Test Data Generation For Event Sequence Graph

by

S. M. Ashraful Hasan

Supervisor: Mark Harman

London, 2007
I would like to express my gratitude to the following people who helped and supported me throughout this project. First of all I thank my family for their support and Mark Harman for his invaluable advice and lessons. Many thanks to Atif M. Memon from University of Maryland who actually helped me throughout the whole project with resources and advice.

Odinaldo Rodrigues for his help during second semester. Thanks to Jammy for helping me to compile TerpOffice. All the members of CREST for answering different project related questions.

Rezaul Sarkar, Wakil Ahmed and Abdur Rahman for helping me with their computers. Yue JIA, Sayed Islam, Fayezin Islam, Yasir Jafar, Xiang Ren for project related discussion during project meeting.

And last but certainly not the least Syed Shahid and Sadia Haque for helping me during the whole course.
Abstract

The widespread use of computer is largely contributed by the Graphical User Interface (GUI) which provides easy to use communication interface between humans and computers. GUIs provide support for building more complex softwares by hiding its complexity from the user. This is achieved by ensuring user friendly environment where the user does not require to know commands to operate the software applications. To ensure the flexibility of end user a set of GUI component has been invented for building simple to complex user interface according to the requirements of the application.

With the growing complexity of softwares, GUIs have become more complex involving both GUIs and underlying software testing. This is because GUIs have different characteristics than traditional softwares and consequently conventional testing techniques do not apply to GUI softwares. As a result, many testing frameworks and tools have been developed in recent years based on different testing technique for testing GUI. Although these tools have different approaches to testing (such as manual, semi automatic, and automatic), their main purpose remains the same i.e., to test the GUI and detect faults. GUI testing has become an integral part of the GUI development process as GUIs got more and more complex. Regardless of this fact no empirical study has been made to compare fault-detection ability of different GUI testing frameworks. Therefore, their contribution to the development process is still unknown. To provide an insight into the problem, this thesis presents the empirical study of fault-detection effectiveness between manual and automated GUI testing techniques based on two different frameworks.

In this paper several questions have been answered through empirical study. At first the code-coverage ability of manual GUI testing framework “jfcUnit” is discussed. Then the fault-detection effectiveness of these test cases with automatically generated test cases using Event Interaction Graph is compared. Several applications have been tested through test cases. Realistic faults have been seeded to measure their fault-detection ability. A comparison has been made to illustrate their effectiveness.
## Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Objective</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Project Proposal</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Project Overview</td>
<td>2</td>
</tr>
<tr>
<td>2 Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 What is GUI?</td>
<td>3</td>
</tr>
<tr>
<td>2.2 GUI Testing Tool</td>
<td>4</td>
</tr>
<tr>
<td>2.3 GUI Testing Framework: jfcUnit</td>
<td>4</td>
</tr>
<tr>
<td>2.4 Test Driven Development</td>
<td>6</td>
</tr>
<tr>
<td>2.4.1 Maintain an exhaustive suite of programmer tests</td>
<td>6</td>
</tr>
<tr>
<td>2.4.2 No code goes into production unless it has associated tests</td>
<td>6</td>
</tr>
<tr>
<td>2.4.3 Write the tests first</td>
<td>6</td>
</tr>
<tr>
<td>2.4.4 Tests determine what code needs to write</td>
<td>6</td>
</tr>
<tr>
<td>2.5 Extreme Programming</td>
<td>7</td>
</tr>
<tr>
<td>2.6 Event Flow Graph</td>
<td>7</td>
</tr>
<tr>
<td>2.7 Event Interaction Graph</td>
<td>8</td>
</tr>
<tr>
<td>3 Problem Statement</td>
<td>11</td>
</tr>
<tr>
<td>4 Empirical Study</td>
<td>13</td>
</tr>
<tr>
<td>4.1 Performed Tasks</td>
<td>13</td>
</tr>
<tr>
<td>4.2 Application Under Test</td>
<td>14</td>
</tr>
<tr>
<td>4.3 Writing Test Cases</td>
<td>14</td>
</tr>
<tr>
<td>4.4 How much coverage is possible using JfcUnit test cases?</td>
<td>15</td>
</tr>
</tbody>
</table>
CONTENTS

4.4.1 TerpPaint .................................................. 15
4.4.2 TerpWord .................................................. 17
4.4.3 TerpSpreadSheet ......................................... 18
4.4.4 TerpPresent ............................................... 20
4.5 What Parts of The code Cannot be Executed by These Test Cases? 21
4.6 What is The Fault Detection Ability of These GUI Test Cases? 22
  4.6.1 GUI fault-detection ability ............................ 22
  4.6.2 Code related fault-detection ability ................. 24
4.7 A Comparison of Fault-detection Ability of JfcUnit Test Cases
  With "Event Sequence Graph" Generated Test Cases ........ 27
  4.7.1 Reason behind poor coverage ......................... 29
4.8 What are the weaknesses of these test cases? .............. 31
4.9 Threats to Validity ......................................... 32

5 Future Work .................................................. 33

6 Conclusion .................................................... 34

Bibliography .................................................... 35

A Appendix ....................................................... 38
  A.1 Clover ...................................................... 38
  A.2 Code Coverge .............................................. 38
  A.3 Widget ..................................................... 38
  A.4 Component ............................................... 38
  A.5 Method Coverge .......................................... 39
  A.6 Test Case ................................................ 39
  A.7 JUnit ...................................................... 39

B Appendix ....................................................... 40
  B.1 Fault Matrix ............................................... 40

C Appendix ....................................................... 41
  C.1 Application Source Code ................................ 41
# List of Figures

2.1 The Microsoft Notepad Graphical User Interface. ....................... 4  
2.2 Typical Java Application Tier. ........................................... 5  
2.3 An Event-flow Graph for a Part of MS Notepad. ....................... 8  
2.4 Example of an Event-Flow Graph. ................................. 9  

4.1 Coverage Result of TerpPaint. ........................................ 16  
4.2 Coverage Result of TerpWord. ......................................... 17  
4.3 Coverage Result of TerpSpreadSheet. .............................. 18  
4.4 Coverage Result of TerpSpreadSheet. .............................. 20  
4.5 GUI Based Fault-Detection Ability of jfcUnit. ....................... 23  
4.6 Overall GUI Faults Detected by jfcUnit. ........................... 24  
4.7 Code Based Fault-Detection Ability of jfcUnit. .................... 26  
4.8 Fault Detection Ability of Automatically Generated Test Cases Using Event-Interaction Graph. ............................... 28  
4.9 Fault Detection Ability of Manually Generated Test Case Using jfcUnit. 28
List of Tables

4.1 TerpOffice Applications............................................. 14
4.2 Comparison of Code Coverage of Different Application in TerpOffice. . 15
4.3 Terp Paint Not Covered............................................. 16
4.4 TerpWord Not Covered............................................. 17
4.5 TerpSpreadSheet Not Covered..................................... 19
4.6 Fault Seeded in TerpOffice....................................... 23
4.7 Fault Seeded in TerpOffice....................................... 25
4.8 Fault Seeded in TerpOffice by Category......................... 25
Chapter 1

Introduction

1.1 Objective

In the early days computer applications were command oriented where the user had to know the commands to operate applications. With increasing processing powers computer applications has changed from command line to graphical user interfaces. Now a single application uses numerous visible and invisible components and widgets for building the GUI to increase the flexibility of the user. For an example, a simple Microsoft Notepad application consists of more than 60 components. All these components are integral parts of the application. These components are not the only elements that change the user interfaces. All these components have underlying events and codes to ensure a seamless communication between user and the application. In Microsoft Word there are a total of 4,210 events. Of that, 80 events open menus, 346 events open windows and 196 events close windows. The remaining 3,588 events interact with underlying codes [4]. As a matter of fact, Microsoft Notepad application performs the same basic set of operations as MS Dos Edit tool (implemented around twenty years ago). But their implementation has changed drastically over the years. Now the end user can perform exactly the same task but in multiple ways. For example, a simple “Cut” operation can be performed by the user in three different ways. First, user can select the desired object, press the “Edit” menu and select the “Cut” menu button. Second, user can click the mouse right button and select the “Cut” option from the pop up menu. Third, the user can press the keyboard shortcut “Ctrl+X” to cut the object.

