6
From Requirements To Test Cases

Program testing can be used to show the presence of bugs, but never to show their absence!
- Edsger Dijkstra

This chapter provides a base of knowledge aimed to help software engineers to write meaningful test cases. The starting point for this process is the understanding of the requirements of the system being tested. Without that information, it is not feasible to design nor implement valuable tests. After that, several actions might be executed before the actual coding of the tests, namely, test planning and test design. Once we start the test coding process, we need to have in mind a set of principles to write code right, and also a set of anti-patterns and bad smells to be avoided. All this information is provided in this chapter in form of the following sections:

- **The importance of requirements**: This section provides a general overview of the software development process, started by the statement of some needs to be covered by a software system, and followed by several stages, typically including analysis, design, implementation, and tests.
- **Test planning**: A document called *test plan* can be generated at the beginning of a software project. This section reviews the structure of a test plan according to the IEEE 829 Standard for Test Documentation. As we will discover, the complete statement of a test plan is a very fine-grained process, especially recommended for large projects in which the communication among the team is a key aspect for the success of the project.

- **Test design:** Before starting the coding of the tests, it is always a good practice to think about the blueprint of these tests. In this section, we review the major aspects to be taken into consideration to design properly our tests. We put the accent on the test data (expected outcome), which feed the test assertions. In this regard, we review some black-box data generation techniques (equivalence partitioning and boundary analysis) and white-box (test coverage).
- **Software testing principles**: This section provides a set of best-practices which can help us write our tests.
- **Test anti-patterns**: Finally, the opposite side is also reviewed: what are the patterns and code smells to be avoided when writing our test cases.

The importance of requirements

Software systems are built to satisfy some kind of need to a group of consumers (final users or customer). Understanding those needs is one of most challenging problems in software engineering due to the fact that it is quite common that consumer needs are nebulous (especially in the early stages of the project). Moreover, it is also common that these needs can be deeply changed throughout the project lifetime. Fred Brooks, a well-known software engineer, and computer scientist, defines this problem in his seminal book *The Mythical Man-Month (*1975*)*:

> *The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements … No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later.*

In any case, consumer's needs are the touchstone for any software project. From these needs, a list of features can emerge. We define a feature as a high-level description of a software system functionality. From each feature, one or more requirements (functional and non-functional) should be derived. A requirement is everything that be true about the software, in order to meet the consumer's expectations. Scenarios (real-life examples rather than abstract descriptions) can be useful for adding details to the requirements description. The group of requirements and/or list of features of the software system are often known as specification.

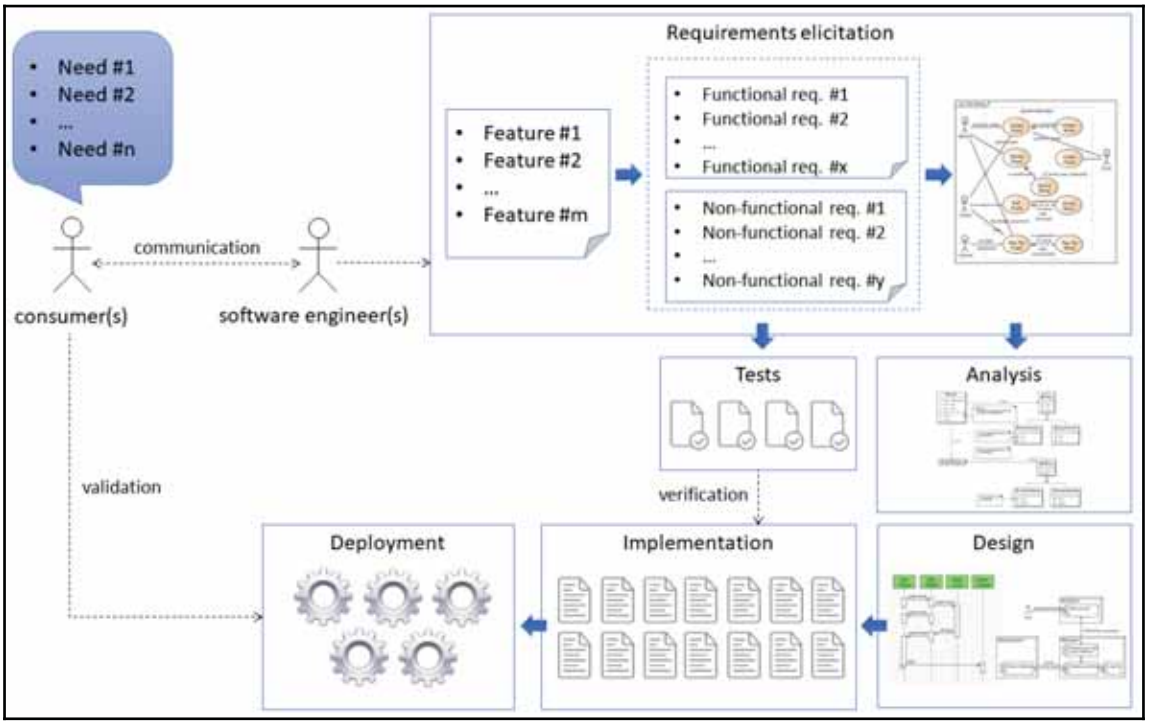
In software engineering, the stage of defining the requirements is called **requirements elicitation**. In this stage, software engineers need to clarify *what* problem they are trying to solve. At the end of this phase, it is a common practice to start the modeling of the system. To that aim, a modeling language (typically UML) is employed to create a group of diagrams. The UML diagrams, which typically fits in the elicitation stage is the use case diagram (model of the functionality of the system and its relationship with the involved actors).

