

Lecture 11: Testing and Debugging

October 1, 2002

In this lecture, we'll look at the related problems of testing and debugging. Testing is the process of detecting bugs in a program; debugging finds and removes them.

11.1 Basic Principles

Testing is only one part of a more general process called *validation*. The purpose of validation is to uncover problems in a program and thereby increase your confidence in the program's correctness. Validation includes:

- Formal reasoning about a program, usually called *verification*. Verification constructs a formal proof that a program is correct, by showing that whenever the preconditions are satisfied, the program always produces a state in which the postconditions are true. Verification is tedious to do by hand, and automated tool support for verification is still an active area of research. Nevertheless, small, crucial pieces of a program may be formally verified, such as the scheduler in an operating system or the bytecode interpreter in a virtual machine.
- Informal reasoning. Having somebody else carefully read your code can be a good way to uncover bugs, much like having somebody else proofread an essay you have written. In industry, this practice goes by various names (with various degrees of formality): code reviews, code inspection, walkthroughs. *Pair programming* is an extreme form of this idea, where two programmers work together on a single computer, with one programmer typing and the other reading and thinking about the code being typed. You can also read your own code, although it is harder to catch your own mistakes.
- Testing — running the program on carefully selected inputs and checking the results.

Validation requires having the right attitude. Your goal is not to see the program work, but to make it fail. There's a subtle difference. It is all too tempting to treat code you've just written as a precious thing, a fragile eggshell, and test it very lightly just to see it work. You have to be brutal. A good tester wields a sledgehammer and beats the program everywhere it might be vulnerable, so that those vulnerabilities can be eliminated.

There are three basic principles of testing:

1. **Be systematic.** Haphazard testing is less likely to find bugs (unless the program is so buggy that a randomly-chosen input is more likely to fail than to succeed), and it doesn't increase our confidence in program correctness. On the other hand, exhaustive testing — running the program on all possible inputs — is usually impossible. Instead, test cases must be chosen carefully and systematically. Some approaches to choosing test cases are discussed below.
2. **Do it early and often.** Don't leave testing until the end, when you have a big pile of unvalidated code. Leaving testing until the end only makes debugging longer and more painful, because bugs may be anywhere in your code. It's far more pleasant to test your code as you develop it. One technique, *test-driven development*, carries this idea to its logical conclusion: you write tests before you even write any code. In other words, the development of a single module might proceed in this order:
 - (a) write a specification for the module;

- (b) write tests that exercise the specification;
 - (c) write the actual code.
3. **Automate it.** Nothing makes tests easier to run, and more likely to be run, than complete automation. A test driver should not be an interactive program that prompts you for inputs and prints out results for you to manually check. Instead, a test driver should invoke the module itself on fixed test cases and automatically check that the results are correct. The result of the test driver should be either “all tests OK” or “these tests failed: ...” A good testing framework, like JUnit, helps you build automated test suites. You can find links to more information about JUnit on the course web page.

11.2 Regression Testing

It’s very important to rerun your tests when you modify your code. This prevents your program from regressing — introducing other bugs when you fix new bugs or add new features. Running all your tests after every change is called *regression testing*.

For large systems, regression testing can take hours, even days. An area of current research interest is in determining which regression tests can be safely omitted after a given change. If you know which of your tests exercise which part of the code, you may be able to determine that a local change in one part of the code does not need all your regression tests run. But in 6.170, the programs we build are small enough (and our computers are fast enough) that it pays to run all your tests after every change.

11.3 Testing Strategy

How do you test a complex system with many modules? The best approach takes advantage of the modularity of your design: first test each module in isolation (*unit testing*), then combine the modules together to test a complete system or subsystem (*integration testing*).

To unit test a module, the module must be isolated from the rest of the system. The test driver acts a client of the module, exercising its methods on carefully selected test cases and checking their results. We will see in the next section how test cases should be chosen.

