8-16-1996

The Experience Of Joining Two Methodologies In Order To Develop Software

M. Perez
Simon Bolivar University, movalles@usb.ve

The Experience Of Joining Two Methodologies In Order To Develop Software

M. Perez
Processes and Systems Dept.
Simón Bolívar University
e-mail: movalles@usb.ve

1. Introduction

This paper presents the experiences of joining Ivar Jacobson's Object-Oriented Methodology (OOSE) with object-oriented formal specifications in the development of a graphic object-oriented programming environment. As a result of this union, a powerful specification tool was generated to develop a robust, unambiguous system. It is worthwhile mentioning that the area of formal specifications, particularly as refers to object orientation, has not been widely developed, due to its proposal being so recent and novel, mixing the traditional benefits of formal specification languages (consistency, completion and lack of ambiguity in the software developed) with object-oriented characteristics such as encapsulation, inheritance and polymorphism, thus enabling the software specification to be modularized, reusable and coherent. A new trend within software engineering was also followed: that of object-oriented visual programming, which seeks to provide visual programming accessibility, keeping as a basis the object-oriented programming principles.

2. Object-Oriented Visual Programming

A key contribution to object-oriented (OO) technology is the capability to write understandable systems for the resolution of complex problems. OO technology fosters a modular architecture programming style which promotes reliability and reusability, two necessary attributes for large-scale programming. The specification of what an object can do is separated from how other objects make use of it. Object-oriented visual programming (OOVP) refers to a visual programming language that supports the OO paradigm or to the use of a visual environment for an OO textual language.

In both cases, the languages support the capacity to describe the functionality of the system in terms of responsible objects and information processing by means of the remittance of messages. The visual systems through OO provide alternatives for the reusability of functions through classes and their instances. The purpose of OOVP is to combine the advantages of each approach: the reusability and extensibility of the OO technology with the accessibility of visual programming. Some of the advantages of such a combination are that the factorization of the functionality backed by the OO technology improves the capabilities for the creation of visual environments.

3.- Formal Specifications

Systems are developed based on their specifications. Hence, the quality thereof depends closely on the quality of these specifications. Incomplete, inconsistent or erroneously formulated specifications usually lead to the development of systems that do not comply

with the functionality requirements [PRE 93]. Formal specifications languages lead to the formal representations of the requirements that may be verified or analyzed [PRE 93]. One of the reasons why the formal methods are so promising as a new approach for the specification and design of problems is that it makes the computer engineer consider a software problem in an analogous fashion to an algebraic derivation or a demonstration of analytic geometry. The exactness involved in a mathematical specification forces the specifier to think more carefully about the problem at hand. One of the more widely used formal specification languages in the last few years has been language Z, "a language recognized as a specification language offering a powerful modularization mechanism using schemes and the calculation of schemes" [HOU 94] but which does not consider the possibility of managing specifications as a whole within other specifications. For this reason, many researchers and software development creators have currently created new languages, adding extensions to notation Z or based thereon and taking into consideration OO characteristics. Some of these languages are MooZ, Object-Z, Z++, OOZE, ZEST and ZERO.

4. Object-Z

The formal specification of a system in Object-Z comprises a series of definitions of class structures related through inheritance and instances. For Object-Z, class is the maximum specification structure. Insofar as Object-Z is an OO extension of language Z, this includes many of the characteristics of the language, especially the notation of schemes to define operations and the basic finite sequence and functions builders. The mathematical tool-kit in Z is also totally accepted in Object-Z. In the syntax of Object-Z, the definition of a class is represented by means of a box with a name that can have generic parameters as an option. Class characteristics are defined inside this box: these are the attributes and methods that define it through diagrams of states and operations. One diagram of states groups variables and defines the relationship existing among its variables [HOU 94], whereas the operation diagram reflects the changes in one or more state diagrams.

5. Why Jacobson ?

According to Booch, every software system has an inherent complexity. One of the elements that originate said complexity is the complicated domain of the problem. In the analysis of a problematic situation in the development of a software, requirements that compete with one another and that may even contradict one another can be found, more so when the user and the development creator have to define what the software should be capable of (its functionality), confronting the different perspectives of the nature of the problem that each one may have due to their different fields of expertise. On the other hand, Booch also affirms that "currently, even if the users were perfectly aware of their needs, there are few instruments for the precise capture of these requisites". Besides, he also says that the way the requirements are currently expressed is in general by means of large texts and in some cases with a few graphs, which makes comprehension difficult, can be interpreted ambiguously and oftentimes include design and implementation elements rather than requirements [BOO 91]. In this sense, one of the major contributions of Jacobson's methodology is the use of the 'use-case' as a basis for all the analysis and

development of the software. The use-cases provide a concrete representation of the system's requirements and its functionality by means of the description of scenarios that begin with the interaction of a user with the system, whereby a transaction or sequence of events is produced. A strong correspondence is guaranteed throughout all the models proposed in the methodology that cover all the different phases in a system's life cycle, thus ensuring a system that is robust, easily adaptable to constant change and highly reusable. In other words, Jacobson, with his OOSE methodology, has provided a series of well-defined steps for the construction of a quality software.

