

Testing Specifications

6.170 Lecture 5

Spring 2004

No amount of experimentation can ever prove me right; a single experiment can prove me wrong.

Albert Einstein

The goal of testing is to ensure that a module functions correctly, and fulfils its intended purpose. Testing cannot prove the absence of errors, only their presence. However, thorough testing can increase confidence in software, under the assumption that most errors would have been exposed by some test. In this lecture, we give a brief overview of the main techniques available for software verification. For the moment we concentrate on *black-box testing*, where a module is tested based only on knowledge of its specification, and not its implementation – we will see in a later lecture how to exploit knowledge of a module’s implementation for so-called *clear-box* testing.

1 Software verification techniques

Here are three techniques for verifying that modules do what they should:

1. **Prove correctness** – This approach requires writing exhaustive, precise formal specifications (preconditions, postconditions, and usually loop invariants), then proving that these specifications are satisfied by the code and its environment. The proof can be performed by hand, but for non-trivial modules, the proofs are long and tedious and humans are prone to errors, so theorem-proving software is usually used instead. It often requires human direction rather than being fully automated, and some module properties (such as many involving pointers and references) are currently beyond the state of the art in software analysis.

While any result is guaranteed to hold for any possible execution, correctness proofs are not cost-effective because of the difficulty of writing and proving specifications. Additionally, this technique only puts off a problem rather than fully solving it: how do you verify that the formal, complicated specification is itself correct?

2. **Run the module on all possible inputs** – If we run the module on every possible input and check each output for correctness, then we are guaranteed that the module is correct: it operates correctly no matter what input it is given. Checking the output is usually done by recording the expected answer and comparing the actual answer against that, though the checking may also be done procedurally. The desired outputs can be generated by hand (for small inputs) or by a simpler implementation that has been independently verified (for larger inputs).

This technique is infeasible in practice because the space of inputs may be infinite (a word counting program may take as input an arbitrary file of characters) or effectively so (an arithmetic routine may take as input three Java `int` values; there are 4294967296 different `int` values, so there are about 8×10^{28} different triples of `int` values).

3. **Run the module on some inputs** – Running a module on only some inputs does not provide the guarantee of correctness that the first two options do. Just because a module works the first 1000 times it is run does not mean it will work the next time. However, if the particular test cases are carefully chosen, then we may have high confidence that the module will work on all inputs.

If we plan to run a module on some inputs, rather than its entire input space, then we need to pick inputs that are likely to expose any errors that might be present. But how do we know what inputs to use? We use educated guesses, based on a set of guidelines or heuristics. These heuristics can be thought of as telling us how to select test cases, or as how to partition the input space. The end result is the same: we have guidance about how to build a test suite.

2 Equivalence partitioning

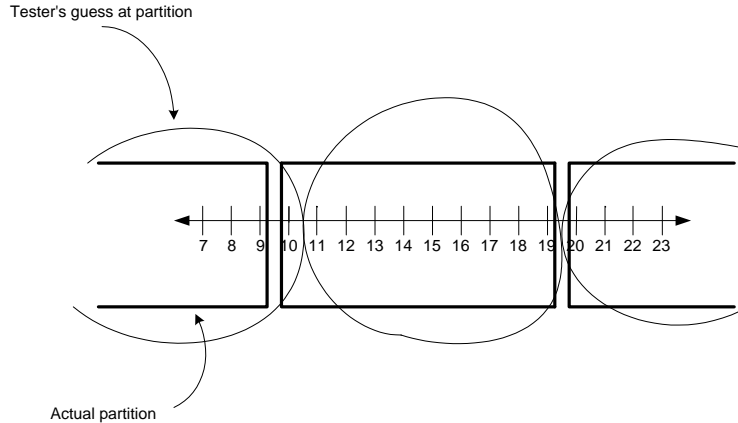
The idea behind this approach is to partition the space of all possible inputs into equivalence groups such that the module “behaves the same” on each group. If the module fails on any member of an equivalence group, then it fails for all members of the equivalence group, and if it works for any member of the equivalence group, then it works for all members. Testing one member is equivalent, in terms of finding errors, to testing every member of the group. Thus, a test suite that includes one (arbitrary) input from each equivalence group provides complete testing of the implementation.

