# Unit Testing with Monkeys

**Karl-Mikael Björklid (bjorklid@jyu.fi)**
University of Jyväskylä
Department of Mathematical Information Technology

## Abstract

Unit testing has become an incresingly popular way of assuring the quality of program code. This is, no doubt, due to the popularity of the light weight engineering methodology Extreme Programming[1] and free test frameworks, like JUnit[2].

This paper argues that conventional unit testing frameworks do not offer proper support for systematically testing object-oriented units. A state machine model -based and more structured system is presented as a suggested improvement.

## 1 What's Wrong with the Conventional Approach?

Two aspects of a OO-unit[3] are of importance when considering how to test it: *state* and *behaviour*. Therefore, exhaustively testing an object-oriented unit means in essence that for all possible states of the unit the behavior of the object is asserted by ensuring that the unit handles all possible input correctly. Of course, exhaustively testing any non-trivial unit is in most cases would take too long a time to be practical. Therefore, the tester should select a number of signifigant states and a number of significant input sequences, by using for example domain analysis or equivalence class partitioning[1].

Conventionally, automatised unit testing is conducted by writing scripts, where the implementation under test (IUT) is brought into a specific state, wherein the functionality of the IUT is tested against the specification.

Unfortunately, conventional test scripts are often written in an ad hoc manner without analysing the true nature of the IUT. Thus, it is easy to miss a state whose functionality should be tested, or fail to test some aspect of the functionality in a specific state.

Furthermore, the conventional way of unit testing is relatively static, meaning that the computer does not participate in the testing in any other

[1]http://www.xprogramming.com
[2]http://www.junit.org
[3]In OO-domain a unit is essentially an object.

way than checking the output of the unit against some set of predefined values.

## 2 State Machine -Model Based Unit Testing

The state machine -model based unit testing approach requires that the tester develops a state machine model of the unit. The model should contain the states that are significant for testing, and transitions between those states. The transitions should test effectively all messages that can be used to get from one state to another.

### 2.1 An Example of A Test Model

Let's consider a simple example: a stack with limited maximum capacity (let's say that the stack can hold a maximum of ten objects). A simple model for this case would include three states: empty (stack contains no objects), full (stack contains the maximum amount of objects), and loaded (stack is neither empty or full). The empty- and full-categories contain only one state, but there are a number of real states that fall into the loaded-category. For simplicity, one real state is chosen to represent all the loaded-category-states. In this example, a stack in loaded state would contain four objects.

In this example, the stack has three methods that change the state of the stack: `push` (adds an object on top of the stack), `pop` (removes the topmost object of the stack), and `clear` (removes all objects from the stack). Given these methods and the set of states previously identified, six transitions are identified:

1. From *empty* to *loaded*: four `push`-calls.
2. From *loaded* to *full*: six `push`-calls.
3. From *loaded* to *empty*: four `pop`-calls.
4. From *loaded* to *empty*: one `clear`-call.
5. From *full* to *loaded*: six `push`-calls.
6. From *full* to *empty*: one `clear`-call.

## 2.2 Relevant Oracles

An oracle produces the expected results for a test case (Binder, 2000). This section describes a few oracle-patterns relevant for state machine model-based testing.

### 2.2.1 Solved Example Oracle

The solved example oracle relies on a set of predefined assertions, typically selected using equivalence partitioning and boundary value analysis. In the case of state machine model-based testing, this means that for each modeled state, there is a set of tests that ensure that the IUT works correctly. In essence, these tests ensure that the state invariant for each state hodlds. To avoid redundant test code, a class invariant test should also be written. The class invariant would be asserted after each transition.

### 2.2.2 Different but Equivalent Oracle

The different but equivalent oracle leverages the characteristics of the state machine model. Whenever two separate IUT-instances are brought into the same state, they should be logically equal, even if the state is reached through different transition paths.

Because it is typically very fast to move the IUT from one state to another, it is easy to explore a vast amount of transitions, and see if any of the message sequences break the implementation.

