# A Genesis of Thinking in the Evolution of Ancient Philosophy and Modern Software Development

Stephan H. Sneed

jCOM1, Münchner Straße 29 - Hettenshausen
85276 Pfaffenhofen, Germany
stephan.sneed@metasonic.de

**Abstract.** This paper is a brief discussion on the issue of modeling evolution. The question posed is where does modeling theory come from and where is it going. A parallel is drawn between the evolution of modeling theory and ancient philosophy. In the end both must come to the conclusion that a theory must be applicable to a human problem and must lead to a solution of that problem. Otherwise, it is useless. It is pointed out, that some methodological approaches are even detrimental to reaching a good solution. They only absorb costs and effort and lead to nowhere. Just as Aristotle rounded out the ancient philosophical discussion, the S-BPM method appears as the next logical step in the evolution of modeling methods. In the end, some research issues are discussed, for the S-BPM method as well as for modeling comparison in general.

**Keywords:** SA, OOA, OOP, S-BPM, paradigm change, modeling methods, programming languages, software development, evolution.

## 1 Introduction

Before introducing a next step in this evolution, one needs to answer some questions: How do we even determine the value of a modeling or programming method? Is there a significant difference between programming and modeling languages? If modeling languages are only meant to describe a system, they can differ from programming languages, but if they are intended to be executed they take on all the characteristics of a programming language. In the end both are formal, artificial languages. If so, then all the experience made with programming languages must apply as well to the evolution of modeling languages. By looking at the evolution of human thinking in ancient philosophy from Parmenides to Plato via Socrates and on to Aristotle, we will see that the evolution of human thought has much in common with the evolution of software development methodologies. We will also see that Aristotle is even one step ahead compared to the IT with Object technology as the last major evolution step. Philosophy in general is about constructing descriptive models of the real world that are easier to comprehend than the real world itself. This is what philosophy has in common with design theories aimed at achieving higher level abstractions of machine code in a graphical notation. They are easier to comprehend than the machine code which is finally executed, which was the main motivation to develop them.

C. Stephanidis (Ed.): Universal Access in HCI, Part I, HCII 2011, LNCS 6765, pp. 129–138, 2011.

## 2 Formal Theories in IT

Let us define executable machine instructions as the base of any abstraction with index 0 and therefore as atomic elements of higher level abstractions. Any higher level abstraction of machine code is a formal theory above this base with an index n. Any model consisting of more than one atomic algorithm of such a formal theory is called a system. The problem with these definitions is that they are recursive and not bounded. Two instructions can already form a system. A function is already a system as well as anything that is not atomic like a single instruction or a single data item. Hence, referring to something as a system consisting of functions that consist of instructions is absolute arbitrary. The fact is that we have systems encapsulated by other systems and so on. There can be an infinite nesting of systems. So each programming language is an abstraction with an index n of the atomic instructions at 0 level, and hence a formal IT theory. Each executable program written in this language is a model of this theory and can be interpreted by the computer.

In case of modeling methods, one has to distinguish between the ones that are executable and ones that are not. Some modeling languages like SADT were never intended to be executable. Originally UML was not intended to be executed. Now it is. Anyway, executable modeling languages (or their executable subsets) are formal IT theories about the code they govern. A model of such a theory can be executed, hence is an abstraction of machine code and hence an IT system. This brings us to an interesting point: What is actually the difference between a programing language and an executable modeling method? They have the following aspects in common:

- Programing languages and modeling methods are both level n abstractions of machine code
- Models of programing languages and models of modeling methods are called IT systems.

The aspects where programming languages and modeling methods differ are the following:

- The algorithms in programming languages have a textual representation whereas the representation of algorithms in an executable modeling method is graphical. Actually most modeling languages reach a point where graphics no longer suffice. Then they revert to text in tags like OCL as an extension of UML.
- Usually the modeling method is an abstraction of a certain programming language and has the abstraction index n+1 when the programming language is of abstraction index n.

