Record and Playback
For Java Software Watermarking

Project Report for COMPSCI780AC
by

Guanglun Yu
Dept of Computer Science
The University of Auckland
gyu005@ec.auckland.ac.nz

Supervised by

Prof. Clark Thomborson
Dept of Computer Science
The University of Auakland
cthombor@cs.auckland.ac.nz

15 Feb 2001

Abstract

Record and playback tools are needed for performing simulations during software regression test, for recording users' operations and later playback during testing time or software run-time. Java Software Watermarking is a process of embedding marks into Java software and recognizing them later. It needs a tool to record and replay the users' operations during testing time and runtime. There are some articles out there, but fewer of them discussing the comprehensive issues about record and playback as a whole. Besides, Record and Playback has its specialties in Java Software Watermarking. This report tries to address the various concepts and effects in this area, discuss some new issues, especially for applying to Java Software Watermarking.

1. Introduction

Java software programming is currently widely spread for its platform independent feature, which makes the ease of decompiling Java byte code. *Java software watermarking* is used for preventing illegally copying and tampering Java programs. Watermarking to Java software is quite new, and is itself a challenge because of this ease. There is a model proposed by Dr. C.Collberg and Prof. C.Thomborson, which aims to provide *resilient* and *stealthy* watermarks[1][2] for Java software.

For the process of *Java software watermarking*, Prof. Thomborson asks a tool to record the inputs and then replay them, as assistance. This tool is also expected to be *generic* to all Java programs and will at best degree keep the Java program "doing once, run anywhere" feature.

Unfortunately, this mostly falls in the area of software testing, the one with more difficulties than software programming, especially for Java software[4]. Besides, Java's security mechanism will enhance this difficulty.

In this report, I will inspect the current *record/playback* methods, discuss the various aspects in this area, and analyze the specialty in applying to Java software watermarking. Some new issues, such as the *console input*, will be raised in this report. Some trial solutions will be given to these issues.

2. Basic Concepts and Definitions

★ 2.1 Java Software Watermarking:

Java Software Watermarking consists of two processes. The *embedding process* will embed a mark (*watermark*) into the *target application* (*target*), while the *retrieving process* will be able to retrieve that watermark from the *target*.

An implementation of such a Java software watermarking system is Sandmark[3], which embeds the watermark into a target and retrieve the watermark from the target. For embedding, the target is first annotated at those inside points on which the watermark structure nodes will be later created and reside. For retrieving, a state dump is made and from analyzing that dump, the watermark is recognized.

The user's input is critical for recognizing an embedded watermark. Only correct series of input, e.g. keystrokes, can retrieve an embedded watermark. During testing the watermarking or recognition of an embedded watermark, it is important to record the inputs to the target and replay them to see if any different results for same input series during watermarking process.

★ 2.2 Basics of Record and Playback (Record/Playback)

There are three major types of *record/playback* tools. The first one (*type 1*) offers the capability to automatically *record* and *playback* users' manual operations. The second one (*type 2*) allows users to use a *script language* to simulate the required user operations to the *target application*. The third type (*type 3*) is the combination of type 1 and type 2.

Most record/playback tools are used for software regression testing and are of the type 2. This type of record/playback tool, by using a scripting language, provides a "step-in" ability for users to programmatically simulate the required user operations. The whole process is like the familiar "debugging" process. Users can set up breakpoints in target applications and stop at these breakpoints or other places by setting stopping conditions. The difference is, by using scripting language, e.g. users can dynamically change the runtime values of variables. More over, it provides *record/playback* functionality (*automation*). Examples are ReplayJava in [6], JVerifier[8], Sun WorkShop Visual Replay[10], etc. A tool of *type 2* does not directly record users' operation. Instead, it records the scripting steps for later playback during the regression testing. The term *automated testing* often refers to the functionality available in this type.

Type 1 is designed for easy record/playback functionality without user's scripting. So it only records the users' operations and replays them. Run-time internal state is not accessible by this type. For technique reason, scripting language may be used internally, not available for users' use, to record the users' steps and program internal states for later playback. The term *input record/playback* specially refers to this type. Examples for this type are Rebecca v1.0[25], Panorama for Java: JavaPlayback [9], etc.

