Models, Domains and Abstraction in Software Development*

Eyðun Eli Jacobsen   Bent Bruun Kristensen   Palle Nowack

The Maersk Mc-Kinney Moller Institute for Production Technology
Odense University, DK-5230 Odense M, Denmark
e-mail: {jacobsen, bbkristensen, nowack}@mip.ou.dk

Abstract

Any software development technology has an underlying model—explicit or implicit—of the development process. In order to understand more about the development process and the methodologies we abstract from these. The perspective chosen for the abstraction includes domains, models developed during the process and the kind of abstraction involved in the phases of the process. A supplementary perspective includes the phases in the process, the actors during the development, and the influencing forces of the quality of the resulting models and systems. In general the perspective controls the nature of the knowledge achieved by means of abstraction. The nature of our result from the abstraction over processes and methodologies is the structure and the interaction of the development process—corresponding to the two chosen perspectives.

1: Introduction

Existing methodologies prescribe how to conduct the software development process—they implicitly assume various models of the development process. Each methodology has its own unique understanding of the process, that is expressed implicitly or explicitly as the underlying model of the methodology. This uniqueness is very important when you are looking at a given methodology, however it is not the purpose of this article to go into such specialties. Rather, the purpose is to describe the commonalities of the methodologies in abstract form. The presentation is a generalized view of the methodologies, together with a number of additional contributions at the abstract, general level of description.

In our abstract, general description of the development process we have chosen a certain perspective on the process. The perspective includes the *models*, the *domains* and the *abstraction forms* involved in the overall understanding of the development process. The content of this article is a model of the understanding of the **entire** development process. By the models **in** the process we refer to the different kinds of models created during the process. Such a model reflects an understanding of a domain of which we either conduct analytic or constructive work during the development process—we model the domain. By domains we mean such domains identified and used during the software development process. By abstraction we mean the kind of abstraction processes we conduct during these kinds of modeling. By our perspective we focus of these concepts in our abstract, general description. Modeling is essential for the software development process—both models that support our understanding of the logical phases of the process including the domains involved in this process, and models (of some of these domains) that are produced during the process. These models produced are of either a abstract or technical nature: a model of a problem domain captures a logical understanding of the concepts in this domain, whereas the description organization of the system can be seen as a model of the architecture of the system. In both cases the models are abstractions over the concrete domain—as abstractions they capture the essential understanding of the domain[1].

The models, the domains and the abstractions are the main elements in our perspective—they form the skeleton of the "structure" in development process. In addition, we focus on the phases, the actors and the

*This research was supported in part by Danish National Center for IT Research, Project No. 74.

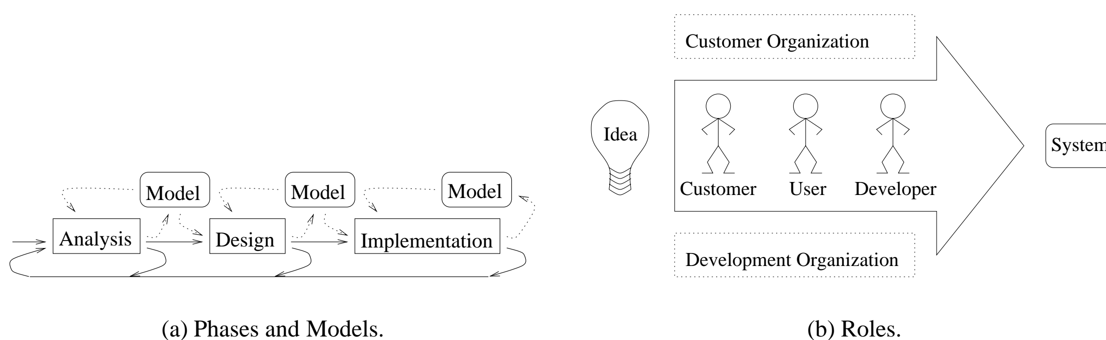
[1] The model of the software development process discussed is primarily intended for information systems.

forces in our model of the development process, that heavily influence the resulting system. The phases each have a process aspect (the activities in the phase), a notation aspect (the diagrams/language for description), and a pattern aspect (the term pattern is used here to cover a distilled, abstract form of experience and qualifications within abstract formations over either process or notation). The actors are described in the form of characteristic roles, that are played during the process. The roles determine the actual choice of certain domains, the requirements to the process, and actual interaction activities of the process—they form the skeleton of the interaction in the development process. Still a number of forces determine the actual outcome of the process—for example environment, skills and experience are essential for the functionality and quality of the resulting system.

The logical phases of the process, i.e. analysis, design and implementation, each have different purpose and characteristics. In object-oriented software development the analysis phase is to some extent constructive by the building of diagrammatic models of domains that are parts of the real world. The challenges during the construction are the choice of perspective on the domains, the formation of the concepts, and the selection of the properties. The implementation phase is also constructive by the transforming and refining of existing descriptions at a more abstract level into the programming language level. The challenge is the simpleness and efficiency of the transformation between the levels. In both cases the outset is given (domains or descriptions) and the important skill is knowledge about the notation in which the constructive work is expressed (diagrams or programming language).

The design phase is even more challenging, and therefore another focus point of the this article. During design we create an abstract model of the system almost from scratch, and with two aspects. One aspects of the model captures the overall architecture of the system, while the other aspect captures the actual functionality of the coming system. Embedded in the model is the essens of the non-functional qualities of the system. The importance of these aspects makes the design phase more demanding than the analysis and implementation phases. Furthermore the type of the fundamental work in the design phase is creative and in some cases innovative. Essentially we create models almost out of nothing—we do not just "mirror" existing domains or "transform" form one level to another—we form new artifacts by creative work based on experience and skills.

The Software Development Process. We focus on the analysis, design and implementation phases of the software development process. The testing, documentation and maintenance phases are not discussed in this article, although the phases also are closely related to the models produced during analysis, design and implementation. We discuss software development methodologies, including the process and the notation, but the support of the methodologies and techniques by tools.



(a) Phases and Models.

(b) Roles.

Figure 1. The Software Development Process.

The software development process is illustrated in Figure 1(a). The development process is iterative—the partial models are revised during the iterations. The activities of the analysis, design and implementation phases illustrated as sequential, but are merged in practice. The models are created during the process, and are the products of the ongoing process.