It becomes clear that the type of algorithm representation, either in text or in graphical representation, is not crucial for the theory itself. We could well think of a graphical Java Editor where an “if statement” could be placed in a diamond shaped form as with structured analysis. Thus, the representation of algorithms is actually not relevant since it can be replaced by other representations. The other aspect is the degree of the abstraction level. A modeling method will usually have a higher

abstraction index than a programming language but this aspect does not have a casual relationship. We could also think of a modeling method that is directly based on a particular machine code as well. In the end, there is no real difference between programming languages and executable modeling methods. They are both nothing but abstractions of a certain level n of machine code. Both are being developed in order to reduce complexity and to make the systems described with them easier to comprehend. Thus both of them have to be treated equally when talking about their value to the user.

## 3 Formal IT Theories and their Value

Value can be expressed in terms of cost effectiveness, reduced time and enhanced comprehension. What does cost effectiveness mean in the sense of formal IT theories? In the first place, any business IT system has the purpose to of enhancing user revenues. So to be of value, a language or model must be cost effective, i.e. inexpensive to use. A language which requires a longer time to formulate is less valuable than one which fits better to the problem and can be expressed more quickly. General purpose languages are generally not suitable to particular problems and require more time to formulate. This is the advantage of domain specific languages.

The third potential value is more problematic. What does it mean to be comprehensible? Are there some certain properties that make a formal language easier to comprehend than others? Are they objective? Can they even be? Nietzsche says that every product of the human mind is only relevant for humans. So, according to him, there are no general criteria for good and bad languages, and he shows good reasons for his claim [Nietzsche1999]. Languages can only be judged in terms of their value to the user. If so, the language would be preferable which requires the least mappings between the human user and the IT system.

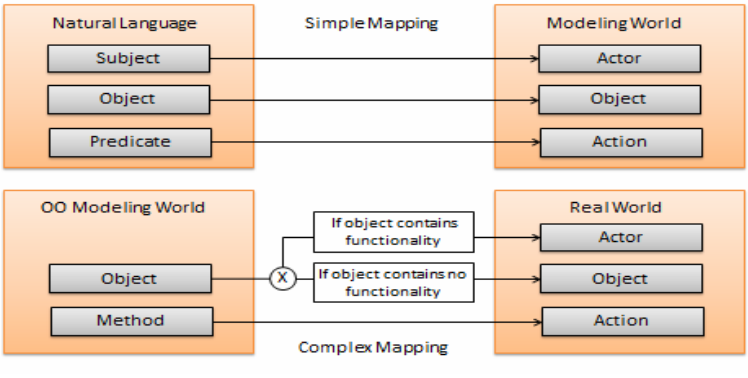


**Fig. 1.** The figure shows a simple and a complex mapping. The simple mapping occurs where real world (or natural language) and modeling world entities are the same, while the complex mapping occurs, where they differ.

## 4 A Genesis of Thinking

It is, of course not possible to discuss the complete evolution of Greek philosophy and IT programming and modeling methods within this short paper. But there is also no need for this. It suffices to concentrate on those key aspects that determined a new evolutionary step. The evolution of model construction seems to be common for both ancient philosophy and modern system design. The requirements for proper models, the arguments for and against certain approaches, as well as the criticism and condemnation are following exactly the same track. This evolution, "based on the problem and dictum of Parmenides describes [in both disciplines] a curios curve of thinking" [Buchheim2003] that will be described in this chapter.

**Parmenides and Bottom-up System Design.** The beginning of software design methodology coincides with the point in time in which the ancient Greeks started to reflect on the nature of the world and the reason for their being. Heraclitus, a contemporary of Parmenides, pointed out the difficulties one has when discussing an ever changing and evolving world, while Parmenides reflected on the construction of models as well: "The one most famous claim made by Parmenides that can serve for constructing any ontology" [Buchheim2003] is the following: "*What is thought must correspond to what exists*" [Parmenides, Fragments, B3]. This claim of Parmenides presupposes a tight relationship between the intelligible and the existing entities. The fulfillment of this axiom is what we will be pursuing when comparing the evolution of ancient philosophy with that of system modeling. Regarding ancient philosophy we have to rely on the conclusions reached at that time, but as for system design, i.e. business engineering, we can determine whether a model fulfills the claim of Parmenides or not: The claim of Parmenides is fulfilled by a modeling approach in IT when the respective model can be automatically transferred into coding instructions performing specific functions with interfaces to the outer world, that is, to an executable system. This is true only if the model is an abstraction of the underlying machine code.

