A Framework Supporting the Utilization of Domain Knowledge Embedded in Software

Eran Rubin and Yair Wand

Sauder School of Business, The University of British Columbia, Vancouver, B.C. Canada

{eran.rubin, yair.wand}@sauder.ubc.ca

Abstract

Software systems embed in them knowledge about the domain in which they operate. However, this knowledge is "latent". Making such knowledge accessible could be of great value to the organization both as a source of explicit knowledge and to systems development and maintenance. We propose a framework aimed at making domain knowledge embedded in software explicit. The framework is based on identifying domain knowledge acquired during the development process (especially in requirements analysis) and formalizing it. The software architecture is then partitioned into two parts: one represents the domain knowledge and the other responsible for the actual processing (using this knowledge). A specific object-oriented design approach is suggested to accomplish this partitioning.

Keywords: Software Architecture, Domain Knowledge, Requirements Engineering, Conceptual Models.

1 Introduction

Software engineering is a knowledge intensive process (Chau et al. 2003), in which the output is software code. The resulting Information System (IS) embeds much knowledge about the domain of discourse. A significant portion of the knowledge required for this process is gathered during requirements analysis, which is an important part of Systems Analysis and Design (SA&D). This knowledge is used in the design phase to shape the implemented system.

This paper is based on the premise that significant domain knowledge is embedded in software code (Rubin and Wand 2005). If this knowledge could be made explicit, it could serve as a valuable resource for the organization using the system.

Knowledge embedded in software, not only presents a usually untapped resource for the organization, but also has a significant advantage. As software is changed to reflect changing requirements, so does the embedded knowledge. Thus, when software is modified to incorporate new requirements, it automatically embeds new knowledge. This makes software a potential up to date and reliable source for knowledge. This is in contrast to domain knowledge available in other artefacts generated during system development (such as various documents and models). Such artefacts are not necessarily subject to the strict updating discipline needed to maintain software that complies with the requirements.

2 Related Research

The framework proposed in this paper views software as a potential *source for domain knowledge*. As such, software is viewed as an artefact that can support organizational processes via knowledge embedded in it, in addition to the functionalities provided by the software. This is fundamentally different from common software-development approaches that usually focus on ways to *deploy domain knowledge* in the developed software.

Common approaches to capturing domain knowledge during system development usually apply software modelling techniques (notably UML) and supporting CASE tools. In general, the use of such techniques is an attempt to abstract away from a specific programming languages or platforms of the target system (Smith 2004). Tools typically enable drawing diagrams or creating a connection between the models and the code typically (Fowler 2003). Some tools provide code generation. However, rather than facilitate extraction of domain knowledge from code, these tools are aimed to support abstraction of the system.

Model Driven Architectures (MDA) is another line of work related to our objectives. MDA is a framework for software development defined by the Object Management Group (OMG). Key to MDA is the importance of models in the software development process. Within MDA software development is driven by the activity of modelling the software system (Kleppe et al. 2003).

The MDA development life cycle is not very different from the traditional life cycle. The major differences lie in the nature of the artefacts that are created during the development process. These artefacts are formal models, i.e., models that can be processed by computers (Kleppe et al. 2003). Central to MDA is the notion of creating different models at different levels of abstraction and then linking them together to form an implementation. A mapping between models is assumed to take one or more models as its input and produce a single output model. Some of these models will be independent of software platforms, while others will be specific to particular platforms. Each model is expressed using a combination

of text and multiple complementary and interrelated diagrams (Mellor et al. 2004).

However, while MDA is expected to take part in all stages of the SA&D process, it has been practically applied only on transformations between the logical design and the implementation code. This likely is an outcome of the difficulties in providing automatic transformation between conceptual models and logical design models. The gap between the two is simply too wide. Conceptual models are created for understanding and communicating about the domain of discourse. On the other hand, logical designs incorporate a large amount of artefacts associated with implementation details such as performance and portability. Enabling automatic transformation between the two, a core motivator of MDA, is a highly complex, if not an impossible task.