Figure 1(b) illustrates the overall development process from the initiating idea to the resulting system. In our model there are no organizations or participants as such, but only roles. The intention is to abstract away the many complex situations and focus on the roles played during the process, both by organizations and participants. In the most simple form Figure 1(b) includes two organization roles only (not discussed further in this article), one representing the customers and another representing the developers. In an actual development process the organization roles can be played by a number of companies, and the same company could probably play both roles. Three participant roles are included in Figure 1(b), the customer, the user and the developer. The participants are supplied by the organization roles, and in general a number of participants play a given role. Also, a participant can play several roles. The customer, the user, and the developer roles are important in the methodologies, and the understanding of the roles is indirectly elaborated further in the article. The roles are characterized as follows

Definition: *Customer:* A role played by a person who initiates the development process, provides conditions and constraints for the process, and stops the process.

Definition: *User:* A role played by a person (or another system), that interacts with the system. A person who has knowledge about the problem domain and the usage domain.

Definition: *Developer:* A role played by a person who has knowledge about the software domain. A person who knows about the process and notations of the software development process.

Example: Customer Servicing System. Our example is a CUSTOMER SERVICING SYSTEM that might be used to schedule and serve customers—this might be found in different contexts such as a bank or supermarket. Customers are served by a number of clerks. In order to keep track of which customer should be next served, a number of *ticket machines* are available. When a customer enters, he/she pushes a button on the ticket machine and a ticket with the next number is printed. The number of the latest customer to be served is shown on several updatable displays (which we call *annunciators*). A customer can check this number with his/her own ticket number to see how many customers are ahead in the line. Each customer will be served by a clerk, stationed at a *clerk desk*. The desk displays the number of the customer currently being served by this clerk. Once the clerk has completed the transaction, the desk will update the display to show the number of the next customer to be served by this clerk. Naturally, the annunciators are updated. The components clerk desk, ticket machine and annunciator are illustrated in Figure 2.

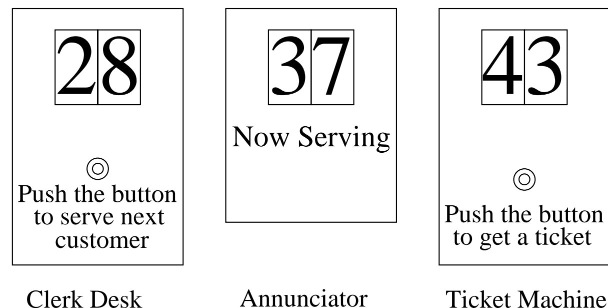


Figure 2. CUSTOMER SERVICING SYSTEM

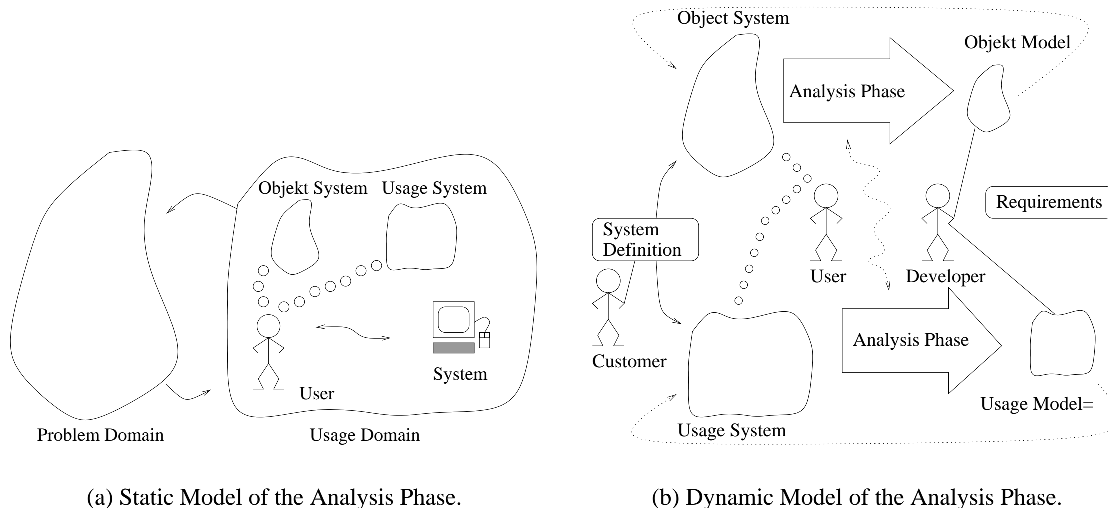
The objective of this example is not to go through a complete developments process of a system of this kind. Rather, we shall rely on an intuitive understanding of this very simple situation, and use it as a means to support the abstract and theoretical concepts presented in the article by concrete and practical examples form this type of application.

Article Organization. In section 2 we discuss a model of the analysis phase, including the modeling of the problem domain and the usage domain. In section 3 we discuss a model of the design phase, including the system model and the architecture model in the software domain. In section 4 we discuss a model of

the implementation phase, including the resulting program as a concrete model in the software domain. In section 5 we discuss the kind of abstractions involved during the phases in the development process, including which domains we actually abstract over. The CUSTOMER SERVICING SYSTEM example is used throughout the article for illustration.

2: Analysis Phase

In the analysis phase the user and the developer closely cooperate to construct models of the problem domain and the usage domain. This process is controlled by a system definition and the customer.



(a) Static Model of the Analysis Phase.

(b) Dynamic Model of the Analysis Phase.

Figure 3. Models of the Analysis Phase.

Analysis: Static Model. The following elements are involved in the analysis phase (most of these are similar to those in the analysis model of [20]):

Definition: *Problem Domain:* The part of the surroundings, that is managed, monitored or controlled by a system.

Definition: *Object System:* The user's understanding of the problem domain—a mental model.

Definition: *Object Model = Model of Object System:* The developer's description of the object system—developed through some given process, expressed in some given notation, and seen from some chosen perspective.

Definition: *Usage Domain:* An organization that manages, monitors and controls a problem domain, including the users.

Definition: *Usage System:* The user's understanding of the usage domain—a mental model.

Definition: *Usage Model = Model of Usage System:* The developer's description of the usage system—developed through some given process, expressed in some given notation, and seen from some chosen perspective.

Definition: *Analysis Model:* Object Model & Usage Model.

