BIF-5-SSD

Lecture 12. Testing

Taken from a previous module Software Engineering Principles (modified by Peter Rosner)

This week's lecture notes are partly drawn from chapter 19 of the recommended book Software Engineering an Engineering Approach, by Douglas Bell

---

Verification versus Validation

The terms *verification* and *validation* might seem to mean the same thing, but – in the context of software engineering they have distinct meanings.

Verification is often described as…

…"building the system right",

Validation is often described as…

…"building the right system".

In other words, verification is to do with confirming that an implementation satisfies its specification, while validation is to do with checking that the implementation really does meet the users' needs. (Bear in mind that the specification may be out of line with the users' needs!)

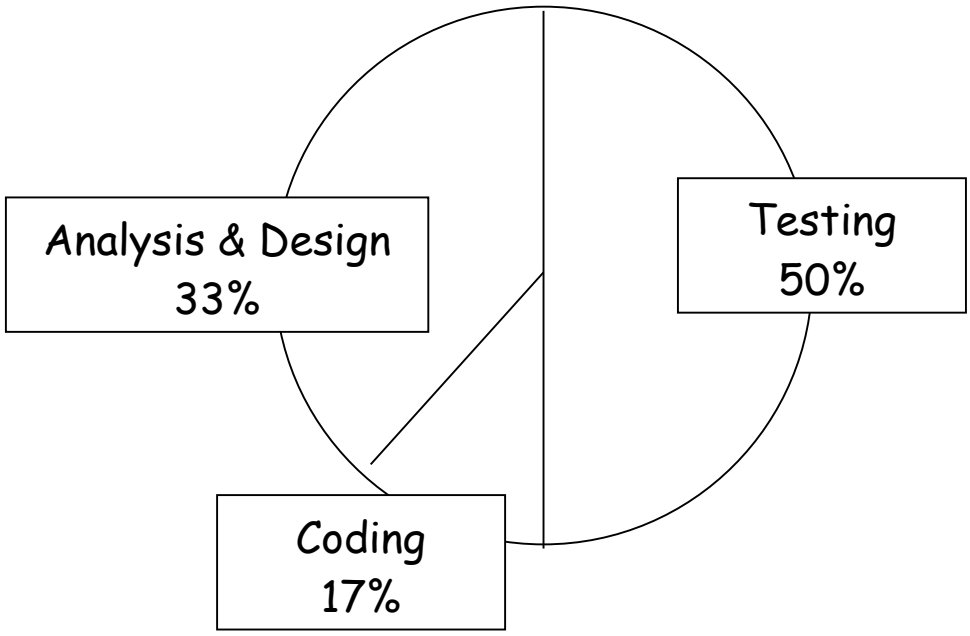
Validation is usually achieved through some form of prototyping and, ultimately (for bespoke software especially) through *acceptance testing*. This is normally carried out by the customer, but it could instead be carried out by an independent third party acting on behalf of the customer.

In the world of formal methods, the term *verification* means proving (as far as possible) that a design conforms to a given formal specification. Often this is referred to as *formal verification*.

Outside of formal methods, verification means testing – i.e. executing the code to demonstrate (verify) that it conforms to the specification, or rather that it *appears to conform* to the specification. The purpose of testing is to find *faults* in the code; faults are (inevitably) the result of human *error* somewhere along the line, and that faults may give rise to *failures* during execution.

The only kind of testing that can be said to address the issue of *validation* is *acceptance testing*.

The following is a pie chart (from Bell) in relation to the costs of developing software…



Bell also points out the following fact, which is in keeping with the pie chart…

"At Microsoft there are as many people involved in testing as there are in programming."

The expression *Beta testing* refers to testing done by customers when a product is in its final stages of development. By contrast, the term *Alpha testing* refers to earlier phases of testing that are carried out in-house.

Customers engage in Beta testing of COTS software in order to gain early experience of a new product, in the belief that this will give them some kind of commercial advantage.

Black Box Testing

Black box testing, also known as *functional testing*, is one of two complementary approaches to testing, the other being white box testing (see below). In the black box approach the focus is on testing program functionality only. No attempt is made to study the program code, so black box testing can be used even when the source code is not available. Only the executable code is needed.

Testing a program using input values that are chosen at random is referred to as *random testing*. This is the easiest way to test software, but it is not the most effective. It's better to choose input values carefully on the basis of *equivalence classes*.

In the context of software testing an *equivalence class* is a set of inputs (or input sequences) that you would expect to be treated in the same way by the software.