2.1 Documentation of Domain Knowledge

The development process incorporates the creation of documentation artefacts which represent knowledge (Abrahamsson et al. 2002). In fact, the literature often refers to the development process as a document creation process (Laitinen 1996, Welsh and Hann 1994). These documents include requirement statements, models, interviews, and more. In particular, models serve as a bridge between the analysis and the design phases (Paetch et al. 2003). In systems analysis, domain understanding is often developed and documented via the use of *conceptual models* (Dieste et al. 2003). Such models can facilitate communication among various stakeholders in the organization (Mylopoulos et al. 1997). Conceptual models can help make real-world concepts and relationships tangible (Motschnig-Pitrik 1993), and can emphasize aspects of reality that are important for the development purpose and downgrade unimportant elements (Borgida 2004).

It has often been claimed that systems evolve in response to changing needs but that the design documents fail to reflect these changes (Clarke et al. 1999, Bennett et al. 1991, Laitinen 1996). Since maintenance and update constitute the majority of a system's lifecycle (Clarke et al. 1999), this failure has accounted for much of the criticism of traditional development processes. The process was claimed to be too cumbersome to provide limited benefits (Fowler 2004). This situation has led to the emergence of agile development methods. However, agile methods have been criticized for avoiding documentation, which is needed for knowledge sharing and for reducing knowledge loss (e.g. when team members become unavailable) (Abrahamsson et al. 2002). While traditional development methods over-document, a weakness of agile methods is not generating sufficient documentation (Paetch et al. 2003).

The issues raised above indicate that documentation artefacts are an important source for sharing knowledge and facilitating communication. However, without a special effort they might "decay" and lose relevance as systems are modified. Hence, software documentation might not be an effective way to preserve knowledge. In contrast, source code stays up-to-date as systems change. However, in its *current* form source code is not effective as a knowledge documentation scheme. We discuss these issues in more depth in the next subsection.

2.2 Design versus Implementation

Traditionally, the design process has been considered to gradually move from the "problem space" to the "solution space" (Booch 1987). In this context, the first step of the implementation phase, the program composition, can also be perceived as the final design product. As such, programs are traditionally considered to be defined entirely in terms of solution space. However, as different programming paradigms emerged, the boundary between the problem and solution domains has blurred (Henderson-Sellers and Edward 1990). Initially, programs did not exhibit problem level information. It was not formally required that a good programmer would make a program understandable and modifiable. Rather, programs were evaluated merely on their efficiency and on being an accurate solution to a specific problem (Abbot 1987). Over time, it has been recognized that programs should be designed and written to be understandable and maintainable. An important step in this direction was the introduction of Object Oriented Programming (OOP). OOP moved programs much closer to the problem domain. Indeed, it is claimed that the success of OOP is not necessarily due to its technical advantage, but more due to enabling programmers to think in terms of the problem domain (Abbot 1987).

The more source code related to the problem space, the more difficult it was to distinguish between analysis, design and the implementation. Different views appeared regarding the difference between the output of Object Oriented Analysis (OOA) and the output of Object Oriented Design (OOD) (e.g. Jacobson 1995, Kaindl 1999, Rumbaugh et al. 1991, Shlaer and Mellor 1992). Some authors (Coad and Yourdon 1991, Jacobson 1995) pointed out that OOA refers only to domain objects, while OOD refers also to future system objects. Yet other authors did not clearly distinguish between the two aspects (Rumbaugh et al. 1991, Shlaer and Mellor 1992). This state of affairs led to the prevailing perception that code is a design artefact. Statements such as "A program is not an object but rather it as a design of an object" (Blum 1996) or "the final source code is the real software design" (Reeves 1992) become increasingly prevalent. While these arguments have merit, it currently cannot be claimed the source code and other design models created during the development process are equivalent. Two major differences exist between the two:

1. The source code is not as accessible to different stakeholders in the organization as design documents are. Basically, only programmers can understand source code. Moreover, familiarity with the development environment and the programming language syntax is needed in order to navigate through source code.
2. The source code contains much more information than included in the intermediate design documents. While the code embeds much of the information available in design documents, it also incorporates implementation

related details. Examples are handles to system objects (e.g. files and devices), algorithmic details (e.g. how an algorithm is accelerated for performance reasons), networking details (e.g. use of ports and socket information), and user interface details (e.g. the pixel location of a specific button on a screen).