The tool of *type 3* not only provides the recording user operation ability, but also "step-in" ability. Fewer tools are in this type. It is most flexible and powerful. An example for this type is Mercury Interactive's WinRunner [22][23].

The focus in this report is not limited on *type 1*, but also type 2 and type 3 because the need for the *input record/playback* also to apply to the regression testing for *Java software watermarking*, and the desire for the *input record/playback* to be a generic.

So, in this report, the term *record/playback* will be used for general purpose, *input record/playback* for the specialty of recording the actual users input.

There are several major issues for *record/playback*. What input types will it deal with? For each input being captured, what is the relevant *context* (internal machine state) to be recorded as well? How to playback?

★ 2.3 Concept of User Interface Types of Target Application

A u*ser interface* provides the ability for users to interact with the target application. For this aspect, an application can be categorized into three types: the *graphic user interface* (GUI) application, *text user interface* (text UI, also called *console interface*) and *none user interface* (none UI) application.

Text UI was the only approach for users to interact with application before GUI technology was developed. Today, applications rarely provide text UI. They provide GUI instead.

Besides, there is a fact which can not be omitted, that in some cases, e.g. a server application probably does not have any interface, neither text UI or GUI, because it does not need users to interact with it.

There was/is a common understanding that testing GUI software is more difficult than testing *console interface* software [5]. As a rule, *automated testing* on *console interface software* can be readily accomplished by the assistance of I/O redirection. This lets the input from file instead of keyboard, output to file instead of screen. This ease is true for *type 2* of *record/playback* on *native applications*. But for *type 1* of *record/playback*, the I/O redirection simply does not work because the users' true input is needed.

Today, nearly all record/playback tools are for GUI applications [6], [9], [20], [21], [22], [24]. For Java GUI application, implementing such a record/playback tool seems to be easier than for console interface application instead. By using GUI's event mechanism, any true user input for *type 1 record/playback* and simulated input for *type 2 record/playback* can be captured through this event mechanism. An example for type 1 is Rebecca v1.0 [20], which provides *real-time recording schema* and record the users' keystrokes and mouse actions through *events* to GUI components. An example for *type 2* is ReplayJava [6], which uses a scripting language JTcl (JACL) [7] to perform *automated testing*, also through *events* to GUI components.

But for *console inputs* in Java applications, no available approaches to use such *event mechanism* for capturing these *console inputs* unless new solutions achieved. This will discussed later.

No one specially mentioned the issues for *none-UI* target applications. The *type 2 record/playback* tool may apply to this application, but only for input simulation during software testing. No *type 1 record/playback* tools could apply to this none-UI application because this application does not need and does not have user input operation at all. But for the watermarking process provided in [1][2], there is really a need to get users input, like a series of keystrokes to retrieve the embedded watermark. In this case, the record/playback tool would be integrated with a *watermark recognizer* to retrieve that embedded watermark. According to the *resilience* and *stealth* features required for watermarking [1][2], this watermarking process will not insert extra GUI components which are used as the receivers of the *inputs* which will activate

the *watermark recognition* and retrieve the embedded watermark. but some *console stream* variables as the *input receiver*; this watermarking process will insert other means to receive users inputs, such as a *console stream* variable. So for the none-UI target application, *console inputs* need to be considered under the same condition as *text UI application* (*console application*).

Further more, there is another problem with *console application* or *none-UI application*. An user operation input to a GUI component can happen and captured at any time during the execution of that GUI application, which means that for capturing a *GUI component input*, the application does not need to wait at special point until the input is finished. Unlike GUI components, for capturing console inputs, program must wait at special point until this console input is completed. So the problem is, for a console application or a none-UI application, how to activate the receiver in the program to capture the user inputs for watermarking recognition? This problem might fall into the area of watermarking recognition, but it is indeed one aspect of record/playback.

★ 2.4 Source Code Access Modes – Intrusive Mode, Non-Intrusive Mode and Invasive Mode

In this report, I assume that we hold the source code of the target application if necessary. For software testing, or the task to check if the watermark is embedded correctly and effectively before the vendor ships their product (target application), this assumption is ok.