Definition: *System Definition:* A definition and a delimitation of the problem domain and the usage domain, that is stated and verified by the customer. It describes conditions and constraints for the resulting system, including the functional and the non-functional requirements as well as the expectations to the execution platform.

Definition: *System:* A collection of hardware and software components, that realizes the customer's system definition.

Definition: *Requirements (analysis):* A set of requirements concerning the properties and qualities of a system.

Domains, systems and models are involved: the (problem and usage) domains are parts of the real world (physical or mental), the (problem and usage) systems are the users understanding of the domains, whereas the (problem and usage) models are the developers description of the systems. Figure 3(a) illustrates the model of the analysis phase with the problem domain and the usage domain. The object model forms the user's understanding of the problem domain during his/her interaction with the system. The usage model forms the user's understanding of his/her interaction with the system (to be) developed.

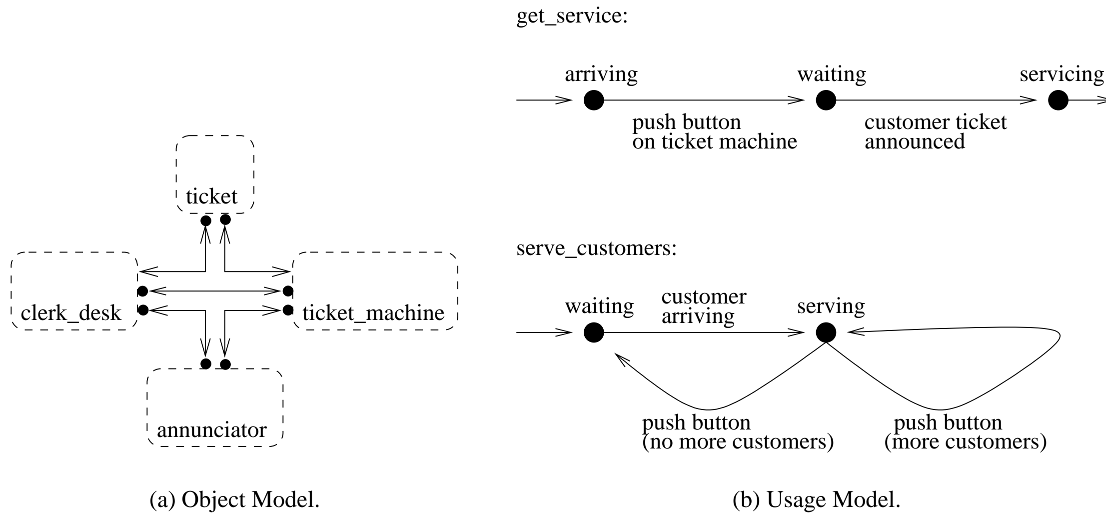
Analysis: Dynamic Model. Figure 3(b) illustrates the analysis process and the resulting models. The process is controlled by the (static) system definition and the (dynamic) interaction with the customer. An important part of this control is the perspective applied in the modeling process. The overall perspective is that of the vision of the resulting system. The process is iterative—the dotted lines indicate the feedback from the descriptions to the mental models. The iteration stops when the user and the developer agree that the descriptions are usable and express a common understanding. The dotted relation between the analysis arrows in Figure 3(b) symbolizes that the two analysis parts have influence on each other. The actual contents of the analysis process of the problem domain influences the analysis of the usage domain, and to a certain extent also vice versa. The results of the analysis are the object model, the usage model and some additional requirements.

The nature of the work during the analysis phase is not purely analytic, but also constructive. In object-oriented analysis we build models to represent the understanding of the problem and the usage domains. This is one example on the fact that some design stimulates the analysis. Another example is given by the preliminary design of graphical user interfaces, that is included in the analysis phase of some methodologies. Such constructive elements may be essential for the communication and mutual understanding between the user and the developer in the iterative cycle of the phase.

Analysis: Customer Servicing System. We illustrate the analysis phase as follows

- Problem domain, object system and model: The problem domain includes the customers, tickets, numbers, clerk desks, ticket machines and the annunciators. The concepts in the object model are `ticket`, which models customer and number, `clerk_desk`, `ticket_machine`, and `annunciator` and their associations (Figure 4(a)).
- Usage domain, system and model: The usage domain includes the roles clerk and customer, and the various ways these behave. We illustrate one important aspect of the usage model with the simplified state diagram in Figure 4(b)). It describes as an example the behavior `serve_customers`, where a clerk will serve a customer, then push a button, then either get another customer and check his/her ticket against the number on the clerk desk, or find out that there are no more customers waiting. Another example is `get_service`, where a customer will push the button on a ticket machine and get a ticket. The customer will then wait, watch the annunciators until his/her number is announced, and finally approach the clerk desk at which his/her number is also announced.
- System definition: More or less the description given to introduce the CUSTOMER SERVICING SYSTEM.
- Requirements: A number of clerk desks, ticket machines and annunciators must be physically distributed in a room. They must be synchronized, so that no tickets with the same number are printed, and no clerk desks announce the same number to be served. There is an upper limit on the number of customers to be allowed to pick a ticket.

Analysis: Process, Notation & Principles. The ingredients of the analysis phase are the process, the notation, and the principles. The process guides how to proceed, the notation defines how to express the



(a) Object Model.

(b) Usage Model.

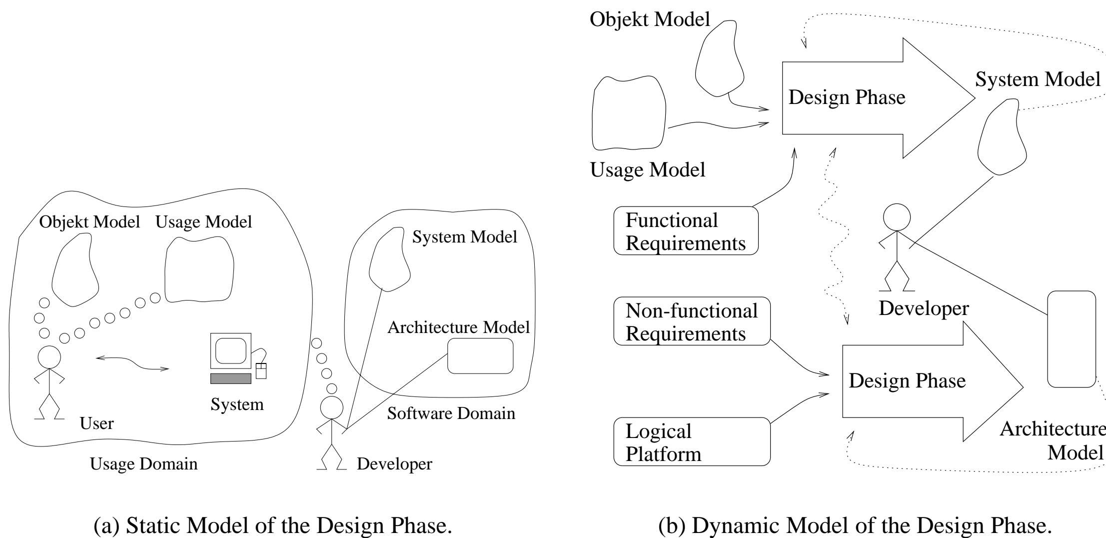
Figure 4. Models of the Customer Servicing System.

