections

2011

CREATING SMART TEST CASES FROM BRITTLE RECORDED TESTS

Santo Carino

Follow this and additional works at: https://ir.lib.uwo.ca/digitizedtheses

Recommended Citation
Carino, Santo, "CREATING SMART TEST CASES FROM BRITTLE RECORDED TESTS" (2011). *Digitized Theses*. 3295.
https://ir.lib.uwo.ca/digitizedtheses/3295

This Thesis is brought to you for free and open access by the Digitized Special Collections at Scholarship@Western. It has been accepted for inclusion in Digitized Theses by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

CREATING SMART TEST CASES FROM BRITTLE RECORDED TESTS

(Spine Title: Creating Smart Test Cases from Brittle Recorded Tests)

(Thesis Format: Monograph)

by

Santo Carino

Graduate Program in Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario
December, 2011

© Santo Carino 2011

THE UNIVERSITY OF WESTERN ONTARIO
SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisor	Examiners
James H. Andrews	Luiz Fernando Capretz
Supervisory Committee	
	Hanan Lutfiyya
	Mike Katchabaw

The thesis by
Santo Carino
entitled

CREATING SMART TEST CASES FROM BRITTLE RECORDED TESTS

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

Date	Chair of Thesis Examining Board

Abstract

Software testing is a large and important part of the software development life-cycle. There exist many methods to test software, such as writing unit tests or manually testing the software. One other such method is called **record and playback**. Record and playback allow a tester to record their interactions with a piece of software and then play back those actions against the same software at a later time. The major fault with record and playback tools is that the tests that are created are often brittle. A test is considered brittle when it no longer works when small changes are made to the software or when the test produces false-positive results. This thesis focuses on the record and playback software we designed and built for the BlackBerry smartphone. The system was designed to create smart tests from brittle, recorded tests. We discuss how we created our software and why it works. Following that, we look at the system's output to determine its accuracy. Finally, we discuss how our methods can be incorporated into general software development.

Keywords: Software Testing, Unit Testing, Record and Playback

Acknowledgments

I would like to thank Dr. Jamie Andrews for giving me the opportunity to work with him and for giving me the chance to advance my education. I have learned a great deal while working with Jamie and appreciate all of the time and effort he has put into aiding me in my journey. None of this would have been possible without him, and for this I am extremely grateful.

A big thank you to Research In Motion and the team that gave so much time to our research. Thank you to Sheldon Goulding, Tony Florio, Pradeepan Arunthavarajah, and Jakub Hertyk.

Thank you to Dr. Hanan Lutfiyya for taking the time to read my thesis proposal.

A special thank you to Sonia Dhaliwal. Her continuous encouragement over the past years has been sincerely appreciated. She has provided the motivation I greatly needed to achieve my goals. I would not be where I am today if not for her.

Finally, I would like to thank my family, Grace Job, Dave Job, Joe Carino, Scott Job, Maria Frocione, and Mario Frocione. They have provided endless support and have always encouraged me to better my education, and for this I will always be indebted to them.

I dedicate my work to my late grandfather, Mario Frocione.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Software development is a long and difficult process. It involves such activities as requirements gathering, designing, architecting, development, testing, and maintenance. We are going to focus on just one of the steps in the process, software testing. Software testing is a large and important part of the software development life-cycle. Software testing is a costly exercise and can take up as much as 60% of the development costs [10]. Software testing ensures that the software under test (SUT) works properly and contains as few bugs as possible. Without proper software testing methodologies, software that is released could potentially contain many bugs that hinder the work of the users or contain security flaws. These bugs could even be life-threatening, depending on the use of the software. Therefore, it is imperative that the SUT be tested thoroughly.

This thesis was completed as a collaborative project between The University of Western Ontario and Research in Motion (RIM), the maker of the BlackBerry, and funded in part by the Natural Sciences and Engineering Research Council in Canada.

1.2 Software Testing

There exist various ways to test software [9], such as *white-box* and *black-box* testing. White-box testing is testing in which a tester can see the source code and writes test cases to try to achieve statement coverage (try to reach every source code line), branch coverage (attempt to make IF statements execute as both true and false), and various other criteria. White-box testing is used to check the correctness of the software code. White-box testing is conducted by a tester who writes code to test the functionality of a method, class, etc. The tester's goals can vary depending on their assigned work, but generally, they are attempting to feed the code under test various values of data in order to produce specific results. If the result returned from the code under test is incorrect, the tester can assume a bug exists. Tests can be written this way to ensure that bugs do not exist; as well, tests can be used to ensure that bugs in the future do not appear. For instance, if a piece of code currently passes a test, but fails to pass the same test in the future, the tester and developer will know that some new revision in the code has produced a bug. This is called regression testing, and is an important step of the software development life-cycle.

On the other hand, black-box testing is where the tester interacts with the SUT's interface, which may be a command line or a graphical user interface (GUI). Black-box testing is used to test the functionality of the SUT. This type of testing allows the tester to interact with the program as a user would. If the tester is able to find bugs by interacting with the interface, then it is possible for a user to do the same. Black-box testing is an important part of the process as it can test parts of the software that are more difficult to test using the white-box method. For instance, it is easier to test the functionality of GUI components such as buttons and menus. As well, it allows for the testing of returned values from the program. It may be possible that the values being calculated from the software are correct, but a bug exists that causes incorrect values to be displayed. Black-box testing can also help with concepts like usability testing, which white-box testing would be of little use with.

1.2.1 Smart Tests Versus Brittle Tests

As the title of this thesis suggests, we want to create *smart tests* rather than *brittle tests*. We consider smart tests those that do not break easily when a change has been made to a GUI component or to the output of a specific method in the program. A test is considered broken when its results are false positives - that is to say, when a test returns false but is in fact true, and vice-versa. As well, a test is considered broken when it fails to run to completion because it has determined that some required information is not available when it in fact is available. For instance, if a button is moved to different locations on the screen, between different versions of the software, the tests should not break. The tests should be adaptive to the changes in the program to a certain extent. Of course, there will be instances when it is near impossible to have a test not break, but for the most part the test should not be so brittle as to fail for minor changes. Another example would be when looking at the results of a method call. The results of a method call between versions may not be the same. It is possible that both results are correct, but a brittle test may look to compare the new result to the old result and deduce an error has occurred, and in this instance the test will be considered broken. Creating smart tests cases by hand is a common practice, but creating smart tests cases automatically using a record and playback tool is not, as it is rather difficult.

1.2.2 Record and Playback

To aid in black-box testing, there exist tools to help the testers test the correctness of the SUT's functionality. One such tool is *record and playback*. Record and playback tools are used to make the testing of GUIs easier. Often times a tester is required to test a GUI by running the SUT and following a script which contains a list of components to test. For instance, a tester may be required to test the buttons, drop-down menus, etc. of the SUT in order to make sure the functionality works as intended. This testing process can be tedious and time-consuming for the tester and therefore costly to the development company. To help combat this time-consuming process, people have developed various record and playback tools. The record and playback tools work by recording the actions of the tester on the GUI and allowing the recordings to be played back against the software at later times. This allows the test to be done once by a human tester and repeated multiple times by the computer when needed.

Record and playback tools have drawbacks. The main drawback is that test recordings will tend to break if any part of the GUI changes. If on the initial recording the tester selects button A but on later versions of the software button A no longer exists, or has been moved to a different screen location, the test will most likely break. This is because many record and playback tools work by storing the location of screen components and so if the component is removed or moved the test will fail. Another drawback to record and playback tools is that it may be difficult to determine if the test has passed or failed. It is common for these types of tools to use screenshot comparisons of the initial test to the recorded tests to determine if the test passes or fails. This can lead to problems as the output in the newer versions may look different than the initial version but may still be correct.

1.2.3 Java Unit Tests

Java unit tests are written by a tester in a programming language to test a piece of software. The tests are usually written as a part of a framework such as the popular JUnit framework [8]. A unit test works by calling a specific method within the SUT and waiting for a return value. The returned value is then compared to a predetermined benchmark to check whether or not the test passes. For instance, a test could be written to test a method that multiplies two integers together. The test would be written to call the method with specific values and store the returned value. The tester would know ahead of time the expected result of the method, and therefore, if the returned value does not match the expected value, the test has failed.

To test a large software system, many hundreds or thousands of unit tests must be written to test all components of the system. There may even need to be multiple tests to verify a single component of the system. The process of writing tests is long as there are many tests to write, and furthermore, the tests themselves are prone to bugs and human error. If the tests themselves are incorrect, or the tester has miscalculated the expected resulting value, the test may fail when it should pass. As there are various ways for problems to arise when writing unit tests, automating the creation of the tests would be ideal. By automating the creation of unit tests, there is less of a chance of human error occurring, and as well, it will speed up the entire testing phase of the software life-cycle. This is the goal of this thesis.

1.2.4 Record and Playback Combined with Java Unit Tests

The aim of our system is to create smart test cases. We plan on creating smart test cases by allowing the record and playback package to transform recordings into Java unit tests. The unit tests should allow for more robustness and make the tests less brittle. The programs will be less brittle because they will be used in combination with a special type of utility software, which makes the tests more stable as it allows for another layer of abstraction. The utility software takes away most of the difficult work of finding GUI components programmatically and allows for a more simple way to create unit tests. If we are able to transform our recordings into these unit tests that take advantage of these special utilities, we will be creating smart tests from brittle tests.

1.3 Thesis Focus

The goal of this thesis is to create a record and playback tool that allows for the generation of smart test cases. To meet this goal, we must gather requirements, design a system, and implement our design via a program or multiple programs. Furthermore, we must deliver our system within the allotted time frame, receive feedback, and make changes to the system where appropriate. Finally, we will analyze the results of the system and see its implications. As this thesis was conducted in partnership with RIM, an industry giant, it can also be viewed as a study of applied software engineering.

Before we can begin writing our first line of code, we must have a full understanding of the system we are dealing with. We must have a complete picture of how the testing package interacts with the BlackBerry device and how we can implement our system to work with RIM's system. Therefore, we will discuss the requirements gathering phase and how we came to make specific design decisions. This phase of the project was time-consuming; therefore, its details should be discussed and understood before we move on to the finer details of the system.

Once we have discussed the information gathering phase of the system, we will need to discuss the design phase. Many choices had to be made during the design phase, such as the structure of recording files, and the inputs and outputs of each program, and the flow of data. In this section we will discuss, in detail, the design of the various components of the system and how each of the components works with one another. This will give us an overview of the system and allow us to understand the flow of data through it. We will get an overview of the four main programs that are part of the system and we will have a look at their required inputs.

When we have a good understanding of how the overall system works, we will look more closely at the details of the four programs that encompass our record and play-back package. We will first look at the recorder itself. The recorder is an application that runs on the BlackBerry device and can capture events when they occur. We will look at how the recorder is able to do this, as well as the log file format that it saves the events to. Next, we will discuss the two setup programs, `UtilityPropSeqGenerator` and `UtilityTraceCollector`. These two programs allow the Java unit test files to be generated. Their purpose is to create utility traces which we can match recorded tests against. Finally, we will look at `TestGenerator`, which is the program that actually generates the Java unit files based on recorded tests. For all of the above programs, we will look at their required inputs and outputs, and their algorithms.

After discussing the details of how the system works, we will look at the resulting Java unit files it is able to generate. We will analyze the results of the system and see how accurate the system is. We will look at the recording files and compare them to the resulting Java unit files. We will discuss the system's strengths and weaknesses to see where it could be improved.

Finally, we will look at the implications of our system. As the system was created for the BlackBerry device, the scope of its real world application is small. However, the ideas behind the project could be implemented in other systems in a more generic way. We will discuss how this could be done and give plausible examples.

