JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 1, January-February 2009

On the difference between analysis and design, and why it is relevant for the interpretation of models in Model Driven Engineering

Gonzalo Génova, María C. Valiente and Mónica Marrero
Knowledge Reuse Group, Informatics Department, Universidad Carlos III de Madrid

Abstract

In this paper we try to clarify the confusions that lie around the widely used terms "analysis model" and "design model" in software engineering. In our experience, these confusions are the root of some difficulties that practitioners encounter in system modeling, and sometimes lead to bad engineering practices. Our approach consists of placing the duality of analysis and design within a three-dimensional modeling space. Models are classified according to the reality they represent (first dimension), the purpose of the model (second dimension) and the abstraction level expressed in the model (third dimension). This classification facilitates the interpretation of models and the comprehension of model transformations as shiftings within this space.

1 INTRODUCTION

The central message of MDE/MDA (*Model Driven Engineering / Model Driven Architecture*) is that we have to move the core of software development from program code to models, up to the point of building models that can be directly compiled and executed [Kleppe et al. 03, Mellor et al. 04]. Models are nearer to human understanding than code, so that working with models will be less error-prone than working with programming languages. There are different kinds of models in software engineering: analysis and design models, structural and behavioral models, etc., so that it is of major importance to understand the meaning of each kind of model, how they are related, and how they evolve [Bézivin 05, Harel & Rumpe 04]. Our goal in this work aims specifically at investigating the place of analysis and design models in model-driven software development. Roughly speaking, "analysis" designates some kind of understanding of a *problem* or situation, whereas "design" is related to the creation of a

Gonzalo Génova, María C. Valiente, Mónica Marrero: "On the difference between analysis and design, and why it is relevant for the interpretation of models in Model Driven Engineering", in *Journal of Object Technology*, vol. 8, no. 1, January-February 2009, pp. 107-127
http://www.jot.fm/issues/issue_2009_01/column7/

solution for the analyzed problem; a "model" is some kind of *simplification* that is used to better understand the problem ("analysis model") or the solution ("design model") [Rumbaugh et al. 91]. However, this view is too simplistic, especially for the term "analysis model", since different kinds of models fall into the "analysis phase" of systems development and not all of them have analytic character [Karow & Gehlert 06].

But let's go deeper into the meaning of these terms. *Analysis* is a Greek word that equates to the Latin *decomposition*: "breaking a whole into its component parts (in order to better understand it)". It is opposed to *synthesis* (Greek), or *composition* (Latin): "building a whole out of its component parts". *Design*, instead, is related to drawing or making a blueprint of something before constructing it. The design anticipates and guides the production process, the "synthesis". In this view, we can say that design is part of synthesis, in a wider sense that includes the preparatory phases of synthesis, i.e. not only the pure construction activities.

The duality of analysis and synthesis, or analysis and design, has been traditionally used in software engineering for structuring the preliminary phases (or better, the *activities*, to avoid temporal connotations) of the software development process, where modeling is of crucial importance. As we will show in this paper, this single dimension is not enough to define the modeling space, i.e. the coordinate system where we can locate the different models employed in a software project. Other authors, such as Kent, have already studied some dimensions of this space, revealing that this is a field worthy of more research [Kent 02]. In this work, we are going to present three dimensions we consider particularly useful, without pretending they are the only ones that can be defined. We call them "dimensions" because they are independent (orthogonal) ways to classify the models and they serve to structure the modeling space. Each one of these three dimensions is related in its own way with the duality of analysis and design. We can add other (more or less orthogonal) dimensions in order to enrich the definition of the modeling space, such as "subject area", "aspect", "authorship", "version", etc. [Kent 02], but they are not necessarily related with the duality of analysis and design.

Our initial concern stems from this question: which is the characteristic difference between an analysis model and a design model? Our research on the related literature will show that there is not a unanimously accepted understanding of this difference among the community of software engineers. In other words, we think this traditional duality conveys really a triple difference that cannot be properly expressed through a single dimension, but rather requires three orthogonal dimensions (see Figure 1). Failing to acknowledge this triple difference leads to confuse the meaning of models, which has a practical relevance for the way models are interpreted and used in real software projects.

Briefly, the "reality" dimension is related to the thing represented in the model, either a software system or its application domain. Besides, a model can be used with two different purposes, to describe an existing system/domain or to specify a system/domain to be built: this is what we identify as the "purpose" dimension. Finally, something can be represented in a model from a more abstract point of view (black-box model, logical view) or from a more concrete point of view (white-box model, physical view): this is

what we call the "abstraction" dimension. The four following Sections progressively expose each one of these three dimensions, unfolding onto them the duality of analysis and design, and then applying this three-dimensional space to a better understanding of the process of model transformation using MDA concepts. In the last Section we give our conclusions.

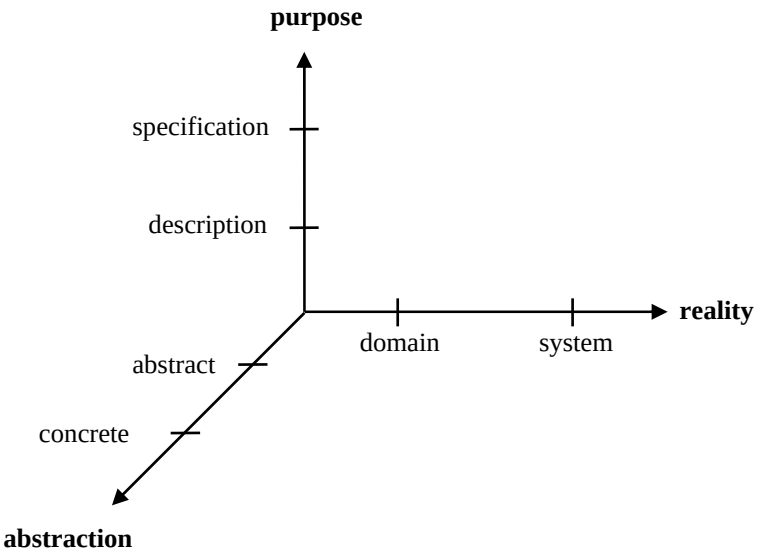


Figure 1. The three orthogonal dimensions onto which the duality of analysis and design will be unfolded. In each axis we identify two values corresponging to analysis (nearest to origin) and design (farthest).

2 FIRST DIMENSION: DOMAIN VS. SYSTEM

The first modeling dimension we are going to consider is related with the answer to this basic question: what does the model represent, i.e. which is the reality represented by it?