In general, the design documents and the system source code are highly related. However, fundamental properties, associated with the structure and composition of code, render source code and design documents two distinct artefacts. This situation impedes the use of code as an explicit source for the knowledge embedded in it.

3 The Proposed Framework

3.1 Fundamental Concepts

Our framework in intended to provide explicit representation of domain knowledge gathered in Requirement Engineering (RE) and in related Conceptual or Enterprise Modeling (EM). We have analyzed various studies in RE and EM to extract the core constructs employed and used a comprehensive ontology of enterprise-related constructs (Uschold et al. 1998). We have identified seven fundamental concepts commonly used in modelling application domains: *Actors, Roles, Resources, Services, Goals, Constraints*, and *States (Consisting of Attributes)*. We now define each of those briefly:

***Actor*:** An active entity in the organization. An actor can either be a person, an organizational unit, or a machine which performs some activity (Uschold et al. 1998).

***Role*:** A role implies a set of services and attributes an actor provides. A specific role is a way an actor is viewed by other actors in the organization. Relationships (interactions) among actors may imply a view (activity or cognition) for one of them by others. This view indicates the actor has an *actor role* (Uschold et al. 1998).

***Resource*:** a passive organizational entity which is used by actors to provide services. Resources are either consumed entities, or entities used during activities (Uschold et al. 1998).

***Service*:** An operation that can be performed by an actor in order to help other actors perform services or accomplish goals.

***State*:** "a situation that can be thought of as holding or being true" (Uschold et al. 1998). A state of an actor or resource reflects its situation and is represented by *attributes*.

***Goal*:** a state or a set of states which an actor has desire to arrive at. "A goal (achieve) is the realization of a state (state of affairs)" (Uschold et al. 1998).

Constraint: An assertion on how the domain behaves which defines which states and actions are allowed in the domain. Constraints defining actions can be expressed in terms of stable and unstable states (Wand 1989). Unstable states indicate the conditions for triggering changes (by executing services). Service execution changes an unstable to a stable state (Wand 1989). Unstable and stable states are akin to preconditions and post-conditions, respectively (Uschold et al. 1998). Together the mapping from unstable to stable states is termed a (transition) law (Wand 1989). Laws can be considered abstractions of services.

3.2 The Architecture

The proposed architecture comprises two separate subsystems: the Conceptual Subsystem (CS), representing the domain knowledge, and the Processing Subsystem (PS). The Conceptual Subsystem (CS) includes the knowledge represented in terms of the constructs listed above. More specifically, it includes elements such as specific roles, actors, resources, and laws. The CS does not include other details (such as data type and structure information, input/output operations, code associated with device interaction, and user interface aspects). The PS, on the other hand, incorporates these aspects. However, the PS does not represent explicitly conceptual knowledge available at the CS. Rather the PS interacts with the CS to use the knowledge available in it.

In the following, we demonstrate the principles underlying the proposed framework via an example. To do this, we first present the example using a "standard" architecture, and then reintroduce the example using the framework.

4 Applying the Framework

We claim that current development practices hinder explicit representation of domain knowledge in code. As our literature review suggests, an implicit assumption of most current practices is that technical considerations increasingly shape the evolving design. The shift of focus to implementation details leads to a gradual loss of the conceptual knowledge. This knowledge, well reflected in earlier stages of development, becomes intertwined with programming details and is not represented explicitly in the final implementation. Thus, we believe a different design approach is needed.

Our framework suggests that both conceptual models and implementation models should evolve throughout the development process in parallel, rather than in sequence (Rubin and Wand 2005). These models should remain separate while influencing each other in well defined ways. In the final implementation, the conceptual models would be independently represented in code. This way, conceptual models can remain free of implementation concerns while the implementation design would stay synchronized with them.

