UD04: Designing tests and testing

1. Planning the tests.

Throughout the software development process, from the design phase, in the implementation and once the application is developed, it is necessary to carry out a set of tests (Process that allows verifying and revealing the quality of a software product. They are used to identify possible failures of implementation, quality or usability of a program), which allow verifying that the software being created is correct and complies with the specifications imposed by the user.

In the software development process, we are going to find a set of activities, where it is very easy for human error to occur. These human errors can be: an incorrect specification of the objectives, errors produced during the design process and errors that appear in the development phase.

By performing software tests, the verification tasks (Process by which it is verified that the software meets the specified requirements) and validation (Process that verifies if the software does what the user wanted. verified) of the software. Verification is the verification that a system or part of a system complies with the imposed conditions. The verification checks if the application is building correctly. Validation is the process of evaluating the system or one of its components, to determine if it satisfies the specified requirements.

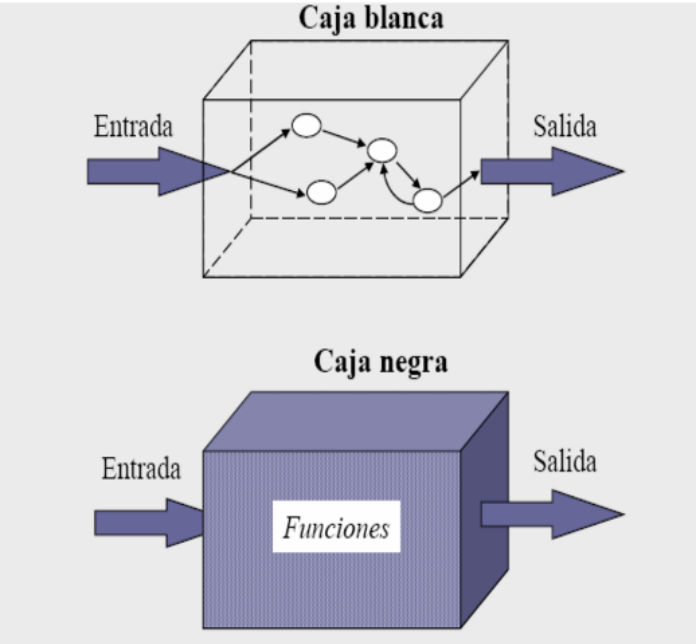
To carry out the testing process efficiently, you need to implement a testing strategy. Following the Spiral Model (Software life cycle life cycle model, in which the activities are formed in a spiral. Each loop or iteration represents a set of activities), the tests would start with the unit test, where the implemented code would be analyzed and we would continue in the integration test, where attention is paid to the design and construction of the software architecture. The next step would be the validation test, where it is verified that the built system complies with what is established in

the software requirements analysis. Finally, the system test is reached, which verifies the total operation of the software and other elements of the system.

2. Test types.

There is no official or formal classification on the various types of software tests. In software engineering, we find two fundamental approaches:

- **Black Box Testing** (Black Box Testing): when an application is tested using its external interface, without worrying about its implementation. Here the fundamental thing is to check that the results of the execution of the application are as expected, based on the inputs it receives.
- **White Box Testing** (White Box Testing): in this case, the application is tested from within, using its application logic.



A Black Box type test is carried out without having to know the structure or the internal functioning of the system. When this type of test is carried out, only the appropriate inputs that the application should receive are known, as well as the corresponding outputs, but the process by which the application obtains these results is not known.

In contrast to the above, a White Box test will directly analyze and test the application code. As derived from the above, to carry out a White Box test, a specific knowledge of the code is necessary, in order to be able to analyze the test results.