To ensure such kind of end user flexibility, development process adapts with new technology. Usage of different types of components and widgets become an integral part of the development process. With the implementation of these components and widgets, the underlying codes become complex. Due to the complexity, unexpected behavior related to the graphical user interface and application is not uncommon.

To solve such kind of bugs GUI testing is very important. GUIs are typically highly complex pieces of softwares, and testing their correctness poses to be an immense challenge. This is due to the fact that the number of possible GUI interactions are enormous and the measure of coverage developed for conventional software testings do not apply well to GUI testing. To overcome this drawback, different kinds of GUI testing frameworks have been proposed in recent years and many of them are successfully implemented. Example of some tools include extensions of JUnit [18] such as JfcUnit, Abbot, Pounder and Jemmy Module [4]. Despite their increasing use, no study has been made to compare their fault-detection capability or code coverage ability.
1.2 Project Proposal

This project aims to answer multiple questions related to GUI testing by generating test cases. In the first stage, the project considers JfcUnit- a popular manual GUI testing framework to generate test cases. Depending on GUI testing, this project tries to measure their method coverage ability, reasons that might affect their coverage ability and the methods that are not possible to cover using the framework. In the second stage, these test cases are used to compare the fault-detection ability with the result obtained from an automatic test case generation tool- “DART”∗, which uses EFG (Event Sequence Graph) and EIG (Event Interaction Graph) in the test case generation process. To compare the fault detection ability, same type of faults has been seeded in both types of test cases to observe their effect.

1.3 Project Overview

The project is divided into the following sections:

1. Background presents a detailed review of related research papers and existing tools and techniques.
2. Problem statement describes the shortcomings of the study in this area and strengthens the purpose of the project.
3. Empirical study answers different questions regarding test cases through proper investigation.
4. Future work suggests some extension of study area based on this paper.
5. Conclusion sums up what has been discovered through research and testing.

∗DART- "Daily Automated Regression Taster" was developed by Atif M. Memon and Quing Xie. For further information please read the mentioned paper in [4]
Chapter 2

Background

2.1 What is GUI?

A GUI is composed of objects (buttons, menus, grids) using metaphors familiar in real life. The software user interacts with the objects by performing events that manipulate the GUI objects as one would real objects. Events cause deterministic changes to the state of the software that may be reflected by a change in the appearance of one or more GUI objects. Moreover, GUIs, by their very nature, are hierarchical [1]. This hierarchy is reflected in the grouping of events in windows, dialogs, and hierarchical menus. For example, all the "options" in MS Internet Explorer can be set by interacting with events in one window of the software’s GUI. The important characteristics of GUIs include their graphical orientation, event-driven input, hierarchical structure, the objects they contain, and the properties (attributes) of those objects. Formally, we define the class of GUIs of interest as follows:

Definition: A Graphical User Interface (GUI) is a hierarchical, graphical front-end to a software that accepts as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output. A GUI contains graphical objects and each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI [2].

The above definition specifies a class of GUIs that have a fixed set of events with deterministic outcome that can be performed on objects with discrete valued properties. This definition would need to be extended for other GUI classes such as web-user interfaces that have synchronization/ timing constraints among objects, movie players that show a continuous stream of video rather than a sequence of discrete frames, and non-deterministic GUIs in which it is not possible to model the state of the software in its entirety and hence the effect of an event cannot be predicted.

Figure 2.1 illustrates Microsoft Notepad’s GUI. With this GUI the user can perform simple text related operation such as open *.txt file or writing text, edit text through cut, copy, paste, save text in a file etc. At the top the GUI contains a menu bar that allows the user to perform selected operations: clicking File, Edit and Format etc menu button. Other related menus open when user clicked one. When the user click any menu button then we say that user perform a GUI action (e.g., clicks the File command) and thereby generate a GUI event (e.g., opening up a pull-down menu)[3].
CHAPTER 2. BACKGROUND

2.2 GUI Testing Tool

Three different approaches are used to handle GUI software when performing testing [4]. First is to perform no GUI testing at all [5], eventually which leads to compromised software quality in terms of GUI. Second is to use test harnesses that "bypass" the GUI and invoke methods of the underlying business logic as if initiated by a GUI. This approach enforces software design issues as if initiated the GUI software "light" and code all "important" decisions in the business logic [6]. This technique also does not perform testing of the end-user software. Third is to use manual GUI testing tools to do limited testing [7], [8]. Examples of some tools include extensions of JUnit such as JFCUnit, Abbot, Pounder, and Jemmy Module [9] and capture/replay tools [10].

Capture/replay tools (also called record/playback tools) operate in two modes: Record and Playback. In the Record mode, tools such as CAPBAK and TestWorks [11] record mouse coordinates of the user actions as test cases. In the Playback mode, the recorded test cases are replayed automatically. The problem with such tools is that, since they store mouse coordinates, test cases break even with the slightest changes to the GUI layout. Tools such as Winrunner [12], Abbot [13], and Rational Robot [14] avoid this problem by capturing GUI widgets rather than mouse coordinates. Although playback is automated, but it requires significant to create the test scripts, detecting failures, and editing the tests to make them work on the modified version of software. Fourth is to use fully automatic testing tool. The most comprehensive and complete solution is provided by DART [15],[16] that addresses the needs of smoke testing of software applications that have a GUI. DART automates smoke testing of GUIs by using model based testing techniques.

2.3 GUI Testing Framework : JFCUnit

Any application that is written usually has a user interface associated with it so that users can interact with the system (for e.g.: inputting data, generating reports based on data that exists in databases, etc.).
For application developers, testing dialog is an integral part of the development life-cycle. As old code is modified for reasons ranging from adding new functionality to speed optimizations, the risk of dependent functionality getting broken increases. Tests that are written once, and can be run repeatedly is the only solution to avoid manually QA testing. To write these tests, for applications written in java, JUnit provides a solid framework for writing unit tests.

![Diagram of Java Application Tier]

**Figure 2.2: Typical Java Application Tier.**

Looking at the figure 2.2 above, it is clear that tests can be written at any layer which traverses all the layers below it and returns with the results of whatever processing it is supposed to perform. This forms the basis of testing with JUnit. So testing at any layer can be considered to be a "coarser"grained testing of the level below it. Interface test harnesses can also be created to mimic the operation of lower levels, and allow the test cases to be isolated to a single level.

But one important fact that usually gets side-lined, is that since all the testing involves only method calls, this practice cannot be directly applied for testing the UI layer. And if the UI layer is not tested, there still exists some chunk of the application pie that is "untested".

Another point in the practical sense, is that method calls are done in real-time, whereas the testing script/framework should allow for the JVM to render the UI objects, before proceeding with the tests.

While writing GUI tests, the developer has to "mimic" a user's interaction with the application screen being tested. JcUnit enables developers to write test cases for Java Swing based applications. It provides support for:

- Obtaining handles on Windows/Dialogs opened by the Java code.
- Locating components within a component hierarchy that occur within the containers found above.
- Raising events on the found components, e.g. clicking a button, typing text in a text component.
- Handling testing of components in a thread safe manner.
2.4 Test Driven Development

Test-Driven Development (TDD) is a style of development where:

- Developer maintain an exhaustive suite of tests.
- No code goes into production unless it has associated tests.
- Write the tests first.
- The tests determine what code needs to write.

