Model Transformation Technologies in the Context of Modelling Software Systems

Óscar Pastor

Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n, 46071 Valencia, España
opastor@dsic.upv.es
Phone: +34 96 387 7000, Fax: +34 96 3877359

Abstract. Programming technologies have improved continuously during the last decades, but from an Information Systems perspective, some well-known problems associated to the design and implementation of an Information Systems persists. Object-Oriented Methods, Formal Specification Languages, Component-Based Software Production... This is just a very short list of technologies proposed to solve a very old and, at the same time, very well-known problem: how to produce software of quality. Programming has been the key task during the last 40 years, and the results have not been successful yet. This work will explore the need of facing a sound software production process from a different perspective: the non-programming perspective, where by non-programming we mainly mean modeling. Instead of talking about Extreme Programming, we will introduce a Extreme Non-Programming (Extreme Modeling-Oriented) approach. We will base our ideas on the intensive work done during the last years, oriented to the objective of generating code from a higher-level system specification, normally represented as a Conceptual Schema. Nowadays, though, the hip around MDA has given a new push to these strategies. New methods propose sound model transformations which cover all the different steps of a sound software production process from an Information Systems Engineering point of view. This must include Organizational Modeling, Requirements Engineering, Conceptual Modeling and Model-Based Code Generation techniques. In this context, it seems that the time of Model Transformation Technologies is finally here…

1. Introduction

It is true that programming technologies are being continuously improved during the last decades. New and more powerful programming techniques and tools have proposed. But it is contradictory to realize that, from an Information System's design and implementation perspective, things are running as bad as in the early seventies. This is a very remarkable situation: better and better programming alternatives, as poor as ever practical results.

The main problem historically faced by the Software Engineering and Information Systems communities is how to obtain a software product of quality. What we mean by quality in this context is to have a final software product that lives up to the

Dagstuhl Seminar Proceedings 06351
Methods for Modelling Software Systems (MMOSS)
http://drops.dagstuhl.de/opus/volltexte/2007/865

customer expectations, as they are gathered in the corresponding source Requirements Model. Going step by step from the Problem Space (conceptual level) to the Solution Space (software technology level) is supposed to be the task of a sound Software Process, intended to lead this Model Transformation Oriented set of activities.

Apparently, it is a simple task to state the quoted problem: how to provide an adequate Software Process that fulfills the goal of building software of quality, using the most modern programming technologies. But what has happened during the last forty years has made clear that this is a major problem. We need the right tools, but the reality is that they are not being provided. This is not a problem of lack of new, improved programming technologies. This is just a problem of failure in the daily practices associated to the always growing field of Information Systems Analysis, Design and Implementation. Looking back to the history, it is logical to conclude that the ever-present software crisis problem is not being solved by just working on programming-based software-production processes. We should wonder why this happens, and how this could be solved. The main objective of this paper is to sketch proper answers to these questions.

To do it, section 2 states the problem and introduces some alternatives intended to overcome it. In section 3, an example of a Software Process that covers the relevant phases of Requirements Modeling, Conceptual Modeling, and Software Product generation will be presented. In section 4, the main conceptual primitives provided by the OO-Method approach are introduced, as an example of what a Extreme Non-Programming environment would looks like. After that, some reflections around the more significant present and future trends related with how to put into practice Model Transformation Technologies will be discussed. Conclusions and references end up the work.

2. The Problem

During the last decades, software engineering courses around the world have extensively argued -using different formats- over how complicated to construct a software system is. Commonly, it is known that

- costs can be increased at any time,
- the final product is too frequently delivered out of time,
- the provided functionality is not the one required by the customers,
- it is not guaranteed that the system will work

It is also well-known and well-reported the cost of this low quality. For instance, in an study done by Giga Group, Gartner Group, Standish Group, CCTA and the Brunel University, and presented in [6], it is reported that at least $350 Billion are lost each year on Information Technology, due to abandoned projects, rectification of errors or consequential losses due to failures. Certainly, not all of this could have been saved by following a better process. Operational errors and changing business conditions are a source of unpredictable problems, even if a sound and complete Software Production Process is provided. In any case, the estimation of potential savings presented in the quoted report is at least $200 Billion per year, a huge amount of money lost due to the absence of the desired Software Process.

