On Models in Object-Oriented Methods - Critique and a new Approach to Reversibility

Marko Boger, Hans-Werner Gellersen
Telecooperation Office (TecO), University of Karlsruhe
Vincenz-Priesnitz-Str. 1, 76131 Karlsruhe, Germany
Ph. [+49] (721) 6902-49, Fax [+49] (721) 6902-16
boger@teco.uni-karlsruhe.de
hwg@teco.uni-karlsruhe.de

Abstract

In this paper object-oriented methods are examined, problems are pointed out and a new method is presented. In object-oriented analysis and design methods two types of models can be identified. On one hand, static architecture models are quite similar in all methods and can be considered as very elaborate. For dynamic models, on the other hand, a number of problems are encountered. They are not expressive enough, they are not reversible and dynamic models do not support a seamless development process.

We have developed the MODERN method. It re-uses known static architecture models. For the dynamic model, though, a new approach is presented: In a task oriented model a delegation relation is examined. This model is fully reversible, strictly object-oriented and highly expressive. In MODERN the entire development process is reversible and seamless.

1 Introduction

There exists a wide range of object-oriented analysis and design methods. Each method introduces a set of different models. In most methods their models can be grouped into static architecture models and dynamic models.

In the static architecture models, classes, their interface and the relationships between them are described. It is widely agreed upon, that these relationships comprise inheritance, association and aggregation. Also the interface of a class is mostly described in a similar way. So, the static architecture models are, besides notational details, usually very much alike.

In the other group of models, dynamic or functional behavior, special scenarios, behavior in time, states or system behavior are described. To distinguish these models clearly from the static models, we will here refer to them as dynamic models. These dynamic models serve to better understand the system and identify requirements for the static architecture.

While the static architecture models are very elaborate and widely agreed upon, the dynamic models still show many weaknesses. Their development costs a lot of time and thus money. The information gained from these models is little and not sufficiently compact. The given information is too coarse to truly specify the software components. Dynamic models can not be reconstructed from the final program and do not support a seamless development process.

In this paper, we propose a new method called MODERN. It contains only two models, a static architecture model and a task oriented model. The static architecture model makes wide re-use of known, well proven static models. The other, the task oriented model, represents a new approach. It is a dynamic model in the sense introduced above. With this model we address the problems encountered in today's dynamic models and we try to eliminate their major disadvantages. The two models of MODERN are simple and compact, they raise expressiveness, support a seamless development process and are fully reversible.

First, we shortly discuss two methods to outline the problems and disadvantages in today's models. We chose to present OMT by Rumbaugh and BON by Waldén and Nerson. Both methods are discussed following certain criteria, stated at the beginning of section 2. In section 3, 4 and 5 we present our new method. The static model is shortly presented. Then the task oriented model is introduced and thoroughly discussed. It is demonstrated with a simple example. Our models are discussed and compared with other models. Finally the current status of our work and a conclusion are given.

2 On existing methods and their models

In this section, existing object-oriented analysis and design methods and their models are discussed. We have studied a wide range of methods, including OOA/OOD [Coad/Yourdon 91a, Coad/ Yourdon 91b], the Booch-method [Booch 94], Fusion [Coleman 94] and OOSE [Jacobson 92]. For sake of space we restrict our discussion to OMT [Rumbaugh 91], for it is one of the most expressive and widely known, and BON [Waldén/Nerson 95], for it is a new and very proper, modern approach. Nevertheless, the conclusions drawn here are representative for all above mentioned methods.

For a proper foundation of our arguments, we start by defining criteria to judge the models of a method. Since most methods use several models to describe a system, it is necessary to not only look at these models separately, but also as a set.

2.1 Criteria for object-oriented models

In this section, some criteria to enable a qualitative discussion and comparison are presented. The methods examined in this paper will be compared by these criteria. For readability and comparability, the criteria will also be presented in form of a table. In each of these tables, each model of a method as well as the set of models together are presented. The criteria are grouped into three classes: Criteria for the *expressiveness* of the modeling concepts, criteria for the *modernity* of the development process induced by the model and criteria for the *usability* in practice.

Under the keyword *expressiveness* we summarize a number of concepts a model can express. We are only looking at object-oriented methods and their models, so it goes without saying that the concepts class, object and method are supported in at least one of the models of a method. This is true also for *static relations* between classes, namely inheritance, association and aggregation. Nevertheless *static relations* are included in the table to indicate, in which model they are expressed. Furthermore, we look for the following criteria, most taken from [Berard 92].

