

A Semiotic Approach to UML Models

Gonzalo Génova⁽¹⁾, María C. Valiente⁽¹⁾, Jaime Nubiola⁽²⁾

⁽¹⁾ Department of Informatics, Carlos III University of Madrid
Avda. Universidad 30, 28911 Leganés (Madrid), Spain
{ggenova, mcvalien}@inf.uc3m.es

⁽²⁾ Department of Philosophy, University of Navarra
31080 Pamplona, Spain
jnubiola@unav.es

Abstract. In this paper we are trying to clarify, with the aid of some semiotic notions, 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 when doing modeling, and sometimes lead to bad engineering practices. 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 models (in fact, a first step in design where the external view of the software system is specified) or building “real world” models. The main danger of confusing both kinds of models would be building a model of the real world and then using it as a specification of the software system, producing a system that needlessly matches the structure of the real world.

Introduction

Since the beginning of the information and computer systems era, knowledge management was soon identified as one of the key problems in the software production process [2]. Communication problems among different communities (users, owners, analysts, architects, designers, programmers...), and even within communities, frequently led to failing projects. The need of a sound documentation containing appropriate models was therefore soon acknowledged, but not consistently followed by everybody. Many disdained documentation and believed that documenting was a boring task to be done when the project is nearly finished, appropriate only for low-level workers in the organization, instead of realizing that documenting and modeling is perhaps one of the most difficult tasks within the software development process, that requires very high-level skills, and that has to accompany the whole process since the beginning. Of course, this led to boring, bad written and useless documents and models, which self-nurtured this erroneous conception.

Fortunately, things have been changing over years, and today the crucial role of models in the software development process is rarely contradicted. The MDA initiative (Model Driven Architecture) [20], which is a consequence of the forces

released by the UML (Unified Modeling Language) [21], seems to be the last episode in this story. The central message of MDA 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 [15, 18]. Supposedly, 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: analysis and design models, structural and behavioral models, etc., so that understanding the meaning of each kind of model, how they are related, and how they evolve, is of principal importance. The OMG documents defining the UML, and many scientific works in the literature, go into great detail about the UML *notation* (the syntax), but they provide only informal discussions, if at all, on the *meaning* of the language (the semantics) [9]. Our goal in this work aims specifically towards investigating the different *way of meaning* of analysis and design models. Roughly speaking, “analysis” designates some kind of understanding of a *problem* or situation, whilst “design” is related to the creation of a *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”) [24, 1]. But let’s take these terms a bit more seriously.

Analysis is a Greek word that equates 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”. Therefore, analysis and design are not properly opposed or complementary. The complement of analysis is synthesis, not design; nevertheless, since design is the first step in synthesis, analysis and design are often treated as complementary.

In Software Engineering, a *model* is an abstraction, i.e. a simplification of reality. A model simplifies the reality it represents by suppressing irrelevant details and retaining the essential aspects. A model *represents* a certain segment of reality, i.e. it is a *sign* or *signifies* that reality. Since Semiotics is the science of signs and signification (from the Greek *semeion*, sign), we will look into some notions of Semiotics in the following Section, in order to better understand *what a model is*.

1. Semiotics of diagrams and models

Semiotics is a modern discipline, co-founded by Charles S. Peirce (1839-1914) in North America and Ferdinand de Saussure (1857-1913) in Europe. According to Peirce’s definition, “a sign is something which stands to somebody for something in some respect or capacity” [23]. Note the triadic nature of signs: “a sign is something which stands *to somebody* for something”. The signification relationship requires always three elements to take place: the *sign* (or signifier), the *object* (or signified), and a third element called by Peirce the *interpretant*. This triadic relation (Figure 1) associates to the sign S, an object O by substitution (“stands for”) on one hand, and on the other hand, the S-O link is itself associated to the interpretant I (“stands to”). In

other words, there is no automatic connection from significant to signified: it is the interpretant who connects the sign to its object. This has, at least, two important implications. First, a sign always requires interpretation, there is no absolutely fixed connection between the sign and its object. Second, a sign is meaningful only for an intelligent being that has to interpret it.

