Bilkent Center for Bioinformatics,
Bilkent University, TURKEY.

Document Topic: Testing in PATIKA
Document Type: Guidelines **Author(s):** Ugur Dogrusoz
Date Started: 06.06.2006

Testing in PATIKA

Software testing is the process, used to help identify the correctness, completeness, security and quality of developed computer software [1]. With that in mind, testing can never completely establish the correctness of arbitrary computer software. Still we must do our best in identifying these problems and fixing them.

1 Assertions

An **assertion** is a programming language construct that indicates an assumption on which the program is based. It takes the form of an expression which is intended to be true. If an assertion proves false, it indicates a possible bug in the program. This is called an "assertion failure." So assertions are the most basic form in which, testing can be performed.

In software engineering, it is commonly believed that the earlier a defect is found the cheaper it is to fix it. So, we should check for things that should hold as often as possible to ensure our objects in computer memory are “intact”. For instance, below we somehow (e.g. search) obtain a `BankAccount` object for processing, which should not be null:

```
BankAccount acct = null;

// ...
// Get a BankAccount object
// ...

// Check to ensure we have one
assert acct != null;
```

Another example is as follows; in the code segment below, we have a 2D point which should have non-negative x and y coordinates after a certain transformation. We make sure it is actually the case at this point by a simple assertion.

```
// Apply transformation to get rid of negative values
PrecisionPoint leftTop = new PrecisionPoint(left, top);
PrecisionPoint vLeftTop = trans.inverseTransformPoint(leftTop);

// Now coordinates should be non-negative
assert vLeftTop.preciseX >= 0.0 && vLeftTop.preciseY >= 0.0;
```

If we don’t have this assertion and the condition that should hold at this point, does not, then chances are this will go noticed at a much later and seemingly unrelated point during execution. So we should not hold back and assert as much as possible for things that are especially doubtful.

By default, in Java, assertions are disabled at runtime. One can enable them with the virtual machine parameter “–ea”.

2 Class Invariants

A **class invariant** is a type of an invariant that applies to every instance of a class at all times, except when an instance is in transition from one consistent state to another. A class invariant can specify the relationships among multiple attributes, and should be true before and after any method completes. For example, suppose you implement a balanced tree data structure of some sort. A class invariant might be that the tree is balanced and properly ordered. Or for a 2D rectangle class, the leftmost coordinate of any valid rectangle object should be less than or equal to its rightmost coordinate.

The assertion mechanism does not enforce any particular style for checking invariants. It is sometimes convenient, though, to combine the expressions that check required constraints into a single internal method that can be called by assertions. Continuing the balanced tree example, it might be appropriate to implement a private method that checks that the tree is indeed valid (i.e., balanced, etc.) as per the dictates of the data structure:

```
/**
 * This method returns true if this tree is valid (i.e. properly
 * balanced, etc.)
 */
private boolean check() {
    ...
}
```

Because this method checks constraints that should be true before and after any method completes, each public method and constructor of the class should contain the following line immediately prior to its return:

```
assert this.check();
```

3 Unit Testing

Testing is not closely integrated with development. This prevents you from measuring the progress of development - you can't tell when something starts working or when something stops working. A **unit test** is a procedure used to validate that a particular module (a non-trivial method of a class - anything other than a simple accessor or mutator - in case of OO programming) of source code is working properly.

Using **JUnit** you can cheaply and incrementally build a test suite that will help you measure your progress, spot unintended side effects, and focus your development efforts [2]. A good point to start learning more about JUnit is [2].

Of course, the amount of stability you receive from unit testing is highly dependent on the quality of the test cases you write. The following are some guidelines to think about when writing test cases:

- One test is infinitely better than no tests at all. One test ensures that your code compiles, links and can run.

- Do not think your unit tests as a one time code. It will be used and reused, and often will be more persistent than the implementations they test. So take them seriously.

- Whenever a bug is fixed, write one or more test cases to verify that the behavior remains fixed. Run all tests before a commit!

- Check boundary conditions heavily. If the parameter of a method expects values in a specific range, your tests should pass in values that lie across that range. For example, if an integer parameter can have values between 0 and 100 inclusive, three variants of your test might pass in the values 0, 50, and 100 respectively.

- Use negative tests to be sure your code responds to error conditions appropriately. Verify that your code behaves appropriately when it receives invalid or unexpected input values. Verify that it returns errors or throws exceptions when it should. You might be surprised to find that a test you expected to fail actually succeeds. For example, if an integer parameter to a method can accept values in the range 0 to 100 inclusive, you might create tests that pass in the values -1 and 101.

- Write tests that combine different code modules to implement some of the more complex behaviors of your application. While simple, isolated tests do provide value, stacked tests that exercise complex behaviors tend to catch many more problems. These kinds of tests simulate the behavior of your code under more realistic conditions, which leads to the discovery of more realistic problems. For example, in addition to just adding objects to an array, you could create the array, add several objects to it, remove a few of those objects using several different methods, and then make sure the number of remaining objects is correct.

- Try to use mock objects for unit testing. A mock object is a class that extends your class that was involved in testing but has some default values of behavior.

- Do not put anything in your tests that require user input. This quickly becomes an annoyance.

Latest versions of IDEs such as IDEA (configure through the JUnit tab under “Run | Edit configurations”) and Eclipse (“File | New | JUnit Test Case”) come with a JUnit plugin.

4 Example

A `Rectangle` class using assertions, class invariants, and unit tests has been implemented as a simple example (see sources under directory `JUnit/Example`).

5 Conclusion

So testing is a crucial part of software development. Thus, when writing code, we should use class invariants and assertions whenever possible. In addition, we should write unit tests which are to be run before committing any new work. In repositories we have separate directories for unit tests organized using the associated package of the original method/class under test (e.g. `patikapro1x\test\org\patika\pro\client\util\SubjectViewTest.java`).

References

[1] http://en.wikipedia.org/wiki/Software_testing, *Software testing*.

[2] http://junit.sourceforge.net/doc/testinfected/testing.htm, *JUnit Test Infected: Programmers Love Writing Tests*.

[3] http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html, *Programming with Assertions*.