Paradigm, Language Model and Method

Jan Bosch[*]
Department of Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25, Ronneby, Sweden

E-mail: Jan.Bosch@ide.hk-r.se
WWW: http://www.pt.hk-r.se/~bosch

Abstract

Programming languages and software engineering methods are highly related in that both are images of an underlying paradigm. In this paper we investigate the role of the paradigm, the language model and the software development method and define requirements on each of the three concepts. Subsequently, we describe the *layered object model*, an extended object model which allows for first-class representation of relations between objects and constraints on the behaviour of an object, and discuss how it achieves *paradigm extensibility*, i.e. extensibility of the paradigm and the supporting language model.

1 Introduction

When investigating the relation between programming languages and software engineering methods one cannot avoid placing this in a larger context. This context consists of the underlying paradigm applied by the software engineer when modelling a part of the real world, i.e. his problem. We define a paradigm as a related set of concepts which are used to model part of the real world. The paradigm used by a software engineer is like the 'glasses' he wears when modelling the real world. For example, a software engineer applying the functional paradigm views his problem in terms of data elements and functions. However, a software engineer working in an object-oriented manner will view the same problem in terms of objects and relations between objects. The paradigm applied by the software engineer plays a central role in the way software is constructed.

In this process, we can recognise several modelling activities. First, the underlying paradigm defines a meta-model of the real world, i.e. it defines a set of concepts which are instantiated by the software engineer when making a model of the world. Referring to the previous example, the software engineer might have the concept *object* as part of the paradigm he uses, which enables him to model an entity recognised during the development process as an object. The software engineer must *have* this concept as part of the paradigm he applies, otherwise he would be unable to *'instantiate'* it. As in any science, the way a software engineer perceives reality is a *model* of reality: it is *not* reality itself. This is what metaphysics is trying to make us aware of: a human being can never perceive reality as a whole. He makes a selection, or an abstraction, of the overwhelming amount of sensory information and associates patterns of sensory information with concepts. In the same way this process of perception is a modelling activity within the

[*]This work has been supported by the Blekinge Forskningsstiftelse.

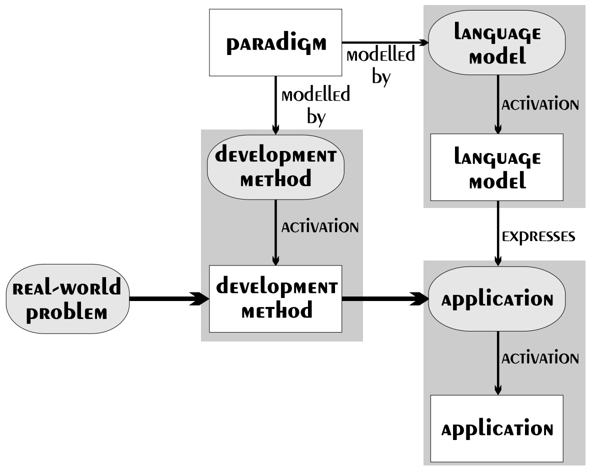

Figure 1: The paradigm and its models

frame of a cultural paradigm, is the modelling of software by a software engineer a modelling activity within the frame of his paradigm.

The second modelling activity is the development of a *programming language model* supporting the paradigm as a, possibly executable, language which can conveniently express the real-world model defined by the software engineer when developing an application. The programming language model again is a model, a reflection of the paradigm. For example, it is therefore that it is advantageous to implement the result of object-oriented analysis and design in an object-oriented language. It can, up to a certain extent, also be implemented in a third generation language, but that language does not reflect the object-oriented paradigm. This requires the software engineer to translate the object-oriented model into a functional model before it can be implemented. In this case it is the software engineer who has to cover the *semantic gap* between the paradigms.

The third modelling activity is the development of a *method*, i.e. the result of methodology, the science of methods, supporting the development process performed by the software engineer. The method also is a reflection of the paradigm in that it, generally, defines steps for the identification and specification of instances of the concepts that are part of the paradigm. Conventionally, methods often define alternative models that capture representations of the problem domain that can not be expressed in the programming language model, but do have some semantics. Examples of this are data- and control-flow diagrams in the structured methods.