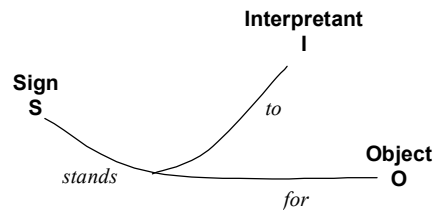



Figure 1. The triadic signification relationship according to Peirce

Always following Peirce, a sign stands for an object in *three non-exclusive ways*, respectively the icon, the index and the symbol [23] (see Table 1).

- **icon:** the object is signified by some kind of *resemblance*;
- **index:** the object is signified by some kind of physical or causal *connection*;
- **symbol:** the object is signified by pure convention or *law*.

Table 1. The three kinds of sign according to Peirce

| Sign | Mode | Sign example | Object example |
|--------|-------------|---|------------------|
| icon | resemblance |  | person |
| index | connection | smoke | fire |
| symbol | law | + | adding operation |

No need to say, the objects that can be signified by signs are not restricted to physical things. A sign can represent everything: a thing, an action, a concept... In particular, a sign can represent another sign, and thus transitively the object of this second sign. For example, a written word is a sign of a spoken word, which in turn is a sign of a concept, which can be itself a sign of something else. This is important for diagrams, which frequently contain signs of natural language words.

Rather than three completely separated kinds of signs, Peirce considers that every sign participates more or less of the nature of icon, index and symbol, even though one of the three aspects can be stressed in a particular sign. For example, traffic signals, whose meaning is stated by convention or law (symbol), are selected so that they have some kind of iconic resemblance with the action they forbid or mandate; a person's picture for an ID card resembles that person (icon), of course, but it is also causally connected with that person through the physical process of photography (index), and, even more, it is a social convention that a face's picture represents the whole person (symbol).

From this semiotic point of view, then, we can consider a UML diagram as *a sign made of signs*, and we can observe this articulation of modes of signification in it

[19]. For example, in a class diagram (Figure 2), which is made out of different kinds of graphical signs, the use of a box to represent a class of objects is purely conventional (symbol); the features of a class are arranged in two lots, to indicate (index) their different nature of attributes or operations; and the names of the classes, attributes and operations are chosen so that they resemble (icon) the natural names of those concepts.

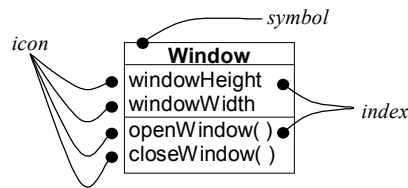


Figure 2. Icon, index and symbol in a class diagram

A UML model, however, is not merely a diagram, or a collection of diagrams. A UML diagram is the graphical representation of a logical set of interconnected elements belonging to a model [1, 21], i.e. *a diagram is a partial view of a model*. In other words, a UML model can be graphically represented by means of UML diagrams following the rules of the UML notation, but it could be graphically represented also in a tree structure (as it is usually found in the typical left panel in CASE tools), or else in a purely textual form (reports generated by tools, XMI serialization, etc.): that is, UML models are independent of UML notation.

A UML model, then, is *a logical set of elements* that represent something, are defined according to the abstract syntax of the language (the metamodel), and can be represented according to its concrete syntax (the notation). The model as a whole signifies a certain reality, and each element signifies one small part of that reality.

In summary, then, a UML diagram is a sign of a UML model; the reality signified by the diagram is that part of the model it graphically represents. The next obvious question is, *what is a UML model a sign of*, what is the reality signified by it, what is its meaning? We will try to answer this in the following Sections.

2. The meaning of a UML model: software models

There is a kind of models for which the answer to this question is rather easy. They are called “software models” because the object signified by the model is a software artifact: a piece of code, an executable component, an entire system or subsystem, etc. Let’s take a rather trivial example, where a piece of code written in the Java programming language is represented using the UML notation with different levels of detail (Figure 3). As we can see, the UML notation for a class is basically a simplified view of the class code, with the advantage that clumsy programming details can be omitted. This responds perfectly to the definition of a model as “a simplified view” or “abstract representation” of some segment of reality, “reality” being in this case the code itself.

