# 6.170 Laboratory in Software Engineering Fall 2005 Problem Set 2: Designing a Comprehensive Blackbox Test Due: **Thursday**, September 29, 2005 at 1:00pm

## Handout PS2

Contents:

---

## Introduction

If you were to apply for an internship at Microsoft this summer, you would be required to apply as either a "Software Design Engineer" or a "Software Design Engineer in Test". There is no single job description title "Application Creator." This distinction is not unique to Microsoft, in fact, almost all software development companies use both designers and testers when creating applications. This is because the testing of an application is **as important** as the initial programming done to implement the application. As one adage goes: "If it hasn't been tested, it hasn't been implemented."

In industry, the developer who implements a module is almost always different than the person who writes the test suite for the module. This allows a more honest test of the specification. When a single developer plays both of these roles, the developer may write a test suite that unwittingly takes advantages of knowledge of the implementation, and thus fails to catch errors. Separating these roles solves this problem.

In this problem set, we play developer and you play tester. Your task will be to design a test suite for a module based on its specification alone. The module you will be testing, `BasicList`, is our own list implementation. Its specification is essentially a subset of **Java 1.4's** `java.util.List`, but BasicList does not implement `java.util.List` because some methods are missing. You are given the class file of `BasicList`, but not its source code. Note the use of Java 1.4's `List`, this implementation differs from Java 1.5's

`List` implementation. Java 1.4 does not have polymetric polymorphism (it was introduced in Java 1.5), so different errors can arise. Therefore, you may need to think more carefully when reading this specification to make sure you understand (and test for) all possible scenarios.

By doing this, you should learn how to design a compact, informative, and thorough blackbox test suite from a module's specification.

- By compact, we mean that your test suite should not include gratuitous tests. You should make it as small as you can, while still achieving good code coverage.
- By informative, we mean that once your test suite finds a bug in the module, it should give enough information about the bug so that someone else could identify and fix the bug in the module quickly.
- By thorough, we mean that if an implementation of the specification passes your test suite, then you can be reasonably sure that it will always adhere to its specification in practice.

---

## Preliminary Exercises (10 points)

Before you begin with `BasicList`, we would like you to answer a few short questions about how to write test cases. These questions should get you thinking in the right direction for testing `BasicList` later. Please answer the following questions in a file named **`exercises.txt`** and put it in the **`doc/`** directory.

1. Look at the specification for `Math.max(int a, int b)`. Identify the input subdomains of this method and list the combinations of `a` and `b` that you would test if you were to write a test suite for this method. Don't forget boundary cases.
2. Suppose a method `d(Object obj)` specifies that it throws an `IllegalArgumentException` whenever `obj` is `null`. Ben Bitdiddle writes the following JUnit code to test for this behavior:

```

  try {
    d(null);
  } catch (Exception e) {
    assertTrue("exception was thrown", true);
  }
```

   Unfortunately, this test does not verify that `d()` meets its specification. Explain why not and then write your own test that verifies that `d()` meets its spec.

---

## Problem: BasicListTest (90 points)

Using the specification for the BasicList class alone, design a JUnit suite `ps2.BasicListTest` that tests an implementation of `BasicList`. Use it to test the particular implementation of `BasicList` that we provide.

`BasicList` should act as specified; however, our implementation is faulty, and contains several bugs. Your test suite should reveal as many of them as possible. Summarize the bugs you found in a file called **`bugs.txt`** in the **`doc/`** directory. This file should discuss all the respects in which our implementation fails to meet the with specification.

Note that your test suite should not be tailored to the particular implementation of `BasicList` that we give you. It should not produce false alarms when applied to a correct implementation, and it should be capable of revealing different bugs in other implementations.

To evaluate your test suite we will run it on several implementations of `BasicList`. First, we will run it on the buggy implementation of `BasicList` that we provided you to see what bugs your test suite reveals. Next, we will run it on a correct implementation of the `BasicList` specification to ensure that your test suite validates correct code. Finally, we will run it on a number of different broken implementations of `BasicList` to see if your test suite covers the entire specification.

