Testing Java Swing-Based Applications

J. D. Newmarch
University of Canberra
jan@ise.canberra.edu.au
http://pandonia.canberra.edu.au/

Abstract

Testing applications with a graphical user interface can be done in several ways, with the most important being by generating events for the event queue. This paper reports on the issues involved in this for Java applications that use Swing (or JFC) objects for their user interface. It also describes the replayJava system which can be used to record and playback user sessions, or can be used to directly generate a session. The resulting state of the application can then be checked for correctness.

1: Introduction

Testing is an important part of the software development process, to detect bugs, improve software design and to reduce usability problems. There are a number of test techniques such as the traditional unit, system, integration and acceptance tests, and a large number of software tools which support testing in different ways.

Object-oriented systems build upon the notion of class to link sets of objects to form larger systems such as packages or applications. Testing an object-oriented system may be done at the individual object level, but will more commonly involve a set of objects accessed through a small number of methods on a small number of objects. Testing may be *white box* or *black box*. In black box testing, no knowledge may be made about internal implementation structures, whereas white box testing will use internal knowledge to drive tests.

Object oriented programming uses both black box and white box techniques to construct applications: the inheritance mechanisms of OO systems are white box mechanisms, where the internal structures are open to child classes; the result is a *component* which is used only by its public interfaces. Black box testing of an object essentially uses only the public interfaces of objects as components. Many of the general issues of testing are dealt with in references such as [1] and [2].

Java is a relatively new language that can be used for building both applications and applets. Testing applets raises a number of specialised issues that have been discussed in [3]. Test tools customised to Java are still thin on the ground, with the major one probably being JavaStar [4]. A test system must be able to drive an application (or set of objects) and be able to examine the resultant state. In general this will require programming language capabilities such as conditional and loop statements.

This paper focuses on testing Java applications which use a graphical front-end. It discusses the various options and constraints in this, some of which are applicable to all

0-7695-0393-4/99 $10.00 © 1999 IEEE

such systems but some of which are caused by the particular structure of Java. The paper then discusses a tool - replayJava - that can be used for applications built using Java lightweight window objects, in particular, the Swing set of widgets.

The paper concentrates on the issues involved in building a testing tool. These issues depend on the specific details of the type of system under test. Related issues are reported for unit testing of C++ objects by Hunt [5] and for collection classes by Hoffman and Strooper [6] in addition to Newmarch [5] for GUI classes using the Xt toolkit.

2: Testing applications with a GUI front-end

A large number of applications are written to run in graphical user environments such as Microsoft Windows or under the X Window System. Such environments use the particular programming model of *event driven programming*, and this leads to distinct methods of software testing. An application with a graphical user interface (GUI) is written to respond to *events* posted into an event queue by the supporting windows system. The application removes events from the queue, decodes them, and forwards them to the appropriate window object for application-specific actions to be taken.

This leads to two levels of driving GUI objects. The indirect, or event driven method is to prepare events and post them into the event queue. The event handling system will then deliver these to the objects. The alternative is to call public methods directly on the GUI objects. The first method is guaranteed to be complete: a GUI object must be able to respond appropriately to all user events. On the other hand, the public methods may be incomplete, and only supplied as convenience methods.

Particular systems may supply intermediate mechanisms. For example, Java has "semantic" events such as `ActionEvent` which are generated by combinations of `MousePressed` and `MouseReleased` events. These are not posted to the event queue, but are delivered directly to the object. As another example, The Xt toolkit (used by Motif) has *actions* which are halfway on the translation from events to public methods. These were used in the earlier replayXt system of the author [5].

A number of GUI test systems act externally to all GUI applications, generating events at the native window system level. These systems will not be considered in this paper.

3: AWT and Swing

Java has a number of GUI toolkits. The first was the AWT, which relied heavily on native GUI objects such as Windows MFC components or Motif components. There is an event queue for such objects, but special purpose code is present to eliminate program-generated events from this queue - this looks like a deliberate defence against incomplete toolkit code. While the intent may have been to complete the code, it looks like this will never be done. So it is impossible to drive AWT objects using program-generated events.