Keeping the target source code clean and unchanged is ideal when applying *record/playback* tool. There are often the times that the source code needs to be changed at some degree to achieve some restrictive purpose. Otherwise, there is no way to do that.

At a normal sense, *intrusive mode* means that source code will be accessed and changed during the time of applying *record/playback* tool. *Non-intrusive* mode means that no source code needs to be changed.

There is another mode existing there, called *invasive mode*[8]. This mode currently is specially invented for Java application. Use that mode, private attributes of a Java class can be accessed without change the Java source code.

The first choice of cause is to apply Non-intrusive mode because any changes (even a very tiny change) will actually make the fact that tested version is not the same version as released. That would always be an implicit trend for producing bugs.

But there are often the cases that *Non-intrusive mode* doesn't help for achieving restricted attributes. At this moment we need apply *intrusive mode* to make some changes to target source code.

★ 2.5 Scripting Language

A scripting language works on the basis of function calls in a shell environment. The big difference from a programming language is that, after one function call finishes, the process control will return back to the scripting shell and the current program run-time context is stored in that shell environment.

This concept has two meanings. Scripting languages have two types. One is "*outside type*" and the other is *interactive type*. The *outside type* can only call the whole target application, e.g. a Java application, from outside the program environment. It has no ability to call individual internal method in the target. This type of script language is used normally to record a series of *black testing*. The *interactive type* provides the ability to access each attribute (variables or methods) and interact with them. By using this type of script, users or testers can interactively *step-in* Java program, stop at a specific point of the target application, assign variable values, etc.

To achieve interaction, the scripting language must have the ability to recognize Java *class, class method and class member* and the ability to internally control the running of Java program.

JTcl (JACL)[7] is such a scripting language which is even written in Java and fully portable. With the shell it provides, a record/playback tool can be implemented to be able to stop at any point of running target application, access the run time values of variables, invoke the method calls of classes, simulate the users' operations. An example of using JTcl(JACL) is J. Newmarch's package described in [14]. The benefit of using JTcl is that the implemented tool is totally compatible with the Java target application and is portable.

But most record/playback tools use their own specific scripting language, like JVerifier[8] which provides *invasive access mode*, etc.

3. Taxonomy of Input

To a Java application, it could be a *Keystroke input* to a component accompanied by a *KeyEvent*, a *mouse input* to a component accompanied by a *MouseEvent* or a *console input* to a variable.

Besides, there is a specialty in Java software watermarking. There are often situations that a Java server application, even with an embedded watermark, does not have any GUI interface. But it does have some inputs in the form of *InputStream* from socket created port.

Currently most Java applications are of GUI applications. Nearly all resources of articles and testing tools are dealing with *GUI component input*. For an old type *console input*, which is in the form of *InputStream*, no articles discuss how to *record/playback* it.

★ 3.1 What is an Input

I define that an *input* to a program is any data flow coming from outside of that program. So a Java application would have input like a *console input* from *console stream* (standard console), a GUI component key input to a GUI component accompanied by a KeyEvent or a GUI component mouse input to a GUI component accompanied by a MouseEvent, or a socket stream input in the form of an InputStream etc.

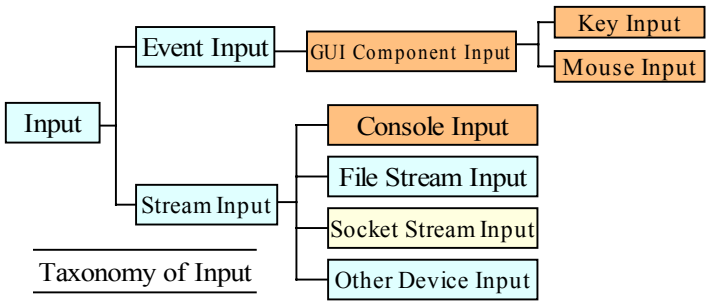

Figure 1.

★ 3.2 Taxonomy of Input

∇ I propose the taxonomy of input as the following, which is shown in Figure 1 diagram.