2.4.1 Maintain an exhaustive suite of programmer tests

First the developer has "Programmer Tests" to test that the classes exhibit the proper behavior. Programmer Tests are written by the developer who writes the code being tested. They’re called Programmer Tests because although they are similar to unit tests, they are written for a different reason. Unit tests are written to test that the written code works. Programmer Tests are written to define what it means for the code to work. Finally, they’re called Programmer Tests to differentiate them from the tests that the Customer writes (called, logically enough, Customer Tests) to test that the system behaves as required form the point of view of a user. Using Test-Driven Development implies, in theory, that the developer has an exhaustive test suite. This is because there is no code unless there is a test that requires it in order to pass. Write the test first, then (and not until then) write the code that is tested by the test case. There should be no code in the system which was not written in response to a test. Hence, the test suite is, by definition, exhaustive.

2.4.2 No code goes into production unless it has associated tests

One of EXtreme Programming’s tenets is that a feature does not exist until there is a suite of tests to go with it. The reason for this is that everything in the system has to be testable as part of the safety net that gives the programmer confidence and courage. Confidence that all the code tests clean gives the courage (not to mention the simple ability) to refactor and integrate.

2.4.3 Write the tests first

Write the tests first means when a tasks to do (i.e., some bit of functionality to implement) then write the code that will test that the functionality works as required before implementing the functionality itself. Furthermore, the style suggests, writing a little bit of test, followed by just enough code to make the test pass, then a bit more test, and a bit more code, test, code, test, code, etc.

2.4.4 Tests determine what code needs to write

By writing only the code required to pass the latest test, actually putting a limit on the written code. Only enough code is written to pass the test, no more. That means that at the end the process has the simplest thing that could possibly work. To clarify the whole scenario an example is included below. When working with a list class a simple test case can be written as:

```java
public void testEmptyList()
{
    MovieList emptyList = new MovieList();
    assertEquals("Empty list should have size of 0", 0, emptyList.size());
}
```
To pass this test a MovieList class is required that has a size() method.

2.5 Extreme Programming

Extreme Programming (XP) is defined by four values and a set of practices. However, XP is more than this, more than just doing the practices. It is a way of creating software that embodies the values. The practices are activities that are usually done by the programmer when learn XP.

No discussion of XP can begin without talking about the four variables: cost, time, quality, and scope. A very short discussion is given below for each to describe how they are interrelated, and how they relate to a project and its risk.

- Cost: Every project has a cost. This includes equipment, facilities, and man-hours. Especially man-hours.
- Time: Every project takes time.
- Quality: Every project results in a system with a certain level of quality.
- Scope: Every project consists of a certain set of functionality.

The really interesting thing is how these variables interact. Increasing cost by adding people to the project, may often cause the decrease of the velocity (due to the costs of integrating the additional people) and, hence, time will increase. On the other hand, increasing the cost by buying faster or better equipment or facilities can boost velocity and thus decrease time. It is not possible to decrease time simply by dumping more money into a project, especially early in the project. In fact, it is limited in what one can achieve by controlling cost. Time is often dictated by external constraints such as trade shows, investor requirements, and business commitments. Because of this, time is often relatively fixed.

Quality can be variable, but lowering the quality of the system has serious negative impact. No worker is happy for long if they are coerced into producing goods of poor quality. This can lead to a serious morale issue. Also, low quality code is harder to extend and maintain. If one tries to shorten the time required by limiting quality, it will surely backfire.

Scope is the thing that is most easily controlled. This entails adding or removing functionality from the requirements of the project. By using a few simple techniques, XP makes this work very well. One important technique is to always be working on the most important outstanding feature, according to the customer. That ensures that if unimplemented features are later removed (thereby reducing scope), the most important things have already been done. One mistake that people make is thinking that they can control all four variables. This is impossible. Cost, time, quality, and scope are interrelated such that the value of any one depends on the values of the others. What this means in practical terms is that one can control at most three of the four, but never all four. Attempting to control all four, will most likely fail the project and will certainly be a miserable one to work on.

2.6 Event Flow Graph

A GUI component may be represented as a flow graph. Intuitively, an event-flow graph denotes all possible interactions among the events in a component.

**Definition:** An event-flow graph for a GUI component $C$ is a 4-tuple $(V, E, B, I)$ where:

1. $V$ is a set of vertices representing all the events in the component. Each $v \in V$ represents an event in $C$.

*Both EFG and EIG is invented by Atif M. Memon. Related background study is collected from his different papers. For further information please read the referenced documents.*
2. \( E \subseteq V \times V \) is a set of directed edges between vertices. We say that event \( e_i \) follows \( e_j \) if \( e_j \) may be performed immediately after \( e_i \). An edge \((v_x, v_y) \in E\) if the event represented by \( v_y \) follows the event represented by \( v_x \).

3. \( B \subseteq V \) is a set of vertices representing those events of \( C \) that are available to the user when the component is first invoked.

4. \( I \subseteq V \) is the set of restricted-focus events of the component.

Figure 2.3: An Event-flow Graph for a Part of MS Notepad.

An example of an event-flow graph for a part of the Main component of MS NotePad is shown in figure 2.3. At the top three are vertices (File, Edit, and Help) that represent part of the pull-down menu of MS NotePad. They are "memopen" events that are available when the Main component is first invoked. Hence they form the set \( B \). Once File has been performed in NotePad, any of Edit, Help, Open, and Save events may be performed. Hence there are edges in the event-flow graph from File to each of these events. Note that Open is shown with a dashed oval. This representation is used for restricted-focus events, i.e., events that invoke components. Similarly, About and Contents are also restricted-focus events, i.e., for this component, \( I = \{ \text{Open, About, Contents} \} \). All other events (Save, Cut, Copy, and Paste) are system-interaction events. After any of these is performed in MS NotePad, the user may perform File, Edit, or Help, shown as edges in the event-flow graph.

2.7 Event Interaction Graph

Since an "EFG" models all possible (an extremely large number of) event interactions, it cannot be used directly for rapid testing; abstractions are used to model only specialized (and, hence, a smaller number of) event sequences. In a typical GUI, 20-25 percent of the events are used to manipulate the structure of the GUI; examples include events that open/close windows/menus. For example, in Microsoft Word, of the total 4,210 events, 80 events open menus, 346 events open windows, and 196 events close windows; the remaining 3,588 events interact with the underlying code. The code for events that open menus and windows is straightforward, usually generated
automatically by visual GUI-building tools. This code is very unlikely to interact with code for other events; hence, very few errors are revealed by executing interactions between these events. The remaining events in the GUI are nonstructural events that do not cause structural changes to the GUI; rather, they are used to perform some action; common examples include the Copy event used for copying objects to the clipboard.

Events that interact with the underlying software include nonstructural events and those that close windows. These events are called system-interaction events. Intuitively, the GUI smoke test cases are composed only of these events (and any other events necessary to "reach" the system interaction events). Once these events have been identified in a GUI, the EFGs are transformed to event-interaction graphs, which are used to generate smoke tests. Some definition is given below to help in this transformation. Intuitively, an event-flow path represents a sequence of events that can be executed on the GUI. Formally, an event-flow path is defined as follows:

Definition: There is an event-flow-path from node $n_x$ to node $n_y$ iff there exists a (possibly empty) sequence of nodes $n_1; n_{j+1}; n_{j+2}; \ldots; n_{j+k}$ in the event-flow graph $E$ such that $\{(n_x, n_1), (n_{j+i}, n_{j+k})\} \subseteq edges(E)$ and $\{(n_{j+i}, n_{j+i+1}) for 0 \leq i \leq (k-1)\} \subseteq edges(E)$.

