CSE 331 Software Design and Implementation

Handout C6: *A Guide to Testing*

Contents:

Testing increases our confidence that our code behaves correctly (i.e., meets the provided specifications). However, testing can only prove that errors **do** exist, not that a program is free from errors. In practice, testing is often the most practical and effective way to find bugs in your program.

Debugging is a distinct activity from testing. A separate document gives hints for debugging.

Types of Tests

Tests can be classified in various ways, according to:

- How much code is tested: unit tests vs. system tests
- The level of specifications being tested: specification tests vs. implementation tests
- Whether you know the implementation when you are writing your tests: black box tests vs. clear box tests

How much is tested

A **unit test** is a test for one component. If a unit test fails, you don't have to spend a lot of time localizing the defect, because you know exactly where the defect is: in the component being tested. Unit tests help you ensure that you are building your system out of reliable building blocks. Commonly, a unit test tests one method of one class. However, a test can be a unit test if it focuses its attention on one class, or one package - so long as that is smaller than the overall system being tested.

System tests or **integration tests** are tests of whether the system as a whole works - whether the (individually correct, unit-tested) modules fits together to achieve correct functionality. A system test usually involves more complicated operations than unit tests. An example is the practice of reading in a file that defines some operations, performing those operations, writing out a result file, and comparing the result file to a version (called the *goal file* or *golden file*) that has been hand-inspected and verified to be correct.

You should run unit tests first, and then system tests only when all the unit tests succeed. It is often a good idea to write system tests first, even before you know how you will modularize your implementation.

The level of specifications being tested

When you write a component, you write it to satisfy some externally-visible need, and it has behavior that clients can depend on. This is its specification. The tests that you deliver with your system help a client to have confidence that your implementation meets its public specification.

The implementation might actually satisfy a stronger specification.

- The implementation may be designed to strengthen some parts of the public specification. For example, a routine that is allowed to return a list in arbitrary order might always return the list in sorted order. Or, an implementation might have well-defined behavior for inputs that are outside the public precondition (`requires` clause).
- An implementation might have more components than the specification requires. For example, a class might have additional methods. Or, the implementation might have helper classes. Each helper class has its own specification, but the client is not aware of, nor can the client use or depend on, these helper classes.

The client cannot depend on extra properties of the implementation, and generally doesn't even know them! However, you might like to ensure that your implementation satisfies those properties, as a double-check that it meets your design. Such tests should not be exposed to the client - they are an internal debugging aid for the implementor.

Another way to think of this is that specifications have different levels of visibility. For example, a client may only care that files are read and written in a specific format. This forms one level of specification. In designing the program to read and write these files, engineers should write specifications for each public method in each class they use (i.e., via Javadoc-style code comments). This forms another level of specification, which is stronger than the client's specification. You should test your code at each level of specification.

In CSE 331, any tests on the level of the client's (the staff's) specification should be called by `SpecificationTests.java`. Any tests for specifications not handed out by the staff (the ones that you design) should go in `ImplementationTests.java`.

Here is a good rule of thumb: If your test could compile and run as a valid test against any other student's code, then it should go in `SpecificationTests.java`. Otherwise, it should go in `ImplementationTests.java`.

If the implementation changes, your implementation tests may become invalid. However, your specification tests will remain valid so long as the specification does not change. They remain valid, but you might want to add new black-box specification tests to ensure that you specification tests have good coverage of your implementation.

You should generally write specification tests first. Then, add implementation tests to cover important functionality that is not covered by your specification tests. Sometimes, a component does not need implementation tests because there isn't anything extra about the implementation that is not covered by the specification tests; that's fine, and you need not duplicate test functionality. Sometimes, a component does not have specification tests because it is not visible in the specification.

Do you know the implementation?

Black box tests

Black box tests are written without knowledge of anything except for a specification.

Their advantage over clear box tests is that they are unlikely to be biased by the same misunderstandings with respect to the specification that might have caused a bug in the implementation. In many circumstances, black box tests are written by people who are not the ones who write the implementation. Black box tests can be written before, or concurrently with, the implementation.

Black box tests should be written first, but they are typically not sufficient on their own. A black box test may not cover (that is, exercise or execute) all parts of an implementation, and therefore it may miss bugs.

They also fail to test specifics of the implementation where the implementation is stronger than the specification. (Doing so requires writing an implementation test have the disadvantage of not testing not being able to differentiate cases where only the implementation (and not the specification) treats inputs differently.

Clear box tests

Clear box tests are written with full knowledge of an implementation.

Clear box tests of the specification have the advantage of being able to test that special cases in the implementation conform to the specification, even if those cases are not particularly special according to the specification alone.

The disadvantage of clear box tests is that people are likely to make the same mistakes when writing the clear box tests as they made when writing the methods that are being tested. This can result in the tests passing when the specification is not actually being accurately followed. Also, if an implementation changes, then the white-box specification tests remain valid tests, but they may no longer provide good coverage.