- An input is either an *event input* which is a input accompanied with a *event* and corresponds to one form of *InputEvent*, or a *stream input* in the form of *InputStream*.

- A *GUI component input* is an input to a GUI component accompanied by an *event* and is either a *key input* or a *mouse input*.
- A *console input* is a *stream input* which comes from standard console.
- A *socket stream input* is a *stream input* which comes from a socket created *InputStream*.
- A *file stream input* is a *stream input* which comes from file I/O stream
- An *other device input* is a stream input which comes from other device.

∇ Our interest

Currently, I will only discuss *GUI component input* and *console input*. Others go beyond the scope of this report.

∇ Explanations

- A *GUI component input* is either a *component key input* or a *component mouse input* both associates with a GUI component accompanied with an *event.*
- A *console input* has such a feature that when it happens it will never return the program control back to the program until stipulated bytes have been read or it meets a "new line" input.
- A *GUI component input* can be arbitrarily redirected to any other suitable and visible component that has the current "*focus*".
- Currently, there is no chance to transfer data from one *Java process* to another Java process without using *socket stream* or *file stream*. If one day, there is a chance there, one *semantic event* created by a JavaBean component for example, could be transferred to a JavaBean component in another Java process, that *semantic event* could thus be an input to the second Java process (program). So we could possibly have, under the *Event Input* branch, another input called *semantic input* for that situation.

□ 4. User Input Capturing Level

The *user input* refers to user's operations to the target application. This applies only to *type 1* or *type 3* record/playback. We can capture a user input at three different levels. The lowest is at *hardware level*, second is at *operating system (OS) level* and the highest level is at *process level*.

★ 4.1 Hardware Level

This is by using hardware devices to record users' *keystrokes* and *mouse actions*. The main reason for choosing this hardware solution is for the security. The hardware device can catch the first level *keystrokes* to monitor the usage of the underlying computer or encrypt the key scan-code to prevent spying made by harmful malicious low level code.

An example of keystroke encryption device is called PC Pay(R)[15], which encrypt critical keystroke information like password and credit card number.

An example of keystroke monitor device is KeyGhost[16], which is a tiny (hardware) device that records every keystroke typed on any PC computer.

Obviously, this capturing level does not apply to the record/playback discussed in this report.

★ 4.2 Operating System (OS) Level

On this level, the capture tool is implemented by software solution. It uses the OS APIs that are normally in native code form. It can not recognize and access Java objects. Using this solution is mainly for providing OS scripting recording for repeating OS level scripting works, monitoring keystrokes at OS level, etc.

An example of OS scripting recording on this level is Keyboard Express[17] which records a serial keystrokes which can be assigned to one key-shortcut to perform batch or macro command effect.

An example of monitoring users' keystrokes on this level is Windows Keylogger[18] which spies and record all keystrokes at OS level.

It may works on this level to perform the record/playback function to Java programs. But I should avoid to discuss or use this level to Java program because Java object can not be fully recognized at this level, and the fact that using native code will not let it portable.

★ 4.3 Process Level

The concept of *process level* refers to catching all input to a specific process. Generally, the record/playback tool on this level could be written in native languages like C++ or Java language. In fact, most record/playback tools to the applications other than written in Java language are on *process level*. They

do have the ability to record input to a specific process (a native application) but only for GUI native application by extracting input from OS provided input stream or input queue for GUI components. The reason for define it on the *process level* is that it can distinguish the inputs to a specific process (a native process) from each other. A tool example of this type is QC/Replay[14], which is implemented in native code and provides the function of record/playback input to GUI X Window application. As mentioned in previous section, this native solution is not portable.

For Java program, it is different as a *Java process* is on the creation of one Java Virtual Machine (JVM). A *Java process* is different from a *native process*. Using native solution can not fully recognize Java objects because a Java object is wrapped via the JVM (but a Java GUI component may have a global ID which may be recognized at OS level). User input is not limited only to a GUI component. So, implementing a record/playback tool by using native solution can not correctly catch the inputs to a Java process. For Java target application, the best solution is to use Java language at process level to apply the record/playback solution.