4.1 The Traditional Implementation

To illustrate how traditional implementations may embed domain knowledge, we first describe the traditional development of a Customer Relationship Management System (CRM). Using this system, operators handle customer requests. A conceptual model using UML diagrams and showing the roles of customers and company personnel is depicted in Figures 1 and 2. The domain consists of customers and valuable customers - a

special kind of customers eligible for discounts (limited by a *Max Eligible Discount*). As well, the domain consists of employees and a special kind of employees-managers, who can allow discounts (limited by a *Max Discount*). Finally, managers serve valuable customers while employees serve regular customers. The model also reflects a service provided by all employees – *ProcessRequest* - by which employees process a customer request. Additional domain services include *RouteRequest* and *ConnectToEmployee*. *RouteRequest* is the service by which an Operator forwards customer requests to a proper destination. *ConnectToEmployee* is a service provided by an operator to connect a customer and an employee by phone. The conceptual model also defines the context of services. Conceptualizations of *RouteRequest* and *ProcessRequest* are shown in Figure 2. This figure depicts knowledge about conditions for providing services and the state of entities upon service completion is made clear. A phone call triggers the *Route Request* action. Routing is complete when an available employee is found, triggering the *process request* service. The available employee serves a customer until the customer is in no need for additional services. Completions of these services are indicated by two constraints, "customer forwarded" and "customer offline" respectively.

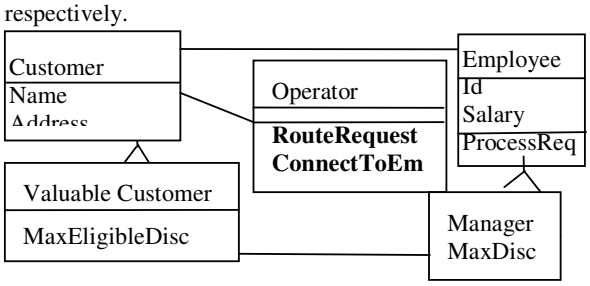

Figure 1: Conceptual Model of the CRM System Domain

The conceptual models can help generate system requirements. Some examples for possible system requirements are listed in table 1.

Req.	Description
1	Valuable customers should be handled only by managers who can apply discounts these customers are eligible for.
2	The request handling service, currently provided by operators, should be fully automated by the system.
3	An employee can handle a maximum of 40 requests a day.
4	The line-connecting service provided by operators would preferably be automated by the system.

Table 1: Part of the Requirements Document for the CRM System

We further use this example to illustrate how new implementation concerns emerging at the logical design phase, might obscure the conceptual knowledge. First, we notice that new implementation artefacts are introduced. For example, the employee class now has a new procedure *UpdateCustomerQueue*, which has no meaning at the conceptual level. Moreover, new classes emerge: *RequestHandler* and *TaskManager*. The purpose of *RequestHandler* is to handle incoming calls. It implements the *RouteRequest* service associated with Operators. Related to this, in this example, imagine that due to development time limitations it was decided not to implement requirement 4, which had a low priority. From an implementation point of view, the introduction of the *RequestHandler* object eliminates the need for an explicit representation of the *Operator* role. Moreover, since the task of handling incoming calls is automated it will have many programming related details embedded in methods implementing it.