In the above definition, the function $edges$ takes an event-flow graph as an input and returns a set of "ordered pairs", each representing an edge in the event-flow graph. The notation $\langle n_1; n_2; \ldots; n_k \rangle$ is used for an event-flow path. Several examples of event-flow paths from the EFG of Figure 2.4 are:

$\langle File; Edit; Undo \rangle; < File; MatchCase; Cancel \rangle$;
$\langle MatchCase; Editbox1; Replace \rangle$; and $\langle MatchCase; FindNext; Replace \rangle$.

Definition: An event-flow-path $\langle n_1; n_2; \ldots; n_k \rangle$ is interaction-free if none of $n_2; \ldots; n_{k-1}$ represent system-interaction events.
Of the examples of event-flow paths presented above, \( \langle \text{File}; \text{Edit}; \text{Undo} \rangle \) are interaction-free (since \text{Edit} is not a system-interaction event) whereas \( \langle \text{MatchCase}; \text{Editbox1}; \text{Replace} \rangle \) is not (since \text{Editbox1} is a system-interaction event). Intuitively, two system-interaction events may interact if a GUI user may execute them in an event sequence without executing any other intermediate system-interaction event.

**Definition:** A system-interaction event \( e_x \) interacts-with system-interaction event \( e_y \) iff there is at least one interaction-free event-flow-path from the node \( n_x \) (that represents \( e_x \)) to the node \( n_y \) (that represents \( e_y \)).

For the EFG of Figure 2.4, the above relation holds for the following pairs of system-interaction events:

\[ \{(\text{New}; \text{Date/Time}), (\text{FindNext}_1; \text{WordWrap}), (\text{Editbox}_0; \text{Editbox}_1), \text{and} (\text{Delete}; \text{Cancel})\} : \]

The interaction-free event-flow-paths for these pairs are

\[ (\text{New}; \text{Edit}; \text{Date/Time}), (\text{FindNext}_1; \text{Format}; \text{WordWrap}), (\text{Editbox}_0; \text{Editbox}_1 >; \text{and} < \text{Delete}; \text{Cancel}), \]

respectively. Note that an event may interact-with itself. For example, the event \text{MatchCase} interacts with itself. Also note that “\( e_x \) interacts-with \( e_y \)” does not necessarily imply that “\( e_y \) interacts-with \( e_x \)” In the EFG example, even though \text{Replace} interacts-with \text{Cancel}, the event \text{Cancel} does not interact-with \text{Replace}.

The interacts-with relationship is used to create the event-interaction graph\(^\dagger\). This graph contains nodes, one for each system-interaction event in the GUI. An edge from node \( n_x \) (that represents \( e_x \)) to node \( n_y \) (that represents \( e_y \)) means that \( e_x \) interacts-with \( e_y \).

\(^\dagger\)For further reference about EIG please read the document of Atif M. Memon
Chapter 3

Problem Statement

Many consider GUI testing is an integral part of software development process. The most common practice is to use manual GUI testing framework such as JfcUnit, Abbot, Pounder etc to test the user interface by simulating user interaction with the application. For example, a login dialog/form is a very common interface for different type of applications. A typical login screen contains two edit boxes for user name and password, one “Ok” button for forwarding the input data for validation and a “Cancel” button for terminating the operation. A complete test case scenario for this login process should be:

1. Creating the login form.
2. Writing test cases to validate two edit boxes and two buttons whether these components are available or not.
3. Checking different attributes of the components (such as component position, button captions etc).
4. Testing the “Ok” button functionality.
5. Testing the “Cancel” button functionality and
6. Destroying the login form.

It is assumed that when the mentioned test cases are executed, the underlying codes are also executed. The most important point of executing such kind of test cases is that it ensures the underlying event implementation of the components. But is it possible from the test case itself to verify whether the underlying event is producing the desired results? Suppose, when the “Ok” button is clicked the underlying event might do nothing as the event may contain no codes or logically invalid codes. How will the test case react in this case? Is it going to pass the test case- as it fulfills all the requirements by firing the underlying event? If this test case becomes successful then it raises some vital questions. How to check the event if it is working as required? Does the programmer need to write a new set of test cases using a dedicated framework (such as JUnit) to test the underlying code?

The research paper tries to investigate such problems through empirical study as no previous study has been undertaken to examine such kind of scenario. In the project emphasis has been given to answer the following correlated questions:

1. How much coverage is possible using JfcUnit test cases?
2. What parts of the code cannot be executed by these test cases?
3. What are the fault detection abilities of these GUI test cases?
4. What are the characteristics of faults which are undetectable?
5. A comparison of fault-detection ability of JfcUnit test cases with “Event Sequence Graph” generated test cases.
6. What are the weaknesses of these test cases?
Chapter 4

Empirical Study

4.1 Performed Tasks

During the project the following steps have been taken to study the applications under test:

1. Considering GUI oriented application under test with less business logic.
2. Writing test cases.
3. Generating test reports.
4. Identifying methods those are not covered by the test cases.
5. Rewriting the test cases to improve coverage.
6. Performing test operation for 100% method coverage.
7. Generating test reports.
8. Determining the methods those are not possible to cover using test cases.
9. Seeding faults to create mutants.
10. Running the test cases again to check the fault detection effectiveness.
11. Comparing the results with the fault detection effectiveness results generated by the test cases using "Event Sequence Graph". *
12. Identifying the robustness of the test cases.

*In this paper "Event Sequence Graph" refers both "Event Flow Graph" and "Event Interaction Graph"
4.2 Application Under Test

The subject applications for the studies are part of an open source office suite developed in the Department of Computer Science at the University of Maryland. It is called TerpOffice and includes TerpWord (a small word-processor), TerpSpreadSheet (a spreadsheet application), TerpPaint (an image editing/manipulation program), and TerpPresent (a presentation tool). They have been implemented using Java. Table 4.1 summarizes the characteristics of these applications.

<table>
<thead>
<tr>
<th>Application</th>
<th>Windows</th>
<th>Widgets</th>
<th>LOC</th>
<th>Classes</th>
<th>Methods</th>
<th>Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>TerpPaint</td>
<td>10</td>
<td>200</td>
<td>18376</td>
<td>219</td>
<td>644</td>
<td>1277</td>
</tr>
<tr>
<td>TerpSpreadSheet</td>
<td>9</td>
<td>145</td>
<td>12791</td>
<td>125</td>
<td>579</td>
<td>1521</td>
</tr>
<tr>
<td>TerpWord</td>
<td>11</td>
<td>112</td>
<td>4893</td>
<td>104</td>
<td>236</td>
<td>452</td>
</tr>
<tr>
<td>TerpPresent</td>
<td>12</td>
<td>294</td>
<td>44591</td>
<td>230</td>
<td>1644</td>
<td>3099</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>42</td>
<td>751</td>
<td>80651</td>
<td>678</td>
<td>3103</td>
<td>6378</td>
</tr>
</tbody>
</table>

Table 4.1: TerpOffice Applications.

These applications are fairly large with complex GUIs. All the applications are roughly the size of MS WordPad. The number of widgets listed in the table are the ones on which system-interaction events can be executed (i.e., text-labels are not included). The LOC are the number of statements in the programs. The Help menu is also not included since the help application is launched in a separate Web browser. Most of the code written for the implementation of each application is for the GUI. None of the applications have complex underlying "business" logic.