models, and the principles offer theory and proved practical experience in a condensed, mature and public form. Separate processes, notations and principles exist for the object model and the usage model. From the list of object-oriented software development methodologies [2], [4], [11], [25], [30], [20] we give the following remarks and examples in relation to process, notation and principles:

- Process: In general the process is described informally by diagrams, text and tables. The analysis process is in the methodology literature described as a set of related activities. Each activity examines a certain aspect of the future system's environment. The activities are sometimes causal related, so that one activity as its result provides the starting point for another activity. Typically though, the activities can be carried out in alternation or even concurrently, implying that the analyzers iterate and update the resulting models.
- Notation: The most important notations applied in analysis are typically graphical. These notations often have precise syntax, but informal semantics. The intention is to capture the intuitive nature of the problem and the usage domain. To support the graphical notations various comments, lists, and prose are necessary in the analysis documentation. To learn how to adopt and apply the notation to meet the specific needs of certain project, it is often very helpful to look at existing examples.
  OOA&D [20] has no own notation element, but in principle it can be parameterized by any appropriate notation, for example UML [28] with some extensions. On the other hand, UML is precisely notation only. Types of analysis models from UML with their own notation include: class diagram, object diagram, state diagram, activity diagram, use case diagram.
- Principles: In general the principles are listed in text or illustrated in diagrams, cf. [20]. The generic principles underlying a certain analysis method can be either implicit or explicit. Explicit statement of principles makes it possible to argue about certain aspects of the method, thus facilitating substitution and elaboration of dedicated issues.
  Principles can be stated in pattern form. A pattern illustrates a principle by explaining a problem, stating a solution to the problem, and describing a context, in which the problem occur and the solution is applicable. Patterns with respect to both process and notation are seen as an advanced way of defining and communicating mature experience in a very condensed form.
  Patterns involved in the analysis phase can support both analytic and constructive aspects. The patterns have intuitive nature and focus on logical structures in the problem domain [10]. Examples of patterns for conceptual modeling are collected and described in [6].

3: Design Phase

In the design phase the developer refines the object model and introduces an architecture model. These parts are related, but an explicit and abstract description of the architecture model essential. The architecture model gives an supplementary perspective, is typically a partial model, is abstract in the sense generalized and generic (parameterization), and describes the logical organization of the system model and the (logical) execution platform. It describes which parts are distributed, which are concurrent, which are to be persistent etc. to enable the developer to transform the object model into a a model that can support the usage of the system, not only the understanding of it.



(a) Static Model of the Design Phase.

(b) Dynamic Model of the Design Phase.

Figure 5. Models of the Design Phase.

Design: Static Model. The following elements are involved in the design phase:

Definition: *Architecture Model:* An abstract model over over the system model in the software domain. The model focus on (the organization of) the structure and interaction embedded in the system model. The purpose is to understand the system model given a structure/interaction perspective on that model, to allow us to reason about and expose the support for the non-functional requirements, and to map the system model onto the logical platform

Definition: *System Model:* A refined, transformed and enriched object model. The system model also supports the usage of the system, and not only the conception of the problem domain.

Definition: *Design Model:* Architecture Model & System Model.

Definition: *Software Domain (at the design level):* Descriptions, partial or complete, of some software. Various design notations are used.

Definition: *Functional Requirements:* A set of requirements concerning the capabilities of a system—what the system should be able to support.

Definition: *Non-Functional Requirements:* A set of requirements concerning the non-functional qualities of a system—for example usability, security, efficiency, correctness, reliability, maintainability, testability, flexibility, understandability, reusability, portability, adaptability.

Definition: *Logical Platform:* A description of the platform on which the system is going to be executed but at a logical level—i.e. requirements in terms of user interface, distribution, persistence and concurrency.

Definition: *Requirements (design):* Functional Requirements & Non-Functional Requirements & Logical Platform.

The result of the design phase is not a model of some **existing** domain similar to the problem domain or the usage domain[2]. Rather, it is a unique design, something constructed and created by the developer based on his skills and knowledge, and from the object model and the usage model. After its construction the system model and the architecture model both works as models in addition to the object model, but from a specific design perspective. Figure 5(a) illustrates the model of the design phase with the usage domain and the software domain. The system model forms the developer's conception of the integration of the object and usage systems at an abstract level. The architecture model forms the developer's conception of the architecture of the system, i.e. the overall structures and their relations and interactions.

Design: Dynamic Model. Figure 5(b) illustrates the design process, its inputs in the form of models and requirements, and its resulting models. The dotted relation between the design arrows in Figure 5(b) symbolizes that the two design parts have influence on each other. On the one hand the object model and the system model are the foundation for creating the architecture. The system model is constructed from the object model. The usage model is used during the design for controlling the transformation of the object model into the system model. The usage model, together with the functional requirements, supply the information for adding and distributing the functionality to the object model to the extent that this information is not already there. The architecture is a constructive solution for how the non-functional requirements and the organization of the logical platform can be combined. The architecture model created by the design arrow at the bottom can have crucial effect on the actual transformation at the top arrow.

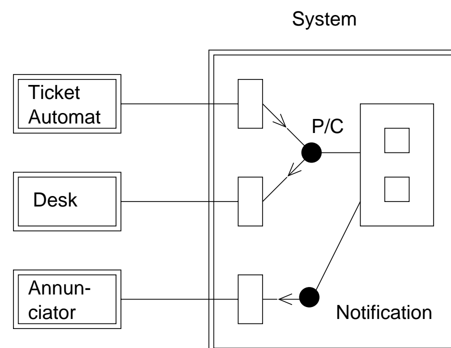


Figure 6. An Architecture Model.