In figure 1, the relation between the paradigm, the language model and the method is shown. Both the language model and the method are models of the paradigm and have a passive (ellipse) and an active (square) form. For instance, when defining method, the method itself is passive (ellipse). However, when the method is applied during application development the method is active (square). This principle extends to the application. The application is first defined and afterwards it is activated, indicated by the ellipse and square representations of the application.

2 The Role of the Paradigm

We use the term ‘paradigm’ to refer to a set of related concepts which are used by a person to perceive the real world or a part of it. Within software engineering, this means that the software engineer uses the concepts within his paradigm to perceive his problem domain.

Do note that the use of the word 'paradigm' in computer science is (slightly) different from its use in philosophy. For example, in [11] a paradigm is defined as 'a universally accepted set of presuppositions'. We generally are unaware of the paradigm, according to philosophy, because we are so familiar with our basic presuppositions that we do not know them. It is only when science *changes* its paradigm, e.g. the transition from Newtonian to Einsteinian mechanics, that one becomes aware of his underlying paradigm. The collapse of a paradigm is, generally, caused by an increasing set of anomalies associated with that paradigm. Anomalies, in this context, refer to aspects of the studied domain that cannot be expressed (conveniently) in the paradigm. Scientists become increasingly unsatisfied with the existing paradigm and a new paradigm, chanceless during the 'normal science' phase(see [9]), replaces the old paradigm. The new paradigm, although it might use some of the same terminology, replaces the *whole* set of presuppositions and it cannot be be expressed in terms of the former.

In computer science, the different paradigms, or schools of thought, have replaced each other relatively quickly. Also, we never had a situation where a large majority of computer scientists accepted a paradigm without being aware of alternatives. We believe it is therefore that software engineers are more aware of their 'presuppositions' than their counterparts in, e.g. mechanical engineering. Nevertheless, it still proves to be very difficult to reflect upon ones paradigm and to recognise anomalies of the paradigm.

The paradigm is the common factor in analysis, design and implementation. During the history of software engineering, one can view a circular development in which first a new paradigm or a paradigm shift is defined, then a programming language supporting this new paradigm is defined and subsequently, a method (or methodology) is defined to support software development based on the new paradigm.

2.1 Historical Overview of Paradigms in Software Engineering

The first 'real' paradigm we recognise is the functional approach to software development with methods defined by, among others, [10]. In this paradigm the emphasis is on the system function which is organised as a hierarchy of functions (or processes) connected by data flows.

The second paradigm — perhaps better described as a paradigm shift — is the entity-based structured approach of which Jackson System Development (JSD) [7] is a good example. Experience had shown that it was not sufficient to focus only on functions, i.e. anomalies of the functional approach were identified. In this new paradigm, therefore, the designer focussed both on relevant entities, in the system and on the system functions. However, these two were still seen as two separate dimensions, i.e. entities were structured but unencapsulated data and functions were still global.

The third paradigm, which is currently state of the art in industry, is the object-oriented paradigm. Again, anomalies with the previous paradigm lead to a software industry-wide frustration which created the basis for introducing a new paradigm. Using the object-oriented paradigm, the world is modelled in terms of objects, incorporating both data and operations on that data. These objects represent entities that are, by humans, perceived as relevant entities in the real world.

The general line of development, as one recognises, is that the paradigm used to make a model of the real world has moved away from its roots in mathematics and shows more and more resemblance with the way, we believe, humans to make a cognitive model of the world. It seems that in the subsequent paradigms the *semantic gap*, referred to by several authors e.g. [6], is decreasing. Also research from cognitive psychology, e.g. [12], indicates that the object-oriented paradigm indeed might be more intuitive than conventional approaches.

However, it is unrealistic to assume that the development of paradigms will stop with objects. The object-oriented paradigm, as any paradigm, is not the perfect solution. Some authors have identified anomalies or obstacles of the conventional object-oriented model, e.g. [1]. However, assuming the development of new paradigms does not stop with objects, what is to be expected?

One direction are the extended object models, which are object models providing some form of reflection. Reflection is defined as the ability of a system to reason about and act upon its internal representation. Examples of extended object models are *CLOS* [8], composition-filters object model [2] and the *layered object model* (LayOM). The latter object model will be addressed in more detail in section 3.

