ECE155: Engineering Design with Embedded Systems Winter 2016

Lecture 17 — More on Testing

Jeff Zarnett and Patrick Lam

Regression Testing

The name "regression" testing comes from the desire that the software only goes forward ("progress") and not backwards ("regress"). The goal is to make sure that bugs previously discovered and fixed do not re-appear. There's nothing more frustrating for users (and developers) than encountering a bug that was previously fixed but came back for an encore.

Regression testing refers to any software testing that uncovers errors by retesting the modified program. Often refers to large suites of test cases which detect regressions:

- of a bug fix that a developer has proposed.
- of related and unrelated other features that have been added.

Attributes of Regression Tests. Regression tests usually have the following attributes:

- **Automated**: no real reason to have manual regression tests.
- **Appropriately Sized**: suites that are too-small will miss bugs; suites that are too-large take too long to run. Optimally, we want to run tests continuously.
- **Up-to-date**: ensure that tests are valid for the version of program being tested.

Tip for the labs: if you create regression tests whenever you fix a bug, you will not get the situation that a last-minute change silently breaks a whole bunch of things right before your Lab 4 demo.

Stress Testing

Stress testing is not something we will focus on, but it is common in industry, especially where reliability is important. As the name suggests, stress testing is testing the system when it is "under stress", to find out how it performs. Ideally, the system performs well under pressure, but in reality systems often crash, become unusable, or otherwise behave badly when under the right (or wrong) kind of pressure.

For a system to be under stress, it has to be constrained in some resource(s). For any multi-user system, stress testing can take place by simply adding more and more users to the system until it can accept no more. Some systems start to break down when there are two concurrent users; other systems happily accept thousands. Stress tests involving client-server architectures can be run by spawning a very large number of clients having them all attempt to simulate user behaviour (making requests of the server).

Other resources can be constrained as well, to see the behaviour of the system under those circumstances. Some things you can restrict: CPU, RAM, hard disk space, network connection speed, screen resolution. It sounds silly, but a great many programs crash or behave erratically if the hard disk is full! In other situations, you may also need to test what happens when one or more components are unreliable or unavailable. For example, you can test what happens with your software when the network connection occasionally loses some data. In life and safety critical applications, you might consider how the system behaves if major components like RAM fail.

As a general rule, you may assume there is some point where your system will fail. Even if running out of hard drive space is not an issue, there will be something else (CPU, or concurrent users, or whatever it is). The best you can do is accept that this will happen and attempt to fail gracefully. Telling the user that they cannot log in because the server is full is much better than accepting logins until the server crashes. Users don't like being told that the server is busy or full (and they might go complain about your product on the internet), but at least it is a graceful failure and the people logged in are able to continue.

Unfortunately, stress testing is very often just a simulation. You might be able to spawn 50 000 clients using 50 computers and have them interact with the server to test it out, but this is not the same thing as having 50 000 users trying to work with the system at the same time. Some obvious differences: the user simulator will never be the same as actual user behaviour, having 50 computers with 1 000 clients each means 50 IP addresses to communicate with instead of 50 000, communication takes place within the internal network so it is fast and reliable, and so on.

System Testing

System testing is also not an area of focus in this course, but it should not be forgotten. When the system is in a running state it's possible to test it as it will actually be used. One way to do this is to get (future) users of the system to execute their normal workflow and see what you uncover. Rare is the software, no matter how well tested, that survives initial contact with actual users.

Industry Processes

Good software companies have certain testing features that help them deliver good quality software. The following practices are used in industry:

- **Unit Tests:** Each class has an associated unit test. If you change a class, you must also modify the unit test.
- **Code Reviews:** Each branch in the central version control system has owners. To commit code to that branch, you must have your code reviewed and approved by one of the owners. This ensures code quality.
- **Continuous Builds:** There is often a machine that continuously checks out and tests the latest code. All unit and regression tests are run and the status is made public to the team. The status contains information about the whether the code was built successfully, whether all unit and regression tests passed, and a list of the last few commits that were made to the branch. Social pressure: if you break something, everyone knows about it :).
- **QA Team:** If all tests have passed, a QA team will look for additional bugs.
- **Release:** Once QA has approved a build, it is released for use.

Choosing Test Inputs

We've talked a lot about what kinds of tests for software should be created, and considered a case study of Android games. However, there are some important points about testing inputs you will need to know before you start writing tests.

Why Choose? Let's do them all!