6.- Graphic Objects Edition System

Following Jacobson's methodology, the first phase was the analysis, where the requirements and analysis models were developed. The limits of the system and its functionality was defined in the requirements model and three models were implemented: the problem domain object model, the use-case model and the user interfaces description model. The intuitive proposal of the problem, the starting point for the requirements model, is set forth as follows:

A programming environment is desired, employing graphical interfaces for the manipulation of objects, that will generate a high-level code. A programming environment is understood to mean an application that provides an integrated environment, comprising programming tools that incorporate a series of predefined objects, with their graphical representation and corresponding semantic meaning and graphical interfaces such as windows, bar menus, icons, dialogue and help windows, that enable the creation, edition, storage, collection and execution of object-oriented programs.

As to the generation of a high-level code, this implies two semantic actions: A.-The generation of the code, translating the object-associated code, recovering the reserved words and placing them in a sequential file; B.-The maintenance and updating of the tables and data necessary to generate a valid code. In order to create a program by means of this application, a graph with the objects to be executed has to be created. The objects can be taken from the objects library or they may be custom-made by the user (based on existing classes). In order to add the corresponding attributes to each object, a protocol must be followed at the time of edition. Another way of doing this is leaving the protocol incomplete until the time of execution of the program, when the user will be required to complete it. Every time an expression is introduced within an object, the environment will validate said expression and if it is not syntactically correct, an error message will be issued and, until corrected, no further steps will be able to be carried out. In this manner, the application shall guarantee that at the time of execution, there will never be syntax errors. The application shall use the programming of events for the interruption of the use-cases and shall retake action when required, handling persistency.

The application, in turn, must be seen as an object that shall carry out the following functions:

File functions: being capable of creating, storing, recovering and bringing up instances of objects, enabling the existence of incomplete objects and their subsequent recovery.

Edition functions: carrying out basic edition operations: copy, cut, stick, select.

Graph functions: enabling operations such as click, double click, drag and drop of objects. The system will also have screens, bars, menus and tool bars.

Collection and Execution functions: to verify the syntax at the time of closing each edited object and to check for the existence of incomplete objects at the time of executing the program.

Based on this initial statement the intuitive objects in the system were extracted, see figure nº 1. Once the problem domain object model was achieved, the use-case model was carried out, identifying actors in the system and its interaction with each of them. In the case of this system, only one type of actor was located, the user or programmer. Likewise occurs with the interfaces description model. Following the analysis stage the analysis model was carried out which included the logical structure of the system, independent of the implementation environment. Here, the objects of the problem domain object model were found refined, expanded and classified according to three types of objects: entity, control and interface.

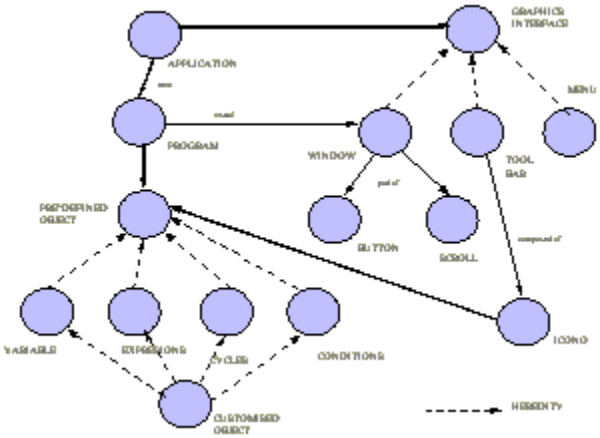

Figure 1. Object Model Problem Domain

Continuing on with Jacobson's methodological proposal, the interaction diagrams were drawn out and the interface for each of the objects in the system was extracted therefrom, taking into account the messages sent among objects. The design obtained was refined by means of formal specifications in the formal Object-Z specification language. These specifications were very useful to facilitate the communication and feedback between the design team and the implementation team. Ambiguities were thus avoided.

7. Conclusions

The fusion of three of the most recent trends in the field of software engineering: object orientation, visual programming and formal specification techniques.

Following Ivar Jacobson's OOSE methodology in all its stages assured the integrity of the product in correspondence with the theoretical globality initially proposed.

An object-oriented visual programming environment was designed that includes the advantages of object-oriented programming and visual programming, thereby providing the user with an integral environment where he can visually manipulate objects.

Not only was the application developed object-oriented; the system development process was also object-oriented in all the phases of its life cycle.

The use of a formal object-oriented language was added to the OOSE methodology for the formal specification of all the components in the system.

8. References

[BOO 91] BOOCH G. Object Oriented Design with Application. The Benjamin/Cummings Company, Inc., 1991.

[HOU 94] HOUGTON H. LANO K. Object-Oriented Specification Studies. Hertfordshire, Prentice Hall, 1994.

[PRE 93] PRESSMAN R. Software Engineering. A Practical Approach. Madrid, McGraw Hill, 1993.

[SPI 92] SPIVEY J. The Z Notation. A Reference Manual. 2nd. Ed. Hertfordshire, Prentice Hall, 1992.