In this section, we have focussed on the main-stream paradigms. We deliberately excluded less common approaches, like the 'real' functional languages, e.g. *Miranda* and *ML*, or the languages based on first-order logic, e.g. *Prolog*. Also, we did not address current, AI-based approaches, like *agent-oriented programming*.

2.2 The Set of Related Concepts

In the introduction we defined a paradigm as a set of related concepts. But what concept sets are related to the paradigms discussed in the previous section? For the functional paradigm, relevant concepts include *function*, *variable* and *data flow*. However, the *function* was the most important concept around which the method and the programming language centered.

The entity-based, structured paradigm consists of concepts like *entity*, *function*, *data structure* and *data*- and *control-flow*. The *entity* and the *function* were equally important and were viewed as two orthogonal dimensions of the system.

The conventional object-oriented paradigm — we will discuss this paradigm in more detail — consists primarily of the following concepts: *object*, *method*, *instance variable* (or nested object), *message*, *inheritance* and *class*.

Despite our description, a paradigm is generally only defined abstractly. It is more through its representations, i.e. the programming language and the development method, that the paradigm implicitly is available. However, when we investigate the programming language models and the methods for the various paradigms, we see that often the supported sets of concepts are not the equivalent. Generally, the programming language supports a subset of the concepts supported by the method. In table 1, we show the concepts related to each paradigm.

Concluding, we can state that the software development process can be seen as a translation of a part of the real world, consisting of a large set of weakly defined, i.e. possibly ambiguous, concepts into a model, expressed in the language model, consisting of a very small set of very precisely defined concepts. The paradigm, in this view, is an intermediate representation, used by the software engineer, consisting of a set in between the real-world and the language model both in size and in the preciseness in which each concept is defined.

2.3 Requirements

Based on the interpretation presented in this paper, a number of requirements upon the relevant entities in software development, i.e. the *paradigm*, *language model* and *method*, can be defined. The general requirement is that the model in which the software engineer has to express his application should be as close as possible to the way the software engineer perceives the world.

The above requirement we believe to be the main requirement of the *paradigm* used by the software engineer. We believe the software engineer, as a human being, to perceive the world

paradigm	*concepts*	
	development method	language model
functional	process (function) data flow variable	function variable
structural	entity process relation data/control flow abstract data type	module function structured type
object-oriented	object message class reuse method nested object relations constraints (e.g. concurrency)	object message class inheritance method instance variable

Table 1: Relation between Paradigms and Concepts

using a meta-model containing a large set of concepts which often are weakly defined. During software development, the software engineer has to convert the domain as he perceived it using his personal knowledge, into a model defined by the concepts that are part of the paradigm. From this view on the design process we can deduce two requirements. First, the paradigm should be as close as possible to the way the software engineer perceives the world, but not so close that it loses the required preciseness of the concepts. Do note that the paradigm does not *have* to be unambiguous. Second, the software engineer should be able to *evolve* the paradigm, i.e. extend the paradigm with new concepts and remove or redefine existing ones. From the historical overview we have learned that no paradigm, by itself, is a complete solution. It is nothing but a *'working hypothesis'* which is replaced when a better is found. Therefore, in anticipation of extensions and changes to the paradigm, we should *explicitly* support this.

The requirements related to the *language model* can be expressed as follows. As the language model itself is a model from the paradigm, it should be as accurate as possible. Accurate, in this context, means that each concept part of the paradigm should relate to (exactly) one language model element. From this, a second requirement can be deduced: the language model should be extensible in order to reflect extensions and changes to the paradigm.

Finally, the *method* also is a reflection of the paradigm, but is also influenced by the language model. The method is a process description of the transformation of the part of the real-world into the language model. The method should support the software engineer while expressing the problem at hand in terms of the paradigm. Generally, this means the identification and the specification of instances of concepts that are part of the paradigm and an ordering on the identification of different concepts. Also the method should be extensible to incorporate paradigm extensions.