The major replacement in the JDK 1.2 (Also JDK 2.0) is the Swing set. All event handling is performed at the Java level. The event dispatch for a mouse button press in a JButton looks like

Swing does not artificially distinguish the origin of events. So it is possible to generate input events, feed them into the event queue, and have them drive the GUI objects.

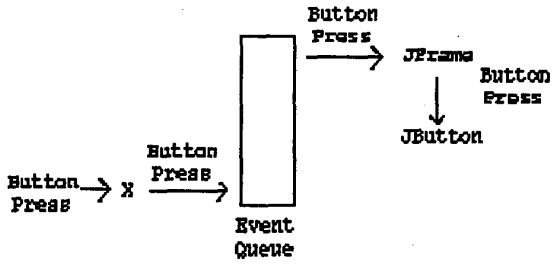

Figure 1. Mobile Infrastructure

Events received by a GUI object are handled by listeners. Each of the Swing objects is actually a set of objects built using a modified form of the Model-View-Controller. For example, a JButton is made up of five objects: the button itself, a model, a user-interface view and two state and event listeners

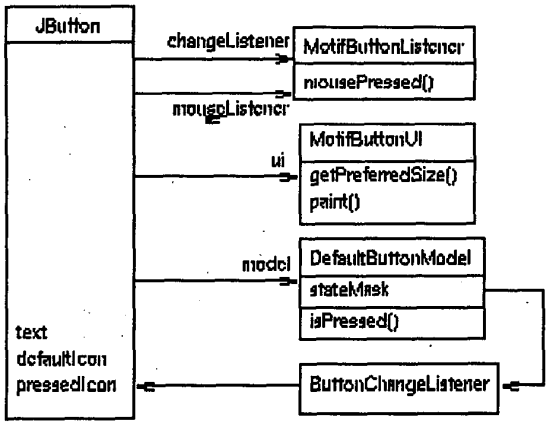

Figure 2. Mobile Infrastructure

The event listener will watch for input events such as mouse and keyboard events, and call various methods in the original object, the model or the user interface. These methods will perform semantic actions based on the input event. The methods may be public or non-public and may or may not be documented.

Some semantic methods may cause additional events to be generated. These semantic level events, such as `ActionEvent` or `ItemSelectedEvent` are not placed in the event queue but instead are passed as parameters to other methods of the GUI object. These methods are typically not public, and so cannot be used in a test/replay system.

4: Input events

Input events are normally generated by native code watching for native-level events and posted into the event queue. Events from this source are caused by a user interacting with an application by the mouse and keyboard. There are some test systems which will also generate these "external" events, but such systems are not further considered here. A separate event-dispatch thread is responsible for removing events from this queue and passing them on to the appropriate object.

For Swing objects events may also be generated programmatically and posted. Like any test/replay system that generates events, this gives rise to a number of issues, some which apply to every such system, some of which are Java specific.

4.1: Event coordinates

Mouse events have `(x, y)` coordinates, which may be examined by the receiving object. For example, a list will use them to determine which element of the list has been selected. These cannot be hard-coded, particularly in a Java system. Such coordinates will depend on user preferences in setting the size and shape of an application and other factors such as font size of text elements in a list. Thus there must be a means of determining at runtime the `(x, y)` coordinates of an event from some semantic level information.

For example, in a `JList`, there is the method `indexToLocation(int index)` which will return a `Point` in list coordinates for a particular index. As another example, a `JTabbedPane` has the method `getBoundsAt(index tab)` which returns a bounding `Rectangle` for the tab itself. An event can then be prepared at (say) the center of this rectangle.

4.2: Object identification

An application once set running will generate objects, some from the Swing objects themselves, most from the application program. A test/replay system must be able to locate Swing objects in order to prepare events for them or to call semantic methods on them. There are several mechanisms for this:

- by role
- by name
- by index

A `JRootPane` is the parent of two objects, a layered pane and a glass pane. These play different roles and may be found by methods such as `getLayeredPane()` and `getGlassPane()`. On the other hand, a set of objects created by an application and added to a container have no identified role, and cannot be found by this means.