Modeling is not always carried out in all software projects. For example, agile methodologies are more based on the principle of sketching rather than in a formal modeling strategy.

After elicitation, requirements should be refined in the **analysis** stage. In this phase, the stated requirements are analysed in order to resolve incomplete, ambiguous of contradictory issues. As a result, in this stage it is likely to continue modeling, using for example high-level class diagrams not linked to any specific technology yet. Once the analysis is clear (that is, the *what* of the system), we need to find out *how* to implement it. This stage is known as **design**. In the design phase, the guidelines of the project should be established. To that aim, an architecture of the software system is typically derived from the requirements. Again, the modeling techniques are broadly employed to carry out different aspects of the design. There is a bunch of UML diagrams that can be used at this point, including structural diagrams (component, deployment, object, package, and profile diagram) and behavioral diagrams (activity, communication, sequence, or state diagram). From the design, the actual **implementation** (that is, coding) can start.

The amount of modeling carried out in the design stage varies significantly depending on different factors, including the type and size of the company producing the software (multinationals, SMEs, governmental, and so on), the development process (waterfall, spiral, prototyping, agile, and so on), the type of project (enterprise, open source, and so on), the type of software (custom made software, commercial off-the-shelf, and so on), and even the background of the people involved (experience, career, and so on). All in all, the designs need to be understood as a way of communication between the different roles of software engineers participating in the project. Typically, the bigger the project, the more necessary a fine-grained design based on different modeling diagrams is.