The difficulty with this idea is that it is just as hard to find the equivalence classes of inputs for which the module behaves the same as it is to prove correctness. Therefore, we will use heuristics (rules of thumb that are generally useful but do not guarantee correctness) to select a set of test cases. We will hope that at least one of the test cases appears in each of the true (unknowable) equivalence classes. We will approach this problem in two ways, both by guessing equivalence classes (and then selecting test cases in them to include in our test suite) and by adding additional test cases that are designed to account for potential problems with our guesses at the equivalence classes. If we could guess the equivalence groups correctly (or, if we could guarantee that our equivalence groups are smaller than the true ones, so that a set of cases that covers all of our groups is guaranteed to cover all of the true groups), then we would just need one test case from each and we would be done, for the resulting test suite would be guaranteed to find all bugs.

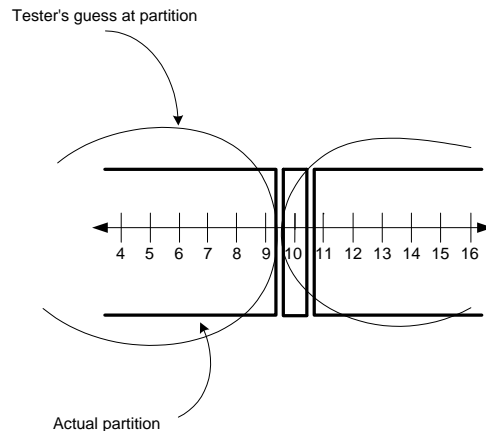
3 Boundary value analysis

When drawing equivalence class boundaries, even if the general idea is correct, the boundaries may be slightly incorrect: perhaps they are shifted a bit, and/or there are small equivalence classes at the boundaries of larger ones.

For instance, you might reason that the module behaves one way for inputs less than 10, another way for inputs between 10 and 20 inclusive, and yet another way for inputs greater than 20. Thus, you might suppose test cases 9, 20, and 21 to be exhaustive. Suppose that you were slightly wrong, and in fact a correct partition would be into inputs less than 10, inputs greater than or equal to 10 but less than 20, and inputs greater than or equal to 20. Then your tests only cover two of the three partitions.



As another example, you might reason that another module behaves one way for inputs less than 10 and another way for inputs greater than or equal to 10; you presume that the test suite consisting of 5 and 15 is exhaustive. Suppose that your reasoning was faulty and the module has three modes of behavior: one for inputs less than 10, one for the input 10, and one for inputs greater than 10. (Maybe the input 10 should have behaved like inputs greater than 10, but your code contains an error.) Again, your test suite fails to cover one of the three true partitions.



As insurance against incorrect selection of equivalence class boundaries, you can include—in addition to “typical” values near the center of the class—test cases near the boundaries. It is wise to include both maximal or minimal values (right on the boundary) and also their neighbors. Don’t forget to test both the minimal and the maximal boundaries. For instance, if a method takes a non-negative BigInteger as its argument, you might supply 0, 1, 2, and some very large values as inputs.

The use of large, complicated inputs (which is essentially the maximal boundary case for unbounded inputs) is sometimes called “stress testing.” It is particularly useful when the system under test does not check its representation invariants, preconditions, or postconditions. If something goes wrong in such a system, it may not be immediately apparent. A lengthy run of the module gives the problem time to manifest itself.

4 Coverage

The goal with coverage is to make some test case exercise each case permitted by the specification, or each element of the implementation code (if that is available). We can also speak of achieving coverage of a domain, such as that of an input parameter. This simply means having an element in each natural subdomain of that domain. For a monolithic domain with no real internal structure, this requires only one test case. Some domains have a natural structure, and in such a case all parts should be covered. For instance, given an integer output, it would be wise to supply negative, positive, and zero inputs. This heuristic, and in fact all the heuristics, should be applied to the domains of:

- ▷ input values
- ▷ output values
- ▷ non-explicit values, such as the size of a set, the number of occurrences of a particular element, and the like. These quantities may affect the behavior of the module qualitatively.

When multiple heuristics are applicable, you should take their cross-product; for instance, if one heuristic suggests two partitions (say, over one input) and another suggests three partitions (say, over another input), then the test suite should include at least six test cases.

5 Duplicates

Modules often behave differently when their inputs have particular relationships to one another. For instance, consider the following routine:

```
// modifies: src, dest
// effects:  removes all elements of src and appends them
//           in reverse order to the end of dest
//           throws NullPointerException if src or dest is null
static void appendList(List src, List dest) {
    while (src.size()>0) {
        Object elt = src.remove(src.size()-1);
        dest.add(elt);
    }
}
```

It works correctly unless its two arguments are the same `List`, in which case it loops forever. *Aliasing*, or multiple references to the same object, is the most common kind of problem, but other kinds of duplication can also be problematic.