4.3 Writing Test Cases

Each application was considered individually for writing a complete set of test cases. For writing test case JfcUnit (an extension of JUnit) was chosen. JfcUnit was chosen as it was a very popular and up to date framework for testing graphical user interface in its kind. The test cases were written in Eclipse; an integrated development environment to minimize the development time. Emphasis was given to write every possible test case to cover every component visible in the application. Then the priority was given to cover all the underlying events implemented by a single component. It means if a component implemented multiple events then the test cases were written in such a way so that it could tests both the test cases. (consider a drop down combo box which had both onValueChnged and onClicked events). A different test case was written for the fact where it is required to write the test case in another module depending on the nature of the component (suppose a modal dialog that requires an individual thread to test).

According to the test driven development process(TDD) it was required to write the test first before implementation to make sure that not a single line of code was written without test (a brief explanation is given in the background study section). This project did not follow the rule as the project goal was not to develop an application through testing but to test the testing framework itself. Multiple test cases were written together (according to the suggestion every test case should contain a single test) with redundant codes (it means refactoring was not applied). All of these standards are compromised while writing test cases because this violation of standards would not affect the project objective.
CHAPTER 4. EMPIRICAL STUDY

4.4 How much coverage is possible using JfcUnit test cases?

Table 4.2 shows the percentage of covered code for TerpOffice using JfcUnit.

<table>
<thead>
<tr>
<th>Subject Application</th>
<th>Total Coverage</th>
<th>Method Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TerpPaint</td>
<td>85.6%</td>
<td>84.9%</td>
</tr>
<tr>
<td>TerpSpreadSheet</td>
<td>68.3%</td>
<td>70.1%</td>
</tr>
<tr>
<td>TerpWord</td>
<td>82.7%</td>
<td>84.6%</td>
</tr>
<tr>
<td>TerpPresent</td>
<td>52.7%</td>
<td>58.1%</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of Code Coverage of Different Applications in TerpOffice.

The following section briefly explains the empirical result for each application.

4.4.1 TerpPaint

TerpPaint is an application similar to Microsoft Paint. It has a canvas for drawing and some dialogs to change attributes (e.g., flip/rotate, stretch/skew etc.). Using JfcUnit most of the methods (84.9%) can be reached while testing GUI. Bar graph 4.1 shows the code coverage report generated by "Clover" [17] for the application. The original Clover generated report is modified by excluding the test cases† so that the report can be clearly viewed. The horizontal line shows the name of the java files and the vertical lines depict their coverage in percentage.

While investigating the cause behind the rest 14.9% methods which are not traversed by the test case, a number of reasons discovered. Table 4.3 includes the possible reasons for each java file which did not have 100% code coverage. In the table the first column represents the java file name, second column represents the number of total methods for each java file and the number of visited methods by JfcUnit (i.e., 4/5 means 4 methods were visited by JfcUnit where the total number of method is 5), % column denotes the coverage in percentage and the Reason column explains reasons behind inadequate coverage. In the table 4.3 'N/A' means it is not possible to cover all methods from GUI testing.

†excluding test cases coverage result is logical since they are not an integral part of the application implementation
**Figure 4.1: Coverage Result of TerpPaint.**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Method</th>
<th>%</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>printer</td>
<td>0/2</td>
<td>0.00%</td>
<td>cross platform dialog, not tested by jfcUnit</td>
</tr>
<tr>
<td>ScrollablePicture</td>
<td>3/7</td>
<td>42.90%</td>
<td>4 methods not used by the application</td>
</tr>
<tr>
<td>FaintContainer</td>
<td>2/4</td>
<td>50.00%</td>
<td>N/A</td>
</tr>
<tr>
<td>selectTool</td>
<td>11/12</td>
<td>91.70%</td>
<td>N/A</td>
</tr>
<tr>
<td>bucketTool</td>
<td>4/5</td>
<td>80.00%</td>
<td>1 method never used by the application</td>
</tr>
<tr>
<td>splash</td>
<td>7/9</td>
<td>77.80%</td>
<td>main and run method never called by the GUI</td>
</tr>
<tr>
<td>ClipboardObject</td>
<td>4/8</td>
<td>50.00%</td>
<td>4 methods never used by the application</td>
</tr>
<tr>
<td>main_canvas</td>
<td>18/24</td>
<td>75.00%</td>
<td>main and 4 methods never called by the GUI</td>
</tr>
<tr>
<td>FaintFileFilter</td>
<td>9/12</td>
<td>75.00%</td>
<td>2 constructor and 1 method never used</td>
</tr>
<tr>
<td>letterTool</td>
<td>2/4</td>
<td>50.00%</td>
<td>2 inherited abstract method and 2 function never called</td>
</tr>
<tr>
<td>Faint</td>
<td>179/208</td>
<td>86.10%</td>
<td>29 methods not covered by GUI testing</td>
</tr>
<tr>
<td>attributes</td>
<td>14/15</td>
<td>93.30%</td>
<td>main</td>
</tr>
<tr>
<td>saveChanges</td>
<td>9/13</td>
<td>69.00%</td>
<td>main, 3 function not used by GUI</td>
</tr>
<tr>
<td>stretch</td>
<td>8/17</td>
<td>47.10%</td>
<td>main, 4 events and 4 events call never used</td>
</tr>
<tr>
<td>mail</td>
<td>8/11</td>
<td>72.70%</td>
<td>main, one function and one windows listener</td>
</tr>
<tr>
<td>eraserTool</td>
<td>5/6</td>
<td>83.30%</td>
<td>one function never called</td>
</tr>
<tr>
<td>SlideShow</td>
<td>10/12</td>
<td>83.30%</td>
<td>main and one function never called</td>
</tr>
<tr>
<td>viewBitmap</td>
<td>7/8</td>
<td>87.50%</td>
<td>N/A</td>
</tr>
<tr>
<td>rotate</td>
<td>15/16</td>
<td>93.80%</td>
<td>main method not tested</td>
</tr>
</tbody>
</table>

**Table 4.3: Terp Paint Not Covered.**
4.4.2 TerpWord

TerpWord is a relatively small application. Empirical study shows jfcUnit can cover 84.6% methods of the application. Graph 4.2 shows the application code coverage and the table 4.4 explain the reasons for the file which does not has 100% method coverage.

![Graph 4.2: Coverage Result of TerpWord.](image)

<table>
<thead>
<tr>
<th>File Name</th>
<th>Method</th>
<th>%</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwoContentArea.PrintView</td>
<td>-</td>
<td>0.00%</td>
<td>Cross platform</td>
</tr>
<tr>
<td>TwoStyledDocument</td>
<td>(1/4)</td>
<td>25.00%</td>
<td>3 constructor, 1 method not called</td>
</tr>
<tr>
<td>TwoContentArea.ImageView</td>
<td>-</td>
<td>0.00%</td>
<td>Not Used</td>
</tr>
<tr>
<td>TwoContentArea.Notepad..</td>
<td>(5/10)</td>
<td>50.00%</td>
<td>-</td>
</tr>
<tr>
<td>TwoContentArea</td>
<td>(41/49)</td>
<td>83.70%</td>
<td>4 inside if statement, 1 throw, print, paste</td>
</tr>
<tr>
<td>TwoPad</td>
<td>(3/5)</td>
<td>60.00%</td>
<td>main and method alive never used</td>
</tr>
<tr>
<td>OpenList</td>
<td>(10/12)</td>
<td>83.30%</td>
<td>2 methods never used</td>
</tr>
<tr>
<td>TwoMenus</td>
<td>(39/47)</td>
<td>83.00%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4.4: TerpWord Not Covered.