It may be easy to spot the equivalence classes in some situations, but in others it may require a lot of thought. It often boils down to a matter a judgement. Lethbridge and Laganière (Lethbridge, T. & Laganière, R. Object-Oriented

Software Engineering McGraw-Hill, 2001) advise their readers to…

> "put an input in a separate equivalence class if there is even a slight possibility that some reasonable algorithm might treat the input in a special way".

As every experienced software developer knows, failures often occur in special or extreme "boundary" cases – for example when maximum or minimum values are encountered. Consequently, these boundary values should be used during testing, in addition to "typical" values.

Deciding which input values to use is one thing, but testers also need to work out in advance what the expected outcomes are. This is essential, although sometimes neglected. It's no good if you run a test and then just assume that the program has given the right result without bothering to check that it is actually the right result!

For example, consider a method that tells whether someone can claim a state pension, which is currently at 60 for women and 65 for men. If an age of 0 or less is submitted to the program the result is an exception.

```
public boolean qualifiesForPension
          (Gender gender, int age)
      throws PensionException {
   . . .
}
```

The enum `Gender` takes values of `Male` or `Female`

A set of inputs used for a test run is known as a *test set*. The following test sets with expected outcomes*, each corresponding to an equivalence class*, can be identified.

Test Set		**Expected outcome**
Male	43	false
Male	67	true
Male	-40	exception
Female	44	false
Female	62	true
Female	-40	exception

For boundary conditions the following tests can be identified:

Test Set		**Expected outcome**
Male	64	false
Male	65	true
Male	66	true
Male	0	exception
Male	1	false
Male	-1	exception
Female	59	false
Female	60	true
Female	61	true
Female	0	exception
Female	1	false
Female	-1	exception

Should we test using illegal input values?

This depends, in general, on the specification. Ideally a specification should state not only how the system must behave given legal inputs, but also how it should respond to illegal inputs.

Maybe the system should simply ignore the inputs, or maybe it should respond by throwing an exception and producing an informative error message.

Alternatively, perhaps it really doesn't matter what the system does in response to an illegal input – and in this case you could argue that there's no need to discuss it within the specification. Bear in mind, though, that this would imply that it's OK (i.e. conforms to the spec) if the system crashes or produces some other kind of catastrophic result in response to an illegal input!

White Box Testing

In white box testing, also known as *structural testing* or *glass box testing*, the idea – roughly speaking – is to devise sets of input data that deliberately force execution of as many different paths through the program as possible. (In fact it's not quite as simple as this, as we shall see.)

Consider the following contrived example of a module that is supposed to return triple the product of its inputs…

```
public int tripleProduct(int x, int y) {
  int p;
  p = (x*y)*3;
  if (p == 999) {
      p = 0;
  }
  return p;
}
```

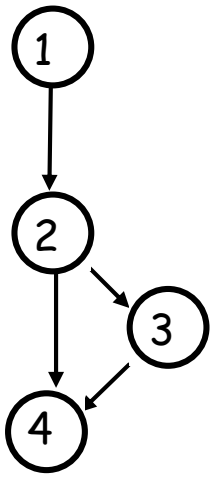
If **x** and **y** are random integers, it's extremely unlikely that **(x*y)*3** will equal **999**, bearing in mind that an integer represented using 32 bits (as in Java) can take any value between -2^{31} and $2^{31}-1$. Of course it's easy to see *by inspection* that there's something wrong with the code, but

testing isn't inspection. You would have to test the software very thoroughly using lots of different values for the inputs x and y, or be VERY lucky, to spot this fault using black box testing.

With white box testing, on the other hand, the fault is easily exposed. For this we need to first analyse the code to find the paths. If we re-write the executable part of the code like this…

```
1.  p = (x*y)*3;
2.  if (p == 999) {
3.    p = 0;
4.  }
```

…then it is easy to see that there are two distinct paths:



For white boxing testing we need to force execution of both paths. The only way we can get node 3 to be executed is by using values of **`x`** and **`y`** such that **`(x*y)*3 = 999`**. If we try using these values in a test we will then discover that the output is not triple the product of the inputs.

Black box and white box testing are complementary

White box testing complements black box testing; the idea is to use both approaches (assuming that you have access to the source code).

Designing white box tests usually takes more effort than designing black box tests because choosing input data to force execution of certain statements or paths can be very difficult.

As in black box testing, it's a good idea to use boundary values where possible when designing white box tests – so long as they force execution of the right paths.

A good strategy is to start off by designing a black box test specification first, and then to check which program statements, branches and paths this covers.
There are CASE tools which can do this automatically; they provide an indication of what is often referred to as *statement coverage* and *path coverage*; i.e. what %age of the statements or paths were covered by a given test specification.