6 Black-box testing

Black-box test cases are generated by examining the specification. These tests are sometimes called “functional tests” because they test the module’s functionality but not its implementation. Black-box tests avoid the pitfall of examining the implementation and repeating its errors or assumptions; they are essentially an independent verification of functionality. Such tests are representation-independent and can be reused — and are just as valid and complete — even if a new implementation is substituted for the old one.

Given the following specification,

```
// effects: returns the absolute value of its input
int abs(int x);
```

Applying our heuristics gives the following test cases:

▷ coverage

- input: include some arbitrary test. The input domain is monolithic rather than partitioned, so one test covers it. (Integers actually do have inherent structure, so realistically it would be better to subdivide it into three equivalence classes for negative, zero, and positive integers, and apply the heuristics to each of the classes separately.)
- output: include some test. The output is the integers also. (Actually, it's the non-negative integers; we will use that fact later.)
- non-explicit values: none are obvious here, so no tests are added.

▷ boundary cases

- input: the integers range from `Integer.MIN_VALUE` to `Integer.MAX_VALUE`, so those values (and those one greater or less, respectively) should be provided as input. Supplying `Integer.MIN_VALUE` would uncover a bug in our implementation: no Java function can satisfy the specification, because `Integer.MIN_VALUE = -2147483648` while unfortunately `Integer.MAX_VALUE = 2147483647`, leaving us in an awkward position when we try to compute `abs(Integer.MIN_VALUE)`.
- output: if the output is viewed as the `ints`, we would attempt to produce outputs of `Integer.MIN_VALUE` and `Integer.MAX_VALUE` (and their neighbors). We would soon discover that the outputs are non-negative, so the lower boundary is at 0; tests producing 0, 1, and possibly 2 would be wise. (This does not specify how to select the inputs; they should be selected to be as different as possible, perhaps by using either -1 and 2 or 1 and -2.)
- non-explicit values: again, none are obvious

▷ duplicates – There is only a single input, a single output, and no non-explicit values, so the only tests suggested are one in which the input and output differ and one in which the input and output are the same.

With the specification

```
// effects: if x<0, returns -x; else returns x
int abs(int x);
```

covering the specification implies executing each of the branches; this is essentially what would be produced by considering the implicit partition of `ints` into negative, zero, and positive. For instance, boundary inputs would include -2 and -1 (the upper boundary of the negative class) and 1 and 2 (the lower boundary of the positive class).

7 Another example

Consider the following method:

```
// effects: returns the length of the longest continuous run of the same
//          character in s
//          examples: for s = "HELLO" return 2 (because of repeated L)
//                   for s ="goooogle" return 4 (because of repeated o)
int stutterCount(String s)
```

We can apply our heuristics:

- coverage
 - input: include some test; the string input domain is monolithic. (Inputting the empty string is a wise choice and implicitly suggested; it will also be covered by other heuristics, below.)
 - output: include some test; the non-negative integers are also monolithic. (Only the empty string would return zero, which is again suggested, if the output domain is implicitly partitioned.)
 - non-explicit values: some non-explicit values are the length of the input, the length of the stutter, the location of the stutter (its distance from the beginning or the end), the number of stutters in the string, and the number of maximal-length stutters. These are all non-negative integers. It would be wise to include tests which make them zero; the boundary cases will take care of that as well.
- boundary cases
 - input: boundary cases include the empty string, one-character strings, and very long strings
 - output: boundary cases include output of 0, 1, and a large number
 - non-explicit values: make tests which cause each of the values listed above be 0, 1, and large. Such values may or may not appear in the implementation. They may or may not be useful, but they don't hurt.
- duplicates – A string (of whatever length) containing only a single character achieves the goal of making the stutter be the same as the whole string. Another kind of duplication is having multiple stutters of the same length, either with the same character or with different characters. Such stutters might be adjacent or not (essentially introducing a new non-explicit value, the distance between them, that can be made to be zero, small, or large). These are reasonable tests: it would not be surprising for the routine not to reset an internal variable correctly, particularly if it were extended to return the character as well as the length of the stutter.

8 Yet another example

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 specification; 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.