Any Swing object may be given a name, which currently defaults to `null`. Most Java applications do not set a value, so this cannot be used in most cases. On the other hand, each `Frame` (or `JFrame`) will have a name if requested by `getName()` as `frame0`, `frame1`, `frame2`, *etc.* Coupled with the static method `Frame.getFrames()`, this allows toplevel frames to be searched for by name. This will give the top of a tree of GUI objects and allows any of the GUI objects to be located.

Unfortunately, the name of a frame may not stay the same between application runs. Unless an application sets the name explicitly it is left as `null`. When requested, a `null`

value is converted to the next unused name in the sequence `frame0`, `frame1`, `frame2`, *etc.* So if one run of an application attempts to get the name of a frame after, say, walking up the GUI tree from a component, and a later run attempts to search for this frame using that name, it may end with a different frame. This is because the search through the list returned by `Frame.getFrames()` may assign names in a different order to the name assigned by finding the top of a tree and asking for *its* name. This has been reported as a potential bug in JDK 1.2. A workaround is to loop through all frames in a standard order calling `getName()` to force all names to be set before a real request is made for a particular name. This is to defeat the "lazy" method of assigning names to frames.

The third way of finding a component from the top of a GUI tree is to walk down the children of the top node, the grandchildren, etc, using the index of a component in its parent. Generally this will be deterministic across application runs as the tree will be built the same way in each run. However, there is at least one exception. When an application is run under the Metal Look-and-Feel, Swing creates an extra top-level frame which is not created under the other L&F's. This alters the index value of application-created frames in `Frame.getFrames()` if the L&F changes.

4.3: Timing

Event dispatch takes place in its own thread. If an event is prepared and posted, it may have some effect on the structure of the GUI tree. For example, pressing a `JMenu` will post a menu. This adds the popup menu to the `LayeredPane` under the `JRootPane`. In other situations a dialog may get created and posted, a frame may get created, an element may be added to a list, and so on. The multi-threaded nature of event processing here may be an issue for timing reasons. It is not an issue to post new events. Problems may arise if an event is posted and then a query on state is made (such as finding a particular object) on the assumption that the event has been processed.

5: Encapsulating input events

The interaction by the user with a component often takes place at a coarser level of granularity than the events. For example, the user action of "clicking a button" generates events such as mouse pressed and released (and others). These may be encapsulated in sequences of events. This also allows the opportunity to make more robust sets of events.

"clicking a button" is an identifiable set. Robustness may be added to the mouse pressed/released in a number of ways

- The set of events can be completed - a mouse click may also involve a mouse clicked event and a focus gained event
- Swing internally caches the object with the focus, to avoid having to calculate it for each event. The cache may need to be invalidated, and this can be done by a mouse moved event
- Mouse events require `(x, y)` coordinates which should be calculated from the current size of the object
- The object can be checked to really be of the expected class

This leads to (pseudo-) code such

```
click btn {
    throw exception if btn isn't a button
    find size of btn
    calculate midpoint (x, y)
    create and post mouse moved event
    create and post mouse pressed event
    create and post focus gained event
    sleep for 1/2 second
    create and post mouse released event
    create and post mouse clicked event
}
```

6: Recording input events

A common way of building a replay/test script is to record a user-driven run through an application. This can then be used as the basis of a test script. JDK 1.2 supplies a hook to do this by the `Toolkit` method `addAWTEventListener(listener, eventMask)`. If an event is dispatched that matches the mask then the listener method `eventDispatched(AWTEvent evt)` is called.

A naive recording will simply save the event in some suitable format in a file. This information can be used to reconstruct an event on playback. Simply recording the `(x, y)` information is not robust, as discussed earlier. A semantic representation of the event is required. For example, a mouse press in a `JTabbedPane` should have the tab recorded, not the `(x, y)` coordinates. Then on replay a suitable point can be constructed, as discussed earlier. While the Swing toolkit will need to be able to perform such semantic extraction, the method to do so need not be public. For the `JTabbedPane` it is, as the method `tabForCoordinate()` of the class `JTabbedPaneUI`. Often such methods are found are in the user-interface component of an object's MVC.