In software design, the first approaches to dealing with program complexity were the hierarchical functional structured approach of IBM in the USA (HIPO) and the hierarchical data structured approaches of Jean Dominique Warnier in France (LCP) [Warnier1974] and Michael Jackson in England (JSP) [Jackson1983]. The situation with software development is according to Warnier something he calls the space and time problem where one images how a program is being executed in time and writes down the instructions to be executed in space [Warnier1974, p. 15]. He claimed that "this approach is not satisfactory, for it leads to us to attach equal importance both to details and to essential points; it must be replaced by a hierarchical approach which allows us to go from abstractions to details, i.e. from the general to the particular" [Warnier1974, p. 15]. This lead to a structured approach, in which data is grouped according to some criteria in a hierarchy of data types where each node of the hierarchy can be a set of data or an elementary data attribute. This data hierarchy was then mapped into a corresponding hierarchy of functions so that for every node of the data tree there was a corresponding node in the function tree for processing it. There are in the end as many functional hierarchies as there are input/output data structures.

Each functional hierarchy corresponds to a subprogram. A program consists of all subprograms derived from data structures used by that program.

This hierarchical data-oriented approach introduced by Warnier was a means of modeling IT-Systems from the bottom up based on the data they use. He also proposed a set of diagrams for depicting the data structures and their corresponding functional structures. The major reason why his method was not widely accepted was the difficulty involved in drawing those diagrams. Michael Jackson in England came up with a similar approach, but his tree diagrams were much more intuitive and easier to draw. Therefore the Jackson method of modeling computer programs became much more widespread. Furthermore, the Jackson structured diagrams could be mapped 1:1 into a program design language from which COBOL programs were generated. Thus, there was a direct uninterrupted path from the model to an executable system. The prerequisite was of course that the model must be at the same semantic level as the program itself, so that in the end the model is the program. Every single data item and elementary operation is contained within the model, thus fulfilling the premise of Parmenides. However, what was the advantage of this method other than changing the textual representation into the graphic one?

**Socrates and Plato, Structured Analysis and Object Orientation.** With the increasing power of the computer, the size of the IT-Systems increased as well. By the end of the 70's the average commercial program size had surpassed 2000 statements rendering it impossible to apply a bottom up approach with the then existing tools. This led to the so called software crisis of the 70ies, a crisis which continues even after 30 years. "The major cause of this crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem" [Dijkstra1972]. Thus the situation for any IT-Manager could be well described by a human that gains for the wisdom of god when trying to understand all the algorithms (either graphical or textual) under his responsibility. But Socrates claimed: "*Human wisdom is worth little or nothing at all.*" [Socrates, Apology 23a]

So, according to Socrates, it is not possible at all for the human to gain the complete and definite truth. The only thing that can be done is trying to get close to it by human assumptions and the testing of hypotheses, not knowing where this may lead to in the end. This is exactly the thesis put forward by the advocates of agile development. They claim it is impossible to model and to plan a complex IT-System in advance. The only way to construct one is through trial and error. "While the Pre-Socratic philosophy had been about achieving the truth by reducing concepts to existing reality, Socrates was looking for the most accurate alphabetization of the claimable, which was directed towards searching for the truth, but without ever really obtaining it" [Buchheim2003, p. 208]. This is equally true for agile development using Object-oriented Design. The analyst starts to study a given problem ("Analysis is the study of a problem…" [DeMarco1978, p. 4]) whose solution may, after many months of discussion with the user and the development team ("The analyst is the middleman between the user, who decides what has to be done, and the development team, which does it." [DeMarco1978, p. 7]), turn out to be completely different in the end than it might have appeared at the beginning. Socrates knew quite well that

human beings cannot grasp complex reality, as depicted in the quote: "*That, what we believe to understand clearly, we should look upon skeptically, for it is surely in error, even though we might all agree that we understand it*" [Socrates, Sophistes 242a].