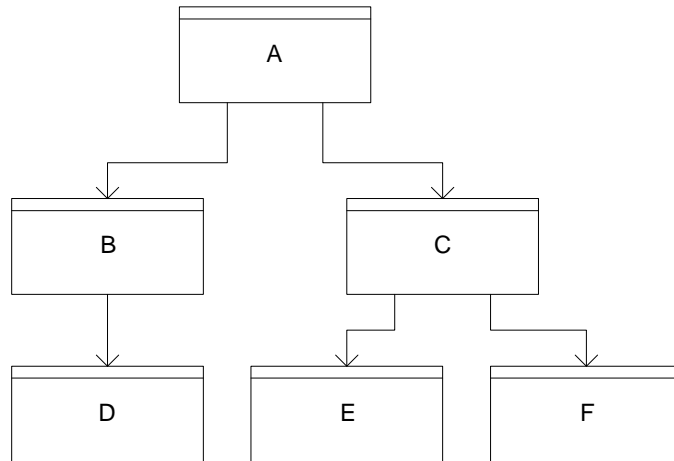
9 Test strategy and automation

Ideally you should test early and often. Black-box testing works best when the test cases are generated without any knowledge of the code, first. Such test cases are ideally written by someone who has never seen the code (or discussed its design) but only read the specification.

You may read about various types or levels of testing, such as unit, module, system, and acceptance testing. These refer to testing larger and larger portions of an entire system and are described in many software engineering books.

Testing can be performed either *bottom-up*, in which subpieces are verified before they are assembled into pieces, then those pieces are verified before being put together into even larger components, or *top-down*, in which the entire system is tested before the pieces are built, and the system is filled in from the root.

Consider the following Module Dependency Diagram:



Bottom-up testing would test D before B; E and F before C; and B and C before A. Top-down testing would test A before B or C; B before D; and C before E or F. Bottom-up testing requires the writing of drivers which exercise the functionality of the class. Drivers simulate both its environment (how it is called) in the system you are building, and all other environments in which it may be called according to its specification. A driver is required in order to perform complete testing, so bottom-up testing does not place any additional demands on the tester. Top-down testing requires writing stubs which simulate the behavior of the parts of the system that are not yet written or tested. The advantage of this approach is that design errors can be identified early rather than only when most of the system is built (and the pieces still don't work together).

In general, the right approach to testing is to do the variety which suits your development style. Testing as you code is more effective because you remember the code better and can find bugs more quickly. Especially if such problems might affect the rest of the design (for instance, a specification is not efficiently satisfiable), it is helpful to be informed of that fact early. It also results in a usable product rather than one that is buggy and thus less likely to be useful in your system. (The same thing goes for documentation.) Additionally, since you are testing only a small amount of code at a time, if there is a problem, it is easier to find. It is reasonable to test an entire class at a time rather than a method at a time (unless the class or method is large and complicated).

Regression testing means re-testing your code after you make a change. This is crucial because it is very easy to introduce a new bug when you fix an old one, but many people don't bother to re-validate their implementation after making corrections, extensions, or other modifications. If tests are easy to run (for instance, you have written a test driver), then regression testing is very easy to do.

Another effective testing strategy is to use assertions. An assertion checks a property and raises an error if the property is not satisfied. These are useful for verifying preconditions, postconditions, and representation invariants. If such properties are violated, then trouble is very likely later on. The more quickly such an error is detected (in terms of how many lines of code have executed between the occurrence of the error and its detection), the easier it is to locate. Locating an error is frequently much, much harder than fixing it.

10 Static verification

The above techniques are *dynamic*—that is, they run the module and perform checks at runtime. An alternative, briefly mentioned in the introduction, is *static* verification techniques which do not

run the module. For example, a checking tool that every programmer uses daily is the compiler. The Java compiler guarantees that there is no type error in its input program (or else it will issue an error and refuse to generate bytecodes). Since type errors are very hard to discover and produce fiendishly difficult-to-debug behaviors, this is a big step forward already. The Java compiler also guarantees other properties, such that there is never a use of a variable before it is assigned to.

Also, don't forget to think! Even if no automatic proof is possible, a human may be able to prove properties of a module, either formally or by informal reasoning. Even the process of writing down the pre- and post-conditions, the representation invariant and the abstraction function (covered next week) may reveal problems with your module by making you think about it in a structured way. Similarly, the act of writing tests (and thinking only about the specification) often leads programmers to realize that their implementation will not handle a particular case.