# Reading 3: Testing

## Validation

Testing is an example 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. 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, or the filesystem in an operating system.
- **Code review.** Having somebody else carefully read your code, and reason informally about it, can be a good way to uncover bugs. It's much like having somebody else proofread an essay you have written. We'll talk more about code review in the next reading.
- **Testing**. Running the program on carefully selected inputs and checking the results.

Even with the best validation, it's very hard to achieve perfect quality in software. Here are some typical *residual defect rates* (bugs left over after the software has shipped) per kloc (one thousand lines of source code):

- 1 - 10 defects/kloc: Typical industry software.
- 0.1 - 1 defects/kloc: High-quality validation. The Java libraries might achieve this level of correctness.
- 0.01 - 0.1 defects/kloc: The very best, safety-critical validation. NASA and companies like Praxis can achieve this level.

This can be discouraging for large systems. For example, if you have shipped a million lines of typical industry source code (1 defect/kloc), it means you missed 1000 bugs!

## Why Software Testing is Hard

Here are some approaches that unfortunately don't work well in the world of software.

**Exhaustive testing** is infeasible. The space of possible test cases is generally too big to cover exhaustively. Imagine exhaustively testing a 32-bit floating-point multiply operation, a*b. There are 2^64 test cases!

**Haphazard testing** ("just try it and see if it works") is less likely to find bugs, unless the program is so buggy that an arbitrarily-chosen input is more likely to fail than to succeed. It also doesn't increase our confidence in program correctness.

**Random or statistical testing** doesn't work well for software. Other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer the defect rate for the whole production lot. Physical systems can use many tricks to speed up time, like opening a refrigerator 1000 times in 24 hours instead of 10 years. These tricks give known failure rates (e.g. mean lifetime of a hard drive), but they assume continuity or uniformity across the space of defects. This is true for physical artifacts.

But it's not true for software. Software behavior varies discontinuously and discretely across the space of possible inputs. The system may seem to work fine across a broad range of inputs, and then abruptly fail at a single boundary point. The famous Pentium division bug affected approximately 1 in 9 billion divisions. Stack overflows, out of memory errors, and numeric overflow bugs tend to happen abruptly, and always in the same way, not with probabilistic variation. That's different from physical systems, where there is often visible evidence that the system is approaching a failure point (cracks in a bridge) or failures are distributed probabilistically near the failure point (so that statistical testing will observe some failures even before the point is reached).

Instead, test cases must be chosen carefully and systematically, and that's what we'll look at next.

## Putting on Your Testing Hat

Testing requires having the right attitude. When you're coding, your goal is to make the program work, but as a tester, you want to **make it fail**.

That's a subtle but important 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.

Instead, 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.

## Test-first Programming

Test 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.

In test-first-programming, you write tests before you even write any code. The development of a single function proceeds in this order:

1. Write a specification for the function.
2. Write tests that exercise the specification.
3. Write the actual code. Once your code passes the tests you wrote, you're done.

The **specification** describes the input and output behavior of the function. It gives the types of the parameters and any additional constraints on them (e.g. sqrt's parameter must be nonnegative). It also gives the type of the return value and how the return value relates to the inputs. You've already seen and used specifications on your problem sets in this class. In code, the specification consists of the method signature and the comment above it that describes what it does. We'll have much more to say about specifications a few classes from now.

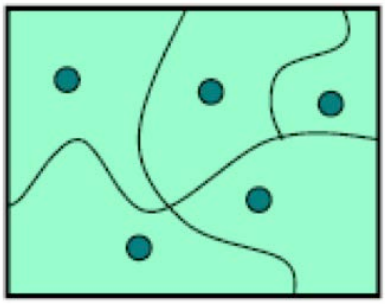
Writing tests first is a good way to understand the specification. The specification can be buggy, too — incorrect, incomplete, ambiguous, missing corner cases. Trying to write tests can uncover these problems early, before you've wasted time writing an implementation of a buggy spec.

## Choosing Test Cases by Partitioning

Creating a good test suite is a challenging and interesting design problem. 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. Taken together 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, and that's our test suite.

The idea behind subdomains is to partition the input space into sets of similar inputs on which the program has similar behavior. 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.