Note that, both in TerpPaint and TerpWord there are some unused files and interface. There are also some set and get methods which were implemented but never used.
4.4.3 TerpSpreadSheet

TerpSheet is a small version of Microsoft Excel with limited functionalities. It supports sum, average, max, min, sine, count etc functionalities. Graph 4.3 shows the application code coverage and table 4.5 explain the reasons for the file which does not has 100% method coverage. One significant difference of TerpSpreadSheet implementation comparing with TerpPaint and TerpWord is it uses custom built component. As this custom component is used to implement the spread sheet it has direct impact on all the functionalities(sum,max,min) provided by the application. During testing the application it is found that jfcUnit is not capable of testing custom built component. The following graph evident the fact.

Figure 4.3: Coverage Result of TerpSpreadSheet.
<table>
<thead>
<tr>
<th>File Name</th>
<th>Method</th>
<th>%</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>(0/5)</td>
<td>0.00%</td>
<td>Used only by Formula class to evaluate functions.</td>
</tr>
<tr>
<td>FunctionAverage</td>
<td>(0/3)</td>
<td>0.00%</td>
<td>Not tested due to custom component</td>
</tr>
<tr>
<td>FunctionCount</td>
<td>(0/3)</td>
<td>0.00%</td>
<td>Not tested due to custom component</td>
</tr>
<tr>
<td>FunctionMax</td>
<td>(0/3)</td>
<td>0.00%</td>
<td>Not tested due to custom component</td>
</tr>
<tr>
<td>FunctionMin</td>
<td>(0/3)</td>
<td>0.00%</td>
<td>Not tested due to custom component</td>
</tr>
<tr>
<td>FunctionSin</td>
<td>(0/3)</td>
<td>0.00%</td>
<td>Not tested due to custom component</td>
</tr>
<tr>
<td>FunctionSqrt</td>
<td>(0/3)</td>
<td>0.00%</td>
<td>Not tested due to custom component</td>
</tr>
<tr>
<td>FunctionSum</td>
<td>(0/3)</td>
<td>0.00%</td>
<td>Not tested due to custom component</td>
</tr>
<tr>
<td>Config</td>
<td>(1/10)</td>
<td>10.00%</td>
<td>External file for tool bar configuration, file not used</td>
</tr>
<tr>
<td>NumberField</td>
<td>(1/8)</td>
<td>12.50%</td>
<td>Related with functions</td>
</tr>
<tr>
<td>ParserException</td>
<td>(1/5)</td>
<td>20.00%</td>
<td>Exception is raised when Formula fails tokenizing</td>
</tr>
<tr>
<td>UMClipObj</td>
<td>(3/9)</td>
<td>33.30%</td>
<td>N/A</td>
</tr>
<tr>
<td>Formula</td>
<td>(17/32)</td>
<td>53.10%</td>
<td>Class for formula processing</td>
</tr>
<tr>
<td>Node</td>
<td>(24/34)</td>
<td>70.60%</td>
<td>N/A</td>
</tr>
<tr>
<td>UMCCellEditor</td>
<td>(3/6)</td>
<td>50.00%</td>
<td>N/A</td>
</tr>
<tr>
<td>CellPoint</td>
<td>(4/10)</td>
<td>40.00%</td>
<td>N/A</td>
</tr>
<tr>
<td>UMOptionPane</td>
<td>(5/9)</td>
<td>55.60%</td>
<td>N/A</td>
</tr>
<tr>
<td>UMCell</td>
<td>(21/29)</td>
<td>72.40%</td>
<td>N/A</td>
</tr>
<tr>
<td>UMTTableModel</td>
<td>(32/43)</td>
<td>74.40%</td>
<td>N/A</td>
</tr>
<tr>
<td>LinesBorder</td>
<td>(12/16)</td>
<td>75.00%</td>
<td>N/A</td>
</tr>
<tr>
<td>UMDialog</td>
<td>(15/27)</td>
<td>55.60%</td>
<td>Provide customize dialog using ShartTools code.</td>
</tr>
<tr>
<td>FileMenuOp</td>
<td>(14/17)</td>
<td>82.40%</td>
<td>Some get methods are never used</td>
</tr>
<tr>
<td>FormatMenuOp</td>
<td>(37/39)</td>
<td>94.90%</td>
<td>UpdateTableModel is never called</td>
</tr>
<tr>
<td>CellRange</td>
<td>(8/14)</td>
<td>57.10%</td>
<td>Constructors and a set of get function never called</td>
</tr>
<tr>
<td>UMSpreadSheet</td>
<td>(49/65)</td>
<td>72.30%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4.5: TerpSpreadSheet Not Covered.

Coverage of first eight rows of table 4.5 are absolutely 0% due to the custom component. Observation shows that while simulating user interaction the underlying event did not fire. Such kind of behavior is not unexpected from the framework as it requires the component type to fire the appropriate event. In case of custom component the type of component is unknown to the framework for further processing. “N/A” means it is not possible to cover more methods using GUI test case.
4.4.4 TerpPresent

TerpPresent is an application similar to JGraphPad [19] or Microsoft Visio. It actually uses the JGraphPad application framework for development. The following bar graph shows the coverage of the application. The graph does not include JGraphPad files due to their large number of files.

Figure 4.4: Coverage Result of TerpSpreadSheet.

Though the original coverage for the application shown in table 4.2 in 58.1% but without JGraphPad’s not used files it shows the application has 90.96% coverage. With JGraphPad framework the application has 267 classes where more than 180 classes were implemented from JGraphPad version 1.0. Due to the use of custom component in JGraphPad, jfcUnit again experienced problem while testing the application.
4.5 What Parts of The code Cannot be Executed by These Test Cases?

The main reason that causes "TerpSpreadSheet" poor coverage is custom component. In TerpSpreadSheet application, custom component is used to implement the main grid cell where user may perform different operation (such as sum, average, multiplication etc). JfcUnit is not capable enough to test custom components as its implementation is basically for java swing based components. As a result all the operation under menu "Functions" became untested. Particularly in this case jfcUnit does not generate any exception rather pass the test case. But the instrumented code shows the section of code is not visited. As the custom component is related with more operations (copy, paste using right click) the overall method coverage remained poor.

There is no absolute way to test main() method using jfcUnit. So every dialog may have at least one method which is not tested and even more if some inline method is implemented inside main function⁷. Then call the new method to test the functionality of the main method. In this way it is possible to test the functionality of the main method, but the main method itself remain untested and the files becomes bulky with code that is never used.

JfcUnit can not test cases when cross platform method is called. Such kind of methods are common under "Page Setup", "Print Preview", "Print" menu options. As a result it is possible to test the event related with menu button only; but not the implemented codes. Corresponding cross platform methods call related with "Print Preview" or "Print" are not covered by the framework. These menu options are fairly common for all kind of applications.

Testing pop up sub menu for mouse right button click is not yet implemented in JfcUnit. So there is no way one can test sub menu when click the right mouse button. This is also true for java swing based "JSlider" component. This cause "TerpPresent" brightness option remains untested.

---

⁷ Suggestion has been given to implement a method same as the main with a different name. Then test the second method to check the functionality of the main method. But no concrete document has been found to verify this strategy.
4.6 What is The Fault Detection Ability of These GUI Test Cases?

jfcUnit enables developers to write test cases for Java Swing based applications which provide support for:

- Obtaining handles on Windows/Dialogs opened by the Java code.
- Locating components within a component hierarchy that occur within the containers found above.
- Raising events on the found components, e.g., clicking a button, typing text in a text component.
- Handling testing of components in a thread safe manner.

As jfcUnit is an extension of JUnit and dedicated for testing GUI, its fault detection ability is divided into two sections for the benefit explanations. First, GUI related fault-detection ability of jfcUnit. Second, underlying code related fault-detection ability.