Models are used in software engineering to basically represent two different kinds of realities [Génova et al. 05]. First, the model can be used to represent a software system, or some part of it. It can be an existing system, or a system under development; likewise, the software system can be represented with different purposes and at different abstraction levels. We will deal later with these two aspects (second and third dimensions). At the moment we are going to focus on the fact that the represented reality is a software system. Most properly, then, we can talk in this case of a "system model", or a "software model". We will prefer the first term to refer to this kind of models. In the terminology of Jackson [Jackson 95], the system is called "the machine", but we think the term "machine model" suggests rather the physical structure of computers.

Second, and opposed to the "system model", models used by software engineers are often not properly referred to a software system, but rather to the *application domain of the software system*. Namely, that portion of reality that affects and is affected by the software system [Jackson 95]. In this case, people usually employ terms such as "model

of the real world", "model of the universe of discourse", "business model" or even "domain model", having different flavors that make these terms not perfectly synonyms. The term "model of the real world" is particularly inadequate, since, finally, the software system is also part of the real world. The term "domain model" seems preferable, since it is closer to the "domain experts" that usually are in charge of building this kind of models.

The terms *problem* and *solution* (or "problem model" and "solution model", "problem domain" and "solution domain", "problem space" and "solution space", and so on) are also commonly employed to denote this distinction. The contrast between the system and its application domain is often expressed with the terms "problem analysis" and "solution design", which shows that our first dimension is related with the duality of analysis and design. However, the terms "problem" and "solution" have a relative character which, in our view, makes them inadequate to characterize the first modeling dimension we are studying. Actually, the software system under development can be understood not only as the solution to a problem of information management in the considered business, but also as a problem that has to be solved itself; for instance, when the system is already specified, but not yet implemented. Furthermore, the considered business can be understood as an enterprise solution to a problem found in the surrounding human world. In short, the terms "problem" and "solution" contain additional connotations regarding the role of models in software development, beyond the pure reference to the *kind of reality represented*, which is the dimension we are interested in at the moment. Therefore, and in order to gain in precision, the terms we will employ are "domain model" and "system model".

The software system, in turn, may include some degree of *simulation of the application domain* it provides a service to, so that some authors and methodologies [Braude 01, Coad & Yourdon 91, Rumbaugh et al. 91] do explicitly recommend including the model of the domain as a part of the model of the system. Nevertheless, in general we should not expect a perfect correspondence between both models, since the goal of the software system usually goes far beyond a simulation of its application domain, or even might not intend it at all, as Potts points out [Potts 06]. Let's give some examples. A radar screen simulates the positions and velocities of aircrafts in a certain aerial space: in this case the degree of accuracy in the simulation is extremely important. On the opposite end, role playing games model imaginary worlds: in this case there is no proper simulation of any external world. Midway between these two examples, an electronic voting system contains some degree of simulation of its environment: voters, choices, etc.; however, the system is designed so that electronic votes do not simulate, but *replace*, manual votes, which will cease to exist once the system is deployed. Besides, the new electronic votes will not be a mere simulation of the old manual votes; instead, they will have different features (for example, a different way to authenticate the voter, or the possibility to change the vote before some specified deadline, and so on). In general, then, the software system will create a new reality that was not present in its application domain.

In other words, the system model and the domain model will usually share some common concepts (see Figure 2), specifically in that part of the system that simulates the domain, whilst some other concepts and descriptions will be true only for the domain or for the system [Jackson 95]. Those common concepts, however, represent different realities in each model: either an entity in the domain or its counterpart in the system; and it well may happen that the relationships between concepts are not exactly the same in the domain model and in the system model, because the system is not intended to perfectly simulate the domain. Besides, the common concepts usually do not constitute a full model of the domain, nor of the system.

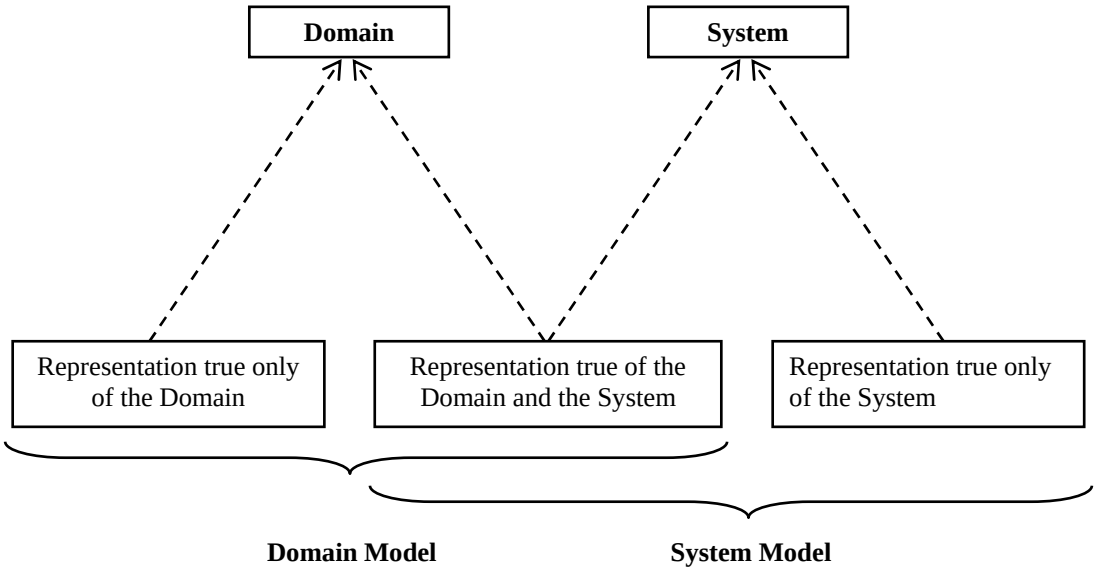


Figure 2. Shared concepts in the representation of the domain and the representation of the system (adapted from Jackson [18]).

Consider the following description of a voting system (see also Table 1):

1. Each **poll** has a **title** and a **date**, and it offers a number of different **questions**, each question having a **text** that expresses it.
2. A number of **voters**, each one with a **name**, can optionally participate in each poll.
3. In each poll, each voter selects one the possible **answers** to each question.
4. Voters can have opinion exchanges among them in order to **influence** each other before selecting their answers.