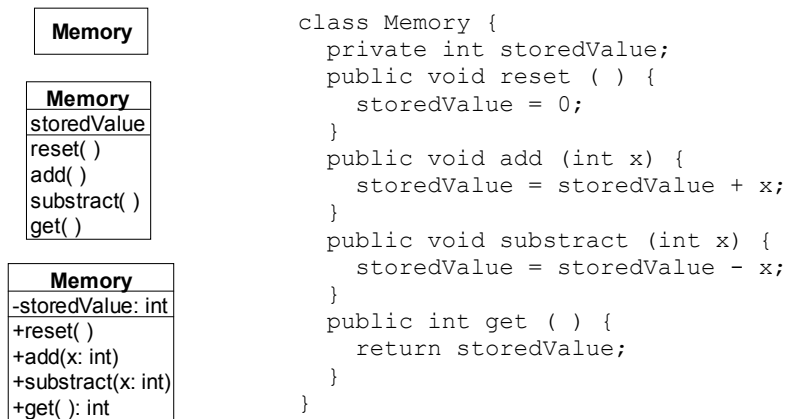


Figure 3. Abstract UML representations of Java code, with different levels of detail

A software artifact can be modeled at *different levels of abstraction*: the more abstract the model, the less amount of detail it represents, the more far-away it is from the signified artifact. For example, Figure 3 shows three different levels of detail to represent a class: the lowest box is the one with more details shown in it, the middle one omits types, operation parameters and visibilities, and the highest one omits even attributes and operations. The three representations are valid signs (abstractions) of the piece of code on the right. Moreover, the higher level representations can be used as signs of the lower level ones; we can regard them as three diverse representations of a single model with different levels of detail, or else as three different models which are ones abstractions of the others. In fact, the Java code itself is a sign or abstraction of a lower level software artifact, i.e. assembly code, which in turn is a sign or abstraction of the bits inside the memory of the computer, which in turn are signs of voltage levels and transitions in the circuits... the history of computing and software engineering can be summarized as a continuous effort to raise the abstraction level of information processing, and bring it near it to human understanding.

A modeling language such as UML can be more or less adequate to represent software artifacts created with a certain programming language, such as Java, C++, Visual Basic, Pascal, C, Ada, etc. In fact, modeling languages often originate in programming languages, so that, depending on the software paradigm, sometimes the matching is good, sometimes it is very bad. For example, UML is well fitted to represent Java or C++ (in general, object-oriented programming languages – OOPLs), but instead it could hardly produce a good abstraction of a program written in C or Ada, good archetypes of structured programming languages, which know nothing about classes, objects and messages. In other words, a modeling language is good at representing a programming language if they share a common set of basic concepts, which eases the translation between model and implementation, and backwards; otherwise the translation will be unnatural.

In addition to classical object-oriented programming languages, the origins of UML are linked to other sources, in particular data modeling techniques from the

field of databases. This explains why some UML constructs are so easy to implement in OOPs (class, attribute, operation, message...), whilst others are so difficult and require a very complex structure to rightly implement the intended semantics in UML [7]. This is the case, for example, of bidirectional associations and other kinds of association, safe from the simple one-way association, which is equivalent to an attribute. Summing up, UML can be very inadequate to represent an implementation language that comes from a paradigm other than OO, and a classical OOP can be very limited to implement those UML constructs which did not originate in object orientation.

Software models can be used mainly in two different ways, traditionally known as forward and reverse engineering. In *forward engineering*, the model is used as an anticipation 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, etc. In *reverse engineering*, the model is used as a conceptual tool to understand an existing system that has to be maintained or improved. 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 [17]: 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 another well-known pair we mentioned before: forward engineering is a process of *synthesis* in which the system is *constructed* starting from a model, whilst reverse engineering is a process of *analysis* where the system is *understood* by means of a model (see Table 2).

Table 2. Two different usages of software models in software engineering: forward engineering uses a model-as-original in a synthesis process of a new system; reverse engineering produces a model-as-copy in the analysis process of an existing system

| Model → System | System → Model |
|---------------------|---------------------|
| forward engineering | reverse engineering |
| model-as-original | model-as-copy |
| synthesis | analysis |

However, and maybe paradoxically, this perfectly legitimate sense of the term “analysis”, very close to the original meaning of the word, is not the most usual among software engineers, as we will show in the next Section.

3. Analysis models as software models

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 models (forward engineering) or building “real world” models (reverse engineering). We explore these two senses of the word in this Section and the following.