Summarizing, in terms of stating the problem, we face a problem that is at the same time simple to state and complicated to solve. The issue is that producing an Information System today is costly (because expensive resources have to be used over extended periods of time), much too slow for modern business conditions, very risky (in the sense that it is hard to control and with a high failure rate) and highly unsafe (because it introduces hidden failure points).

The main problem is that the development process has not changed much over the past 40 years; that is, the task of programming is still the "essential" task. Programming is the key activity associated with the fact of creating a software product. We don't say that programming technologies have not improved year after year, with new proposals and their associated tools. But the issue here is that considering the very bad results the programming-oriented techniques are historically obtaining, a simple question arises: is it worth looking for a better way of producing software?

We should ask ourselves why so many software systems historically fail to meet the needs of their customers. For decades, the "silver bullet" has apparently been provided to the community in the form of some new, advanced software technology, but always unsuccessfully. We have a large list of technologies that were intended to solve the problem: we could set down in this list Assembler, Third Generation Languages, Relational Databases, Declarative Programming, Methodologies and CASE tools (Structured Analysis and Design, Object-Oriented Modelling, UML-based), Component-based Programming, Aspect-based, Agent-Oriented, Extreme Programming, Agile Methods, Requirements Engineering, Organizational Modelling… But always, the same "phantom of the opera": the unsolved Crisis of Software notion.

Going back to this idea of "programming" as the key strategy for building a software product, the characteristics of the well-known Extreme Programming approach should be commented. Generally speaking, this approach is mainly based on the idea of having developers (programmers) that interpret the business descriptions providing by the Business Owner. Developers create and refine the source code that finally conform the operational system by properly integrating all the corresponding software modules. In any case, not only programming, but programming at the extreme is the issue here. The key skill is programming, and the most important idea is that "the code is the model".

According to all the negative results commented above, just going one step further on programming does not seem to be the correct way. On the contrary, it seems that one could easily conclude that, with such an strategy, no solution will be provided for the problem of providing software of quality. Under this argumentation, Extreme Programming could perfectly make things going worse till the extreme. But are there any alternatives?

2.1 Extreme Non-Programming

Proposed initially in [6], there is a first challenging alternative, and it is centered on the idea that programming is not the right strategy for Information Systems Design and Implementation. Extreme Non-Programming (XNP) stresses the idea that the

conceptual model is the key artifact of a Software Production Process. The model includes a structured business description, obtained through the interaction with the Business Owner (or any people playing this role from an organization perspective), and the corresponding machine readable views (the programs) should be generated from the model following a model transformation process.

In this case, the key skill is modeling, and the main idea is that "the model is the code" (instead of "the code is the model"). Under this hypothesis, a sound Software Production Process should provide a precise set of models (representing the differents levels of abstraction of a system domain description), together with the corresponding transformations from a higher level of abstraction to the subsequent abstraction level. For instance, a Requirements Model should be properly transformed into its associated Conceptual Schema, and this Conceptual Schema should be converted into the corresponding Software Representation (final program).

Assuming that behind any programmer decision, there is always a concept, the problem to be properly faced by any Model Transformation Technology is the problem of accurately identifying those concepts, together with their associated software representations. A precise definition of the set of mappings between conceptual primitives or conceptual patterns and their corresponding software representations, provides a solid basis for building Conceptual Model Compilers.

2.2 Conceptual Schema-Centric Development (CSCD)

XNP is not the only alternative. In [1], Prof. Olivé presents his CSCD approach as a grand challenge for Information Systems Research. Close to the central ideas of XNP, CSCD is based on the assumption that, to develop an Information System (IS), it is necessary and sufficient to define its Conceptual Schema.

Being the advancement of science or engineering the primary purpose of the formulation and promulgation of a grand challenge, the proposal focuses on the advancement of IS engineering towards automation. Even if this is not at all a new, disruptive proposal (see how [10], dated at 1971, already argues on the importance of facing the automation of systems building to improve the software production process), the fact is that, forty years later, this goal of automated systems building has not been achieved. There has been a lot of progress, definitely yes, but still today the design, programming and testing activities require a substantial manual effort in most projects. In neither of the two most popular development approaches –Visual Studio or Eclipse- the automation of the system building plays a significant role. Again, programming is the central activity and most (if not all) tool improvements are directed to help with the complex task of writing a program, but not with the task of compiling a conceptual model to avoid the repetitive, tedious and complex task of programming.