Delegation: Another relation is the delegation relation. It describes the message passing or delegation call between methods. It will be introduced in more detail in section 4. Can delegation be expressed by the considered model?

Contracting: Is the concept of contracts as introduced in [Meyer 88] supported? Meyer suggests to define an invariant for a class and pre- and postconditions for methods.

Dataflow: In object-orientation only one mechanism to transport data is given, the message passing or method call. Does the model enable the expression of flow of data, reffering o this mechanism?

Conditional constraints: Can the order of actions or behaviors in time be modeled and can it be expressed that certain actions are only executed under some conditions? Is this done for scenarios or for the entire system?

Clustering: Software systems can become very large. Does a model support grouping of classes in some kind of clusters or modules?

System border: Software systems communicate with other systems or the user. Can a proper interface between the system and the outside world be identified?

Systemevents: Can single interactions between the system and the outside world be described? If the previous question was answered positively, the systemevents are the crossings of the system border.

States: States as defined in [Harel 87] can be a powerful technique to model behavior. Is the notion of states included explicitly? If not, is it optional?

In the second class of criteria, we want to examine how well new philosophies in the process of software engineering are supported. We are here primarily talking about models, but models and the development process are closely intertwined. So here we examine the ability of a model to support a certain development process. Many methods are still based on the waterfall lifecycle model. New lifecycle models have been introduced and discussed. Taking the waterfall model as a minimum reference point, more flexibility and more automation is considered positively. The criteria are inspired by [Waldén/Nerson 95]. Under the keyword *modernity* we summarize the following points:

Conceptual integrity: We are talking about devotion to the object-oriented paradigm here. A models integrity is regarded high, if only object-oriented concepts are used and if these are used purely and to their full extent.

Seamlessness: We consider a model seamless, if its development is not bound to one phase, the analysis phase for example, but rather if more and more details and insight can easily be added during the whole process of development. A seamless model should support early analysis as well as serve as implementation specification. How well is seamlessness supported by the model?

Reversibility: During the process of development, the level of abstraction slowly sinks from abstract to more detailed until we finally reach program level. We consider a model reversible if this process is not uni-directional: Can the level of abstraction be raised from detailed to abstract? And thus, can the model be regained from the final code?

In the third class the *usability* of a model in practice is looked at: We summarized the following criteria:

Simplicity: How easily can the model be learned and used? Is it easy to read also for non-software-developers?

Scalability: Can the model be used for small as well as for huge systems?

Effectivity: Some models can be developed very fast, expressing or help to find much information. Others are tedious and time consuming and only deliver little information. How good is the ratio of time needed to develop a model and the information captured in it?

Consistency: We want to examine how well the different models are related. Two models are consistent if their information is related and if changed in one, it is also changed in the other. If supported at all, with which model is the considered model consistent?

2.2 Discussion of OMT

OMT has been developed by Rumbaugh and his colleagues from 87 to 91 at General Electric [Rumbaugh 91]. It has become one of the most widely spread object-oriented methods. It comprises three models, an object model, a dynamic model and a functional model. Terminology differs here a little. The object model is a static architecture model, while dynamic and functional model both are dynamic models in our terminology. In this subsection we will use the term dynamic model in the sense defined by OMT.

In the object model the static architecture is described. Rumbaugh introduces a powerful notation to express the relations inheritance, association and aggregation with many informative details. This model has influenced many static architecture models in other methods. Fusion, for example, fully adopts this model [Coleman 94]. Only few points can be criticized in this model. Following the criteria list given in the last section, only two week points can be identified (see Table 1). Contracting has not jet been introduced. When working with a language that supports contracting, like Eiffel [Meyer 91], this feature is really missing. But even just for the design phase contracting can help a lot, for it is a proper specification of what a software component is to do. Secondly, consistency can not be supported with any other model. Information changed in this model can not affect the other models, because they are conceptually too different. Overall, this model is very good, as can also be seen in Table 1.