	Types of software tests
Unit tests	With it you will test the correct operation of a code module
Load tests	This is the simplest type of performance test. A load test is generally performed to observe the behavior of an application under an expected number of requests. This load can be the expected number of concurrent users using the application and performing a specific number of transactions during the time that the charge lasts. This test can show response times for all important transactions in your application. If the database, application server, etc. are also monitored, then this test may show the bottleneck in the application.
Stress test	This test is normally used to break the application. The number of users added to the application doubles and a load test runs until it breaks. This type of testing is performed to determine the robustness of the application during times of extreme load and helps administrators to determine if the application will perform sufficiently in the event that the actual load exceeds the expected load.
Stability test	This test is normally done to determine if the application can withstand a continued expected load. Generally this test is done to determine if there are any memory leaks in the application
Peak tests	The spike test, as the name suggests, tries to observe the behavior of the system by varying the number of users, both when they go down, and when it has drastic changes in its load. This test is recommended to be carried out with software automated system that allows changes to be made to the number of users while administrators keep a record of the values to be monitored
Structural test	The structural or white box approach focuses on the internal structure of the program (analyzes the execution paths).
Functional test	The functional or black box approach focuses on the functions, inputs and outputs that a specific module or function receives and produces.
Random tests	The random approach consists of using models (in many cases statistical) that represent the possible inputs to the program to create the test cases from them.
Regression tests	They are any type of software tests that try to discover the causes of new errors, lack of functionality, or functional divergences with respect to the expected behavior of the software, induced by changes recently made in parts of the application that were not prone to before the aforementioned change. this kind of error. This implies that the error dealt with occurs as an unexpected consequence of the aforementioned change in the program.

It is common for a software development company to spend 40 percent of the development effort on testing. Why is testing so important? What types of errors are the tests trying to fix?
The tests are very important, since they allow to discover errors in a program, failures in the implementation, quality or usability of the software, helping to guarantee the quality.
The tests try to verify that each component that has been designed, be it a method,

> function, module, etc. performs the function for which it was designed. An attempt is also made to verify that there are conditions in which all the paths of an application are executed.

2.1. Functional.

We are facing evidence of the black box. It is about testing whether the outputs returned by the application, or part of it, are as expected, depending on the input parameters that we pass to it. We are not interested in the implementation of the software, only if it performs the functions expected of it.

Functional testing follows the Black Box testing approach. They would include those activities whose objective is to verify a specific or functional action within the code of an application. Functional tests would attempt to answer the questions can the user do this? or does this application utility work?

Its main task will consist of checking the correct functioning of the components of the computer application. To carry out this type of test, the inputs and outputs of each component must be analyzed, verifying that the result is as expected. Only the inputs and outputs of the system will be considered, without worrying about its internal structure.

If, for example, we are implementing an application that performs a certain scientific calculation, in the approach of functional tests, we are only interested in verifying that before a certain input to that program the result of its execution returns the expected data as a result. This type of test would not consider, in any case, the code developed, nor the algorithm (Ordered set of steps to follow to solve a problem), nor the efficiency, nor if there are unnecessary parts of the code, etc.

Within the functional tests, we can indicate three types of tests:

- **Equivalent partitions:** The idea of this type of functional tests is to consider the smallest possible number of test cases, for this, each test case has to cover as many different inputs as possible. What is intended is to create a set of equivalence classes, where the test of a representative value of the same, in terms of error checking, would be extrapolated to the one that would be achieved by testing any value of the class.
- **Analysis of limit values:** In this case, when implementing a test case, we will choose as input values, those that are at the limit of the equivalence classes.
- **Random tests:** Consists of generating random inputs for the application to be tested. Test generators are often used, which are capable of creating a volume of random test cases, with which the application will be fed. These types of tests are usually used in non-interactive applications, since it is very difficult to generate the appropriate test input sequences for interactive environments.

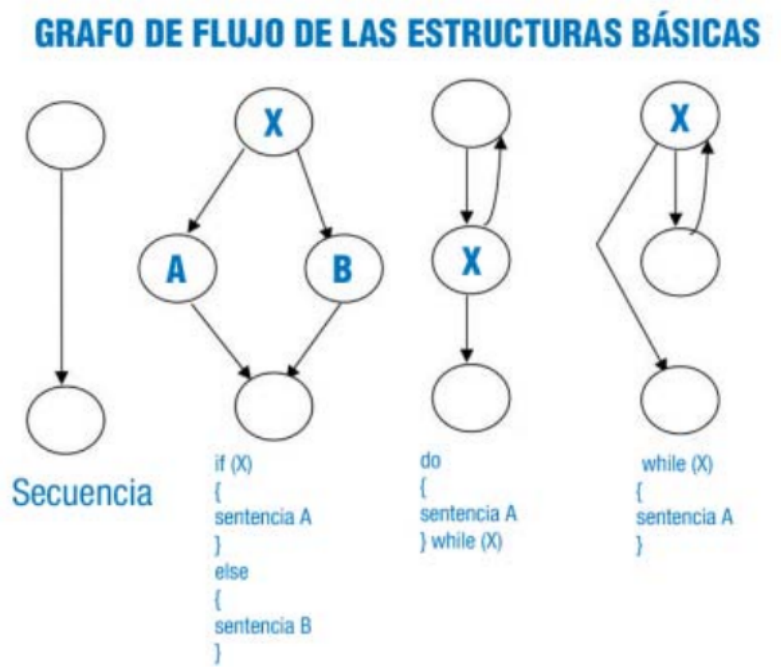
There are other types of functional tests, although they all share the same objective, and that is to verify, only by acting on the application interface, that the results it produces are correct based on the inputs that are introduced to test them.

