Mappings between Object–oriented Technology and Architecture–based Models

Peter Tabeling, Bernhard Gröne
Hasso–Plattner–Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany
{peter.tabeling, bernhard.groene}@hpi.uni-potsdam.de

Abstract

In recent publications, two prominent approaches can be found which deal with the complexity of large software systems. First, there is the object–oriented approach, where ”objects“ and ”classes“ are the prevalent abstractions. The second type of view is focused at the ”architecture“ of software systems, where a system is usually considered to be a structure of ”connectors“ and ”components“, or more general, a structure of active and passive elements.

While both approaches have their benefits, they seem to be contradictory with respect to their basic abstractions. We view this inconsistency a severe obstacle to the effective application of architecture–based methods in the context of object–oriented technology.

This paper addresses the problem by suggesting a set of alternative mappings between objects or classes and elements of architecture–based models. Criteria on choosing a mapping as well as some examples are also included in the paper.

Keywords: Object Orientation, Architecture, Modeling

1. Introduction

1.1. Object Orientation versus Architecture?

It is a widely accepted insight in academia and practice that building large software systems requires high–level abstract models. They are often named software architectures or system architectures [1] [2]. On the other hand, there are the established object–oriented methods [3] [4]. Unfortunately, there is no clear relationship between architecture–oriented models and object–oriented models. Architecture–oriented models seem to be a supplement to OO models presenting a completely different, alternative view. Some advocates of architecture–oriented approaches even state that object orientation is incompatible to the architecture–oriented view because their basic abstractions are different. In the context of software architecture, the terms ”component“ and ”connector“ typically represent fundamental elements, whereas object–oriented methods focus on objects and classes [5] [6] [7].

In our opinion, ”architecture–oriented thinking“ should not replace ”object–oriented thinking“ and vice versa. Each point of view has shown its practical relevance and usefulness. Nevertheless, it is necessary to clarify the relations between both views and the respective terminologies.

1.2. Limitations of Object–oriented Modeling

Many text books dealing with object–oriented methods promote the “seamlessness“ of object orientation: It seems that in all phases—analysis, design and implementation—the same concepts and terms can be used [8, p.2]. However, various authors question this ”seamlessness“ because the understanding of the term ”object“ strongly depends on the context and, in some cases, yields even contradicting interpretations. For example, Kaindl views objects during OO analysis and OO design as ”inherently different things“ [9]. He criticizes that the difficult transition from analysis to design models is usually described as much too simple in the literature. Habra complains about a similar ambiguity in the interpretation of the term ”class“ and says that this ambiguity is only hidden by the repetitive use of the same word—”class“ [10].

We share the point of view that there is no common and precise understanding of the term ”object“. In particular, there is no defined interpretation in the context of architecture–oriented models.

1.3. Architecture–oriented Models

There are many definitions of the term ”architecture“. Often, this term is used as synonym for ”structure“. Looking at the description of a system, for example a car, we find two types of structures: The structure of the system (the car) and the structure of the description itself (the car's blueprints). Software is a description of a system which can be interpreted by a computer. In this paper, “software architecture” refers to software structures while “system architecture” refers to the intended system being described by the software.[1]

While the terms to describe software structures are common, like e.g. ”procedure“, ”module“ or ”class“, describing system structure has no standards yet. In this paper, we will use the terminology and notation of the Fundamental Modeling Concepts (FMC) [11] [12]. While Architecture Description Languages (ADL) use the terms ”component“ and ”connector“, FMC introduces more general elements: Active components called ”agents“ which perform all operations, and passive ”storages“ or ”channels“ for storing or transmitting information.

The architecture models in this paper are *system* architecture models. Although UML class diagrams show the software structure of some examples, software architecture is not in the focus of this document.

2. Mappings between Objects and Architecture–oriented Models

In the literature, three prevalent interpretations of objects can be identified which we call ”analysis view“, ”abstract data type view“ and ”object agent view“—they will be discussed in the following sections.

While these interpretations are somewhat contradicting, each of them is useful in a certain context. Instead of seeking a ”new, integrated interpretation“ of objects, we relate the various interpretations to architecture–oriented system models, i.e. we present different alternatives to map objects to architectural elements of a system.

In addition to the three interpretations mentioned above, we present further interpretations: Low–level mappings deal with problems that can only be understood and solved when looking at the implementation and execution of an object–oriented program. High–level interpretations allow the mapping of collections of objects to high–level architectural models.