In the dynamic model of OMT the behavior of a system in time is described. The model is divided into several submodels. On one hand, the behavior is described in several scenarios. These scenarios are presented in an event trace diagram. Then the event traces from all the inspected scenarios are combined in the event diagram. In the event diagram, the necessary methods are identified. On the other hand, the behavior is described in state diagrams. For each or for the most important classes such a state diagram is developed. In the dynamic model several points are criticized. It suffers mainly because of two reasons: Lack of conceptual integrity and the system is only examined in examples, the scenarios.

Expressiveness can only be judged middle because conditional constraints are always only regarded for scenarios. It is very difficult to cover the behavior of an entire system only by looking at examples.

Modernity must be considered rather low. The examination of scenarios is done in an object-oriented manner, but object orientation is not exhausted. States do not directly relate to an object-oriented concept. Conceptual integrity is thus middle. The process of development is closely related to the waterfall model. Different models are developed in different phases. Thus it can not be considered very seamless. Reversibility is spoiled by the scenario concept. Men chosen scenarios can not automatically be reconstructed from program code. Also state diagrams can not be reconstructed.

Usability we considered only acceptable, mostly because effectivity is low. Due to the scenarios, one is never done with trying to get the full picture.

	Object model	Dynamic model	Function model	**OMT**
Expressiveness	**high**	**middle**	**middle**	**middle**
Static Relations	yes			yes
Delegation				no
Contracting	no			no
Dataflow			coarse	coarse
Conditional constraints		Scenario		partial
Clustering	yes			yes
System border				no
Systemevents		yes		yes
States		yes		yes
Modernity	**high**	**low**	**low**	**low**
Concept. Integrity	high	middle	low	low
Seamlessness	high	low	low	low
Reversibility	yes	no	no	no
Usability	**high**	**middle**	**middle**	**middle**
Simplicity	high	high	middle	middle
Scalability	yes	yes	yes	yes
Effectivity	high	low	middle	middle
Consistency	no	with Obj.	no	no

Table 1: Discussion of OMT

The functional model describes the data and controlflow within the system. It models the internal process of a software system. Processes, actor objects and data store objects are introduced and connected by arcs. This way a good model of reality can be built.

It has to be stated, though, that these concepts are not strictly object-oriented. Conceptual integrity is spoiled. Because of this, expressiveness stays middle. Dataflow can be modeled, but this is done in a rather coarse way. The notion of message passing is not used.

Modernity is low. A major problem is the lack of conceptual integrity. In this model new concepts as processes and actors are introduced. This spoils reversibility as well as seamlessness.

Usability is considered middle. The lack here is also conceptual integrity. Since this model can not directly be translated into object-oriented concepts, effectivity is lost by some amount. Also changes made in this model can not be presented in other models in a consistent way.

Overall the dynamic and functional model of OMT are fairly good and useful models. There is, though, some substantial critique. Expressiveness stays, mainly because of the restriction to scenarios, coarse. Problems are encountered due to the lack of conceptual integrity. Neither model is reversible nor can a seamless development process be supported.

2.3 Discussion of BON

BON has been developed by Waldén and Nerson in an ESPRIT program from 1989 until 1994 and was published in spring 1995 [Waldén/Nerson 95]. The authors take a very modern standpoint. In their book "Seamless Object-Oriented Software Architecture" they fight a battle for pure object orientation, contracting, seamlessness and reversibility. In the preface Bertand Meyer says: "By ensuring seamlessness and reversibility it is possible to obtain a continuos software development process, essential to the quality of the resulting products". We agree with this standpoint. Nevertheless we claim that the development process of BON and its models can be substantially improved. Especially the dynamic model needs improvement with regard to seamlessness and reversibility.

In BON two models are supported. The first is the static model, the second the dynamic model. In both, modeling charts are included to gather information in the early phase of analysis.

In the static model these charts are the system chart, the cluster chart and the class chart. The core of the static model form a static architecture diagram and several class interface models. In addition, a class dictionary can be generated automatically. Compared to the object model of OMT, the introduction of the contracting concept is new. This information is kept in the class interface model. The static model is quite similar to the object model of OMT. Consistency with other models, as in OMT, can not be kept. They are conceptually too far apart. Nevertheless this model is highly elaborate(see Table 2).

In the dynamic model three modeling charts, for events, scenarios and for creation are kept. The main part of this model, though, are dynamic diagrams, one for each identified scenario. In each dynamic diagram, the interaction of several classes relevant for the scenario is modeled. The classes, if needed instances, are connected by an arrow, indicating a message relation. The different message links are labeled with sequence numbers, to model time. Each message link is explained in a scenario box. Conditional control and dataflow are explicitly excluded.