1.4 Thesis Organization

We have discussed the introduction and basic concepts in chapter 1. In chapter 2 we will we look at related work that has been done in relation to record and playback tools. We will look at a well known tool and see how it differs from our own. In chapter 3 we will discuss the information gathering phase and how we set up our systems to work with the RIM software. In chapter 4 we will discuss the design phase and go into detail about the choices we made regarding our system. In chapter 5 we will look at the two setup programs, `UtilityPropSeqGenerator` and `UtilityTraceCollector`, and see how they work in detail. As well, will look at the `TestGenerator` program and go over its algorithm and see how it is able to generate smart test cases. In chapter 6 we will look at the results of our system and compare the recorded tests to the generated tests. We will look at the accuracy of our system and discuss the results. Finally, in chapter 7 we will look at the implications of our system and how its design can be applied to generic pieces of software.

Chapter 2

Background and Related Work

In this chapter we will give some definitions and follow that by looking at work that relates to this thesis. The concept of testing software using record and playback tools has been around for many years [12], [5], and there have been many programs created to implement the various methods [1], [2], [7]. We will first discuss a system named Abbot and its associated tool Costello [1], which contains a record and playback function. Following that, we will look at the work of other researchers who have studied record and playback tools and see how their work compares to our own.

2.1 Definitions

Here we define some terms that are required to be understood for this thesis.

A **failing test case** is a test that causes the SUT to produce the incorrect value.

A test has failed when the SUT does not return the expected value or behave as expected.

A **passing test case** is one that causes the SUT to return the correct values. The SUT returns a value that is expected or behaves as expected.

A **recording** is a a log of events that occurs as the tester interacts with the SUT's interface. The log can be stored in various ways, such as XML, and plain text, and can contain any information that the developer of the recorder deems pertinent.

A **playback** is the act of a program reading in the log of a recording and replaying the events that occurred against the SUT.

2.2 A Real World Package

In this section we will be looking at a real world package named Abbot and Costello. The package derives its name from the creator's description, "A Better 'Bot". As we are creating our own record and playback testing package, it is a good idea to see how others have attempted to solve this problem.

The goal of Abbot is to give the testers a framework in which to test the GUI components of the program. Abbot works in conjunction with JUnit to create unit tests. It is able to allow GUI testing by giving the tester references to the various GUI components being used in the SUT. The tester can write unit tests to retrieve a GUI component and perform some action on it, such as clicking a button or selecting a menu item. Abbot works by implementing a "robot class". The robot class is able to control GUI components and mimic user events such as mouse clicks and keyboard events. As well, Abbot takes advantage of the reflection feature of Java so that it may find GUI components programmatically.

Abbot also allows for a special type of scripting that can be edited and run to test a GUI. The script editor, Costello, reads in the script and runs it against the SUT. The scripts are stored as XML files. Storing script files as XML files is a common pratice, as well as creating custom scripting languages [3], [12]. The point of the script files is to allow for a higher level of usability. The scripts are considered "higher level" as they do not require the tediousness of writing an entire program. As well, scripts allow for more accessibility for testers as they may not be strong programmers, but they may be able to design simple script files.

The package also contains various types of component recorders. The recorders allow the testers to capture mouse and keyboard events that occur while running the SUT. The recorders allow for easier script-editing as they can fill in most of the work for the testers. The scripts can then be read back by Costello and run against the SUT.

In figure 2.1 we can see an example unit test written using the Abbot system. The test works by getting access to a GUI component, the left arrow button, and proceeding to press and hold it down for 5 seconds. The test asserts true if the number of mouse events received is greater than 1. This is a fairly simple test with little complexity, but as one could imagine, if we wanted to test multiple GUI components at once the complexity would increase. The more complex these tests are, the greater the chance is for bugs to occur within the unit tests themselves. Even with the complexity in mind, the Abbot tool is powerful and gives freedom to the testers as they can automatically run these tests once they have been written. The goal of this thesis is

Figure 2.1 Example Abbot test from the Abbot and Costello website. The example shows the pressing of a button for 5 seconds.

```
private int count = 0;
public void testRepeatedFire() {
    ArrowButton arrow = new ArrowButton(ArrowButton.LEFT);
    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
            ++count;
        }
    };
    arrow.addActionListener(al);
    showFrame(arrow);

    Dimension size = arrow.getSize();
    // Hold the button down for 5 seconds
    tester.mousePress(arrow);
    tester.actionDelay(5000);
    tester.mouseRelease();
    assertTrue("Didn't get any repeated events", count > 1);
}
```

to have unit tests such as in our example be automatically written, therefore cutting down on potential bugs and the need to write tests by hand. The style of unit tests created using Abbot was a big influence on this thesis.

In figure 2.2 we can see an example Abbot XML script file. The script file describes a test to be run. We can see the that there are various GUI components described in the component class lines, and there are actions described in the action lines. Mixed in with the action lines are assert lines, which check that the GUI components are correct. These are the type of script files that will be generated from the recording tools provided by Abbot and read in and run by Costello. Trying to edit one of these XML files is challenging as the tester would have to know the different types of XML tags and attributes available and what their values should be. Furthermore, a downside to storing tests as script files is that the Abbot and Costello software is

Figure 2.2 Example Abbot test script from the Abbot and Costello website.

```
<?xml version="1.0" encoding="UTF-8"?>
<AWTTestScript>
  <component class="java.awt.Button" id="?" index="3" tag="?"
      window="Applet Viewer 0" />
  <component class="sun.applet.AppletViewer" hOrder="0"
      id="Applet Viewer 0" tag="Applet Viewer: example.SimpleApplet" />
  <component class="java.awt.Dialog" id="Dialog" title="Dialog" />
  <component class="java.awt.Button" id="High Button" tag="High"
      window="Applet Viewer 0" />
  ...
  <appletviewer archive="lib/example.jar" code="example.SimpleApplet"
      height="250" width="250" />
  <action args="textField" class="java.awt.TextComponent"
      method="actionFocus" />
  <action args="textField,some text" class="abbot.tester.ComponentTester"
      method="actionKeyString" />
  ...
  <action args="?" method="actionClick" />
  <wait args="Dialog" class="abbot.tester.ComponentTester"
      method="assertFrameShowing" />
  <assert component="This is a dialog" method="getText"
      value="This is a dialog" />
  <action args="Dialog" class="java.awt.Dialog" method="actionClose" />
  <wait args="Dialog" class="abbot.tester.ComponentTester"
      invert="true" method="assertFrameShowing" />
  <action args="5000" method="actionDelay" />
  <terminate />
</AWTTestScript>
```

always required to run the tests. The project described in this thesis attempts to generate test cases from recordings that can be run independently of the software that created it. The log files generated from our project are simple and require no human editing; a tester is able to record a test, convert it to a unit test, and run it without ever having to program a test or edit a script.

The Abbot and Costello package was a big influence on the work done in this thesis.

We are creating a similar tool that we hope will be able to create smarter test cases, while requiring less technical knowledge for our users. The users of Abbot and Costello must either know how to create unit tests in Java or know how to edit XML script files. The users of our system should only have to know about recording tests and converting them to Java unit tests automatically via a provided program. The tests created from our package should also be less brittle as they have a layer of abstraction to take away complexities such as locating GUI components; if the component exists, we simply interact with it.

2.3 Related Work

In this section we will be looking at other researchers who have done work with record and playback. It is important for us to see how others have approached the problem of capturing interactions between a user and a system as we may be able to borrow ideas to make our own systems better. As well, we will be able to see how other researchers were able to store the recordings and play them back against the SUT. We will be looking at papers by Orso and Kennedy [13], Elbaum et al. [4], Saff et al. [14], and Memon et al [11]. Each of these papers approaches a very specific problem, so the details are not always directly relatable to our own work, but many of the ideas are applicable.

2.3.1 Capturing User-Subsystem Interaction

Orso and Kennedy [13] created a technique to allow for capture and replay of a user interacting with a system or subsystem. Since there is a large amount of data flowing through the system, some of it even being confidential, Orso and Kennedy decided to only capture a small subset of the information between the user and whichever subsystem the user was interacting with. This would allow them to generate unit tests and analyze the system offline. They were working with a large system and so capturing all relevant information was challenging; they would have to know the states of various databases and users. Therefore, they decided to capture only a select portion of the interaction based on the user's preferences. Orso and Kennedy were able to perform capture by instrumenting the code; that is to say, they inserted probes to log the events of the system. They created a technique that would create proxy methods that would stand in between the calling method and the called method. It works by adding some code before and after the method is called. The proxy method logs the parameter values sent to the method being called and also logs the value returned from the called method. To replay the recorded event, the system determines which subsystem is being interacted with and generates the objects based on the recording. It uses stubs to mimic the behavior of external systems that the subsystem needs to interact with.

Joshi and Orso [7] did further work in the area of capture and playback of tests. They created a tool called SCARPE that allows for the capture and replay of subsystems. The paper describes the use of instrumentation to capture the events, just as in the Orso and Kennedy paper. The recorded tests have their events logged to a text file; as well, the output of the system is also stored. To replay the recorded events, SCARPE builds a scaffolding system around the subsystem. The scaffolding mimics the behaviour of the external system that the subsystem interacts with. The scaffolding then supplies the values from the log and checks the return values from the subsystem. If the return values do not match the log file, the system queries the user on how to proceed. SCARPE is able to record and replay the events well, but at the cost of some overheard. For some cases, the overhead is too large to make the test creation feasible, but Joshi and Orso claimed to be working on making the system more efficient.

Orso and Kennedy's work was very influential at the beginning of our own project as we had initially believed that we would have to instrument code as they had done. Before we had a complete understanding of the RIM software we would be working with, we explored the idea of instrumenting the system code and inserting probes so that we could view the interactions between various subsystems. After we had met with the RIM team it was discovered that there was a much easier way for us to record the events of the system, by use of listeners. Listeners are are type of observer pattern that allow programmers to probe software and log the events occurring within. It turns out that RIM's software has a listener class in place in which we could essentially plug our own code into to begin recording events.

2.3.2 Capturing Unit Tests from System Tests

Elbaum et al. [4] recognize the efficiency of unit tests and the importance of system tests and therefore want to merge the two into what they call differential unit tests (DUT). System tests involve testing the functionality of the system, but the tests can be slow to complete (days or weeks at a time) and that is why they want to create these hybrid DUTs.

Their general method to create DUTs is to carve the system components, during a system test's execution, that influence the target unit's behavior. The carving can then be re-assembled so that the target unit can be tested as it was by the system test. These carved tests would be closer to unit tests and therefore retain some of the advantages of unit tests. The carving of unit tests from the system is essentially a recording, and replaying the tests against the target unit is analogous to playing back a recording. The method works by recording pre and post states of the system. Before a unit is executed, the pre state is recorded and after the unit executes, the post state is recorded. These two states can then be used as the baseline tests for future versions. New versions of the software also have their pre and post states recorded and then compared to the original pre and post states. If the post states are different, then it is known that the unit is not acting as it should.

The work done by Elbaum varies greatly from our own. Our system records and replays unit tests in a much different way then the system above, though originally we did consider using a state-based method for recording and replaying tests. Our system, as we will see in the coming chapters, only deals with recording unit tests and does not deal with system tests. Since we are not dealing with system tests, we have no need to worry about the complexities of carving tests. During our system's development, we considered using states to help generate unit tests. In a similar way that Elbaum's system uses pre and post states to determine unit correction, we were going to use pre and post states to determine which method to call. Once we discovered the limitations of the RIM software we were dealing with, we had to discard the idea of using states. However, the idea of states could potentially be used in our methodology if it were to be implemented with generic software. As we had no control over how the RIM software worked, we could not modify it to work in a state-based manner.