Discussion

A black box test and a clear box test are testing the *same* specification. Each one is valid for any implementation of that specification. The only difference is the methodology you used for deciding the test inputs. It is perfectly fine to re-use clear box tests when your implementation changes. (Why throw away tests?) You probably need to add some additional clear box tests to ensure good coverage of the new implementation, however.

You should write black box tests first, and then add clear box tests as needed if there is implementation behavior that is not exercised by the black box tests.

Relating the ways to classify tests

The notions of unit vs. system testing, specification vs. implementation testing, and black box vs. clear box testing are **completely orthogonal**. You can have all 8 varieties of tests - though you probably won't explicitly segregate all those 8 varieties in your test suite.

Oftentimes but not always, unit tests are implementation tests and system tests are specification tests. However, it is possible to have a unit specification test (if the specification is sufficiently detailed), and to have a system implementation test (if your implementation has a stronger overall specification than the public documentation requires). Black-box and clear-box approaches for generating test input data are always appropriate, no matter the other details about your tests.

How to test a private method

Tests are written in a separate class from the code being tested - this separates distinct concerns, and it keeps the tests from cluttering the code. However, this makes it impossible to directly test private methods, which are not accessible outside the class in which they appear. (When the tests are in a separate package than the code being tested, a similar problem occurs for methods with protected or default accessibility. In JUnit 4 it's possible to test protected (but not private) methods of non-final classes by using subclassing.)

Suppose you want to test a private method. There isn't really a good way to do this. Four approaches are suggested by Bill Venners. (This is a rewording of his summary, for clarity.) The first one tests the private methods indirectly, and the last three test the private method directly.

- Devise tests of the public methods that exercise all the functionality of the private methods.
- Make the methods non-private, and write normal tests.
- Write the tests inside the class, where they have access to the private method.
- Use reflection to circumvent Java's access restrictions on private methods.

We recommend the first approach. It creates a specification test that will continue to be useful even if your implementation changes. Since the private method is designed for a particular, specific purpose, it should be possible to exercise all its functionality just via calls to the public methods. If there is more functionality in the private methods, then it is irrelevant to the behavior of the system, and it should perhaps be removed.

To use the third approach, you would write the tests in the same class as the method itself, in a method with greater visibility. For instance, to test private method `foo`, you would write a public method named `testFoo`. The specification of `testFoo` is the same as that of `checkRep` (another public method that tests private, inaccessible details of the class). Each of these takes no arguments and return no results, but throws an exception if any of its tests fail.

Other approaches are less desirable. Making the method public breaks modularity, and clients could come to depend on it, restricting the flexibility of the implementer. Using reflection to temporarily make the method public is ugly and hard to understand. Not testing the method is even worse, as bugs in a helper method may be much harder to debug when they are exposed only by incorrect behavior of a method that calls them.

If you choose to test the private methods directly, using one of the last three approaches, then be sure to put those in your implementation tests, not your specification tests.

Additionally, it's always a good idea to write assertions (including checkRep) within all your methods. Writing assertions is orthogonal to writing test cases.

Testing Methodology

Before you start writing code

- Use the specification to identify the abstract-value domain of each non-trivial public method: what is the set of objects that the method can be called on, and the set of allowed inputs?
- Partition each domain into "similar cases." For example, if the domain contains a number, does the method only operate on positive numbers, odd numbers, etc...?
- Write black box unit tests for each non-trivial public method in `SpecificationTests.java`.

As/after you write your implementation

- As one of the first things you implement, write a private `checkRep()` method and call it on entrance and exit from every non-trivial public non-constructor method (and on exit from all constructors). If the only fields of your class are primitives that are declared as `final`, then you can just call `checkRep()` after the constructor.
- Whenever reasonable, write clear box tests for both your specification and implementation in `ImplementationTests.java`.

Hints for testing

- Don't write tests in the test suite constructor; don't even invoke the constructors of the class-under-test.
- Write small tests. That way, if the test fails, you will have an idea of what was happening at the time.
- Choose good names for your tests, use the proper instance of the assert method, and write good messages. This will ensure that all the information you need to debug a test failure is present.
- Think carefully about each test. Don't write a specification test but accidentally call it an implementation test. Don't just use your solution as an oracle, but think carefully whether alternative solutions are correct (e.g., is there more than one shortest path for the given graph?).
- Remember that testing requires writing clear, concise, and targeted tests, not just coming up with an arbitrary number of random examples.

Test code reuse

Just as abstraction and re-use are good in your program, they are also good in your tests. We encourage you to re-use!

Just be careful that you do not end up obscuring the cause of an error. That could happen because you have no test that tests one particular method in isolation, so it is hard to isolate the code error (or the error in the test). If you build up your tests carefully, you can avoid this problem.

Back to Conceptual Info
Back to the CSE 331 home page.