### Example: BigInteger.multiply()

Let's look at an example. BigInteger is a class built into the Java library that can represent integers of any size, unlike the primitive types int and long that have only limited ranges. BigInteger has a method multiply that multiplies two BigInteger values together:

```
/**
 * @param val  another BigIntger
 * @return a BigInteger whose value is (this * val).
 */
public BigInteger multiply(BigInteger val)
```

For example, here's how it might be used:

```
BigInteger a = ...;
BigInteger b = ...;
BigInteger ab = a.multiply(b);
```

This example shows that even though only one parameter is explicitly shown in the method's declaration, multiply is actually a function of *two* arguments: the object you're calling the method on (a in the example above), and the parameter that you're passing in the parentheses (b in this example). In Python, the object receiving the method call would be explicitly named as a parameter called self in the method declaration. In Java, you don't mention the receiving object in the parameters, and it's called this instead of self.

So we should think of multiply as a function taking two inputs, each of type BigInteger, and producing one output of type BigInteger:

**multiply : BigInteger × BigInteger → BigInteger**

So we have a two-dimensional input space, consisting of all the pairs of integers (a,b). Now let's partition it. Thinking about how multiplication works, we might start with these partitions:

- a and b are both positive
- a and b are both negative
- a is positive, b is negative

- a is negative, b is positive

There are also some special cases for multiplication that we should check: 0, 1, and -1.

- a or b is 0, 1, or -1

Finally, as a suspicious tester trying to find bugs, we might suspect that the implementor of BigInteger might try to make it faster by using int or long internally when possible, and only fall back to an expensive general representation (like a list of digits) when the value is too big. So we should definitely also try integers that are very big, bigger than the biggest long.

- a or b is small
- the absolute value of a or b is bigger than Long.MAX_VALUE, the biggest possible primitive integer in Java, which is roughly 2^63.

Let's bring all these observations together into a straightforward partition of the whole (a,b) space. We'll choose a and bindependently from:

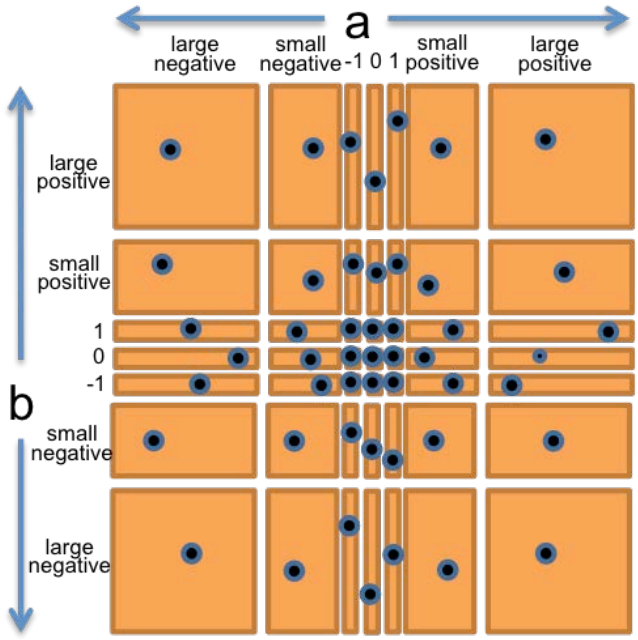
- 0
- 1
- -1
- small positive integer
- small negative integer
- huge positive integer
- huge negative integer

So this will produce 7 × 7 = 49 partitions that completely cover the space of pairs of integers.

To produce the test suite, we would pick an arbitrary pair (a,b) from each square of the grid, for example:

- (a,b) = (-3, 25) to cover (small negative, small positive)
- (a,b) = (0, 30) to cover (0, small positive)
- (a,b) = (2^100, 1) to cover (large positive, 1)
- etc.

The figure below shows how the two-dimensional (a,b) space is divided by this partition, and the points are test cases that we might choose to completely cover the partition.




