



**ASIGNATURA:**

PROGRAMACIÓN ORIENTADA A OBJETOS

**Profesor:**

ING. EDISON LASCANO

**ALUMNO:**DENNIS PARRA

**NRC:** 2963

2019 - 2020

Summary

**Chapter 1: Clean Code**

**In chapter 1** we talk about the correct code, since it happens to us that we are glad to see that our program with many errors worked and we left for later, and maybe edit it, although we knew we wouldn’t do it; thus complying with the law of LeBlanc, which dates: “After is equal to never”. Making a clean code becomes a necessity and is “a form of professional survival”. The line of ethics and professionalism must always be followed, and we must defend our code until the end. We must avoid falling into a disaster in the code, whether due to troubles, setbacks or demands, since this same disaster will slow down our work. “The clean code shows great attention to detail”, the code design should only carry what is necessary, it must be specific and concrete, it should be a pleasure to read it. The Boy Scout rule tells us: “Leave the camp cleaner than what has been found”, in our case, the code.

The practice makes the teacher, and we will only have a clean code practicing.

**Chapter 2: Meaningful names**

**In chapter 2** we talk about the right choice of what we are going to create, "choosing a right name takes time, but it also saves work." The name of any implement that we are going to use will have to indicate why it exists, how it is used and what it does. We must be very careful with spelling errors and it is preferable to differentiate the variables by different names, not by one uppercase and the other lowercase variable. You should not redraw words, creating pronounceable names is the right thing, because if not, its use cannot be explained. ”The name of a class should not be a verb”. Using simple and easy-to-remember or identify names will be best for designating a name, and also using technical names that are understood by programmers who will see or work with the code we are creating.

The fundamental goal of us as programmers is to create the code easy to understand, and it becomes a skill to be able to correctly designate the names, but following the recommendations will be achieved.

**Chapter 3: Functions**

**in chapter 3** we talk about functions should be as small as the action they take. It is mentioned that a good rule of thumb is to never have more than one line of code per block, and one or two indentations per function. We must take into account what Robert C. Martin tells us: "Functions should do one thing. They should do it well. They should do it alone.". The main thing to keep in mind is that the functions should be as small as possible, you can talk about something compact, to do what the programmer needs, that is tidy and that is short. Object oriented programming relies on abstraction as a way to organize code by specific levels of detail. Levels of abstraction can be considered, generally, as levels of detail. “Functions should either do something or answer something, but not both. Either it should change the state of an object, or it should return some information about the object”. It is important to keep functions organized in a meaningful way. suggests a top-down approach where the most broad and general concepts are stated at the top, with deeper and deeper detail the farther down the methods go. The name of the function must be a verb since the function is performing an action and make the names consistent. We must try not to fill a function with arguments, as arguments make testing more difficult because if there is more than one argument, more cases would have to be tested for different combinations of possible inputs. Another important concept about clean functions is that they do not have side effects. A function doing one thing will have one outcome for the function.

All rules must be applied to achieve the creation of functions that are small and concise, with appropriate names, and that are well structured.

**Chapter 4: Comments**

**in chapter 4** mention that, as a general rule, comments are bad. In addition to cluttering the code, comments become useless as code changes because often the comments are not changed when the code is changed. It mentions that comments are necessary if they have where they needs to be, but rarely comments are necessary, but they can be useful for things like adding legal headers or footers, or when the intent of a function is not obvious. It might be helpful to comment non-plain English parts of the code such as a regular expressions. They can also serve as a note for something to come back to if something needs attention that could not be met at the time of writing. Usually if you find yourself needing to write a comment, it is because the code is not clear. There could be many reasons that the code is not clear. Sometimes the solution is simply a better denomination. If the names of the functions are clear with their intention, if the action they will perform is indicated in the name of the function, it is not necessary to comment on their operation. Similarly, if the objects are named in a way that describes them, there should be no doubt about the object that describes the name, and thus adding comments would be avoided.

Try to avoid comments and be as clear as possible in the functions and in the code in general.

**Chapter 5: Formatting**

**in chapter 5** we talk about the format that our code should have, and although it is not a consequence of how the code really works, it makes a big difference in how the future user of the code will be able to navigate and understand what is happening. It is said that the broken window theory is applied, and takes its name from the idea that a single neglected broken window in an abandoned building would eventually invite more acts of crime and vandalism. This applies to programming and the creation of our code, because an application with clean and correct format sets us positive standards for future results such as editions, corrections or even a fluid reading of the code. A poorly made format, on the other hand, will make it difficult to edit the code later. The vertical and horizontal format are some considerations to consider when considering how your code and files should be organized.

The vertical format points to the total size of a file with respect to the lines of code per file. As a general rule, you don't want to have files with more than 500 lines of code, but even smaller files are easier to classify. "Smaller files are usually easier to understand than large files."

In the horizontal format, generally, 100 to 120 characters per line are appropriate. "We use horizontal white space to associate things that are strongly related and dissociate things that are more weakly related." Bleeding is another important concept of horizontal format. Larger concepts, such as the class or module declaration, would appear more to the left, with smaller details within the functions and loops that have more indentation. This helps readers know where concepts begin and end.

The main thing in this chapter is to know how to structure a good format so that the edition, reading and even edition after the creation of the code is optimal and fast, so we will save time and the code would be a beautiful code.

**Chapter 6: Objects and Data Structures**

**in chapter 6** this chapter mentions that "objects expose behavior and hide data, while data structures expose data and do not have significant behavior."

The programs will have data structures and objects, and each one will have its purpose within the code. Since the data in the objects is hidden, new objects can be added without changing the available behaviors. And also vice versa, however, adding new functions would mean that classes would have to change to accommodate new behaviors. This type of code structure is what is known as Object Oriented. The opposite approach would be **Procedure**, where new functions can be added without changing existing data structures because the data structures do not have any behavior that is affected.

This speaks to the broader theme of **abstraction**. "Allow users to manipulate the essence of the data, without having to know the implementation." Abstraction allows programmers to present a representation of an object without actually exposing it. This helps keep the variables private.

“Good software programmers understand these problems without prejudice and choose the most appropriate approach for each task,” in this way a code will be made in a correct way.