3 Our Approach

As described in section 2.1 the object-oriented paradigm currently is the state of the art in the software industry. Our research effort, at the University of Karlskrona/Ronneby in Sweden, is to investigate the conventional object-oriented paradigm in order to identify anomalies and to propose evolutionary extensions to the conventional paradigm that might resolve the anomalies that we identified. Our research includes identification of anomalies related to the paradigm, the language model and the development method.

In the remainder of this section, we first describe our language model. Then we describe how we achieve *paradigm extensibility*, i.e. how the software engineer can *extend* the language model. In the last subsection, our approach to defining development methods and to providing automated support is described.

3.1 Layered Object Model

The language model that we use as a 'test vehicle' is LayOM, the layered object model, which extends the conventional object model by encapsulating it with *layers*. A layer can, among others, represent a relation with another object or a constraint, e.g. a concurrency or a real-time constraint, on the behaviour of the object. To express a certain type of behaviour, e.g. partial inheritance, a layer of type *PartialInheritance* is defined and configured to provide the required behaviour of the object. The software engineer is able to define new layer types and use them interchangeably with other, predefined, layer types. Do note that this is different from the conventional class definition and instantiation. When defining a new layer type, the software engineer extends the object model *itself*, i.e. the expressiveness of the object model is increased. When defining a new class in the conventional object model, the object model itself remains the same.

The layered object model (LayOM) is an extended object model, i.e. it takes the conventional object model as a basis and extends that to improve the expressiveness of the model. An object in LayOM consists of five major components, i.e. *variables* (nested objects), *methods*, *states*, *conditions* and *layers*. LayOM does not support *inheritance* as a part of the model, but deals with *inheritance*, *delegation*, *part-of* and many other types of relations between objects through *layers*.

The *layers* encapsulate the object such that messages sent to the object or sent by the object itself have to pass all layers. Each layer can change, delay, redirect or respond to a message or just let is pass. Currently we use layers for (1) representing relations the encapsulated object has with other objects and (2) to defined constraints on the behaviour of the object. Below, layers are discussed in more detail.

The variables of the object are nested objects within the object in which they are declared. Variables as well as methods of the object are defined as in most object oriented languages.

A state, as defined in LayOM, is a dimension of the, externally visible, abstract object state. Each dimension has a domain and the state expression 'maps' the concrete object state (or a part of it) to the domain associated with the state. LayOM also supports *active states*. An active state is defined based on the static abstract state defined for the object and can be seen as a derivate function of (a part of) the static abstract state in time. Client categories are defined in LayOM using *conditions*. A condition, just as a state, can be seen as a specialisation of a method. A condition has a single (invisible) argument *msg*, which is the message for which it is determined whether its sender belongs to the client category defined by the condition. The return value of the condition is restricted to class *boolean*. We refer to [5] for a detailed discussion

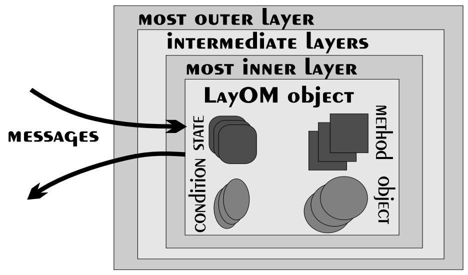

Figure 2: LayOM Object

about *states* and *conditions*.

The layers are the central concept in LayOM. As mentioned, a layer can model a relation between the object and another object or a constraint on the behaviour of the object. Examples of relation types that can be directly represented by a layer include *inheritance*, *delegation* and *part-of* but also application specific relations as *works-for* and *is-author-of*.

Layers can be used to restrict access by certain clients but also to extend the interface of the object. For instance, a layer of type *Condition* can be used to define a new client category which was not defined within the object itself. From the perspective of a layer, it makes no difference whether it is in between other layers, the most outer or the most inner layer. A object is defined such that at the outside of each layer, one 'sees' an object with a interface consisting of methods, states and conditions.

A message that is sent to the object has to pass the layers encapsulating the object, starting at the outermost layer and working his way in. Each layer receives the message and inspects it in order to determine what it will do with the message. A layer can delay the message until some precondition is fulfilled or it can change the message contents. Other possibilities are that the layer itself replies to the message or that it rejects the message, generating an exception. Basically, each layer object has the opportunity to handle a message in virtually any way it likes. However, in practice, a layer will just pass most messages on to the next layer and only intercept a subset of the messages. In figure 2 the structure of an object in LayOM is illustrated.