**Example: max()**

Let's look at another example from the Java library: the integer max() function, found in the Math class.

```
/**
 * @param a  an argument
 * @param b  another argument
 * @return the larger of a and b.
 */
public static int max(int a, int b)
```

Mathematically, this method is a function of the following type:

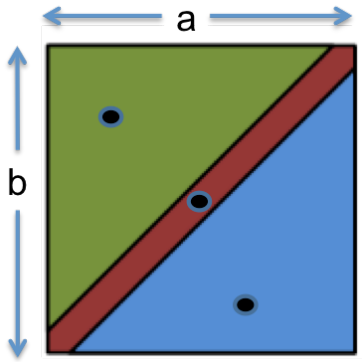
**max : int × int → int**

From the specification, it makes sense to partition this function as:

- a < b
- a = b
- a > b

Our test suite might then be:

- (a, b) = (1, 2) to cover a < b
- (a, b) = (9, 9) to cover a = b
- (a, b) = (-5, -6) to cover a > b




## Include Boundaries in the Partition

Bugs often occur at *boundaries* between subdomains. Some examples:

- 0 is a boundary between positive numbers and negative numbers
- the maximum and minimum values of numeric types, like int and double
- emptiness (the empty string, empty list, empty array) for collection types
- the first and last element of a collection

Why do bugs often happen at boundaries? One reason is that programmers often make **off-by-one mistakes** (like writing <=instead of <, or initializing a counter to 0 instead of 1). Another is that some boundaries may need to be handled as special cases in the code. Another is that boundaries may be places of discontinuity in the code's behavior. When an int variable grows beyond its maximum positive value, for example, it abruptly becomes a negative number.

It's important to include boundaries as subdomains in your partition, so that you're choosing an input from the boundary.

Let's redo **max : int × int → int**.

Partition into:

- *relationship between a and b*
    - a < b
    - a = b
    - a > b
- *value of a*
    - a = 0
    - a < 0
    - a > 0
    - a = minimum integer
    - a = maximum integer
- *value of b*
    - b = 0
    - b < 0
    - b > 0
    - b = minimum integer
    - b = maximum integer

Now let's pick test values that cover all these classes:

- (1, 2) covers a < b, a > 0, b > 0
- (-1, -3) covers a > b, a < 0, b < 0
- (0, 0) covers a = b, a = 0, b = 0
- (Integer.MIN_VALUE, Integer.MAX_VALUE) covers a < b, a = minint, b = maxint
- (Integer.MAX_VALUE, Integer.MIN_VALUE) covers a > b, a = maxint, b = minint

## Two Extremes for Covering the Partition

After partitioning the input space, we can choose how exhaustive we want the test suite to be:

- **Full Cartesian product**.
  Every legal combination of the partition dimensions is covered by one test case. This is what we did for the multiply example, and it gave us 7 × 7 = 49 test cases. For the max example that included boundaries, which has three dimensions with 3 parts, 5 parts, and 5 parts respectively, it would mean up to 3 × 5 × 5 = 75 test cases. In practice not all of these combinations are possible, however. For example, there's no way to cover the combination a < b, a=0, b=0, because a can't be simultaneously less than zero and equal to zero.

- **Cover each part.**
  Every part of each dimension is covered by at least one test case, but not necessarily every combination. With this approach, the test suite for max might be as small as 5 test cases if carefully chosen. That's the approach we took above, which allowed us to choose 5 test cases.

Often we strike some compromise between these two extremes, based on human judgement and caution, and influenced by whitebox testing and code coverage tools, which we look at next.

## Blackbox and Whitebox Testing

Recall from above that the *specification* is the description of the function's behavior — the types of parameters, type of return value, and constraints and relationships between them.

**Blackbox testing** means choosing test cases only from the specification, not the implementation of the function. That's what we've been doing in our examples so far. We partitioned and looked for boundaries in multiply and max without looking at the actual code for these functions.

**Whitebox testing** (also called glass box testing) means choosing test cases with knowledge of how the function is actually implemented. For example, if the implementation selects different algorithms depending on the input, then you should partition according to those domains. If the implementation keeps an internal cache that remembers the answers to previous inputs, then you should test repeated inputs.

When doing whitebox testing, you must take care that your test cases don't *require* specific implementation behavior that isn't specifically called for by the spec. For example, if the spec says "throws an exception if the input is poorly formatted," then your test shouldn't check *specifically* for a NullPointerException just because that's what the current implementation does. The specification in this case allows *any* exception to be thrown, so your test case should likewise be general to preserve the implementor's freedom. We'll have much more to say about this in the class on specs.