It is very common to characterize analysis as specifying the *what*, whilst design as specifying the *how*: what is the system supposed to do for its users, how will it do it (in fact, this expresses a classical principle in software engineering: separation of the specification from the implementation). The *analysis model* must then *capture user requirements* without prematurely adopting implementation decisions, i.e. omitting technology-dependent details [14], and using concepts solely drawn from the problem domain. In contrast, the *design model* must *define a software solution* that effectively and efficiently satisfies the requirements specified in analysis, and in doing this the model will incorporate new artifacts (new classes, new attributes and operations for the classes, etc.) and will take into account the concrete technological platform on which the software system is to be built. As it can be easily observed, this distinction between analysis and design is parallel to the one existing in MDA between PIM (Platform Independent Model) and PSM (Platform Specific Model) [15, 18].

The essential point here is that both kinds of models, analysis and design, or PIM and PSM, do represent the same system, *both are software models* used in the context of a forward engineering process, even though they do have an important difference in purpose and perspective [12], and thus in *the way they signify* the system-to-be-built. Let's recall that "a sign is something which stands to somebody for something in some respect or capacity" [23]. This *respect or capacity* is different in each case: the analysis model represents the *external view* of the system (the specification), whilst the design model represents the *internal view* (the implementation). Each one exercises a *different kind of abstraction* that produces a different kind of model of the same system, so that it is signified in a different way, too: the design omits more or less implementation details, whilst the analysis omits the implementation itself; the analysis specifies what the design realizes. The visible effect can be a *different level of complexity* in the models, but the essential point is not that, but a *different purpose*. (Therefore, we are not pretending that the analysis is merely a first draft of the design: this would miss the point. Notice, too, that the common expression used to distinguish them, "different level of abstraction", even though not essentially wrong, can be misleading.) This notion of analysis, in fact *a first step in synthesis*, can be easily recognized in a multitude of engineering practices and text books, even though it will be rarely recognized explicitly (there are exceptions, of course [13]).

The *use case model* is a good example of this. Due to its "high level" character, closer to the user than to the implementation, use cases are usually defined within the "analysis" phase 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 [8]: 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 [10]. Consequently, any modeling activity that is derived from the use case model will be referred to the same system-to-be-built, and therefore must be considered as producing a software model, too: for example, a structural model of the main concepts present in the use case model (often known as *conceptual model*), such as data items exchanged between the user and the system, or stored inside the system, etc. The same applies to *user requirements* in general (expressed through use cases or not): they are specifications of a desired computer information system, they do not describe the world outside of the computer. In fact, user requirements, where the textual form usually predominates, can be properly

considered as a sign, i.e. a model, of the required system (a *textual model* indeed, who said a model has to be graphical?). Since many conceive analysis as an activity where user requirements are elucidated and clearly written [5, 14], it is clear that they are regarding analysis as part of a forward engineering process, i.e. synthesis.

In this view, the *transition* from the analysis model to the design model can be hard or not, but offers no conceptual difficulty from the semiotic point of view: both are models of the same software system, yet from a different perspective and with a different purpose. Our point here is not that the transition is easy or difficult. In fact, the design has to provide a creative solution for the problem specified in the analysis, and this rarely will be easy [12, 14]. Moreover, a good design model that takes into account non-functional requirements such as performance, reuse, maintainability, etc., might produce a system that hardly resembles the system specified in the analysis model [11, 12, 22]. But this does not negate that both models are signs of the same software system, which is our point: both analysis and design models are software models. Analysis and design models do not have to resemble, but they signify the same system. However, if analysis is understood as in the next Section, a new conceptual problem arises regarding the transition between analysis and design.

4. Analysis models as domain models

The other common way to explain the use of an analysis model in a software project is to understand it as a model of the domain, or context, of the problem to be solved: a model of the “real world” outside of the software system [4, 16, 24]. This is sometimes referred to as *business model*, too, or Computation Independent Model (CIM) in MDA. The difference between analysis and design is not anymore a different kind of abstraction exercised, but *a different reality signified*: analysis classes represent real world concepts, whilst design classes represent pieces of code [6, 12, 14]. The level of abstraction can be high or low in both cases.

This notion of analysis model (a model-as-copy of the domain or business, produced through reverse engineering), much closer to the original meaning of the word, poses nevertheless several difficulties. First of all, the fact that we use *a uniform notation* to represent *very different kinds of things* may be, and in fact it is very often, misleading [14]. The most common error occurs when the analysis model is supposed to represent the real world (domain model), but then it is used as a specification of the system to be constructed (software model). Even though some methods do recommend starting with a model of the “real world”, and then include this model within a design model that is completed with other artifacts as required to provide the solution [4, 24], this cannot be a valid general rule. The software system may include a model of the external world to *simulate* it, but the solution that the system is expected to provide is probably far beyond a simulation of the structure or behavior of the external world, or even might not require it at all. The objects inside the system that provide an adequate solution to the problem may or may not be copies of the objects in the external world. Therefore, since the system is not generally a copy or simulation of the world outside, *a reverse-engineering model of the world cannot serve as a forward-engineering model of the system*.