2.3.3 Automatic Test Factoring

Saff et al. [14] conducted research on automatic test factoring for Java. Test factoring is the method of creating unit tests from system tests. The unit tests only test a subset of the functionality that the system tests test. However, the unit tests can be run more quickly and help isolate bugs.

The method uses the idea of mock objects to factor tests. Mock objects are simulations of another object and mimic the same behavior of the original object in a controlled way. If a component of the system, **T**, interacts with an environment object, **E**, a mock object can be used in place of **E** when running the test. By using mock objects, the running time of tests can be reduced. The mock object checks the input and output values from the test and compares them to the initial system test. While capturing the test, a mock object is wrapped via instrumentation around the real object and a transcript of the actions is created. When the tests are replayed, the mock objects read the transcripts and check that the test's inputs match against the transcript. This means that the actual system objects do not have to be run in order for the tests to be conducted. Saff et al. found that test factoring using mock objects can cut down on running times by up to an order of magnitude.

The research done in the above paper does not directly correspond to the work we did in our own project, aside from the idea of record and playback. The research done by Saff highlights a possible method that we may have used if it was decided that we needed to instrument the code at all. The use of mock objects to record and play back tests is intelligent; however, the goal of their research was to create unit tests from system tests and the goal of our research was to simply create unit tests from user actions. If we had decided to use instrumentation to record events, it is more likely that we would have used probes in the way that Orso and Kennedy did in their paper.

2.3.4 GUI Testing Using Automated Planning

The research done by Memon et al. [11] discusses their automated test generation program, PATHS, that generates test cases from a hierarchical stand point using the AI technique of planning. The input to the system is a set of initial states, goal states, and operators. The operators tell the system how to navigate the GUI. The planning technique utilizes these operators to create a path from the initial state to the goal state. It is possible for multiple paths to exist from one initial state to one goal state. The system uses a hierarchical methodology to encapsulate the GUI into more abstract concepts. This encapsulation allows for tests to be generated at a much quicker speed than if only one layer of GUI operators were to be used. The user of the system must define the GUI operators and state the preconditions required for the operator to be invoked. For instance, in order to close a file, you must first open a file. The system uses a mapping mechanism to break down the higher, more abstract operators into their lower-level, simple operators.

The work done by Memon et al. has some relation to the work done in this thesis. For instance, we also use a mapping mechanism to go from low-level events to higher, more abstract method calls. This allows us to encapsulate the work of finding GUI components and interacting with them. Another concept that relates to our own thesis is the idea of ordering. Some of the operators in the above paper require preconditions to be true in order for the operator to be able to be invoked. In our system, we require that the methods be placed in order as well, as some of them require the system to be in a specific state, and this can only be accomplished if some other method is called before.

Chapter 3

Information Gathering

In this chapter we will look at the information we gathered from our meetings with RIM, such as how their system works, the testing package they use, and the requirements for the record and playback tool we created. We needed to have a picture of how everything worked so that we could properly design our software. Furthermore, we needed to understand RIM's requirements. We also needed to understand how their system works and how their test package works so that we could integrate our system with theirs.

3.1 RIM System Information

Since we worked with RIM, we designed software to work in conjunction with the BlackBerry smartphone. The BlackBerry uses a Java-based operating system and all of the applications written for the BlackBerry are also written in Java. It follows that our own software is written in Java as well. Technically, we only had to write the recorder in Java; we could have potentially written the test generation software in some other language, but as it is my most proficient language and the language used by RIM, we stuck to Java. The BlackBerry contains many different layers of software, as one would expect, but we are only interested in the operating system layer as that is where the events are sent out and available for recording. The testing software used by RIM is created in-house and named PuppetMaster. PuppetMaster has the ability to inject events into the Java Virtual Machine running on the BlackBerry; it also has the ability to listen to the events being produced. We will look more closely at PuppetMaster in section 3.2.

When developers create software for the BlackBerry, they need to test it on a BlackBerry to see if it actually works. Since not all developers own a BlackBerry or want to use their expensive phones as testing devices, RIM has created a virtual device, called a simulator, on which developers can test their programs. The simulators are Windows applications that look and act like a real BlackBerry. There exist simulators for all of the different types of phones that RIM releases. This allows the developers to test their applications on multiple devices to ensure compatibility. RIM also uses these simulators to test out their own software. Instead of providing every tester with their own testing device, the testers can simply install the necessary simulator and test that the software works correctly. Of course, simulators do not always act the same as real phones and so RIM uses a mix of both real devices and simulators when testing their software. For our own purposes in this thesis, we tested our software against a simulator. Since we developed the software in our lab rather than at RIM, we had to set up the environment in our lab so that we could create and test our software properly.

3.1.1 System Setup

Setting up the RIM environment on our own systems proved to be a greater challenge than originally thought. We had to do the following things.

- Install a development environment.
- Install a BlackBerry simulator.
- Install the BlackBerry USB drivers.
- Install the PuppetMaster Software.
- Connect a test program to the simulator and run events.

We first had to decide on the development environment we would be using to create our software. The choices were narrowed down to two possibilities: Eclipse with the BlackBerry plug-in and the BlackBerry Integrated Development Environment (IDE). The BlackBerry IDE was created by RIM to allow developers to easily write applications for the BlackBerry as it came installed with a device simulator. We decided to try both Eclipse and the BlackBerry IDE to determine which one best fit our needs. It turned out that the BlackBerry IDE was a little primitive for our needs and harder to work with, so we decided to use Eclipse. The BlackBerry plug-in for Eclipse comes packaged with a simulator, but the simulator it came with is not the device we wanted to test on. Therefore, we downloaded the latest version of the BlackBerry 9700 simulator and used that instead. Installing the simulator and getting it to work was fairly painless.

Next we had to install the BlackBerry USB drivers. The purpose of the drivers is to allow a device to connect to a computer so that information can be passed back and forth between the two. For instance, it allows for a user to back up their data or install new applications on their phone. The simulator we would be using simulates a connection via the USB port and tricks the desktop software into thinking an actual device is attached to the computer. This allows us to treat the simulator like a real device and connect to it as such. We were able to easily install the USB drivers from the BlackBerry website.

Following that, we needed to install the PuppetMaster software. The PuppetMaster software is what allows us to create a link between our program and the simulator. The software comes packaged with Java classes that can create connections to a simulator or device connected to the USB port. One problem with installing PuppetMaster was that there were very specific version requirements. The software could only work with one specific version of the BlackBerry operating system. It took us some time to communicate with RIM and get matching simulators and PuppetMaster software. Once we were finally able to install PuppetMaster on our desktops and on the simulators, we had to create a test program to verify the communications link between a program and the device.

Getting a test program to work properly is where all of our problems occurred during the process. We developed a simple program to make simple method calls such as opening a menu or pressing a key on the device. The program we created was rarely able to connect to the simulator we had running on the desktop. The software kept reporting errors stating that there was no device connected to the computer. When the software was able to connect to the simulator, it would not always work and would fail halfway through completion. After much investigation and trial and error, it was determined that the PuppetMaster software could not work properly on a 64 bit operating system. As we were running the software on Windows Vista 64, this posed a problem. To remedy this situation, we installed VirtualBox that was running a copy of Windows XP 32 bit. We then had to install all of the software again. When we ran our test program on the new environment, everything went as planned and the test program was able to connect to the simulator and run methods as we had hoped. Now that we had our environment set up, we could begin working on designing the system we would be creating. We will talk about the design phase in the next chapter.

3.2 PuppetMaster and Utilities

PuppetMaster is a software package developed by RIM and is used to create unit tests. PuppetMaster is similar to the Jemmy testing library [6], which allows testers to interact with AWT/Swing GUI components programmatically. PuppetMaster works by giving unit test programmers an easy way to manipulate the applications running on the BlackBerry. It provides an interface to programmers that allow the tests to access GUI components without having to worry about the low level details, such as finding a specific button or clicking the trackball. PuppetMaster provides a set of static utility methods that can be called from a unit test. The software consists of a set of utility classes, each of which contains a set of utility methods. Some of the classes provided are application-specific; for example, some of the utilities can only be used to interact with the `Browser` application or the `Email` application. There are also other utilities that allow general GUI component access. These utilities allow a test programmer to access any button, menu, etc. that is on the interface. Some of the many utility classes provided are `BrowserUtilities`, `EmailUtilities`, ApplicationUtilities, and ButtonUtilities. All of these classes contain methods that can be called, such as BrowserUtilities.openBrowser(), which opens the Browser application, or MenuUtilities.selectMenuItem(itemName), which selects a menu item. By using these utilities, a test programmer can quickly create unit tests without worrying about how to find the GUI components using reflection. Instead, the programmer can create a unit test that simply contains a list of method calls that flow in a logical order. An example test is shown in figure 3.1.

Figure 3.1 Example test case using utility calls

```
/*Test that opens the browser, goes to example.com and then
 *closes the browser
 */

\\Code to set up method here
BrowserUtilities.openBrowser();
BrowserUtilities.goToURL("www.example.com");
BrowserUtilities.close();
\\Code to end method here
```

PuppetMaster contains two major components, the desktop side and the device side. The device side of the software runs on an actual phone or on a simulator. The device side of the software allows for communication between the tests and the device. The desktop software is where one would create and compile unit tests. The desktop software has classes that can be called to create a connection to the device and send and receive information. A properly set up environment contains a simulator or device running the PuppetMaster software and a development environment set up to create and compile PuppetMaster test cases. The tests can then be run from the desktop against the device or simulator that is connected to the computer. The tests are determined to have passed or failed by checking to see if an exception has been thrown from the unit under test. If an exception is thrown a test is said to have failed.

Now that we have a good understanding of how PuppetMaster works, we need to understand how our own project ties in to it. Our project works in conjunction with PuppetMaster to create unit tests that call PuppetMaster utilities. Our software allows a tester to record tests from a device or simulator and then have that recorded test be transformed into a unit test that contains utility calls. We will see the major details of how this is done in the coming chapters.

3.3 System Requirements

Before we began the design and development of our system, we first had to figure out the requirements of the system. As we worked with RIM and developed the software for their needs, we had to figure out what exactly they were looking for in our testing package. The following is a list of functional requirements [1] for the system.

1. The system must implement recording via listeners.
2. The system must plug its listener class into the PuppetMaster listener hub.
3. The system must take in a recorded test and produce a Java unit test.
4. The system must produce PuppetMaster type unit tests.
5. The system must use a BlackBerry application to record events.

The following is a list of non-functional requirements for the system.

[1]By convention, functional requirements use the word "must" in their description, whereas non-functional requirements use the word "shall".

1. The system shall work with the current BlackBerry architecture.

2. The system shall be documented; both in the source and externally.

3. The system shall be written in Java.

4. The system shall be completed within a 6 month time frame.

As we can see, there were not too many constraints on how the system should have been developed. Since we were trying to solve the problem for RIM, we had the freedom to create any method we desired to transform a recorded test into a unit test. As we developed our system, we created different formats for storing data and different methods for transforming recordings, but since these were not part of the requirements from RIM, we do not have them listed above. Now that we have a good understanding of the system requirements and what our end goals were, we can look at the design of our system.

Chapter 4

Design

In this chapter we will look at the design of our system and how we came to decide on its current form. Before we could begin programming the system, we needed to figure out how we could potentially transform a recorded log file into a unit test. To transform a recording into a test, we needed to understand the flow of information through our system. We will see the overall design we settled on and discuss why we think it is a good for our needs.

