Software Artifacts as Autonomous Agents

Jan Bosch
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby Sweden
Tel.: +46-457-787 26 Fax: +46-457-271 25
E-mail: Jan.Bosch@ide.hk-r.se

Abstract
In this paper, a number of problems of conventional automated software engineering support environments are described. These problems are related to the functional approach these environments take, the lack of initiative these systems exhibit and the causal connection gap between design information and the operational entity. As an alternative, we suggest an autonomous agent approach. Each artifact is modelled as an autonomous agent that is responsible for its own development. Artifacts are organised in specialisation hierarchies which can be traversed by agents. As a consequence, the methodology needs to be inverted, i.e. specified from the artifact's perspective rather than from the software engineer's perspective.

1.0 Introduction

Software engineering, as a discipline, can be seen as being combined of a methodology, incorporating a notation, a collection of methodological steps and a process, a conceptual computational model and an operation model. As software engineering is a very complex activity, which has proven to be error prone and difficult to control, many attempts have been made to automatically support the process of developing software.

A typical automated support environment for software engineering, generally referred to as an ICASE environment, consists of a database of design information in the centre and a collection of tools around this central database to manipulate the design information. Pieces of design information are referred to as artifacts, which are typically data structures which are filled in during the development process.

In this paper we describe a number of problems with this approach to providing automated support. These problems are related to the functional approach conventional environments take, the lack of initiative these systems generally have and the causal connection gap between the design information and the operational entity that results from it.

As a solution, we suggest an autonomous agent approach. An artifact, in this approach, is modelled as an autonomous agent that is responsible for its own development. The agent contains the structure of the operational component it represents at the design level. Secondly, it contains the methodological steps that can be executed on the artifact. Thirdly, it has a reasoning part that has as a main goal to collect information concerning the artifact and to execute the methodological steps when the preconditions for a step are satisfied.

This paper is organised as follows. In the next section we describe a framework for software engineering. In section 3, a number of problems with the traditional approach to software engineering are described. Section 4 discusses an alternative approach to providing automated support to software engineering. In section 5 we briefly discuss some related work and in the last section we evaluate and conclude the paper.

2.0 Software Engineering Framework

Software engineering is an engineering discipline, in which a rich base of design principles, hints, methods and techniques are available. What distinguishes an engineering discipline from an ad-hoc approach is that the analysis of the problem, the design of the solution and the associated implementation occur in a systematic manner. In the case of object-oriented software engineering, the following components can be recognized:

- **Methodology**: a methodology consists of a notation, a collection of methodological steps and a process, describing an ordering of these steps. The methodology guides the software engineer while converting user requirements into an application.
- **Object Model**: the resulting application is expressed in terms of an object model with certain semantics. Although the conventional object model is most common, also extended object models have been defined. Two examples are the composition-filters object model [Aksit et al. 94] and the layered object model [Bosch 94].
- **Operation Model**: the lowest level is the operation model, implementing the semantics of the object model. Traditionally, the operation model was implemented by the execution environment, but nowadays additional functionality is often used from libraries for particular domains. As an example the research in micro-kernel operating systems, e.g. Choices [Campbell et al. 89], can be used. Microkernels aim at providing flexible functionality. However, in this paper we are not concerned with the operation model.

Each of these three (conceptual components) have practical, automated counterparts. As mentioned, the operation model is implemented by means of an operating system and some additional elements. The object model is implemented through a compiler and, possibly, other low-level tools. The methodology can be implemented by means of an integrated computer aided software engineering (ICASE) environment, providing counterparts for the notation, the methodological steps and the process, or a subset.

In figure 1, the relation between the different components is exemplified. We use the notion of a dual form for each component, it is either passive (or reified) or active (deified). In its reified form the internal structure of the component can be accessed and changed, whereas in its deified from the component is actively operating within its context, rather than being operated upon. For instance, an application is first designed by applying the methodology and the model, and is afterwards ‘activated’. In the same line of reasoning, both the model and the methodology can be seen in their active and their passive form. The model, in its passive form, can be extended with additional semantics and then activated to be used in application development.

However, when the semantics of the used model change, the methodology needs to be extended too, in order to use the additional semantics of the model. The grey boxes around the meta-methodology and the passive form of the model indicate that these elements are generally not part of the automated environment. In figure 1, the methodology is also expressed in terms of the object model. Although it is not required to do this, it is generally beneficial. One reason is the uniformity of the environment. The software engineer does not have to change paradigm while working with the system. A second reason is that it allows one to use the same environment to develop and extend the methodology itself.

