Methodology: specification, analysis and design.

A methodology tries to offer a systematic way to proceed, which has become something essential when working in teams. Methodologies are normally based in experience, but they are not panaceas, although they make things easier they do not make better a poor technician. (There are elements that make better a technician that are difficult to acquire, such as mental order and others which are personal and non-transferable, like experience.)

The methodology proposed for this subject is based upon a classic lifecycle and it applies:

- UML (Unified Modeling Language) in use-case diagrams and class diagrams.
- Guidelines for first object oriented projects (Wirfs-Brock, Wilkerson and Wiener).
- 3 layered architecture in the design stage.
- Include in the specification the creation of the domain model, that is, the data structure. As we work with OO the data structure is the link between the first four stages (specification, analysis, design, and implementation) and gives support to seamless transition.

The specification stage is essential to get a good product, because it consists of defining it, of establishing what the program or programs have to do.

The collaboration of the customer or future user of the software is required because he/she is the expert, who perfectly knows how the company works and how the new product will be applied, while the computer technician (IT) may ignore that. Very often the user (expert) and the customer (the one who pays) do not coincide in the same person. The customer has the maximum interest in the success of the project. It is about encouraging the user or expert to collaborate and to be interested in the project's success (sometimes it is only necessary to make him/her think that the success of the project relies on him/her, giving him/her the leading role).

A first difficulty is that nowadays it is not usual to start a project from scratch, the customer usually has computer processes, which are sometimes really good, that provide a satisfactory service and the objective is to substitute them and improve them by adding new functionalities and technologies. But it is very frequent that the objectives are not well-enough defined, for example, it is necessary to improve the "fidelity" of the customers by offering additional services to selling, but these services are not specified.

It is necessary to make a serious criticism of the current system, searching the non covered aspects and the non satisfied necessities. It is also convenient a critical attitude with the proposals being made, do not ever be completely satisfied with the decisions being chosen, always evaluating the pros and cons of each decision.

In addition, it is necessary to estimate the limitations of the new proposed system, the non covered aspects and the ones left as possible extensions.

The differences in language are another factor of difficulty. The terminology of the user and his/her business does not necessarily have to be familiar for the IT personnel, neither has the information technologies terminology (informatics and telecommunications) to be familiar to the user or customer. This can make difficult a
correct communication, which can be mitigated by establishing glossaries. An abuse of professional terminology can result pedant, which can be annoying. You should avoid making the mistake of having a high-handed attitude —by which too many times ITs are characterized— when dealing with customers. This attitude is increased due to the abuse of information technologies terminology.

Another aspect that also increases the difficulties is the essential use of natural language. The user normally has serious difficulties to understand the formalities related to technology and he/she is only able to understand the descriptions made in natural language, oral or written, of the software system being defined. An IT person will write the system specification using the appropriate formalisms, but unfortunately, he/she will also have to describe the system in such a way that it can be understood and accepted by the customer (for everything to be correct, the specification should be approved by the customer), that is to say, the documents for the customer must be written in natural language, avoiding technical formalisms, in any case using the professional terminology of the user, and in writing.

The main errors or vices that a free text in natural language could have, apart from a bad writing (which is not understandable), are the following:

- Noise, unnecessary information
- Overdefinition or overspecification, too much detail (unnecessary details).
- Silence, lack of information.
- Ambiguity, lack of details or false descriptions (use of synonyms).
- Regrets, correct what has been said.
- Advance references, to refer to aspects not yet known by the reader, although they may be explained later in the text.

A technical text is written always in impersonal form and must follow a precise structure, and it is composed at least by three parts: introduction, discussion or justification and conclusions.

The introduction is used to place the reader into the document, that is, to explain what the subject is, "what it is about". It can be structured as a report, collecting the antecedents that motivate it, it is just a historical relation of the facts that lead to the subject of the document. This kind of introduction makes also a lot of sense when it refers to an aspect that is part of a greater or more complex set. An introduction can be limited to a simple declaration of the document's objectives, for example: "this document specifies program <name> which is necessary to <do that>".

The discussion or justification is the main and the most complex part of the document. Normally, if it is big enough it is structured in sections and subsections. It will include the statement of the set of problems, the different alternative solutions that have been studied (in case there are any), and it will give a justification to the reader of the conclusions in the next part. The critical study of the current system must be as well included in this section of the specification; normally it would be the justification of some of the conclusions.

Finally, the conclusions, they refer to everything that has been discussed and justified in the document. The conclusions of the specification document must be a detailed description of the new system: functionalities and other requirements. The
different aspects which could be optional, that is, which may not eventually be done because they are not essential, will also be distinguished here.

