Software documentation

Introduction: in this chapter we will do a brief reflection about some of the elements involved in software development. We start with a question: have you wondered why the computer science degree was changed from a bachelor to an engineering? Do you know the difference?

The dictionary says that a bachelor is a "person that has acquired a degree in a university faculty that makes him/her able to work in a specific profession". On the other hand, an engineer is a person that applies "scientific knowledge to invention, improvement or use of techniques in the industrial field".

Technology is defined as the science that is about industrial arts, so that it is like the theory of the practical industry. And science is a knowledge set, and the related activity to achieve that knowledge, which is characterized by the intersubjectivity (which means that two or more subjects or consciences take part) and in practise by the capacity to do accurate forecasts about a part of the reality. That is, that science deals with knowledge and technology with its industrial application.

The FIB was the first faculty of the UPC. Why does the UPC have technical schools and faculties? We could say that a bachelor (who’s educated in a faculty) is a scientific and an engineer is a technologist (usually educated at a technical school).

Obviously, and in consequence, there are elements that affect technology and do not or cannot have the same impact in the scientific fact. The most important ones are the economic and profitability facts, plus the planning ones which are highly related, especially in software production, because in this activity the manpower (neurons if you prefer) is essential and its cost is attached to temporal components and therefore, planning.

Possibly, you will wonder what do all of these things have to do with documentation? Well, on one hand we could now speak about planning, but we are again in front of an aspect that is beyond this subject’s scope, although it is necessary that you organize yourselves well in order to get the best performance out of your effort. The course planning is closed and you know when you have to submit each delivery and what must be included in each of them.

However it is necessary to speak about other elements that have a direct incidence over the productivity of the people that work with software, both for the technicians developing it and for the users of the product. I am talking about documentation.

The need of documentation: the documentation is a tool to make things easier, it must be useful for reducing the effort and improving the product’s comprehension, thus increasing the productivity of both the user in front of the program and the technicians during the development and maintenance stages.

The problem is that documenting means an additional effort, technicians are not usually good at it and they generally do not like it (maybe because of the problems that we usually have with free text and which we have already mentioned when talking about the specification); furthermore a big part of the documentation is not up to date anymore very fast, as soon as the first maintenance operations are done,
sometimes even before the project is finished. That is to say, we might state that it increases the cost of the product and we don't get much profit out of it.

In addition, it is easy to reduce the cost and time of development (which is sometimes necessary, mainly due to an inaccurate planning) by giving up documentation, specially the technical one, but many times even the user's.

This is a bad habit because on one hand it can complicate the development itself (this is alleviated by oral transmission within the development team), but also in face of the future maintenance of the program, especially when the technicians in charge of this maintenance will not be the same ones who wrote the software.

The absence or bad quality of the documentation for the user tries to be alleviated by providing additional training about the use and functionalities of the product. This training normally involves the computer technicians who have developed the product or are doing its maintenance; which as you can imagine gives a training of poor quality.

Similarly there is a tendency to manufacture by means of processes that can guarantee the final product's quality, that is to say, without a later control, because the quality is integrated into the construction process. There is a tendency to incorporate the documentation into the final product, when it is devoted to the user, or in the own process of development in the case of technical documentation.

The technical documentation must be integrated into the methodology itself, it is usually part of the contributions of the corresponding CASE tools (Computer-aided software engineering), and if one proceeds correctly and each time a change is done the process is revised from the lifecycle stage where the modified aspect was defined, the technical documentation will always be up to date.

In our case you will have to define, for each cluster, a set of programming standards whose main goal is to define how to apply in a uniform way the auto documentation techniques in the source code. (Each cluster will work as an independent software development organization.) This standard should allow to not recognizing who is the author of a class (or source file), if it weren't because the source text itself says so, never because of the style or look.

A programming standard must facilitate the interchangeability of the technicians along the development and the maintenance of computer applications since it must make the comprehension of the source code easier.

In some organizations the source's format is validated according to the chosen standard and a non-valid source is not allowed to be compiled. It only makes sense to proceed like this when we do not have any suitable CASE tool because we have to use customized programs from and for the organization.