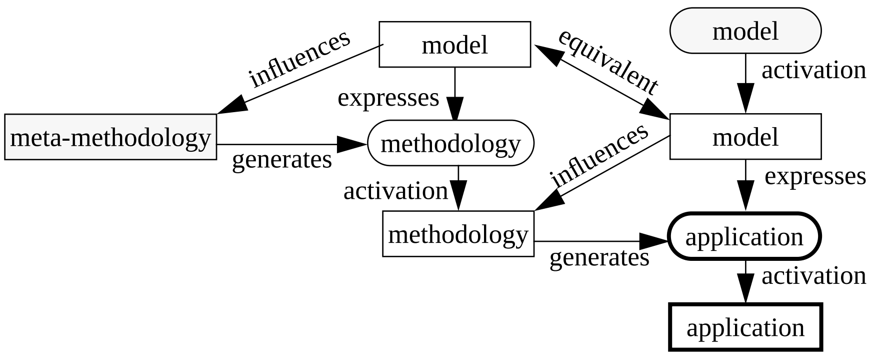

Figure 1 - illustrating the relation between the application, methodology and model

At this point we should mention that, in this paper, we use a specialisation, i.e. an extension, of the object-oriented model, i.e. an autonomous agent model, for the methodological part. Software artifacts are represented as autonomous agents. Each agent tries to collect design information in its context and from the software engineer, in order to define the operational entity it represents. This operational entity can be, for instance, an object, a class or a subsystem. This approach to automated support of software engineering is capable of addressing the problems of conventional automated environments.

3.0 Problems

Ever since the term ‘software crisis’ was introduced, more than 25 years ago, software engineering has been a field with many problems. Although, over the years, very complex and impressive systems have been built, the conventional engineering disciplines have been much more mature in the quality of both the process and the product. In this section a number of problems with current software engineering practices are discussed.

Despite the major attention automated support for software engineering got since the 80’s, in particular the integrated computer aided software environments, the industrial use of these environments did not take off as expected. The expected advantages of ICASE were not fulfilled by the products that became available. Many software engineers who had access to an ICASE environment only used a small part of it, e.g. the drawing tools. Most of the problems we discuss in this section are related to the automated support of the software engineering process, rather than to software engineering itself.

The problems we will address are:

- functional approach to automated support
    - lack of controlled access to software artifacts
    - artifact changes have global impact
    - changes to the methodology are difficult to implement
- lack of initiative by the system
- causal connection gap
    - design rational for an entity gets lost
    - lack of control by an entity when being defined
    - relation between a concept and its implementation disappears

We are aware of the fact that not all ICASE environments suffer from all described problems. A number of environments have found solutions. However, the claim we make in this paper is that we believe that the fundamental approach of these environments is wrong. The conventional, functional approach to automated support does not work properly in automated support for software engineering in the same way as it did not work in programming. What is required is an object-oriented approach to providing automated support, extended with artificial intelligence features.

3.1 Functional Approach to Automated Support

We have recognized that there is a problem analogous to the functional approach to programming in software engineering. The conventional, functional approach to programming divides the application into functions and data, where the functions operate on the data. There are many problems associated with this approach, such as the lack of maintainability and reusability.

Analogous to the functional approach to programming, we can see an equivalent approach in the design process. In the case of an ICASE environment, one can view the individual tools as functions that operate on the application, which is actually data.

The same problems that are associated with the functional approach to programming can also be recognized at the methodological level. In figure 2, the conventional interaction between tools and software artifacts is illustrated. Below we describe a number of the resulting problems.

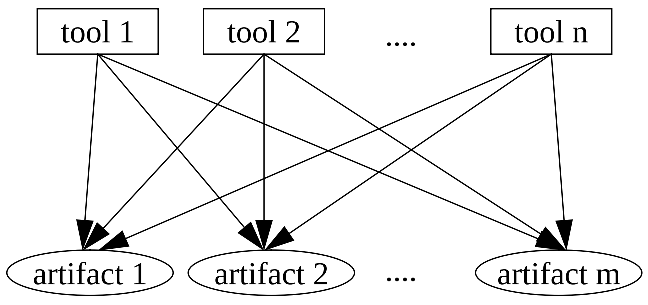

Figure 2 - illustrating conventional tool - artifact interaction

3.1.1 Lack of controlled access of software artifacts

In the functional approach to automated support, software artifacts, which together make up the application, are nothing but passive data structures that can be freely accessed. Often, there is some form of access control, but because most of the tools need to access most of the artifacts at some point during the development process, it generally is extremely difficult to define suitable access rules. Similarly, consistency of the artifact is also difficult to define, because each tool must be programmed for doing consistency checks on the artifact before it makes the required changes. This results in the distribution of consistency information to the different available tools.