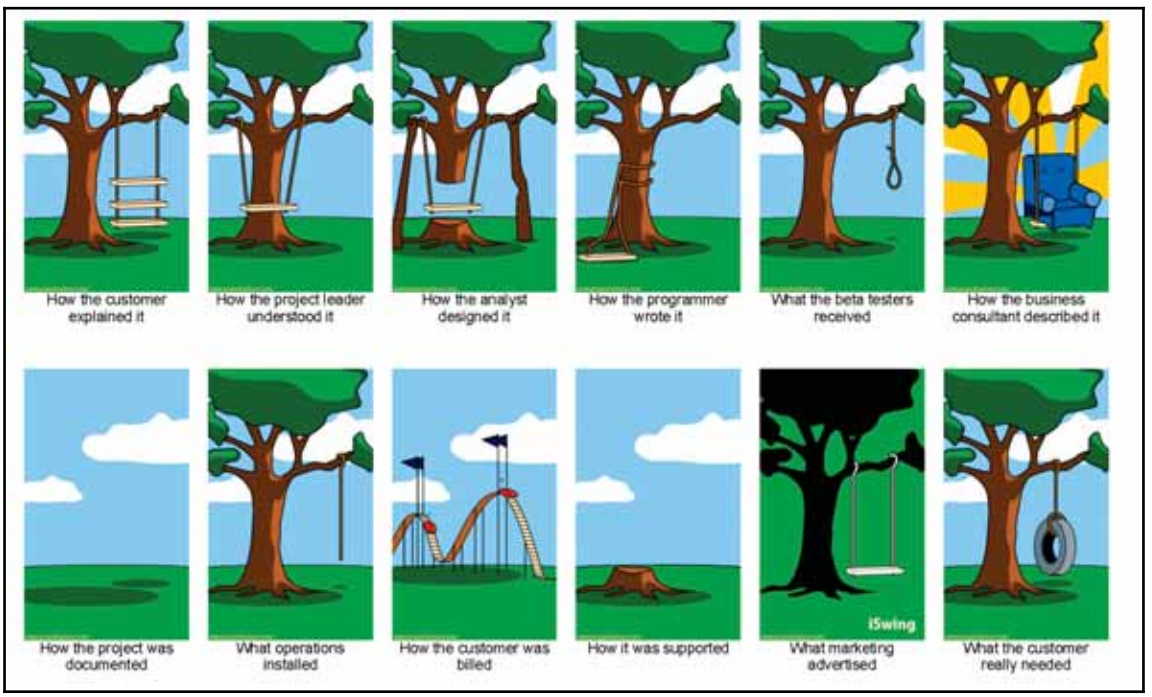
Concerning **tests**, in order to make a proper test plan (see next section for further details), again we need to use the requirements elicitation data, that is, the list of requirements and/or features. In other words, in order to verify our system, we need to to know beforehand what we expect from it. Using the classic definition proposed by Barry Boehm (see `chapter 1`, *Retrospective On Software Quality And Java Testing*), verification is used to answer the question *Are we building the product right?* To that, we need to know the requirements, or at least, the desired features. In addition to verification, it would be desirable to carry out some validation (according to Boehm: *Are we building the right product?*). This is necessary since sometimes there is a gap between what has been specified (the features and requirements) and the real needs of the consumer. Therefore, validation is a high-level assessment method, and to carry out it, the final consumer can be involved (validating the software system once it is deployed). All these ideas are depicted in the following picture:



Software engineering generic development process

There is no universal workflow for the terms presented so far (communication, requirement elicitation, analysis, design, implementation/test, and deployment). In the preceding diagram, it follows a linear process flows, nevertheless, in practice, it can follow an iterative, evolutionary, or parallel workflow.

To illustrate the potential problems involved in the different phases in software engineer (analysis, design, implementation, and so on), it is worth to review the classical cartoon *How project really works?* The original source of this picture is unknown (there are versions dating back to the 1960s). In 2007, a site called *Project Cartoon* emerged (`http://www.projectcartoon.com/`), allowing to customize the original cartoon with new scenes. The following chart is the version 1.5 of the cartoon provided on that site:



How projects really work, version 1.5 (illustrated created by `www.projectcartoon.com`)

If we think about this picture, we discover that the root of the problems comes from the requirements, badly explained by the customer at the beginning, and worst understood by the project leader. From that point, the whole software engineering process turns into the *Chinese whispers* children game. To solve all these problems is out of the scope of this book, but as a good start, we need to take special care in the requirements, which guide the whole process, including, of course, the tests.

Test planning

A first step in the testing path can be the generation of a document called *test plan*, which is the blueprint to conduct software testing. This document describes the objective, scope, approach, focus, and distribution of the testing efforts. The process of preparing such document is a useful way to think about the needs to verify of a software system. Again, this document is especially useful when the size of the SUT and the involved team is large, due to the fact that the separation of work in different roles makes the communication a potential deterrent for the success of the project.

A way to create a test plan is to follow the IEEE 829 Standard for Test Documentation. Although this standard might be too much formal for the most of software projects, it might be worth to review the guidelines proposed in this standard, and use the parts needed (if any) in our software projects. The steps proposed in IEEE 829 are the following:

1. **Analyze the product**: This part reinforces the idea of extracting the understanding the requirements of the system from the consumer needs. As already explained, it is not possible to test a software if no information about it is available.
2. **Design the test strategy**: This part of the plan contains several parts, including:
    - Define scope of testing, that is, the system components to be tested (in scope) and those parts which do not (out of scope). As explained later, exhaustive testing is not feasible, and we need to choose carefully what is going to be tested. This is not a simple choice, and it can be determined by different factors, such as precise customer requests, project budget and timing, and skills of the involved software engineers.
    - Identify testing type, that is, which levels of tests should be conducted (unit, integration, system, acceptance) and which test strategy (black box, white box, non-functional).
    - Document risks, that is, potential problems which might cause different issues in the project.
3. **Define the test objectives**: In this part of the plan, the list of features to be tested are listed together with the target of testing each one.