	Static model	Dynamic model	**BON**
Expressiveness	**high**	**low**	**middle**
Static Relations	yes		yes
Delegation			no
Contracting	yes		yes
Dataflow			no
Conditional constraints		Scenarios	partial
Clustering	yes		yes
System border	no		no
Systemevents		yes	yes
States			option.
Modernity	**high**	**low**	**middle**
Concept. Integrity	high	middle	middle
Seamlessness	high	low	middle
Reversibility	yes	no	no
Usability	**high**	**middle**	**middle**
Simplicity	high	high	high
Scalability	yes	yes	yes
Effectivity	high	low	middle
Consistency	no	no	no

Table 2: Discussion of BON

We consider the expressiveness of this model as very weak. The mentioned message relation is only inspected for scenarios, which are just examples, not for the whole system. System events are only shortly described on a chart card. Conditional control and dataflow are not modeled at all. A functional description of a system is never given. Viewed that BON only has these two models, the expressiveness has to be considered as too low.

Modernity stays behind the promises of the authors. Conceptual integrity is kept: only object-oriented concepts are used. Nevertheless, object orientation is not exhausted. Important aspects are left away. Neither is this model seamless nor reversible. The chart cards are very useful at the beginning of development. During later development, though, they are more or less useless. They will usually loose consistency and relevance during development. The scenarios can help a lot to understand and simulate a problem. Thus they will be developed during analysis. They do not help to design and specify a program. Reversibility is spoiled by the restriction to scenarios, as in OMT.

2.4 Summary

For these two as well as for all methods mentioned earlier, the following can be stated:

The static architecture model is usually very expressive, modern and useful. Nevertheless, the problem of consistency is not addressed.

Dynamic models are in its entirety more or less weak. They lack of conceptual integrity; dynamic models make use of concepts that have no correspondence in object-oriented programming or they do not model object-oriented concepts exhaustively. Dynamic models can not be considered seamless. They are restricted to scenarios. They are only helpful in early phases of development. They do not model the entire system and do not help enough to specify software components. In the group of dynamic models not one was found to be reversible.

Overall, three things can be pointed out. No method can support reversibility to its full extent. No method achieves proper seamlessness for all its models. Dynamic models have to be revised in its entirety.

3 A Different Approach: MODERN

We will now present a newly developed object-oriented method for analysis and design. It is called MOdeling and DEsign with a Reversible Notation, short MODERN. We will here present the models rather than the process of development.

It consists of only two models. The first is a Static Architecture Model, short SAM, that is very similar to the static architecture diagram of BON, mixed with some elements of the object model of OMT.

In SAM we express static relations between classes, we group classes to clusters, and we model the interface of each class. Different from BON, we do not distinguish between a static architecture diagram and an interface diagram, but we view these two aspects, static architecture and interface, as different zoom depths in the same model. Also we do not share the notation of the static architecture diagram of BON. Much more we agree with the notation used in the object model of OMT. Nevertheless, the notation of the interface of BON is adopted.

The second model is a Task Oriented Model, short TOM. This model is different from existing dynamic models. It will be presented in detail in the next section.

3.1 An example

To present our models and our notation, we give a simple example. We present a small bilingual dictionary. The dictionary will keep a list of word pairs, an english word and its german translation. The user can ask for a translation or enter new, unknown word pair. We need the following classes: DICT, our dictionary, some word pairs, W_PAIR., the general class LIST, its specialization for word pairs, W_LIST. The W_PAIRs are associated to the W_LIST, which itself is contained in DICT. The static architecture is shown in Figure 1. For shortness we only present the interface of DICT in Figure 2.