The lack of control on the access can lead to several problems. First, a software engineer might change some part of the artifact without considering the global effects of such a change. Second, from several tools, changes can be made on the artifact that, when combined, lead to an inconsistent artifact, or an artifact that is inconsistent with the artifacts it is related to.

3.1.2 Artifacts changes have global impact

When the structure of an artifact, for whatever reason, needs to be changed, then these changes have a global effect on the environment. All or most of the tools that access the artifact at some point during the process, must be adapted to the change. Also, if the change to the artifact is due to a change in the methodology, one or more tools will need to be extended to incorporate the changed methodology. This leads to the next problem.

3.1.3 Changes to the methodology are difficult to implement

Methodological changes generally affect the structure of an artifact or the functionality of a tool. In case of a tool functionality change, one or more artifacts often have to be changed to facilitate the tool. Visa versa, if the structure of an artifact changes, some of the tools need to be changed to deal with the new aspects. In any case, a change to the methodology causes multiple changes in the automated environment.

3.2 Lack of initiative by the system

The conventional automated software engineering environments generally aim at storing information and maintaining the consistency of that information. What we consider to be problematic are three issues:

- The *initiative* during the software development is constantly on the side of the software engineer. The automated environment does not take the initiative at points where it is lacking certain information with a high priority. This only occurs when consistency is at stake, in which case the software engineer is forced to attend to that before continuing. We believe an environment where the initiative of taking up activities is divided between the software engineer and the environment to be preferable.
- Conventional environments generally hardly automatically perform activities during the software development process for which sufficient information is available.
- Due to the lack of initiative by the environment, the environments generally offer hardly any process support. By process support we refer to helping the software engineer in selecting the next activity to engage during the software development process.

3.3 Causal Connection Gap

From the field of reflective systems we have taken the term '*causally connected*' to indicate an analogous concept in the software development process. 'Causally connected' in reflective systems refers to a strong relation between an object, representing an entity in the problem domain, and its associated meta-object, which describes the structure and behaviour of the base level object. The relation between the base-level object and the meta-object causes any change at one object to immediately reflect at the other: the two are *causally connected*.

We believe there exists a similar relation between an object or class at the operational level, where it is an active entity, and the design level, where it is a passive entity and its structure and behaviour are defined. The structure and behaviour of an entity are defined according to the real-world equivalent of the entity and the user-requirements. However, in today's software engineering practice, the causal connection relation between design-level entity and the operation-level entity is not modelled as such. Generally, the operation-level entity is defined based on requirements and domain analysis, but once it is defined, the relation with the design level is lost. There does not exist a design level entity that represents the passive, definition form of the operation level entity.

This problem we refer to as the ***causal connection gap***, what lacks is a connection between the operation-level entity and its design-level equivalent. When this connection would be available, changes and problems of the operation-level entity would immediately reflect at the design-level. Visa versa, changes at the design level entity, due to requirement changes or whatever, would immediately reflect at the operation-level.

The lack of a design-level entity representing an operation-level entity results in a number of problems:

- the design rationale for an entity get lost
- when being defined, the entity is passive and has no control over its design
- the relation between the concept (what is in the software engineer's head) and its implementation (what is, e.g. in the class definition) is not available

3.3.1 Design rationale for an entity gets lost

At the time of designing an artifact, the software engineer knows exactly why the artifact is designed as it is. However, after some time a person tends to forget why a design decision was taken, i.e. the design rationale has been lost. As humans are not very good at storing this information, the artifact itself should store the design rationale together with the design decision.

3.3.2 Lack of control by an entity when being defined

In an average project, multiple software engineers work on an artifact during the development process. Also, a software engineer might work at different times on an artifact. As the design rationale is not stored, a software engineer might extend the artifact in inappropriate ways, e.g. violating consistency, defining parts of the artifact double, etc. The artifact has no means of control to limit the software engineer in the actions that can be performed on the artifact.

3.3.3 Relation between a concept and its implementation disappears

An operation-level entity is constructed from design decisions taken during the software development process. When, at a later point in time, one or more of the design decisions are changed, then this should affect the operation-level entity directly. However, generally the relation between the operation-level entity and the associated design decisions is lost. This requires the regeneration of these relations when the application has to be changed.