Well, what is this description about? Does it describe the application domain, or the software system, or both? The description itself does not tell it. Now suppose this description represents a certain domain where we want to replace the manual votes by electronic ones. We want to take advantage of the new possibilities offered by the future software system, so that we add a new item to the description:

5. A voter can maintain provisional answers after the **opening date** and before the **closing date** of the poll; each answer becomes **definitive** when the voter marks it as such, otherwise it is discarded on the closing date of the poll.

On the other side, suppose we are not interested in the implementation of a mechanism to facilitate the opinion exchanges among voters. Those exchanges still exist in the domain, no doubt, but they must not be translated in any way into the system. In other words, Item 4 is not a requirement for the system. The description of the existing domain consists of Items 1-2-3-4, whereas the description of the future system consists of 1-2-3-5. The situation is graphically depicted in the two diagrams of Figure 3. Most elements are shared by both diagrams, but some of them are exclusive of either Figure 3a or Figure 3b. The reflexive association **influences** between two **voters** in (a) disappears in (b), the single **date** of **poll** in (a) is replaced by the pair **openingDate** and **closingDate** in (b), and a new attribute **isDefinitive** has been added in (b) to the association class **selects**. (Note: there should be a constraint on both diagrams, specifying that voters can select answers of questions belonging only to the polls they participate in; we have omitted it for simplicity.) In the following Section we show that, besides a change in the represented reality, these two diagrams have a different purpose, too.

1. Each **poll** has a **title** and a **date**, and it offers a number of different **questions**, each question having a **text** that expresses it. 2. A number of **voters**, each one with a **name**, can participate in each poll. 3. In each poll, each voter selects one the possible **answers** to each question.
4. Voters can have opinion exchanges among them in order to **influence** each other before selecting their answers.
5. A voter can maintain provisional answers after the **opening date** and before the **closing date** of the poll; each answer becomes **definitive** when the voter marks it as such, otherwise it is discarded on the closing date of the poll.

Table 1. Description of a voting domain/system.

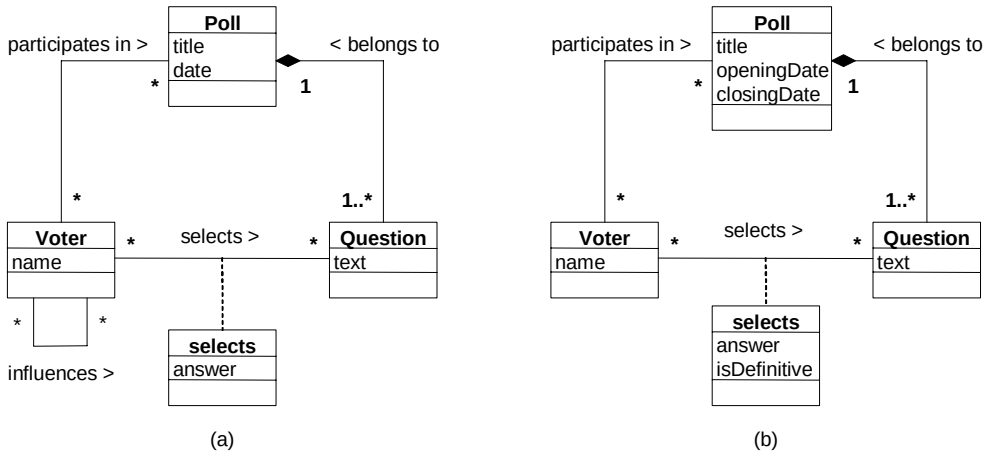
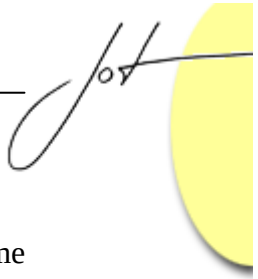


Figure 3. A model of the current manual voting domain vs. a model of the future electronic voting system. Is this analysis vs. design?



Deciding what part of the domain or business is going to be automated implies that some part of it will not be. This is an important decision, which can be better settled if the part of the domain that will not be automated is also modeled. That is, building a complete model of the current domain should be an activity performed before deciding the requirements for the new system.

3 SECOND DIMENSION: DESCRIPTION VS. SPECIFICATION

The second modeling dimension is derived from this question: what is the model used for?

In software engineering, models can be used mainly in two different ways, traditionally known as *forward engineering* and *reverse engineering*. Models will be either *descriptive* or *specificative*, as Seidewitz denominates them [Seidewitz 03], i.e. a description of *something that exists*, or a specification of *something that must exist*. (Instead of "specificative" models, others would prefer "normative" or "prescriptive" models [Bézivin 05].) Let's focus first on the system model. In forward engineering, the model is used as an anticipation or *specification* of the software system to be built; the model can be used as a template to guide the construction of the system, as a platform to simulate the behavior of the system before actually constructing it, even as a starting point to (semi)automatically generate the system, etc. In reverse engineering, instead, the model is used as a conceptual tool or *description* to understand an existing system that has to be maintained or improved. In both cases, models are analogues of the things they model, sharing some interesting properties and structure that makes them useful in engineering [Jackson 95].

As we can see, these two usages of "scale models" in software engineering are very similar to those which occur in other branches of engineering, such as architecture, electronics, mechanical engineering, etc. Besides, these two usages are related to the useful distinction between *model-as-original* and *model-as-copy* [Marcos & Marcos 01]: in forward engineering the scale model is used as the original from which the software artifact is constructed; in reverse engineering the scale model is a simplified copy of the software artifact it represents, used to better understand it. Finally, we can put all these notions in relation to the duality of analysis and synthesis (or analysis and design): reverse engineering is a *process of analysis* where the existing system is understood by means of a model-as-copy, whereas forward engineering is a *process of synthesis* in which the system is constructed starting from a model-as-original (see Table 2). These two models (model-as-copy at the end of the analysis process as its result, and model-as-original at the beginning of the synthesis process as its anticipation) are often called in engineering "analysis model" and "design model", respectively. The development process will rarely consist of pure forward or reverse engineering: therefore, if forward engineering is preceded by some kind of reverse engineering, then the design model will be immediately preceded by the analysis model.

Forward engineering	Reverse engineering
model → system	system → model
Specification	description
model-as-original	model-as-copy
design model	analysis model
process of synthesis	process of analysis

Table 2. Two different usages of models in software engineering.