Design: Customer Servicing System. We illustrate the design phase as follows:

- System model: The system model describes which methods are needed in order to support the usage of the system, and in which classes the methods belong (Figure 7). The methods of `clerk_desk` include `next` (to let the clerk notify that he/she is ready for the next customer) and `show` (to let the `clerk_desk` display the number of the next customer it expects). The `ticket_machine` includes the methods `push` (to let a customer request a ticket with a number) and `print` (to print a physical ticket with the next available number).
  During the design phase a central scheduler component is introduced to keep track of the customers awaiting service. The `scheduler` helps to coordinate the other components. The `scheduler` includes the methods `next_customer` (to check if there are customers in the queue and determine

[2]It could possibly be seen as a model of a imaginary vision of the developer of the system, that he/she is going to design—assuming that the developer is able to envision such a design domain during his work, and from a chosen perspective and through his/her understanding of the requirements actually forms the model.

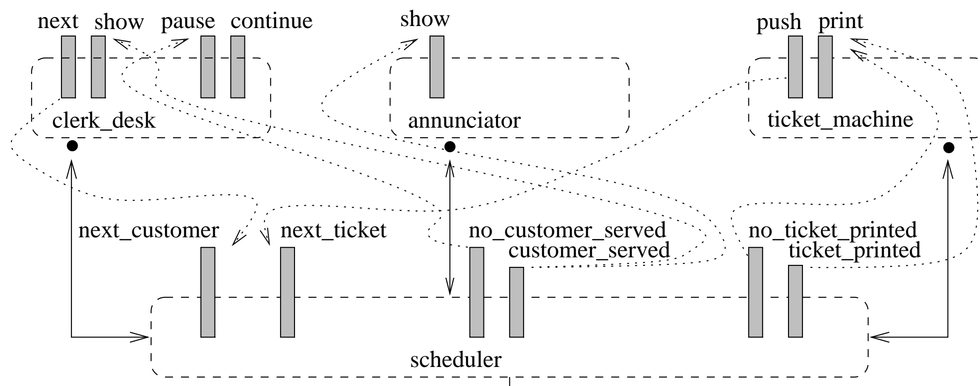


Figure 7. The System Model.

who is next) and `next_ticket` (to check if it is possible to assign a number to another customer and determine the number to be assigned).

- Architecture model: In Figure 6, we highlight that the system can be seen as a combination of two patterns for organizing components, a Producer/Consumer organization and a Notification organization. The `ticket_machine` and `clerk_desk` play the roles of producer and consumer, respectively, and the `annunciator` is being notified when tickets are consumed.
- Software domain: We illustrate this domain only by the diagrams in Figure 7 and 6. We have used a diagrammatic representation of the software domain—a textual representation is another possibility.
- Non-functional requirements: The printing of tickets and the announcements of next customer must be correct—the essential objective of the servicing system is exactly to keep track of customers and to ensure that the sequence of services is fair.
- Logical platform: The platform is distributed and the objects representing clerk desks, ticket machines and annunciators are autonomous objects. Synchronization which is neccessary is supported by an autonomous `scheduler` object.

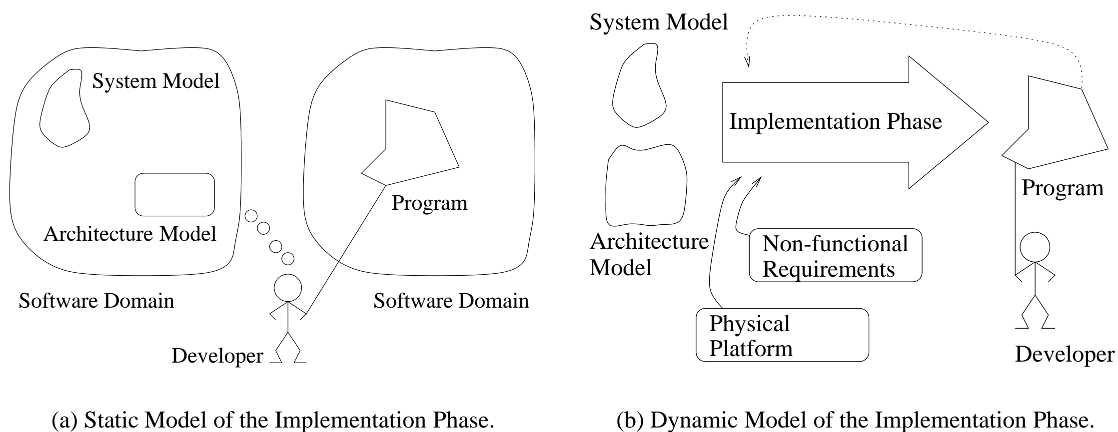
Design: Process, Notation & Principles. The ingredients of the design phase are the process, the notation, and the principles. The process guides how to proceed, the notation defines how to express the model, and the principles offer theory and experience in a condensed, mature and public form. Separate processes, notations and principles exist for the system model and the architecture model. From the list of object-oriented software development methodologies [2], [4], [11], [25], [30], [20] we give the following remarks and examples in relation to process, notation and principles:

- Process: In general the process is described informally by diagrams, text and tables. OOAD has a process element which is characterized by being advices on what kind of activities to perform and in which order, but not by being advices on how to perform the proposed activities. The process element in OOAD is still preliminary and insufficient. Booch and OMT have some guidelines for the design process, but there is no distinction between the system model and the architecture model.
- Notation: In general the notation is some kind of diagrams. The notation has a precise syntax, but no formal semantics. The system model is described in a design notation. A design notation can be seen as class structure and object interaction diagrams. The description of the architecture model can also be described in some design notation or in an architecture (description) language. A design notation can be used selectively to describe selected important parts of the system and in this way describe the architecture model. An architecture description language is dedicated to expressing the architecture model and it can not be used to express the system model. Booch and OMT have a lot of notation, but they do not really distinguish between notation for analysis and notation for design—also, their notation is semantically close to programming languages.