An example is ReplayJava in [6] which uses the Java language and JTcl – JACL (also of Java language) to implement this tool with the ability to fully recognize Java objects and full portable feature.

5. Capturing User Inputs

Although several input types have been defined, I am currently interested in *console input* and *GUI component input*.

Even if only Java language is used on the process level to capture the user input, it is still hard to do it for *console input.* The reason is that it is difficult to decide which object (as the *input receiver*) will receive that *console input* and when it happens. It is can be done to blindly record all console input by inspecting the input stream at OS level, but without knowing what variable or object will receive that console input.

★ 5.1 Capturing GUI Component Input

Today, most applications are of GUI applications. Most record/playback tools are also designed to capture *GUI component input*. For native application, the

input can be *peeked* or taken from OS level input queue because every GUI component has a unique global identification at OS level.

For Java application, the capture could be done by inspecting the EventQueue object in the JVM. Java currently has two types of GUI components, AWT and Swing. Any GUI component input, *key input* or *mouse input*, correspond to an event called KeyEvent or MouseEvent which is in the EventQueue when the *key input* or *mouse input* happens. Another good thing is an event will always be accompanied by an *event source* – for KeyEvent and MouseEvent the *event source* is exactly the input receiver. So, to capture the input and get its *receiver component*, there is no need to change any source code.

★ 5.2 Console Input Capturing

Up to now, no any resource has been found for discussing *console input* to Java application. Although most of Java application use GUI interface, I must consider some special applications like a server that has no GUI interface. For embedding a *resilient* and *stealthy* watermark[1][2] into such an application, GUI components should be avoided to use for receiving input for recognizing that watermark, thus console input will be chosen.

Capturing console input is not easy. A console input is a stream input. When a console input is being processed, the program control will never return back until the designated length of bytes have been read or a *new-line* input has been met – I call this as the "*console input is finished or completed*".

I propose a trial solution by applying event mechanism to console input object for Java application. First create a wrapping class EventConsoleInput, then every System.in object at any place of the program will be replaced by this EventConsoleInput.

```
public class EventConsoleInput extends InputStream {
    InputStream in = System.in;
    InputEvent event;
    ConsoleInputStore store;
    Playback playback;
    public EventConsoleInput(InputEvent consoleInputEvent,
                             ConsoleInputStore store,
                             Playback playback) {
        super();
        event = consoleInputEvent;
```

```
            this.store = store;
            this.playback = playback;
        }
        public int read()  throws IOException{
            int input;
            if (playback.isConsolePlayback() &&
                store.hasReplayElement() ) {
                input =((Integer)store.nextPlaybackElement()).intValue();
            } else {
                event.dispatchPre(); // before input event
                input = in.read();
                // broadcast console input finished
                event.dispatchPost(new ConsoleInputEvent(), input);
            }
            return input;
        }

        .. ..

    }
```

Figure 2. Class EventConsoleInput

The proposed solution for console input

In Figure 2, the class ConsoleInputStore will catch the ConsoleInputEvent dispatched by the function call event.dispatchPost(ConsoleInputEvent, input) and store the console input value, "input" in this case.

In this proposed trial solution, any Java presentation of a *console input* – System.in object will be wrapped to a new EventConsoleInput object which can dispatch one pre-event before console input happens and dispatch one post-event after that console input finishes.

This is actually an *intrusive access mode* because it changes the source code. This may be a solution, but it has at least two problems. One is the event can only be dispatched after the *console input is finished.* Another problem is if the target application contains the code that redirects the System.in object with other stream input by calling System.setIn() method, this solution will actually capture the wrong input source.

This solution is too rough to express its correctness and accuracy. But here I am trying to show an idea that the console input may be captured with events, and thus can stored on the same mechanism – event mechanism – with the

GUI component input. In this solution, console input playback is also possible because the *read* method in Figure 2 will actually first check if it is playback mode or capture mode. If it is in playback mode, stored value will be send to the original *input receiver*.

