**Name:** Diego Páez

**NRC:** 2963

**Summary Clean Code**

**Chapter 1: Clean Code**

This chapter it shows us the problems it causes if the code is not cleaned and what is done with a code that is not clean.

It is important that we will never get rid of the code, because the code has the details of the requirements, which are of importance if a redesign is made.It must also be taken into account that when making a bad code the users and the company are affected, in Users code presents errors and in the company economic losses.

Every time a change is made to a bad code, a disorder occurs, if the disorder increases, the productivity of the programming equipment decreases to zero, which causes the staff to increase in order to improve the code, but the new staff It should start from the beginning as they have no idea what the program has to do or how the code is structured.

If the code is not fixed, the team rebels asking for a change in the design, if this occurs, 2 teams of programmers are created. The one team starts from scratch while the other team must continue maintenance until the new team develops the new code with the same functions as the bad code.

Keep in mind that you always have to clean the catch every so often, since with the passage of it the code gets corrupted. If the code is cleaned every so often, you will not need a massive cleanup. Therefore, the only way to meet the deadline is to keep the code as clean as possible at all times.

**Chapter 2: Meaningful Names**

The chapter does not teach the importance of choosing names for the variable, function or class, it takes time because it must reveal the intention of the names, but it saves work.

It is mentioned that the name of a variable, function or class must answer all the big questions. That should tell you why it exists, what it does and how it is used. If a name requires a comment, then the name does not reveal its intention.

Programmers should avoid leaving false clues that obscure the meaning of the code. We should avoid words whose entrenched meanings differ from our intended meaning.

A good programmer does not add prefixes to the names of the variables since they have a necessary size to avoid doing so. Also, the name of a class must not be a verb, instead the methods must have verb names.

**Chapter 3: Functions**

The chapter talks about 2 rules which are: the first rule is that they should be small. The second rule is that they should be smaller than that. If the functions are too large it is complicated to understand but if they are small they will be easier to read and understand. Something important to know is that functions should do one thing. They should do well. They should just do it.

In order to make sure our functions are doing “one thing,” we need to make sure that the statements within our function are all at the same level of abstraction. It is advisable to read the code from top to bottom although it is very difficult for programmers to learn to follow this rule and write functions that remain at a single level of abstraction. But learning this trick is also very important. It is the key to keeping functions short and making sure they do "one thing".

It’s also hard to make a switch statement that does one thing. By their nature, switch statements always do N things. Of course, every circumstance is unique, and there are times when I violate one or more parts of that rule. It is important to place a name that says what the function does, no matter if it takes time to place it, because then it will be easy to understand the function for the other programmers. The ideal number of arguments for a function is zero (niládico). The more arguments the function has, the more difficult it is to make tests as they increase the number of cases.

**Chapter 4: Comments**

Clear and expressive code with few comments is far superior to messy and complex code with many comments. Instead of spending time writing the comments that explain the disorder that has been made, it is better to spend time cleaning up that disorder. In many cases, it is simply to create a function that says the same as the comment you want to write. It is important the copyright and author declarations are necessary and reasonable elements to put in a comment at the beginning of each source file.

Sometimes, with all the best intentions, a programmer makes a statement in his comments that isn’t precise enough to be accurate. Sometimes it is useful to warn other programmers about certain consequences.

TODOs are jobs that the programmer thinks should be done, but for some reason can’t do at the moment. It might be a reminder to delete a deprecated feature or a plea for someone else to look at a problem. It might be a request for someone else to think of a better name or a reminder to make a change that is dependent on a planned event. Whatever else a TODO might be, it is not an excuse to leave bad code in the system.

Compulsory comments simply cram the code, spread lies and lend general confusion and disorganization. The noise comments these comments are so loud that we learn to ignore them. As we read the code, our eyes simply omit them. Finally, the comments begin to lie as the surrounding code changes. In commented code the programmers will think that it is there for a reason and that it is too important to eliminate it.

The connection between a comment and the code it describes must be obvious. The idea of writing a comment is that the reader can see the comment and the code and understand what the comment is talking about.