2

The Purpose of this Work

The main goal of Software Engineering is to obtain quality products that offer high productivity. The way to achieve this goal with adequate software production methods is, nevertheless, always a problem. To date, it has not been solved satisfactorily, even though the multiple solutions that have appeared in the last decade have always presented themselves as being the ultimate solution to the problem.

We could talk about the dozens of third- and fourth-generation programming languages that have been proposed, or about the multitude of software production methods, for a start the structured languages, then the object-oriented ones. We could also discuss the most advanced programming techniques based on the use of components (Component-Based Development, CBD), or the more recent Extreme Programming proposals, the introduction of Agent-Oriented Programming and Aspect-Oriented Programming.

We could talk about Agile Software Development Methods proposals, or about techniques that are linked to Requirements Engineering. There is a true universe of technologies that have repeatedly claimed to have found the Philosopher's Stone for developing an automated process to develop quality software. However, these same technologies have taken us back to the starting point not long after their introduction.

All the evidence shows that the development of software products (commonly referred to as applications) has always been and still is a complex task, especially in management environments (which are linked to Organizational Systems). What is worse is that their complexity, rather than decreasing, continues to increase. This has become a constant factor in Software Engineering and is due mainly to the fact that the customers' needs for products are always more sophisticated. Also, the resulting applications are developed using highly dynamic, constantly changing technologies, which are usually presented with a facade of simplicity.

Complexity also increases because development technologies must structure the final software product according to protocols that are associated to different architectures, which are usually based on object-oriented models and on the development of distributed components and their associated software architectures (DCOM, CORBA, multi-tiered architectures as an evolution of the traditional client/server architectures, internet/intranet environments, etc.)

In most situations, the technological complexity ends with the software engineer devoting more effort to getting to know the technical aspects of a particular solution, rather than focusing on understanding the problem to be solved by the application. This raises the following questions to pursue: modelling vs. programming; understanding and representing the problem vs. implementing the solution; focusing on the modelling (the problem space) and temporarily ignoring the technical aspects (the solution space).

We must bear in mind that only after we have a thorough understanding of what the software product must do, will we be able to define how it should be built. It is absolutely necessary to have production methods that allow us to specify and represent the Conceptual Schema of an Information System and to then transition to the implementation phase, which is responsible for producing a software product that is functionally equivalent to the specification created in the conceptual modelling phase.

Therefore, the question arises as to why this "family of methods" is not common practice in Software Engineering. Perhaps the answer is simply that we do not possess methods and development environments that are advanced enough to allow the construction of software applications from conceptual models, methods that are advanced enough to allow the engineer to focus on *what* the system is, and not on *how* the system will be represented in a given programming environment, methods that are advanced enough to allow the use of notations that are close to the problem space, and not the solution space.

These goals have guided the history of Software Engineering for years. In order to obtain a quality software product in an effective manner, we have witnessed how the level of abstraction in programming languages, tools and environments has evolved, coming ever closer to the user space. This is only natural, if we take into account that any analysis of an information system is characterized by the existence of two fundamental actors:

1. The stakeholders, who know the problem to be automated by a software product. They are the final users of this software product.
2. The software engineers, who must build a coherent model from the knowledge of the stakeholders. The resulting Conceptual Schema must represent (in the Problem Space) the concepts comprehended and captured during the interaction with the stakeholders. Naturally, the next step is to correctly represent this Conceptual Schema by using the specific set of software representations provided by a given programming environment, in what we have denoted the Solution Space.

From the times of the first assembler languages to the modern conceptual modelling environments and the production of software products driven by conceptual models, Software Engineering has tried to provide engineers with production methods that are closer to the Problem Space with the twofold goal of:

1. Easing the specification of Information Systems to make it feasible in terms that are close to the knowledge of the stakeholders.
2. Easing the conversion to the notation of the Solution Space in an organized and precise manner by defining the corresponding transformation mechanisms (the ideal case being the use of model compilation techniques).

As a result of this, one of the critical tasks in modern Software Engineering is Requirements Specification, and the Conceptual Modelling tasks associated with it that lead to building a Conceptual Schema where these requirements are adequately represented. This phase of Conceptual Modelling is strongly related to the final quality of the software product and to the productivity of the corresponding software process because the requirements describe the goals of the effort to develop the software, provide guidelines for the design of the software architecture, and set the basis for measuring the quality of the final product.