This was the main motivation for the structured and later the object-oriented design methods. "The principal goal of Structured Analysis is to minimize the probability of critical analysis phase errors" [DeMarco1978, p. 9]. If errors are to be avoided within the analysis phase, they are avoided by constructing the proper models, gained by proposal and discussion. To quote the literature, "OO analysis and design […] are powerful ways to think about complex systems" [Martin1992, p. xi] Hence all analysis and its models, whether being structured or object-oriented, serve as a base of discussion to avoid errors within the analysis phase, thus corresponding to the philosophical approach of Socrates: "*The highest good for man is to produce [trains of thoughts] each day that you hear me talking about in discussions and thereby put myself and others in question*." [Buchheim2003, p. 210]

It was pointed out before that Socrates had his doubts about the Parmenides approach to depicting the reality in accordance with a single detailed view of the world. He was quoted as saying "*Human wisdom is insignificant compared to the true wisdom of god that is not at human disposal*" [Buchheim2003, p. 206]. The wisdom of god in terms of information technology would mean nothing less than knowing the content of each and every single instruction in a system, or its representation within a model. Creating models by abstracting elementary statements in the way that Warnier and Jackson did might work for small programs. However with the increasing power of the computer, the size of the IT-Systems increased as well.

But the entities of DeMarcos modeling approach are not physical entities like modules, interfaces and code blocks. They are abstract features defined in a specification made independent of the machine code. This meant it was never intended to derive executable programs from the model. DeMarco proposed to stop modeling this top down approach in a downward direction when one believes that the insights of his "lowest-level bubbles can be completely described in a mini-spec of about one page" [DeMarco1992, p. 84].

Object-oriented Analysis and Design is more of a bottom-up approach in the sense of Warnier and Jackson. It starts with the data objects of a system and determines what happens with them. Once the building blocks are determined, the designer then decides what to do with them. The object model is really derived from the data model as depicted by Shlaer and Mellor in their book on "Modeling the World with Data" [ShlaerMellor1988]. Unlike structured design, which was never intended to be used as a basis for code generation, object-oriented design can correspond to the physical system, i.e. to the code, provided it is broken down far enough. The UML-2 language makes this possible as claimed by Ivar Jacobson in his keynote speech at the International Conference on Software Maintenance in Florence in 2001 with the title "UML – all the way down" [Jacobson2001]. By that he meant that with the new UML-2 it would be possible to go all the way from the requirements to the code. There would only be one language required and that is UML.

The problem with this one language approach is that the one language inevitably becomes just another programming language. As Harry M. Sneed noted in his article "The Myth of Top-Down Software Development" by quoting DeMillo, Perles and

Lipton, "'the program itself is the only complete and accurate description of what the program will do.' The only way to make the specification a complete and accurate description of the program is to reduce the specification to the semantic level of the program. However, in doing this, the specification becomes semantically equivalent to the program" [Sneed1989, p. 26].

So, the OOA and OOD methods are nothing more than graphical representations of OO programming languages. The relevant principles of OO programming languages are inheritance and encapsulation (information hiding), locality of reference and polymorphism [Rumbaugh1991]. Each and every class extends the all integrating super class "Object". This corresponds exactly to Plato's teaching of the ideas. Individual ideas are all derived from a single overriding idea, the idea of the good, and each lower idea holds its part in the higher one. With his notion of what is good, Plato introduced something that he assumed to be "the organizing principle of the world" [Buchheim2003, p. 211]. The benefit of this concept is that information can be outsourced to the super classes (in a complex inheritance structure) in order to minimize the size of the lower level classes. But, of course, this requires a very intelligent design of the super classes, a degree of intelligence most software developers don't possess. "*Our constitution will be well governed if the enforcement agencies are always aware of and adhere to the higher order governing principles.*" [Plato, Politeia 506b].