Any written document, especially if it has more than 3 pages, must always include an index which allows the reader to know the structure of the document beforehand and lets him/her access a specific section in a faster way.

In order to start writing a document like this it may be advisable to start doing a schema, mainly if the writer does not have much experience in these concerns. This schema will be usually used as a summary or index of the document. Sometimes it is more convenient to start the schema from the end, the conclusions that must be reached, and structure the discussion and justification of each of them from this point. A good schema lets us order the discourse in the sequence considered as more easily understandable.

The specification is still far from the implementation (in spite of now being difficult to you to separate one thing from the other), it must define what the program has to do, and must not think about how it will be done, that is, the implementation details. Therefore, the specific programming language is not taken into account, neither the operating system, file system or databases, although many times that cannot be avoided and keep thinking whether it is adequate or not.

At this stage, specification, software must be seen as a part of an information system and the software must be specified as a component of the complete system. Therefore, the environment in which it will be operative must be taken into consideration: the different users, the physical real locations, the complete procedures, etc.

As ITs are more comfortable in the formal languages field, the program is usually defined by means of formal languages of specification, and after that a textual description is written from that first specification. In fact, a correct construction process is iterative: the IT studies a specification presented in a textual form to the customer, who adds the appropriate observations and as a consequence the IT has to review his/her document and give it again to the customer until a satisfactory and consensus result is reached.

It is very usual at this stage to use prototypes (preliminary version that permits to navigate through the different views of the program (menus, windows, forms, lists, etc.) and gives an idea of its behaviour once it will be done) that give the user an approximate idea of the reality of the final product. Obviously the use of a prototype requires the previous design of the views. Previously, to give the idea about how the program should be it had been done a schema or first draft of the user manual.

To identify the problem being presented to the user, in first place, his/her specifications must be taken into account, they will be the starting point. It is advisable to include a study of the current system (both what it is done manually and the automatic procedures). What the customer explicitly wants will be got from this study. At this point I would like to do a brief reflection based on the different views of the problem:

1. What the customer wants
2. What he/she really needs
3. What the IT understands that must be done.
4. What the final product is

Obviously a difference between the IT view of the product (3) and what the final program actually is (4) would only emphasize the professional incompetence, so, 3 and 4 must always coincide. The problem is to make coincide what is needed (2) with the final product (4). That is to say, that the objective of the specification is to find out what is really needed. It must not be forgotten that is convenient to make happy the customer with what he/she wanted (1) settling it to what is needed (2). A classic joke:

```
what is asked
what is needed
specification
analysis

what is asked
what is needed
specification
analysis
design
programming
testing
what is finally accepted
```

To enhance the information given by the customer, and with the goal of knowing what is needed, it will be advisable to study all the existent documentation that can be obtained, either normative (legislation, standards, etc) or descriptive (bibliography).

Secondly, it is appropriate to consult other experts to consider different alternatives.

Finally, it is necessary to review similar systems existent in the market that at least will be helpful to get some extra ideas. We may also get to the conclusion that the final product is already done, and it doesn’t make sense to redo it, or that some of the existent solutions partly cover the needed functionalities and can be adapted or not to the stated problem. It is also possible that no alternative solution is found, which is very strange nowadays and a reason should be found.

Obviously, these aspects must be taken into account in your projects. Once what has to be done is explained, you should find out other alternatives for the program by searching documentation, other experts and products in the market.

The specification of the program is done with 4 documents, three technical documents and one descriptive text for the customer. One of the formal documents is a use-case diagram, in UML, which describes the system functionalities, another one is a class diagram also in UML, and the third one is a list of non-functional requirements (NFR) which collects all the system requirements that are not included in its func-
tionality, such as the operating system where the program will run or the response times.

The use case diagram collects the system functionalities, that is, everything that the users of the system must be able to ask or make the system do.

When we talk about functionality it must be taken into account that all systems have two types of functionalities: the user functionalities and the system functionalities. The former are the ones that the user needs for his/her work and that are asked for explicitly or implicitly, while the latter are the ones that the system needs in order to satisfy performance aspects not directly related to the functionalities required by the user. These ones sometimes coincide with the NFR. An example of a system functionality can be a functionality to perform security backups specific to an application, another one, the security access management (user access control).

The diagram shows all the functionalities that the system must have and which users will be able to use each of them.

A use case diagram is composed by three types of elements: functionalities, actors and the relationships between them.