4. **Define the test criteria**: These criteria are typically made up by two parts, namely:
    - Suspension criteria, for instance the percentage of failed tests in which the development of new features is suspended until the team solves all the failures.
    - Exit criteria, for example, the percentage of critical tests that should be passed to proceed to next phase of development.
5. **Resource planning**: This part of the plan is devoted to summarize the resources required to carry out the testing activities. It could be personnel, equipment, or infrastructure.
6. **Plan test environment**: It consists of the software and hardware setup on which test are going to be executed.
7. **Schedule and estimation**: In this phase, managers are supposed to break out the whole project into small tasks estimating the efforts (person-month).
8. **Determine test deliverables**: Determine all the documents that has to be maintained to support the testing activities.

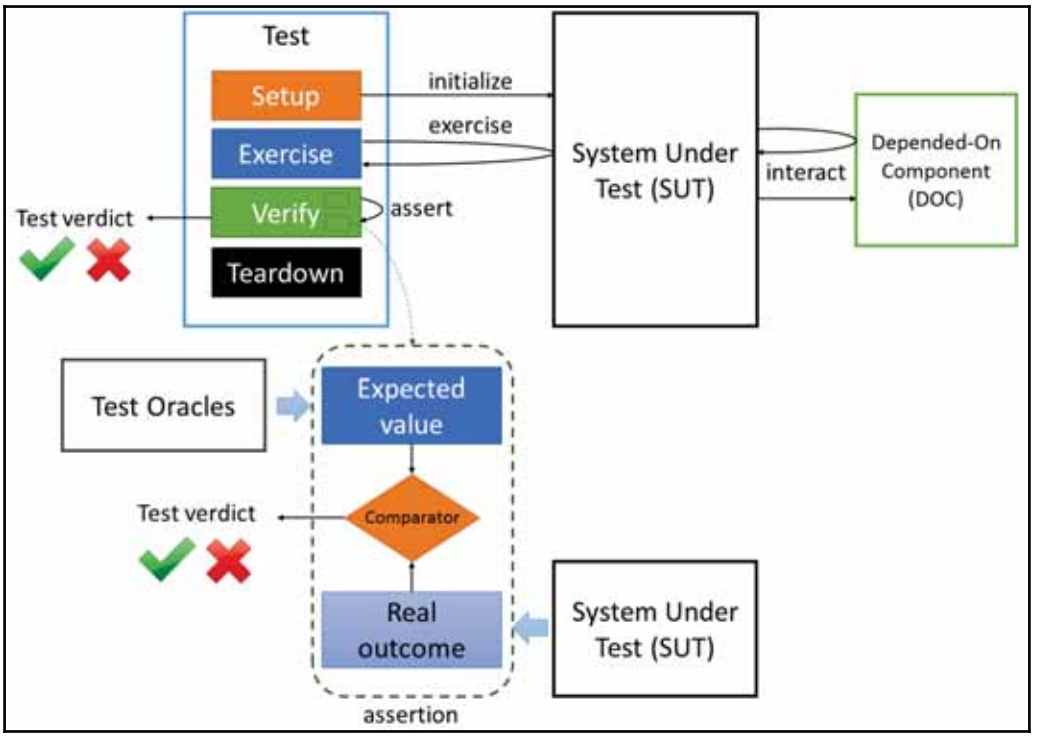
As can be seen, test planning is a complex task, typically carried out in large projects by managers. In the rest of this chapter we continue discovering how to write test cases, but hereinafter from a point of view closest to the actual test coding.