4.0 Our Approach: Artifact Agents

What we propose in this paper is an alternative approach to providing automated support for software engineering. As stated in section two, we believe it to be beneficial to apply the same paradigm to the automated environment as we apply to application development, i.e. the object-oriented paradigm. This means that, similar to meta-objects in reflective systems, we define software artifacts as objects that are able to control the design operations that occur on the operation-level entity.

However, although defining artifacts as objects is already very beneficial, as we will illustrate later, it lacks at one point: automation. We would like to maximize the automated support by the system. To achieve that, artifacts need to be active entities with reasoning capabilities. As the conventional object-oriented model does not offer these types of facilities, we decided to use a specialisation of the object-oriented model, i.e. an autonomous agent model. The agent model is based on agent models defined by [Shoham 93] and others like [Gustavsson&Hägg 94].

The software artifacts are modelled as agents, rather than as passive data structures, and have the following characteristics:

- The data part of the agent contains the structure and behaviour definition of the operation-level entity which it represents, i.e. it contains the reified entity. This definition can be almost empty, e.g. early in the development process, or large and complex, e.g. at the end of the process and when representing a complex class. But in any case, the data part is encapsulated from the rest of the environment.
- The methods of the agent represent methodological activities that can be performed on the entity being defined. These activities can be the smallest, atomic steps, but also more complex, composed activities. We will elaborate on this later.
- Artifact types are organised in specialisation hierarchies. An agent is created based on an artifact type definition, generally the top type or close to the top and starts executing as a autonomous entity. Agents can climb and descend along the specialisation hierarchy, depending on the information they collect.
- Artifact agents are ***active*** entities. Each agent has as a main goal to develop itself into such a state that the contained reified operation-level entity can be ‘deified’, i.e. can function as a class, object, etc. at the operation level. To achieve this, the agent needs to perform the aforementioned methodological activities. It, however, requires information and the preconditions of each activity need to be fulfilled before it can be executed. The agent tries to collect information and to fulfil the preconditions of activities and when it succeeds it executes the activity.

4.1 Agent Model

In figure 3, a graphical view of the agent model is shown. The agent itself consists of a head and a body. The body contains the conventional object-oriented model, but the head contains the reasoning part of the agent. The body consists of methods and data. The methods consist of a method body, i.e. the code associated with the method, and preconditions on the execution of the method. The reasoning part uses the preconditions to deduce what additional information it needs to perform the next step. The data part consists of the structure and behaviour definition of the operation-level entity which it represents and additional data, which is used by the agent itself. Messages that are sent to the agent are always redirected to the reasoning part. The reasoning part then decides how to respond. This response might be to just start the execution of the requested method. But if not all preconditions of the method hold (and the client might be able to provide this information), the agent can start a dialog with the client to collect the required information to perform the method.

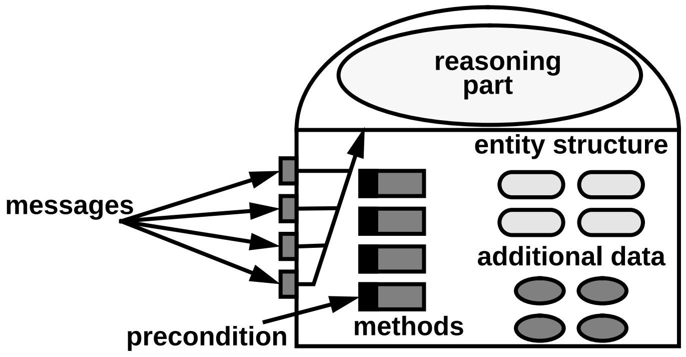

Figure 3 - graphical view of the agent model

4.2 Methodology Aspects

We believe there is an important difference between this approach and conventional approaches. In conventional approaches, the methodology is always defined from the software engineer’s perspective. All methodological steps are defined with the software engineer as the one performing these steps. The software engineer will do subsystem identification, object identification, relation identification, etc. The associated automated support environment is based on this methodology and will interpret the methodological steps in the same manner. The tools either replace or support the software engineer while performing methodological steps.

We believe this approach not the most appropriate one for constructing automated support for software engineering. Many problems with conventional environments do stem from this approach. The functional approach to software engineering environments is a result from this direct translation of steps performed by the software engineer that need to be supported or automated. Also, the causal connection gap stems from the fact that a computer cannot store what is in a software engineer’s mind and therefore should only aim at supporting the software engineer.