4.1 Recording Design

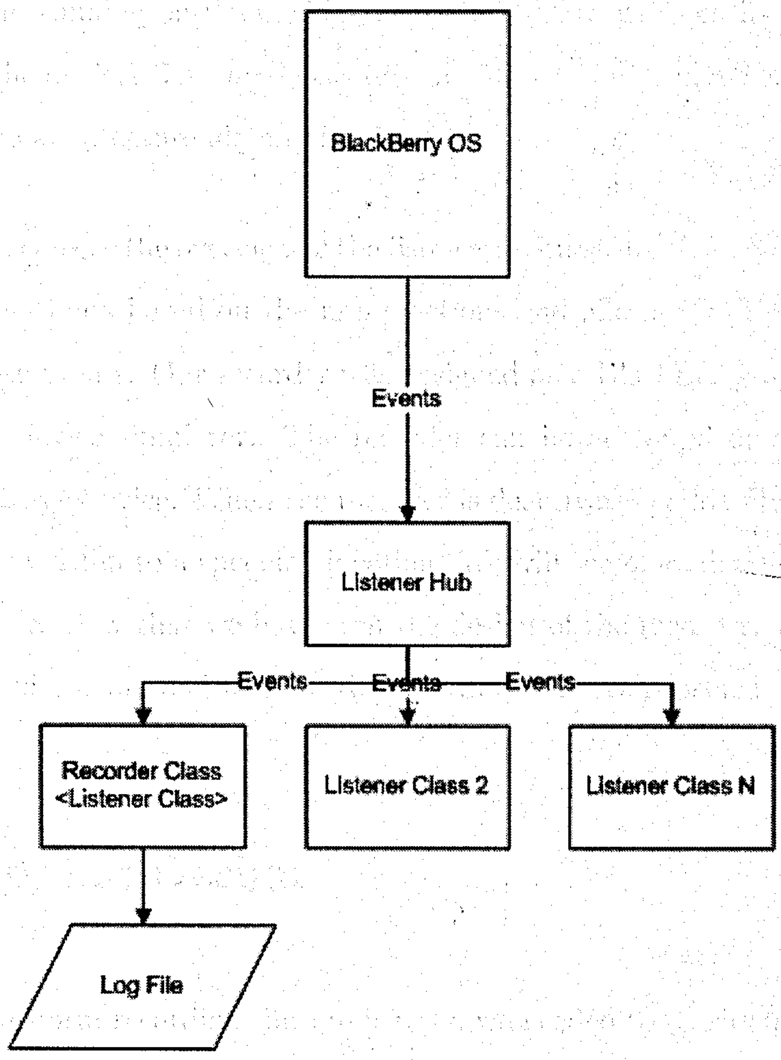
In the following two subsections we will look at our initial thoughts on how to design a recorder and how we refined them. We will see the evolution of our design as we gained more information about the RIM system.

4.1.1 Initial Thoughts on Recording

When we initially began to design our system, we decided that instrumentation would probably be required. As we had previous experience using instrumentation on Java classes, we knew that our idea was possible to a certain extent. As we had not learned much about the RIM system at this time, this would have been the best method for recording tests on a live system. Our goal was to instrument the Java class files and insert probes into key locations. These locations would be the beginning of methods, at the end of methods before the return value (if there was one), and before and after method calls. The idea was to record the values passed into methods and the values passed back by methods. By knowing which values were being input to the system and which values were being returned, we would know how to replay the test because we would have the values stored; we would also know if a test passed or failed because we would know what to expect as return values.

The problem with instrumentation is that we would need access to the underlying software running on the BlackBerry platform. It is easy enough to instrument desktop software as one tends to have direct access to the Java class files. However, the software running on the BlackBerry that we would need to instrument is much different than that which is on a desktop computer. The BlackBerry is well known for its security, so it would have been difficult to insert probes into running applications on the BlackBerry and record events. As we had little knowledge of the system so far, this still seemed liked the best option, even if it seemed extremely difficult.

Figure 4.1 Listener Architecture



4.1.2 Refinements

After meeting with RIM and learning about how their system works and how we could go about creating a record and playback tool, we had to disregard our previous thoughts on a recording method. We learned that PuppetMaster comes with a listener class and a listener hub class that allow for a program to catch and look at the events produced by the system. By extending the provided listener class, we could create our own custom listener in the form of a BlackBerry application and plug it into the listener hub running on the device. This would allow us to easily log the events occurring on the device. This made the design of our recorder much easier and more in line with the architecture already in place.

In figure 4.1 we can see the overview of the listener architecture. The operating system simply outputs events based on the user's actions and allows for other applications to observe these events. Our recorder was designed as a BlackBerry application and loaded onto a device simulator. The recorder can be activated or deactivated by pressing the `ALT` key twice. When the recorder is deactivated, a log file containing all of the events is written to a specified location. We will see more details about this in the next chapter. Now that we have seen the design of the recorder, we will look at how we designed our method for how the recordings are transformed into unit tests.

4.2 Mapping Design

In order to transform recordings into unit tests, we needed to teach our system what to look for. Since the end goal of our system is to generate unit tests that contain utility calls to PuppetMaster (`BrowserUtilities`, `MenuUtilities`, etc.), we needed to show our system what utility calls look like. The log files that are output from the recorder contain two types of events: `keypress` and `display`. `Keypress` events are the events generated whenever a user presses a key on the phone. A `display` event is generated whenever the screen is updated with new information. Since these are fairly low-level events, it would be hard for a program to determine if the user is sending an email or typing in the calculator by just observing the events. In order to determine exactly what the user is doing on the phone based on the events, we need a baseline for comparison. The baseline we created is a type of mapping mechanism. Our mapping mechanism works by mapping new recordings against previously recorded utility traces. The utility traces we use are recordings of the utility methods within PuppetMaster, along with some metadata. Utility traces are created as follows.

1. An administrator selects a list of utility methods to record for the baseline.
2. The utility methods are placed in a logical order (e.g. you must open a browser before you can close it).
3. The methods are run against the simulator or a device.
4. Recordings are made for each method being called.
5. The recordings are saved and placed in a special location to be used later for mapping.

Now that the system contains a baseline of utility traces, the testers can begin recording tests and comparing them against the utility traces. By creating these utility traces, we are telling our system what to look for in the recordings. The system will now be able to tell if the user is sending an email or working within some other application. Since the system now knows what the user is doing, the system can generate the proper method calls in the Java unit files it will generate. The basic steps of how the utility traces are used are as follows.

1. A tester records a test case.
2. The tester runs the test case through the TestGenerator program.

3. The TestGenerator program compares the contents of the recording to all of the utility traces.

4. For each block of events in the recording that matches a utility trace, a method call is generated.

5. When the mapping is complete, all of the method calls generated are output to a Java unit file.

Figure 4.2 Recording and Utility Trace Matching

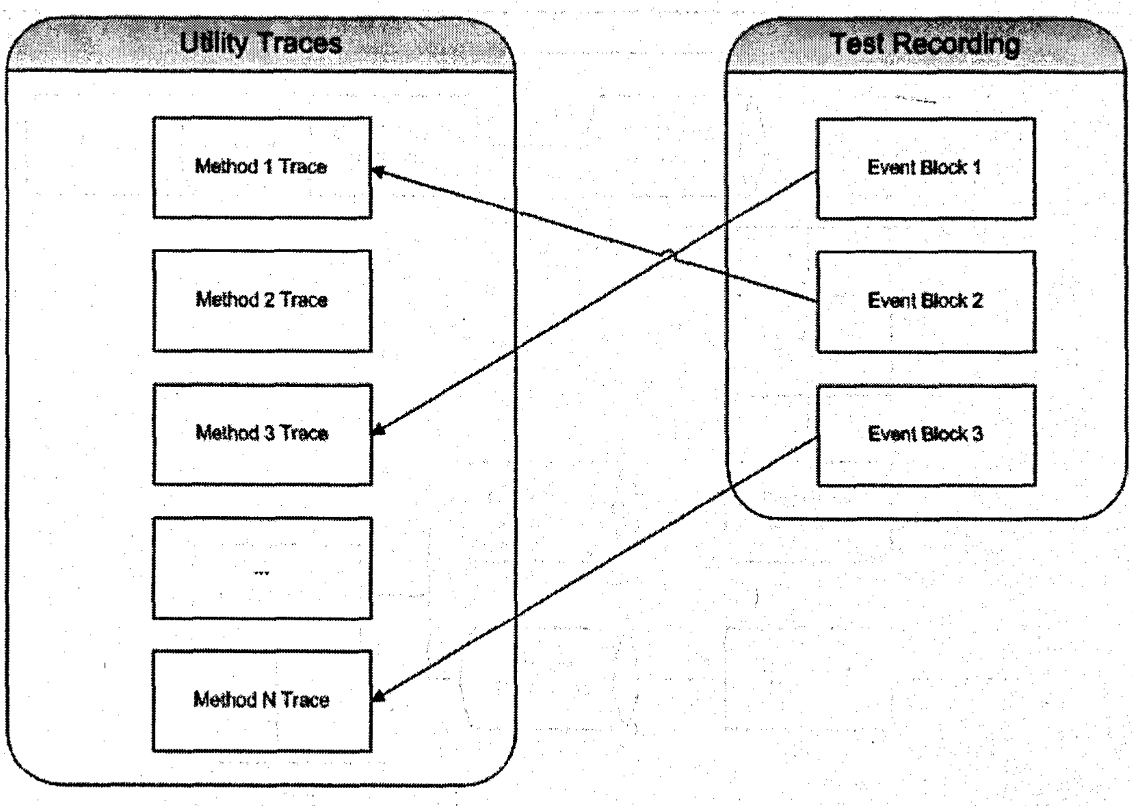


Figure 4.2 is a diagram of how the overall system works with the utility traces. We can see event blocks from the recordings being matched to utility traces. For every matched utility trace, a method call is generated that corresponds to that method. For example, if Trace 2 corresponds to the method `BrowserUtilities.openBrowser()`,

a call to that method is generated in the Java unit file. We will look at how `TestGenerator` actually matches the recordings to the utility traces in the next chapter as that is the most complex part. Furthermore, we will look at more details of how the utility traces are generated exactly by the two programs `UtilityPropSeqGenerator` and `UtilityTraceCollector`.

4.3 System Overview

Figure 4.3 System Overview

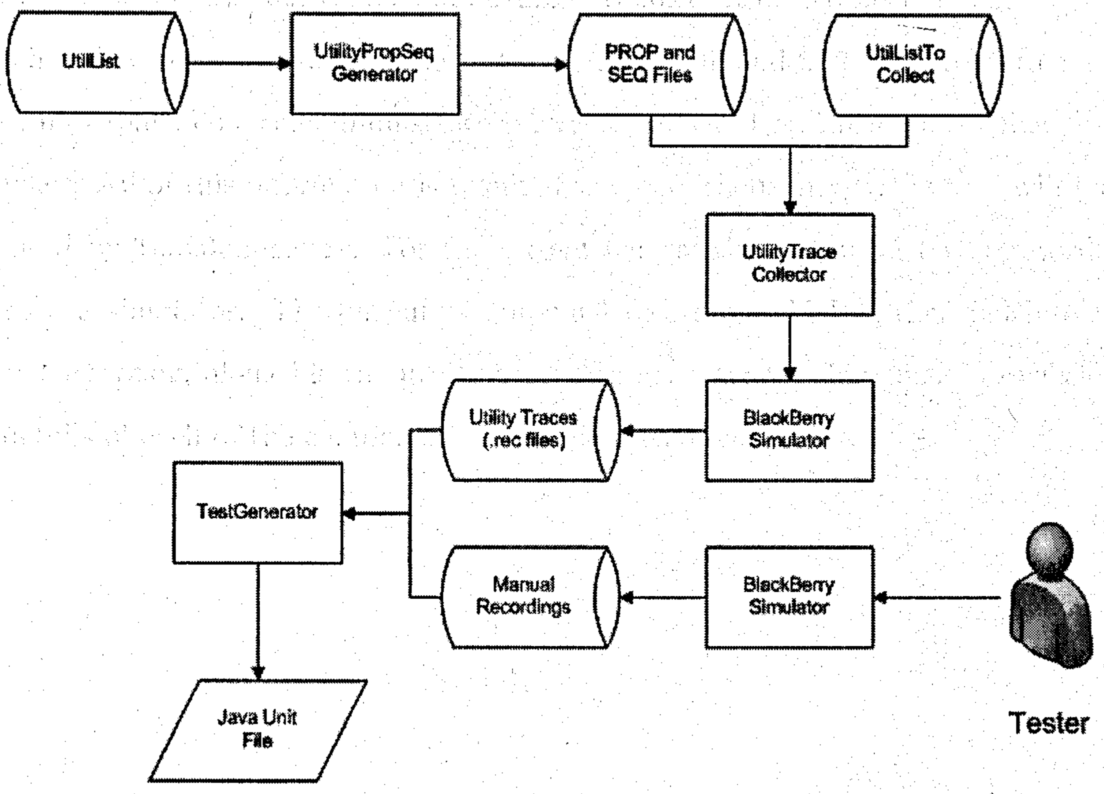


Figure 4.3 shows our complete system design. We can see the three main programs: `UtilityPropSeqGenerator`, `UtilityTraceCollector`, and `TestGenerator`. The two former programs are what create the utility traces, while the latter program

generates the Java unit tests. The recording software is located on the BlackBerry simulator. The cylindrical objects are the inputs and outputs of their respective programs and the Java unit file is our final generated test case.