At this point, is it necessary to think about why this goal has not been achieved. Is it probably unachievable? Is it perhaps worthless to continue trying to? Of course, we don't think so. On the contrary, even accepting that a number of problems remain to be solved (especially technical problems, and problems related with the maturity of the field and the lack of standards), the situation is changing drastically, mainly from this last aspect of the absence of standards.

Recent progress on widely accepted and well-known standards as UML [4], MDA [3], XMI, MOF, J2EE, .NET, among others, are really providing an opportunity to revisit the automation goal. The MDA proposal is specially relevant, because it intends to provide Software Processes where Model Transformation is a natural consequence of this view of using models at different levels of abstraction, to represent the Computer-Independent Model (CIM) at the highest level, the Platform Independent Model (PIM) as the subsequent level of abstraction, and finally moving progressively to the solution space through the Platform Specific Model (PSM) and to the final code. The corresponding transformations between those models provide a kind of Software Process where all the previous ideas fit perfectly well. Industrial tools as OptimalJ [2] and ArcStyle [5] allows to experiment these approaches in practice with quite interesting reported results.

2.3 Model Transformation Technology through Model Compilation

In the same line of argumentation, there is still a third alternative. The overall objective of Model Transformation Technology (MTT) is to improve software development productivity. This objective is pursued by contributing to the realisation of the vision of model-driven software development. Model Compilation plays a basic role in achieving this objective. Under the hypothesis that behind any programmer decision there is always a concept, the task of a sound MTT approach is to correctly get those concepts, to define the corresponding catalog of conceptual patterns, and to associate to each of them its corresponding software representation. This set of mappings constitutes the core of a Conceptual Model Compiler.

Assuming that the history of Software Engineering (SE) has been one of growing levels of abstraction, MTT technology is just one step further. From the programming languages point of view, SE first moved from Binary to Assembler code, then from Assembler to what we consider today to be "Source Code". Next logical step is to move from Source Code to Conceptual Schemas. This is the step taken by Conceptual Model Compilers.

From the industrial point of view, there are some industrial tools providing such a kind of full Model-Based Code Generation environments, e.g. OlivaNova Model Execution [7], what allow us to conclude that we are really entering the time of Model Compilation seen as an operational technology. More details on this potential type of solutions are given in the next section.

Finally, before closing this section we want to remark that these three approaches are not the only existing alternatives. We could add the works of S. Kent on Model Driven Engineering [11], or the proposal of K. Czarnecki around Generative Programming [12] Al these works are providing a context where we can talk about real solutions to the problems commented for the current software production processes.

3. The Solutions

In practical terms, what we mean by Extreme Non Programming is just a software production process, where Requirements Modeling and Conceptual Modeling are the key issues, and where the Conceptual Model becomes the program itself in the sense that it can be transformed into the final software product through a process of Model Compilation.

Any Software Process XNP-compliant will start by specifying a precise Requirements Model, where the organization structure and relevant components are properly described, and where the software system is not –yet- an active actor. This Requirements Model has to be transformed into a Conceptual Schema, where a particular solution for the Software System is considered, but still at a higher level of abstraction than the programming level. Model Execution tools will be the responsible of compiling the model and generating the final application. Working within a XNP strategy, the final product will be compliant to the business model that originates it, as the whole process is conceived to assure that.

The alternatives presented in the previous section are closely related when we talk about concrete solutions. In the rest of this section, we present what could be seen as an example of a complete software production process that combines functional requirements specification, conceptual modelling (including data, behaviour and user interaction design), and implementation. It is defined on the basis of OlivaNova Model Execution (ONME) [7], a model-based environment for software development that complies with the MDA paradigm by defining models of a different abstraction level. Figure 1 shows the parallelism between the models proposed by MDA and the models dealt with in OO-Method [8], the methodology underlying ONME.

At the most abstract level, a Computation-Independent Model (CIM) describes the information system without considering if it will be supported by any software application; in OO-Method, this description is called the *Requirements Model*. The Platform-Independent Model (PIM) describes the system in an abstract way, still disregarding the underpinning computer platform; this is called the *Conceptual Model* in OO-Method. ONME implements an automatic transformation of the Conceptual Model into the *source code* of the final user application. This is done by a *Model Compilation* process that has implicit knowledge about the target platform. This step is equivalent to the Platform Specific Model (PSM) defined by MDA.