Analogous to a message sent *to* the object, a message sent *by* the object has to pass all the layers starting at the innermost layer. Again, each layer has the possibility to treat outgoing messages in any way it sees appropriate. Do note that each layer, either in response to a message it needs to evaluate or otherwise, can send messages to the object it encapsulates. For instance, a layer can request the value of a state or a condition.

3.2 Extending the Object Model

The original rationale for defining the layered object model was to provide the object model with the ability of expressing relations between objects as first-class entities. We consider this very important because object-oriented methods explicitly model relations between objects and classify these relations, i.e. give each relation a certain type. However, the conventional object-oriented model *only* supports the *inheritance* and the *part-of* relation within the model. All other relations identified during analysis and design have to be implemented *on top of* the model, i.e. implemented as method code and message sends.

As described in section 2.3, we believe that the language model should be capable of representing the concepts that are defined within the paradigm. Thus, also relations should be expressable within the object model. This lead to the definition of the layered object model. LayOM allows the definition of relations between objects as layers encapsulating one of the objects. Also, the software engineer is able to define new layer types and use them as any layer type.

However, from the further investigation of the layered object model we have learned that the layer mechanism offers a second, very important feature: *extensibility of the paradigm itself!* If the software engineer defines a new layer type, then the object model is basically *extended* because the behaviour defined in the new layer type, e.g. a relation type, which first had to be expressed on top of the model can now be represented within the model as a first-class entity. An analogy would be the implementation of an object-oriented application using a third generation language. In that language the software engineer has to express objects, messages, etc. on top of the language, whereas in an object-oriented language these can be expressed directly in the language.

Therefore, we believe to have defined a language model which supports extensibility of the paradigm and the language model. Whenever the software engineer, as a result of reflective thinking about the underlying paradigm, identifies a new concept that had not been recognised before, this new concept can be adopted in the conceptual paradigm *and* extend the language model with a new layer type that implements the behaviour of the concept.

3.3 Development Method

The role of the development method is to guide the software engineer in the modelling process. The real-world problem has to be expressed in terms of the concepts in the paradigm and the process of achieving consists of a number of steps or phases. For instance, generally an object-oriented method consists of the following steps (1) requirements and domain analysis, (2) object and subsystem identification, (3) relation identification, (4) structure specification and (5) behaviour specification. These steps are performed in a sequential order, but iterations can occur at any point and do not necessarily restart at the first step.

However, this type of development method is defined from the perspective of the software engineer. This approach seems the most natural one from a conventional perspective, but we believe it not to be the optimal approach. Especially when developing complex and large applications using automated support for the development process, we have identified problems with the conventional software engineer centered development process. In [3], we describe three problems:

- *Functional approach to automated support*: The conventional development method, described above, can be seen as analogous to the the functional paradigm in programming. Hence, the tools for automated support can be seen as functions operating on the passive, data-like design artifacts in which the analysis and design model is kept. This results in problems analogous to the problems the functional paradigm has in programming, i.e. (1) lack of controlled access to design artifacts, (2) artifacts changes have global impact and (3) changes to the methodology are difficult to implement.
- *Lack of initiative by the system*: This 'functional' approach to automated support leads to the situation where the initiative always is on the side of the software engineer. The paradigm does not facilitate consequent automation and, generally, the automated support environment does not provide *process support.*
- *Causal connection gap*: In the conventional approach it is very difficult to maintain a *causal connection* between the design artifact and a part of the application. The term 'causally

connected' is taken from the field of reflective systems and refers to a causal connection between an entity and its meta-level representation. In the context of this paper, this refers to the causal connection between an object in an application and its corresponding design artifact. Problems resulting from the causal connection gap are that (1) the design rationale for an entity gets lost, (2) lack of control by an entity when being defined and (3) relation between a concept and its implementation disappears.

In order to address these problems, our approach to automated support is the opposite from the conventional approach. Rather than defining a development method from the perspective of the software engineer, we define it from the *perspective of the design artifacts*, i.e. the entities under development. The development method is not defined as one large description, but is decomposed into small development methods for each design artifact type. A design artifact type corresponds, in our approach, to a concept in the paradigm. Thus, for each concept in the paradigm, the development steps are explicitly defined.