By looking at the overview we can see the flow of data through the system. The system works in two parts, the setup part and the testing part. The setup part consists of the two programs, `UtilityPropSeqGenerator` and `UtilityTraceCollector`, that ultimately output the utility traces, and the second part consists of a tester recording a test and running the `TestGenerator` program.

For the first part, the `UtilList` and `UtilListToCollect` data objects tell our system which utility classes we would like to trace. The `PROP` and `SEQ` files are our custom data files that allow the administrator or tester to select which methods they want to trace. All of this information is required to create the utility traces that will later be used by `TestGenerator`. For the second part of the system, a tester records a test on a simulator. The output is a manual recording which is then fed into the `TestGenerator`, alongside the utility traces, to generate the Java file. We will look at details of each of the components in the next chapter.

Chapter 5

The System

In this chapter we will be looking at the programs that make up our system. First, we will be looking at the recorder and how it saves the system events. Following that, we will look at how we create the utility traces. The utility traces are created in two steps using the programs `UtilityPropSeqGenerator` and `UtilityTraceCollector`. We will see how these two programs work, what their inputs and outputs are, and how they work together to create the utility traces. Finally, we will look at how we generate the Java unit files from the `TestGenerator` program.

5.1 The Recorder

The recorder was created as a BlackBerry application. It is loaded onto a device or simulator and activated/deactivated by pressing the `ALT` key twice. The application is set to automatically run when the simulator is started so that the tester can begin recording test cases immediately. The recorder implements a listener class that is part of the PuppetMaster system. The listener class contains various methods to allow for the capture of events. The listener class attaches to a listener hub that outputs the system events that can then be captured by any attached listener. All of the events captured by the recorder are output to a log file. The methods that we are interested in implementing are `onUserKey` and `onUpdatedisplay`. The `onUserKey` method captures all of the `keypress` events. `Keypress` events are generated whenever a user presses a key on the device. The `onUpdateDisplay` method captures all of the `display` events. Whenever the display is updated with new information, `onUpdateDisplay` is notified. The method headers are as follows.

```
public void onUserKey(UiEngineInstance uie, Screen screen, int event,
int key, int keycode, int time)

public void onUpdateDisplay(UiEngineInstance uie)
```

The important parameters for `onUserKey` are the `screen`, `key`, and `keycode`. We are interested in these three parameters as they tell us which key is being pressed and on which screen the key is being pressed. We need to be able to distinguish which application is currently running, if any. We need to know which application is running because we need to store that information in the log file so that the `TestGenerator` program will be able to match the recordings to the utility traces more accurately.

All of the events passed in to the `onUserKey` method are saved in the log file as `keypress` events. `Keypress` events are stored in a log file in the following format.

```
keypress:key value:application class name:focused field type:
```

```
focused field's text
```

The keypress line is colon-separated into five segments. The first segment is simply the word "keypress"; this tells us that this is a keypress line and not a display line. The second segment stores the key value of the key that was pressed. For example, the Enter key is represented by the value 27. The third segment is the class name of the currently running application; this tells us which program is currently running. The fourth segment is the current field type in focus such as a text box or a radio button. The fifth and final segment contains the text contained within the currently focused field, such as the text contained within a text box. With all of this data, we are able to know which key has been pressed, which application is running, and in which GUI component the key value was inserted into.

The display lines are stored in the log file as follows.

```
display:application class name:focused field's text
```

The display line, like the keypress line, is colon-separated. The display line is split into three segments. The first segment simply contains the word "display". This tells us that the line is a display line and not a keypress line. The second segment is the application class name, which tells us the class name of the currently running application. Finally, the third segment tells us that text contained in the currently focused field. There are two special cases regarding naming conventions for our system. The first case is when the user is selecting a menu item and the second case is when the user is pressing an on-screen button. These actions required multiple lines of logging and so we want to make sure the lines are grouped together. Therefore, for all actions dealing with the menu, the `application class name` segment of the `display` line is named "menu". Similarily, for all actions dealing with buttons, the `application class name` segment is named "button". These special cases end up being pivotal in the generation of test cases as they allow us to properly determine which menu item or which button is being selected.

Figure 5.1 Example recording for menu selection

```
display:net.rim.device.apps.internal.ribbon.RibbonLauncherApp:none
keypress:268566528:net.rim.device.apps.internal.ribbon.RibbonLauncherApp:
 net.rim.device.apps.internal.ribbon.launcher.ApplicationAreaGridField:
  Browser
display:menu:Open Tray
display:net.rim.device.apps.internal.ribbon.RibbonLauncherApp:none
```

In figure 5.1 we can see an example log file. The log file contains both the `keypress` and `display` lines. The figure depicts a user pressing the `menu` key and selecting the `Open Tray` menu item. Now that we know how log files are created, we need to see how the utility traces are created so that Java unit tests can be generated properly.

5.2 UtilityPropSeqGenerator

The `UtilityPropSeqGenerator` program is the first step in creating the utility traces. The input to the program is a text file called `utilList` and the output of the program is the property and sequence files. The `utilList` text file contains a list of PuppetMaster utility classes that we want to trace.

Figure 5.2 shows the contents of a `utilList` file, which can be edited by the user. As we can see, there is a list of the various utility classes, such as `ApplicationUtilities`,

Figure 5.2 Example utilList file

```
#utilList
net.rim.puppetmaster.utilities.application.ApplicationUtilities:nostrict
net.rim.puppetmaster.utilities.browser.BrowserUtilities:strict
net.rim.puppetmaster.utilities.contacts.ContactsUtilities:strict
net.rim.puppetmaster.utilities.email.EmailUtilities:strict
net.rim.puppetmaster.utilities.menu.MenuUtilities:nostrict
net.rim.puppetmaster.utilities.ribbon.RibbonUtilities:strict
net.rim.puppetmaster.utilities.tasks.TasksUtilities:strict
net.rim.puppetmaster.utilities.notes.NotesUtilities:strict
```

and `BrowserUtilities`. Each line also contains the keyword `strict` or `nostrict`. These keywords indicate whether or not the classes pertain to a specific application. For example, the `BrowserUtilities` class is `strict` as the methods within the class can only be used with the `Browser` application, whereas the `MenuUtilities` methods are `nostrict` because they can be used with *any* application. This distinction helps make the Java unit test generation smarter as the `strict` classes always get precedence over the `nostrict` classes when it comes to mapping. The `UtilityPropSeqGenerator` reads in the `utilList` file and extracts all of the method data from the classes, using reflection, and saves the output to the property and sequence files.

A property file contains data for all of the methods of a specific class listed within the `utilList` file. For instance, the class `BrowserUtilities` has a property file that contains its method details. The purpose of the property file is to allow a tester to define the parameters of the methods to be traced. To create utility traces, we need to call the methods we want to trace. In order to call the methods, we need to supply parameter values. Therefore, a tester must define the parameter values within the properties file. The properties file will be used by the `UtilityTraceCollector` program to create utility traces.

Figure 5.3 Example property file for BrowserUtilities

```
#net.rim.puppetmaster.utilities.browser.BrowserUtilities
#Wed, 22 Jun 2011 12:00:58

strict=true

#openBrowser:0:params=0
net.rim.puppetmaster.utilities.browser.BrowserUtilities.
openBrowser.0=callableMethod

#openBrowser:1:params=1
net.rim.puppetmaster.utilities.browser.BrowserUtilities.
openBrowser.1=callableMethod
net.rim.puppetmaster.utilities.browser.BrowserUtilities.
openBrowser.param.type.0=java.lang.String
net.rim.puppetmaster.utilities.browser.BrowserUtilities.
openBrowser.param.value.0=default

...
```

Figure 5.3 shows a snippet from the `BrowserUtilities` property file. We can see the header comments telling us the name of the utility class and the time when the file was generated. Following that, we can see the `strict` value is set to `true` since this class only pertains to the `Browser` application. After that, we have a list of methods. Each block in the property file corresponds to a public method in the `BrowserUtilities` class file. In this example we can see the method `openBrowser` listed. The method is polymorphic and that is why we see two versions of it in the example. The first line of each block, which ends with `callableMethod`, tells us that this is a new method being defined. The lines following that are the parameter types and values. For example, the line ending with `param.type.0=java.lang.String` tells us that this is the first parameter (we start counting from 0) and its type is `java.lang.String`. The following line, ending with `param.value.0=default`, indicates the value of this parameter; in this case it is still `default`. It is up to the tester to replace all of the default values with proper values for each method they wish to trace. This is required since it is not possible for the program to automatically determine the parameter values for each of the methods.

Figure 5.4 Example sequence file for BrowserUtilities

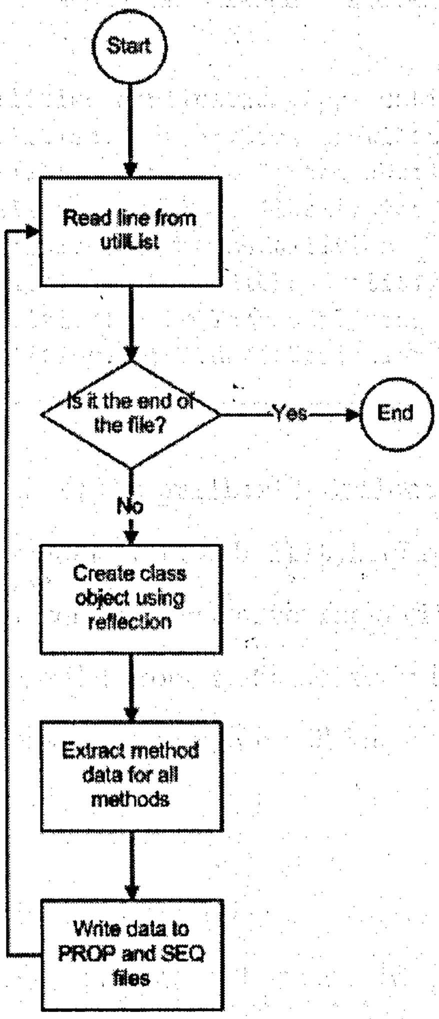
```
#net.rim.puppetmaster.utilities.browser.BrowserUtilities
#Wed, 22 Jun 2011 12:00:58

1:openBrowser:0:params=0
0:openBrowser:1:params=1
0:openBrowser:2:params=2
0:openBrowser:3:params=1
0:goToURL:4:params=2
..
```

Figure 5.4 shows us a piece of the sequence file for `BrowserUtilities`. The sequence files are used to place the methods in the order in which they are to be called. We need to open the `browser` before we can work with it, therefore the `openBrowser` method should be called first; that is one example of the need for a sequence file. The program cannot determine the order in which to call the methods, therefore, a tester needs to select the order.