2.1. Common Mappings

2.1.1. Analysis View

A typical interpretation of objects in the context of object–oriented analysis is to view an object as an abstraction of the so–called “problem domain”.[2] This “analysis view” is of no further interest in this paper because, during the analysis phase, objects do not (yet) correspond to elements of architecture–oriented models.

2.1.2. Object Agent View

Considering an object to be an agent is a common interpretation in object–oriented design. An object agent is an active and abstract component of the system. Calling a method is interpreted as sending a message to a receiver object which carries out the desired operation and responds with an answer. Only an object agent has access to attributes, the data associated with an object. Methods describe which messages the object can handle, what operations on its data it can perform and which messages it sends to other objects [13, p.6].

A typical example for this view in context of architecture is a dispatcher. GUI toolkits usually use a dispatcher object to react to GUI events like mouse movement to or from a defined area, or button press or release events. The dispatcher doesn't process the event itself but calls a method of a certain object which has been registered as handler for this event.

The block diagram in figure 1 views objects as agents consisting of methods and an internal storage holding object data. Only the object's methods have access to the internal storage. 'Knowing' other objects by their object ID (reference) is symbolized by a channel between them used to send messages. A direction symbol indicates which object sends a message and which receives.

1 Michael Jackson writes: ”Software is a description of a machine. We build the machine by describing it and presenting our description to a general–purpose computer that then takes the attributes and behavior of the machine we have described“ [14].

2 See [3, p.39]: “Object–oriented analysis is a method that examines the requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.”

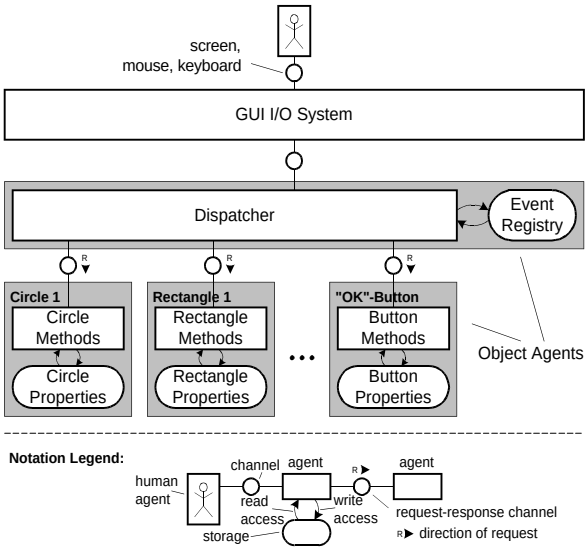

Figure 1 Object Agent View: A dispatcher (Block diagram)

This view is charming because of its vivid association with active system components responsible for certain tasks and exchanging messages. Unfortunately, the usefulness of this view scales down the same way the size of the total system scales up: Given many objects of different type, this view would result in an extremely fine granular system architecture consisting of a myriad of object agents—an ”anthill architecture“. Furthermore, although this view implies that object agents can operate concurrently, there is a strict sequence resulting from the order of method procedure calls. Introducing true concurrency with threads results in severe comprehension problems—for example, no encapsulation of object data exists between two threads, and two threads can execute the same method procedure with the same object. To avoid inconsistency, additional mechanisms like a lock management have to be used.

2.1.3. Abstract Data Type View

Another common interpretation can be called the ”abstract data type view“. Here, an object is seen as a storage for an abstract data type which is described by the corresponding class. Bertrand Meyer presents this interpretation of objects—he defines a class as ”an abstract data type equipped with a possibly partial implementation“ [4, p.142]. The class not only lists the operation types (i.e. the method signatures) of the abstract data type, it may also provide the implementation of the data storage (i.e. the attribute storages) and the implementation of the operation types (i.e. the method bodies). From this point of view, an object can be seen as a passive system component (a storage), and a method call can be interpreted as an operation which is performed on the object—rather than a message causing the object to perform an operation. The internal structure of the storages is hidden—the model does not reflect knowledge about the implementation of a data type.