2.2. Structural.

We have already seen that functional testing focuses on results, on what the application does, but not on how it does it.

To see how the program is running, and thus check its correctness, structural tests are used, which look at the paths that can be traversed:

Structural tests are the White Box test suite. With this type of tests, it is intended to verify the internal structure of each component of the application, regardless of the functionality established for it. This type of test does not pretend to verify the correctness of the results produced by the different components, its function is to verify that all the instructions of the program are going to be executed, that there is no unused code, to verify that the logical paths of the program are going to go, etc.



These types of tests are based on criteria of logical coverage, the fulfillment of which determines the greater or lesser security in the detection of errors. The coverage criteria that are followed are:

- **Coverage of sentences**: sufficient test cases have to be generated so that each instruction of the program is executed at least once.
- **Decision coverage**: it is about creating enough test cases so that each option resulting from a logical test of the program is evaluated at least once to be true and another time to false.
- **Condition coverage**: it is about creating enough test cases so that each element of a condition is evaluated at least once to false and once to true.
- **Coverage of conditions and decisions**: consists of simultaneously fulfilling the two previous ones.
- **Road coverage**: it is the most important criterion. It establishes that each sequence of chained statements must be executed at least once, from the initial statement of the program, to its final statement. The execution of this set of sentences is known as a path. As the number of paths that an application can have can be very large, to perform this test, the number is reduced to what is known as the test path.
- **Test path coverage**: Two variants can be performed, one indicates that each loop should be executed only once, since doing it more times does not increase the effectiveness of the test and another that recommends that each loop be tested three times: the first without entering it, another by executing it once and another by executing it twice.

2.3. Regression.

During the testing process, we will be successful if we detect a possible failure or error. The direct consequence of this discovery involves the modification of the component where it has been detected. This modification can generate collateral errors, which did not exist before. As a consequence, the modification made forces us to repeat tests that we have carried out previously.

The objective of regression tests is to verify that changes to one component of an application do not introduce unwanted behavior or additional errors in other unmodified components.

- Regression tests must be carried out every time a change is made to the system, both to correct an error and to make an improvement. It is not enough to test only the modified or added components, or the functions that are carried out in them, but it is also necessary to control that the modifications do not produce negative effects on the same or other components.

Normally, this type of test involves the repetition of tests that have already been carried out previously, in order to ensure that no errors are introduced that could compromise the operation of other components that have not been modified and confirm that the system works correctly once changes made.

In a broader context, successful software tests are those that result in the discovery of bugs. As a consequence of the discovery of errors, they are corrected, which implies the modification of some component of the software that is being developed, both the program, the documentation and the data that support it. Regression testing is what helps us ensure that these changes do not introduce unwanted behavior or additional errors. Regression testing can be done manually, by rerunning a subset of all test cases, or by using automated tools.

The regression test suite contains three different classes of test classes:

- A representative sample of tests that exercises all the functions of the software;
- Additional tests that focus on the software functions that are likely to be affected by the change;
- Tests that focus on the software components that have changed.

To prevent the number of regression tests from growing too large, they should be designed to include only those tests that deal with one or more kinds of errors in each of the main functions of the program. It is neither practical nor efficient to rerun every test of every function in the program after a change.

3. Procedures and test cases.

According to the IEEE, a test case is a set of inputs, execution conditions and expected results, developed for a particular objective, such as, for example, exercising a specific path of a program or verifying the fulfillment of a certain requirement, including all associated documentation.

Given the complexity of computer applications that are currently being developed, it is practically impossible to test all the combinations that can occur within a program or between a program and the applications that can interact with it. For this reason, when designing test cases, it is always necessary to ensure that with them an acceptable level of probability is obtained that existing errors will be detected.