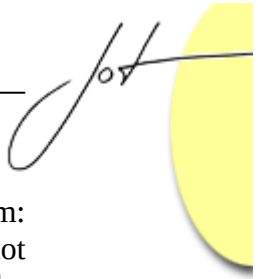
Electrical engineers use these terms as we have just explained: "circuit analysis" means finding a mathematical model-as-copy of the circuit, namely its transfer function (reverse engineering); "circuit design" means finding a circuit, among the various possible ones, that satisfies a given transfer function, which is the mathematical model-as-original (forward engineering). However, and maybe paradoxically, this perfectly legitimate sense of the term "analysis", very close to the original meaning of the word (see the Introduction), is not the most usual among software engineers. That is, we have to distinguish between the classical sense of the word "analysis" as explained above, and the usual sense in software engineering (see next Section).

The distinction between specification and description can be applied not only to the system model, but also to the domain model; therefore, they are *two orthogonal dimensions*, as shown in Table 3. Indeed, software engineers often work with domain description models as an aid to understand the requirements the software system must meet, and to build an adequate vocabulary to represent the elements in the system. Equally, if the application domain is not regarded as an immutable reality, but modifying it is within the scope of the project (business re-engineering), then it will be necessary to create a specification model of the desired new domain. In fact, an application domain that will remain completely unaffected by the introduction of a software system that supports part of the tasks that previously were manually achieved, or by means of other software systems to be substituted, is hardly conceivable. However, this aspect is very often ignored in software projects.

Specification	domain specification model	system specification model
Description	domain description model	system description model
	Domain	System

Table 3. First and second modeling dimensions: domain vs. system, description vs. specification.

Let's have a new look at Figure 3. These two models respectively *analyze* (describe) the current application domain and *design* (specify) the future software system, showing again that our second dimension is related with the duality of analysis and design. But both parts of the figure could be referred to the domain: a description of the current domain, and a specification of the future domain, where the possibility of some voters influencing others in Figure 3a is not any more considered in Figure 3b, but the answers



can be either provisional or definitive. Equally, both parts could be referred to the system: Figure 3a, a description of the current system that provides for opinion exchanges but not for provisional answers, and Figure 3b, a specification of the future system where the required changes will be implemented. The two dimensions, domain-system and description-specification, are orthogonal.

A typical software project will include a domain description model (reverse engineering), starting from which a system specification model is built (forward engineering). Note that, in this case, in addition to a different *represented reality*, the *purpose of the model* is also different. Nevertheless, too often a lack of rigor is observed in distinguishing both models, putting the success of the project at risk. This is what happens when it is encouraged to include the model of the domain as a part of the model of the system [Braude 01, Coad & Yourdon 91, Rumbaugh et al. 91]. The elements of the domain that will have a counterpart in the system must be carefully selected, and uncritically assuming that the whole domain is simulated by the system is pernicious.

It cannot be denied that understanding the real world, i.e. *analyzing* it (in the classical sense of the word), is most useful to produce a good user requirements specification, and therefore a practical software system that solves the needs of users. This is often reflected in the fact that the system specification model uses the concepts found in the domain description model (see again Figures 2 and 3). The difference is subtle, but real: using the *same vocabulary*, these two kinds of models represent two different realities, with a different purpose, too. Maybe this is the source of the confusion we try to avoid. Making a model-as-copy of the real world to better understand it is a perfectly legitimate and useful practice, even though it must be carefully distinguished from making a model-as-original of the future system.

The contradictory uses of the term “analysis” often foster this confusion, since analysis, as opposed to synthesis, means for many “understanding the real world”, i.e. making a domain description model (classical sense); whereas, for many others, analysis means, as opposed to design, “high level system specification”, i.e. making a system specification model (software engineering sense). In the next Section we will give more details about this last distinction between analysis and design.

4 THIRD DIMENSION: ABSTRACT VIEW VS. CONCRETE VIEW

The third modeling dimension we are considering finally stems from this question: how is the reality represented in the model, i.e. what is the abstraction level expressed in the model?

As we have already commented on, when software engineers say “analysis”, they can refer mainly to two different kinds of modeling activities, or they can even mix them carelessly: building software specification models (system forward engineering) or building “real world” description models (domain reverse engineering).

Let’s take first the context of software systems forward engineering. It is very common to characterize analysis as specifying the *what*, whereas design as specifying the

how [Jackson 95]: what is the system supposed to do for its users, how will it do it (in fact, this expresses the classical principle of information hiding in software engineering: separation of the specification from the implementation). The analysis must then *capture user requirements* without prematurely adopting implementation decisions, i.e. omitting technology-dependent details [Kaindl 99], and using concepts solely drawn from the problem domain. In contrast, the design must *define a software solution* that effectively and efficiently satisfies the requirements specified in analysis. In doing this, the design model will incorporate new artifacts (new classes, new attributes and operations for the classes, etc.) and it will take into account the concrete technological platform on which the software system is to be built. This particular distinction between analysis and design has been often expressed as *logical design* vs. *physical design* [Graham et al. 98, Jackson 95], or *specification model* vs. *implementation model* [Cook & Daniels 94].

The essential point here is that *both kinds of models*, analysis and design, or whatever we call them, *do represent the same system*: both are system models, not domain models, and both are used in the context of a forward engineering process, even though they do have an important difference in perspective [Høydalsvik & Sindre 93], and thus in the way they represent the system-to-be-built. The *way of representation*, the relationship with the represented reality, adopts a different viewpoint in each case: the analysis model represents the *external, higher level or logical view* of the system (the more abstract conceptual view), whereas the design model represents the *internal, lower level or physical view* (the more concrete implementation view). We can call them, respectively, "black box model" and "white box model", too. (Note we are using different metaphors to express the abstract-concrete dimension that are not perfectly equivalent: higher-lower, external-internal, logical-physical, black-white, etc.). Each view adopts *a different abstraction level* that produces a different kind of model of the same system, so that it is represented in a different way, too: the design (physical view, internal view, or white box model) omits more or less implementation details, whereas the analysis (logical view, external view, or black box model) omits the implementation itself; the analysis specifies what the design realizes.

This notion of analysis does not correspond to the classical notion of analysis, as something dually opposed to synthesis. On the contrary, this notion of analysis is in fact *a first step in synthesis*, followed by design, the second step, and other subsequent steps leading to the complete construction of the system. This notion of analysis as part of a synthesis process can be easily recognized in a multitude of software engineering practices and text books, even though it will be rarely recognized explicitly — there are exceptions, of course, such as Jackson or Jacobson [Jackson 95, Jacobson 95]. That is why we call the attention on the difference between the "classical sense" and the "software engineering sense" of analysis.