Take into account that these rules are just another element in technical documentation, but not all of it, that is to say which in addition to the sources, the class diagrams done using UML will be also needed to completely describe the program.

When complementary elements to the ones given by the corresponding CASE tool are introduced with the programming standards, it is necessary to be careful when adapting the rules to the used tools; we should avoid the rules making us modify what is automatically generated.
Elements in a standard of programming: we could classify the most relevant aspects that a standard must cover as the following:

- Nomenclature aspects, that is, the construction of the identifiers or element names.
- Comments policy, where the criteria to take into account are defined, the places where you should write them and the mandatory level, that is, which comments are mandatory and which ones are just recommended.
- Program structure, in general questions of look, such as indentation, and distribution of the language control structures.

As you can see, these aspects are highly related to the programming language, specially the questions of program structuring.

Nomenclature of the identifiers: nowadays it is required that they are mnemonic, that is, that the name helps to recognize (or remember) the identified element.

Since many times they are formed by more than one word it will be necessary to define whether the word's separation will be indicated or not, and in the former case we should define how it will be done. Currently there are two main tendencies to mark these separations:

1. Using capital letters, for example, thisIsAnIdentifier
2. Using underscores “_”, for example this_is_an_identifier

An identifier is mainly composed by letters (limited to the 26 letters of the English alphabet), numbers between 0 and 9 and some other characters from the 128 ones of the ASCII code without extension (only a few are used, for example “_”, “$”, …).

It is recommended to just use the difference between capital and lowercase letters when the programming language gives support, if it doesn't this difference will only have an esthetic sense, not real.

Another important aspect is the type. The modern concept of types, that makes as consider, in addition to the values that an object can take, which operations can be done with this object (its methods), forces as to a more extensive typology of objects. That is why there are proposals to include a reminder of the type at the identifiers of the elements of the program that have type, and one of these is the Hungarian notation. Nowadays, these proposals have many detractors because this information is somehow redundant.

The Hungarian notation has not been made in Hungary, in fact it was proposed by an American with Hungarian ancestors (Charles Simonyi who now works at Microsoft, when he worked at Xerox) and as the identifiers seem a bit weird, he was told that the notation reminded of the Hungarian language.

It is about each identifier having a prefix indicating its type. That is, that the identifier of any element having a type starts with a first part indicating its type, for example, indRegisteredPersons could be the subindex of the table of registered persons, provided “ind” were the prefix of type subindex.

It will be necessary to define the criteria to form prefixes. Basically we must take into account the following aspects:
• The prefixes of the basic classes are defined (integer, real, boolean, character, table, list, queue, etc.).

• Normally we work by juxtaposition, for example, if "i" is the prefix for integers and "p" for pointers, "pi" will be the prefix of a pointer to an integer.

• The maximum length of each prefix is usually limited to 3 or 4 characters, whenever you get to a bigger length by juxtaposition, a new one is defined. Therefore, if the maximum length is three and we want a pointer to a table of names, whose prefix is "tnm", instead of using "ptnm" we will have to define a new one, for example "pn" (in case it is not being used already, of course).

There is no need to say that they cannot be repeated, there must be bivivocity: a prefix is only from a class and a class will only have one prefix.

• Each time a new type is created (a non-abstract class), its prefix is defined according to the defined criteria, and this prefix must be part of the information included at the header of the class in the source file.

The comments are the most literary aspect of the documentation in the source code of the program, comments don't have any effect on the executable, although they are the basic for legibility. In order to make the code more legible it is recommended to include only the essential comments.

A textual comment must be short and concise, it must provide additional information and it has to be understandable. For example, "do   // this is a loop" doesn't give any information (because it is obvious). (Remember that direct sentences with a simple and ordered syntax are easier to understand).

It will be necessary to define which will be the mandatory comments, such as the ones in the header of a class (date, brief description, author (the author of a class must be only one person, each class will be written just by one student), history of modifications, etc.), the ones of the attributes (brief description, etc.) or ones of the methods or operations (precondition, postcondition, brief description, summary of the dialogue, etc.).