The tests should seek a compromise between the amount of resources that will be consumed in the test process, and the probability obtained that existing errors will be detected.

There are several procedures for designing test cases:

- **Functional or black box approach**. In this type of test, we focus on that the program, or part of the program that we are testing, receives an input properly and produces a correct output, as well as that the integrity of the external information is maintained. The test does not verify the process, only the results.
- **Structural approach or white box**. In this type of testing, we must focus on the internal implementation of the program. This is to check that the internal operation conforms to

specifications. In this test, all the paths that the execution of the program can follow should be tested.
- **Random focus**. From statistically obtained models, test cases are developed that test the program inputs.

4. Debugging tools.

Every development environment, regardless of the platform, as well as the programming language used, provides a series of debugging tools, which allow us to verify the generated code, helping us to carry out both structural and functional tests.

During the software development process, two types of errors can occur: compilation errors or logic errors. When developing an application in an IDE, be it Visual Studio, Eclipse, or Netbeans, if when writing a statement, we forget a ";", reference a non-existent variable, or use an incorrect statement, a compilation error occurs. When a compilation error occurs, the environment provides us with information on where it occurs and how to solve it. The program cannot be compiled until the programmer fixes that error.

The other types of errors are logical, commonly called bugs, these do not prevent the program from being compiled successfully, since there are no syntactic errors, nor are undeclared variables used, etc. However, logical errors can cause the program to return erroneous results that are not expected, or they can cause the program to terminate prematurely or never terminate.

To solve these types of problems, development environments incorporate a tool known as a debugger. The debugger allows you to monitor the execution of your programs, to locate and eliminate logical errors. A program must compile successfully before it can be used in the debugger. The debugger allows us to analyze the entire program, while it is running. It allows suspending the execution of a program, examining and setting the values of the variables, checking the values returned by a certain method, the result of a logical or relational comparison, etc.

4.1. Breakpoints.

Inside the debugging menu, we find the option to insert breakpoint. Select the line of code where we want the program to stop, to inspect variables from it, or perform a step-by-step execution, to verify the correctness of the code.

During the testing of a program, it can be interesting to verify certain parts of the code. We are not interested in testing the entire program, since we have defined the specific point where to inspect. To do this, we use the breakpoints.

Breakpoints are markers that can be set on any line of executable code (a comment, or a blank line, would not be valid). Once the breakpoint has been inserted, and debugging has started, the program to be evaluated would be executed up to the line marked with the breakpoint. At that time, you can perform different tasks, on the one hand, you can examine the variables, and check that the values assigned to them are correct, or you can start a step-by-step debugging, and check the path taken by the program from the breaking point. Once the verification is done, we can abort the program, or continue its normal execution.

Within an application, multiple breakpoints can be inserted, and they can be removed just as easily as they are inserted.

4.2. Execution types.

In order to debug a program, we can run the program in different ways, so that depending on the problem we want to solve, one method or another is easier for us. We find the following types of execution: step by step by instruction, step by step by procedure, execution until an instruction, execution of a program until the end of the program,

- Sometimes it is necessary to run a program line by line, to find and correct logical errors. The **step by step** through a part of the program can help us to verify that the code of a method is executed correctly.
- The **step by step by procedures**, allows us to introduce the parameters that we want to a method or function of our program, but instead of executing instruction by instruction that method, it returns its result. It is useful, when we have verified that a procedure works correctly, and we are not interested in debugging it again, we are only interested in the value it returns.
- In the **execution up to an instruction**, the debugger executes the program, and stops at the instruction where the cursor is located, from that point on, we can debug step by step or by procedure.
- In the **execution of a program until the end of the program**, we execute the instructions of a program until the end, without stopping at the intermediate instructions.

The different execution modes will be adjusted to the debugging needs that we have at all times. If we have tested a method, and we know that it works correctly, it is not necessary to perform a step-by-step execution on it.
In the NetBeans IDE, within the debugging menu, we can select the specified execution modes, and some more. The objective is to be able to examine all the parts that are considered necessary, quickly, easily and as clearly as possible.

4.3. Variable browsers.

During the software implementation and testing process, one of the most common ways to verify that the application is working properly is to check that the variables are taking the appropriate values at all times.

Variable browsers are one of the most important elements in the debugging process of a program. Once the debugging process has started, usually with step-by-step execution, the program advances instruction by instruction. At the same time, the different variables take on different values. With the variable browsers, we can check the different values that the variables acquire, as well as their type. This tool is very useful for error detection.

