Object Orientation

Its origin is placed at the beginning of the 70's in a friendly environment project that Xerox Company developed in Palo Alto with a team made up of Alan Kay, Adele Goldberg and Dan Ingalls. In fact this project has provided the computing world with elements as interesting as the mouse, the first graphic environment with icons and the Smalltalk language, which is the first OO (object-oriented) language. To develop Smalltalk they benefited from concepts contributed by the Simula67 language (Norway, 1967).

The OO concepts arrived to the market through C++ and the operative systems. Nowadays, every programming language, operative system, DBMS, etc. is OO. In fact, it is not strictly true because, in several cases, what has been done is adding objects too, as an option. For example, one can make OO programs with Turbo Pascal which gives support to objects, but one can make them also without any object.

Introduction to Object Orientation: the basic idea must look familiar to you, since we will start from the notion of type as an encapsulation of values and the operations that may be carried out with them (data and processing), you know this concept as abstract data types (ADT). The differences will be that in OO we will add an inheritance mechanism to ADT, and we will use this concept extensively instead of using it only for data structures.

OO is considered to be supported in three points which are usually symbolized as the three vertexes of a triangle:

**Encapsulació i ocultació**

**Classificació i abstracció**

**Polimorfisme i herència**

Both the encapsulation and the hiding aspects (the latter mean that an ADT is a module or a capsule from which the implementation does not have to be known) and the classification and abstraction aspects (an ADT is an abstraction that represents an element of a specific class or type) must be familiar to you since they are already used for ADTs. That's why we will only comment on the third one.

The idea of polymorphism, although it may seem new to you, is also known by you and, in fact, you are using it when, for example, making additions and subtractions. If
I write \( a := b + c \), could anyone say the type to which \( a \), \( b \) and \( c \) belong? Logically, they can belong to any integer or real type, and in general any numerical type.

It is said that the \( + \) symbol has been overloaded. We could also say that the adding operation has several shapes, that it is polymorphic. This polymorphism of the operations is essential for the inheritance mechanism.

(In the OO terminology, the fields of an ADT are called attributes, the operations are called methods or services and the type is called class.)

We say that a class inherits from another one when it contains all its attributes and all its methods. Therefore, neither defining them again nor making a copy is necessary, the compiler will take care of it. If when defining class \( B \) we declare that it inherits from class \( A \), \( B \) will have all the attributes and the operations from \( A \) with the same name; and that's why polymorphism is necessary. Normally, one or more new attributes or methods will be added to \( B \) in order to complement those inherited from \( A \).

This mechanism allows the definition of non-instantiable classes (ADT), which means they have been designed to inherit from them. We will say these classes are abstract. In some cases, this class is basically abstract because its direct instances do not make any sense, but in some other cases they are abstract since instantiating them is also impossible because a full definition of the class has not been defined. An abstract class may defer the definition of a part of their operations to those that inherit from it; this abstract class is called deferred class. For example, we could define a NUMERIC class, which guarantees, through the inheritance, that every numeric type has addition, subtraction, multiplication and division; but, due to the fact that in some cases it may be advisable to define the operation in a different way, it might be a good idea to assert or to be able to force to those that inherit from NUMERIC to have these operations, without having to define how they are. Defined in this way, NUMERIC will be a deferred class.

Some OO languages allow us to define generic classes, which will require one or more parameters in order to be instantiated. This mechanism will allow us to define generic structures such as the array, the stack or the queue, where the basic element is a parameter. This allows us to define just once the queue class and to instantiate, for example, queues of different types, such as integer, string or person, considering a person as a class that can be a more or less complex structure.

**OO terminology**: we must bear in mind that you may find several terminologies in the OO bibliography, depending on the author or the translator. The most usual one is:

- **Class**: the ATD or encapsulation of data and operations.
- **Object**: specific element of a class (variable, constant, etc.), we usually call instance. OO programmes are made up of objects.
- **Attribute** or **field**: internal datum of an object (a class has the structure of a tuple), plus the definition of the range of values that this datum can take.
- **Method** or **service**: accessible operation (function, procedure or routine) of an object.
**OO philosophy**: It is about a new way of making software. We want to build a computing model of the real system, where each class represents each element of the system, making a model of the reality. For example, the subscript of an array, even though it can be implemented by means of an integer, will belong to the class that represents the numeric subscripts.

With this computer system, a model that simulates the reality is created, from which it extracts the functionality that is of its interest. That is, we don't start from the functionality which is interesting in order to define the system, but from the model of the reality from which only the interesting part is extracted.

Another goal is reusability: the reuse of the work which has already been carried out for further developments. This reuse could arrive to be 80% (this means that 80% of the new program has been extracted from libraries) when the elements have been created with this concept of reflecting the reality and properly joint.

An OO program consists of the joint of predefined components, as we have to do when we set up a PC by fitting boards. You may find in some bibliographies the concept of the software integrated circuit (software-IC).

**Graphical notation and relationships**: we use UML (Unified Modelling Language) for this project and the tool is Rational Rose (or optionally ArgoUML or others). In fact, UML is a quite recent standard for OO. Rational Rose is a CASE tool (computer-aided software engineering) that implements UML, and that is able to generate code in several languages, among which we can find Java. Both UML and Rational Rose are used in the subjects of software engineering.