A recording will also need to be able to store the identity of the object the event occurred in so that it can be reconstructed on playback. In some cases it may be possible to determine the role of an object *e.g.* if the object has a `JRootPane` as parent, is it the `glassPane`? If the name is non-null this can be recorded, but this is unlikely. The simplest is to record the index in each ancestor upto the toplevel `Frame`, which can have its name recorded. This is not completely robust, as discussed earlier.

7: Semantic events

When a Java program is written to handle a JButton, a JList or many other of the Swing objects, the event handlers are for the *semantic* events such as `ActionEvent`, `ListSelectionEvent`, *etc.* These events are generated by Swing in response to certain combinations of input events, and are not posted to the event queue. It is not often a simple matter to find where they are generated, as this is buried in the modified MVC mechanism used by Swing.

For example, a JButton has a BasicButtonListener registered to listen for mouse events in the button. This calls methods in DefaultButtonModel such as `setArmed()` and `setPressed()`. Within the (current) implementation of `setPressed()` some aspects of state are checked,

and if they match then a new `ActionEvent` is created and sent to all the JButton's action listeners by calling their `actionPerformed()` method.

While the model's methods `setArmed()` and `setPressed()` are public, the actual code to generate and post the semantic event is not. Thus there appears to be little point in generating semantic events.

8: Semantic methods

Input events trigger responses in the Swing objects. This is also tied into the MVC mechanism used by Swing. In the JButton, a BasicButtonListener translates input events into semantic methods by methods such as `mousePressed()`. This uses semantic methods of the model such as `setArmed()` and `setPressed()`

```
public void mousePressed(MouseEvent e) {
    if ( SwingUtilities.isLeftMouseButton(e) ) {
        AbstractButton b = (AbstractButton) e.getSource();
        ButtonModel model = b.getModel();
        if (!model.isEnabled()) {
            // Disabled buttons ignore all input...
            return;
        }

        model.setArmed(true);
        model.setPressed(true);
        if(!b.hasFocus()) {
            b.requestFocus();
        }
    }
}
```

These methods can also be used by a replay/test system by

```
{\tt
model = getModel()
model.setArmed(true);
model.setPressed(true);
}
```

If these semantic methods are used in addition to input methods, then the issue of timing arises again: if the input queue is non-empty then calling a semantic method directly will "leapfrog" the pending events. There is no "flush queue" method for the event queue, but an object can be added to the queue by `invokeLater(Runnable)` which will be added to the queue and its `run()` method called when appropriate.

In some cases, there are apparent semantic methods available which are not quite the same as the real execution mechanism. An example is the `JList`, with methods `setSelectedIndex()` and `setSelectedIndices()`. The response to input methods is more complex:

- A mouse press sets state (`isAdjusting` is set to `true`).
- A mouse drag uses this state

- A mouse release resets this state

This illustrates the main problem in using semantic methods: while the mapping from input events to methods is sometimes documented (as in `AbstractButton`), it is not commonly done. There is the possibility of error, or changing implementation that can break assumptions of the meaning of methods.

9: Encapsulating semantic methods

For some Swing objects, public methods have been made available as part of the specification that encapsulate the semantic methods. An example is the Java `doClick()` method of AbstractButton:

```
public void doClick(int pressTime) {
    Dimension size = getSize();
    model.setArmed(true);
    model.setPressed(true);
    paintImmediately(new Rectangle(0,0, size.width, size.height));
    try {
        Thread.currentThread().sleep(pressTime);
    } catch(InterruptedException ie) {
    }
    model.setPressed(false);
    model.setArmed(false);
}
```

Only a few classes have such encapsulation, but similar methods can be devised for many. Some methods, such as `JList`'s `setSelectedIndex()` are *not* encapsulations in this sense, but are assumed to be functionally equivalent.