If a module has dependencies on other modules of the system, those dependencies may be replaced with *stubs* that provide only limited or canned functionality. A stub is particularly useful when the other module hasn’t been written yet, or when its behavior might be nondeterministic, such as a network communication package. Regression testing requires *repeatable* behavior. A well-designed network stub can be instructed by the test driver to simulate different kinds of network conditions (connection failures, missing hosts, high latency, or heavy congestion), so that a module that uses the network can be exercised repeatably under conditions.

You may sometimes choose not to write a stub — e.g., if the module is already written and tested, or if writing a stub sufficient for testing would be as complex as the module itself. Modules which are used only as data structures are often not worth stubbing out for this reason. Be aware, however, that if you *don’t* write a stub, you are no longer testing your module in complete isolation, and a failure uncovered by testing may be due to a bug not in the tested module but in an insufficiently-tested dependent module.

11.4 Choosing Test Cases

Exhaustive testing is clearly infeasible — you can’t pass all possible floating point arguments to `sqrt` and check that it returns the right result for each. Haphazard or random testing, on the other hand, is less likely to discover bugs. We want to pick a set of test cases that is small enough to run quickly, yet large enough to validate the program.

To do this, we divide the input space into *subdomains*, each consisting of a set of inputs. The subdomains completely cover the input space, so that every input lies in at least one subdomain. Then we choose one test case from each subdomain.

The idea behind subdomains is to partition the input space into sets of similar inputs. Then we use one representative of each set. This approach makes the best use of limited testing resources by choosing dissimilar test cases, and forcing the testing to explore parts of the input space that random testing might not reach.

Ideally, the subdomains we choose should be *revealing*, which means that the program either fails or succeeds on all inputs in the subdomain. If all the subdomains are revealing, then our test suite is guaranteed not to overlook any failures.

In practice, since we don't know *a priori* where the program succeeds or fails, we must fall back on heuristics to choose subdomains that are likely to be revealing. Two common heuristics are *black box testing* and *glass box testing*.

11.4.1 Black Box Tests

Black box tests are generated by examining only the specification of a method or abstract data type. Hence the name: the module is treated as a black box whose internal structure (code) is not relevant. In test-driven development, black box tests are written after the spec is created but before any code has been written.

To generate black-box test cases, examine the specification to find subdomains that produce different behavior. For example, consider the specification for `sqrt`:

```
public double sqrt (double x)
// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
```

Two subdomains immediately leap out: $x < 0$, since an exception is thrown, $x \geq 0$, since the method returns normally. We should also create a subdomain at the *boundary* of these two subdomains, since bugs often appear at discontinuities, so we would also add a test case for $x = 0$.

Other subdomains for `sqrt` are more subtle:

- we might test perfect squares (where `sqrt` returns an integer value) separately from other numbers (where `sqrt` returns a non-integer);
- we might test cases where $x < \text{sqrt}(x)$ separately from cases where $x > \text{sqrt}(x)$, along with the boundary case $x = \text{sqrt}(x)$. The former subdomain is $0 < x < 1$, the latter subdomain is $x > 1$, and the boundary cases are $x = 0$ and $x = 1$.

Combining all these subdomains might give us the following black-box test cases:

- -1 (for the subdomain $x < 0$)
- 0 (for the boundary case)
- 0.5 (for both $x > \text{sqrt}(x)$ and `sqrt(x)` non-integer)
- 1 (another boundary case)
- 4 (for `sqrt(x)` integer)

You can draw two lessons from this example. First, subdomains are not always obvious from the spec; you should think carefully about the different kinds of behavior that the method could have, and which behaviors might reveal bugs. Second, you can sometimes cover two subdomains with the same test case, as we did with 0.5.

Boundary cases are particularly important test cases. Here are some common boundary cases:

- 0, 1
- smallest, largest

- empty (string, array, list, etc.)
- null
- aliased parameters: two parameters pointing to the same object. For example, suppose you're testing an array copy method that copies part of one array into part of another array. You'd want to have at least one test case where the source and destination arrays were the same.

11.4.2 Glass Box Tests

Black box testing is good for checking that a module's implementation satisfies its specification. But black box tests are not sufficient in general to validate a program.