The sequence file begins with header comments, just as the property file does. Following that is the list of methods. The lines are split into four segments. The first segment is the *sequence number*. Methods with a sequence number of 0 are not run. The sequence order begins at 1 and counts upwards. The second segment contains the name of the method to be called. We can see there are four methods with the name `openBrowser`, so in this case we choose one of them to call. The third segment is the method's *ID*. Since we have polymorphic methods, we need a way to distinguish them. The *IDs* in the sequence file correspond to the *IDs* in the property file so that the tester can match up the proper methods. The fourth segment contains the number of parameters that the method contains. This is there to help the tester easily identify the method they want to call.

To use the sequence file, a tester places the methods they want to trace in order by giving the methods a sequence number greater than 0 and in the order they should be called in. The methods must also have their parameters defined in their property file. Once the sequence is defined and the parameters are defined, the `UtilityTraceCollector` program can be run and the utility traces can be created.

Figure 5.5 UtilityPropSeqGenerator Algorithm


Start
Read line from utilList
Is it the end of the file?
Yes
End
No
Create class object using reflection
Extract method data for all methods
Write data to PROP and SEQ files

5.3 UtilityTraceCollector

The `UtilityTraceCollector` program is the second step in the process for creating utility traces. Its goal is to read in the property and sequence files, build the utility method calls using reflection, and run the methods against the simulator to create log files (which are the utility traces). The simulator will output one log file for each of the methods being called against it. The program also reads in a text file named `utilListToCollect` that tells it which utility classes the tester wants to trace.

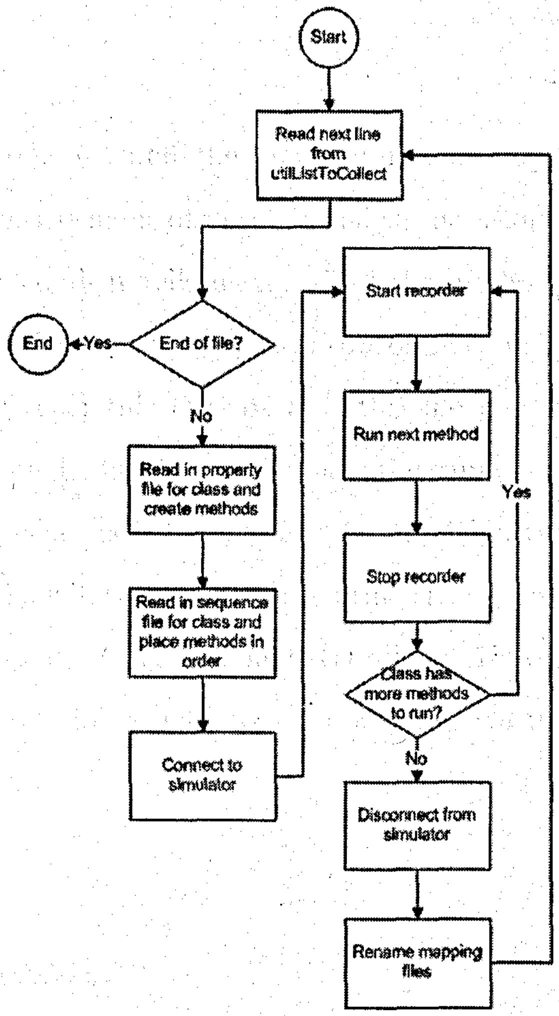
Figure 5.6 Example utilListToCollect file

```
#utilListToCollect
net.rim.puppetmaster.utilities.application.ApplicationUtilities
net.rim.puppetmaster.utilities.browser.BrowserUtilities
net.rim.puppetmaster.utilities.contacts.ContactsUtilities
net.rim.puppetmaster.utilities.email.EmailUtilities
net.rim.puppetmaster.utilities.menu.MenuUtilities
net.rim.puppetmaster.utilities.ribbon.RibbonUtilities
net.rim.puppetmaster.utilities.tasks.TasksUtilities
net.rim.puppetmaster.utilities.notes.NotesUtilities
```

Figure 5.6 shows an example of the `utilListToCollect` text file. It looks almost exactly the same as the file input into the `UtilitySeqPropGenerator` program, except that this file does not contain the `strict/nostrict` keywords. It does not contain the keywords because the property files already have the line `strict=true` or `strict=false` and therefore, the program will know when reading the property file if that class is strict or not.

Figure 5.7 shows the algorithm for the program. We can see that each individual method for each class is run and recorded separately. One run of the algorithm consists of a sequence of method calls, but the trace for each method is stored in

Figure 5.7 UtilityTraceCollector Algorithm


Start
Read next line from utilListToCollect
End of file?
Yes
End
No
Read in property file for class and create methods
Read in sequence file for class and place methods in order
Connect to simulator
Start recorder
Run next method
Stop recorder
Class has more methods to run?
Yes
No
Disconnect from simulator
Rename mapping files

separate file. This means that each utility trace corresponds to only one utility method. As we will see in the next section, when a block of events within a recording are successfully matched against a utility trace, a single method call can be generated to represent that block of events. When all of the event blocks of a recording are matched to method calls, the translation is complete and the Java unit file can be generated. When the utility traces are generated, they are re-named so that their meta-data is stored within the name itself. An example of the naming convention is as follows.

```
net.rim.puppetmaster.utilities.browser.BrowserUtilities.openBrowser.
0.strict.rec
```

The name of the file tells us, and the `TestGenerator` program, details about the method. First, it tells us the name of the class and the name of the method represented by the utility trace. Second, it tells us the `ID` of the method, in this case the `ID` is 0. Finally, it tells us if the method is `strict` or not; in this case it is `strict`. The file extension is ".rec"; this tells us that this file is a utility trace and not a manual recording. All of the utility traces follow the same naming convention. After the utility traces are generated, they are all placed within the same directory. The `TestGenerator` program will read in all of the utility traces contained in the directory and use them for mapping against manual recordings. Now that we know how the utility traces are generated, we need to see how the Java unit files are created by using the utility traces.

5.4 TestGenerator

The `TestGenerator` program is what generates the Java unit tests. The program works by reading in all of the utility traces from a specified directory, as well as reading in a manual recording, then mapping the manual recording to the utility traces, and finally outputting a Java unit file. The program takes in two parameters, the directory containing the utility traces and the manual recording file. The algorithm is shown in figure 5.8 and figure 5.9.

We will now go through the algorithm in detail to understand how it works. We

Figure 5.8 TestGenerator Algorithm Part 1

```
In: utility trace directory UTD, manual recording MR
Out: Java unit file

MappingObject M = []
MatchedMethod methods = []

For each file F in UTD do
    Create mapping object m
    Read data D from F
    Clear noise in D and store in m
    Store F's meta-data in m
    Store m in M
end for

Clear noise from MR
MatchedMethod currentMatch = null
currentLine = 0

while(currentLine < MR.length) do
    matches[] = matchMRtoMappingObjects(MR, M, currentLine)

    if(matches.length == 0) then
        currentMatch = getDefaultMethod()
    else
        currentMatch = matches[0]
        for each match i in matches do
            if matches[i] matches more lines than
              currentMatch then
                if currentMatch is strict then
                    if matches[i] is strict then
                        currentMatch = matches[i]
                    end if
                else
                    currentMatch = matches[i]
                end if
            end if
        end for
    end if
```

Figure 5.9 TestGenerator Algorithm Part 2

```
    Find parameters for the currentMatch method
    Store parameters in the currentMatch object
    Store currentMatch in methods
    currentLine = currentLine + currentMatch.length
end while

Create header text for Java unit file
For each method in methods do
    Create method call text with parameters
end for
Create footer text for Java unit file

Output Java unit file text to a .java file
Return
```

will look at how we clean the noise from the utility traces and manual recordings to make them match better. We will see how we map the utility traces to a manual recording. We will look at how we discover the parameter values for the mapped methods. Finally, we look at how we create the Java unit file.

5.4.1 Noise Removal

The algorithm begins by creating objects to represent the utility traces. The data for each utility trace, along with the meta-data, is stored within its own object. Therefore, each utility trace object corresponds to one utility method. Next, we need to clear the **noise** from the utility traces. The PuppetMaster utility methods execute in a peculiar way and this causes noise in the recordings. For example, noise would be considered extra `keypress` lines at the beginning of the log file. We consider it noise because the actions taken within these methods are not actions a real tester would be likely to make. Therefore, we need to remove the noise from the recordings to allow the utility traces to align properly with the manual recordings.

After the noise is stripped from the utility trace objects, we need to repair the recording data for both the utility traces and the manual recording. We need to repair three things for utility traces and the recording: text input lines, menu lines, and button lines. The first thing we need to repair is the text input lines. For instance, when a user types a string on the device, each character typed generates its own event and takes up a line of text in the recording, such as the following.

```
keypress:S:email app:Name:
keypress:a:email app:Name:S
keypress:n:email app:Name:Sa
keypress:t:email app:Name:San
keypress:o:email app:Name:Sant
```

This is a problem for a couple of reasons. First, we want to differentiate between when a user is typing a value into a text field and when a user is simply pressing a key to navigate. Second, we want to use the value typed in by the user as a parameter later on, and so we will have to extract that value from the recording. It is difficult to extract the proper value from the recording when it is spread across multiple lines. To address the first problem, we have to define navigation keys and alphanumeric keys so that we can tell when the user is navigating the screen or typing in a value. We defined the navigation keys as the `SEND`, `MENU`, `END`, `ENTER` and `ESCAPE` keys, and we defined all alphanumeric and punctuation keys as simple input keys. To deal with the input values being spread across multiple lines, we decided it would be best to compress the lines into a single line, which we call an `alpha` line. The `alpha` lines represent input from the user. The alpha lines are created by compressing lines from the same input into a single line. The result looks as follows.

```
keypress:alpha:email app:Name:Santo
```

To create such a line, we take the last line in the sequence and the character that was typed in last, in this case the "o", and append it to the end of the string. Next, we change the second segment of the line, which is the keyvalue, to the word alpha. This tells us that this line is an alpha line and represents input into a field. We do this for every block of input for both the utility traces and the manual recording.

The remaining two things we need to repair are how the menu and button lines are stored. They are originally stored as in the following example.

```
keypress:menu:email app:email field:email value
display:menu:send
display:menu:save
display:menu:close

keypress:button:email app:email field:email value
display:button:cancel
display:button:ok
```

In the first example, the user scrolled through the menu and selected the close option. However, the user scrolled passed the send and save menu options. The last display line containing the "menu" keyword is the value that was selected by the user. Therefore, the other options shown are not required. We only need to know the menu option selected by the user and so we remove the non-required lines. The same goes for the button lines. The result of clearing the lines in the above example is as follows.

```
keypress:menu:email app:email field:email value
display:menu:close

keypress:button:email app:email field:email value
display:button:ok
```

5.4.2 Mapping Recordings to Utility Traces

Now that we have fixed the format of our recordings, we can begin mapping the manual recording to the utility traces. The mapping algorithm works slightly differently for `strict` and `nostrict` methods. When mapping `strict` methods to the recordings, the values for the selected components (display fields, application class names, etc.) must match exactly those in the manual recording. For `nostrict` methods, this is not required. It is not required for `nostrict` methods because the methods can be applied to multiple applications and therefore it is not likely that the names of the GUI components would match. Therefore, we can say that the `strict` methods have a textual strictness while the `nostrict` methods do not. Since the `strict` methods have a textual strictness, we allow them to match the manual recordings with a *weak shape* strictness. This means that not every line in the utility trace must match every line of the event block in the manual recording. The opposite is true for the `nostrict` methods. Since they have no textual strictness, we make them have *strong* *shape* strictness, which means that every line in the utility trace must match every line in the manual recording. This allows us to bring some balance to the mapping algorithm for the two different types of methods.