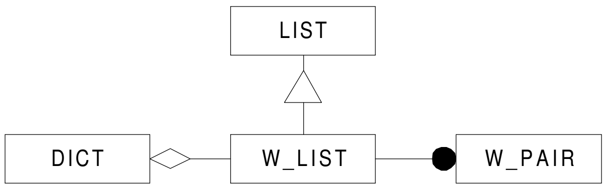
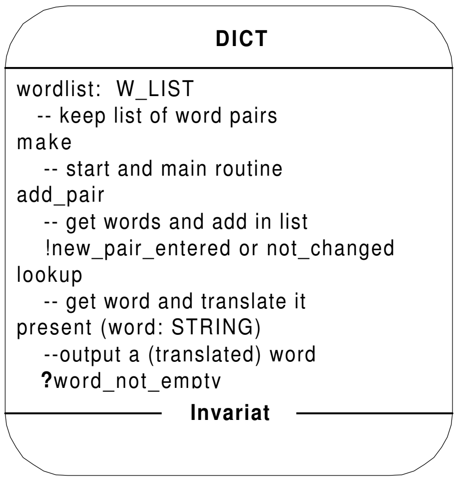


Figure 1: Static architecture



```
DICT
wordlist: W_LIST
  -- keep list of word pairs
make
  -- start and main routine
add_pair
  -- get words and add in list
  !new_pair_entered or not_changed
lookup
  -- get word and translate it
present (word: STRING)
  --output a (translated) word
  ?word_not_empty
Invariat
```

Figure 2: Interface of class DICT

4 A Task Oriented Model

4.1 Solving Problems

The aim of a programmer when writing a program is to solve a given problem. In fact, programs can solve a class of related problems. Problems solved by a program are solved by a sequence of instructions, including branch and loop instructions. This sequence is, for readability and re-usability, devised into short segments, in object-oriented programming called methods. Each such method solves a part of the original problem. Large problems can be broken down to several smaller ones.

And these again can be broken down to jet smaller problems, until we reach an atomic grain size.

To model the relationship between problems and subproblems and to support the process of breaking down problems, we have created a new model. It is called the Task Oriented Model, or TOM for short. In the following paragraphs we will show that TOM is strictly object-oriented, highly expressive, compact and simple. We especially want to point out that TOM is fully reversible and can be kept consistent with SAM.

4.2 The Basic Model

In TOM problems are referred to as tasks. The instance that performs a task is called a process. Tasks can have several subtasks. A task delegates a subproblem to a subtask. The relation between a task and a subtask is called delegation relation or just delegation. For each system only one task oriented model (TOM) is developed. TOM is a graphical model. It has in general the appearance of a tree. The nodes of the tree are processes, notated as ovals. To each process exactly one task is attached, shortly described by a word or a couple of words.

The general problem that is to be solved is called the root task. This root task constitutes the root of the delegation tree. First the root task is devised into subtasks. Each task delegates subtasks to other processes, until the grainsize of the task is atomic. By this method a delegation tree starts building up. This tree is not necessarily a tree in the strict definition. Recursion can introduce cycles. It is then a directed connected graph. Nevertheless, the general appearance of a tree remains.

In the following sections, the model, the development process and the notation of TOM are introduced by presenting the example introduced in section 3.1, a simple english-german dictionary.

The root task is called keep_dictionary. The subtasks that have to be performed to keep a dictionary are to add a new word pair, to look up a word and to choose which action shall be performed. These tasks can be broken down further, as shown in Figure 3.

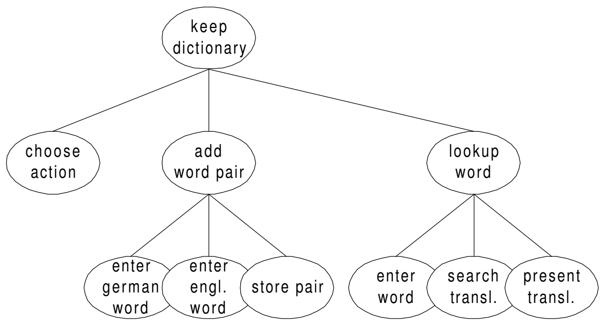


Figure 3: The basic model of TOM

4.3 Identifying Methods and Classes

In object orientation two concepts play a major role, classes and methods.

A class is a combination of some datastructure and a collection of methods. Classes are related to other classes. These relations comprise inheritance, association and aggregation. These relations define a static architecture on the classes of a system. These static architectures are well understood and all analysis and design methods contain a static model to express them.

