**Name:** Diego Páez

**NRC:** 2963

**Chapter 8: Boundaries**

The chapter does not talk about users wanting an interface that focuses on their particular needs and third-party package and framework providers strive for broad applicability so that they can work in many environments and attract a wide audience.

Referencing is that learning tests are not only free, they also have a positive return on investment. When there are new versions of the third-party package, we run the learning tests to see if there are behavioral differences.

These learning tests verify that the third-party packages we are using work the way we expect. Once integrated, there is no guarantee that the third-party code will continue to be compatible with our needs. The original authors will have pressures to change their code to meet their own needs.

Interesting things happen at the limits. Change is one of those things. Good software designs adapt to change without major investments and modifications. When we use a code that is beyond our control, special care must be taken to protect our investment and ensure that future changes are not too expensive.

**Chapter 9:** **Unit Tests**

The chapter refers to TDD, this asks us to write unit tests first, before writing the production code.

Nor does it say that the worst is having no dirty evidence, but the worst is having no evidence. Sometimes the problem is that the tests should change as the production code evolves. The dirtier the tests, the harder it will be to change them. The more tangled the test code is, the more likely it is that you will spend more time entering new tests in the set of those needed to write the new production code. As you modify the production code, old tests begin to fail, and the clutter in the test code makes it difficult for those tests to pass again. Therefore, tests are considered a growing responsibility.

If you do not keep your tests clean, you will lose them. And without them, you lose everything that keeps your production code flexible. It is the unit tests that keep our code flexible, maintainable and reusable. The reason is simple. If you have proof, don't be afraid to make changes to the code! Without evidence, each change is a possible error.

F.I.R.S.T

Clean tests follow ﬁve other rules that form the above acronym:

Fast Tests should be fast. They should run quickly.

Independent Tests should not depend on each other.

Repeatable Tests should be repeatable in any environment.

Self-Validating The tests should have a boolean output

Timely The tests need to be written in a timely fashion.

**Chapter 10: Classes**

The chapter deals with the class which tells us that a class must start with a list of variables. Public static constants, if any, must appear first. Then static private variables, followed by private instance variables. Public functions must follow the list of variables. If the rules for class creation are followed, this helps the program read like a newspaper article.

 Sometimes we need to protect a variable or utility function so that it can be accessed through a test. For us, tests are the rule. If a test in the same package needs to call a function or access a variable, we will make it protected or within reach of the package. However, we will first look for a way to maintain privacy.

There are rules: the first rule of classes is that they must be small. The second rule of classes is that they should be smaller than that. But as with the functions, smaller is the main rule

The Single Responsibility Principle (SRP) states that a class or module must have one, and only one, reason to change. This principle gives us a definition of responsibility and a guide for class size. Classes must have a responsibility, a reason to change. For most systems, change is continuous. Each change exposes us to the risk that the rest of the system will no longer function as intended. In a clean system, we organize our classes to reduce the risk of change.

**Chapter 11: Systems**

Something important in the chapter is that it makes us a reference to the construction of use is simply to move all aspects of the construction to main, or modules called by main, and design the rest of the system assuming that all objects have been constructed and connected properly.

It is important to know that the main function creates the necessary objects for the system, then passes them to the application, which simply uses them. This means that the application has no knowledge of the main process or the construction process. Just wait for everything to be built correctly.

A powerful mechanism to separate construction from use is Dependency Injection (DI), the application of Control Investment (IoC) to dependency management. Control Investment moves the secondary responsibilities of an object to other objects that are dedicated on purpose, supporting like this.

He also tells us about The Principle of Unique Responsibility. In the context of dependency management, an object should not assume responsibility for creating instances of dependencies itself. Instead, this responsibility should be passed to another "authoritarian" mechanism, thus reversing control.

The systems must also be clean. When the domain logic darkens, quality is affected because errors are easier to hide and stories become more difficult to implement.

**Chapter 12: Emergence**

A system may have a perfect design, but if there is no simple way to verify that the system really works as intended, then the entire design is questionable.

Keep in mind that a system that is thoroughly tested and passes all its tests all the time is a testable system. Systems that are not testable are not verifiable. We could say that a system that cannot be verified should never be implemented. Fortunately, making our systems testable pushes us towards a design where our classes are small and for a single purpose. Therefore, making sure that our system is fully verifiable helps us create better designs.

Once the code passes the tests, we have a free step to keep our code and classes clean. We have to keep in mind that for every few lines of code we add, we have to pause and reflect on the new design. if we implement new lines of code we have to think if we degrade the code. If so, we clean it and run our tests to show that we have not broken anything.

Duplication is the main enemy of a well-designed system. It represents additional work, additional risk and unnecessary additional complexity. Duplication manifests itself in many ways. Lines of code that look exactly alike are, of course, duplications. Lines of code that are similar can often be massaged so they look even more similar so they can be easier.