

Java GUI Testing Tools

Well, this is my take, and while I try to be unbiased, I *am* the author of one of the frameworks. Be sure to take a look at some of the yahoo java-gui-testing archives as well; there are links to a few tools that aren't covered here. Note also that these are Java-only solutions. There are probably equivalents for other environments, but the same sorts of decisions you make picking a context-appropriate java tool would apply elsewhere. From a high-level perspective the same sorts of testing ideas still apply.

There are three or four main frameworks out there. I think Jemmy has been around the longest, although it only became public about the time abbot came out (april 2002).

abbot: my own piece o' work. internally used at Oculus Technologies, released to the wild around 4/2002. <http://abbot.sf.net>.

jemmy: mostly the work of Alexandre "Shura" Iline, of sun. acquired with the netbeans IDE. <http://jemmy.netbeans.org>.

jfcunit: maintained by Vijay Aravamudhan and Kevin Wilson. it's been around a while, through a few owners. <http://jfcunit.sf.net>. Some ties to Thoughtworks.

marathon(man): A Thoughtworks production. probably the most recent project. <http://marathonman.sf.net>.

There are a few other projects listed at <http://groups.yahoo.com/java-gui-testing>, but are either lesser-used or not entirely free.

Here's a bullet list of features which might help you direct future research:

Type of Testing

Two main goals here. First, unit testing for developers. Abbot, Jemmy, and JFCUnit provide ways to unit-test GUI code. If you're developing components or groups of components and want to ensure they work as advertised, this is the way to go.

Abbot and Marathon provide higher-level scripting actions more suitable for acceptance testing (JFCUnit has started work on some level of scripting support). The types of actions you want to be able to do here are "log in" or "select user records", which are usually definable as a script comprising more granular actions.

The underlying mechanics for the two types of testing are (or should be) identical, although unit testing generally has a much finer granularity of

events/actions.

Finally, Abbot and Marathon provide methods for recording a series of user actions on a system under test for further manipulation by hand or later playback.

Test Drivers

Tests can be driven in three ways (or any combination). Java provides a Robot class (as of release 1.3) which can generate native OS events. These get processed by the OS, passed on to and processed by the VM, and get converted to Java's AWT events handled on its event dispatch thread. This level of testing is closest to real user behavior, since the mouse pointer actually moves and keys are actually pressed (well, maybe your keyboard doesn't actually dance). That of course means you can't do anything else with the system while it's busy testing, and other programs can potentially interfere with input.

Tests can also be driven by posting events directly to the Java AWT event queue. This method allows you to (mostly) run tests even while the system is busy doing other things, or even potentially running several tests simultaneously. While this is more consistent in many situations, it also encodes assumptions about the actual event streams that will be generated (which aren't necessarily relevant to the test at hand). The framework must also simulate pseudo events (like `MOUSE_CLICKED`) that normally the VM would generate in response to native events. There are also some features (Drag & Drop) which can't be simulated by posting AWT Events.

Finally, GUI components can be driven programmatically, by directly invoking methods on them. The design of Java components is not nearly consistent enough for this method to be complete. The `selectItem` method on one component might notify listeners while on another it might not, and yet another component might not even make such a method public.

The appropriate drive depends on your situation so let your context be your guide.

Abbot provides both robot (default) and AWT modes, and throws in a little programmatic control when all else fails. Jemmy and JFCUnit use AWT mode by default but provide a robot version if needed. Marathon uses a mix of AWT control and Java's Accessibility libraries (programmatic control).

Actions API

Abbot's API was originally designed with the intent of making any GUI object scriptable. That means exposing any action that the user can make (clicking buttons, selecting from lists, double-clicking on rows in a tree) as a named action. That, combined with a method for getting a reference to any given component in the hierarchy. See <http://abbot.sourceforge.net/doc/api/abbot/doc-files/about.html>

for some additional background. Each GUI class has a corresponding tester class which knows how to invoke the user actions available for that component.