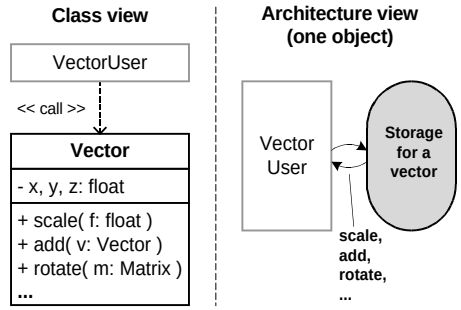

Figure 2 Abstract Data Type view (Class diagram / Block diagram)

Figure 2 shows a typical example of this type of view. The ”Vector“ class defines an abstract data type ”Vector“ with operation types ”scale“, ”add“, ”rotate“ etc. The class also provides the implementation of the Vector data and the operation types. From an abstract point of view, an instance of the ”Vector“ class can be seen as a storage which holds a certain ”Vector“ value. Other system components can operate on this storage, thereby being restricted to the operation types defined for the abstract data type.

If we would derive an architectural model from this view, the system would solely consist of a collection of storages (i.e. objects) for abstract data types. Since only storages with related operations have been specified, there must also be some active component doing these operations. Therefore, we can only assume a generic component, called ”the system“, which performs all operations on the data. This system ”architecture“ does not foster system understanding, because is too primitive and generic.

In case of concurrent systems, the abstract data type view shows further limitations. The fact that an abstract operation is actually implemented by several operations on object attribute data leads to inconsistency problems which, in turn, can only be discussed on the level of threads and their accesses to attribute data. Unfortunately, the abstract data type view hides these implementation details—which makes it impossible to understand problems of concurrent accesses and discuss their possible solutions.

2.2. Low–level Mappings

The views described in section 2.1 mirror the idea of encapsulation: Either the non–public attribute storages are local to the object agent or—in case of the abstract data type view—they are not shown at all. However, there are situations where lower level models of objects are needed which explicitly show the inner details of an object.

2.2.1. Data Record View

Taking a closer look on how an object–oriented program is executed, you soon learn that an object is just a data record in memory. This view—which we call the ”data record view“—explicitly shows the ”inner structure“ of objects, i.e. an object is described as a set of storages for attribute data. In case of the ”Vector“ example (see above), the storages for the vector components x, y and z now become visible—see figure 3. The implicit ”system“

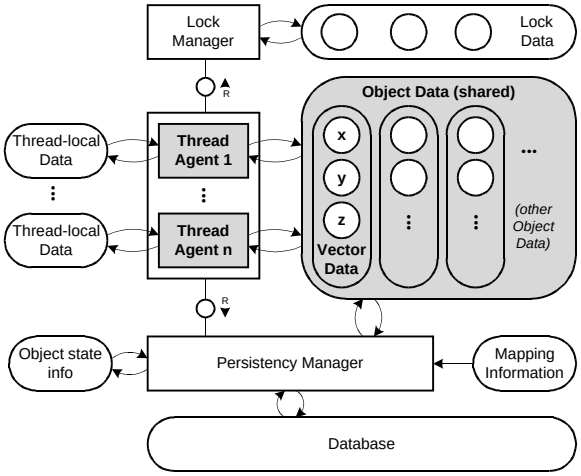

Figure 3 Data Record View (Block diagram)

agent of the abstract data type view is also described in more detail—each thread is modeled as an agent operating on the (shared) objects. Each thread also has a private storage for its state, e.g. for local variables.

In contrast to the abstract data type view (see above), the discussion of inconsistency problems in the context of multithreading is now possible. The architecture model makes clear that object data is shared between threads—see figure 3. In a sense, multithreading breaks encapsulation because the implementation of methods and attributes causes effects (i.e. inconsistencies) beyond the scope of an object. In order to retain (or reestablish) encapsulation, thread's operations on object data must be synchronized, e.g. by using locks. Thread agents can set and release locks by sending corresponding requests to a central lock manager (e.g. an operating system's mutex service).

This view also provides an elegant way to describe object persistency. A persistency manager can access object data directly, for example, to increase performance. The persistency manager typically needs additional information for the mapping between attributes and database fields—see figure 3 below.

2.2.2. Processor View

The processor view refines the thread agents of the data record view:

A program thread is implemented by a virtual processor executing the code of the program described by the classes. The thread–local data consists of the Program Counter (PC), the stack for parameters and local variables and other data. Each virtual processor has read access to the code in the program storage. In this view we show the code a compiler generates from an OO program and the (virtual) machine which executes it. For example, we see that the code provides a (default) constructor procedure for every class which reserves memory for a new object data record and initializes the fields of the record.