The *use case model* is a good example of this. Although the use case model includes both use cases and actors, its main goal is to model *the system* as it interacts with the outer world (i.e. the use cases); the actors represent mere external agents without any contents. The use case model does not represent interactions among actors, interactions

among actors and other systems, or interactions among other systems in the environment, i.e. it is not a complete business model. Due to its "high level" character, closer to the user than to the implementation, use cases are usually defined within the "analysis" activity of a project. However, a use case is a specification of the expected functionality a software system must provide, described through typical user-system interactions [Génova & Llorens 05]: it should be obvious for everyone that a use case is modeling a software system to be built, not at all the "real world" as it is before the system exists, or as it will be afterwards [Hay 99]. In other words, use case modeling is part of a *synthesis activity* [Bittner 03]. The same applies to *user requirements* in general (expressed through use cases or not): they are specifications of a desired computer information system, i.e. they are not aimed at describing the world outside of the computer. That is, user requirements *originate* in the environment (the problem domain), but they are *about* the system (the solution domain). In fact, user requirements, where the textual form usually predominates, can be properly considered as a model of the required system (a *textual model* indeed, who said a model has to be graphical?): a model-as-original from which the system has to be built. Since many conceive analysis as an activity where user requirements are elucidated and clearly written [ESA 95, Jackson 95, Kaindl 99], it is clear that they are regarding analysis as part of a forward engineering process, i.e. synthesis.

Figure 4 illustrates the abstraction dimension with the electronic voting system again. Figure 4a shows the abstract view of the system we already know. Figure 4b shows a refined, more concrete view, where two associations have been replaced by intermediate classes (a usual transformation when the platform does not provide primitives to express many-to-many associations). Both models express the same electronic voting system at different abstraction levels, which are related with the traditional duality of analysis and design. Contrary to a change in the represented reality (first dimension) or in the model's purpose (second dimension), the change of abstraction level along the third dimension is rather gradual, i.e. there may be many intermediate levels between the highest and the lowest ones, many gray tones between the black box and the white box. In other words, there is no sharp distinction between analysis and design, if they are understood as system specifications at different levels of abstraction.

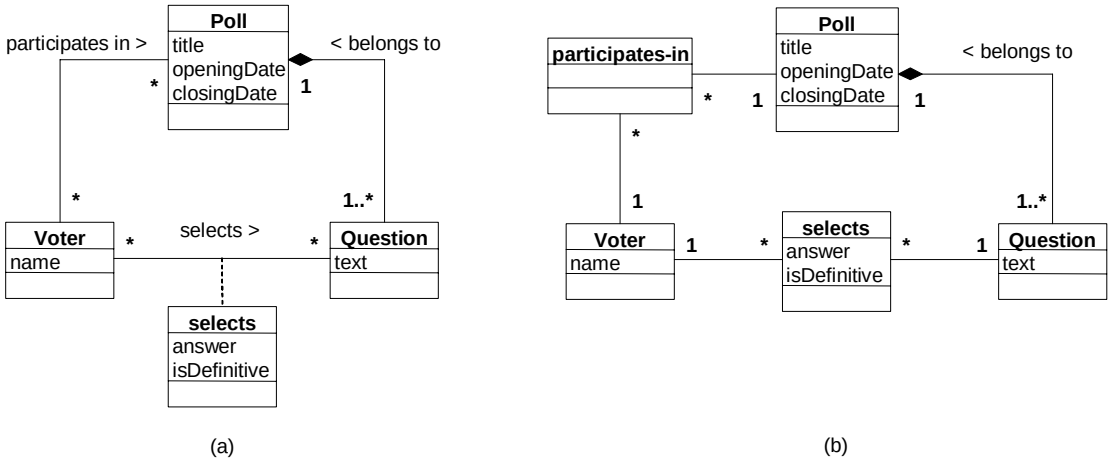
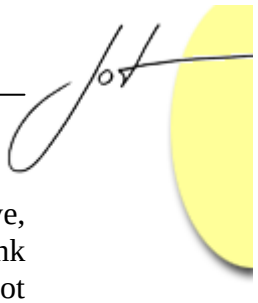


Figure 4. Two views of the electronic voting system at different abstraction levels.
Is this analysis vs. design?

Higher level models are built onto lower levels using the more concrete primitive constructs provided by the latter; in other words, higher level representations abstract away *low level* details. Nevertheless, an abstract model may contain a great amount of detail if necessary, provided they are *high level* details. Do not confuse the mere amount of detail with the abstraction level. For example, Figure 4a could be more or less detailed, showing more or less attributes and operations: these changes would not affect the technology independence character of that model. Equally, Figure 4b is rather low detailed, being nevertheless technology dependent through the use of intermediate classes to solve the problem of many-to-many associations.

In this view, the transition from the analysis model to the design model (from the logical view to the physical view) can be hard or not, but it neither changes the represented *reality* (first dimension) nor the model's *purpose* (second dimension): both of them are, within a synthesis process [Jacobson 95], specification models of the same software system, yet at a different *abstraction level*. Our point here is not whether the transition is easy or difficult. In fact, the design has to provide a creative solution for the problem specified in the analysis (i.e. the required system), and this rarely will be easy [Høydalsvik & Sindre 93, Kaindl 99]. Moreover, a good design model that takes into account non-functional requirements such as performance, reuse, maintainability, etc., might hardly resemble the analysis model specifying the same system [Haythorn 94, Høydalsvik & Sindre 93, Karow & Gehlert 06, Parnas 72]. But this does not negate that both are models of the same software system, which is our point: both analysis and design models are *system models, not domain models*.

We have considered until now in this Section the duality of analysis and design (or better, abstract view-concrete view) within the context of software systems forward engineering, i.e. in relation to system specification models. However, *this dimension is also orthogonal* to the other two previously examined dimensions, domain-system and



description-specification. Specifically, domain models, either descriptive or specificative, can be abstract or concrete. For example, the process of getting a mortgage from a bank can be modeled at different levels of abstraction. In this case, "concrete" does not necessarily mean "technology dependent", but it is still meaningful. Therefore, we can make sense of any combination of "coordinates", although actually some of them are less frequently used in typical software engineering processes. For example, we can talk about a system reverse-engineering model that represents its internal view (system-description-concrete view), with the intention of understanding well the details of the current system before building a more abstract model; or about a domain forward-engineering model that represents its internal view with the intention of modifying the business (domain-specification-concrete view); and so on. In other words, it makes sense defining a *three-dimensional modeling space*, with three orthogonal coordinates and two different values for each coordinate, with a total of eight potential combinations, as shown in Table 4 (see also Figure 1).