In the case of the NetBeans development environment, we find a panel called Inspector Window. In the inspection window, we can add all those variables that we are interested in inspecting their value. As the program runs, NetBeans will show the values that the variables take in the inspection window.

As we can see, in a step-by-step execution, the program reaches a function named power. This function has three variables defined. Throughout the execution of the loop, we see how the result variable changes its value. If with input values for which we know the result, the function does not return the expected value, "Examining the variables" we can find the wrong statement.

Debuggers are programs that allow an exhaustive step-by-step analysis of what happens within the code of a program. Thanks to debuggers, it is easy to test applications to find possible errors, analyzing their causes and possible solutions.

5. Validations.

In the validation process, the customer is decisively involved. It must be taken into account that we are developing an application for third parties, and that they are the ones who decide if the application meets the requirements established in the analysis.

In the validation they try to discover errors, but from the point of view of the requirements (behavior and use cases that the software being designed is expected to fulfill). Software validation is achieved through a series of black box tests that demonstrate conformance to requirements. A test plan outlines the kinds of tests to be carried out, and a test procedure defines the specific test cases in an attempt to discover errors according to the requirements. Both the plan and the procedure shall be designed to ensure that all functional requirements are met, that all performance requirements are met, that the documentation is correct and intelligible, and that other requirements are met,

When proceeding with each validation test case, one of the following two conditions may exist:

- The operating or performance characteristics are in accordance with the specifications and are acceptable or
- A deviation from the specifications is discovered and a list of deficiencies is created. Deviations or errors discovered in this phase of the project can rarely be corrected before planned completion.

6. Code testing.

The test consists of executing a program with the aim of finding errors. The program or part of it will be executed under previously specified conditions, so that once the results have been observed, they will be recorded and evaluated.

To carry out the tests, it is necessary to define a series of test cases, which are going to be a set of inputs, execution conditions and expected results, developed for a particular objective.

For test case design, three main approaches are typically used:

- Structural or white box approach. This approach focuses on the internal structure of the program, analyzing the execution paths. Within our testing process, we apply it with the coating.
- Functional or black box approach. This approach focuses on functions, inputs and outputs. Limit values and equivalence classes apply.
- Random approach, which consists of using models that represent the possible inputs to the program, to create the test cases from them. In this test, we try to simulate the usual input that the program is going to receive, for this, input data is created in the sequence and with the frequency in which it could appear. Automatic test case generators are used for this.

6.1. Coverage.

With this type of test, what is intended is to verify that all the functions, sentences, decisions, and conditions are going to be executed.

For instance:

```
int prueba (int x, int y){
    int z=0;
    if ((x>0) && (y>0)){
        z=x;
    }
    return z;
}
```

Considering that this function is part of a larger program, the following is considered:

- If during the execution of the program, the function is called, at least once, the coverage of the function is satisfied.
- The coverage of sentences for this function will be satisfied if it is invoked, for example as test (1,1), since in this case, each line of the function is executed, including z = x;
- If we invoke the function with test (1,1) and test (0,1), the decision coverage will be satisfied. In the first case, the if condition will be true, z = x will be executed, but in the second case, it will not.
- The condition coverage can be satisfied if we test with proof (1,1), proof (1,0) and proof (0,0). In the first two cases (x <0) it evaluates to true while in the third, it evaluates to false. At the same time, the first case makes (y> 0) true, while the third makes it false.

There are another series of criteria to check coverage.

- Linear sequence of code and jump.
- JJ-Path Coverage.
- Entry and exit coverage.

There are commercial tools and also free software, which allow the testing of covering, among them, for Java, we find Clover.

6.2. Limit values.

In the attached Java code, there are two functions that take the parameter x. In function1, the parameter is of type real and in function2, the parameter is of type integer.

```
public double funcion1 (double x){
    if (x>5)
        return x;
    else
        return -1;
}

public int funcion2 (int x){
    if (x>5)
        return x;
    else
        return -1;
}
```

As can be seen, the code of the two functions is the same, however, the test cases with limit values will be different.

Experience has shown that the test cases with the highest probability of success are those that work with limit values.

This technique is usually used as a complement to equivalent partitions, but it differs, in that not a set of values are usually selected, but a few, at the limit of the range of values accepted by the component to be tested.