The mapping algorithm starts at the first line of the manual recording. It attempts to match the first line to *any* line in the `strict` utility trace and the *first* line in the `nostrict` utility trace. If it matches a line in a `strict` file, then all subsequent lines must match until either the end of the utility trace or until the end of the manual recording. If, for the `strict` files, all lines match from the first matched line, the matching is successful and the utility trace is a potential candidate. The `nostrict` utility traces must match all of their lines to the manual recording. If all of the lines are matched, it is considered a successful mapping and is a potential candidate. Once all potential candidates have been matched, they are compared to each other to see which one matches the best. The criteria to determine which utility trace matches the best are as follows.

1. Strict utility traces are better than nostrict traces.
2. The utility trace that covers more lines is best.

We compare the potential candidates and see which is the best based on the above criteria. We give precedence to `strict` utility traces as they pertain to a specific application and are therefore better to use. If more than one `strict` trace matches or if no `strict` traces match, but more than one `nostrict` trace matches, we check to see which candidate covers more lines and use the one that covers the most. We consider this better as it tells us that the method does more work than the other candidates. Once we have matched a utility trace to the block of events in the manual recording, we match again starting from the next unmatched line and continue until all lines have been matched.

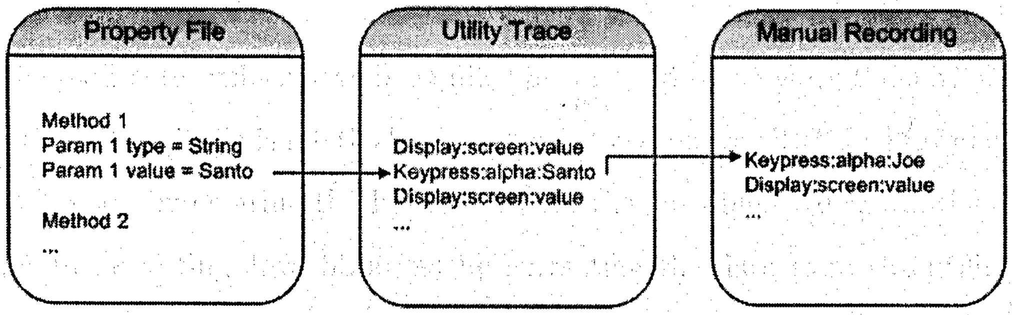
It is possible that a line in the manual recording does not match any utility trace. In this case we generate a default method call. If the line that was unmatched is an `alpha` line, we generate a method call that types a phrase, as all `alpha` lines represent a user typing a phrase. If the unmatched line is not an `alpha` line, we generate a method call that simulates pressing a single key. We take the value of the key being pressed from the data in the line. We then continue the matching algorithm from the next unmatched line in the manual recording.

5.4.3 Parameter Discovery

Each matched utility trace represents a utility method in PuppetMaster. It is possible that the methods contain parameters. Therefore, we must attempt to discover the parameters that the user entered so that we can supply them to the methods in the Java unit file when it is generated. In order to determine where the right parameters are for each method, we need to make connections all the way back to the property files. The property files contain the parameters used in the creation of the utility traces. The idea is to look up the parameter value in the property file, locate that value in the utility trace, and then find the corresponding line in the manual recording. We then take the value from the line found in the manual recording as the parameter.

Figure 5.10 shows how the parameter matching works. In the example shown, the property file contains the string value "Santo". The line containing this string is located and matched against the lines in the manual recording. Since there is a suc-

Figure 5.10 Parameter Matching



cessful match, we can then take the value from the manual recording as our parameter; in this case the value would be "Joe". The string Joe is what the user entered when they were recording the test case and so it should be the value supplied to the method when it is called.

When the parameter algorithm is attempting to match the utility trace line to the manual recording line, it only attempts to match the utility trace line to a line within the block of events of the manual recording that correspond to the utility trace. This stops the algorithm from taking a parameter value used in another method as its own as it is possible that the line in the utility trace will match more than one line.

It is possible that a parameter will not be found in the utility trace or that it is not found in the manual recording. In this instance the parameter is set to the default value "FILL_ME_IN". After the Java unit test is generated, the tester will have to supply the parameter themselves. This happens because not all parameters defined in the property files can be located within the utility traces. Some of the parameters defined are used privately within the method call and are not used as input into a GUI component, and therefore, are never recorded. This makes the parameters impossible to locate within a recording file.

5.4.4 Creating the Java Unit File

Once the parameter values have been filled in, we need to generate the Java unit file. Generating the unit file is relatively easy as we already have all of the information we need. We start by creating the header text for the Java file. We create the `import` lines (as in importing Java libraries) by extracting the data from the utility trace objects. For each utility trace class that is to be used, we create an `import` line. Next, we generate the method calls to the utility methods. We know which methods to call as we just ran the mapping algorithm. We create the method calls and supply the parameter values that we discovered.

Finally, we generate the footer information for the Java file. The footer information is simply the line to disconnect from the simulator or device, and some closing braces. We output the generated text to a Java file and the entire algorithm is complete.

Chapter 6

Results

In this chapter we will look at the Java unit files generated by our system. First, we will look at the utility traces that we generate. Next, we will look at some example Java unit files that were generated and we will analyze them to determine if they are doing what they are supposed to do. We will determine the accuracy of the unit tests by comparing the generated method calls against the initial recording. If, for example, in the initial recording, the tester recorded an email being sent, but in the unit file, the email methods are not called, we will know the generated test was not successful. Following that, we will look at the limitations of our system.

6.1 The Utility Traces

Before we could begin recording tests and generating unit tests, we had to create the various utility traces. We decided to pick the main utility classes and generate utility traces for their methods. The classes we chose are as follows.

Utility Class	Purpose
BrowserUtilities	Methods for the browser application (open, close, bookmarks, etc.)
ContactsUtilities	Methods for phone contacts (names, numbers, etc.)
EmailUtilities	Methods for the email application (send, open, etc.)
MenuUtilities	Methods to interact with any menu (open, close, etc.)
TasksUtilities	Methods for the task application (create, delete, etc.)
NotesUtilities	Methods for the memo application (create, edit, delete, etc.)

This set of utility classes would allow us to generate tests for the various applications, such as `Browser`, `Contacts`, `Memo` and so on. As well, there are generic utilities such as the menu utilities that can be applied to any application running on the BlackBerry. The program already incorporates the generic key utilities for when it generates method calls when no utility trace can be matched to the manual recording, and so that class is not included here. We ran these classes through `UtilityPropSeqGenerator` to create the property and sequence files. We then filled in the necessary parameter values for the methods we wanted to map. After that was done, we ran the `UtilityTraceCollector` program to generate the utility traces. The utility traces that were generated are listed here.

```
BrowserUtilities.createBookmark.16.strict.rec

BrowserUtilities.exitBrowser.11.strict.rec

BrowserUtilities.goToURL.5.strict.rec
BrowserUtilities.openBrowser.0.strict.rec
BrowserUtilities.refreshContent.57.strict.rec
ContactsUtilities.createNewContact.4.strict.rec
ContactsUtilities.deleteContact.36.strict.rec
ContactsUtilities.openContacts.0.strict.rec
EmailUtilities.closeEmail.49.strict.rec
EmailUtilities.deleteEmail.35.strict.rec
EmailUtilities.openApplication.0.strict.rec
EmailUtilities.openEmail.42.strict.rec
EmailUtilities.sendEmail.13.strict.rec
MenuUtilities.selectMenuItem.3.nostrict.rec
NotesUtilities.createNote.4.strict.rec
NotesUtilities.deleteNote.6.strict.rec
NotesUtilities.editNote.7.strict.rec
NotesUtilities.openNotesApp.2.strict.rec
TasksUtilities.createTask.2.strict.rec
TasksUtilities.deleteTask.12.strict.rec
TasksUtilities.openTasksApp.0.strict.rec
```

The file names all contain a prefix of net.rim.puppetmaster.utilities.* where * is the name of the specific utility, but these were left out for ease of reading. Now that we had our utility traces, we began recording and generating test cases that correspond to the above methods.

6.2 Java Unit Tests

In this section we will look at three example test cases that we recorded and transformed into Java unit files. Instead of showing the actual recording, which consists of the keypress and display lines, we will look at the steps taken during the recording. We will then look at the unit file that was generated and see how well it matched with our recording steps.

6.2.1 Browser Bookmark Test

In this test we conducted the following steps.

1. Open the Browser application.
2. Press the menu button.
3. Select the Create Bookmark option.
4. Type "home" as the bookmark name.
5. Type "www.home.com" as the bookmark URL.
6. Press the add button.
7. Press the menu button.
8. Select the Delete option.
9. Confirm deletion.

The test consists of a tester creating a bookmark through the `Browser` application and then deleting it. After we ran the recording through TestGenerator we got the Java unit file in figure 6.1.

Figure 6.1 Browser Bookmark Test

```
//import lines snipped

public class BookmarkTest {
    public static void main(String[] args) {

        try {
            System.setProperty("PuppetMasterHome",
                "C:/PuppetMaster/");
            DeviceController.getInstance().setup();
            BrowserUtilities.openBrowser();
            BrowserUtilities.createBookmark("home", "www.home.com");
            MenuUtilities.selectMenuItem("Delete", "FILL_ME_IN");
            FieldUtilities.focusByName("Delete");
            KeyUtilities.pressKey(Key.ENTER);
            DeviceController.getInstance().shutdown();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

We can see that the unit test generated reflects accurately the steps previously listed. The lines up to and including the `DeviceController` line are the standard header for a test, which is discussed in section 5.4.4. The `DeviceController` is what makes the connection to the simulator. The unit test starts by calling the `openBrowser` method to start the `Browser` application. Following that, it calls the `createBookmark` method and supplies the correct parameters. The test then selects the `Delete` option from the menu item list and finally it presses the `Delete` button to confirm the deletion. The `focusByName` and `pressKey` method calls are both examples of the generic method generation. There was no utility trace that corresponded to the pressing of a button, and so generic method calls were created to deal with pressing the button. The only thing required by the tester at this point is to fill in the parameter that says FILL_ME_IN. This parameter could not be discovered in the recording file. The reason this parameter could not be discovered is because the parameter required is the device PIN. The device PIN is not logged in the recording and therefore it is impossible for the parameter to be discovered.

6.2.2 Create New Contact Test

In this test we conducted the following steps.

1. Open the Applications sub-folder.
2. Open the Contacts application.
3. Select New Contact from the menu list.
4. Enter the name "Santo" as the first name.
5. Select Save from the menu list.
6. Select Delete from the menu list.
7. Close the Contacts application.

This test consists of a tester opening the `Contacts` application from the `Applications` folder. The tester selects the `New Contacts` button from the menu and enters the first name `Santo`. The tester then saves the contact and then selects the `Delete` option from the menu. The contact is deleted and the tester closes the application. The results are in figure 6.2.

Figure 6.2 Create Contact Test

```
//import lines snipped

public class ContactTest {
    public static void main(String[] args) {

        try {
            System.setProperty("PuppetMasterHome",
                "C:/PuppetMaster/");
            DeviceController.getInstance().setup();
            ContactsUtilities.openContacts();
            ContactsUtilities.createNewContact("FILL_ME_IN", "Santo");
            ContactsUtilities.deleteContact("Santo");
            MenuUtilities.selectMenuItem("Close", "FILL_ME_IN");
            DeviceController.getInstance().shutdown();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

We can see that the unit test reflects accurately the steps listed previously. The unit test calls the `openContacts` method, which will open the `Contacts` application. The `createNewContact` method is called with the parameter `Santo`, which matches what was typed in during the recording. The method also contains a FILL_ME_IN parameter. The reason for this is that the parameter required is a constant that tells the method which field to place the name in. This constant is not output to the recording and so it cannot be discovered. After the contact is created, it is deleted. This is done using the `deleteContact` method. The name of the contact is correctly supplied to the method as a parameter. Finally, the application is closed via the `selectMenuItem` method. Once again, the FILL_ME_IN parameter requires the device PIN. The conversion was near perfect in this case and only requires a small adjustment from the tester.