Object oriented coverage tools

For object oriented systems, coverage tools are available (some of them open source such as as the Java profiler Emma http://emma.sourceforge.net/) that detect the coverage by test code, such as JUnit tests, of all statements in each method in a system.

At a higher level such coverage tools also record which methods have been been run in a suite of test runs. At an even higher level they can record which classes have been covered in a suite.

These coverage tools therefore can provide an indication of the extent to which test programs are actually testing all parts of a system.

Exhaustive testing is usually impossible!

Some articles and textbooks may give you the impression that exhaustive testing means testing every path within the software, but this is incorrect. Even if a path executes successfully given particular values of the input data, that doesn't mean it will execute successfully for all values of the input data!

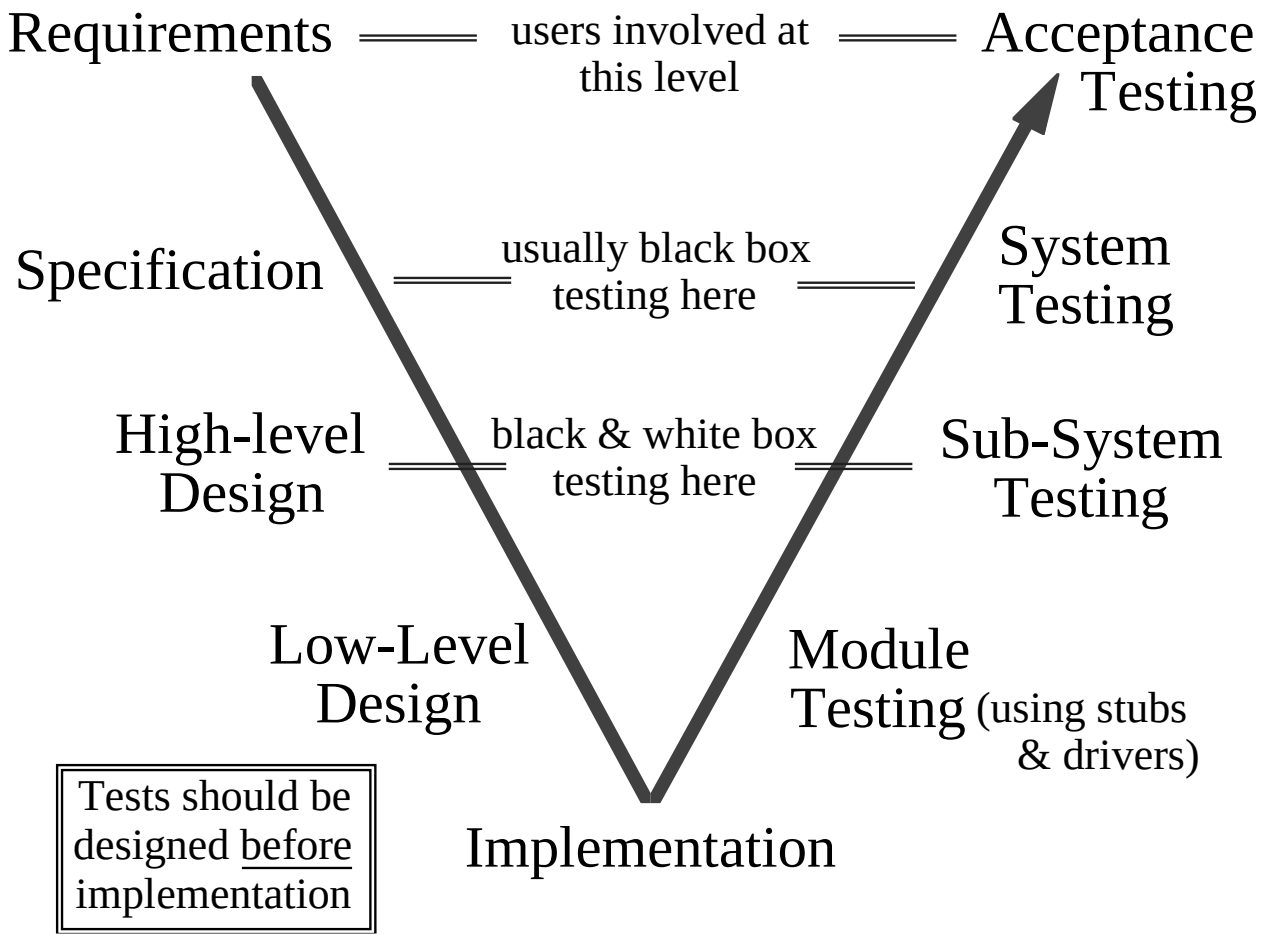
Bell is correct when he says (on p. 269) that exhaustive testing means "to use all possible data values, in each case checking the correctness of the outcome".

