

Model-Based Testing for Java and Web applications
By Jeffrey Feldstein
jbf@sherps.com
<http://www.sherpas.com>

Copyright 2005-2006 Jeffrey Feldstein

Most software testing begins with a test engineer sitting down at the computer, starting the application, randomly navigating it's features, searching for bugs and tripping across crashes. Since the test engineer knows (or learns while he is testing) what the application is supposed to do, he can file a bug when the application doesn't get it right. In addition he will file bugs for all the crashes he is able to reproduce. Eventually, the test engineer generates enough test cases that he wants to remember to re-run on the next build or release. This results in writing a comprehensive test plan. The core of the test plan is the description of all the test cases to run on each build. After running through the test plan a few times, the good test engineer decides that the computer can do a faster, more accurate job of running some of these test cases, so he automates as many as he can. He might then logically sequence these test cases and run through the entire use case, checking for errors and crashes. This automation forms the basis of his regression test. The good engineer might even data-drive the test cases so that the same logic can be re-used against large sets of input and expected-result data. I will refer to this type of automation as "classic test automation".

One problem with this evolution is that the random sequencing which was implicit (and probably unconscious), in the manually testing is gone. As soon as the application can run through the sequence of test cases with no apparent errors or crashes, it's shipped. Now the customer, having just purchased and installed the application, uses functions in an order that differs from the automation and finds some nasty bugs. The customer calls support, support reports the sequence to development who fixes the bug. The test engineer, in reviewing the bug, sees what he missed in his automation, adds this new sequence to his tests, and is happy. At least until the next customer tries the same functions in yet a different order and breaks the application in yet a new way. Right about now, management is probably thinking that maybe an army of manual testers wasn't such a bad idea.

There is however, another approach. Imagine if the test automation knew where the current feature (or screen) being tested lived in the application relative to the other features (or screens). Imagine further, that it knew all of the possible ways to transit out of the current feature and where in the application it is supposed to be after taking this transition. The automation could execute the current feature, check for the correct results, pick a transition at random, end up in a new feature, test this new feature and randomly pick yet another transition. This process can then be repeated indefinitely, bringing back the random testing that was lost when the human was removed from running the tests.

Model-Based Testing, is a form of automated testing which implements random behavior in your test automation software. Each application feature(or screen), becomes a “state” and each button, link, menu selection or command leading to the next function becomes a “transition”. Your application’s model is a memory based representation of each state, all of its transitions and the destination state that should result when a particular transition is selected.

In a web or graphical Java application a screen or message box usually corresponds to a state. All the buttons, links, tabs and other objects that take the user to another state or screen are the transitions. The validation of the business or application logic at each state remains the same, or similar to, the validation logic from classic test automation, described above.

By applying various navigation algorithms to the model (i.e. all random, randomly choose an unvisited state, visit all states in order), the application is exercised in new or more complete ways than is possible in classic automation. In addition, with model-based testing, adding computers will increase the test coverage over a given time by generating even more paths through the code. In classic automation no matter how many times you run through the automation it’s always the same. In model-based testing, since the navigation can be random computers running in parallel to each other will be executing different paths through the code. In Java or web applications where many clients run against a single server, this method can is a valuable tool to test multi-user access as well.

Model-based testing can be used for many types of software or application testing. The concepts remain the same whether you are testing a cell phone's embedded system or a browser based Internet application. There is a problem, however, when trying to implement model-based testing for Java or Web applications. Most of the legacy test tools are designed primarily for script recording and playback. Although logic can be hand-manipulated and tuned by directly editing the source code, their native languages are not industrial strength programming tools. They do not allow for easy declaration and manipulation of large scale data structures and object oriented design is nearly impossible. These issues are solved with IBM Rational's Functional Test for Java and Web, because Java itself is the test script language. This is not some subset of Java or any “Java-like” language. It’s the real thing, it runs on a variety of JVMs and is implemented as a plug-in of the Eclipse 3.0 IDE and runtime environment. The test engineer, therefore, has all of the programming power afforded to application programmer. This environment lends itself to highly-engineered, highly-scalable, flexible and reliable test automation software.

Implementation Details

Now let’s explore the basic data structures, logic flow, and navigation algorithms required for model-based testing of Java and Web applications. All of the source code, a working version of a model-based test system, a sample application and pointers to download an evaluation version of the IBM Rational Functional Test for Java software are available from my model-based test web page: <http://www.sherpas.com/mbt>.