One class is represented in UML:

```
Nom de la classe
attribut1
attribut2
 ...
attribut n
mètode1()
mètode2()
 ...

NOM DE LA CLASSE
 | attribut1  | attribut2  |
 |           |            |
 | mètode1() | mètode2()  |
 |           |            |
 |           | paràmetres |
```

In case it is abstract, the name of the class will appear in italics. When the class is generic (template class), it will be represented as follows:

```
Nom de la classe
attribut1
attribut2
 ...
attribut n
mètode1()
mètode2()
 ...

NOM DE LA CLASSE
 | attribut1  | attribut2  |
 |           |            |
 | mètode1() | mètode2()  |
 |           |            |
 |           | paràmetres |
```
The classes of a diagram are connected between themselves in several ways. The first relationship we will see is the inheritance: it is said that class $B$ inherits from class $A$ (or also that $B$ belongs to the class $A$ with specific characteristics or that $A$ is a generalization of $B$). For example, a student is a person who studies (or at least it should be like that): student inherits person.

In the class $B$ we only see the attributes and the methods that have been added, but, in addition to the ones from $B$, it has also all the attributes, methods and relationships from $A$, although they are not explicitly displayed. In UML it is represented:

Another static relation among classes is aggregation. When a class $C$ has one or more attributes from another class $D$ we will say that $D$ is aggregated to $C$. It is understood that the object of the class $D$ is a part of the class $C$, which is a whole. This relationship is represented in UML in one of the following two ways:

In a relation of this type a name may be specified (the one of the attribute) or not. An aggregation has always to be quantified, it is indicated herein that for every $D$ there is only one $C$ and that a non-specific number ($n$) of attributes which will be objects of the class $D$ are in $C$. Since the number of objects $D$ that will be contained in an object of the class $C$ is in this case variable, the attribute of $C$ must be a container (an array, a list, etc.) of elements of the type $D$.

It is not necessary to specify these details of implementation, such as the specific container or the name of the attribute, until it will be required. On the other hand, it is advisable to notice that when an aggregation relation occurs, the attribute does
not appear in the box corresponding to class $C$; it would be redundant since it has already been made explicit by means of the relationship.

Notice that these two relationships, aggregation and inheritance, are represented in a vertical line, the inherited class above the one that inherits from it and the added class beneath the class where it has been added.

The graphical representation of the inheritance relationship is usually done from the parent class to all the son classes, in the shape of a comb, whereas it has no sense doing it like that with the aggregation, it will always be from class to class, in such a way that every one of them can be clearly quantified:

![Inheritance Diagram]

In inheritance relationships, it is very normal to refer to related classes as if they were family-related (parents, children, grandparents, grandchildren...).

Another possible relationship between two classes is dependency (or message). It occurs when a class $F$ uses another class $G$ either as a parameter of an operation or because $F$ uses an operation of $G$. It is not necessary to specify the first type of dependency if it is specified as parameter of the header of the operation. $G$ will not know anything about the class $F$ in any case. The representation in UML is:

![Dependency Diagram]

Finally, we will speak about associations (or instance relationships). The relationships defined so far will help you to model classes that represent different levels of importance or abstraction. With the association, relationships between classes which are at the same level (because of their role in the system and because of their abstraction level) are being modelled. It is employed to model structural relationships. In the Software Engineering subjects you will see that these relationships are those that compose the entity-relation or Chen model.

In fact, what they model is the behaviour of the system; the relationships may vary depending on the state of the system. They are relationships existing among certain instances of the elements, for example, in a specific state the lecturer Pau that teaches the subject PROP has Pere as student, but the following year he does not have him anymore. The structures lecturer, subject and student belong to the same level. Only specific values are connected, concrete instances of these classes.
The lecturer Pau is only interested in Pere as a PROP student and in any further detail that has to do with him as student, but he is not interested in any other of his data, their relationship only makes sense in this way.

We will say that an association relationship models the fact that an object of the class \( A \) needs to know about another class \( B \) and vice versa. This relationship is always quantified too. In UML it is represented as follows:

\[
\begin{array}{c}
\text{A} \\
1..n \\
\hline
\text{B} \\
1
\end{array}
\]

In this case every instance of \( A \) is related to an only instance of \( B \); on the other hand, every instance of \( B \) may know one or several instances of \( A \) at the same time. In some cases, this relation is not symmetrical; then we say it can only navigate in one of the two directions. When navigation is limited in UML it is represented in this way, with an arrow pointing it:

\[
\begin{array}{c}
\text{A} \\
1..n \\
\hline
\text{B} \\
1
\end{array}
\]

In this case it means that from the instances of \( A \) we can find the instances of \( B \) that are related to it, but from \( B \) we cannot have access to the related instances of \( A \). In other words, it is only possible to navigate in one direction of the relationship. Navigation is also a detail of implementation to be defined in the appropriate moment, during the design phase.

To finish, it is advisable to notice that both the dependency and the association, and specially the association, are commonly represented by a horizontal line, with the related classes located one beside the other one, so to speak.

There is a small manual on Rational Rose on the page of the subject ES:E that we will use to explain what you need to know about the way in which this program works. (The page is \texttt{http://www.lsi.upc.edu/~es-e/}, there you have to \textit{click} on "Laboratori" and to pick "Documents d’interès", the manual is the first downloading it offers. There is a copy on the subject site \texttt{http://www-assig.fib.upc.edu/~prop/RationalRoseManual.pdf})

In order to fill the associated templates or specifications of a class you must bear in mind that the most important elements of documentation are:

\begin{itemize}
  \item Name
  \item Description
\end{itemize}
• Type: abstract, generic or instantiable. It is instantiable by default: that is to say that there will be objects of this class.
• Visibility (when it is known)
• Persistency

In the documentation of an attribute:
• Name
• Type: class
• Description
• Visibility (when it is known)
• Value by default or initial value (if there is any)
• Validation: accepted values and other determining factors of its value (relationship with other attributes of the class, with attributes of other objects, if it has to be the identifier of the class, etc.)