Probably the perfect inheritance structure is just as much a utopia as was Plato's perfect state he describes in the Politeia. If we define a super class for creatures that walk on two legs and call them humans as well as one class for animals that fly through the sky which is called birds, what do we do with the ostrich? Often things have less in common that we think they have.

**The Next Step of the Evolution: Aristotle and S-BPM.** The advantages of object oriented design and object oriented programming are clear: The top down approach of structured analysis and UML1 makes things simple and quick to discuss them, where the most important parts need the deepest discussions as Socrates proposed. Object-oriented design languages, e.g. UML, are attempting to be both a descriptive language and an implementation language in the sense of Plato's universal, all-encompassing model.

Aristotle's criticism of his predecessors in ancient philosophy applies equally well to the early design methods. SD and UML1 as well as non formal BPM methods (like ARIS) were never intended to be an abstraction of the physical solution, i.e. the machine code. They exist only on paper or as a graphical model within some drawing tool. In fact, they do not have any guaranteed reference to the executable machine code. They may be describing something totally different than that what has been implemented. And, since the code changes, but the design documentation remains static, the model soon loses its value, if it ever had one. In our fast changing world they represent a lost investment. The least change to the code renders the model obsolete. The two descriptions no longer match. The problem is that model and code are depictions of separate worlds. This is the same as with Plato's teaching of the ideals. They form another reality with a complete different ontological state. How can we use an explanation of World A to explain something that lies in world B when the two worlds differ so much? This was the reason for Aristotle to claim the following:

"*Are these now separated from each other, there would be no science of the ones while the others wouldn't be existing*." [Aristotle, Metaphysics 1031b].

Object-oriented design methods and their graphical representation UML2 have reunited the model and reality, but are lacking in another respect. The formal OO theory defines objects, methods and properties as their main entities. But in the real world, objects are passive. An invoice in the real world does not have any functionality like "calculate sum". In the real world, this functionality belongs to the actor responsible for preparing the invoices. He calculates the sum and adds it to the invoice. If this actor is automated, it still is an actor and hence a subject, not an object. An object which controls the behavior of other objects, such as the control object proposed by Jacobson is an artificial entity that should exist, neither in the real world nor in natural language. The use case and the control object are no more than patches that were added to make the original method usable. So, instead of defining new artificial entities like Plato's ideas or control objects with functionality, Aristotle demands that one should define subjects: "*Definitions exist either only or at least preferable from substances (subjects).*" [Aristotle, Metaphysics 1031a].

The only objects that should be allowed are data beans. Everything with more functionality than get and set methods is no longer an object but a subject. With its introduction of the subject, the SPBM method is in tune with this major criticism of Aristotle. The S-BPM method takes this claim seriously in introducing the notion of subjects denoting any kind of actor within a business process: "The acting elements in a process are called SUBJECTS. SUBJECTS are abstract resources which represent acting parties in a process. SUBJECTS are process specific roles. They have a name, which gives an indication of their role in the process. SUBJECTS send or receive messages to or from other process participants (SUBJECTS) and execute actions on business objects" [Fleischmann2010, p. 12]. SUBJECTS are distinct from each other by their behavior, which consist of chains of elementary operations such as SEND, RECEIVE and ACTION.

Thus the SUBJECTS of an S-BPM model are already implemented sub systems in form of prefabricated building blocks. Their specific behavior can be modeled by drag and drop in order to define the unique behavior of their relationships. They might either represent an existing or planned system or a human actor or an organizational unit. In case of a system, they need to be linked to the system, but the systems functionality can be simulated as long is the connection is not yet completed. In case of a human actor, the human interface is already provided by the SUBJECT. The relations of the SUBJECTS are defined within the S-BPM communication view, where "it is defined which SUBJECTS participate in a process and which messages they exchange" [Fleischmann2010, p. 12]. Altogether they form an S-BPM business process.

It is important to note that a modeling method defines only the syntax of a language. The semantics come together with modeling and are provided by the entities and relationships the model artifacts are representing in the real world. Since existing legacy systems or actors have to interact with the new actors, they should be modeled first. For existing actors and constraints, the model is made according to the real world. For actors that are not yet existing, they are modeled according to the role

they should play. Otherwise the process might be executable according to a weaker definition, but not to the strong definition required for process operation. It seems that the bottom-up approach has partly to be also applied to the semantics (not discussed within this paper) in order to get a fully operational process.