10: Testing

A replay system can be used for automated demonstrations. These just need to drive an application through a series of steps. A test system must go further than this, and allow the ability to examine application state at the end (or during) a replay. At a minimum it should allow location and examination (by public methods) of the state of any GUI object. It may need to go beyond this in various ways:

- a test may need to examine state of non-GUI objects. This will require a mechanism to access such objects, and will need to be designed into the application
- a test may need to examine external files or run external applications
- a test may need to examine visual appearance of the application. While the appearance of many components may depend on geometry or on the L&F adopted, some may not: for example, is a particular image showing correctly

11: The replayJava system

A package has been built that attempts to supply a test/replay mechanism for Swing-based applications based on the above considerations. Primary requirements for this were:

- it should be based on a full programming language to allow arbitrarily complex tests to be performed
- it should run on every Java platform
- an interpreted scripting language is more convenient for black-box testing than a compiled O/O language (as evidence the success of Visual Basic as a language for manipulating components)
- it should allow playback using input events as well as semantic methods
- it should allow some method of recording events
- it should allow location of objects and acces to their public methods
- it should be able to access files and applications external to the application

The JACL system [11] was chosen as scripting language because:

- it is based on the well-known tcl language, with full access to a range of features such as file access
- it is written in Java and is portable across Java platforms
- it uses the Reflection API [12] to gain full access to Java objects and methods from tcl

A number of commands were written in Java and exported to the JACL interpreter. These include commands to create and post input events, to sleep for an interval and to find a toplevel `Frame` by name. This allows a button in a frame to be located and clicked by

```
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
```

Other objects may be driven in a similar manner.
At present, replayJava

- implements input events
- has some encapsulation of input events
- has a primitive recort mechanism
- has an incomplete documentation of semantic methods plus semantic encapsulation

Version 1.0 of replayJava was designed to work with JDK 1.0 and early versions of Swing. This included workarounds for various problems in these. Version 2.0 is designed to work

with JDK 1.2 and later, and is cleaner in code.

12: Acknowledgements

The author is currently on a sabbatical program at the CRC for Distributed Systems Technology, and the work reported in this paper has been funded in part by the Co-operative Research Centre Program through the Department of Industry, Science and Tourism of the Commonwealth Government of Australia.

13: Conclusion

This paper has discussd the issues involved in building a test tool for Java applications using a GUI frontend. For the Swing objects, it is fairly straightforward to build a test tool that uses event generation to drive GUI applications. There are some minor problems, but these will disappear in later versions of the JDK and the Swing objects.

The `replayJava` package is under active development, and is available as Open Source from http://pandonia.canberra.edu.au/java/replayJava. Future development will concentrate on filling gaps in the present system, with extensions to handle a range of semantic checks that can be performed.

14: References

[1] B. Bezier *Black-box Testing*, J. Wiley and Sons, 1995
[2] E. Kit *Software Testing in the Real World*, Addison-Wesley, 1995
[3] J. D. Newmarch *Issues in Testing Java Applets* First Int Workshop on Web Engineering, Brisbane, 1998
[4] SunTest Home page, http://suntest.sun.com
[5] J. D. Newmarch *Using tcl to Replay Xt Applications* Proc AUUG 94, Melbourne 1994 http://pandonia.canberra.edu.au/repla
[6] N. Hunt *Unit Testing* JOOP, Feb 1996 [7] D. Hoffman and P. Strooper *The Testgraph methodology: Automated testing of collection classes*, JOOP, Nov/Dec 1995
[8] J. D. Newmarch *Advanced Event Handling* in *Tricks of the Java Programming Gurus*, ed G. Vandenburg, SAMS Net 1996
[9] J. D. Newmarch *The Delegation Event Model* in *Mastering Java* ed G. Vandenburg, SAMS Net 1997
[10] Swing Connection, http://java.sun.com/products/jfc/swingdoc-current
[11] Scriptics *Tcl Resource Center*, http://www.scriptics.com/java/
[12] D. Green, *The Reflection API*, http://java.sun.com/docs/books/tutorial/reflect/index.html