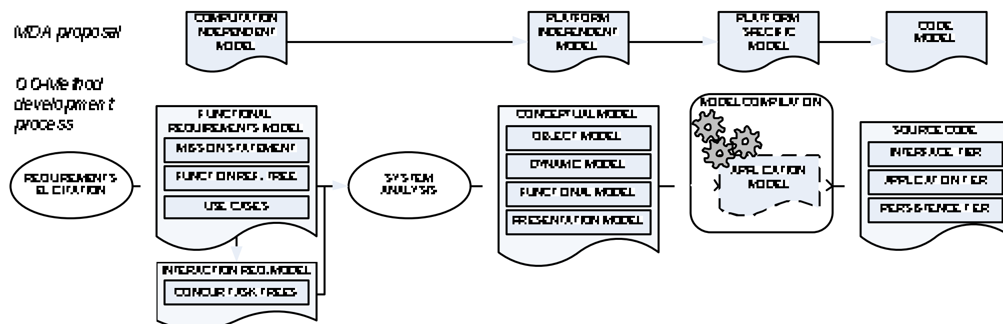

Fig. 1. An MDA-based development framework for Information Systems development

An important aspect to highlight is that data, function and user interaction modeling are the three basic components that must be present in every considered level of abstraction. The success of the final software product is only achieved when all these three basic modeling perspectives are properly integrated and managed.

The modeling strategies vary depending on the considered level of abstraction. For instance, at the requirements modeling level, either a functional, use-case oriented strategy or a goal-oriented modeling approach could be followed from a functional point of view. Concurrent Task Trees [9] could provide a sound basis for User Interface-oriented Requirements Gathering.

At the Conceptual Modeling level, an Object Model, a Dynamic Model, a Functional Model and a Presentation Model (see Figure 2) are the correct projection of the Requirement Models, while a Model Compiler is the responsible of generating the fully-functional Software Product. As these different and complementary view of the Conceptual Model are a basic part of the approach, we will introduce them in further details in the next section. With all that together, this approach provides a rigorous Software Production Process, that makes true the statement "the Model is the Code", and where programming in the conventional, old-fashioned way is not anymore the essential task, but instead modeling is.

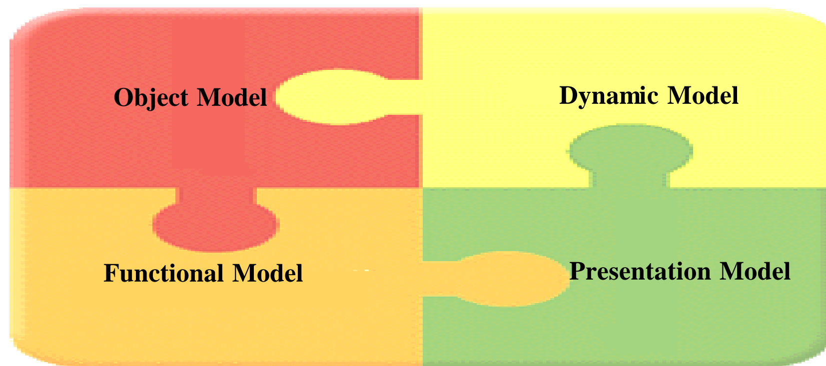

Fig. 2. A Conceptual Schema must integrate the four complementary views of object modeling, dynamic modeling, functional modeling and presentation modeling.

4. The OO-Method approach

As briefly commented in the previous section, OO-Method is a software development methodology based in a clear separation between the Problem Space (***what*** we want to build) and the Solution Space (***how*** are we going to build it). The definition of a problem (the abstract description of a system, represented in the corresponding Conceptual Schema), can be enacted regardless of any particular reification (concrete implementation of a software solution). This positions OO-Method as a sound

methodological foundation on which to build tools that embrace the MDA directive of separating the logic of software applications from their (multiple) possible implementations.

The formalism underlying OO-Method is OASIS, a formal and object-oriented specification language for the specification of information systems [Pas92]. This formal framework provides a characterization of the conceptual elements that we will need to accurately specify an information system. It encompasses two main components: the Conceptual Model and the Execution Model. Being the Execution Model the characterization of how a model is implemented in a target software development technology, we will focus in this section on the OO-Method Conceptual Model, to see how the basic building unit required to build a Conceptual Model are fixed and provided for practical use by the method.