A functionality or use case is represented in UML like this:

```
  +----------------+    +----------------+    +----------------+
  | Nombre de la funcionalidad |  con | Nombre de la funcionalidad |
  +----------------+    +----------------+    +----------------+
  |                 |  con |                 |  con |                 |
  |                 |      |                 |      |                 |
  +----------------+      +----------------+      +----------------+
```

There should be one use case for each functionality, although sometimes elements of this type are created for a group of functionalities of the same category or family. For example:

```
  +----------------+    +----------------+    +----------------+
  | Nombre de la funcionalidad |  con | Nombre de la funcionalidad |
  +----------------+    +----------------+    +----------------+
  |                 |  con |                 |  con |                 |
  |                 |      |                 |      |                 |
  +----------------+      +----------------+      +----------------+
```

This one would be the CRUD (create, retrieve, update and delete) of the user.

The use case specification is completed with a dialogue, which explains the behaviour of this functionality. The complete dialogue description is studied in the software engineering subjects, in this subject only a summary of the dialogue will be required. The dialogue consists of explaining the normal behaviour (typical dialogue) of the use case, and as well the exceptional or irregular behaviours (possible errors and
alternative fluxes). The dialogue should provide an idea about how the use case will behave, in this subject not much detail is required, but it is necessary to provide a minimum idea of how everything works. For example, the user registration will be as follows:

- **Behaviour:** once the actor has decided to do a registration, the system asks for the following data: name and surname of the user, username, password – which will not be shown and will be asked twice – and the type of user – which will be chosen from a listbox included in the input form. The system validates the values and coherence of the input data, that is, that no user with that code already exists in the system and that both passwords are the same. In the end, the system stores all the new data.

- **Possible errors and alternative fluxes:**
  1. When a user with the proposed name exists in the system: it will give a warning and will ask for the username again, giving also the possibility to abandon this functionality without registering the new user.
  2. When both input passwords are not the same: it will give a warning and will ask again to introduce the password twice.

If you prefer, it could be summarized further as follows:

- **Behaviour:** when the actor chooses to register, he/she must introduce name and surname, username, password – twice – and type of user – chosen upon a list. The system validates the values and coherence of the input data and stores them.

- **Possible errors and alternative fluxes:**
  1. The user code already exists: change it or abandon.
  2. Both passwords do not coincide: introduce them again.

The use case information is completed with a brief description and the list of actors that can use it. (See the document 'Information about the first submission diagrams and Rational Rose' at [http://www-assig.fib.upc.edu/~prop/InfoRatRose_EN.html](http://www-assig.fib.upc.edu/~prop/InfoRatRose_EN.html).) Those who are already familiar with these techniques must keep in mind that in this subject sequence diagrams will not be used.

Subject’s rules establish that for each use case, you must specify:

- **Name** of the use case
- **Actors** who take part
- **Description** (one or two lines)
- **Typical dialogue** between the actors and the system and changes produced in the system
- **Possible errors and alternative fluxes** of the use case

Sometimes, a use case is defined in function of another. UML defines a few ways to do it, but for this subject’s level just one makes sense, the one that corresponds to when a use case uses another one (include), that is, a functionality calls another one inside its normal flux of execution. The called use case is usually a utility functionality. For example:
The other important element of the use case diagram is the actor, there must be as many actors as different users has the system. Its representation is:

Inheritance can be applied: an actor inherits from another actor when it can do everything the other does and a few more things. For example:

User's typology is described in the NFR, for each type of user there must be an actor in the use case diagram.

Each actor is associated with the functionalities that he/she can execute from the system, which defines the actor's menu:
Sometimes it is necessary to describe some specific functionalities that cannot be accessed by any user, that is, automatic tasks that are triggered by the system in some circumstances. It would be the case for example of an automatic security backup. In these cases an actor named "timer" is created who has the mission of being in charge of these automatic executions.

As we have already talked about the class diagram at the OO session, let's talk about what the NFR are. These are all the requirements of the system that are not reflected in the definition of the functionalities, although some of them may have repercussion in the use case or in the class diagram. Each requirement must be defined so that it can be measured and verified whether the final product satisfies it or not. For example, instead of saying "the system must be easy to learn" it must be specified: "the needed learning time in order to use the program is two days full-time of an administrative employee".

At the subject's website (http://www-assig.fib.upc.edu/~prop/) you will find a document with a list of the most common NFR to include in a project (http://www-assig.fib.upc.edu/~prop/rnf_EN.html). In this list some aspects are missing, like the ones related to the execution environment: operating system, whether it will be executed inside a network, client/server, distributed, etc; if there is a file system management or DBMS previously decided to manage persistence; the importance and need of transportability. It might happen that any other requirements are missing.