We aim at taking an *inverted* approach, the perspective of the methodology is inverted. The methodology is ***not*** defined from the software engineer’s perspective, but *from the perspective of the entities*. Each entity has a number of methodological steps that it can perform on itself to further define itself. Each entity is responsible for defining itself according to the requirements of the application. An entity can define part entities, e.g. a subsystem can define a class as part of itself, but this involves the creation of a new *class agent*. After creation is the class agent itself responsible for itself and its development and not the subsystem which it is part of. The subsystem remains to have the right to move the class or to remove it.

Inverting the methodology from software engineer perspective to the entities has major advantages in dealing with the problems described in the previous section. A complicating factor is that one has to totally redefine the methodology. We experienced that this feels ‘strange’ at first, but becomes easier after some practice. In many cases we experienced it as more ‘natural’ to define methodological steps from an entity perspective.

4.3 Artifact Type Hierarchies

When modelling artifact agents, a naive solution is to provide each agent with all possible knowledge it may require during its course of operation. This, however, causes each agent to be very large and complex. Also, defining an agent becomes a clumsy and complex activity. What is required is a structuring mechanism that reduced the complexity of defining artifact types.

To handle the complexity of the artifact agents, we define *artifact type hierarchies*, which are a kind of class hierarchies. However, we do not instantiate agents from nodes in the artifact type hierarchy, but allow agents to traverse the hierarchy in the course of their operation.

In figure 4, a small example artifact hierarchy is shown. The top of the hierarchy is the entity artifact type. The entity has three subclasses, object artifact type, subsystem artifact type and class artifact type. Early during analysis, entities are identified of which it is uncertain whether they will be a class, an object, subsystem or an instance variable. When such an entity is identified, an agent is created and initialized with a methods and data part as defined by the entity artifact type. The entity artifact type contains methodological steps relevant for an entity, but also *specialise* information. When the associated preconditions hold, the agent is able to specialise itself to one of the three subtypes. This means that the current method and data specification is extended with the specification of, e.g. the subsystem type.

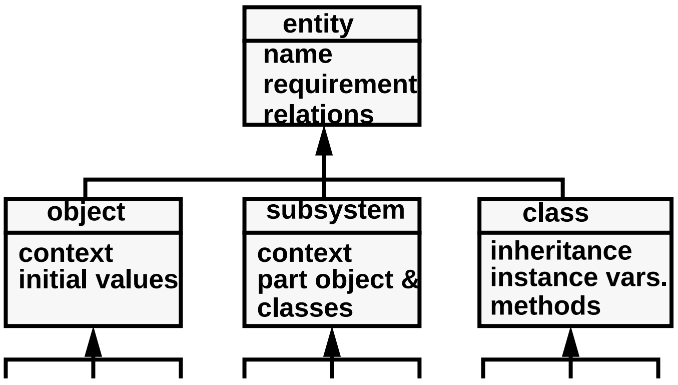

Figure 4 - simple example artifact type hierarchy

All but the top artifact type have, next to the specialise operation also a generalise operation. Generalising allows an agent to go one level up the hierarchy. This is required for two reasons. First, the software engineer might provide information to the artifact agent which later turns out to be wrong. This forces the artifact agent to go back to a state which is still valid. Second, an artifact agent, being actively collecting information, might deduce that it will, most likely, become a specific subtype. In that case will the agent specialise itself into that subtype, without consulting the software engineer. If, at a later point in time, it turns out to be a wrong conclusion, for instance because of new information became available, the agent needs to go back to the higher level.

5.0 Related work

To the best of our knowledge, most systems for providing automated support for software engineering take the conventional, functional approach. These approaches generally suffer from some or most of the problems described in section 3.

However, there is a similar work, called hermeneutic software development (HSD), described in [Aksit & van Oosten 94]. The approach described in our paper was partially inspired by the Aksit's work. Nevertheless, we believe there to be at least three differences. First, in HSD the system's representation of the methodology is defined from the software engineer's perspective, whereas we define the methodology from the artifact's perspective. Second, HSD uses the composition-filters object model to express the artifacts and methodological aspects, whereas we apply the autonomous agent paradigm. This results in a different way of domain knowledge modelling and the use of this knowledge by the agent. Thirdly, in HSD artifacts are passive objects and processes, called actors, are used to operate on the artifacts. These actors are centrally scheduled. In our approach, agents are autonomous entities with their own thread of control.