4.1. Conceptual Model

The Conceptual Model comprises four complementary views: the Objects Model, the Dynamic Model, the Functional Model and the Presentation Model. All of them together constitute the whole Conceptual Schema specification. These four views allow the definition of all functional aspects of a system in an abstract (implementation-independent) yet accurate fashion by means of a set of conceptual elements (to which we will refer as conceptual primitives or conceptual patterns) with a precise semantics. Any of these conceptual patterns have an UML-based graphical notation, which hides from the developer (we should say the modeler in this context) the complexity and formalism of the underlying OASIS formal specification.

3.1.1. Objects Model

The Objects Model comprises a Class Diagram that graphically describes the structure of the system in terms of its classes with their properties and structural relationships (generalization, association/aggregation), thus providing a static view of the architecture of the system.

Classes have attributes of three kinds: constant attributes (those that get a value when the class is instantiated and do not change), variable attributes (those whose value can change during the lifetime of objects) and derived attributes (those whose value is calculated from the values of other attributes). In order to specify how the value of a derived attribute is calculated, a set of well-formed formulas called derivations can be defined.

Classes also have services, which define the signature of operations that can be invoked upon objects. Services, like “Operations” in UML have arguments or parameters, but also fall into two categories: events, which are atomic execution units; and transactions, which are molecular execution units that encompass other services (either events or transactions).

Events, in turn, can be stereotyped as “new” or “destroy”, whose semantics is that of creating a new instance of the class (assigning value to all its properties and relationships) in the case of the former, and that of destroying an instance of the class (and breaking relationships with related instances or cascade-destroy related

instances, depending on the features of each relationship). Figure 3 shows an example of the graphical notation in the class diagram for a class with attributes and services (a vehicle class taken from the context of a simple Rent a Car System). The semantics of events that are not stereotyped as "new" or "destroy" as we will later see, are defined in the Functional Model.

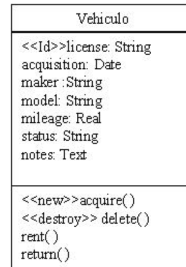

Fig.3 Class "Vehicle"

The semantics of a transaction is defined via a formula that states:

- the set of services that comprise the transaction,
- the initialization of each argument of each of said services, and,
- (optionally) Boolean conditions that must hold for each of said services to execute

The ability to define the functionality of atomic services (events) coupled with the ability to compose an arbitrary number of services into another (molecular) service, is a clear contribution of the OO-Method and allows to fully specify the functionality services (equivalent to UML operations) of classes.

The functionality of services can be further constrained by defining preconditions, well-formed Boolean formulas equivalent to precondition constraints in UML. Also, integrity constraints (class invariants) can be defined to prevent that the occurrence of services leave objects of the system in an invalid state.

As said before, generalization and association/aggregation relationships can be defined between classes. Generalization relationship specification deals with the inheritance specification. Both generalization and its inverse, specialization, can be specified. A child class can be seen as a role, activated when a given event or a class condition is fulfilled, and it incorporates the corresponding set of new properties that characterizes the role. It can be left out when the specified event occurs, or a leaving condition is satisfied. In any case, signature compatibility is required to assure consistency between parent and child classes.

Additionally, association/aggregation relationships allow declaring those well-known binary relationships between classes, including cardinality, static or dynamic aspect of the relationship, potential identity dependencies and stronger form of aggregation if composition is present. Figure 4 below depicts sample association and generalization relationships.

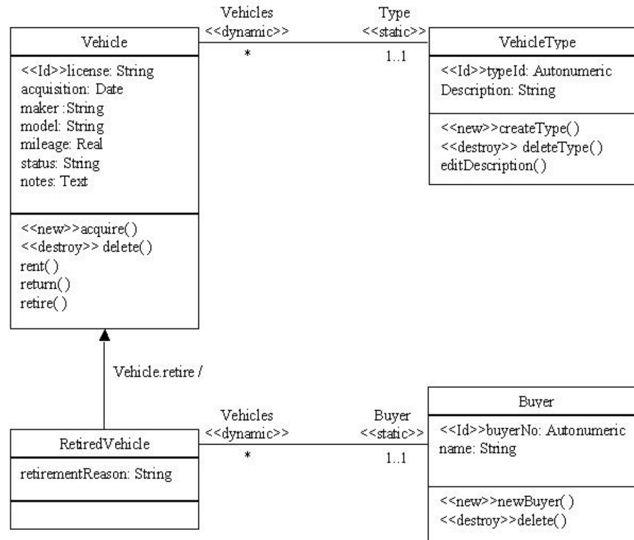

Fig. 4 Associations between "Vehicle" and "VehicleType", "RetiredVehicle" and "Buyer". Class "Vehicle" specializes into "RetiredVehicle" when event "retire" is executed.