When you have to select a value to perform a test, you choose those that are located just at the limit of the allowed values.

For example, suppose we want to test the result of executing a function, which receives a parameter x:

- If the input parameter x has to be stricter than 5, and the value is real, the limit values can be 4.99 and 5.01.
- If the input parameter x is between -4 and +4, assuming they are integer values, the limit values will be -5, -4, -3.3, 4, and 5.

6.3. Equivalence classes.

The equivalence classes are a type of functional test, where each test case aims to cover as many entries as possible.

The domain of input values is divided into a finite number of equivalence classes. As the input is divided into a set of equivalence classes, the test of a representative value of each class allows us to suppose that the result obtained with it will be the same as with any other value in the class.

Each equivalence class must meet:

- If an input parameter must be within a certain range, there are three equivalence classes: below, in and above.
- If an entry requires a value between those of a set, two equivalence classes appear: in the set or outside of it.
- If an input is Boolean, there are two classes: yes or no.
- The same criteria apply to the expected outputs: you have to try to generate results in each and every one of the classes.

In this example,

```
public double funcion1 (double x){
    if (x>0 && x<100)
        return x+2;
    else
        return x-2;
}
```

the equivalence classes would be:

1. Below: x <= 0
2. At: x> 0 and x <100
3. Above: x> = 100

and the respective test cases could be:

1. Below: x = 0
2. In: x = 50
3. Above: x = 100

7. Quality standards.

The standards that have been used in the software testing phase are:

- BSI standards
  - BS 7925-1, Software Testing. Part 1. Vocabulary.
  - BS 7925-2, Software Testing. Part 2. Testing of software components.
- IEEE Software Testing Standards:
  - IEEE Standard 829, Software Test Documentation.
  - IEEE Standard 1008, Unit Testing
  - Other standards ISO / IEC 12207, 15289
- Other sectoral standards

However, these standards do not cover certain facets of the testing phase, such as the organization, process and management of the tests, they present few functional and non-functional tests, etc. Faced with this problem, the industry has developed the ISO / IEC 29119 standard.

The ISO / IEC 29119 standard for software testing, intends to unify in a single standard, all the standards, in such a way that it provides vocabulary, processes, documentation and techniques to cover the entire software life cycle. From test organization strategies and test policies, project testing to test case analysis, design, execution and reporting. With this standard, any test can be performed for any software development or maintenance project.

ISO / IEC 29119.
The ISO / IEC 29119 standard is made up of the following parts:

- Part 1. Concepts and vocabulary:
  - Introduction to the test.
  - Risk-based tests.
  - Test phases (unit, integration, system, validation) and test types (static, dynamic, non-functional, ...).
  - Testing in different software life cycles.
  - Roles and responsibilities in the test.
  - Metrics and measures.
- Part 2. Testing processes:
  - Organization policy.
  - Management of the test project.
  - Static test processes.
  - Dynamic test processes.
- Part 3. Documentation
  - Content.
  - Template.
- Part 4. Testing techniques:
  - Description and examples.
  - Static: reviews, inspections, etc.
  - Dynamics: Black box, white box, Non-functional test techniques (Security, performance, usability, etc).

To know more
In the following link you can visit the international page, where the standards to be followed for software testing are detailed: Standards for software testing http://softwaretestingstandard.org/

8. Unit tests.

Unit testing, or unit testing, is intended to test the correct operation of a code module. The aim is that each module works correctly separately.

Later, with the integration test, the correct operation of the system can be ensured. A unit is the smallest part of the application that can be tested. In procedural programming, a unit can be a function or procedure. In object-oriented programming, a unit is usually a method.

With unit tests, all non-trivial functions or methods must be tested so that each test case is independent of the rest.

When designing unit test cases, the following requirements must be taken into account:

- **Automatable**: no manual intervention should be required.
- **Complete**: must cover the greatest amount of code.
- **Repeatable or Reusable**: you should not create tests that can only be executed once.
- **Independent**: the execution of one test should not affect the execution of another.
- **Professionals**: the tests should be considered the same as the code, with the same professionalism, documentation, etc.

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. Individual tests provide us with five basic benefits:

1. **They encourage change**: Unit tests make it easier for the programmer to change the code to improve its structure, since they allow to test the changes and thus make sure that the new changes have not introduced errors.
2. **Simplify integration**: Since they allow to reach the integration phase with a high degree of security that the code is working correctly.
3. **Document the code**: The tests themselves are documentation of the code since there you can see how to use it.
4. **Separation of the interface and the implementation**: Since the only interaction between the test cases and the units under test are the interfaces of the latter, you can change either one without affecting the other.
5. **Errors are more limited and easier to locate**: since we have unit tests that can unmask them.