## Coverage

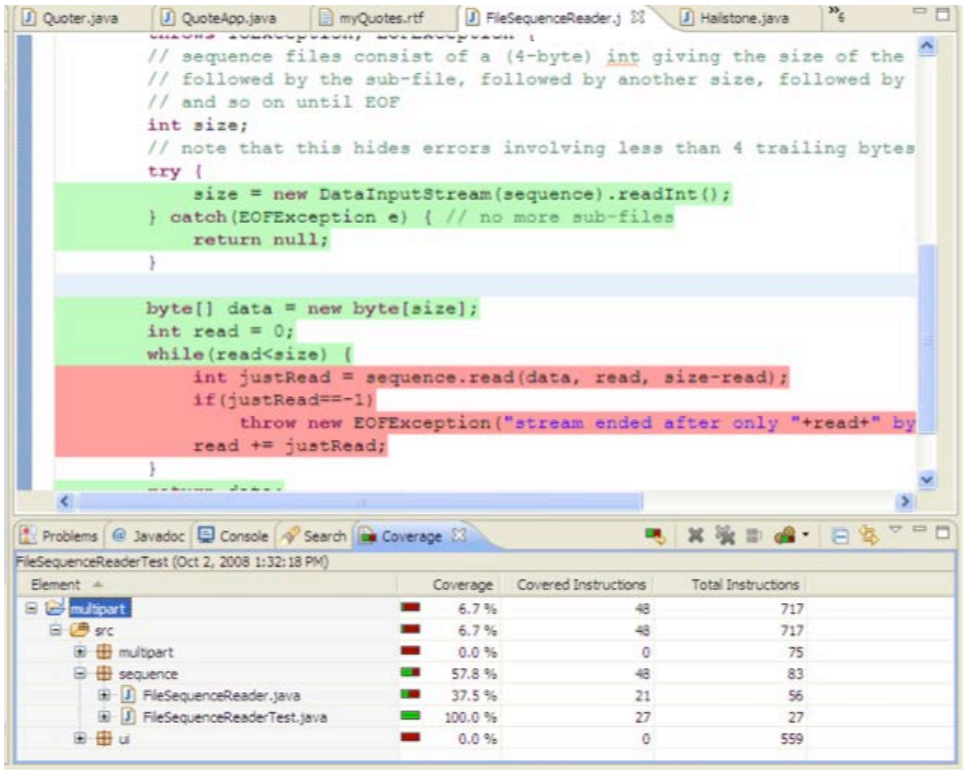
One way to judge a test suite is to ask how thoroughly it exercises the program. This notion is called *coverage*. Here are three common kinds of coverage:

- **Statement coverage**: is every statement run by some test case?
- **Branch coverage**: for every if or while statement in the program, are both the true and the false direction taken by some test case?
- **Path coverage**: is every possible combination of branches — every path through the program — taken by some test case?

Branch coverage is stronger (requires more tests to achieve) than statement coverage, and path coverage is stronger than branch coverage. In industry, 100% statement coverage is a common goal, but even that is rarely achieved due to unreachable defensive code (like "should never get here" assertions). 100% branch coverage is highly desirable, and safety critical industry code has even more arduous criteria (e.g., "MCDC," modified decision/condition coverage). Unfortunately 100% path coverage is infeasible, requiring exponential-size test suites to achieve.

A standard approach to testing is to add tests until the test suite achieves adequate statement coverage: i.e., so that every reachable statement in the program is executed by at least one test case. In practice, statement coverage is usually measured by a code coverage tool, which counts the number of times each statement is run by your test suite. With such a tool, white 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.

A good code coverage tool for Eclipse is EclEmma, shown below.




Notice how lines that have been executed by the test suite are colored green, and lines not yet covered are red. If you saw this result from your coverage tool, your next step would be to come up with a test case that causes the body of the while loop to execute, and add it to your test suite so that the red lines become green.

## Unit Testing and Stubs

A well-tested program will have tests for every individual module (where a module is a method or a class) that it contains. A test that tests an individual module, in isolation if possible, is called a **unit test**. Testing modules in isolation leads to much easier debugging. When a unit test for a module fails, you can be more confident that the bug is found in that module, rather than anywhere in the program.

The opposite of a unit test is an **integration test**, which tests a combination of modules, or even the entire program. If all you have are integration tests, then when a test fails, you have to hunt for the bug. It might be anywhere in the program. Integration tests are still important, because a program can fail at the connections between modules. For example, one module may be expecting different inputs than it's actually getting from another module. But if you have a thorough set of unit tests that give you confidence in the correctness of individual modules, then you'll have much less searching to do to find the bug.