For the project you only must take into consideration what is not previously fixed by the subject rules. In fact in your NFR relation there should be just a few requisites, usually below half a dozen, depending on the number of user types defined. You must keep in mind that the specifications written must be satisfied by your program.

Finally, there are some of these NFR which you are not yet prepared to include in the submitted list, because you will have not developed them yet, as the programming standards which must be defined in the second submission documentation.

To do the first OO projects it is good to have some help to find candidates to be functionalities (use cases) and to arrange the data structure.

The starting point is usually a descriptive text, in case it does not exist it can be done by using the interviews or talks with the user (it might even be the detailed definition). The proposal is simple, we start from a reading of the definition, all actions expressed verbally point to the explicit functionalities of the system, and all nominal sentences point to the classes or their attributes.

It will be needed to delete plurals (transform them to singular) and delete synonyms as well. It is necessary to find the subject of those sentences where it might be omitted, specially on passive sentences. The adjectives will usually represent the attributes.

The attributes altogether with the methods are the class responsibilities, the knowledge to be preserved. Nowadays, the attributes are normally accessed through the methods, because the attributes are defined as private.

The attributes store information of the class and they are the object of its operations (methods). A class without responsibilities doesn’t make sense at this stage.
A class with only one attribute usually corresponds to another class' attribute which is not being used correctly. Information redundancies must be avoided, as well as attributes with derived values, such as the age of a person if the birth date is also stored.

The class diagram is completed with the relationships between classes. The ones really important to the specification are the association relationships, due to their influence in the description of the system's behavior.

As to the use case diagram, the afore-mentioned information—as well as the verbal expressions—may lead to functionalities, since this information must be obtained and validated (specially the one that has to be stored and handled). Finally, the own role of every class within the system may lead to responsibilities.

A way to review if the system's defined functionality is complete is to get a list of circumstances in which the user may want to retrieve information from the system. To do this review walkthroughs are used, which consist of repeating step by step the processes the user performs and see whether the specification includes the needed use cases and the data that allow the user to perform them.

For the project it is necessary to do the use case and class diagrams using Rational rose and document the diagrams following the instructions given in the document "Information about the first submission diagrams and Rational Rose" (http://www-assig.fib.upc.edu/~prop/InfoRatRose_EN.html)

For the analysis stage there is not much work left to do, because a few tasks that were traditionally done at this stage have been done in the previous one. In first place, it is necessary to complete the defined classes with the methods assignment. As the system implementation has not started yet, a more strategic than real distribution is done. It is about transforming use cases into methods of the defined classes.

Sometimes it may not seem very clear where to assign a specific use case as a method, you can decide to do a provisional assignment, to divide the use case in different methods or to create an specific class to that particular use case (or for each use case that we don't know where to place).

During this task it will be necessary to define the dependencies, because it is the moment in which the calls to the methods defined in the different classes show up.

Once the class diagram is complete with the methods, all the classes have their responsibilities, it is the moment to study generalizations (inheritance), that is, to create abstract classes. It is about finding out similar classes in the diagram and seeing whether an abstract class can be created with a common part. Also, we may be lucky and find out that there is a clear inheritance relation between two classes already defined, but this is not very frequent.

Finally topics and subsystems are defined, that is, classes are grouped into families in order to structure the class library, to organize them. Analogously you could think about a directories structure suitable to store the source files (in Eiffel, Java—a package—, etc). Normally, the subject's projects are not complex enough to distribute in topics or subsystems.
Design is the first implementation phase of the program. Now what is important is how to implement the program. It is about time to start it.

In order to do the minimum possible work, it will be important to try and reuse as many things as we can.

To reuse existing code, if you cannot find a needed class in a library, you can try and reuse it through inheritance, aggregation (composition) or combining both. It is about finding classes in the libraries which can be abstractions of the needed ones and take benefit from the inheritance mechanism to try and write the minimum code. We can also look for classes that can be aggregated to the one we have to develop. Or a class that may inherit from another one that will be aggregated, etc.

On the other hand it will be necessary to adapt the implementation to the programming language and its characteristics. For instance, Eiffel implements multiple inheritance, but Java does not.

When multiple inheritance is needed in a language that doesn't support it we can use a known mechanism called delegation: it consists of inheriting directly from a class (the most significant one, in general) and for the rest of classes replicating the code inside the source code of the class that inherits. This can be implemented by simply defining in the child class an attribute of the class of the parent by aggregation. To complete it a method must be created for each of the methods in the parent class, that simply calls the other (parent.method_in_the_parent_class(arguments)), like this the effects of inheritance over the methods can be simulated better. In this case we must control when we add or delete methods from the parent class.