4.6.1 GUI fault-detection ability

After generating test cases a set of GUI related faults has been chosen assuming that these faults are the common mistake usually done by the developer in terms of interface design. The seeded faults are:

1. Modify GUI by deleting components.
2. Change component caption.
3. Enable/disable one or more components.
4. Changes component visibility status.

Usually jfcUnit locates a component either by using its name or its type (button, text field etc) and the component must be visible during testing. The framework treats the visibility disabled components same way as deleted components. So jfcUnit will generate same error message for faults type one and four. For this reason fault type four related data is not included in the following section. After seeding faults the test case has been tested again to check the fault-detection ability. The following table 4.6 lists the number of seeded faults for different application:
For each kind of faults jfcUnit perform very well to report specific error. The following column graphs state the empirical result found for each application:

![Graphs showing the number of faults detected by jfcUnit for different applications](image)

Table 4.6: Fault Seeded in TerpOffice.

<table>
<thead>
<tr>
<th>Subject Application</th>
<th>Fault Type 1</th>
<th>Fault Type 2</th>
<th>Fault Type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TerpPaint</td>
<td>20</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>TerpSpreadSheet</td>
<td>11</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>TerpWord</td>
<td>25</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>TerpPresent</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Empirical result shows that jfcUnit is very much efficient of detecting bugs related with graphical user interface. When investigate the reason behind some undetected faults it is found that most of the undetected faults were related with application menu. For an example; in TerpWord when 'Open' menu button is pressed then a JChosser dialog appears. If the "Open" menu
button is disabled then the test case fails to locate JFileChooser and the test case fails. On the other hand disable "Insert Time/Date" will pass the test case as the corresponding test case only fire the "Onclick" mouse event and do nothing else as it does not has any other statement to verify with any other dialog as "Open" menu has. Figure 4.6 shows the overall faults detected by jfcUnit according to faults category.

### 4.6.2 Code related fault-detection ability

While seeding faults for testing code related fault-detection ability, care was taken to make sure test case traverse the faulty codes. To ensure the criterion, first the underlying methods were identified by running a test case. Then one fault seeded at a time and executed the test case again to see the effect. Same types of faults were not implemented in same circumstance. A set of realistic faults were chosen based on history-based approach during TerpOffice development. The following fault classes were chosen for the study:

1. Modify relational operator ($>, <, \geq, \leq, \neq$);
2. Invert the condition statement;
3. Modify arithmetic operator ($+, -, *, /, +=, -=, *=, /=, ++, --, +=, -=, * =, /=$);
4. Modify logical operator;
5. Set/return different Boolean value (true, false);
6. Invoke different (syntactically similar) method;
7. Set/return different attributes.
8. Modify bit operator;
9. Set/return different variable name;
10. Set/return different integer value;
11. Exchange two parameters in a method; and
12. Set/return different string value.

The following table 4.7 shows the number of faults seeded in each application and table 4.8 shows the empirical result obtained from the study.

<table>
<thead>
<tr>
<th>Subject Application</th>
<th>Number of Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>TerpPaint</td>
<td>60</td>
</tr>
<tr>
<td>TerpSpreadSheet</td>
<td>55</td>
</tr>
<tr>
<td>TerpWord</td>
<td>45</td>
</tr>
<tr>
<td>TerpPresent</td>
<td>62</td>
</tr>
</tbody>
</table>

Table 4.7: Fault Seeded in TerpOffice.

<table>
<thead>
<tr>
<th>Faults Type</th>
<th>Pass</th>
<th>Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>04</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>02</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>02</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>08</td>
<td>02</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>34</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>09</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>03</td>
</tr>
<tr>
<td>Total:</td>
<td>33</td>
<td>188</td>
</tr>
</tbody>
</table>

Table 4.8: Fault Seeded in TerpOffice by Category.

Four column graphs are also included for clear understandings. Remember all four graphs have different scale due to different number of seeded faults in each application. The fifth one is the overall presentation of detected faults according to faults category.

From the graph it is clear that the fault-detection ability of jfcUnit is very poor for code related faults. An explanation is included later for this poor performance.
Figure 4.7: Code Based Fault-Detection Ability of jfcUnit.
4.7 A Comparison of Fault-detection Ability of JfcUnit Test Cases With “Event Sequence Graph” Generated Test Cases

For comparing the fault-detection ability of both type of test cases, event sequence graph\(^5\) generated data have been collected from the paper “Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software” by Atif Memon and Qing Xie, TSE 2005. In this paper writers automated test case generation process through their newly built tool, seeded realistic faults (faults used in previous section for studying code related fault-detection ability of jfcUnit) and used an oracle to study the fault-detection ability of these test cases. Before presenting the empirical result (obtained by the writers) a brief relative discussion is presented below from the paper.

To automate the process of generating test cases writers invented DART (Daily Automated Regression Tester) for the empirical study. DART’s test-case generator used the event-interaction graphs to automatically generate 5,000-11,000 smoke test cases for each application. Writers used an automated test-oracle generator to obtain the oracle information which automatically executes test cases to obtain application all level states (widgets, properties, and values). 200 faults were seeded in each subject application, thereby creating 200 faulty versions for each application. Exactly one fault was seeded in each version to avoid fault-interaction. To determine the number of test case failures, the number of mismatches between the executing GUI’s state and the oracle information were counted. The original column graphs of number of faults detected and classes of faults detected are shown below. Remember, graph 4.8 contains application TerpCalc which is not an application under test for this project and fault number seven (set/return attributes) is not considered while comparing with jfcUnit generated test cases in this section as this type of faults has already been discussed in the previous section.\(^\*$

\(^{5}\)Here event sequence graph denotes both event flow graph and event interaction graph.

\(^{\*}\)The original paper only contains graphs but not enough data to redraw the graphs. That is why the original graphs are presented to maintain data accuracy.
Empirical result for jfcUnit generated test cases is given below.

Please notice that the empirical result of jfcUnit generated test case is presented using different scale. As the fault-detection ability of manually generated test cases very poor comparing with automatically generated test cases; making same scale causes some result unclear. Moreover there is a huge difference between the numbers of faults seeded in both type of test cases. The reason behind fewer faults seeded in jfcUnit generated test cases is; fault must be seeded in test execution path of the test case. Otherwise it is impossible for the test case to detect the changes (as the changed code is not traversed by the test case).
4.7.1 Reason behind poor coverage

jfcUnit uses a set of assertion methods for writing tests. Only failed assertions are recorded and there are some limited set of such kind of methods. A short description is given below:

- `assertArrayEquals`: Asserts that two object arrays are equal.
- `assertEquals`: Asserts that two doubles or floats are equal to within a positive delta.
- `assertFalse`: Asserts that a condition is false.
- `assertNotNull`: Asserts that an object isn’t null.
- `assertNull`: Asserts that an object is null.
- `assertNotSame`: Asserts that two objects do not refer to the same object.
- `assertSame`: Asserts that two objects refer to the same object.
- `assertTrue`: Asserts that a condition is true.
- `fail`: Fails a test with no message or with message.

While performing test jfcUnit simulate user action and call the underlying event when a component is clicked. In this process jfcUnit first identify the GUI component and call the corresponding event. During the execution of the event jfcUnit has no control on the underlying event and has no ability to check or compare the code implemented in the event procedure. It means any kind of change inside the event procedure is completely unknown to the test case. Sometimes changing inside the event procedure may fire new event or restrict the event from firing another event. This may invalidate one or more statements of the test case by creating new dialog or not creating expected dialog. Such kind of mismatch causes the test case failure and reports the bugs. To clear the fact an example is given below.