Methods, on the other hand, are usually not modeled at all. Only their interface, maybe their pre- and postconditons are expressed. We believe, that methods deserve more attention. We will explain why. The concepts of operations (functions and procedures) in procedural programming and of methods in object-oriented programming are not very far apart. Nevertheless, some major differences are apparent. First, while operations are general, methods are encapsulated within the context of their class. Secondly, methods are usually much shorter then operations. This results from a different programming paradigm in object-oriented programming. Methods should be rather short, readable and easy to understand. Complexity is reduced by delegating subproblems to other methods. Thus, complexity of methods themselves is reduced. But the complexity of the relation between methods, the delegation relation, rises. It is this complexity that is not paid attention to in today's analysis and design methods. It is in the task oriented model TOM, that the delegation relation is modeled. We want to state here, that if a task tree is modeled properly, each task can be associated to a method from the SAM model.

As the delegation tree grows, some regions will become stable, while others still change. In the more stable regions (or in the final tree), we can look for a task that a method could take care of. Doing this, we might re-use known methods from known classes from the static architecture model, or we might identify the need for a new method in a known class. We might also identify the need for a new class. Once identified, the notation of a process slightly changes. In the middle we add a line. On top of this line, the name of the method is notated, the name of its corresponding class underneath (see Figure 4).

Some tasks will not be solved within the system, but outside of the system. This can be i.e. a task of external equipment or the user. In the dictionary example there are several tasks, that the user has to take care of, namely choose an alternative action and enter the english and/or german word. These tasks should not be related to a class. Nevertheless, these tasks are marked with a double oval, to indicate the detection of this insight.

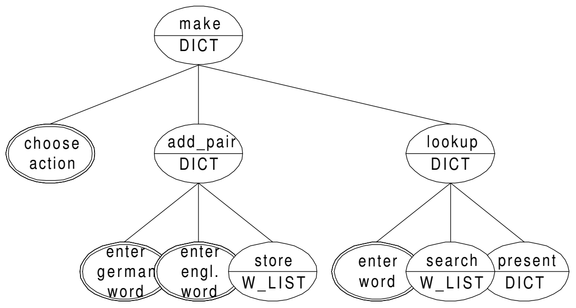


Figure 4: Identifying methods and classes

4.4 A System Borderline

A software system can interact with other systems. These can be for example other software systems or physical measurement equipment, very often this will be the user. Between the inner system and outer world we can model a system border. We notate this border as a line (Figure 5). The mechanism to interact with the outside world again are methods. Methods interacting with the outside world should always be leafs of the delegation tree in TOM.

The tasks identified to be not within the system itself can be dragged beneath a system border line. This way a clear system border line is identified. Especially the border between the system and the user, the system-user border line, is interesting. The direction in which the border is crossed, input or output, can be identified as well. A special method for interactive systems has been developed, based on this notation [Gellersen 95].

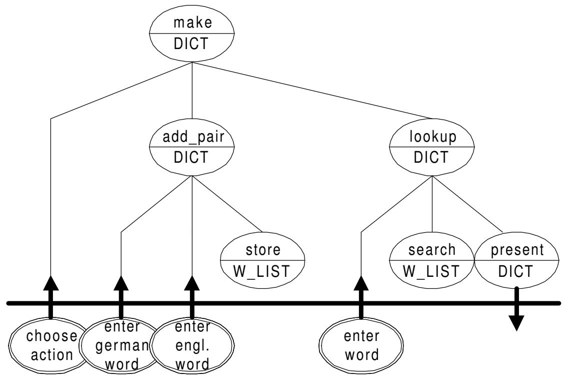


Figure 5: System border

4.5 Dataflow

In a proper object-oriented system, data can only flow by a method call. For this reason, method calls are also called message passing. Method calls, or delegations as we call them, can have no dataflow, dataflow from the caller to the server or from the server to the caller. The first confirms to a procedure call, where only the flow of control is transmitted. The others confirm to procedures with arguments and functions, respectively.

In the delegation tree of TOM all delegations are modeled. So all control and dataflows can be expressed here as well. Dataflow is indicated by an arrow close to the caller or the server, respectively. Also the type or the identifier of the transmitted data can be notated.

In the dictionary example, data flows within the add-pair process, from the user to the add-pair method and from add-pair to store, and within the lookup process as shown in Figure 6.