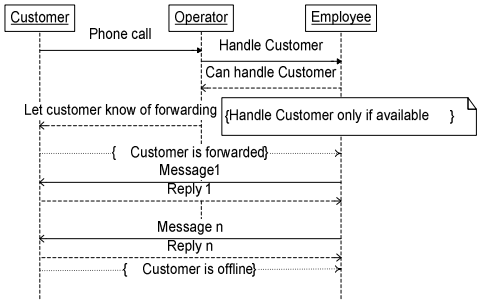

Figure 2: Conceptual model of *Route Request* and *Process Request* services

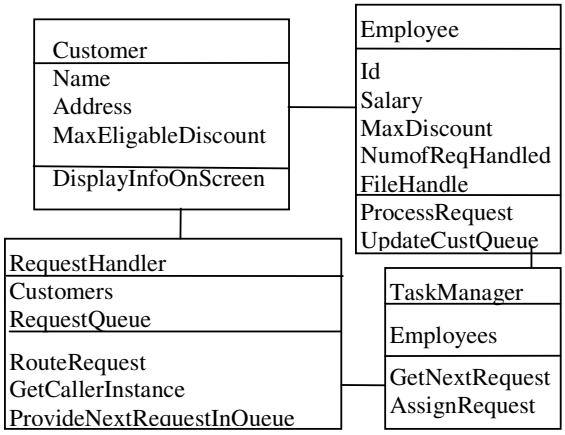

Figure 3: Part of the Implementation design of the CRMS

Nonetheless, the definitions of customers, valuable customers, employees, managers, as well as the fact that *RequestHandler* incorporates a service that relates to the role of an operator, are part of the domain knowledge. Moreover, the conceptual meaning of the services, as illustrate in figure 2, is part of the domain knowledge.

4.2 Design under the Proposed Framework

In this section we illustrate the principles of our framework by outlining the way the exemplified system would be designed under it.

Under our framework the conceptual model will be codified in final implementation. Conceptual elements will be kept only at the conceptual model and the processing model will interact with these elements. Thus, we will now alter the logical model constructed in the pervious section, to become the processing model of our implementation architecture. Similarly, the conceptual model will be preserved and updated to qualify as a conceptual model in our implementation architecture.

A comparison of the traditional logical class diagram (Figure 3) and the conceptual class diagram (Figure 1) reveals that *Employee* and *Customer* conceptual roles appear both at the conceptual model and the processing model. It also appears that some properties associated with the conceptual roles *Manager* and *Valuable customer* are included in both (those are *MaxDiscount* and *MaxEligableDiscount*). Under our framework these elements must be represented only in the conceptual model. However, in the traditional logical model design, the objects *Customer* and *Employee* incorporate new elements, which do not appear at the conceptual model. These include the *FileHandle* and *NumOfRequestHandled* attributes of *Employee* and the services *DisplayInfoOnScreen* and *UpdateCustomerQueue* of the *Customer* and *Employee* objects, respectively. Under our framework, these new elements are handled in one of two ways – if these elements are meaningless outside the implemented system they should only be included in the processing model. However, if these elements are meaningful at the conceptual level, they should be conveyed in the conceptual model. In our case, of the above elements, the attribute *NumOfRequestHandled* has a conceptual meaning, while the rest are implementation oriented. Hence, the new attribute *Num of Req. Handled* was added to *employee* at the conceptual level to fulfil requirement 3 (Table 1). Also, since requirement 4 was not implemented there is no longer need for representing the service *ConnectToEmployee* of the *Operator* role in the conceptual model. Based on the above, in the new architecture the conceptual and processing models will assume the forms shown in Figures 4 and 5.

Notably, a new implementation class, *DomainElements*, is introduced at the processing model. This class facilitates access to domain related instances and the handling of events associated with the changes of these instances. Hence, the class *DomainElements* is used by processing objects that relate to *conceptual* elements. These objects incorporate *DomainElements* through a new attribute which they posses – *RelatedConceptualElements*. In the example, the processing model includes this attribute in the *Customer*, *Employee*, and *Request Handler* classes. In particular, *Customer* objects relate (via *DomainElements*) to the domain roles *ValuableCustomer* or *Customer*; *Employee* objects relate to the domain roles *Manager* or *Employee*; and *RequestHandler* relates to the domain role *Operator* as it implements the service *RouteRequest*, defined in the conceptual role *Operator*.

We now briefly explain how processing objects use information available as conceptual elements. In our framework, conceptual services are defined at the conceptual model (subsystem) while the processing related to these services is done at the processing subsystem. The need to execute a service is triggered by the conceptual subsystem via "command" events sent to the processing objects. In figure 5, the method dedicated to handling such events is *HandleDomEvent*, which is part of the *DomainElements* class.