In other words, the analysis of the “real world” where the software system does not yet exist cannot be used to model what the system must do in the future. To do this, we really do not need a model of the real world, but a model of the required system, i.e. a software model, as explained in the preceding Section. In fact, a specification of user requirements for the system should not be regarded as a model of the real world, but as a model of the system to be build [10]. Moreover, a model of the envisaged real world *after* the new system already exists, as opposed to a model of the real world *before* the new system exists, does not serve either as a model of what the system must do: it is always a model of the world, not a model of the system. 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.

Nevertheless, it cannot be denied that *understanding the real world*, i.e. analyzing it (in the original 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. In fact, we claimed in the preceding Section that the analysis-as-software-model should use the concepts found in the analysis-as-domain-model. The difference is subtle, but real: using the same vocabulary, these two kinds of models represent two different things. Maybe this is the source of the confusion we try to avoid. Making a model-as-copy of the real world to 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. Now then, is UML an adequate language to produce a model of the real world? Even more, is object orientation a good paradigm at all to understand the real world? What is the “real world”, in fact?

A very naïve view is transmitted by many books on object orientation: the real world is made up of the physical things we can see or touch, such as persons, books, planes, etc.; the real world is made up of objects, therefore object orientation is the most natural way to understand the world, and to develop software systems to solve human needs [24]. Maybe the roots of object orientation can be traced back to the beginning of Western philosophy and culture. An example is Aristotelian philosophy, where the central notion is that of “entity”. But the similarities end here: the archetypical *entity* in Aristotle’s philosophy is the *living being*, which is essentially “one”, i.e. it possess a strong unity in itself that cannot be decomposed into parts without killing it. Instead, the notion of *object* in software is inspired rather on the *mechanical device*, an assembly of parts fruit of human artwork. For example, in the object-oriented world, operations are performed *on* the objects, whilst in the Aristotelian world, operations are performed *by* the entities, what makes the translation between the two worlds rather difficult and denies the claimed straightforwardness of object-orientation. Moreover, there are other competing philosophical views of the world, where the central notion is not the object or entity, but rather the process (see for example the philosophy of Alfred North Whitehead).

This is not the place to attempt an answer to the question, what is the true structure of the world? But we don’t need to: if we only want to build information systems, all we need to understand is *the world as we speak about it*, the information world. Therefore, instead of the expression *real world*, it seems much more adequate the less compromising *universe of discourse* (UoD), already used in other branches of computer science and information technology.

Object orientation is a particular way of conceiving and constructing software systems. Is it adequate to model the universe of discourse, the world as we speak about it? Well, there is probably no harm in it, as long as we are conscious of the *limitations of the method*, and of the modeling language in particular. The danger appears when it is acritically accepted that “everything is an object”; in fact, rather than expressing a property of the “real world”, this expresses a property (in this case, a limitation) of the modeler. It is known that, for a person who holds a hammer, everything is a nail...

We have already mentioned the inadequacy of UML to model software artifacts designed with non-OO programming languages, such as C or Ada, because they do not share a set of basic concepts. What happens if the concepts in the UoD are naturally conceived in a non-OO way? Well, object orientation, in this case, could enrich our understanding of the UoD, but it can impoverish it, too. In fact, when one tries to “objectify” a non-OO world, one is really creating a new world: *an object oriented universe of discourse*. An inadequate language ontology [3] can explain some of the limitations perceived in UML when it is used specifically for modeling domain objects instead of software objects. It can be more or less natural for the typical Western mind to identify objects, and classes of objects, in the UoD; even attributes are usually not difficult to identify; however, the concept of “message” as operation invocation, which is central to OO, is much more unnatural in many domains. A behavior model based on message interchanges, with parameters, returned values, polymorphic invocations, and so on, can lead to really twisted models that complicate, rather than simplify, the comprehension of the domain.