6.2.3 Create and Edit A Memo Test

The `Memo` application allows a user to create basic text files. The user can create a memo that consists of a memo title and a memo body. The user can edit the memo, both the title and the body, and re-save it. The utility methods that access the `Memo` application use the word "note" instead of memo, but it means the same thing for our purposes. The test consists of the following steps.

1. Open the Applications sub-folder.
2. Open the Memo application.
3. Select the New Memo option from the menu.
4. Give the memo the title "memo title".
5. Give the memo body the value "memo body".
6. Save the memo.
7. Select the Edit option from the menu.
8. Change the title to "new title".
9. Change the body to "new body".
10. Save the memo.

11. Select the Delete option from the menu.

12. Close the application.

This test tests the process of creating, editing and deleting a memo. It creates the initial memo, changes the values of the title and body and saves the changes. The memo is then deleted and the application is closed. The unit test generated is in figure 6.3.

Figure 6.3 Create and Edit Memo Test

```
//import lines snipped

public class MemoTest {
    public static void main(String[] args) {

        try {
            System.setProperty("PuppetMasterHome",
                "C:/PuppetMaster/");
            DeviceController.getInstance().setup();
            MenuUtilities.selectMenuItem("Open Tray", "FILL_ME_IN");
            FieldUtilities.focusByName("Applications");
            KeyUtilities.pressKey(Key.ENTER);
            NotesUtilities.openNotesApp();
            NotesUtilities.createNote("memo title", "memo body");
            NotesUtilities.editNote("Memo title", "new title",
                "new body");
            NotesUtilities.deleteNote("New title");
            MenuUtilities.selectMenuItem("Close", "FILL_ME_IN");
            DeviceController.getInstance().shutdown();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Once again the unit test reflects to the recording accurately. The unit test calls the `openNotes` method, which opens the `Memo` application. It then calls the `createNote` method with the correct parameters, which will create our memo. Following that, it opens the memo we just created and edits the title and body. Finally, it deletes the memo using the newly-edited memo title and closes the application. The only FILL_ME_IN spots required are the device PIN values we have seen in the previous tests.

6.2.4 Limitations

The three results shown above show the accuracy of the unit files generated. However, there are instances when the accuracy is not always so great. These instances occur when something unexpected interferes with the recorder, such as a pop-up text box. If something appears on the screen when a user is typing a value into a field, the recorder will insert a display line between the keypress lines, thus not allowing the program to properly match the manual recording to the utility traces. Furthermore, this makes it not possible to locate the proper parameters for the method calls. If this occurs, the `TestGenerator` program will simply generate the generic methods to locate the fields on the screen and input text. Since the parameters are not able to be properly discovered, the incorrect values will also be used as parameters.

Fortunately, this does not happen often and only occurs in a few applications. The problem is that the PuppetMaster software was not designed with a recorder in mind and so no considerations were made to deal with these scenarios when they occur. However, if one were to design their own utility classes with a recorder in mind, these problems could be dealt with accordingly.

Regarding the FILL_ME_IN parameters, we have tried to use the data from the property files to replace them and it does work to some extent. This method works well for finding PIN values and other values that are used privately in the utility methods. However, this can cause problems for parameters that are output to the recording file but are interrupted by a display message. The parameter used in this instance will be from the property file and not what the user entered when they recorded the test. Since this occurs, we decided to not use this method.

Chapter 7

Conclusion

In this chapter we will conclude the thesis. We will first look at the implementations of our software and how our methodologies can be applied to software in general. After that, we will look at the future work to be done with our own software to see where things can be improved and how to make our software more robust. Finally, we will summarize the work done in this thesis.

7.1 Implementations of Our System

The methodologies described in this thesis for a record and playback system can be applied to any system that uses a GUI and allows for GUI components to be discovered programmatically. In order to apply the methods we have described, one would need to create a utility system of their own that is capable of interacting with GUI components. The system we described was created to work with the BlackBerry architecture and worked within their PuppetMaster test system. The PuppetMaster system came with very specific utility classes that allowed us to interact with specific applications such as the `Browser` application or the `Email` application. However, if one were to create a system of their own following our methods, they would only need to create general utilities that allow for interaction with the GUI components and not with any specific application.

An example system that could be created is as follows. We want to create a record and playback tool for all Java software that uses Swing. Swing is the main Java GUI toolkit and contains the various components to create a GUI, such as frames, textboxes, and buttons. In order to create this record and playback tool, we first need to create utility classes. We could create these classes in the following manner. For each GUI component that a user can interact with, we create a utility class (For example, `JButtonUtilities`, `JCheckBoxUtilities`, and `JTextFieldUtilities`). Each of these classes contains methods that can be called to interact with a GUI component. The `JButtonUtilties` class could contain methods to press a button, hold a button, and so on. These utility classes would also need to implement a method to discover the GUI components programmatically using reflection. As there is a limited number of ways in which a user can interact with a GUI component, the utility classes would not contain an excessive number of methods. With the utility classes created, a user should be able to write unit tests that allow them to call the utility methods and interact with the GUI.

The utilities listed above are fairly generic and low-level as they deal directly with the Swing components. If we remember the utilities provided in PuppetMaster, we know that there are application-specific utilities, such as `BrowserUtilities`, that are smart enough to locate the `Browser` application and run it. The above utilities regarding

Swing would not be this smart. Therefore, to create these smart utilities, the system creator would have write application-specific code. These application utilities could use these low-level Swing utilities with some extra logic involved. For instance, if there was an application utility called `MenuUtil.selectItem(item)` that selects a menu item based on the parameter, then this utility would contain an algorithm to find the appropriate menu item and select it using one of the generic Swing utilities. Creating the low-level utilities, and discovering the GUI components, would likely be the most difficult part. The application-specific utilities would consist of calls to these low-level utilities and so they would be easier to create.

Following that, the system creator would need to create a method to record the user's interactions; this is where the method may differ from our own. The method may differ for the recorder because our recorder received events generated from the operating system, where this will not be possible for desktop Java applications. In Java, every GUI component that a user can interact with must implement action listeners. Action listeners receive the event notifications when a user interacts with the GUI component that the action listener is connected to. Therefore, to record the events of the system, the programmer of the system could add lines to the action listeners that output data to a text file. The data would contain the name or type of GUI component, and the details of the event that occurred. The programmer could then run the utility methods on the system to record the baseline utility traces. Once the baseline has been created, tests could be recorded and transformed using an algorithm based on our own as described in this thesis. The resulting Java file would contain method calls to the utilities described earlier.

One downside to this method is that the recording lines for the action listeners would need to be added in every time a component was created and for every new application. Furthermore, depending on how the event details are recorded, the algorithm may have to be changed as the string matching might break. A smarter way to go about it would be to write an instrumentation tool that inserts probes into the action listeners automatically. This would allow for consistency across systems and remove the problem of the string matching algorithm breaking. An instrumentation tool could insert probes wherever an action listener is defined so that the event details are logged properly. The inserted probes could be customized to match the type of GUI component being logged so that each type of component can save their necessary information such as the user's input text.

It would be ideal for the creators of the GUI systems to create and release these record and playback tools themselves so that their users can use them immediately. Of course, this is unlikely. However, it is possible for any user to create their own record and playback tool in light of our system and any company designing their own system could also implement our methods.

7.2 Future Work

The future work for our system involves generating more utility traces so that more kinds of tests can be recorded and generated. As well, we want to figure out ways to deal with interruptions when recording test cases such as when the system updates the display when a user is inputting text. We would also like to figure out a way to deal with parameters that cannot be located or that do not show up in the recording file. Our current thought is to stop or ignore display updates that occur when a user is inputting data into a field; this way the input is not divided and should allow the matching algorithm to work properly. We would also like to allow our system to work with RIM's new operating system. They are updating their systems to a new architecture and so our record and playback system may no longer work. We would like to aid RIM in upgrading our system to work with their new architecture.

7.3 Summary

The system created for, and described in this thesis is a record and playback system for the BlackBerry smartphone. The system works in conjunction with RIM's testing framework, PuppetMaster. PuppetMaster contains a set of utility classes that allow programmers to easily interact with the GUI components of the system, such as a button or menu. Our system consists of a recorder, which records events that occur on the BlackBerry and outputs the events to a log, and a test generating system, which translates recordings to Java unit tests. The tests are generated by using a mapping algorithm that matches a manual recording against a set of pre-recorded utility traces. For each match that occurs within the algorithm, a method call to a utility method is generated and output to a Java unit file. Once all the lines of the manual recording have been matched to a method, the algorithm is complete. The resulting test case will consist of calls to PuppetMaster utility methods.

References

[1] Abbot framework for automated testing of Java GUI components and programs. `http://abbot.sourceforge.net/doc/overview.shtml/`. [Online. Accessed August 2011].

[2] Mohamed A Abdel Salam, Arabi E Keshk, Nabil A Ismail, and Hamed M Nassar. Automated testing of Java menu-based GUIs using XML visual editor. *2007 International Conference on Computer Engineering Systems*, pages 313–318, 2007.

[3] M. Assem, A. Keshk, N. Ismail, and H. Nassar. Specification-driven automated testing of Java swing GUIs using XML. *5th International Conference on Information and Communications Technology, 2007.*, pages 84–88, 2007.

[4] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proc. Foundations of Software Engineering*, pages 253–264, 2006.

[5] Monty L. Hammontree, Jeffrey J. Hendrickson, and Billy W. Hensley. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '92, pages 431–432, New York, NY, USA, 1992. ACM.

[6] Jemmy: Java UI testing tool. `http://java.net/projects/jemmy/`. [Online. Accessed December 2011].

[7] Shrinivas Joshi and Alessandro Orso. SCARPE: A technique and tool for selective capture and replay of program executions. In *International Conference on Software Maintenance*, pages 234–243, 2007.

[8] JUnit.org resources for test driven development. `http://www.junit.org/`. [Online. Accessed October 2011].

[9] Edward Kit. *Software Testing in the Real World: improving the process*. Addison-Wesley Publishing Company, Inc., 1995.

[10] Atif M. Memon. GUI testing: Pitfalls and process. *Computer*, 35:87–88, August 2002.

[11] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27:144–155, 2001.

[12] J. D. Newmarch. Testing Java Swing-based applications. In *Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*, TOOLS '99, pages 156–165, Washington, DC, USA, 1999. IEEE Computer Society.

[13] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. In *Proceedings of the third international workshop on Dynamic analysis*, WODA '05, pages 1–7, New York, NY, USA, 2005. ACM.

[14] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D.Ernst. Automatic test factoring for Java. In *ASE 2005: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, USA, November 9-11, 2005.