Consider a method `isPrime`, which returns true if its integer argument is prime. Black box testing would suggest testing this method on a prime number, say 5, and a composite number, say 4. But suppose `isPrime` is implemented with a table lookup for small integers:

```
int isPrime (int x) {
    if (x < 100) look up in a table
    else compute answer
}
```

If we tested this `isPrime` only on 4 and 5, we would only test canned answers from the table, and never actually exercise the code that computes the answer.

In glass box testing, you examine the code of the module or method to discover additional subdomains. In this case, `isPrime` should also be tested with $x < 100$, $x > 100$, and (the boundary case) $x = 100$.

A standard approach to glass box testing is to add tests until the test suite achieves adequate *statement coverage*: so that every statement in the program is executed by at least one test case. In the `isPrime` example, we had inadequate statement coverage from the black box tests alone, since they never caused the "compute answer" code to be executed.

In practice, statement coverage is usually measured by a *code coverage tool*, which instruments your program to count the number of times each statement is run by your test suite. With such a tool, glass box testing is easy; you just measure the coverage of your black box tests, and add more test cases until all important statements are logged as executed. (Note that you can't achieve 100% statement coverage when your program includes defensive code, like always-false assertions or exception handling, that is included for safety but should never be executed.) Code coverage tools for Java exist, but we won't require you to learn a new tool for 6.170. It's sufficient to consider your test cases carefully enough that you can make an argument that you have achieved reasonable coverage.

Statement coverage is not the only kind of coverage, and in fact is not always sufficient to evaluate a test suite. Consider the following buggy program:

```
int min (int a, int b) {
    int r = a;
    if (a <= b)
        r = a;
    return a;
}
```

For this code, a single test case with $a < b$ would achieve 100% statement coverage, but it would fail to reveal the bug in the program, which reveals itself only when the if branch is *not* executed.

To handle this problem, we can turn to a stronger notion of coverage called *decision coverage*, in which the test suite must cause every edge in the control flow graph to be executed — roughly, every conditional branch and loop test must be executed in both directions.

There are still stronger forms of coverage. *Condition coverage* requires that the individual boolean expressions tested in a conditional evaluate both to true and false, so

```
if (a.isEmpty() && b.isEmpty())
```

would be tested for all four possible combinations of a or b being empty. *Path-complete coverage* requires that all paths through the program be tested by the test suite. Stronger forms of coverage may be used for testing safety-critical code, such as avionics.

11.5 Debugging

The best approach to debugging is to *avoid it*. Many of the design techniques we have studied in this course can reduce the costs of debugging by producing correct programs from the outset:

- **Modularity.** Divide your program into small modules, each of which is easy to understand and captures a single concern. Use abstract data types accessed through a well-defined interface. Avoid rep exposure, so that an ADT's rep can be changed only by the ADT's code.
- **Specification.** Write specifications for all your modules, so that an explicit, well-defined contract exists between the module and its clients.

Other techniques significantly reduce debugging by localizing bugs to a small part of your program:

- **Assertions.** Assertions help catch bugs early, closer to the original cause of the bug, before the failure has a chance to contaminate further computation.
- **Unit testing.** When you test a module in isolation, you can be confident that any bug you find must be in the module you're testing (unless it's in the test driver).
- **Regression testing.** Run all your regression tests as often as possible while you are changing code. When a regression test fails, chances are the mistake can be found in the code you just changed.

When a bug is localized to a small amount of code, such as a single method or module, it can usually be found simply by studying the program text and imagining how it could have produced the incorrect behavior.

Sometimes you have no choice but to debug, however — particularly when the bug is found by integration testing or by a user, in which case it may be hard to localize it to a particular module. For those situations, we can suggest some strategies for more effective debugging.