The terms "analysis" and "design" usually have additional connotations that link them to the context of software systems forward engineering. That is why expressions such as "domain description design model" (domain-description-concrete) are strange to our ears. The terms "external" and "internal" could also induce to confusion with the first dimension (domain vs. system), whereas "specification" and "implementation" partially conflict with the terms we chose for the second dimension (description vs. specification). The terms "logical view" and "physical view" could seem acceptable to characterize this dimension; however, they are less satisfactory to characterize different abstraction levels of models that represent the domain. Therefore, we consider the general terms "abstract view" and "concrete view" more adequate to characterize this dimension.

Specification	Abstract view	domain specification abstract model	system specification abstract model
	Concrete view	domain specification concrete model	system specification concrete model
Description	Abstract view	domain description abstract model	system description abstract model
	Concrete view	domain description concrete model	system description concrete model
		Domain	System

Table 4. Third modeling dimension: abstract view vs. concrete view.

5 MDA CONCEPTS AND THE MODELING SPACE

Within the conceptual framework of model engineering, the software development process can be understood as a *trajectory of transformations through the modeling space* we have defined. A typical trajectory could be, for example (see Figure 5), starting from a domain-description-abstract model (which is often called "real world model", or even "analysis model"), and then jumping simultaneously over the first and second dimension to obtain a system-specification-abstract model ("requirements analysis", often called

"analysis model", too, but having a radically different sense from the previous one, which can turn out into dangerous misunderstandings), and finally jumping over the third dimension to generate a system-specification-concrete model ("design model"). Another typical trajectory is found in the process of substituting a legacy system with a technologically more up-to-date one, which basically provides the same functionality: legacy system-description-concrete model (this might be omitted), legacy system-description-abstract model, new system-specification-abstract model, and new system-specification-concrete model. In this case we have moved only over the second and third dimensions (the domain has not been modeled), even though the represented system has certainly been changed in the second jump.

Shifting the model over the three dimensions we have explained *is not conceptually simple*, nor should be presented as such. Establishing the relationship between a domain-description-abstract model and a system-specification-abstract model is not easy. Although the distinction might seem obvious according to our arguments, in fact too often both models are confused, mainly because of the *ambiguity* of the term "analysis model", and possibly also because of the employment of *a uniform notation* to represent very different realities, as Kaindl points out [Kaindl 99]. For example, when the analysis classes are said to represent real world concepts, whereas design classes represent code fragments [Fowler & Scott 04, Høydalsvik & Sindre 93, Kaindl 99], here "analysis model" means domain-description-abstract model. On the contrary, when the recommendation is to start with an analysis model that will be included later within a design model, and completing it with other artifacts that are required to provide the solution [Braude 01, Coad & Yourdon 91, Rumbaugh et al. 91], then "analysis model" means system-specification-abstract model. It is a common error to confuse both models, taking the analysis model as a description of the real world (domain model), and then using it as a specification of the system to be constructed (system model), i.e. failing to properly acknowledge the limited way in which the software system simulates the application domain (see again Figure 3 and Table 2). As we have already shown, the system specification cannot be automatically inferred from the domain description: no machine can replace the stakeholders in the task of deciding which part of the domain must be simulated, and which part must not. Since the system is not generally a copy or simulation of the world outside [Potts 06], *a reverse-engineering model of the domain cannot serve as a forward-engineering model of the system* [Génova et al. 05]. Taking an analogy from civil engineering, a model of cars and rivers cannot be a model of the bridge the cars need to cross over the river.

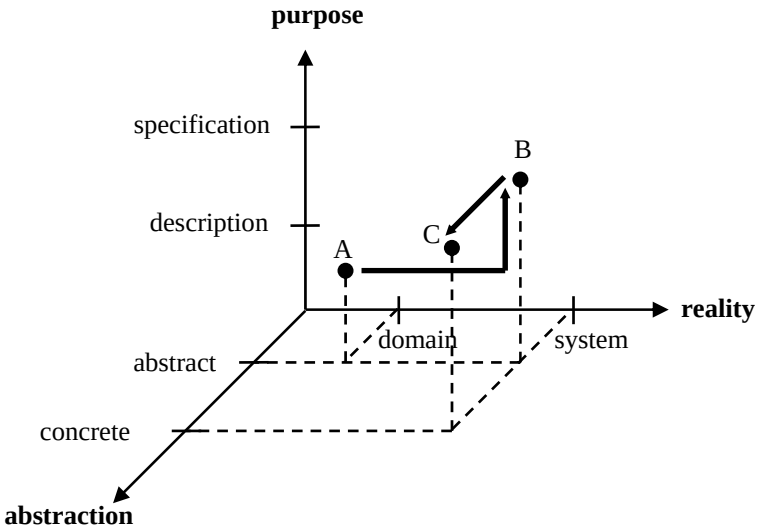


Figure 5. A typical trajectory through the modeling space. A: domain description abstract model. B: system specification abstract model. C: system specification concrete model. Informally, both A and B are often called “analysis model”, and C is called “design model”.

The relationship between the abstract view and the concrete view is not simple either, as the multiple efforts to define and implement model transformations demonstrate. In MDA [OMG 03] the distinction between abstract and concrete has been particularized both as computation independence and platform independence, but there are still conceptual and terminological problems. The term CIM (Computation Independent Model) is defined as “a view of a system from the computation independent viewpoint”, “a model of a system that shows the system in the environment in which it will operate” (note our emphasis on being a model about the system, not about the application domain), which fits well with the notion of analysis as specifying the what, not the how, of the system. The purpose of the CIM is to model the requirements for the system, describing the situation in which the system will be used. Like the use case model, then, the CIM may include external agents of the system, but it is aimed at modeling the system, not the external agents. Unfortunately, the CIM is nearly equated in MDA to a domain model or a business model, which fosters the confusion between the domain model and the system model (our first dimension). In earlier versions of OMG documents [OMG 01], where the term CIM had not been coined yet, the term employed was “computation independent business model”.