Test design

In order to design properly a test, we need to define specifically what needs to be implemented. To that aim, it is important to remember what is the generic structure of a test, already explained in `chapter 1`, *Retrospective On Software Quality And Java Testing*. Therefore, for each test we need to define:

- What is test fixture, that is, the required state in the SUT to carry out the test? This is done at the beginning of the test in the stage called setup. At the end of the test, the test fixture might be released in the stage called teardown.
- What is the SUT, and if we are doing unit tests, which are its DOC(s)? Unit test should be in isolation and therefore we need to define test doubles (typically mocks or spies) for the DOC(s).

- What are the assertions? This a key part of tests. Without assertions, we cannot claim that a test is actually made. In order to design assertion, it is worth to recall which is its generic structure. In short, an assertion consists in the comparison of some expected value (test data) and the actual outcome obtained from the SUT. If any of the assertions is negative, the test will be declared as failed (test verdict):



Test cases and assertions general schema

Test data plays a crucial role in the testing process. The source of test data is often called test oracles, and typically can be extracted from the requirements. Nevertheless, there are some others commonly used sources for tests oracles, for example:

- A different program, which produces the expected output (inverse relationship).
- A heuristic or statistical oracle that provides approximate results.
- Values based on the experience of human experts.

Moreover, test data can be derived, depending on the underlying testing technique. When using black-box testing, that is, exercise some specific requirement based using some input and expecting some output, different techniques can be employed, such as equivalence partitioning or boundary analysis. On the other side, if we are using white-box testing, the structure is the basis for our test and therefore the test coverage will be key to select the test input which maximizes these coverage rates. In the following sections, these techniques are reviewed.

Equivalence partitioning

Equivalence partitioning (also known as equivalence class partitioning) is a black-box technique (that is, it relies in the requirements of the system) aimed to reduce the number of tests that should be executed against a SUT. This technique was first defined by Glenford Myers in 1978 as:

> *"A technique that partitions the input domain of a program into a finite number of classes [sets], it then identifies a minimal set of well-selected test cases to represent these classes."*

In other words, equivalence partitioning provides a criteria to answer the question *How many tests do we need?* The idea is to divide all possible input test data (which often is a enormous number of combinations) in a set of values for which we assume to be processed in the same way by the SUT. We call equivalence classes to these sets of values. The idea is that testing one representative value within the equivalence class is consider sufficient because it is assumed that all the values are processed in the same way by the SUT.

Typically, the equivalence classes for a given SUT can be grouped in two types: valid and invalid inputs. The equivalence partitioning testing theory ensures that only one test case of each partition is needed to evaluate the behavior of the program for the related partition (both the valid and the invalid classes). The following process describes how to systematically carry out the equivalence partitioning for a given SUT:

1. First, we determine the domain of all possible valid inputs for a SUT. To find out these values, we rely on the specification (features or functional requirements). Our SUT is supposed to process these values (valid equivalence class) correctly.
2. If our specification establishes that some elements of the equivalence class are processed differently, they should assigne to another equivalence class.
3. The values outside this domain can be seen as another equivalence class, this time for invalid inputs.

4. For every single equivalence class, a representative value is chosen. This decision is an heuristic process typically based on the tester experience.
5. For every test input, the proper test output is also selected, and with these values we will be able to complete our test case (test exercise and assertions).

Boundary analysis

As any programmer knows, faults often appear at the boundary of a equivalence class (for example, the initial value of an array, the maximum value for a given range, and so on). Boundary value analysis is a method, which complements equivalence partitioning by looking at the boundaries of the test input. It was defined by the National Institute of Standards and Technology (NIST) in 1981 as:

> *"A selection technique in which test data are chosen to lie along 'boundaries' of the input domain [or output range] classes, data structures, and procedure parameters."*

All in all, to apply boundary value analysis in our tests, we need to evaluate our SUT exactly in the borders of our equivalence class. Therefore, typically two tests cases are derived using this approach: the upper and the lower boundary of the equivalence class.