It is well known that *exhaustive* testing of even quite small programs is usually impossible, and by now you should understand why that is the case.

Hence Dijkstra's famous remark…

"Testing can only show the presence of bugs, never their absence."

The "V" process model



The "V" model emphasises testing. It fits in with the idea that 50% of the effort in software development is spent on testing. It also acknowledges the fact that testing should pervade the whole software development lifecycle; test sets can and should be designed before implementation.

Remember that one of the approaches to requirements analysis involves drawing up use cases and scenarios; the latter are perfect as a basis for *acceptance tests*.

Large/complex systems are invariably made up of sub-systems, which in turn are often made up of smaller sub-systems. Bottom level sub-systems are sometimes referred

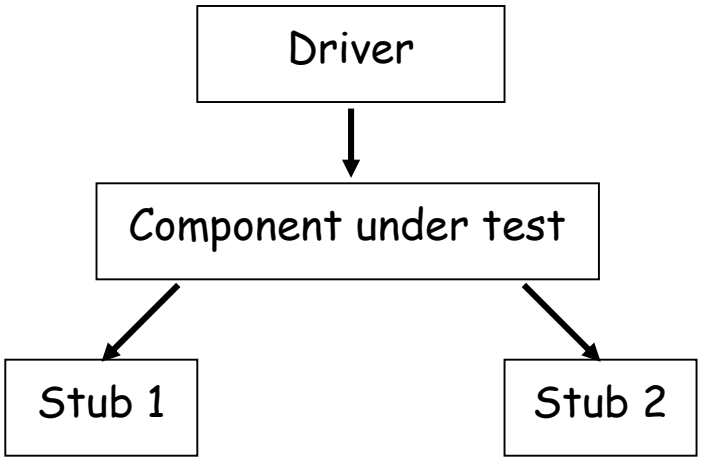
to as *components*. At each level the aim is to maximise *cohesion* and minimise *coupling*. As far as testing is concerned, the idea is to test each of the components/sub-systems in isolation before they are integrated into the overall system.

Testing at the sub-system and system levels is sometimes called *integration testing*; this means integrating together the various parts of the system (which have already been tested in isolation) and testing the combination of these parts.

Use of *stubs* and *drivers* in system development

The concept of *top-down* development involves developing the system/sub-system architecture prior to developing its constituent components.

Even if certain lower-level components don't yet exist it's still possible to do some testing on the higher-level components that depend on them. This can be achieved by using a *stub* in place of each of the missing components…



Each stub acts as a kind of dumb version of the lower-level component it represents. For example, imagine designing a system that will require a yet-to-be-developed component to calculate the median value of a set of integer values; a stub for this might simply return a fixed integer value. (The tester will obviously need to make allowances for this.)

Just as a stub takes the place of a lower-level component for the purposes of testing in top-down development, so a *driver* (sometimes referred to as a *test harness*) takes the place of a higher-level component in *bottom-up development*.

Automating the testing process

Testing is often done manually, with the user supplying the inputs, but this can be too labour intensive if the test specification consists of many test sets.

There are many commercially available tools that have been designed to automate the testing process. Some can even create mutants (e.g. by randomly changing a ">" to a ">=" within a conditional *if* statement) before carrying out the mutation testing automatically.

Other automated testing frameworks are open source. For Java the most widespread for low level class testing is JUnit.

Automated test tools are vital for *regression testing*; this means repeating all of the tests whenever a modification is made to the code (as opposed to just trying to test the modification).

Automated testing tools are also available to replay user key strokes in an interactive system repeatedly in regression tests.

Incremental integration testing

Since testing lies on the *critical path* in terms of project progress, it's a good idea to begin testing parts of the system early on, while other parts of the system are still being implemented.

Incremental integration testing (sometimes referred to just as *incremental testing*) involves testing one part of the system, then adding another part and testing the combination of the two, and so on in an incremental manner. The idea is that if failures occur these are most likely to be due to faults in the most recently added part.

An extreme programming view of testing

Extreme programmers say that modules should be written in a *test-code-test-code* manner (rather than a *code-test-code-test* manner). The following (slightly edited) quotes are taken from the website www.extremeprogramming.org

"Creating a unit test helps a developer to really consider what needs to be done. Requirements are nailed down firmly by tests."

