Chapter 36
Testing

36.1 Introduction

This chapter considers the different types of tests that you might want to perform with the systems you develop in Scala. It also introduces test-driven development.

36.2 Types of Testing

There are at least two ways of thinking about testing:

1. It is the process of executing a program with the intent of finding errors/bugs (see Glenford Myers, The Art of Software Testing).
2. It is a process used to establish that software components fulfil the requirements identified for them; that is that they do what they are supposed to do.

These two aspects of testing tend to been emphasised at different points in the software lifecycle. Error testing is an intrinsic part of the development process, and an increasing emphasis is being placed on making unit testing a central part of software development.

It should be noted that it is extremely difficult—and in many cases impossible—to prove that software "works" and is error free. The fact that a set of tests finds no defects does not prove that the software is error free. 'Absence of evidence is not evidence of absence!'. This was discussed in the late 1960s and early 1970s by Dijkstra and can be summarised as:

Testing shows the presence, not the absence of bugs

Testing to establish that software components fulfil their contract involves checking operations against their requirements. Although this does happen at development time, it forms a major part of QA and user acceptance testing.

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_36

It should be noted that with the advent of test-driven development, the emphasis on testing against requirements during development has become significantly higher.

There are of course many other aspects to testing; for example, performance testing helps identify how a systems will perform as various factors that affect that system change. For example, as the number of concurrent requests increase, as the number of processors used by the underlying hardware changes, as the size of the database grows.

However you view testing, the more testing applied to a system the higher the level of confidence that the system will work as required, and therefore the lower the risk of building a business around that system.

36.3 What Should Be Tested?

What aspects of your software system should be subject to testing? In general anything that is repeatable should be subject to formal (and ideally automated) testing. This includes (but is not limited to):

- The build process for all technologies involved.
- The deployment process to all platforms under consideration.
- The installation process for all runtime environments.
- The upgrade process for all supported versions (if appropriate).
- The performance of the system/servers as loads increase.
- The stability for systems that must run for any period of time (e.g. 24 x 7 systems).
- The backup process.
- The security of the system.
- The recovery ability of the system on failure.
- The functionality of the system.
- The integrity of the system.

Notice that only the last two of the above list might be what is commonly considered areas that would be subject to testing. However, to ensure the quality of the system under consideration, all of the above are relevant. In fact, testing should cover all aspects of the software development lifecycle and not just the QA phase. During requirements gathering testing is the process of looking for missing or ambiguous requirements. During this phase consideration should also be made with regard to how the overall requirements will be tested, in final software system. Test planning should also look at all aspects of the software to test for functionality, usability, legal compliance, conformance to regulatory constraints, security, performance, availability, resilience, etc. Testing should be driven by the need to identify and reduce risk.

36.4 Types of Testing

As indicated in Fig. 36.1 there are a number of different types of testing that are commonly used within industry. These types are:

- **Unit Testing** which is used to verify the behaviour of individual components.
- **Integration Testing** that tests that when individual components are combined together to provide higher-level functional units that the combination of the units operates appropriately.
- **Regression Testing**. When new components are added to a system, or existing components are changed, it is necessary to verify that the new functionality does not break any existing functionality. Such testing is known as regression testing.
- **Performance Testing** is used to ensure that the systems' performance is as required and, within the design requirements, is able to scale as utilization increases.
- **Security Testing** ensures that access to the system is controlled appropriately given the requirements. For example, for an online shopping system there may be different security requirements depending upon whether you are browsing the store, purchasing some products or maintaining the product catalogue.
- **Usability Testing** which may be performed by a specialist usability group and may involved filming of users while they use the system.

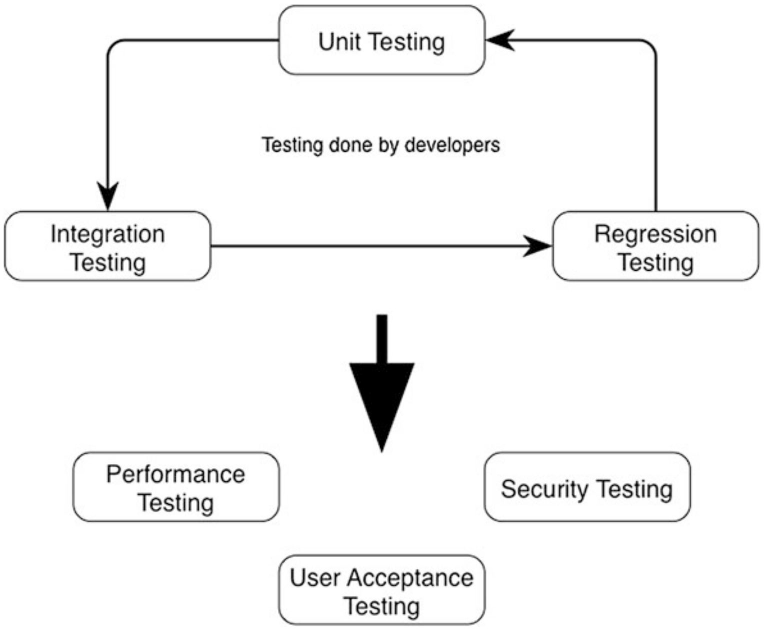

Fig. 36.1 Types of testing

- **User Acceptance Testing** validates that the system actually meets the user requirements and conforms to required application integrity.