Apart from generalization and association/aggregation relationships, OO-Method allows for the definition of agent relationships. An agent relationship is a directed relationship between two classes (one playing the role of agent class, the other acting as server class) which defines:

- which attributes in the signature of the server class, the agent class is allowed to query,
- which roles (the equivalent to association ends in UML) in the signature of the server class, the agent class is allowed to navigate, and
- which services in the signature of the server class, the agent class is allowed to execute

As illustrated in Figure 5, the graphical notation for the agent relationship is a dashed arrow from the agent class to the server class stereotyped with "agent".

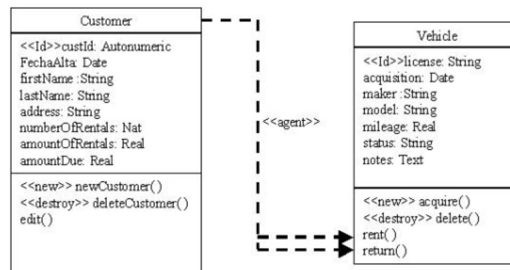

Fig. 5 Class "Customer" is agent of services "rent" and "return" of class "Vehicle"

This allows the modeler to specify a Conceptual Schema as a client-server model, where both what services are provided by classes to the system and what kind of permissions have classes to see or activate other's class properties are properly stated.

3.1.2. Dynamic Model

Once the static architecture of the system has been defined, the dynamic aspects (associated to intra-object control and inter-object communication) must be specified. This is faced with the Dynamic Model, that includes a State Transition Diagram per class, Triggers and Global Transactions specification.

There is a State Transition Diagram per class present in the Objects Model. A UML State Machine Diagram is used to specify the valid lives of objects of a class, that is, the way in which services of the class can orderly occur along the life of its objects.

To specify Class Triggers and Global Transactions, UML Collaboration Diagrams are used. Triggers represent those class services to be activated when a predefined condition holds in the source class. Global Transactions allow defining global services, meaning by global that they group different services declared in different classes, making them to constitute a single execution unit.

3.1.3. Functional Model

Once the system class structure has been specified, and system dynamics in the form of valid lives of class objects and inter-object communication mechanisms are properly defined, the only additional aspect to be considered is how a class service occurrence will change the local state of the involved objects. This is done through this Functional Model in which, to specify events functionality, Dynamic Logic Axioms of the form

$$f\,[s()]\,f'$$

are defined, where f and f' are well-formed formulae built over an alphabet of class attributes and s is the corresponding service whose functionality is being specified. The informal meaning of such formulae is that "assuming that f is true, f' will express the new resulting object state after the occurrence of s". These dynamic logic axioms are the result of applying a pre/post specification technique.

3.1.4. Presentation Model

Finally, user interaction has to be considered. If we want to build a complete system description, the way in which users will perceive the system when interacting with it needs to be incorporated in the constructed system specification that the Conceptual Model is. Hence, the three previous system views are complemented with a Presentation Model.

To specify user interaction properties, the Presentation Model creates an abstract UI specification built from a set of UI patterns belonging to a predefined catalogue of patterns provided by OO-Method. They collect the different situations that have to be considered for UI specification purposes (for instance, selection criteria for looking

for determined class instances, display set to fix which attributes should be shown to give more information about a selected object, available action set, to determine what actions can be executed within the scope of a given class service etc.).

Those UI patterns are structured in three different levels, as it is shown in Fig. 6. The first level of the hierarchy is called Hierarchical Action Tree, and it defines the first level of interaction. It groups the set of second-level interaction units that are those of:

1. Service Interaction Unit, intended to represent the IU components involved in the execution of a service.
2. Class Population Interaction Unit, oriented to determine how to access to (subsets of) the population of a class, including potential filter conditions, attributes to be viewed, potential class services available when in a particular class instance, etc.
3. Instance Interaction Unit, that is in charge of applying a similar idea but focusing on how to present a selected instance of a class.