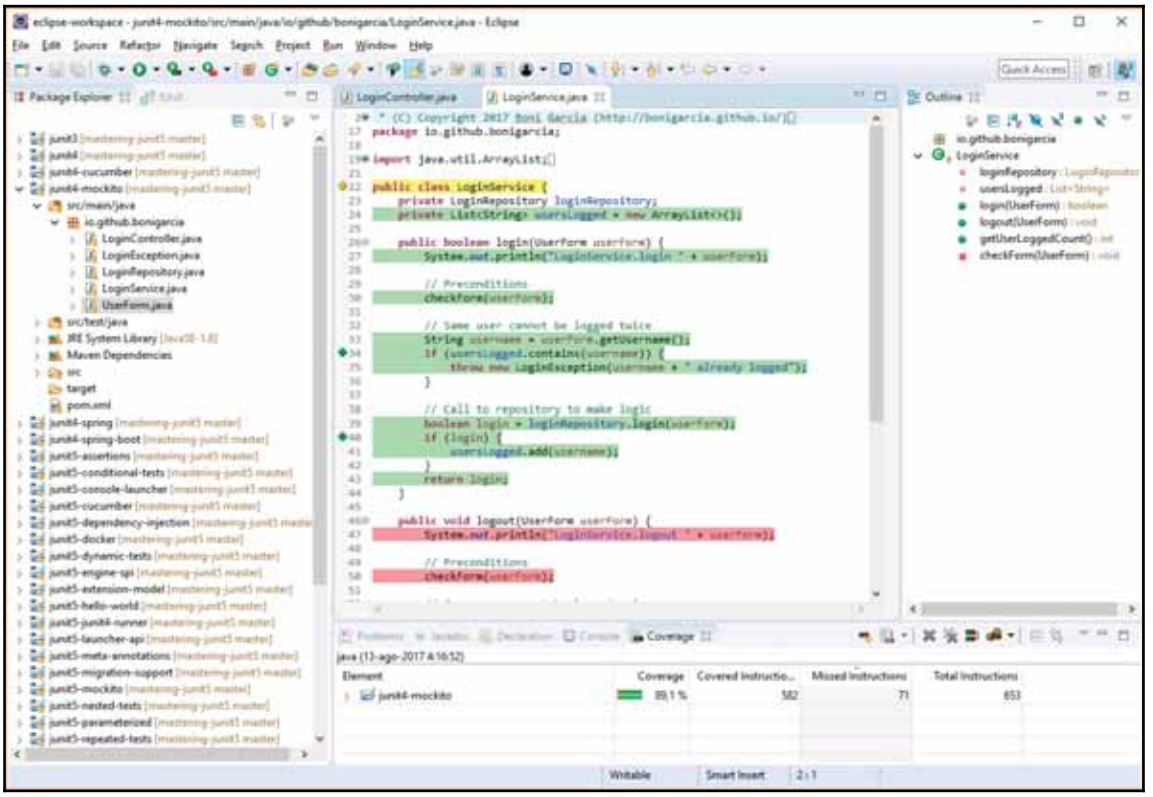
Test coverage

Test coverage is the rate of code in SUT that is exercised for any of their tests. Test coverage is very useful to finding untested parts of our SUT. Therefore, it can be the perfect white box technique (structural) to complement the black box (functional). As a general rule, a test coverage rate of 80% or above is considered reasonable.

There are different Java libraries, which allows to make test coverage in a simple manner, for instance:

- Cobertura (`http://cobertura.github.io/cobertura/`): It is an open source reporting tool, which can be executed using Ant, Maven, or directly using the command line.

- EclEmma (`http://www.eclemma.org/`): It is an open source code coverage tool for Eclipse. As of Eclipse 4.7 (Oxygen), EclEmma is integrated out of the box in the IDE. The following screenshot shows an example on how EclEmma highlights the code coverage on a Java class in Eclipse:



Test coverage with EclEmma in Eclipse 4.7 (Oxygen)

- JaCoCo (`http://www.jacoco.org/jacoco/`): It is an open source code coverage library created by the EclEmma team based on other old coverage library called EMMA (`http://emma.sourceforge.net/`). JaCoCo is available as a Maven dependency.
- Codecov (`https://codecov.io/`): It is a cloud solution offering a friendly code coverage web dashboard. It is free for open source projects.

Software testing principles

Exhaustive testing is the name given to a test approach, which uses all possible combinations of test inputs to verify a software system. This approach is only applicable to tiny software systems or components with a close finite number of possible of operations and allowed data. In the majority of software systems, it is not feasible to verify every possible permutation and input combination, and therefore exhaustive testing is just a theoretical approach.