Take a look at Figure 1. It shows a simple web portal to illustrate a browser based application consisting of six tabs. Each tab brings you to a new screen. Each screen is a state. Since all six tabs are available from each other, you can navigate to any state by clicking it's associated tab. Each tab becomes a transition to a new state. Figure 2 illustrates a three-state model using just three tabs, "Home", "Images" and "News". Each screen is a state. From the "Home Page" state, three transitions are show, "Home", "News" and "Images". A transition from a state is shown as an arrow leading out of a state. Each arrow points to it's corresponding destination state. This model shows, that when you click on the "News" tab, you should transition to the "News" screen, the "Images" tab should take you to the "Images" screen and the "Home" tag should return you to the home state. Similar information is stored for the "Images" and "News" states. These states, transitions and destinations form the model of the application that we can test against. One way to represent this model memory as in table:

From State	Transition	Target State
Home Screen	News Tab	News Screen
Home Screen	Images Tab	Images Screen
Home Screen	Home Tab	Home Screen
Images Screen	Home Tab	Home Screen
Images Screen	News Tab	News Screen
Images Screen	Images Tab	Images Screen
News Screen	Home Tab	Home Screen
News Screen	Images Tab	Images Screen
News Screen	News Tab	News Screen

When testing the application, after selecting a transition you check to be sure you are in the correct state, then check the logic and other functions inside that screen for correctness. IBM's term for the logic that tests the correctness of any screen is a "Verification Point".

Three main objects are needed to implement this model in Java:

- An array of states
- A list of all transitions from a state
- Properties of each transition (including the destination state)

State Class:

A class called `State` stores state information and holds the constructor. For more information on classes, constructors and other Java terms see one of the Java references below. Here is the `State` class:

```
class State
{
    public int m_state;
    public transitionList m_transitions;
```

```

    public String m_stateName;
    public boolean bVisited;
    public int distToDest;
    public int transitionToDest;
    // Method VerifyMethod;
    String VerifyMethod;
    State(int s)
    {
        bVisited = false;
        m_state = s;
        m_transitions = new transitionList();
    }
}

```

The `m_state` field, is a sequence number used to index each state. It's simply convenient shorthand to refer to a specific state. `m_transitions` is the list of transitions from this state. The details are explained below. `m_stateName` is a text label for the state used for logging and debugging. It's there to make it easier for a human to read, the program itself does not refer to it. `bVisited` is a boolean that is initialized to FALSE in the constructor but set to TRUE when the state is visited. `distToDest` and `transitionToDest` together with `bVisited` are used by the advanced navigation algorithms that attempt visit all nodes, instead of completely random navigation. The detailed explanation for these can be found in the comments of the downloaded source code. Ignore them until the basics of the model are understood. `VerifyMethod` is the name of the verification point method for this state. In other words, to check the back end logic of a state this, this method will be called after navigating to the destination state.

Transition Classes:

```

public class transition
{
    public GuiTestObject m_trigger;
    public State m_destination;
    public transition(GuiTestObject o, State s){
        m_trigger = o; m_destination = s;
    }
}

public class transitionList
{
    private java.util.List list = new ArrayList();

    public void add(transition m) { list.add(m); }
    public transition get(int index) {
        return (transition)list.get(index);
    }
}

```

```

        public int size() { return list.size(); }
    }

```

The `transition` class describes all of the properties for a particular transition. Each element in the `transtionList` contains a transition plus some other objects. An `m_trigger` is the object on the screen which is selected to transition to the next state. It's type, `GuiTestObject`, is defined by the object map provided by Functional Test for Java and Web. `m_destination` is the destination state, stored as an index into the state array. The constructor for the transition state simply stores these two objects.

->The `transtionList` class contains three methods that operate on the list element created in the constructor. This is `add` (a transition to the list), `get` (a transition from the list) and `size` (returns the number of transitions from this state.).

Combining the above objects we can write code to represent the model. For each transition you write a line of code that looks like this:

```

m_states[States.HOME].m_transitions.add(
    new transition ButtonImages(),m_states[States.IMAGES]);

```

It may look a little daunting, but it's too bad once you code a few lines of it. Translated to English this says: In my array of states (`m_states`), add a transition from the "Home" tab (`States.HOME`) to the "Images" tab. The object that performs this transition is the "Images" tab (`ButtonImages()`). A similar line is written for each transition in the model.

Major Features

The source code from the web site contains several features that are useful for model-based testing which are described below. To take advantage of these features, small modifications to the code itself are required such as commenting a line of code while un-commenting it's neighbor. Changing lines of code to access features of course, isn't the best idea for production software. It was implemented this way here for illustration purposes only. For actual test systems, these options should be parameterized and passed into the test.

Navigation Algorithms

There are three navigation algorithms implemented, `RANDOM`, `ALLPATHS` and `PLAYLOG`.

`RANDOM`, as the name implies, starts with the initial state, runs it's verification point method, randomly picks one of the transitions associated with the state, and clicks that transition to navigate to the target state. When the program arrives in the target state, the

new state's verification point method is called and the process is repeated until the stop time is reached.

ALLPATHS, like RANDOM begins with the initial state and runs it's verification point method. It then finds the first transition that leads to an unvisited state. If all of the states which are neighbors to this one (accessible by one transition), have been visited, it searches through the entire state array for an unvisited state. If an unvisited state is found, a path to that state is computed. The program navigates to that state. Once all the states have been visited, all states are marked as unvisited and the entire process begins again.