"By creating tests first your design will be influenced by a desire to test everything of value to your customer."

"There is a rhythm to developing software [when you] unit test first. You create one test to define some small aspect of the problem at hand. Then you create the simplest code that will make that test pass. Then you create a second test. Now you add to the code you just created to make this new test pass, but no more! Continue [in this vein] until there is nothing left to test."

"The code you will create will be simple and concise, implementing only the features you wanted. Other developers can see how to use this new code by browsing the tests."

When is a fault a fault?

Consider this fragment of code…

```
if (x > 2) then
  do A
else if (x = 10) then
  do B
else
  do C
endif
```

Whatever the value of **x** here, **B** can never happen, because **B** is "unreachable".

Problems like this ought to be found through inspection or white box testing, or code coverage analysis. Clearly this is bad code, but it would only strictly count as a fault if it could potentially give rise to a failure during execution. Unreachable code may not lead to any failures, but it doesn't inspire confidence in the quality of the code!

While many faults turn out to be specific erroneous source code statements, not all faults have specific locations. For example, the underlying algorithm may be erroneous, or there might actually be some functionality (i.e. code) missing.

Debugging

Locating the fault that caused a particular failure can be very hard work. Here are three complementary approaches to debugging…

- try variations of a test set that causes failure,
- insert diagnostic print statements into the code,
- use a debugger.

Having found a test set that causes the software to fail it can be very revealing to experiment with variations of that test set; this may give a clue as to the nature of the fault.

Inserting print statements into the code is probably the most common approach to debugging. This is sometimes referred to as *instrumenting the code*. It's essentially a manual trial-and-error process that aims to home in on the fault. Inserting print statements into the code is related to *assertion checking*.

The third way of finding faults in code is through the use of a debugger, which is an essential tool in any professional program development environment. A debugger allows you to set pre-determined break points during execution of the code, or to step through the execution one instruction at a time. At each step, or break point, the debugger allows you to examine the value of every item of data. This saves the developers the effort of building in print statements.

Non-deterministic behaviour makes debugging hard

Most of the software you will have written or looked at during your studies so far will have been deterministic; in other words it will have behaved the same way every time it is executed. Not all software is like that, however.

Think of software that consists of concurrently executing sub-systems; for example a client-server system, or a standalone application involving two or more concurrent threads of execution. It may be possible for such software to behave a different way in different executions, even though the inputs are identical from one execution to the next, because of differences in the way the instructions are interleaved by the operating system of the host machine.

Concurrent systems are notoriously difficult to debug. They can exhibit any or all of the following problems…

- deadlock,
- livelock (also known as *starvation*),
- failure due to violation of mutual exclusion

…and, worse still, they often do so intermittently!

Hard (as opposed to soft) real-time systems can also be very difficult to debug, because if you try to use a debugger – or if you instrument the code to show the values of key variables at certain points within the

execution of the program – then you inevitably change the run-time characteristics of the code.

(What's the difference between a *soft* real-time system and a *hard* real-time system? The former is a system with which users interact, so it should respond as quickly as possible. Hard real-time systems have much stricter time constraints; a response that is too slow is considered a failure.)

What else can be done besides testing?

We've already seen (in previous weeks) the importance of peer inspection. We saw that there can be faults in software that would be very unlikely to lead to failures during black box testing although they should be easy to spot by inspection. Some studies have actually concluded that peer inspection is more effective at finding faults than testing.

An added benefit of peer inspection is that it can help to improve the code even if no actual faults are found. For example, there might be a simpler or more efficient way to achieve something, or improvements might be made in respect of commenting or choice of variable names.

Inspection should be carried out before, rather than after, testing. It's much better to aim for high quality through fault *avoidance* rather than fault *elimination*.

In the *cleanroom* approach to software development, software design is treated like silicon fabrication. Here the idea is to do everything you can to check your code before even compiling it, let alone testing it. If you've done the job properly, it should compile and pass all the tests at the first attempt!

Some organisations offer financial rewards for individuals who discover significant faults or failures. I've seen these referred to as "bug bounties".

One famous computer scientist – Donald Knuth – was well known for this; he offered financial rewards for anyone who found errors in either his textbooks or his software.

The rewards doubled every year; they were $2.56 per error found in the first year and then doubled every year until he was offering $327.68 per error found, at which point he stopped doubling the reward!

Apparently there are a number of computer scientists around the world with uncashed cheques on their walls signed by Donald Knuth! If you'd like to read more about this, look up Knuth's *All Questions Answered* lecture transcript from 2001…
[http://www.ams.org/notices/200203/fea-knuth.pdf].