For that reason, it is said that the absence of defects in a software system cannot be proved. This was stated by the computer science pioneer Edsger W. Dijkstra (see quote at beginning of this chapter). Thus, testing is, at best, sampling, and it must be carried out in any software project to reduce the risk of system failures (see `chapter 1`, *Retrospective On Software Quality And Java Testing,* to recall the software defect taxonomy). Since we cannot test everything, we need to test properly. In this section, we review a set of best practices to write effective and efficient test cases, namely:

- **Tests should be simple**: The software engineer writing the test (call him or her tester, programmer, developer, or whatever) should avoid attempting to test his or her program. In regards to testing, the right answer to the question *Who watches the watchmen?* Should be nobody. Our test logic should be simple enough to avoid any kind of meta-testing, since this would lead to a recursive problem out of any logic. Indirectly, if we keep tests simple, we also obtain another desirable feature: tests will be easy to maintain.
- **Do not implement simple tests**: One thing is make simple tests, and another very different stuff is to implement dummy code, such as getter or setters. As introduced before, test is at best sampling, and we cannot waste precious time in assessing such kind of part of our codebase.
- **Easy to read**: The first step is to provide a meaningful name for our test method. In addition, thanks to the JUnit 5 `@DisplayName` annotation, we can provide a rich textual description, which defines without Java naming constraints the goal of the test.
- **Single responsibility principle**: This is a general principle of computer programming that states that every class should have responsibility of a single functionality. It is closely related to the metric of cohesion. This principle is very important to be accomplished when coding tests: a single test should be only referred to a given system requirement.

- **Test data is key**: As described in the section before, the expected outcome from the SUT is a central part of the tests. The correct management of these data is critical to create effective tests. Fortunately, JUnit 5 provides a rich toolbox to handle test data (see section *Parameterized tests* in `chapter 4`, *Simplifying Testing With Advanced JUnit Features*).
- **Unit test should be executed very fast**: A commonly accepted rule of thumb for the duration of unit test is that a unit test should last a second at the most. To accomplish that goal, it is also required that unit test isolates properly the SUT, doubling properly its DOCs.
- **Test must be repeatable**: Defects should be reproduced as many times as required for developers to find the cause of the bug. This is the theory, but unfortunately this is not always applicable. For example, in multi-threaded SUT (a real-time or server-side software systems), race conditions are likely to occur. In those situations, non-deterministic defects (often called *heisenbugs*) might be experienced.
- **We should test positive and the negative scenarios**: This mean that we need to write tests with for input condition that assess the expected outcome, but we also need to verify what the program is not supposed to do. In addition to meet its requirements, programs must be tested to avoid unwanted side effects.
- **Testing cannot be done only for the sake of coverage**: Just because all parts of the code have been touched by some tests, we cannot assure that those parts have been thoroughly tested. For that to be true, tests have to analyzed in terms of reduction of risks.

The psychology of testing

From a psychological point of view, the objective of testing should be executing a software system with the intent of finding defects. Understanding the motivation of that claim can make the difference in the success of our tests.

Human beings tend to be goal oriented. If we carry out tests to demonstrate that a program has no errors, we will tend to implement tests selecting test data with a low probability of causing program failures. On the other hand, if the objective is to demonstrate that a program has errors, we will increase the probability of finding them, adding more value to the program than the former approach. For that reason, testing is often considered as a destructive process, since testers are supposed to prove that the SUT has errors.

Moreover, trying to demonstrate that errors are present in the software is a goal feasible, while trying to demonstrate their absence, as explained before, it is impossible. Again, psychology studies tell us that people perform poorly when they know that a task is infeasible.