In TerpPaint a file chooser dialog appear when menu button "Open" is clicked. User can open a file by writing the file name and press "Open" button from the file chooser. For this scenario the test case will be:

```java
public void testFileMenuOpen()
{
    1. makeAndShow();
    2. PathData path = new PathData(new String[] {"File", "Open"});
    3. JMenuBar bar = (JMenuBar) path.getRoot(myMenu);
    4. getHelper().enterClickAndLeave(new JMenuMouseEventData(this, bar, path.getIndexes(bar), 1, 0, false, 0));
    5. dialogs = new DialogFinder("Open").findAll();
    6. assertEquals("Dialog not found!", 1, dialogs.size());
    7. myContainer = (Container) dialogs.get(0);
    8. myFinder = new ComponentFinder(JTextField.class);
    9. txtfield = (JTextField) myFinder.find(myContainer, 0);
    10. assertNotNull("Could not find field", txtfield);
    11. getHelper().sendString(new StringEventData(this, tfield, "Zapotec.bmp"));
    12. myFinder = new ComponentFinder(JButton.class);
    13. JButton button = (JButton) myFinder.find(myContainer, 7);
    14. assertNotNull("Could not find button", button);
    15. getHelper().enterClickAndLeave(new MouseEventData(this, button));
}
```
In the above mentioned code line number four will fire underlying event “OpenActionPerformed” and file chooser dialog will be opened. Line five will locate the dialog and report failure through line number six if file chooser dialog is not found. During this test case execution any kind of change inside ”OpenActionPerformed” that does not affect the creation of the file chooser will not be reported by the test case. On the other hand deleting (or blocking through if statement) the file chooser dialog will fail the assertion on line number six and report test case failure. In this way any seeded faults that do not interrupt the execution sequence of the test case through changing the element of graphical user interface is not possible to detect by the framework.
4.8 What are the weaknesses of these test cases?

For testing a dialog and its components jfcUnit first identify the dialog. During this procedure it requires a caption for the dialog. A dialog with no caption is not always possible to capture. Moreover not all characters are valid for caption. While testing TerpPaint jfcUnit failed to capture dialog due to the use of characters '(' and ')'. So changing the dialog caption eventually invalidate the whole test case and requires change in test case also. To avoid such kind of problems an effort was made to capture dialog by name using some deprecated functions but the result was negative.

For testing a dialog and its components jfcUnit first identify the dialog. During this procedure it requires a caption for the dialog. A dialog with no caption is not always possible to capture. Moreover not all characters are valid for caption. While testing TerpPaint jfcUnit failed to capture dialog due to the use of characters '(' and ')'. So changing the dialog caption eventually invalidate the whole test case and requires change in test case also. To avoid such kind of problems an effort was made to capture dialog by name using some deprecated functions but the result was negative.

Sometimes the framework fails to identify the difference between button and combo box during test case execution. Sometimes it is experienced the failure of a test case (which was successfully tested previously) due to the reason. In that case it requires changing the sequence number of the button. In line number 13 code segments presented on page 34; the button sequence number is changed to 7 (which was previously 5).
4.9 Threats to Validity

Threats to external validity are conditions that limit the ability to generalize the results of experiments to industrial practice. Several such threats are identified in this study. First, four GUI-based Java applications have been used as subject programs. Although they have different types of GUIs, this does not reflect a wide spectrum of possible GUIs that are available today. Moreover, the applications are extremely GUI-intensive, i.e., most of the code is written for the GUI. The results will be different for applications that have complex underlying business logic and a fairly simple GUI. Second, all the subject programs were developed in Java by students, which might be more bug-prone than software implemented by professionals. Finally, although the abstraction of the GUI maintains uniformity between Java and Win32 applications, the results may vary for Win32 applications.
Chapter 5

Future Work

This project implemented only jfcUnit test cases to compare fault detection ability of graphical user interface. A mixture of JUnit test cases along with jfcUnit will increase the fault detection ability of test cases.

All four application tested in this project are graphical user interface oriented with less business logic. During this empirical study no measure has been taken to test any other part rather than graphical user interface. So an application with more business logic can be a subject of experiment to test the affect.

It would be a very good idea to practice "Test Driven Development" for a single project using JUnit and jfcUnit. In that case it is assumable that the coverage ability will be increased along with fault detection ability.

Coverage ability can be increased by writing testing framework for components which are not yet supported by the frame work. As there is no absolute way to test custom component in jfcUnit, it would be a niece work to write an extension for supporting and testing custom component.

In the testing phase it is observed that it requires writing test case with same kind of behavior. Developing a semi-automatic system would be a good solution to deal with such kind of testing.

jfcUnit is written for java swing based application. There is no AWT support in the jfcUnit frame work. In this section a lots of work is yet to be done.
Chapter 6

Conclusion

The project tried to answer multiple questions related with graphical user interface testing. By writing a set of test cases the project studied method coverage ability of a manual GUI testing framework- jfcUnit. Then the project seeded different types of realistic faults to observe the fault detection ability of test cases. After the successful completion of the first part, the project compared the fault detection ability of a manual testing framework with an automated testing framework. During the whole research study, effort has been made to answer the following questions:

1. How much coverage is possible using “JfcUnit” test cases?
2. What parts of the code cannot be executed by these test cases?
3. What are the fault detection abilities of these GUI test cases?
4. What are the characteristics of faults which are undetectable?
5. A comparison of fault-detection ability of JfcUnit test cases with "Event Sequence Graph" generated test cases.
6. What are the weaknesses of these test cases?

During the empirical investigation- strength, weakness and incompleteness of the framework had been identified. The project compared manual testing framework, JfcUnit, with automated testing framework using EFG and EIG. It has never been the project’s intention to class the frameworks according to their order of superiority or to compare them in terms of their functional superiority. Now it is up to the framework developers to strengthen their framework by minimizing the weaknesses identified in this project.
Bibliography


[cited at p. 20]
Appendices
Appendix A

A.1 Clover

Clover is a powerful and highly configurable code coverage analysis tool. It discovers sections of code that are not being adequately exercised by your unit tests. Developers and team leads use Clover to quickly find untested java code and measure testing completeness. This feeds back into the testing process to improve tests. Coverage reports can be published in HTML or PDF and shared with the development team or project management.

A.2 Code Coverage

Code coverage measurement simply determines those statements in a body of code have been executed through a test run and those which have not. In general, a code coverage system collects information about the running program and then combines that with source information to generate a report on test suite’s code coverage.

Code coverage is part of a feedback loop in the development process. As tests are developed, code coverage highlights aspects of the code which may not be adequately tested and which require additional testing. This loop will continue until coverage meets some specified target.

A.3 Widget

A generic term for the part of a GUI that allows the user to interface with the application and operating system. Widgets display information and invite the user to act in a number of ways. Typical widgets include buttons, dialog boxes, pop-up windows, pull-down menus, icons, scroll bars, resizable window edges, progress indicators, selection boxes, windows, tear-off menus, menu bars, toggle switches and forms.

A.4 Component

A reusable piece of software in binary form that can be easily integrated with other components with relatively little effort.
A.5 Method Coverage

Method coverage measures if a method was entered at all during execution of coverage measurement.

A.6 Test Case

TestCase is a mechanism to allow fixture reuse. Each Test Case subclass represents a fixture, and contains a group of tests that run in the context of that fixture. A fixture is the set of preconditions and assumptions with which a test is run.

A.7 JUnit

JUnit is a simple, open source framework to write and run repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks. JUnit features include:

- Assertions for testing expected results
- Test fixtures for sharing common test data
- Test runners for running tests
Appendix B

Appendix

B.1 Fault Matrix
Appendix C

Appendix

C.1 Application Source Code