- 6. Input Simulation

The above discussion for user input are all for *type 1 record/playback*. For *type 2 record/playback*, the input is provided by the *input simulation* using scripting language. This is well described at other place. Here an example (from [6]) is used for illustration as shown in Figure 3:

```
set frame [frames frame0]
set contentPane [$frame getContentPane]
set components [$contentPane getComponents]
set comp [$components get 0]
set btn [java::cast javax.swing.JButton $comp]
mousePressed $btn
sleep 500
mouseReleased $btn
mouseClicked $btn
```

Figure 3. Using scripting language to provide input simulation

Some features are shown in this illustration for simulation.

- A variable in a target application can be accessed, e.g.

  set fram [frames frome0].
- A class method can be invoked from the scripting shell level, e.g.

  Set contentPane [$frame getContentPane]
- A customized method at shell level can be mix used with objects in target appplication, eg.

  MousePressed $btn
- Program control can be fully controlled by the scripting language

- 7. Recording and Recording Schema

Recording schema denotes how to record the context or program internal state when capturing an underlying input.

There are two types of *recording schemas*, *real time* and *widget*.

★ 7.1 Real Time Schema

With *real time*, the key input and mouse input are replayed exactly as recorded[11]. This type is the simplest *recording schema*. But the downside is that it is easy to apply an inconsistent playback when the target application responds to events with different timing.

An example of this tool is Rebbeca 1.0[20], which uses real time recording schema to store the input context and playback.

★ 7.2 Widget Schema

The *widget schema* is actually for object level. Besides it records the input value, it also records its widget information. Using widget schema, a minimum set of test cases can be recorded. A *widget* is actually a GUI component. So a widget schema refers to record an input and its correspondent GUI component, which is the *receiver*, by sequence.

If the previously mentioned trial solution to capture *console input* is correct, the *widget schema* will not only limit to GUI components and will extend to any Java component with event mechanism. So a widget schema could be defined to record an input and its correspondent receiver object.

The downside of this schema is that exact time delays between events may be important for replay consistency.

Most record/playback tools use this schema, like QC/Replay[14], XRunner[21], Rational Visual Test[24], etc.

□ 8. Playback

The simplest way to replay the recorded input data is to replay at the restart of the target application. Often the case, this will bring overhead. The ideal is to start the playback at a desired execution point.

For GUI application, it is not a problem. Any GUI component input is to the GUI component which has current focus. This focus can be transferred and controlled by the program. So it is easy to set focus to a desired GUI component and dispatch the KeyEvent or MouseEvent back to the EventQueue with the desired GUI component focus.

For application with console input, first it is hard to locate the starting point for starting playback; second, the recorded data must be retrieved from storage to its receiver – i.e. i/o redirection is needed for all console input receivers[5]. The solution in Figure 2 shows this idea. It's better to start the playback at the point of restarting the target application. But if we have GUI input playback as

the first, then no need to start the playback from the very beginning of the whole program.

The above solution is for non-interactive playback (record/playback type 1) – playback without user's interruption. If with integration with other testing tools or to have the ability to inspect internal state of the program during the playback, the target application needs to stop at a specific point during that playback (record/playback type 2). In that case, a scripting language like JTcl[7] is definitely needed with that scripting shell environment.

9. Conclusion

This report aimed to discuss comprehensive issues for record/playback to Java application, including for the process of Java software watermarking. But it did not intend to provide a full discussion or a concrete solution to implement such a *record/playback* tool.

There are several definitions proposed in this report, such as the three types of record/playback(s); the taxonomy of input including console input, socket stream input etc; the three types of target applications – GUI application, text UI application and none-UI application; and three user input capturing levels – hardware level, OS level and process level.

This report proposed a trial solution for capturing console input; raised a specific question for console or none-UI Java application (specially for Java software watermarking) of how to activate an input receiver for beginning to receive the user inputs; discussed the recording schema – real time and widget.

10. Acknowledgement

Thanks to Dr. C.Collberg who showed me the first entry point for this project, otherwise I would hesitate for a while without getting the point. Special thanks to my supervisor, Prof. C.Thomborson, who guided me performing this research project; and gave the helps of showing me how Java software watermarking works with inputs, of how to analyze issues raised in record/playback, of keeping a good format to a technical report.

