# TESTING YOUR CODE

Testing is necessary to ensure the quality and security of your web applications. Put simply, does it do what is supposed to do; is there a way to make it do what it shouldn't?

The trick to test design is finding the appropriate amount of fastidiousness that strikes the balance between testing perfection, and getting it launched.

Consider a calculator: Do you need to test every possible combination that could be entered? You should, but that could take years. It all comes down to risk management. You must know what the code is supposed to do, and do enough testing to expose as many bugs as possible.

Despite our best efforts, inevitably there are errors that are only revealed once our work is in the end user's hands (that is why companies will sometimes release free "beta" versions of their software). The objective of testing is to minimize the work we will have to do to fix those errors.

## TESTING CATEGORIES

Generally speaking, we can classify the myriad types of tests into two categories: Software Testing, and User Testing. This lesson will focus on Software Testing.

## WHAT DO WE TEST?

We are not just testing the software, we need to test it against a set of rules to know if the software is doing what is supposed to. This set of rules is referred to as the **specification** ('spec') or, alternatively, the **requirements**.

### WHAT ARE WE TESTING FOR?

Defects, errors, anomalies, failures, variances, and inconsistencies are all terms covered by the term, "bug". (Patton, 2006, p. 13)

A bug occurs when one of the following five rules is true:

- The software doesn't do something that the product specification says it should do.

- The software does something that the product specification says it shouldn't.
- The software does something that the product specification doesn't mention.
- The software doesn't do something that the product specification doesn't mention but should.
- The software is difficult to understand, hard to use, slow, or – in the software tester's eyes – will be viewed by the end user as just plain not right. (Patton, 2006, p. 15)

## BUG PREVENTION

The longer it takes to discover a bug, the more expensive it becomes to fix it. (Patton, p. 18) In the traditional "waterfall" project management style testing isn't executed until the project is almost ready to launch. Fixing anything at this stage becomes extremely expensive as each change can have a ripple effect throughout the software. In the "agile" project management style, code is tested on a regular and frequent basis – ideally, by the end user. Naturally, the agile method takes more time, but the belief is that catching bugs as early as possible is the most cost-efficient way in the long run.

*"The goal of a software tester is to find bugs, find them as early as possible, and make sure they get fixed."* (Patton, 2006, p. 19)

Surprisingly, most bugs do not come from programming errors. Rather, the number one cause of bugs has consistently traced back to a poor, constantly changing, or non-existent specification document. (Patton, pp. 16-17) Not having a clear idea from the start leads to difficulties down the road of an exponential nature. Therefore, the most profitable time for debugging is during the specification and design stages.

## TESTING THE SPEC

In creating the specifications for your features, here are some words to watch for:

- **Always, Every, All, None, Never**. If you see words such as these that denote something as certain or absolute, make sure that it is, indeed, certain. Put on your tester's hat and think of cases that violate them.

- **Certainly, Therefore, Clearly, Obviously, Evidently**. These words tend to persuade you into accepting something as a given. Don't fall into the trap.
- **Some, Sometimes, Often, Usually, Ordinarily, Customarily, Most, Mostly**. These words are too vague. It's impossible to test a feature that operates "sometimes."
- **Etc., And So Forth, And So On, Such As**. Lists that finish with words such as these aren't testable. Lists need to be absolute or explained so that there's no confusion as to how the series is generated and what appears next in the list.
- **Good, Fast, Cheap, Efficient, Small, Stable**. These are unquantifiable terms. They aren't testable. If they appear in a specification, they must be further defined to explain exactly what they mean.
- **Handled, Processed, Rejected, Skipped, Eliminated**. These terms can hide large amounts of functionality that need to be specified.
- **If…Then…(but missing Else)**. Look for statements that have "If…Then" clauses but don't have a matching "Else." Ask yourself what will happen if the "if" doesn't happen. (Patton, 2006, p. 61)

## TYPES OF SOFTWARE TESTING

No web developer's toolbox is complete without a set of testing tools. Knowing which tool to use when will provide a comprehensive coverage for your work.

### Dynamic & Static Testing

**Static testing** refers to testing something that's not running—examining and reviewing it. **Dynamic testing** is what you would normally think of as testing—running and using the software. (Patton, 2006, p. 57)
Static testing would be used to compare the interface with the mockups, or for browser testing. Dynamic testing is used for any interaction with the application.

### Black Box & White Box Testing

Sometimes referred to as *functional testing* or *behavioral testing,* **Black Box** testers cannot see the code, they only know what the code is supposed to do. In **White Box testing** the tester has access to the code and, therefore, can tailor their test to specifically challenge the application. For example, if you know a date field on your form is programmed to validate against past dates, you would test the app by entering an invalid date. Being able to see the code

and know its inner workings lets you test the code more specifically. (Patton, 2006, p. 55)

## Test-to-Pass & Test-to-Fail

There are two fundamental approaches to testing software: ***test-to-pass*** and ***test-to-fail***. When you test-to-pass, you really assure only that the software minimally works. You don't push its capabilities. You don't see what you can do to break it. You treat it with kid gloves, applying the simplest and most straightforward test cases. After you assure yourself that the software does what it's specified to do in ordinary circumstances, it's time to put on your sneaky, conniving, devious hat and attempt to find bugs by trying things that should force them out. Designing and running test cases with the sole purpose of breaking the software is called testing-to-fail or *error-forcing*. (Patton, 2006, pp. 66-67)

In our labs last semester, you may have noticed me doing this to check your code. I would test your code to see if it did what it was supposed to first, then I would test it again with values that I knew should throw errors.

## Boundary Conditions

Boundary conditions are those situations at the edge of the planned operational limits of the software. (Patton, 2006, p. 72)

This means testing just inside and just outside the minimum and maximum expected values. If your event application asks for a month (1-12), you would test 1 and 12, then 0, -1 and 13.

## Unit Testing

Unit testing is challenging your code right down to the level of individual functions. In essence, you create functions to test your functions. It also enforces the notion of modularity, where you can break off sections and reuse them elsewhere – which also follows the Object Oriented Programming principle of abstraction and encapsulation.

The classical definition of unit testing is that it is a piece of code (usually a method) that invokes another piece of code and later checks the correctness of some assumptions. (Saleh, 2013, p. 7)

The basic principle of unit testing is that you pass in a value to the function you wish to test, and make an assertion as to what you expect the result will be based on the passed in value. For example, if I had a function that validated

postal codes, I would pass in “M9W5L7” and expect that it would pass. Then, I might pass in “xxx444” and expect that it would fail.

There are numerous automated tools that will run your unit tests for you – some, will run every time you hit Save!

There are two approaches to implementing unit tests: creating the tests after the code is completed (traditional); and creating the tests before the code is created. This approach is known as Test-Driven development. (Saleh, 2013, p. 10)

### Test-Driven Development (TDD)

The arguments against Test-Driven development are that it is laborious, slow, and ‘puts the cart before the horse’. Arguments in favour state that it creates a greater understand of the code before it is created, and improves documentation, coverage and a streamlined code base.

For more information on the benefits of Test-Driven development:
http://www.base36.com/2012/07/benefits-of-test-driven-development/

## References

Patton, R. (2006). *Software Testing 2nd Ed.* Indianapolis, Ind: Sams Pub.

Saleh, H. (2013). *JavaScript Unit Testing.* Birmingham UK: Packt Publishing.