Jemmy's approach reflects its concern with doing every action on the event dispatch thread. Every GUI component is wrapped in an Operator object with an API similar to the component. Each method "hops" threads to the event dispatch thread and hops back with a result. For good or ill, every component API method is duplicated in the operator API, so the javadocs are rather long. Whether or not the API reflects a user's actions depends on the design of the underlying component.

JFCUnit uses custom event classes to represent types of user input. For instance, there is a JComboBoxMouseEvent class which represents a user mouse event on a combo box. It's certainly a different abstraction, but I feel it focuses too much on the type of low level event rather on the type of user action that is being generated. The provided actions in the helper classes are relatively low level, so higher semantic level control is limited. JFCUnit also goes to great effort to ensure the event dispatch thread does not run at the same time as the test code, which introduces some extra complexity.

Marathon provides user-level actions exposed as methods available to Jython (Java-based Python) scripts. It uses the Java Accessibility API and JFCUnit among other things to actually control GUI components. The developers have tried to make the scripts easy to read and comprehend.

Component Lookup

A framework needs to be able to take a number of known or expected attributes and convert them into an actual reference to a GUI component. Developers shouldn't be forced to expose internal GUI components just for testing (there is no need for a "getOKButton()" in the API to a login dialog). The Java GUI hierarchy can (mostly) be walked given the Component/Container APIs; the framework should provide various methods do this so the developer/tester doesn't have to.

Abbot uses a dedicated class to represent GUI components (since the components don't necessarily exist when you first need to reference them). It stores a range of attributes to be used when looking for a matching component in the hierarchy, and does a fuzzy lookup to avoid spurious errors due to component movement or minor changes to the GUI. The lookup mechanism is very general due to the fact it is used by the scripting layer which has no a priori knowledge of the GUI hierarchy or what needs to be looked up. The framework also provides a

number of utilities to facilitate inspecting the GUI hierarchy itself.

Jemmy provides a few different lookup classes that find a component based on different criteria built into the class; the programmer needs to pick the right lookup class based on knowledge of the hierarchy.

JFCUnit provides component-specific lookup classes, e.g. `AbstractButtonFinder`. These are used in conjunction with a test helper class to look up a desired component.

Marathon mainly expects to find components by name, assuming the use of `Component.setName()`.

Architecture/Extensibility

Mostly this consists of “what do I have to do to test a custom component”? Chances are, nothing. You can build up basic clicks and keystrokes to control most anything, but you probably want something that explicitly exports the “spin me till I’m dizzy” action (assuming you’re not the developer; if you were, you’d already have that as a public method on your class, right?).

Abbot provides for extending the basic tester classes for new components, and has a well defined method for naming those new actions so they are automatically available at the scripting level. It also provides extensibility for converting strings from scripts into arbitrary classes, and introducing new individual steps into scripts. Scripts can call directly into java code (the script is actually just a thin veneer over method calls).

Jemmy has a rather high hurdle for introducing a new operator, although you are by no means obliged to duplicate your entire API in the operator. Jemmy has a higher-level xml-based scripting tool called Jelly (no relation to the apache tool) that facilitates higher-level application actions, and I believe this is extensible. It is what the netbeans IDE group uses to test the IDE.

JFCUnit requires you define new event types for your component derived from the basic ones, e.g. `MyComponentMouseEvent` derived from `AbstractMouseEvent`.

Since Marathon is based on Jython, you can either provide your own Python libraries or call into underlying Java code.

Testing Context

Do you run one test in isolation, many with the same fixture, or every test in a new fixture? Everyone has some integration with JUnit.

Abbot provides both a script environment and a JUnit fixture, both of which handle setup and teardown of the complete GUI environment. Scripts also support launching a single application instance and including other scripts.

I’m not familiar with what the other frameworks provide, but I’m sure they

all have some flavor of `junit.framework.TestCase`.

When choosing one of these, take into account your specific needs. They're all freely available, so it makes sense to spend a little time with each one to see which feels right for you. And please provide feedback. These are all relative babies and are really just exploring the world of GUI testing.

Timothy Wall

twall@users.sourceforge.net

copyright © 2004 Timothy Wall, all rights reserved