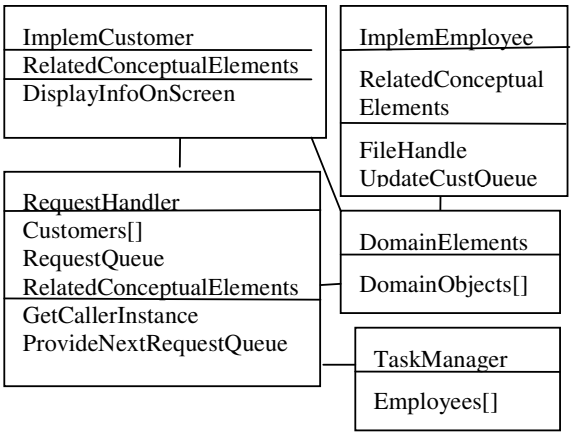

Figure 4: CRM Final Implementation Design: The Processing Subsystem Model.

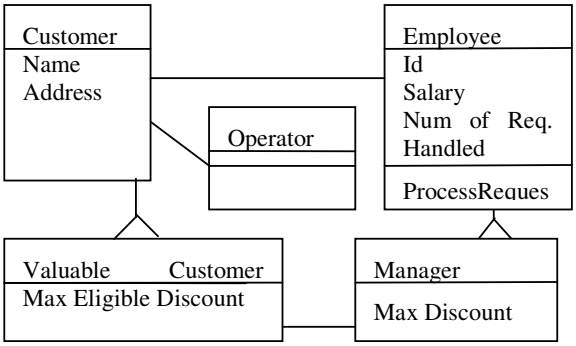

Figure 5: CRM Final Implementation Design: The Conceptual Subsystem Model.

Conceptual Activity	**Unstable State**	**Stable State**
Operator-RouteRequest	Incoming phone request is present	Last handled incoming call was forwarded.
Employee-ProcessRequest	Customer online	Customer offline
Operator-CheckAvailability	Asked employee to handle request	Employee provided answer

Table 2: Service Representation at the Conceptual Subsystem.

We also briefly illustrate the way domain knowledge of services is separated in our framework from their implementation. The conceptual specification of services in terms of service laws (Wand 1989) is kept at the conceptual subsystem. As previously stated, service laws

indicate when a domain object assumes an unstable state and the expected stable state upon completion. In our example, the service laws for the *RouteRequest* and *ProcessRequest* services are given in Table 2 (which exhibits the conceptual representation of services). The information in this table is represented at the conceptual subsystem but the actual processing statements related to it are coded in the processing subsystem. Once an unstable state, as documented in the second column of the table, is reached, a "command" event is sent to the processing subsystem. After processing, a "completion" event is sent from the processing subsystem. Note, the definition of stable state condition is used to confirm that the system has performed properly.

5 Summery

We suggest that since implementation code embeds domain knowledge, and is regularly updated, it can serve as a useful source of knowledge. In this paper we outlined a framework intended to enable the utilization of this knowledge. Under the proposed framework the conceptual knowledge should remain separated from the processing knowledge throughout the system development process and, most importantly, in the resulting code.

We have suggested the basic concepts needed to formalize knowledge embedded in software and demonstrated the ideas using a small scale example.

The contribution of this work is in viewing software systems as potential sources of explicit domain knowledge. We further provide a framework and implementation architecture to demonstrate how the idea can be applied. This is relevant to both software development process and to knowledge management.

References

Abbot, R.J. (1987): Knowledge Abstraction. Communications of the ACM, 30:8,664-671

Abrahamsson, P., Salo ,O., Rankainen, J. and Warsta, J. (2002): Agile Software Development Methods - Review and Analysis. VTT Electronics

Bennett, K., Cornelius, B., Munro. M. and Robson, D. (1991): Software Maintenance. In: McDermid, J. A., ed. Software Engineer's Reference Book, chapter 20. Butterworlh-Heinemann, Oxford, England

Blum, B.I. (1996): Beyond Programming, to a New Era of Design, Oxford University Press, NY

Booch, G. (1987): Software Engineering with Ada. Benjamin-Cummings.