Other complex interaction units can be built by composing the three previous types, to deal with more sophisticated kind of user interactions (as, for instance a Master-Details Interaction Unit).

Finally, the third level of the UI pattern hierarchy is constituted by the set of UI patterns that fixes the lower-level rules guiding the final concrete interaction to be executed. Depending on the interaction unit type, this includes -for instance- if a service argument is to be introduced and/or to be selected using a condition built on pre-specified class attributes, how to group service arguments, which attributes will be seen or what concrete set of services can be activated when accessing one instance, etc.

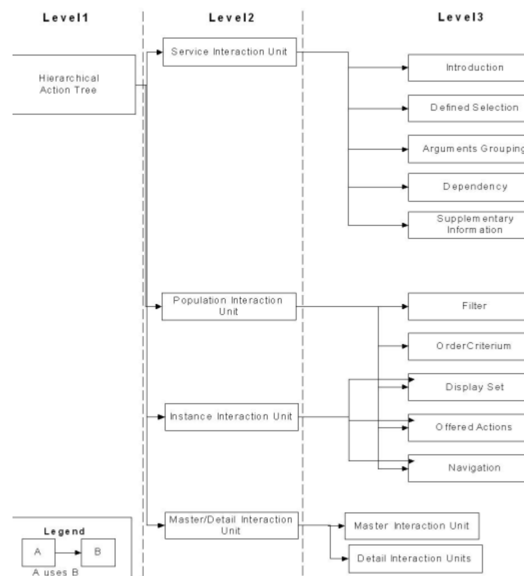

Fig.6 Structure of the Presentation Model

When all the components of the Conceptual are specified, it is time to proceed with the Model Transformation Process that will convert every conceptual primitive into its corresponding projection in the target software development environment where the final software product is to be built. To do that, an execution model needs to be defined. Further details on the corresponding execution strategy can be found in [8].

4. The present and the future…

We introduce in this section some final reflections. It seems to be clear that adoption of MTT requires tools. Why don't sound tools accompany these initiatives? Probably it is just a matter of time, and we are just now facing the right time. The usefulness of thes e tools is quite obvious considering the bad current situation. Usability should just be a logical consequence of adoption and usefulness.

We have discussed that we face a precise problem –how to produce software with the required quality-, and we have seen a set of potential valid solutions (XNP, CSCD, MTT…). A relevant set of challenging proposals are on the table to do SE better, and for the first time, CASE tools could be perceived as a powerful help to solve the problem of obtaining a correct and comp lete software product. For too many years, CASE tools have been perceived as a "problem generator" more than a "code generator" software artefact, mainly because instead of providing a solution for having a better software production process, they were seen as a new problem –how to use the CASE tool properly- added to the main software production problem. To avoid this situation, the modelling effort must be strongly and clearly perceived as a basic effort for generating quality software. And to do that, it must be proved that the model can be the code of the application to be. As it has been said before, this is the real challenge to overcome, and it is successfully being faced.

But would this be accepted even if tools are available? Why do industry mainly ignore current tools that really implement MTT? The programming inertia is still too strong. In a world were programming is seen as the essential SE-oriented task, it is not easy to make programmers accept that the model compiler will do the greatest part of the conventional programming work.

On the other side, this is not a new situation. Assembler programmers were in the past very sceptical when they were told that a close-to-english programming language called COBOL would do the Assembler programming wo rk through the associated Compiler, and even better than them. But nowadays no people are worried about the quality of the Assembler generated by Compilers, because they have reached by far the required level of reliability.

We will face the same situation with the Model Compilers. It is hard to accept by now that a Java program is just the result of a Conceptual Model Compilation process. But the truth is that this is possible, and that in fact there are already tools that provide this capability.

Another important question is to wonder about the role of Interaction / Presentation Modelling. It is curious to remark that, being user interface design and

implementation, one of the most costly tasks related with software production, it is not normally present at modeling level. Everybody knows the Entity-Relationship or the UML Class Diagram Model as a representative of Data / Object Modeling resp. When we talk about Functional Modeling, Data Flow Diagrams, UML Interaction Diagrams, etc. are well-known techniques. But what about asking which is a well-known, widely accepted User Interaction Modelling Approach? There is no consensus on that matter.