### 2.2.3 Smoke Testing

Somewhat trivial but nevertheless important aspect of model-oriented testing is the smoke test. A smoke test means that the IUT is exercised with a number of supposedly legal message sequences to see if an abnormal termination does occur at some point. The smoke test is passed simply when no abnormal condition is encountered.

Again, a smoke test made with a state machine model-based tester requires only specifying how the IUT is brought from one state to another (although exercising the methods that do not take part in making the state transitions certainly helps)

If the stack that was used as an example uses an array to contain its contents, a smoke test might, for example, reveal logic errors in referring to the indexes of the table (assuming the used programming language implementation can determine when the array index is out of bounds), or if the last possible push-operation does not add an element to the stack (the last pop-operation would then probably fail).

This kind of smoke testing may also discover memory leaks if a very long sequence of transitions is executed for a single instance of the system under test.

### 2.2.4 Built-in Oracles

Some languages – like Eiffel and Java (since version 1.4) – have direct support for an assertion facility that can be used to write inline assertions into code to ensure that the invariants hold true.

However, using inline assertions does not ensure that the bugs are actually found - the code must be first executed with the input(s) that reveal the bugs. A well-written state machine model-based tester gives the code a good workout, giving the inline assertions a chance to reveal the possible bugs.

## 2.3 Selecting Transition Paths

There are a number of ways to select the transition paths for the execution of state machine model -based tests. This section describes a few choices.

### 2.3.1 All Transitions

The computer chooses the paths so that all transitions get executed at least one time. If all transitions cannot be executed with one instance of the unit under test, the computer creates as many instances as necessary.

This method can be used in situations when other methods would take an unnecessarily long time to execute.

### 2.3.2 Random Paths

Given a seed for the random generator, a maximum path length, and an amount of repetitions, the computer randomly transitions an instance of the unit under test in accordance to the state machine model. When a dead-end state or the maximum path length is reached, the computer creates a new instance of the instance and starts again.

### 2.3.3 Transition Tree

Given a maximum depth, the computer generates a transition tree, and exeuctes all transitions all the way to the leaf nodes of that tree. In other words, the computer executes all different paths that are at maximum the given length (depth of the tree).

### 2.3.4 The Cyclic Test

While walking the state machine model, the computer chooses (if possible) transitions that do not lead to a dead-end. In other words, the computer executes only cyclic subpaths. The objective is to make a very large amount of transitions with one single instance of the unit under test. This is effective for smoke testing.

## 3 Monkey Unit

Monkey[4] Unit is a state machine model based unit testing framwork for Java currently under construction. It will be an extension to the popular JUnit-framework.

To define a Monkey Unit test suite, the test writer makes the following steps:

1. For each transition, a method is written that executes the method calls of the unit under test to bring it to the transition's end state

2. The state machine model is described by defining the source- and target- states for each of the transitions

3. The Factory Method-pattern[2] is used to define how to create a new instance of the unit under test. The factory is bound to its target state, that is, the state in which the unit under test will be after it has been created. There may be more than one factory.

4. Typically, the class and state invariant solved example tests are written and bound to their states.

To put it all together, the user provides a reference to the state machine model that is used for testing. The user also provides information on which types of oracles are to be used (the initial version of Monkey Unit will at least include explicit support for state/class invariant testing and different but equivalent -oracle).

The Monkey Unit suite is executed through one of the standard JUnit `TestRunners`. If a fault should occur, the Monkey Unit shall generate a test

[4]"Monkey" is a metaphor for the algorithms that select how the model is traversed and tested.

report describing (at least) what transition path was executed, what kind of test revealed the fault, and on what line.

One possible future use for Monkey Unit is to use it to test threaded units. It would probably mean just executing multiple Monkey Unit tests in separate threads.

## References

[1] Robert V. Binder, “Testing Object-Oriented Systems”, Addison–Wesley, 2000

[2] Eric Gamma et al, “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison–Wesley, 1995