Key testing approaches are discussed in the remainder of this section.

36.4.1 Unit Testing

A unit can be as small as a single function or as large as a subsystem but typically is a class, object, self-contained library (API) or Web page.

By looking at a small self-contained component an extensive set of tests can be developed to exercise the defined requirements and functionality of the unit.

Unit testing typically follows a *white box* approach, (also called *Glass Box* or *Structural* testing), where the testing utilises knowledge and understanding of the code and its structure, rather than just its interface (which is known as the *black box* approach).

In *white box* testing, test coverage is measured by the number of code paths that have been tested. The goal in unit testing is to provide 100% coverage: to exercise every instruction, all sides of each logical branch, all called objects, handling of all data structures and files, (including the absence of a file) normal and abnormal termination of all loops, etc. Of course this may not always be possible, but it is a goal that should be aimed for. Many automated test tools will include a code coverage measure so that you are aware of how much of your code has been exercised by any given set of tests.

Unit testing is almost always automated—there are many tools to help with this, perhaps the best known being JUnit for Java and ScalaTest for Scala (with similar tools being available for other languages). Developers write test and stubs, allowing us to:

- focus on testing the unit
- simulate data or results from calling another unit (representative good and bad results.)
- try to create data-driven tests for maximum flexibility and repeatability.
- Often rely on *mock* objects that represent elements outside the unit that it must interact with.

Having the tests automated means they can be run frequently, at the very least after initial development and after each change that affects the unit.

Once confidence is established in the correct functioning of one unit, developers can then use it to help test other units with which it interfaces, forming larger units that can also be unit tested or, as the scale gets larger, put through integration testing.

36.4.2 Integration Testing

Integration testing is where several units (or modules) are brought together to be tested as an entity in their own right. Typically integration testing aims to ensure that modules interact correctly and the individual unit developers have interpreted the requirements in a consistent manner.

An integrated set of modules can be treated as a unit and unit tested in much the same way as the constituent modules but usually working at a "higher" level of functionality. Integration testing is the intermediate stage between unit testing and full system testing.

Therefore integration testing focuses on the interaction between two or more units to make sure that those units work together successfully and appropriately. Such testing is typically conducted from the bottom up but may also be conducted top down using mocks or stubs to represented called or calling functions. An important point to note is that you should not aim to test everything together at one (so-called *Big Bang* testing) as it is more difficult to isolate bugs so that they can be rectified. This is why it is more common to find that integration testing has been performed in a bottom-up style.

36.4.3 System Testing

System testing aims to validate that the combination of all the modules, units, data, installation, configuration, etc. operates appropriately and meets the requirements specified for the whole system. Testing the system has a whole typically involves testing the top most functionality or behaviours of the system. Such behaviour-based testing often involves end users and other stake holders who are less technical. To support such tests a range of technologies have evolved that allows a more *English* style for test descriptions. Cucumber is one example of a behaviour-driven development system, and ScalaTest has a behavioural aspect to its testing styles.

36.4.4 Installation Testing

Installation testing is the testing of full, partial or upgrade install processes. It also validates that the installation and transition software needed to move to the new release for the product is functioning properly. Typically, it

- verifies that the software may be completely uninstalled through its back-out process
- determines what files are added, changed or deleted on the hardware on which the program was installed

- determines whether any other programs on the hardware are affected by the new software that has been installed
- determines whether the software installs and operates properly on all hardware platforms and operating systems that it is supposed to work on.

36.4.5 Smoke Tests

A smoke test is a test or suite of tests designed to verify that the fundamentals of the system work. Smoke tests may be run against a new deployment or a patched deployment in order to verify that the installation performs well enough to justify further testing. Failure to pass a smoke test would halt any further testing until the smoke tests pass. The name derives from the early days of electronics: If a device began to smoke after it was powered on, testers knew that there was no point in testing it further. For software technologies, the advantages of performing smoke tests include:

- Smoke tests are often automated and standardised from one build to another.
- Because smoke tests test things that one expects to work, when they fail, one might suspect that the program may have been built with a wrong file, or that the new build introduced a new bug.
- If a system is built daily, it should be smoke tested daily.
- It will be necessary to periodically add to the smoke tests as new functionality is added to the system.

36.5 Automating Testing

The actual way in which tests are written and executed needs careful consideration. In general, we wish to automate as much of the testing process as is possible as this makes it easy to run the tests and also ensures not only that all tests are run but that they are run in the same way each time. In addition, once an automated test is set up it will typically be quicker to re-run that automated test than to manually repeat a series of tests. However, not all of the features of a system can be easily tested via an automated test tool and in some cases the physically environment may make it harder to automated tests.

Typically, most unit testing is automated and most acceptance testing is manual. You will also need to decide which forms of testing must take place. Most software projects should have unit testing, functional testing and acceptance testing as a necessary requirement. Not all projects will implement performance or regression testing, but you should be careful about omitting any stage of testing and be sure it is not applicable.