Probably, the reason of this situation is that traditionally the SE community has been interested in defining methods and processes to develop software by specifying its data and behaviour disregarding user interaction. On the other hand, the Human-Computer Interaction community has defined techniques oriented to the modelling of the interaction between the user and the system, proposing a user-oriented software construction. Within a sound MTT-based technology, these two views have to be reconciled by integrating them in a whole software production process.

In any case, the lack of a precise User Interaction Modelling strategy, properly embedded in the conventional data and functional modelling tasks is one common problem of many MTT-based approaches. To overcome this problem, a User Interaction Model must be present in any software production process to deal with those aspects related with the user interface to be. This model must be integrated with the other systems views in the context of a unified system model, and this must be properly faced at all the relevant levels of abstraction (requirements, conceptual, code).

5. Conclusions

Even though we've been talking about the Crisis of Software for the last decades, producing an Information System today is still a costly process (expensive resources are used over extended periods), much too slow than required for modern business conditions, very risky (hard to control, and with a high failure rate) and highly unsafe (due to the many hidden failure points). Considering that the software development process has not changed much over the last forty years, and that it has been basically centered on the idea of programming, it is time to wonder if it is worth looking for a better way. The fact is that Programming by itself, or complemented with some initial modeling activity, has failed in providing a solution to the ever-present software crisis problem. A new perspective is needed to solve the problem of generating a correct and complete Software Product.

If Programming is not the answer, its most modern versions as Extreme Programming could just generate "extreme failures", and we could be talking about the Crisis of Software issue for many more years. Extreme Programming is just putting even more emphasis on the programming task, while the well-known problems associated to the software production process are still present, if not enlarged. What we propose is just avoiding programming at the traditional low-level, machine-oriented level of abstraction, and to accomplish a new non-programming-based process, centered on concepts which are closer to the user domain and the

human way of conceptualizing reality. We called it Extreme Non Programming (XNP).

Accepting that this new perspective is really needed, in this paper we argue on a software production process based on the idea of Non-Programming. What we call Extreme Non-Programming (a term already introduced in [6]), is basically in the line of the modern approaches based on MDA, Model Transformation-based Technologies, Conceptual-Schema Centered Development and the like.

We have presented the fundamentals for providing those XNP methods (with their supporting tools) based on a different "motto": the idea that "the model is the code" instead of the conventional, programming-based where "the code is the model". According to that, Computer-Aided Software Development Environments intended to deal with Information Systems Development through the required process of successive Model Transformations, are strongly required.

In particular, precise Conceptual Schemas must be provided, and how to convert them into their corresponding software products, by defining the mappings between conceptual primitives and their software representation counterparts, should be the basis for a sound transformation process. These mappings are the core of a Model Compiler, intended to make real the following statement: "to develop an Information System, it is necessary and sufficient to define its Conceptual Schema". The automation of systems building becomes in this way an affordable dream, just looking for tools (some of them already existing as commented before) to justify its adoption in practice.

6. References

[1] A. Olivé, "Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research" Procs. Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005. Lecture Notes in Computer Science 3520 Springer 2005

[2] Compuware. www.compuware.com/products/optimalj/, last visited June 2006

[3] http://www.omg.org/mda, last visited June 2006

[4] http://www.uml.org , last visited June 2006

[5] Interactive Objects www.arcstyler.com/ , last visited June 2006

[6] Morgan,T. "Business Rules and Information Systems – Aligning IT with Business Goals", Addison-Wesley, 2002

[7] OlivaNova Model Execution System, CARE Technologies, http://www.care-t.com.

[8] Pastor O., Gomez J., Insfrán E. and Pelechano V. "The OO-Method approach for information systems modeling: from object-oriented conceptual modeling to automated programming". In Information Systems 26(7), Elsevier, 2001, pp. 507-534.

[9] Paternò, F., "Model-Based Design and Evaluation of Interactive Applications", Springer-Verlag, Berlin, 2000.

[10] Teichroew,D.,Sayani,H. "Automation of System Building", Datamation, 1971

[11] Kent,S. "Model Driven Engineering. IFM'02. Proceeding of the Third International Conference on Integrated Formal Methods, Springer Verlag, 2002, 286-298.

[12] Czarnecki,K. Generative Programming. Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Department of Computer Science and Automation. Technical University of Ilmenau, 1998