To implement genericity when the language does not support it a substitution mechanism is used, as the one named "defined" in the C compiler or simply using an editor. It is about substituting the parameter or parameters for the specific type in each case. In fact, proceeding like this we will end up with a library of specific classes of the generic class being used. The creation parameter can also be transformed into an argument of the creation operations.

Finally, it is necessary to apply the adequate architectonic model. In this subject we will apply a layered model, in particular one with 3 layers: presentation, domain and data management. This model is described in the document: "A 3 layered architecture" (http://www-assig.fib.upc.edu/~prop/A3COO_EN.pdf), we will talk about it later. Mainly, it is about separating the components of interface management and the ones of persistence from the ones comprising the logic base of the program or domain. There are different reasons why this separation is recommended, we will point out the fact that technological evolution usually affects in a different rhythm to the elements of data and interface management from the ones of the program's logic. If there were no layer separation each technological change in any aspect - interface or data management- could directly affect the majority, if not all, the program classes. It is also recommended because nowadays in distributed work there is the idea of distributing the layers in different equipments (client, application server and data server). In fact it is the architecture adopted by Microsoft for its .net platform.

This architectonic model is characterized by the ease to both do changes (which means: extensibility, portability, maintainability and restructurability) and testing, on
the contrary if makes more difficult to get an optimum efficiency because it adds unnecessary or redundant tasks to the system.

The application of **reusing** in your project: we do not have a library (repository) of classes, where we can find the classes to reuse. On one hand, we can choose to use existing libraries (on the internet or somewhere else), but in that case the tutor must give his/her authorization. What is mandatory for the subject is to find classes to share among the cluster. This task is normally complicated and sometimes a reason of conflict, that's why we present the following schema which may help you to get everything to work appropriately. Your behavior schema should be as follows:

1. Each group works separately in its project and applies a 3 layered architecture. (It is necessary to specify well the classes, with the responsibilities well defined, which will be useful for the team work, both for the group and for the cluster).
2. The classes developed by the three groups of the cluster are placed in common in order to search for abstract classes and aggregations (compositions) that can be shared out. I'll tell you some possibilities.
3. A list of classes that can be shared and the groups interested in using them is written (this list is a commitment, that is, these groups must use these classes). It is not necessary that all the cluster use all the classes, a shared class might be used only by two groups.
4. A consensus is reached about the specification of each of the shared classes. The specification must be formalized in order to submit it to the tutor. It is very important that the groups that are going to use a class completely agree on their specification, this task must never be left to the person implementing it.
5. The implementation of the shared classes is distributed as fairly as possible according to the following rules:
   - The group in charge of implementing a class must be interested in using it.
   - The groups receiving the implemented classes must validate them, that is to say, check whether they satisfy their specifications.
   - The distribution must take into account that the development of the tasks must be disjoint, no one can depend on anyone else, even less if they belong to different groups. Development of shared classes must be done totally separately. In case there are any dependencies (two classes that cannot be developed separately) it will only be accepted if only one person is in charge of both ones.

It is recommended:

1. For the search of shared classes, focus mainly on the domain classes. From the presentation and data management layers just share the more abstract and generic classes, if there are any.
2. Meetings of more than 4 or 5 persons are not very productive, so you should try to send just a representative of each group to them.
3. Make an effort to provide a good specification of the shared classes, it avoids problems. (It is also possible to establish conditions of performance or class quality, but be careful because it can add some risks).
Some possibilities to find classes to share:

1. Abstraction of specific or abstract classes:

   - Compartida
     - HeretaGrupA
     - HeretaGrupB
     - HeretaGrupC
   - Classe1GrupA
   - Classe2GrupA

   (Where A, B and C can take values 1, 2 or 3, and knowing that A ≠ B ≠ C)

2. Compositions:

   - ClasseGrupA
   - ClasseGrupB
   - ClasseGrupC

   - Compartida

3. Compositions' abstractions:

   - Compartida
     - ClasseGrupA
     - ClasseGrupB
     - ClasseGrupC
   - AgregadaGrupA
   - AgregadaGrupB
   - AgregadaGrupC
4. **Generic classes:**

```
Compartida
```

```
ConcretaGrupA
ParàmetresA
```

```
ConcretaGrupB
ParàmetresB
```

```
ConcretaGrupC
ParàmetresC
```

5. **Others derived from these basic ones.**