Summing up, a “computation independent” model may be understood mainly as: a) representing the domain, not the system; or b) representing a very abstract view of the system. Probably, the ambiguity of the term CIM can be traced back to the terms “domain model” and “business model” themselves. In this paper, we have always referred these terms to the complete model of the application domain. But we cannot ignore that many people uses these terms as referred to the common part only, i.e. the shared concepts in the application domain and the system (see Figure 2), omitting those elements in the application domain that are not so relevant for the system (true only of the domain) and

the technological aspects of the system that do not fit in an abstract model (true only of the system).

Later on in the MDA Standard we read: "an MDA specification of a system CIM requirements should be traceable to the PIM and PSM constructs that implement them", which clearly relates both terms PIM (*Platform Independent Model*) and PSM (*Platform Specific Model*) to the notion of design we have considered, as realization of an abstract model. Correspondingly, some authors regard the PIM-PSM mapping as a *refinement of designs* [Kent 02], i.e. both models should be considered as a result of the design activity. However, other authors think that analysis vs. design is parallel to PIM vs. PSM [Kleppe et al. 03, Mellor et al. 04]. But this different usage of the terms is not very relevant: if analysis is considered a first step in synthesis, then there is no practical difference between "analysis-design" and "refinement of designs".

Some authors consider PIM-PSM and PSM-PIM mappings are *vertical*, whereas PIM-PIM and PSM-PSM mappings are *horizontal* [Clark et al. 08, Sendall & Kozaczynski 03], suggesting that vertical mappings correspond to a refinement from higher to lower levels of abstraction. However, the CIM is also considered in MDA to represent the highest level of abstraction in system modeling, i.e. computation independence and platform independence are not seen as orthogonal dimensions, but rather as subsequent steps in the abstraction refinement dimension (our third dimension). Moreover, the MDA Standard acknowledges that "platform independence is a matter of degree" and that "any model is platform independent or platform specific only relative to some particular class of platforms", as illustrated by the repeated use of the MDA pattern. Some authors have emphasized the relative character of the MDA term "platform" [Atkinson & Kühne 05, Bézivin 05, Brown 04, Karow & Gehlert 06, Kent 02], and consequently of the PIM-PSM relationship: the output of a PIM-PSM transformation is used as the input for a new transformation, i.e. the output PSM becomes a new input PIM. In other words, PIM and PSM are not essentially different models; rather, they should be considered grey-box models ranging from the absolute black (the CIM) to the absolute white (the final implementation) in successive transformations, corresponding to a shifting along our third dimension. However, this is not the only existing interpretation of PIM and PSM. Bézivin [Bézivin 05] reinterprets Jackson's relationship between "the world" and "the machine" (see Figure 2) in the sense that the PIM is a model of the world that is merged with a model of the machine, the PDM (*Platform Description Model*), to yield the PSM. While this may be partially true, it is not the whole truth: the PIM may well represent the shared concepts between domain and system, but it is not a complete model of the domain, as we have already shown.

A last remark on the term "abstraction". We have constantly used "abstraction" in this paper with the meaning of "simplification of low level details". But there are other forms of abstraction that do not fit well in this "vertical" dimension. For example, when a model concentrates in a certain aspect of a system, other aspects which are not necessarily "low level" are also "abstracted" away: think of a blueprint of a building that concentrates in the mechanical structure, omitting the electrical and sanitary systems.

These other kinds of “horizontal” abstractions may well be used to define other useful dimensions of the modeling space, but we have not considered them in detail, since they are not clearly related with the duality of analysis and design.

6 CONCLUSIONS

We have examined *two different meanings of analysis models* that can be found among software engineers: a model that specifies the abstract or logical view of a software system (the “software engineering” sense), or else a model describing the “real world” that constitutes the context of the desired software system (the “classical” sense). The software engineering sense is closely related with our third dimension, whilst the classical sense involves both our first and second dimensions. Both views are very often confused. Actual practice corresponds generally to the use of system specification models, even though theoretical explanations go frequently for domain description models. A moderate danger of the confusion is to believe that we are modeling the real world, when we are really doing a high-level specification of the software system. A much more serious danger is to build a system that needlessly matches the structure of the real world. On the contrary, it would be legitimate to *conceive analysis as a task where both models are built*, in a close relationship but properly distinguished at the time. If you have to use the word “analysis” in the context of software engineering, choose whatever sense you prefer for it, but be conscious of your choice and of the possible misunderstandings among your audience.

Even though the conceptual framework of object orientation, and a modeling language such as UML, may make the transition from analysis to design easier, we must not think that it is a quasi-automatic, seamless transition: this would be a misleading simplification of the question. We believe that *unfolding the duality of analysis and design* onto the three-dimensional modeling space we have presented may contribute to avoid the frequent misunderstandings that hinder the use of models in software engineering, and that it may be useful to better define model transformations as shiftings within this space.

If we want to automate model transformations, then we need a set of *transformation rules (mappings)* that take into account the particularities of each dimension. Hence we need, first of all, a good understanding of the dimension(s) involved in the transformation. The first and second dimensions are respectively related with the kind of *reality* represented in the model (domain vs. system) and with the *purpose* of the model (description vs. specification). Both of them are conceptually simple to understand but, in our view, moving over these dimensions requires rules that generally will neither be simple nor automatable. Indeed, the function of the system cannot be automatically deduced from the structure of the domain (safe when the system merely simulates it): no machine can replace the stakeholders in the task of establishing the requirements the system must meet.

The third dimension is related with the *abstraction level* adopted by the model (abstract vs. concrete). The difference between the abstract and concrete views can be difficult to characterize in practice, probably because it has a gradual nature. In any case, it has nothing to do, in principle, with the difference in the represented reality (domain vs. system) or the difference in the purpose of the model (description vs. specification). Most of the efforts in MDE tools research have been devoted to the shifting along the third dimension, since automation seems to be a promising fruit at hand in this field.

It is a common opinion that analysis precedes design in the software development process. But this can happen in two different ways that depend on how we interpret an analysis model: a) in the "classical" sense, the analysis model is the result of a reverse engineering process (model-as-copy), where it represents the "real world"; b) in the "software engineering" sense, the analysis model is an anticipation of a forward engineering process (model-as-original), where it represents a conceptual view of the system. These two views are radically different in their relationship to design models, and should not be confused. How can we judge the correctness of a model if we do not know what it is supposed to mean, what for, and how?

Acknowledgements

This research is supported through the Spanish Ministerio de Ciencia y Tecnología, Project TIN2004-07083, "GPS: Plataforma de Gestión de Procesos Software: modelado, reutilización y medición".