Note that the staff will grade your assignment by looking at the output of your JUnit test suite. This output includes the message that you provide in each `assertZZZ()` method as well as the name of the `test` method in which the failure occurred. Therefore, choose these messages and method names carefully so that the staff can tell if the failure that your test suite throws corresponds to the bug that we are looking for. For example, if we run your test suite on an implementation of `BasicList` where `size()` always returns `0` and we see a failure in your test suite in a method named `testSize()`, with a failure message that says `"size() did not return the correct value"`, then we can be confident that you successfully tested for the errant behavior that we wanted you to find.

## Mechanics

First, you must make sure that your checkout of lib6170 is up to date. This will ensure that the `/lib/ps2-lib.jar` file is found in your `CLASSPATH` when you go to execute JUnit. To do this, from the **Java** perspective right click on **lib6170**. Select **Team >> Update**. This will raise a window that indicates that **lib6170** is being updated. After this is done, you may continue. **Note:** There is no need to import `ps2.BasicList` directly. It is contained within `/lib/ps2-lib.jar` and will automatically be included in your `CLASSPATH` after you update `lib6170`.

Second, you will check out the **ps2** package from CVS as you normally do. There you will find an empty implementation of `ps2.BasicListTest` that extends `junit.framework.TestCase`. Fill in this class with test cases for `BasicList` that try to discover deviations from its specification.

If you do not remember how to run JUnit from Eclipse, then refer to Problem Set 0. Note that JUnit works by executing every method in your class with a signature of the form:

```
   public void testZZZ()
```

Thus, you can have other methods in your test suite whose names do not start with `test` that will not be invoked by JUnit. Conversely, every method that you want JUnit to run must start with `test`, take no parameters, and have `void` as its return type.

## Hints

Here are a number of hints and reminders that will help you create robust tests for this problem.

- You'll save a lot of time, and produce a much smaller and more effective test suite, by selecting your input subdomains first, and only then choosing test cases.
- Use appropriate names for your methods and descriptive text in your failure messages. Note that every `assert` method in `junit.framework.TestCase` has two sets of parameters: one that takes a `String message` as its first parameter and one that does not. Use the `String message` to describe the nature of the error that the test finds. This makes your test suite more useful for someone who is using it as a debugging tool, such as your TA. For an example of a test suite that uses these messages, refer back to `ps0.FibonacciTest` from problem set 0.
- When your JUnit test code reveals bugs in the supplied implementation of `BasicList`, failures will occur when running JUnit on it. This may seem annoying, and you may even decide to comment these tests out of your code. However, bear in mind that the staff will be running your test code on the version of `BasicList` that we give you, as well as on other broken implementations of `BasicList`. Your test code is expected find failures in any broken implementation. Thus, if you comment out tests in your code, then your test may not reveal problems with `BasicList` when the staff runs it. Do not make the mistake of removing tests from your test suite because then it will appear to the staff that your test does not test all of `BasicList`'s specification.
- Make sure that your test validates a working implementation of `BasicList`. For example, you may accidentally write the following:

  ```

    public void testFoo() {
       methodThatRevealsProblemWithFoo();
       assertEquals(true, false); // this will always fail!
    }
  ```

  When you run this test on an implementation where `foo()` is broken, then it will fail as expected. However, if you run it on a correct implementation of `foo()`, then it will also fail because `true` is not equal to `false`.

## Grading

The grading of this problem will be is separated into four categories.

- 15 points - for writing clear, well-organized and efficient-enough code that is not excessive
- 30 points - for finding all of the bugs in the version of `BasicList` that we give you and for explaining how your unit test exposes those bugs (`doc/bugs.txt`)
- 15 points - if your test validates a correct implementation of `BasicList`
- 30 points - if your test exposes bugs in other broken implementations of `BasicList`

---

## Errata

There are no known problems with the problem set thus far.

---

## Q & A

This section will list clarifications and answers to common questions about problem sets. We'll try to keep it as up-to-date as possible, so this should be the first place to look (after carefully rereading the problem set handout and the specifications) when you have a problem.