8.1. Tools for Java.

Among the tools that we can find in the market, to be able to carry out the tests, the most outstanding would be:

- **Jtiger**:
    - Unit testing framework for Java (1.5).
    - It is open source.
    - Ability to export reports in HTML, XML or plain text.
    - Junit test cases can be run through a plugin.
    - It has a complete variety of assertions such as the verification of compliance with the contract in a method.
    - The metadata (Data sets that are used to describe other data) of the test cases are specified as Java language annotations
    - Includes an Ant task to automate testing.
    - Very complete documentation in JavaDoc, and a web page with all the necessary information to understand its use, and to use it with IDE such as Eclipse.

    - The Framework (Conceptual and technological support structure defined, usually with specific software modules, based on which another software project can be organized and developed) includes unit tests on itself.
- **TestNG**
    - It is inspired by JUnit and NUnit.
    - It is designed to cover all kinds of tests, not only unit tests, but also functional ones, integration tests ...
    - Uses Java 1.5 annotations (long before Junit).
    - It supports Junit tests.
    - Support for passing parameters to test methods.
    - Allows the distribution of tests in slave machines.
    - Supported by a wide variety of plug-ins (Eclipse, NetBeans, IDEA ...)
    - Test classes do not need to implement any interface or extend any other class.
    - Once the tests are compiled, they can be invoked from the command line with an Ant task or with an XML file.
    - Test methods are organized into groups (a method can belong to one or more groups).
- **Junit**
    - Unit testing framework created by Erich Gamma and Kent Beck.
    - It is an open source tool.
    - A multitude of documentation and examples on the web.
    - It has become the de facto standard for unit testing in Java.
    - Supported by most IDEs like eclipse or Netbeans.
    - It is an implementation of the xUnit architecture for unit test frameworks.
    - It has a much larger community than the rest of the Java testing frameworks.
    - Supports multiple types of assertions.
    - From version 4 it uses the annotations of the JDK 1.5 of Java.
    - Possibility of creating reports in HTML.
    - Organization of tests in test suites.
    - It is the most widely used testing tool for the Java language.
    - The development environments for Java, NetBeans and Eclipse, incorporate a plugin for Junit.

8.2. Tools for other languages.

Currently, we find a wide set of tools aimed at automating the test, for most of the most widely used programming languages today. There are tools for C ++, for PHP, FoxPro, etc.

The following tools should be highlighted:

- **CppUnit**:
    - Unit testing framework for the C ++ language.
    - It is a free tool.
    - There are various graphical environments for the execution of tests such as QtTestRunner.
    - It is possible to integrate it with multiple development environments like Eclipse.
    - Based on the xUnit design.
- **Nunit**:
    - Unit testing framework for the .NET platform.
    - It is an open source tool.
    - It is also based on xUnit.
    - It has various expansions such as Nunit.Forms or Nunit.ASP. Junit

- **SimpleTest**: Testing environment for applications made in PHP.
- **PHPUnit**: framework to perform unit tests in PHP.
- **FoxUnit**: OpenSource unit testing framework for Microsoft Visual FoxPro
- **MOQ**: Framework for the dynamic creation of mock objects (mocks).

9. Documentation of the test.

As in other stages and tasks of application development, the documentation of the tests is an essential requirement for their correct performance. Well-documented tests can also serve as a knowledge base for future testing tasks.

Current methodologies, such as Métrica v.3 (Information Systems Planning, Development and Maintenance Methodology. Metrica v3 can be used freely, with the sole restriction of citing the source of its intellectual property, which is the Ministry of the Presidency), propose that the documentation of the testing phase be based on the ANSI / IEEE standards on software verification and validation.

The purpose of the ANSI/IEEE standards is to describe a set of documents for software testing. A standard test document can facilitate communication between developers by providing a common frame of reference. Defining a standard test document can be used to verify that the entire software testing process has been completed.