In [Bosch & Krammer 94] an approach to domain knowledge modelling in the domain of intelligent tutoring systems is described where specialisation hierarchies of programming plans are defined. The program matcher uses rewrite rules to convert a student program, which is generally expressed in very specialised programming plans, into a program expressed in higher level programming plans. We use a mechanism analogous to these rewrite rules for specialising and generalising agents.

6.0 Evaluation and Conclusion

In section 3, we identified a number of problems conventional automated environments can suffer from. As a solution to these problems, we suggest the use of autonomous agents. Each agent is the design-level representation of an operation-level entity. The agent is responsible for developing itself, i.e. the artifact part it contains, in a consistent and correct manner. Each agent contains, next to the artifact part, the methodological steps that can be performed on the artifact. Each agent has a type which can change during its lifetime. The artifact types are organised into specialisation hierarchies. Agents can traverse the specialisation hierarchy by generalising (up) or specialising (down) themselves. Conversion, or rewrite, rules are applied to convert an agent into another type.

In this section, we will evaluate the described approach with respect to the top level problems and leave the sub-problems to the reader:

- **functional approach to automated support**: the use of artifact agents is clearly an object-oriented approach, i.e. a non functional approach, to providing automated support. It does, therefore, not suffer from the problems caused by taking a functional approach to automated support.

- **lack of initiative by the system**: storing the methodological knowledge for developing an artifact at the artifact itself allows the reasoning part of the artifact agent to perform any action it has sufficient information for. Also, it can initiate a dialog with the software engineer when it is unable to collect the information from its context. Concluding, any methodological action that is modelled in the system can and will be performed by the system itself provided that the preconditions for performing this action are satisfied.
- **causal connection gap**: the artifact agent and the operation-level entity are modelled as a single object and the object exists both at the level of the software engineering environment and at the operational level. The two levels are *causally connected*, i.e. a change at one level is immediately reflected at the other level.

As a conclusion we can state that the autonomous agent approach, as described in this paper, does not suffer from the problems that we identified in section 3. We believe that inverting the methodology specification, i.e. defining it from the artifact perspective rather than from the software engineer perspective, provides a very good basis for automating the software development process. In addition, the use of artifact type specialisation hierarchies allows a modular and extensible way of formalising methodological knowledge.

In the future, we aim at implementing a prototype that allows us to experiment with the described approach. Therefore, we need to invert an object-oriented methodology, as described in section 4.2 So far, we have only inverted parts of a methodology to use as examples. As part of the methodology inversion, we also need to define the artifact type hierarchies in detail.

Acknowledgements

This work was kindly supported by professor Rune Gustavsson's *Societies of Computation* project at the University of Karlskrona/Ronneby. Some of the ideas presented in this paper developed while I worked with the TRESE and ITSSEL projects at the University of Twente. In particular the discussions with Mehmet Aksit, project leader of TRESE were very beneficial. Lennart Ohlsson, Lars Lundberg and Peter Molin gave comments on different versions of this paper. I'd like to thank the participants of the 'Artificial Intelligence for Object-Oriented Software Engineering' workshop at ECOOP'94 for their comments on this work.

Literature

[Aksit et al. 94]

M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, *Abstracting Object-Interactions Using Composition-Filters* in *Object-based Distributed Processing*, R. Guerraoui, O. Nierstrasz & M. Riveill (eds.), Lecture Notes in Computer Science series 791, Springer-Verlag, 1994.

[Aksit & Oosten 94]

M. Aksit, H. van Oosten, *Hermeneutische objectgeorienteerde software-*

ontwikkeling: ontwerp en productie in een hand, to be published in *Informatie*, 1994.

[Bosch 94]
J. Bosch, *Layered Object Model* (preliminary title), Working paper, University of Karlskrona/Ronneby, 1994.

[Bosch & Krammer 94]
J. Bosch, H.P.M. Krammer, *Modelling Domain Knowledge*, Working paper, University of Karlskrona/Ronneby & University of Twente, 1994.

[Campbell et al. 89]
R. Campbell, G. Johnston, P. Madany, *Choices, Frameworks and Refinements*, Computing Systems 5(3), 1992.

[Gustavsson & Hägg 94]
R. Gustavsson, S. Hägg, *Societies of Computation: A Framework for Computing & Communication*, Research Report 1/94, University of Karlskrona/Ronneby, Sweden, 1994.

[Shoham 93]
Y. Shoham, *Agent-oriented programming*, Artificial Intelligence, pp. 51-92, January 1993.