One suggestion for improvement is enhancing the algorithm to add some randomness. Either when picking a transition or searching for an unvisited state, a random number can be used instead of a sequential search.

Test Duration

Three variables control the length of time that the test will run. They are called `days`, `minutes` and `seconds`. The values of these variables are used to calculate the test's duration. For instance:

```
int days = 2;
int minutes = 7;
int seconds = 5;
```

will set the test duration for 2 days, 7 minutes and 5 seconds.

Two suggested additional features not yet in the code but which could be added is to include an "hours" variable and the ability to run forever by adding logic to check for all values = 0 (or some other magic number that you choose).

Run Log

Functional Test for Java and Web automatically generates an HTML (or optionally a text) log. This log will automatically give you the following events: script start and end, application start, timer start and stop, exceptions, and verification point results. In addition, since source code line numbers and file names are logged, this file becomes a nice diagnostic tool for your automation. Functional Test gives you the ability to add your own text lines to this log using these methods: `logInfo()`, `logWarning()`, `logError()`, and `logTestResult()`.

The example code uses the `logInfo()` method to log the calculated start and end times, when a state reached and which verification point method was called. These log entries help you debug the model itself.

Navigation Log

One problem with model-based testing is reproducing the steps to a crash or error. For manual testing, the test engineer often spends a lot of time isolating and reproducing an

error so that it can be easily fixed by development. This is not successful 100% of the time. In classical test automation, the problem is mitigated somewhat because all that is needed to reproduce the problem is to re-run the automation. But in model-based testing, the steps to being run are random. In addition as the time variables described above suggest, I am advocating running tests for days.

Let's say five days into the testing a crash or other error occurs. How are you going to be able to reproduce the problem so that it can be fixed or re-run after a fix to prove the problem is solved? How can one remember five days worth random navigations? The way I chose to solve this problem is by writing recording a navigation log file, which records all the steps that were executed. This log is separate from the HTML or Text log created by Functional Test software itself. The navigation log is not intended to be used for logging pass/fail results or debugging the model. It contains only the information needed to re-trace the same steps that were previously executed (and presumably caused the problem). This log was implemented in XML although a normal text file would have worked fine. I chose XML because it's a good mechanism for adding additional log data in the future if that was needed. The play-back logic, by ignoring fields that it doesn't need, can remain independent of specific versions of the log file. This added flexibility is helpful when needing to run an older log file on newer builds of the application.

The sample source code writes this log to a file called "log.xml" that is written to the default directory where Functional Test is installed. Here is an example of a 3 transition log:

```
<?xml version="1.0" ?>
  <NavigationLog>
    <changeState>
      <transition>3</transition>
    </changeState>
    <changeState>
      <transition>1</transition>
    </changeState>
    <changeState>
      <transition>5</transition>
    </changeState>
  </NavigationLog>
```

The transitions are the logged as integers indicating the relative place in the `transitionList` class.

Verification Points

Verification points are the methods that you write to verify the back-end logic of a state or a screen. This is sometimes referred to as the "business logic". After navigating to a screen the model-test software will call and run the verification method specified in the `State` object. A class called `VerifyPage` holds all of these methods. For complex applications with lots of logic to check, each method declared in the `VerifyPage`

class should, in turn, refer to a separate class for each verification point. The example shows all of the verification logic in one for illustration purposes.

Beginning with your application:

Model-based testing, like classic test automation, does not come for free; it requires an up-front investment in resources to yield useful results. To sell this approach to management and others so that they make the proper investment, I recommend starting with a small, simple pilot. A good starting place might be to develop a simple model of a subset of your application. Pick 4 to 10 main states or screens and use the scripts you've already developed as verification point methods. Even if you don't have extensive verification point methods for every screen, it's probably worth running this simple model for a few hours or days. In addition, try running a resource monitor or a code coverage tool to check for memory leaks and other program behavior. If the application breaks, crashes, leaks memory or some other resource, you've found a bug that you otherwise might have passed on to your customers. This is an excellent way to demonstrate success to management.

Conclusion

Model-based testing is a good way to augment your existing test automation. Since the random nature of the navigation actually yields new test cases with each run you are likely to uncover more bugs than repeating your classic automation indefinitely. You can further increase the variety of combinations by running the same code on several computers simultaneously. This generates even more combinations while keeping the time in test constant. If your application is a browser based web application running against a single server, model-based testing is one way of simulating simultaneous users accessing one server.

A side benefit of this technique is that it provides an interesting, creative environment for Java programmers to specialize in the field of software test. Attracting and keeping good developers in test organizations can be a challenge. After writing simple code to test other's work for awhile the programming can become dry. Model-based testing however, is one way to challenge the Java programmer working in a test organization. The techniques described here can form the basis of fertile ground for programmer productivity.

References:

[Intelligent Test Automation](#) by Harry Robinson.

eclipse.org

[Thinking in Java](#) by Bruce Eckel

On line User guide for IBM Rational Functional Test for Java and Web
Portions of this article were first published by IBM developer Works at
<http://www.ibm.com/developerWorks/>.