REFERENCES

[Atkinson & Kühne 05] C. Atkinson, T. Kühne. "A Generalized Notion of Platforms for Model-Driven Development". In S. Beydeda, M. Book, V. Gruhn (Eds.), *Model-Driven Software Development*, pp. 119–136, Springer Verlag, 2005.

[Bézivin 05] J. Bézivin. "On the Unification Power of Models". *Software and Systems Modeling* 4(2):171–188, May 2005.

[Bittner 03] K. Bittner, I. Spence. *Use Case Modeling.* Addison-Wesley, 2003.

[Braude 01] E. Braude. *Software Engineering. An Object-Oriented Perspective.* John Wiley & Sons, 2001.

[Brown 04] A. W. Brown. "Model Driven Architecture: Principles and Practice". *Software and Systems Modeling* 3(4): 314-327, December 2004.

[Clark et al. 08] A. Clark, P. Sammut, J. Willans. *Applied Metamodelling. A Foundation for Language Driven Development*. Ceteva, 2008 (http://www.ceteva.com/).

[Coad & Yourdon 91] P. Coad, E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, 1991.

[Cook & Daniels 94] S. Cook, J. Daniels. *Designing Object Systems: Object-Oriented Modeling with Syntropy*. Prentice-Hall, 1994.

[ESA 95] European Space Agency. Board for Software Standardisation and Control. *Guide to the Software Requirements Definition Phase*. PSS-05-03, March 1995.

[Fowler & Scott 04] M. Fowler, K. Scott. *UML Distilled.* Addison-Wesley, 2004.

[Génova et al. 05] G. Génova, M. C. Valiente, J. Nubiola. "A Semiotic Approach to UML Models". *First International Workshop on Philosophical Foundations of Information Systems Engineering-PHISE 2005*, 13 June 2005, Porto, Portugal. *Proceedings of the CAiSE'05 Workshops*, vol. 2, pp. 547-557.

[Génova & Llorens 05] G. Génova, J. Llorens. "The Emperor's New Use Case", *Journal of Object Technology*, 4(6):81-94, Aug 2005 (http://www.jot.fm/issues/issue_2005_08/article7).

[Graham et al. 98] I. Graham, B. Henderson-Sellers, H. Younessi. *The OPEN Process Specification.* Addison-Wesley, 1998.

[Harel & Rumpe 04] D. Harel, B. Rumpe. "Meaningful Modeling: What's the Semantics of 'Semantics'?". *IEEE Computer*, October 2004:64-72.

[Hay 99] D. Hay. "There Is No Object-Oriented Analysis". *Data to Knowledge Newsletter*, 27(1), January-February 1999.

[Haythorn 94] W. Haythorn. "What Is Object-Oriented Design?". *Journal of Object-Oriented Programming* 7(1):67-78, 1994.

[Høydalsvik & Sindre 93] G. M. Høydalsvik, G. Sindre. "On the Purpose of Object-Oriented Analysis". *VIII Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-93)*, September 26 - October 1 1993, Washington, DC, USA. *ACM SIGPLAN Notices* 28(10):240-255.

[Jackson 95] M. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.

[Jacobson 95] I. Jacobson. "A Confused World of OOA and OOD". *Journal of Object-Oriented Programming* 8(5):15-20, 1995.

[Kaindl 99] H. Kaindl. "Difficulties in the Transition from OO Analysis to Design". *IEEE Software*, 16(5):94-102, 1999.

[Karow & Gehlert 06] M. Karow, A. Gehlert. "On the Transition from Computation Independent to Platform Independent Models". In *Proceedings of the Twelfth Americas Conference on Information Systems*, August 4-6, 2006, Acapulco, Mexico.

[Kent 02] S. Kent. Model Driven Engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods–IFM 2002*, May 15-17, 2002, Turku, Finland, pp. 286-298.

[Kleppe et al. 03] A. Kleppe, J. Warmer, W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[Marcos & Marcos 01] E. Marcos, A. Marcos. "A Philosophical Approach to the Concept of Data Model: Is a Data Model, in Fact, a Model?". *Information Systems Frontiers*, 3(2):267-274, 2001.

[Mellor et al. 04] S. J. Mellor, K. Scott, A. Uhl, D. Weise. *MDA Distilled. Principles of Model-Driven Architecture*. Addison-Wesley, 2004.

[OMG 01] Object Management Group, Architecture Board ORMSC. *Model Driven Architecture (MDA)*, 2001 (http://www.omg.org/docs/ormsc/01-07-01.pdf).

[OMG 03] Object Management Group. *MDA Guide version 1.0.1*, 2003 (http://www.omg.org/docs/omg/03-06-01.pdf).

[Parnas 72] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". *Communications of the ACM*, 15(12):1053-1058, December 1972.

[Potts 06] C. Potts. "Modes of Correspondence between Information System and World". *Second International Workshop on Philosophical Foundations of Information Systems Engineering-PHISE 2006*, 5 June 2006, Luxembourg. *Proceedings of the CAiSE'06 Workshops*, pp. 745-756.

[Rumbaugh et al. 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Seidewitz 03] E. Seidewitz. "What Models Mean". *IEEE Software* 20(5):26-32, Sep-Oct 2003.

[Sendall & Kozaczynski 03] S. Sendall, W. Kozaczynski. "Model Transformation: The Heart and Soul of Model-Driven Software Development". *IEEE Software* 20(5):42-45, September-October 2003.

About the authors

Gonzalo Génova received in 2003 his PhD in Computer Science at the Universidad Carlos III de Madrid, Spain, where he is currently an Associate Professor of Software Engineering. His main research subject is modeling and modeling languages in model-driven software engineering, as well as requirements engineering. He can be reached at ggenova@inf.uc3m.es.

María C. Valiente is currently a lecturer in the Department of Informatics and a PhD student of Computer Science at the Universidad Carlos III de Madrid. Her research and teaching interests include software process improvement and reuse, software modeling and model transformations (MDE, MDA, UML, MOF, etc.), workflow management and Petri nets. She can be reached at mcvalien@inf.uc3m.es.

Mónica Marrero is currently a teaching assistant of Software Engineering and Information Engineering at the Universidad Carlos III de Madrid. She is studying a PhD in Computer Science inside the Knowledge Reuse Group, and her central areas of research are: Information Retrieval, Entity Extraction, Pattern Recognition and Text Mining, but she is also interested in Software Modeling and Reuse. She can be reached at mmarrero@inf.uc3m.es.