Borgida, A. (1991): Knowledge Representation, Semantic Modeling: Similarities and Differences. In: Kangasal, H (ed.), Entity Relationship Approach: The Core of Conceptual Modeling. Elsevier Science Publishers.

Chau, T., Maurer, F. and Melnik, G. (2003): Knowledge Sharing: Agile Methods vs. Tayloristic Methods. Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises: 302-308

Clarke, S., Harrison ,W., Ossher, H. and Tarr, P. (1999): Subject-oriented Design: Towards Improved Alignment of Requirements, Design and Code. OOPSLA '99: 325-337

Coad, P. and Yourdon, E. (1991): Object-Oriented Design. Yourdon Press, Upper Saddle River.

Dieste, O., Genero, M., Juristo, N., Mate, J.L. and Moreno, A.M.(2003):A Conceptual Model Completely Independent of the Implementation Paradigm. Journal of Systems and Software. 68:3,183-198

Fowler, M. (2004): The new methodology. www.martinfowle.com/articles/newMethodology.html

Fowler, M. (2003): UML Distilled. Addison-Wesley.

Henderson-Sellers, B. and Edwards, J.M. (1990): The Object Oriented Systems Life Cycle. Communications of the ACM. 33:9, 142-159

Jacobson, I. (1995): A Confused World of OOA and OOD. J. Object-Oriented Programming, 8:5,15-20.

Kaindl, H. (1999): Difficulties in the Transition from OO Analysis to Design. IEEE Software. 16:5, 94-102

Kleppe, A., Warmer, J. and Bast, W. (2003): MDA Explained: The Model Driven Architecture- Practice and Promise. Addison Wesley.

Laitinen, K. (1996): Estimating Understandability of Software Documents. ACM SIGSOFT Software Engineering Notes, 21: 4: 81-92

Mellor, S.J., Scott, K., Uhl, A. and Weise, D. (2004): MDA Distilled: Principles of Model-Driven Architecture, Addison Wesley.

Motschnig-Pitrik, R. (1993): The Semantics of Parts Versus Aggregates in Data/Knowledge Modeling. In: Proceedings of CAiSE 93. LNCS (65:1). Paris, France.

Mylopoulos, J., Borgida, A. and Yu, E. (1997): Representing Software Engineering Knowledge. Automated Software Engineering, 4:3: 291-317

Paetch, F., Eberlin, A. and Maurer, F. (2003): Requirements Engineering and Agile Software Development. Proceedings of the twelfth IEEE international workshops on enabling technologies: 308-313.

Reeves, W. J. (1992): What is software design? C++ Journal 2:2

Rubin, E. and Wand, Y. (2005): Software Architecture to Support Domain Semantics Representation. IEEE International Conference on Software-Science, Technology and Engineering SWSTE 2005: 3-12

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991): Object-Oriented Modeling and Design. Prentice-Hall, Inc., Upper Saddle River, NJ

Shlaer, S., and Mellor, S. J. (1992): Object Lifecycles: Modeling the World in States. Yourdon Press, NJ

Smith, H. H. (2004): On tool selection for Using UML in System Development, Journal of Computing Sciences in Colleges: 53-63

Uschold, M., King, M., Mpralee, S. and Zorgios, Y. (1998): The Enterprise Ontology. The Knowledge Engineering Review, Volume 13: 1:31-89

Wand, Y. (1989):A Proposal for a Formal Model of Objects, in Object Oriented Concepts, Databases and Applications, Edited by Kim, W., Lochovsky, F, H., ACM Press: 538-559

Welsh, J. and Han, J. (1994): Software Documents: Concepts and Tools. Software - Concepts and Tools, 15:1, 12-25