The first step is finding a small, repeatable test case that produces the failure. If the bug was found by regression testing, then you're in luck; you already have a failing test case in your test suite. If the bug was reported by a user, it may take some effort to reproduce the bug. For graphical user interfaces and multithreaded programs, a bug may be hard to reproduce consistently if it depends on timing of events or thread execution. Nevertheless, any effort you put into making the test case small and repeatable will pay off, because you'll have to run it over and over while you search for the bug and develop a fix for it. Furthermore, after you've successfully fixed the bug, you'll want to add the test case to your regression test suite, so that the bug never crops up again. Once you have a test case for the bug, making this test work becomes your goal.

The second step is understanding the location and cause of the bug. For this, you can use the scientific method:

- **Study the data.** Look at the test input that causes the bug, and the incorrect results, failed assertions, and stack traces that result from it.
- **Hypothesize.** Propose a hypothesis, consistent with all the data, about where the bug might be, or where it *cannot* be. It's good to make this hypothesis general at first. Here's an example. You're developing a web browser, and a user has found that displaying a certain web page causes the browser to crash. You might hypothesize that the bug is *not* in the networking code that fetches the page from the server, but in one of the modules that parses the web page or displays it.

- **Experiment.** Devise an experiment that tests your hypothesis. The experiment might be a different test case. In our web browser example, you might test your hypothesis by downloading the page to disk and loading it from a disk file instead of over the network. Another experiment inserts probes in the running program — print statements, assertions, or debugger breakpoints. It's tempting to try to insert *fixes* to the hypothesized bug, instead of mere probes. This is almost always the wrong thing to do, because your fixes may just mask the true bug. For example, if you're getting a `NullPointerException`, try to understand what's going on first; don't just insert a little test that avoids the exception without fixing the real problem.
- **Repeat.** Add the data you collected from your experiment to what you knew before, and make a fresh hypothesis. Debugging is a search process, and you can sometimes use *binary search* to speed up the process. For example, in the web browser, the web page might flow through 10 modules before the program crashes. To do a binary search, you would divide this workflow in half, guessing that the bug is found somewhere in the first 5 modules, and insert probes after the fifth to check its results. From the results of that experiment, you would further divide in half.

When making your hypothesis, you may want to keep in mind that different parts of the system have different likelihoods of failure. For example, old, well-tested code is probably more trustworthy than code recently added. Java library code is probably more trustworthy than yours (except where it depends on your code for correct behavior, as `HashMap` depends on your classes properly implementing the object contract). The Java compiler and runtime, operating system platform, and hardware are increasingly more trustworthy, because they are more tried and tested. You should trust these lower levels until you've found good reason not to.

A few more pieces of good advice for finding bugs:

- Make sure your object files correspond to your current source. It may help to delete all your `*.class` files and recompile everything.
- Swap components. If you have another implementation of a module that satisfies the same interface, and you suspect the module, you may try swapping in the alternative. For example, if you suspected `java.util.ArrayList`, you could swap in `java.util.LinkedList` instead. If you suspect the Java runtime, run with a different version of Java. If you suspect the operating system, run your program on a different OS. If you suspect the hardware, run on a different machine. You can waste a lot of time swapping unfailling components, however, so don't do this unless you have good reason to suspect a component.
- Get help. It often helps to explain your problem to someone else, even if the person you're talking to has no idea what you're talking about. Lab assistants and fellow 6.170 students usually *do* know what you're talking about, so they're even better.
- Sleep on it. If you're too tired, you won't be an effective debugger. Trade latency for efficiency.

Once you've found the bug and understand its cause, the third step is to devise a fix for it. Avoid the temptation to slap a patch on it and move on. Ask yourself whether the bug was a coding error, like a misspelled variable or interchanged method parameters, or a design error, like an underspecified or insufficient interface. Design errors may suggest that you step back and revisit your design, or at the very least consider all the other clients of the failing interface to see if they suffer from the bug too. Think also whether the bug has any relatives. If I just found a divide-by-zero error here, did I do that anywhere else in the code? Also consider what effects your fix will have. Will it break any other code?

Finally, after you have applied your fix, add the bug's test case to your regression test suite, and run all the tests to assure yourself that (a) the bug is fixed, and (b) no new bugs have been introduced.