Conclusions

We have examined *two different meanings of analysis models* that can be found among software engineers: a model that specifies the external view of a software system, or else a model of the “real world” that constitutes the context of the desired software system. Both views are very often confused. Actual practice corresponds usually to analysis-as-software-model, even though the explanation goes frequently for analysis-as-domain-model. A *moderate danger* of the confusion is to believe that one is modeling the real world, when one is really doing a specification or a high-level design of the software system. A much more *serious danger* is to build a system that needlessly matches the structure of the real world; this can be worsened if one uses a modeling language that forces the natural way of speaking about the domain, so that the produced model is much more complicated than necessary, or misses essential details that cannot be properly expressed in the language used.

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. Hopefully the new terms coined in the Model Driven Architecture initiative, i.e. Computation Independent Model, Platform Independent Model and Platform Specific Model, will help to avoid confusions, and reserve the original meaning for the word “analysis”.

References

1. G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
2. F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.
3. J. M. Cañete, F. J. Galán, M. Toro. "Some Problems of Current Modelling Languages that Obstruct to Obtain Models as Instruments". *Jornadas de Ingeniería del Software y Bases de Datos*, 10-12 November 2004, Málaga, Spain, pp. 13-24.
4. P. Coad, E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, 1991.
5. European Space Agency. Board for Software Standardisation and Control. *Guide to the Software Requirements Definition Phase*. PSS-05-03, March 1995.
6. M. Fowler, K. Scott. *UML Distilled*. Addison-Wesley, 2004.
7. G. Génova, C. Ruiz del Castillo, J. Llorens. "Mapping UML Associations into Java Code", *Journal of Object Technology*, 2(5):135-162, September-October 2003, (http://www.jot.fm/issues/issue_2003_09/article4).
8. G. Génova, J. Llorens, J. Nubiola. "Métodos abductivos en ingeniería del software". *2º Workshop en Métodos de Investigación y Fundamentos Filosóficos en Ingeniería del Software y Sistemas de Información-MIFISIS'04*. November 5-6, 2004, Valladolid, Spain.
9. D. Harel, B. Rumpe. "Meaningful Modeling: What's the Semantics of 'Semantics'?". *IEEE Computer*, October 2004:64-72.
10. D. Hay. "There Is No Object-Oriented Analysis". *Data to Knowledge Newsletter*, 27(1), January-February 1999.
11. W. Haythorn. "What Is Object-Oriented Design?". *Journal of Object-Oriented Programming* 7(1):67-78, 1994.
12. G. M. Høydalsvik, G. Sindre. "On the Purpose of Object-Oriented Analysis". *VIII Conf. on Object-Oriented Prog., Syst., Lang., and Appl. (OOPSLA-93)*, September 26 – October 1 1993, Washington, DC, USA. *ACM SIGPLAN Notices* 28(10):240-255.
13. I. Jacobson. "A Confused World of OOA and OOD". *Journal of Object-Oriented Programming* 8(5):15-20, 1995.
14. H. Kaindl. "Difficulties in the Transition from OO Analysis to Design". *IEEE Software*, 16(5):94-102, 1999.
15. A. Kleppe, J. Warmer, W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.
16. C. Larman. *Applying UML and Patterns*. Prentice-Hall, 1998.
17. 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.
18. S. J. Mellor, K. Scott, A. Uhl, D. Weise. *MDA Distilled. Principles of Model-Driven Architecture*. Addison-Wesley, 2004.
19. B. Morand. "Modeling: Is it Turning Informal into Formal?". *The First International Workshop on the Unified Modeling Language*, June 3-4, 1998, Mulhouse, France, Springer LNCS 1618, pp. 37-48.
20. Object Management Group. *MDA Guide Version 1.0.1*. (<http://www.omg.org/>).
21. Object Management Group. *Unified Modeling Language Specification*. Version 1.5, March 2003 (<http://www.omg.org/>).
22. D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". *Communications of the ACM*, 15(12):1053-1058, December 1972.
23. C. S. Peirce. "Ground, Object and Interpretant" (*CP* 2.228, 1897), and "A Second Trichotomy of Signs" (*CP* 2.247-249, 1903). In C. Hartshorne, P. Weiss y A. W. Burks (eds.), *The Collected Papers of Charles Sanders Peirce*, vols. 1-8. Harvard University Press, Cambridge (Massachusetts), 1931-1958.
24. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.