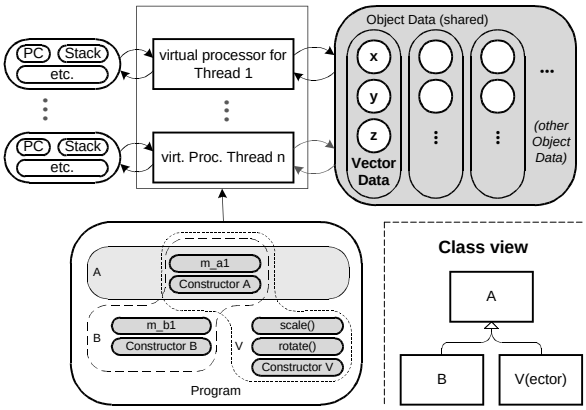

Figure 4 Processor View (Block diagram / Class diagram)

2.3. High–level Mappings

The views presented above allow objects to be mapped to certain elements of architectural system models. While these mappings foster a more intuitive understanding of a single object in a certain context, they do not solve the ”granularity problem“: Reflecting each object as a dedicated element in an architectural model still yields models of (too) extreme granularity—larger systems

with thousands or millions of objects cannot be modeled this way. Furthermore, such models would only present short–lived snapshots, because objects are created and destroyed very frequently. Hence, mappings are needed where a (dynamic) collection of objects can be mapped to a (static) high–level architectural element.

2.3.1. High–level Abstract Data Type View

A quite obvious way to model a set of objects is to extend the idea of the abstract data type view. There are cases where the definition of a single class is not sufficient for the realization of an abstract data type. For example, a data type ”tree“ could be needed which stores arithmetic expressions as binary trees. For this purpose, one might define the classes as shown in figure 5. Instances of classes derived from ”Node“ would represent a tree's nodes and a ”Tree“ object would be the placeholder of the whole tree. This object provides methods to access the whole object structure, such as calculating the value of the tree (i.e. the value of the corresponding expression).

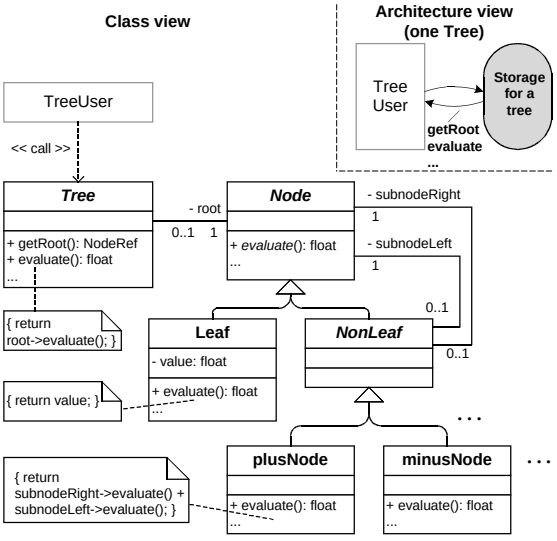

Figure 5 High–Level Abstract Data Type View (Class diagram / Block diagram)

The basic idea behind the tree–related classes is to provide the possibility to store trees. Hence, at a higher level, the complete object set holding a certain tree can be viewed as a single storage for an abstract data type ”Tree“—see upper right corner of figure 5. This ”high–level abstract data type view“ yields a single storage as an abstract, compact model of an object set. This model remains valid even if the underlying object structure changes over time—at the higher level, only the current value (i.e. the tree) in the storage is changed. The operation types defined for the abstract data type (e.g. tree evaluation) are not implemented by a single method but the combined methods of several classes (here: the ”evaluate“ methods).

2.3.2. High–level Object Agent View

It is sometimes reasonable to combine many objects to one agent. Take for example a persistency service which consists of a singleton object of the persistency manager class and a set of class agents, one singleton object for each persistent class.

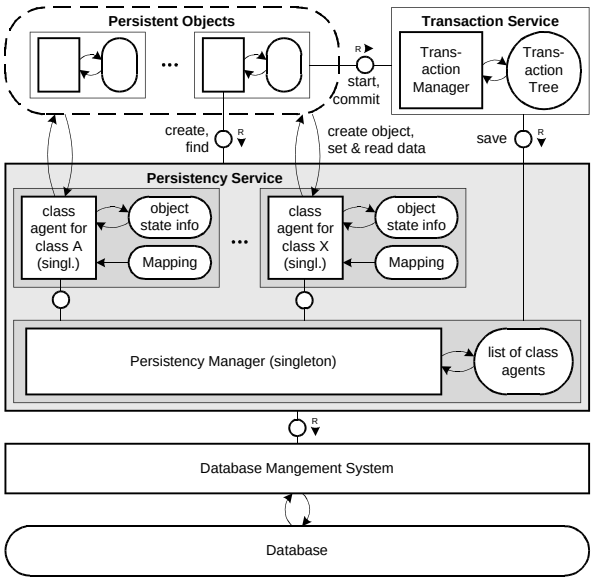

Figure 6 High-level Object Agent View: Persistency Service (Block diagram)