- Principles: In general the principles are listed in text or illustrated in diagrams, cf. [20]. The principles express very general statements about how to work, and hence they need exemplification in order to be useful. Patterns are examples of principles. Patterns involved in the design phase can support both the aspect of system functionality and the aspect of architecture. Patterns in the design phase are characterized by expressing structure and interaction, both in the system model (for example [7]) and in the architecture model (for example [3]).

4: Implementation Phase

In the implementation phase the developer transforms the system model into a refined model in the form of a program, and integrates the architectural model into this program during the process[3]. During the implementation the perspective on the platform becomes a physical perspective.



(a) Static Model of the Implementation Phase.

(b) Dynamic Model of the Implementation Phase.

Figure 8. Models of the Implementation Phase.

Implementation: Static Model. The following elements are involved in the implementation phase:

Definition: *Program:* A description of the system in an object-oriented programming language. We used the term "program" for any collection of source code i.e. a program can also be a set of related programs.

Definition: *Physical Platform:* The platform on which the system is going to be executed at the physical level—i.e. in terms of for example operating systems, graphical user interface systems, file systems, computer networks, dedicated hardware devices etc.

Definition: *Software Domain (at the programming level):* Descriptions of software. The notation used is a programming language. Complete descriptions form the programs of applications, whereas partial descriptions form the program-fragments that make up class libraries and object-oriented frameworks

Figure 8(a) illustrates a model of the implementation phase including the software domain at two different levels, the abstract level with the system and architecture models and the programming level with the program. The developer constructs the program from his combined understanding of the system and architecture models.

[3]The architectural elements become expressed in the programming language. Given the existence of architecture languages we may have a separate description of the architectural elements and have integrated program and architecture compiler do the integration job.

Implementation: Dynamic Model. Figure 8(b) illustrates a model of the implementation process including an iterative cycle, where the program is constructed from the system and architecture models under influence of relevant non-functional requirements and the physical platform.

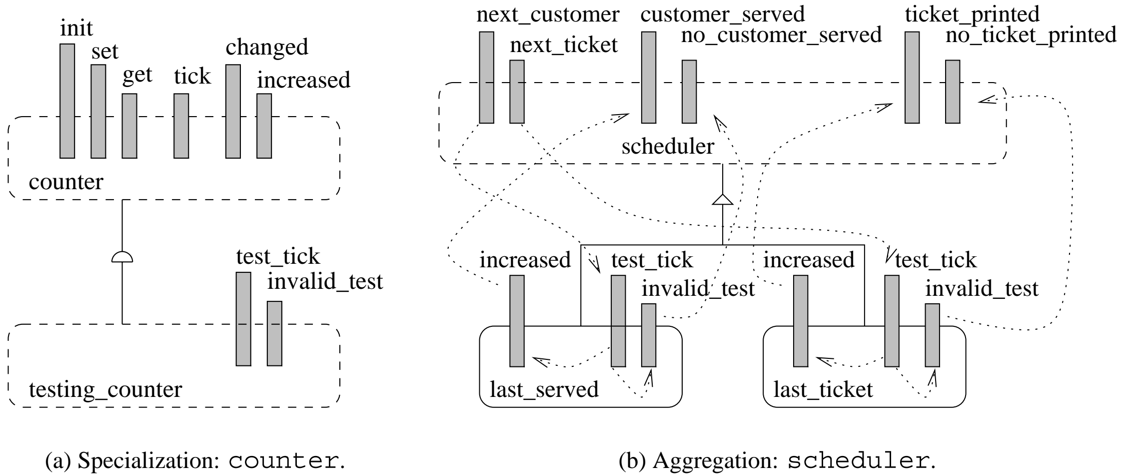
A number of well-known object-oriented programming languages exist, for example [1], [5], [8], [13], [21], [27], [29], [19]. However, in general none of these languages support the expressive power used for the description of the resulting models from the design phase. Therefore, a transformation is necessary from the abstract (design) language level to the concrete (programming) language level—the abstractions introduced need to be implemented.

Dependent on the available and chosen design notation, examples of abstractions to be implemented include: subjectivity (multiple classification of objects, object roles), object relations, containers (lists, dictionaries, stacks, queues, etc.), abstract searching and sorting functionality, concurrent (active) objects, distributed objects, design patterns, architectural styles (layers, model-view-controller, broker architectures [3]).

To support the expression of these design abstractions a wide range of techniques and tools are available. Design patterns typically include an implementation guideline and some source code examples in their descriptions. Meta patterns [22] can be used to implement (different variations of) design patterns [10]. Class libraries typically provide various container and data structure classes as well as iterators for searching. Well-known searching and sorting algorithms can be applied to implement the requested (derived from the non-functional requirements) trade-off between time and space constraints. Object-oriented frameworks [12] (realizing many design patterns) provide functionality for implementing e.g. graphical user interfaces, or persistence and distribution aspects. Problem domain specific frameworks provide domain specific architectures to be reused.

Implementation: Customer Servicing System. We illustrate the implementation phase as follows:

- Software domain: We still illustrate the software domain in this article by means of diagrams, although the textual form (the programming language) dominates at the programming level. At the programming level we illustrate the implementations of the abstractions `scheduler` and `counter`. These are examples of simple translations between design and implementation, because both the design and the implementation languages directly support them with similar notational (language) constructs: classes with attributes and methods, objects with state and behavior, object aggregation, and class specialization.
  In the most simple version, the `scheduler` is aggregated from two simple `counters`. The methods of `counter` include a `tick` method, which will increase the current value by one. The class `testing_counter` is specialized from `counter` by adding another method `test_tick` as illustrated in Figure 9(a). The method `test_tick` will check for some condition. If the condition is fulfilled, `test_tick` will invoke the `tick` method, otherwise it will invoke `invalid_test`. The actual condition is specified in an application of `testing_counter`.
  The `scheduler` comprises two instances of `testing_counters` as illustrated in Figure 9(b). One instance (called `last_served`) counts the total number of customers being served; the other (called `last_ticket`) counts the total number of customers (both modulus some fixed amount of customers allowed to pick a ticket).
- Non-functional requirements: Efficiency is needed, so that the ticket is actually printed within one second from the push of the bottom, and the announcements at the annunciators and the clerk desks within a similar time limit.
- Physical platform: At this level the platform is a network of hardware devices, the implementation language is JAVA, and the interoperability layer is supported by Java's RMI. The devices consists of a conventional PC; a small, quick, and reliable printer; a set of large displays; a set of buttons; and some standard network hardware technology.