We have to choose, because it's impossible to test exhaustively (test all inputs to the software), and similarly impossible to test all states of the software. And I do mean impossible, not just unpleasant. Consider an incredibly simple function:

```
public int compute(int number1)
```

Assuming `int` is a 32 bit integer, if we tried giving every input (there are 2^{32} possible values), and we can execute 1 billion (1×10^9) tests per second, it will take about 0.07 minutes to execute this test (thanks to Wolfram Alpha for doing this math, because I can't do it in my head). That's not very long and seems like it's testable. Didn't I say it was impossible? Well... a function that takes one integer parameter is a trivial example. What if the `compute` function takes two integers?

```
public int compute(int number1, int number2)
```

Again, using 32 bit integers, there are now $2^{32} \times 2^{32}$ possible values we could give in to this function (all possible combinations). Again, executing the test at a rate of 1 billion tests per second, it would take 584.9... years!

Okay... that's a really, really long time... So a test running at that rate started in the year 1428 would finish about now. For historical reference, Christopher Columbus sailed for the new world (the Americas) in the year 1492, 64 years later. In fact, in 1428, Columbus hadn't even been born. The first 32 bit processor (a Motorola 68k, according to Wikipedia), wasn't introduced to the world until 1979. That chip ran at about 700 000 instructions per second (and let's assume we have 1 instruction = 1 call of this function), so it would take that chip about 835 362 years to complete this test. If we started in 1979, in 2013 (34 years later!) we'd be about 0.00407% done.

And if there are three 32 bit integers as parameters?

```
public int compute(int number1, int number2, int number3)
```

Wolfram Alpha now suggests that at a rate of 1 billion tests per second, this would take 2.512 trillion years, or about 180 times the age of the universe.

What is the important take-away from all this math? It's impossible (in general) to test every possible input to a function. So we're going to have to choose carefully what inputs we use.

Valid and Invalid Inputs

Typically when you write a test you will choose "good" inputs, but it's also important to test what happens when invalid input is given. We just made the problem of exhaustive testing worse, of course, because when we looked at the integers we considered only valid integer values.

As we discussed in the specification lecture, there should be some expected behaviour for your software when invalid input is given (and you can verify that this behaviour takes place). Sometimes the answer is that invalid input is ignored, but most of the time we report an error or exception.

The Fibonacci function, for example, is only meaningful for positive integers. So as input you might consider giving positive integers (like 2, 17, 1000), as well as zero and negative numbers (-1, -5000). Does the function fail silently when -1 is given as an input? Does it return an error?

Black Box and White Box Testing

In *Black-Box* testing, tests are written without looking at the source code. In some cases, this is only accomplished by having someone else write the tests. The name comes from the idea that the object being tested is a black box – we have no access to or knowledge of the internals and we care only about the inputs and outputs. If following the test-driven development process, then most tests are created this way, because the test is created before the implementation. Looking at the specification of the input and output is our source of what values are to be used for testing.

In the Fibonacci example above, the implementation of the Fibonacci function is irrelevant; we chose the positive integers, zero, and negative integers just by understanding the specification of what the function is supposed to do.

In *White-Box* testing, tests are written while looking at the source code. It's called white-box testing because it's the opposite of black-box testing. In addition to the test values we choose from the black box testing process, we have the benefit of analyzing the source code while we do this. We can use values already identified in the code:

For example, if we find the following statement in the code:

```
if (parameter > 42) {...  }
```

Then we know that there's something about 42 that makes it important for testing. We should then choose as test inputs for the variable `parameter` a number less than 42, a number greater than 42, and the value 42.

Sometimes there is not quite so clear a relationship between the input and a comparison. So to generalize this, we should make sure we go through all the branches of the code; that is, give input so that every if statement is evaluated as both true and false.

From the exhaustive testing rules, we know it will not be possible to test every state of the program with the values we generate from black-box and white-box testing. Complicated functions with many parameters cannot be tested completely.

When to Stop Testing

As we just saw, testing is potentially endless; there is no way to say we have found all the bugs in the software. At some point, it is necessary to say it is enough and ship the software (or, at least, move on to the next part). Realistically, testing is a trade-off between budget, time, and quality. Under bad circumstances, testing stops because of time pressure. Under best circumstances when the benefit of additional testing exceeds the cost of testing [Pan99]. It will be necessary to apply engineering judgement about how much is "enough" to be able to say the software is adequately tested.

For more on that, take the Software Testing course in 4th year.

References

[Pan99] Jiantao Pan. Software testing, 1999. Online; accessed 26-December-2013. URL: https://www.ece.cmu.edu/~koopman/des_s99/sw_testing/.