Suppose you're building a web search engine. Two of your modules might be getWebPage(), which downloads web pages, and extractWords(), which splits a page into its component words:

```
/** @return the contents of the web page downloaded from url
 */
public static String getWebPage(URL url) {...}

/** @return the words in string s, in the order they appear, where a word is a contiguous
sequence of non-whitespace and non-punctuation characters
 */
public static List<String> extractWords(String s) { ... }
```

These methods might be used by another module makeIndex() as part of the web crawler that makes the search engine's index:

```
/** @return an index mapping a word to the set of URLs containing that word, for all webpages
in the input set
 */
public static Map<String, Set<URL>> makeIndex(Set<URL> urls) {
    ...
    for (URL url : urls) {
        String page = getWebPage(url);
        List<String> words = extractWords(page);
        ...
    }
    …
}
```

In our test suite, we would want:

- unit tests just for getWebPage() that test it on various URLs
- unit tests just for extractWords() that test it on various strings

- unit tests for makeIndex() that test it on various sets of URLs

One mistake that programmers sometimes make is writing test cases for extractWords() in such a way that the test cases depend on getWebPage() to be correct. It's better to think about and test extractWords() in isolation, and partition it. Using test partitions that involve web page content might be reasonable, because that's how extractWords() is actually used in the program. But don't actually call getWebPage() from the test case, because getWebPage() may be buggy! Instead, store web page content as a literal string, and pass it directly to extractWords(). That way you're writing an isolated unit test, and if it fails, you can be more confident that the bug is in the module it's actually testing, extractWords().

Note that the unit tests for makeIndex() can't easily be isolated in this way. When a test case calls makeIndex(), it is testing the correctness of not only the code inside makeIndex(), but also all the methods called by makeIndex(). If the test fails, the bug might be in any of those methods. That's why we want separate tests for getWebPage() and extractWords(), to increase our confidence in those modules individually and localize the problem to the makeIndex() code that connects them together.

Isolating a higher-level module like makeIndex() is possible if we write **stub** versions of the modules that it calls. For example, a stub for getWebPage() wouldn't access the internet at all, but instead would return mock web page content no matter what URL was passed to it. A stub for a class is often called a **mock object**. Stubs are an important technique when building large systems, but we will generally not use them in 6.005.

## Automated Testing and Regression Testing

Nothing makes tests easier to run, and more likely to be run, than complete automation. **Automated testing** means running the tests and checking their results automatically. 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.

Note that automated testing frameworks like JUnit make it easy to run the tests, but you still have to come up with good test cases yourself. *Automatic test generation* is a hard problem, still a subject of active computer science research.

Once you have test automation, 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**.

Whenever you find and fix a bug, take the input that elicited the bug and add it to your automated test suite as a test case. This kind of test case is called a *regression test*. This helps to populate your test suite

with good test cases. Remember that a test is good if it elicits a bug — and every regression test did in one version of your code! Saving regression tests also protects against reversions that reintroduce the bug. The bug may be an easy error to make, since it happened once already.

This idea also leads to *test-first debugging*. When a bug arises, immediately write a test case for it that elicits it, and immediately add it to your test suite. Once you find and fix the bug, all your test cases will be passing, and you'll be done with debugging and have a regression test for that bug.

In practice, these two ideas, automated testing and regression testing, are almost always used in combination.

Regression testing is only practical if the tests can be run often, automatically. Conversely, if you already have automated testing in place for your project, then you might as well use it to prevent regressions. So **automated regression testing** is a best-practice of modern software engineering.

## Summary

In this reading, we saw these ideas:

- Test-first programming. Write tests before you write code.
- Partitioning and boundaries for choosing test cases systematically.
- White box testing and statement coverage for filling out a test suite.
- Unit-testing each module, in isolation as much as possible.
- Automated regression testing to keep bugs from coming back.

The topics of today's reading connect to our three key properties of good software as follows:

- **Safe from bugs.** Testing is about finding bugs in your code, and test-first programming is about finding them as early as possible, immediately after you introduced them.
- **Easy to understand.** Testing doesn't help with this as much as code review does.
- **Ready for change.** Readiness for change was considered by writing tests that only depend on behavior in the spec. We also talked about automated regression testing, which helps keep bugs from coming back when changes are made to code.