In the late 1990s, several studies on software development in Client/Server and Object-Oriented environments (Douglas et al. 1996) supported the thesis that the most critical tasks in the software production process are still the specification and analysis of requirements. They also recognized that errors produced in this phase of the software development process could have a huge impact on the reliability, cost and robustness of the system. These errors are commonly attributed to the lack of tools that offer integral support to the development process and that are closely linked to the subsequent phases of the development process.

An immediate conclusion is that 21st century developers require new development tools that provide them with higher-level constructs, so that they can specify applications using concepts that are closer to the ones used by humans in our cognitive and communicative processes. The "programs" that are built with these conceptual primitives or conceptual patterns of a higher level should then be transformed into the equivalent software product through a translation process that associates each conceptual primitive to its corresponding software representation. Under this hypothesis, the automatic generation of programs from Conceptual Schemas is no longer an unreachable dream but a solid reality.

In accordance with what has so far been discussed, it is useful to analyse the development methods that are currently being used in the software industry. This analysis shows that these methods propose development processes that consider software construction to be the completion of a high number of tasks in several phases. In most of the cases, the resulting process is extremely complex and of little use in day-to-day practice.

Another problem that is commonly observed in current approaches is that some of the effort put into the completion of some of the tasks and in the production of documentation has little or no effect at all on the final product. That is, in the line of what we could refer to as "traditional CASE" methods, much of the effort required to set up the model of a system is often nothing more than an elegant (in the best of cases) documentation of it, but this then has to be manually transformed into a software product by using a notation and concepts that are totally different from those used to build the model.

This "semantic gap" between the model notation and the programming language usually makes the use of a CASE tool a problem because engineers not only have to obtain the software product but they have to model it as well. This explains the problems that have historically prevented the universal use of CASE. In addition to this, when maintenance problems arise and there are changes to the specification of the systems, it is almost impossible to avoid the temptation of performing the modifications directly to the software product, so that the model and its implementation usually are not synchronized.

To avoid this situation, the construction of information systems of a certain size and complexity requires the use of methods to carry out the development process in a rigorous and systematic way. These methods must perform the phases that are strictly needed to obtain a quality software product in a practical and productive manner. Experience shows that in Software Engineering, as in any other discipline where the goal is to obtain a certain quality product, simplicity and effectiveness are the two quality attributes that must be co-guaranteed. Software production methods that are based on an exaggerated and unjustified number of tasks and phases will simply not be used in day-to-day practice and will inevitably be discarded by developers.

The goal of this book is to present a development method that provides a solution to these challenges. This introductory chapter will present some of the problems that application development faces today in an attempt to find a functional and practical solution to these. To do so, we will focus on the most relevant features that any advanced development method or environment should provide for the software development process to be viewed as an iterative process of constructing and refining conceptual schemas.

With such a process, the traditional implementation phase will play a secondary role because the traditional role of the programmer is played by the higher-level role of the modeller. That is, the level of abstraction of the artefact used as the programming language is raised, following an evolution that is similar to the one that took us from assembler languages to third-generation programming languages. This evolution attempted to provide the programmer with languages of which the notation was closer to the problem space and less linked to the solution space.

Within this context, the subsequent sections of this chapter will present three fundamental ideas:

1. A justification of the need for new development environments that provide a solution to the endemic problems that for the last two decades have typically been associated with the term "Software Crisis".
2. The use of a precise ontology of concepts that act as the foundations of our proposal. Specifically, we will analyse why the Object-Oriented approach is the candidate that is best suited to characterize our proposal.
3. The advantages of using patterns at three different levels:
   - Conceptual primitives or conceptual patterns, which are appropriately catalogued and have a formal support.
   - Architectural and design patterns, which are in charge of the definition of the Software Architecture that is best suited for the resulting software product.

– Design patterns, which associate every conceptual pattern with the software representation that is best suited in the solution space.

This book presents a software production method that has been created as a response to all of these problems, the OO-Method. The OO-Method provides an environment for object-oriented conceptual modelling and automatic generation of the software product, which allows engineers to build applications from conceptual models in an automated fashion. This introductory chapter is structured in accordance with the three points stated above. It includes the introduction to the most relevant features of the OO-Method, and also includes the analysis of other approaches that share the goals of OO-Method in order to highlight the main contributions of our approach.