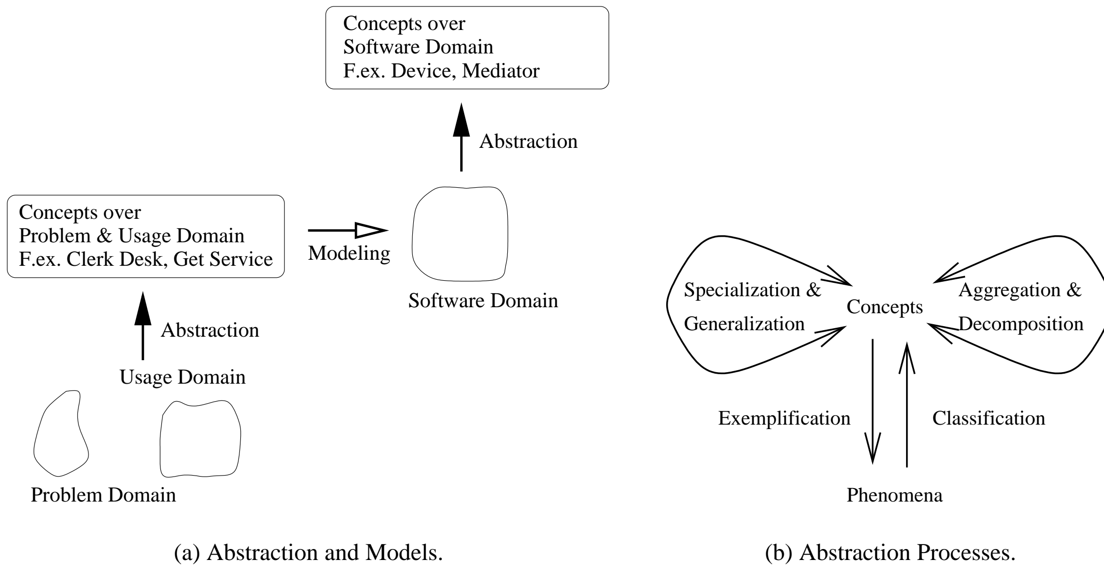


(a) Specialization: `counter`. (b) Aggregation: `scheduler`.

Figure 9. Implementation: Customer Servicing System.

5: Abstractions.

The models of the phases in the development process are used in research to invent new or improved elements of process and notation of methodologies. This improvement of the methodologies involves an interplay between the model of the development process and the process and notation that are available in this process. Abstraction is essential for the notation and thus for the process.



(a) Abstraction and Models. (b) Abstraction Processes.

Figure 10. Abstraction.

Figure 10(a) illustrates the difference and relation between two kinds of abstraction involved during the software development process. During analysis we form abstractions over the problem domain and the usage domain—the abstractions model concepts and phenomena in these domains. We need an understanding of these domains and some notation for expressing the abstraction. During design (and implementation) we form abstractions over the software domain—such abstractions do not model anything from the problem and

usage domains (at least only indirectly). We need a description in some notation for software (abstract for the design level or concrete at the implementation level)—we focus on the description and we form abstractions over the description. Such abstractions are (to a certain extent [4]) dependent of the notation used [16], but are application domain independent.

Figure 10(b) illustrates the abstraction processes in conceptual modeling. Conceptual modeling can be seen as the underlying theoretical understanding of object-oriented programming although existing object-oriented notation on in a limited way support this theory [14]. The processes are fundamental abstraction processes. An object/class is seen as a model of a phenomenon/concept—to give the class of an object corresponds to classification—to give an object of a class to exemplification. To let a (specialized) class inherit from a (general) class corresponds to form a specialization of that (general) class. To describe a (whole) class by the use of references to (part) objects corresponds to form an aggregation of the (part) objects[5].

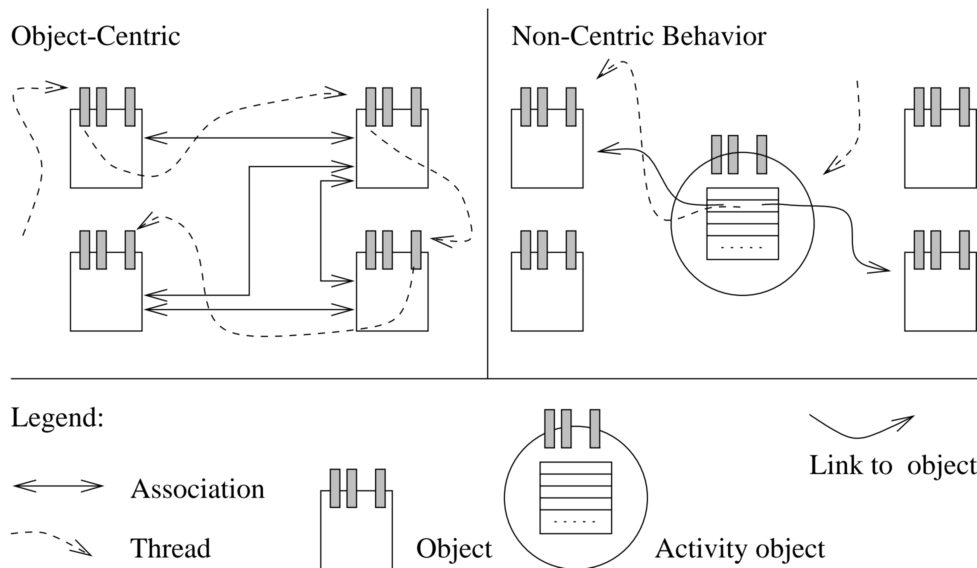


Figure 11. Object-Centric & Non-Centric Abstractions

Problem & Usage Domain. The abstractions used during the analysis phase are based on object-centric and non-centric modeling [23] as illustrated in schematic form in Figure 11. In object-centric modeling, the focus is on capturing the properties of a given concept from a given perspective. The resulting model is a class/object with methods and instance variables that represents the properties of the concept. In the CUSTOMER SERVICING SYSTEM object-centric abstraction in the problem domain is illustrated by the concepts `clerk desk`, `ticket machine` and `annunciator`.