Each instance of a design artifact type (or concept) is an active object (or agent) that is responsible for defining itself and it communicates with other design artifact objects to collect information concerning itself and its context. A design artifact object always performs the steps in its development method for which is can collect sufficient information. If certain information concerning the object is lacking, it turns to the software engineer to request for this information.

The above description is very brief and we refer to [3] for a more detailed description. However, our approach to development methods and automated support for these methods does not suffer from the described problems, because (1) it is based on the object-oriented paradigm, rather than on the functional paradigm, (2) each design artifact object will perform as much activities as possible before turning to the software engineer and (3) a design artifact object corresponds to exactly one concept in the paradigm, which is also an entity in the language model.

4 Conclusion

We have discussed the relation between the language model and the software engineering method and have recognised that it is the underlying paradigm which is the major factor. Both the language model and the development method can be viewed as reflections of the paradigm used by the software engineer. Their quality is determined by the accuracy of the respective 'reflections'.

Based on this perspective we discussed the different paradigms that historically have played an important role in software engineering and concluded that the subsequent paradigms seem to move away from their mathematical roots and become closer to the way we believe humans to perceive the world. However, it is unrealistic to assume that the development of new paradigms will stop today and we mentioned some possible developments. To provide guidelines, we defined requirements that the paradigm, the language model and the method have to fulfill. These requirements can be summarised as *accuracy* and *extensibility.*

Our approach to investigating the expressiveness of object models is the layered object model (LayOM), an object model that extends the conventional object model with layers encapsulating the object. These layers can express relations, constraints and other types of behaviour of the object as *first-class entities.*

A second important property of LayOM is that it facilitates the extensibility of the paradigm itself. The software engineer is able to define new concepts and add them to the layered object model in the form of new layer types. We expect that only a certain class of paradigm extensions can be expressed using layers and we are currently evaluating its generality. One future direction we plan to investigate is related to *design patterns*. We believe it to be possible to express design patterns as first-class entities, preferably layer types, within the object model. This would facilitate a very natural implementation of a design pattern based design in LayOM.

Our approach to defining development methods and providing automated support is unconventional in that we define a development method from the perspective of the design artifact. Each type of design artifact refers to a concept in the paradigm. This approach avoids problems associated with conventional, software engineer centered approaches.

Acknowledgements

I would like to thank Lennart Ohlsson and Berthel Sutter for their comments on this paper.

References

[1] M. Aksit, L. Bergmans, "Obstacles in Object-Oriented Software Development," *OOPSLA'92 Proceedings,* pp. 341-358, October 1992.

[2] L. Bergmans, "Composing Concurrent Objects," *PhD Dissertation*, Department of Computer Science, University of Twente, The Netherlands, 1994.

[3] J. Bosch, "Software Artifacts as Autonomous Agents," submitted to *the 17th International Conference on Software Engineering (ICSE-17).* Also Research Report 6/94, University of Karlskrona/Ronneby, September 1994.

[4] J. Bosch, "Relations as First-Class Entities in LayOM," *Working paper,* Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, October 1994.

[5] J. Bosch, "Abstracting Object State (preliminary title)," *Working paper,* Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, October 1994.

[6] G. Booch, *Object Oriented Design (with applications),* Benjamin/Cummings Publishing Company, Inc., 1990.

[7] M. Jackson, "System Development," *Prentice Hall,* 1983.

[8] G. Kickzales, J. des Rivières, D.G. Bobrow, *The Art of the Metaobject Protocol,* The MIT Press, 1991.

[9] T.S. Kuhn, *The Structure of Scientific Revolutions,* The University of Chicago Press, 1962.

[10] T. DeMarco, "Structured Analysis and System Specification," Prentice-Hall, 1979.

[11] I. Marková, "Paradigms, Thought, and Language," John Wiley & Sons Ltd., 1992.

[12] M.B. Rosson, S.R. Alpert, "The Cognitive Consequences of Object-Oriented Design," *Human-Computer Interaction,* Volume 5, pp. 345-379, 1990.