There is also a need to define when, why and in which circumstances additional comments (either mandatory or not) will be written. For example, some languages as Eiffel let you introduce predicates of the formal specification (precondition, postcondition, invariant, etc.) by means of elements of the language itself, and in other languages this is not possible, so we can introduce them as additional comments. It may also happen when a gruesomeness is used, such as the following example:

```c
int a, b;
...
a = a + b;
b = a - b;
a = a - b;
...
```

This "trick" is only valid for integers, if you analyze it you will see that it swaps the values of a and b without using an auxiliary variable, but it is not obvious so it is necessary to add a clarification in an additional comment. I cannot leave unsaid that nowadays it doesn't make much sense to do this to save an integer variable.
A good organization of the comments allows that we can think about processing them in face of documenting the program and the class libraries. Some languages have different options of comments for this purpose or even additional possibilities such as the *indexing* clause in Eiffel, as well as the additional tools of documentation as *javadoc*.

**Programs structure**: as we want all the programs to have the same look, we must define all the characteristics that configure this look. That is, we have to define the relative position of each of the elements in a source file, which are mainly the following:

- Position and format of the comments, both the mandatory as well as the optional ones, and both at the header of the class and in each of its attributes and methods.
- Distribution of the elements of the language, specially the control structures, both alternative (as the *if*) and iterative ones. For example, in the alternative structures *if* we could define:

  ```
  if ( c ) or also if ( c ) {
    s1
    s1
  }
  else {
    s2
    s2
  } //if
  ```

  Obviously there are other possibilities; these are only a pair of examples.

- A good habit is to make use of the proposal made by Eiffel to mark with a comment the kind of each *end* clause (Eiffel closes all the structures with an *end* clause) and export it to other languages where the place where structures are closed may cause ambiguity. For example in the case of C and Java this happens when we close a bracket "}".
- We must define if the indentation is done by using spaces or tabs. The mix between tabs and spaces usually gives problems, especially when we change the editor and the tabs are not correctly defined anymore.
- Fix the maximum length of a line - including the indentations.
- Limit the number of lines of a method. It is used to limit the complexity of the methods. Sometimes two values are fixed, the recommended maximum and the limit. For example, we could say: it is recommended that a method does not exceed 25 lines, but a method will never have more than 40 lines.
- Criteria about separations with blank lines, when we leave a blank line or not; or lines of dashes, etc.
- Documentation of the define constants.
- Etc.

It is usually easier to use examples or sketches to define the look rules rather than describing them textually.

**Other elements**: there are other aspects to keep in mind such as:
• Management of versions and revisions of the classes.
• Language elements to recommend, to consider not recommended or to pro-
  hibit.
• Versions of the development products, compiler, CASE tools, etc.
• Others.

**User's documentation:** in this case it is about writing the necessary texts for the user to use the software getting the highest performance. The goal is to coordinate the objectives of the user with the product capacities.

Nowadays, this documentation is usually integrated into the programs as hyper-
text in a help guide (*help* in format *hlp*) or in HTML format. Remember that the text must be clear and easy to follow, so it must be well written, with a simple, well structured, ordered and direct syntax.

Types of documentation for the users:
1. Manual or reference guide: a list and exhaustive description of the program functionalities (all the options one by one: how to use them and what they do).
2. User's guide: a description of the behaviour that the user should have in front of the program in order to achieve his/her objectives. That is to say, which options he/she must choose and in which order, to perform a specific opera-
tion. In other words, to explain what the logical sequence of operation for each of the user tasks is.
3. Learning manual or tutorial: an interactive learning or self-training course.
4. Quick reference cards: nowadays they have fallen into disuse; they are a sum-
mary or extract of the reference manual.
5. Dictionaries and glossaries: description of special vocabulary, mainly computer related, that has been used in the program and manuals.

The user documentation demanded for the 4th submission must cover, at least, points 1 and 2 of this list, that is to say, the reference manual and the user's guide.