## 5 Conclusion

The issues to be considered in developing a "better" implementation language as a successor of the OO paradigm according to Demelt and Maier are the following [DemeltMaier2008]: Better ability of structuring, higher language abstraction, development of formal methods and tool support.

A more valuable model in terms of cost effectiveness, ease of use and comprehensibility is achieved by the S-BPM method, by taking the subject out of the objects and putting it there where it belongs, as in the real world where subjects are independent of the objects they deal with. Not only that, but S-BPM is a pure abstraction of the underlying Java code and remains in tune with it throughout the system evolution. The S-BPM Method is based on formal theory and is supported by the Metasonic Tool Suite, which ensures that model and implementation are always synchronized.

Just as Aristotle adapted from Plato the principle of a complete and all encompassing description of reality, a description which combines generality with specifics, so has the S-BPM method taken over the principles of abstraction from object-oriented design. However, instead of having a single, central hierarchy as with Plato and OO, S-BPM follows the Aristotelian philosophy by having several parallel hierarchies. It distributes the complexity by subject. In that way, the problems associated with deep inheritance trees are avoided. Also the objects are separated from the subjects. This corresponds to the principle of Aristotle, in distinguishing between the category and the members of categories, which can be either subjects or objects.

The S-BPM Method is devoted to being a mirror image of the real world, reflecting the real life subjects, objects and actions. The model can be mapped 1:1 not only to the code but also to the business process the code is embedded in. In so doing, it becomes cost effective, easy to use and easy to understand, thus achieving a maximum value to the user. Finally S-BPM fulfills all the requirements of Demelt und Maier in regard to creating a new paradigm for software development.

## References

1. Buchheim, T.: Ist wirklich dasselbe, was zu denken ist und was existiert? – Klassische griechische Philosophie. In: Fischer, E., Vossenkuhl, W. (eds.) Die Fragen der Philosophie, C.H. Beck, Munich (2003)
2. DeMarco, T.: Structured Analysis and System Specification. Yourdon, New York (1978)
3. Demelt, A., Maier, M.: Paradigmenwechsel: Was kommt nach der Objektorientierung? In: Objektspektrum 2008 / 6, SIGS DATACOM, Troisdorf (2008)
4. Djikstra, E.: The Humble Programmer (EWD340). Communications of the ACM (1972)

5. Fleischmann, A.: What is S-BPM? In: Buchwald, H., Fleischmann, A., Seese, S., Stary, C. (eds.) S-BPM, CICS Band, vol. 85. Springer, Heidelberg (2010)
6. Jackson, M.: System Development. Prentice/Hall International, Englewood Cliffs (1983)
7. Jacobson, I.: Four Macro Trends in Software Development“. In: Proceedings Keynote of: Conference on Software Maintenance, Florence. IEEE Computer Society Press, Washington (2001)
8. Martin, J., Odell, J.: Object-oriented Analysis & Design. Prentice/Hall International, Englewood Cliffs (1992)
9. Nietzsche, F.: Über Lüge und Wahrheit im außermoralischen Sinne, Kritische Studienausgabe Band 1, DTV de Gruyter, München (1999)
10. Rumbaugh, J., Blaha, M., PRemerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs (1991)
11. Shlaer, S., Mellor, S.: Object Oriented Systems Analysis: Modeling the World in Data. Prentice Hall, New Jersey (1988)
12. Sneed, H.: The Myth of ‘Top-Down’ Software Development and its Consequences for Software Maintenance. In: Proceedings of Conference on Software Maintenance, Miami 1989. IEEE Computer Society Press, Washington (1989)
13. Sneed, H.: Software Entwicklungsmethodik. Verlag Rudolf Müller, Köln (1996)
14. Warnier, J.D.: Logical Construction of Programs. Van Nostrand Reinhold Company, Paris (1974)