Test anti-patterns

In software design, a pattern is a reusable solution to solve recurring problems. There are a bunch of them, including for example singleton, factory, builder, facade, proxy, decorator, or adapter, to name a few. Anti-patterns are also patterns, but undesirable ones. Concerning to testing, it is worth to know some of these anti-patterns to avoid them in our tests:

- **Second class citizens**: Test code containing a lot of duplicated code, making it hard to maintain.
- **The free ride** (also known as *Piggyback*): Instead of writing a new method to verify another feature/requirement, a new assertion is added to an existing test.
- **Happy path**: It only verifies expected results without testing for boundaries and exceptions.
- **The local hero**: A test dependent to some specific local environment. This anti-pattern can be summarized in the phrase *It works in my machine.*
- **The hidden dependency**: A test that requires some existing data populated somewhere before the test runs.
- **Chain gang**: Tests that must be run in a certain order, for example, changing the SUT to a state expected by the next one.
- **The mockery**: A unit test that contains too much test doubles that the SUT is not even tested at all, instead of returning data from test doubles.
- **The silent catcher**: A test that passes even if an unintended exception actually occurs.
- **The inspector**: A test that violates encapsulation that any refactor in the SUT requires reflecting those changes in the test.
- **Excessive setup**: A test that requires a huge setup in order to start the exercise stage.
- **Anal probe**: A test which has to use unhealthy ways to perform its task, such as reading private fields using reflection.
- **The test with no name**: Test methods name with no clear indicator about what it is being tested (for example, identifier in a bug tracking tool).
- **The slowpoke**: A unit test which lasts over few seconds.

- **The flickering test**: A test which contains race conditions within the proper test, making it to fail from time to time.
- **Wait and see**: A test that needs to wait a specific amount of time (for example, `Thread.sleep()`) before it can verify some expected behavior.
- **Inappropriately shared fixture**: Tests that use a test fixture without even need the setup/teardown.
- **The giant**: A test class that contains a huge number of tests methods (God Object).
- **Wet floor**: A test that creates persisted data but it is not clean up at when finished.
- **The cuckoo**: A unit test which establishes some kind of fixture before the actual test, but then the test discards somehow the fixture.
- **The secret catcher**: A test that is not making any assertion, relying on an exception to be thrown and reporting by the testing framework as a failure.
- **The environmental vandal**: A test which requires the use of given environment variables (for instance, a free port number to allows simultaneous executions).
- **Doppelganger**: Copying parts of the code under test into a new class to make visible for the test.
- **The mother hen**: A fixture which does more than the test needs.
- **The test it all**: Tests that should not break the Single Responsibility Principle.
- **Line hitter**: A test without any kind of real verification of the SUT.
- **The conjoined twins**: Tests that are called *unit tests* but are really integration tests since there is no isolation between the SUT and the DOC(s).
- **The liar**: A test that does not test what was supposed to test.

Code smells

Code smells (also known as *bad smell* when referred to software) are undesirable symptoms within the source code. Code smells are not problematic per se, but they can evidence some kind of issue nearby.

As described in previous sections, tests should be simple and easy to read. With that promises, code smells should be present in our tests under no circumstances. All in all, generic code smells might be avoided in our tests. Some of the most common code smells are the following:

- **Duplicated code**: Cloned code is always a bad idea in software, since it breaks the principle **Don't Repeat Yourself** (**DRY**). This problem is even worst in tests, since test logic must be crystal clear.
- **High complexity**: Too many branches or loops may be potentially simplified into smaller pieces.
- **Long method**: A method that has grown too large is always problematic, and it is a very bad symptom when this method is a test.
- **Unappropriated naming convention**: Variables, class, and method names should be concise. It is considered a bad smell to use very long identifiers, but also use excessive short (or meaningless) ones.

Summary

The starting point for the test design should be the list of requirements. If these requirements have not been formally elicited, at least we need to know the SUT features, which reflects the software needs. From this point, several strategies can be carried out. As usual, there is no unique path to reach our goal, which in the end should be reducing the risks of the project.

This chapter reviewed a process aimed to create effective and efficient tests cases. This process involves the analysis of requirements, definition of a test plan, design of test cases, and finally writing the test cases. We should be aware that, even though software testing is technical task, it involves some important considerations of human psychology. These factors should be known by software engineers and testers in order to follow know best practices and also avoiding common mistakes.

In `chapter 7`, *Testing management*, we are going to understand how software testing activities are managed in a living software project. To that, first we review when and how to carry out testing in the common software development processes, such as waterfall, spiral, iterative, spiral, agile, or test-driven development. Then, the server-side infrastructure (such as Jenkins or Travis) aimed to automate the software development process in the context of JUnit 5 is reviewed. Finally, we learn how to keep track of the defects found with the Jupiter tests using the so-called issue tracking systems and test reporting libraries.