In non-centric modeling, the focus is on the interaction between various phenomena again from a given perspective. The resulting model is a description of how some classes/objects interact to fulfill a common task. As an example we use activity classes/object in Figure 11 to illustrate one possibility of non-centric modeling. In the CUSTOMER SERVICING SYSTEM object-centric abstraction in the usage domain is illustrated by the roles `clerk` and `customer` although these are only used in the non-centric abstractions in the usage domain, namely the usage patterns `serve_customers` and `get_service`.

[4]The design patterns [7] were "discovered" given object-oriented languages as the notation. Other patterns (although not named patterns at that time) have been used given for example procedural languages as the notation used in the software domain.

[5]References can also be used to model general associations between objects—aggregation is usually seen as a special case of (general) association. Some languages allow part objects to be described as integral parts of the whole object in the sense that their existence is dependent on the whole object (e.g. by means of inner classes/objects in Java [1]).

The two types of modeling can form complementary descriptions of a model. Most notations to support the object model are of the object-centric type, whereas most notations to support the usage model are of the non-centric type. Examples of notation that is intended to support the description of the object model in terms of non-centric abstractions include [9], [15], [17], [18], [24].

Software Domain. Abstractions can support both the organization of the software (with respect to the development platform) and with respect to the (logical or physical) execution platform. The latter is described by the architecture model. The former, which is covered by for example module descriptions and is important for software version and configuration control, is not considered part of the architecture of the system and therefore not discussed further in this article. Architectural abstractions are supported by means of patterns [7], frameworks [12] and architecture notation/languages (AL's) [26]. Both pattern and framework technologies are powerful means of capturing abstractions over the software domain. Still the notation used for the representation of these is usual object-oriented notation—we simulate the architectural description. Only preliminary elements of specific architectural notation are available, and the understanding of the necessary expressive power of such a notation is mainly captured through architectural styles (pattern-like abstractions at the architecture level) [26] and [3].

Both object-centric and non-centric modeling is used in the design phase for abstractions over the software domain.

6: Summary

The contributions of this article include

- The characterization of the nature of work during the analysis, design and implementation phases. The work during the analysis phase has an investigating and delimiting nature, during design the nature of the work is constructive and innovative, during implementation transformative and mechanic.
- The clarification of the design phase as the most demanding phase during the development process with creative and innovative work—the existence and use of principles is essential.
- The characterization of the domains involved in the development process, the problem domain, the usage domain and the software domain (covering both the design level and the implementation level).
- The identification of the models constructed during the development process, including the object model, the usage model, the architecture model, the system model and the program.
- The model of each of the analysis, design and implementation phases including a process, notation and principle aspect.
- The identification and characterization of the roles of the participants that interact during the development process, including the user, the customer and the developer.
- The characterization of abstraction forms applied during the development process, including object-centric and non-centric perspectives, architectural abstraction forms.

Acknowledgments. We thank the OOA&D group at Aalborg University, Lars Mathiassen, Peter A. Nielsen and Jan Stage for inspiring research cooperation.

References

[1] K. Arnold, J. Gosling: The JAVA Programming Language. Addison-Wesley, 1996.

[2] G. Booch: Object Oriented Analysis and Design with Applications. Benjamin/Cummings, 1994.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture: A System of Patterns. Wiley & Sons, 1996.

[4] P. Coad, E. Yourdon: Object-Oriented Analysis. 2/E, Prentice Hall, 1991.

[5] O. J. Dahl, B. Myhrhaug, K. Nygaard: SIMULA 67 Common Base Language. Norwegian Computing Center, edition February 1984.

[6] M. Fowler: Analysis Patterns: Reusable Object Models Addison-Wesley, 1997.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[8] A. Goldberg, D. Robson: Smalltalk-80: The language and its implementation. Addison-Wesley, 1983.

[9] W. Harrison, H. Ossher: Subject-Oriented Programming (A Critique of Pure Objects). Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, 1993.

[10] E. E. Jacobsen, B. B. Kristensen, P. Nowack: Characterising Patterns in Framework Development. Proceedings of the 25th International Conference on Technology of Object-Oriented Languages and Systems, 1997.

[11] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard: Object-Oriented Software Engineering, A Use Case Driven Approach. Addison-Wesley, 1992.

[12] R. E. Johnson, B. Foote: Designing Reusable Classes. Journal of Object-Oriented Programming, 1988.

[13] S. E. Keene: Object-Oriented Programming in Common Lisp. Addison-Wesley, 1989.

[14] B. B. Kristensen, K. Østerbye. Conceptual Modeling and Programming Languages. SIGPLAN Notices Vol.29, No.9, 1994.

[15] B. B. Kristensen: Complex Associations: Abstractions in Object-Oriented Modeling. Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, 1994.

[16] B. B. Kristensen. Architectural Abstractions and Language Mechanisms. Proceedings of the Asia Pacific Software Engineering Conference '96, 1996.

[17] B. B. Kristensen, D. C. M. May. Activities: Abstractions for Collective Behavior. Proceedings of the European Conference on Object-Oriented Programming, 1996.

[18] B. B. Kristensen, K. Østerbye. Roles: Conceptual Abstraction Theory & Practical Language Issues. Theory and Practice of Object Systems, 1996.

[19] O. L. Madsen, B. Møller-Pedersen, K. Nygaard: Object Oriented Programming in the Beta Programming Language. Addison-Wesley 1993.

[20] L. Mathiassen, A. Munk-Madsen, P. A. Nielsen, J. Stage: Objektorienteret Analyse og Design. Marko 1997. (In Danish).

[21] B. Meyer: Eiffel, The Language. Prentice Hall, 1992.

[22] W. Pree: Design Patterns for Object-Oriented Software Development Addison-Wesley 1995.

[23] O. Smørdal: Classifying Approaches to Object Oriented Analysis of Work with Activity Theory. Proceedings of the 4th International Conference on Object-Oriented Information Systems, 1997.

[24] J. Rumbaugh: Relations as Semantic Constructs in an Object-Oriented Language. Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, 1987.

[25] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: Object-Oriented Modeling and Design. Prentice Hall 1991.

[26] M. Shaw, D. Garlan: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

[27] B. Stroustrup: The C++ Programming Language. 2/E, Addison-Wesley 1991.

[28] Rational. UML v.1.1. http://www.rational.com.

[29] D. Ungar, R. B. Smith: Self: The Power of Simplicity. Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, 1987.

[30] R. Wirfs-Brock, B. Wilkerson, L. Wiener: Designing Object-Oriented Software. Prentice Hall, 1990.