COURSEWARE

Functional Testing

Contents

Overview

Testing is an activity that gathers information about code and reduces the risks of things going wrong in a production setting.

It is impossible to write a perfect piece of code as there will always be ways to make it more efficient (*measured by performance testing*), to make it more secure (*measured by security testing*), or even to just make it more readable (*static analysis testing*).

Testing is usually broken down into two categories, **functional** and **non-functional**.

Functional Testing

Functional testing is primarily focused on how code is structured, compiled, and runs. If there is a value being returned that doesn't meet our expectations, we might want to know precisely where this is happening and how.

> The following points are testing strategies that help us improve our code by checking for certain points of failure, whilst it is good to know about each of these points, you will find certain approaches have greater foundational importance.

We will cover the following functional testing strategies:

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing
- End-to-End Testing
- Regression Testing
- Smoke Testing
- Sanity Testing

Unit Testing

A unit is the smallest whole increment, from which this testing gets its name. It is by far one of the most important tests and it tests a small amount of code, usually a single method, to see if it returns the expected output.

A unit test is meant to be a single item under test, so break up the complexity of methods calling methods, we might use `stubs` and `drivers`.

In Object-Oriented languages such as `Java` and `C#`, a unit test might be best represented by a class object, we can try to isolate the units (methods and attributes) and ensure that they function correctly when called in isolation.

> This testing is usually done by programmers themselves as it requires an understanding of what the code is doing.

The following resources/utilities may help us support our efforts to perform Unit testing:

- Stubs

  Stubs are replacements for called methods, rather than calling the real method we hard code what the response from that message will be - as long as the other method is doing its job correctly, the method under test should therefore pass.

  > To appreciate how we might use a stub, consider a class that defines a `has-a` relationship (The class defines an attribute that itself is a class). Sometimes we might stub the behaviour of this unique attribute because it is not the target "unit" of the test and falls outside our scope.

- Drivers

  Drivers are replacements for inputs, they "drive" information into the method under test. Drivers can be beneficial when testing methods that take in information from an external system.

  > A driver may facilitate the test, but it will not be the target of testing. Perhaps a driver may return a certain resource unique to its related technology.

Integration Testing

Integration Testing is usually the next step after unit testing, once the functionality of the individual components is confirmed.

Next, we verify that the functionality of the combined methods and modules. The approach to do this varies, but there are three common ways:

- Top-down integration

  Top-down integration is where you start with high-level components or modules and you test that they successfully interact with a low-level module. If the low-level module calls an even further low-level module - that input will be stubbed.

- Bottom-up integration

  Bottom-up integration is where you start at a low-level component or module and you test that the driving inputs from a high-level module are correct.

- Big Bang integration

  Big bang integration doesn't follow many guidelines. In essence, you combine everything and hope that it works. It is quick, however, if the test fails it does not give much information as to why.

System Testing

System testing is where the entire system is put under test.

It should not include any drivers or stubs. The system is thought of as a "black-box", where a known input is given, it will process it and then check that the outputs are in line with the specifications.

Imagine integration testing but we allow the whole system to run; rather than a portion of it.

> For system testing, we aren't concerned with resources such as databases.

Acceptance Testing

User Acceptance Testing (UAT) is performed by the clients to make sure that the system meets their business requirements or of the end-users requirements.

However, this might come in the form of user stories; where developers instead use the stories to create tests that verify and then validate the behaviour of the system to meet user expectations.

There are also two sub-categories of acceptance testing:

- Contractual Acceptance Testing (CAT)
  done to determine whether the requested software meets the standards set out in the contract.
- Regulation Acceptance Testing (RAT) checks whether it meets software regulations and will usually be specific to local legal and government regulations.

End-to-End testing

End-to-End testing is testing that from start to finish is behaving as expected. This is used to identify system dependencies and to ensure that the data can be sent between various system components and systems.

In End-to-End testing, we might be concerned about resources like our database, hosting server, and other dependencies. As such we might consider having an entire test environment to support our End-to-End testing.

Points of failure may include errors about:

- Data parsing issues
- Key resources not found
- Unexpected data types

Regression Testing

Regression testing is running old tests to determine that any new code changes haven't affected previously working functionality.

> Note: These tests must be run before releasing software.

▶ Why we run Regression tests?

Smoke Testing

A test that gets its name from hardware testing, to check for smoke from the electronics when powered on.

> There is a context for smoke testing in non-functional testing and it is good to be aware of the distinction.

In the context of functional testing, a smoke test is a test that simply checks the application does not encounter any failures during build and runtime.

> The idea is to run through the most critical/fundamental path/s of your code.

▶ Why we run Smoke tests?

Sanity Testing

A non-scripted process where a section is checked to see it is behaving as expected.

Sanity testing is a way for us to naturally identify areas of concern in our code, perhaps our input is only translated into one type of data and we expect it to be used for `strings` and `integers`.

> Think: "Just for sanity" - I want to ensure this runs as I expect it.

We may have cases whereby we want to check if "incorrect data triggers exceptions by providing data in a quick run" or if "a complex data type is being handled correctly during runtime".

▶ Why we run Sanity tests?

Summary

Whilst all these testing approaches bring a way of identifying and capturing errors and providing good means of preventing errors from reaching our releases, we will follow the `V-Model` approach which focuses primarily on the following approaches:

- Unit testing
- Integration testing
- System testing
- Acceptance testing

It is also good to be aware of the idea that there isn't one definitive outline in which categorises functional testing approaches.

> There are cases where functional tests can also double as non-functional tests. So be aware that functional tests focus on the code functional!

Tutorial

There is no tutorial for this module.

Exercises

There are no exercises for this module.