11. Reference

1. Christian Collberg, Clark Thomborson. Software Watermarking: Models and Dynamic Embeddings. In Principles of Programming Languages 1999, POPL'99, San Antonio, TX, January 1999. Available at http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson99a/index.html, Feb 2001
2. Christian Collberg, Clark Thomborson. Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. Presented at the DIMACS Workshop on Management of Digital Intellectual Property, April 2000, Rutgers University (NJ, USA). Submitted to *IEEE Transactions on Software Engineering*, July 2000. Available at http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson2000a/index.html, 8 Feb 2001
3. Christian Collberg. Tool: SandMark. Software Watermarking for Java – The Sandmark home. Available at http://www.cs.arizona.edu/sandmark/, 8 Feb 2001
4. Testing Java Application. Available at http://www.byte.com/art/9711/sec17/art4.htm, 8 Feb 2001
5. Alan Walworth. Automated testing is as problematic as it is essential. Java GUI Testing. *Dr. Dobb's Journal* February 1997. Available at http://www.ddj.com/articles/1997/9702/9702c/9702c.htm?topic=java, 8 Feb 2001
6. J.D. Newmarch. Testing Java swing-Based Applications. Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems. Available at http://pandonia.canberra.edu.au/java/replayjava/paper.html, 8 Feb 2001
7. Scriptics Tcl Resource Center. Available at http://www.scriptics.com/java, 8 Feb 2001
8. Krishnan Rangaraajan. How Can I Test Java Classes? Dr.Dobb's Journal, July 1999, pp. 107-110. Available at http://www.ddj.com/articles/1999/9907/9907k/9907k.htm?topic=java, 8 Feb 2001
   Jverifier™ -- Java class/API testing tool. Available at http://www.mmsindia.com/JVerify.html, 8 Feb 2001
9. Panorama for Java: JavaPlayback. Available at http://www.softwareautomation.com/panojava/webman/playback.htm, 8 Feb 2001
10. Sun WorkShop Visual Replay. Available at http://docs.sun.com/htmlcoll/coll.82.5/iso-8859-1/SWVISUG/Replay.html, 8 Feb 2001
11. Software Research, Inc. Capture/Playback Modes. Available at http://www.soft.com/AppNotes/capmodes.html, 8 Feb 2001
12. Kerry Zallar. Automated Software Testing – A Perspective. Available at http://www.scs.ubbcluj.ro/~pd27936/diploma/perspective.html, 8 Feb 2001

13. Linda J. McAlpine. Automated Testing: A Hat That Fits? Comm Central June/July 1996. Available at http://www.prairienet.org/cil_stc/cc2-5.html, 8 Feb 2001
14. Software: QC/Replay. Available at http://www.centerline.com/productline/qcreplay/qcreplay.html, 8 Feb 2001
15. PC Pay(R) Device. Available at http://www.innovonics.com/endtoend.html, and http://www.innovonics.com/pcpay/pcpayhome.html, 8 Feb 2001
16. KeyGhost ™ device. Available at http://www.keyghost.com/ or for introduction http://www.keyghost.com/kgpro.htm, 8 Feb 2001
17. Software: Keyboard Express. Available at http://www.keyboardexpress.com/, 8 Feb 2001
18. Software: Windows KeyLogger. Available at http://www.littlesister.de/, 8 Feb 2001
19. Brad Myers. Input Models. March, 1999. Available at http://www.cs.cmu.edu/~bam/uicourse/1999spring/lecture12input.html, 8 Feb 2001
20. Bob Dugan. Tool: Rebecca 1.0 home page. Available at http://www.cs.rpi.edu/~dugan/rebecca.html, 8 Feb 2001
21. XRunner home. Available at http://www-heva.mercuryinteractive.com/products/xrunner/, 8 Feb 2001
22. Mercury Interactive. Tool: WinRunner home. Available at http://www-svca.mercuryinteractive.com/products/winrunner/, 8 Feb 2001
23. HalloGram. WinRunner Introduction. Available at http://www.hallogram.com/winrunner/, 8 Feb, 2001
24. Rational. Tool: Rational Visual Test. Available at http://www.rational.com/products/visual_test/index.jsp, 8 Feb 2001