Figure 6 shows the persistent objects, the persistency service, the transaction service and the database. The task of the persistency service is to create or load persistent objects from the database and to save altered object data whenever the transaction service requests it in case of a commit. The persistency *manager* just keeps a list of the classes, while the class agents read and write object data and create objects; they have all information about their class and know which objects have been altered. Class agents are therefore factories for persistent objects and part of the persistency service. The persistency manager is the representative of the persistency service. The decomposition of the transaction service is not shown here—for example, the tree of the (nested) transactions can be implemented with objects of a transaction class managed by a singleton object of a transaction manager class.

This architecture has been chosen for the project ”Object Services“ at SAP in 1999. The goal was a new persistency and transaction service for ABAP Objects, an OO extension of the R/3 programming language ABAP. The services had to fit as seamless as possible into the R/3 framework and cooperate with existing non–OO applications [15].

2.3.3. Functional View

The high–level views presented above have in common that one architectural element (storage or agent) is mapped to many objects. However, this one–to–many mapping is not appropriate if the architectural model primarily represents a functional decomposition. Figure 7 shows an architectural model of a simple graphic editor where each component provides a certain functionality, namely editing, displaying, printing and persistency. The central storage holds all data describing the graphic drawing currently being modified. The agents rely on certain components of the underlying platform, e.g. the file system.

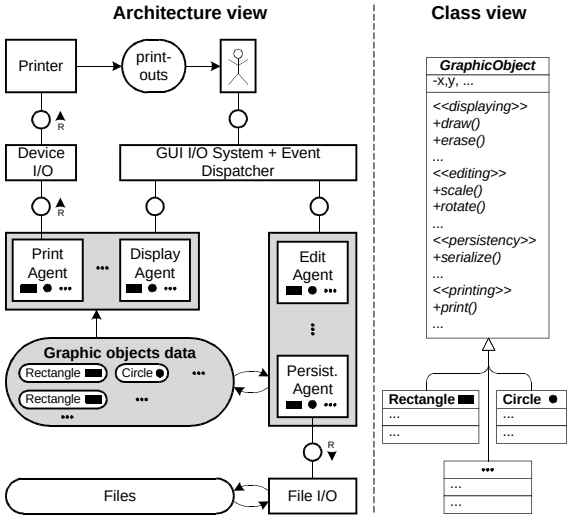

Figure 7 Functional View (Block diagram / Class diagram)

While this model is very useful for presenting a system overview, a one–to–one or one–to–many mapping of architectural elements to objects would not result in an appropriate software structure. Following object–oriented design principles, we should define a class ”GraphicObject“ with subclasses for the different types of graphical elements, i.e. rectangles, circles etc.—see class view in figure 7. The implementation of displaying, editing and other operations depends on the implementation of the graphic element data. Hence, each of the classes should not only define a storage format for graphic data but also contain methods corresponding to the various operation types. (Of course, additional classes had to be defined beside these classes.)

The example shows a many–to–many mapping of objects to architectural elements. The collective object attribute data is mapped to the ”Graphic objects data“ storage, and all methods implementing a certain functionality (e.g. editing methods) are mapped to a corresponding agent (e.g. the ”Edit Agent“). We call this mapping the ”functional view“ because the architectural model reflects the functionality provided by the program.

2.4. Guidelines for Choosing a View

Each of the different views presented above is suitable depending on the context and the type of object(s) under consideration. Hence, some criteria for choosing one of these views should be given:

The object agent view (sections 2.1.2 and 2.3.2) is suitable if an object's main purpose is controlling other parts of the system. In this case, the object can be understood best as an active system component which communicates with other components by sending requests to them. Such objects are often singletons and have a method which runs (potentially) endless (e.g. the dispatcher's loop in section 2.1.2). If several objects closely interact to fulfill a control function, they can be modeled as one controlling agent, according to the high–level object agent view.