The documents to be generated are:

- **Test Plan**: At the beginning a general planning will be developed, which will be reflected in the "Test Plan". The test plan begins the System Analysis process.
- **Specification of the test design**. From the expansion and detail of the test plan, the document "Test design specification" emerges.
- **Specification of a test case**. Test cases are specified from the test design specification.
- **Specification of test procedure**. Once a test case has been specified, it will be necessary to detail the way in which each of the test cases will be executed, being included in the document "Specification of the test procedure".
- **Record of tests**. Events that take place during the tests will be recorded in the "Test log".
- **Test incident report**. For each incident, detected defect, request for improvement, etc., a "test incident report" will be prepared.
- **Summary test report**. Finally a "Test Summary Report" will summarize the test activities linked to one or more test design specifications.

> **To know more**
> In the following link you can visit the page of the Ministry of Territorial Policy and Public Administration, dedicated to [Metrica v.3] (https://administracionelectronica.gob.es/pae_Home/pae_Documentacion/pae_Metodolog/pae_Metrica_v3.html)

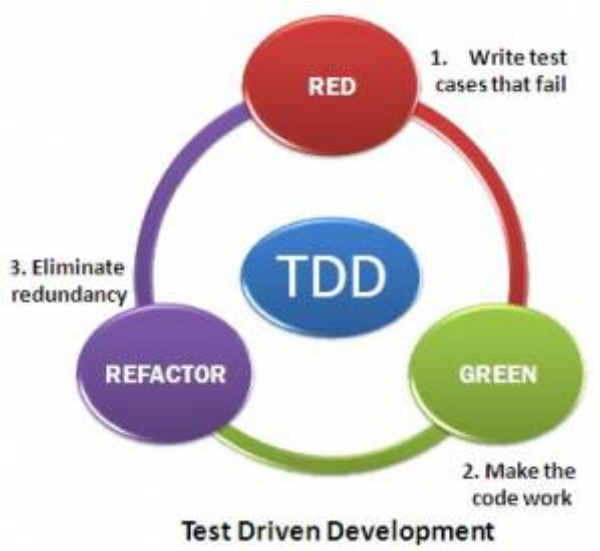
10. Test Driven Development

Test Driven Development (**TDD**) is a software design and implementation technique that falls within the XP (eXtreme Programming) methodology. **TDD** focuses on three fundamental pillars:

- The implementation of the just functions that the client needs, **and no more**. You should avoid developing functionalities that will never be used.
- Minimization of the number of defects that reach the software in the production phase.
- The production of modular software, highly reusable and ready for change.

- Test Driven Development raises tests before implementing code, so code should focus solely and exclusively on passing tests. Based on some requirements for a project, the test cases are designed. The requirements are functionalities that my program must carry out, so it is possible in advance to design the tests that verify said functionality and later the code that will pass each test.

10.1. The TDD algorithm



The essence of TDD is simple, but putting it into practice correctly is a matter of training, like so many other things. The TDD algorithm has only three steps:

1. Write the test for the requirement (the example, the test).
2. Implement the code according to that example.
3. Refactor to eliminate duplication and make improvements.

10.2. Write specification first

Once we are clear about the requirement or functionality, we express it in the form of code. How do we write a test for a code that doesn't exist yet? Isn't it possible to write a spec before implementing it? A test is not initially a test but an example or specification.

In order to write the tests, we first have to think about how we want the program's API to be, that is, what methods we want the program to have and how they will work. But only a small part, a well-defined program behavior and only one (method). We have to make the effort to imagine what the code of the program would be like if it were already implemented and how we would verify that, indeed, it does what we ask it to do.

We do not have to design all the specifications before implementing each one, but we go one by one following the three steps of the TDD algorithm. Having to use a feature before writing it gives the resulting code a 180 degree turn. We are not going to start by annoying ourselves but we will take care to design what is most comfortable, clearest, as long as it meets the objective requirement.

10.3. Implement the code that makes the test work

Having the example written, we code the minimum necessary for it to be fulfilled, for the test to pass. Typically, the minimum code is the one with the fewest number of characters because minimum means the one that took us the least time to write it. It doesn't matter if the code looks ugly or sloppy, we will amend that in the next step and in subsequent iterations.

11. Information sources

- Wikipedia
- Code&Coke (Fernando Valdeón)
- Apuntes IES El Grao (Mª Isabel Barquilla?)
- Apuntes IOC (Marcel García)
- Apuntes José Luis Comesaña
- Apuntes IES Luis Vélez de Guevara 17-18 (José Antonio Muñoz Jiménez)