The abstract data type view (sections 2.1.3 and 2.3.1) should be chosen if the main idea behind a class is to provide a data type which is missing in the programming language (e.g. the vector type in 2.1.3). In this case, methods implement operations which are typically defined with purely operational semantics, e.g. by defining pre– and post–conditions. If a set of objects is used to implement a data structure, the high–level abstract data type view is a good way to model this.

In general, the data record view (section 2.2.1) is helpful if the implementation details of methods and attributes cannot be ignored and objects ”must be seen as data records“. Multithreading with shared objects is one example.

If aspects of code execution have to be described, the processor view (section 2.2.2) yields a helpful model. It is also useful to explain dynamic code management like ”class loading“ in Java or object migration between processor nodes.

The functional view (section 2.3.3) is valuable to keep the overview even if the system becomes very complex. For example, the functional view helps to check if all requirements (especially functionality) have been taken into account, it shows how the components interact with each other and the environment, it shows what kind of data is stored and which agents have access to it, etc. In contrast, an pure OO model like a class diagram would not offer this insight because data definitions and functionality are spread across classes, and communication links (channels) between agents are only implicitly given by certain method calls. The functional view is also useful whenever non–OO or even non–software components are involved in the system.

3. Conclusion and Future Work

We presented several approaches to interpret objects in the context of architecture–oriented models. These mappings do not only foster a better, more intuitive understanding of objects. They also span a conceptual bridge between the object–oriented and the architecture–oriented view of software systems.

While we could give some criteria for the application of the views, we do not think that the mapping between object–oriented models and architecture–oriented models is always straightforward. It is a non–formal, creative task which requires experience. Further research should support this task, for example, by conserving these experiences as architecture patterns. Furthermore, additional views on objects are possible. For example, objects can be viewed as values or (broadcast) channels.

The mappings presented in this paper can be used both during system analysis or reengineering and during system construction. In the context of reengineering, the views help deriving high–level architectural models from an existing object–oriented code base. During system construction, conceptual architecture models can be used as a basis for object–oriented design (see [15] or [16]). Here, further research is necessary to establish a corresponding, refined development process.

References

[1] Hofmeister, C.; Nord, R.; Soni, D.: Applied Software Architecture. Addison Wesley Longman, 2000

[2] Bass, L.; Clements, P.; Kazman, R.: Software Architecture in Practice. Addison Wesley Longman, 1998

[3] Booch, G.: Object–oriented Analysis and Design with Applications. Benjamin / Cummings, Redwood City, 1994

[4] Meyer, B: Object–Oriented Software Construction, 2nd Ed.. Prentice Hall, 1997

[5] Shaw, M. et al.: Abstractions for Software Architecture and Tools to Support Them. Computer Science Dept. Carnegie Mellon University, Pittsburg PA, March 1995

[6] Medvidovic, N.; Taylor, R.: A Classification and Comparision Framework for Software Description Languages. Dept. of Information and Computer Science, University of California, Irvine , 1999

[7] Clements, P.: A Survey of Architecture Description Languages. IEEE Intl. Workshop on Software Specification and Design, 1996

[8] Balzert, H.: Lehrbuch der Objektmodellierung. Spektrum Akademischer Verlag, Heidelberg, Berlin, 1999

[9] Kaindl, H.: Difficulties in the Transition from OO Analysis to Design. IEEE Software September / October , 1999

[10] Habra, N.: Separation of Concerns in Software Engineering Education. Institut d'Informatique, University of Namur, Namur, Belgium, 1999

[11] Keller, F.; Tabeling, P. et al: Improving Knowledge Transfer at the Architectural Level: Concepts and Notations. Intl. Conference on Software Engineering Research and Practice (SERP), Las Vegas, 2002

[12] Wendt, S. et al: The Fundamental Modelling Concepts Home Page, http: / / fmc.hpi.uni–potsdam.de /

[13] Goldberg, A.; Robson, D.: Smalltalk–80: The Language. Addison Wesley, 1989

[14] Jackson, M.: The World and the Machine. ICSE, Seattle 1995

[15] Gröne, B.; Knöpfel, A.; Langhauser, J.; Krane, R.: Objects Services für R / 3 Release 99 – Konzepte. SAP AG Walldorf, 1999

[16] Kappel, M.: Entwurf und Implementierung eines Simulators für Register-Transfer-Netze. Diploma Thesis, University of Kaiserslautern 1995