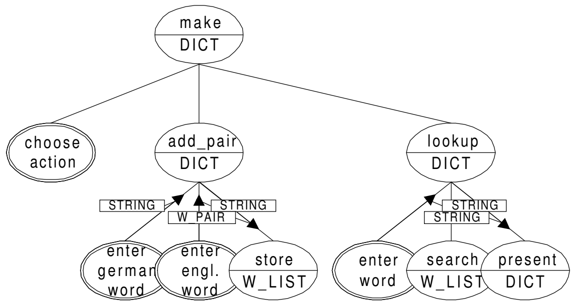


Figure 6: Dataflow

4.6 Conditional Constraints

In the basic model a task is related to a number of subtasks offering their help to solve the given problem. The set of subtasks of a task indicate all possible delegations that the task could call on. Usually though, not all possible delegations are actually affected. The delegation calls underlie conditional constraints. We can express this in TOM. For each task the conditional constraints for its subtasks is stated. A subtask can be optionally called under a condition. A subtask (or more) can be called repeatedly. Two (or more) subtasks can be executed alternatively. Two subtasks can be called in sequence. These conditional constraints are expressed in a regular expression. To each arc representing a delegation relation, a unique letter is attached. This is only done for shortness. It is possible to use words or the original name of a method as well. The mentioned four types of constraints are notated as follows. For a sequence of subtasks, the letters representing them are separated by a colon. An optional subtask, or its representing letter is enclosed in two horizontal lines. Alternative subtasks are separated by an or-sign. Repeated delegation, finally, is expressed by a star, for an unknown number of loops, by a plus, for at least one loop, or a number or a variable name for a known number of iterations. Of course, subtasks can be grouped in brackets. These constraints clearly correspond to programming constructs. Optional and alternative delegation is realized by an if-, case- or inspect-statement. Repeated delegation clearly corresponds to a loop-construct. The details, though, the exact definition of the boolean condition for an if-clause for example, are left away. For the dictionary example this is shown in Figure 7.

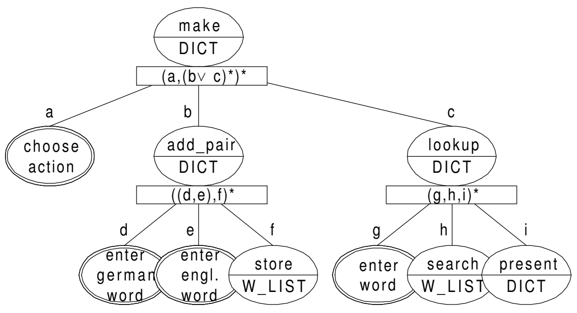


Figure 7: Conditional constraints

5 Discussion of MODERN

In this section we want to critically revise our method MODERN. We hope that the last section has convinced the reader that our new model TOM is simple, highly expressive and very compact. In one single model, we can cautiously analyze and model our system, identify needed methods and classes, roughly specify the implementation of methods, find the system borderline, express dataflow and conditional constraints. Nevertheless, our model is simple, readable, can intuitively be understood or developed and leaves a high amount of flexibility to the developer.

In the next sections we want to point out and discuss three more important features of MODERN and TOM. We then summarize our presentation and cheque the criteria introduced at the beginning. Finally we compare our method MODERN to OMT and BON.

5.1 Conceptual Integrity: Nothing but Object Orientation

At the heart of our method lies a principle: All of object orientation and nothing but object orientation. In our two models we can express all aspects of an object-oriented software system. We can model classes and relations between them. We can group different parts of a system to a cluster. We can express the interface of a class and the contracts for it. This had been well understood in other methods before and has been adopted from others. But only the specification of a class is not the whole story. This is the information the user of a class will need. The developer or implementer of a class needs a lot more. And the object-oriented paradigm has more to offer: In our approach we model methods and the relations between methods as well. The methods really do the work, it is here where a problem is solved and it is here, where the problem has to be understood. It is between methods that the control- and data flow flows. And it is within the methods, that the system border is crossed. The most important relations between methods is the delegation relation. This is the fundamental mechanism of object orientation. By delegation the thread of control is woven, following conditional constraints, data is transmitted and the system border is crossed. These are important aspects of an object-oriented system and should be modeled and specified. We do this in our approach.

5.2 Reversibility

In the waterfall lifecycle model, the development process was basically uni-directional. A system was analyzed, then designed and finally implemented. If an error made in an early stage is encountered, the whole process is taken back to the erroneous point and the entire system is redesigned.

In real life though, this model has proven to be too inflexible. Usually analysis decisions are continuously changed in the design phase and design decisions during implementation. At the end none of the delivered documents is correct.

We view the development process as a stepwise refinement of an abstract model to a more and more detailed model until we finally end up with a program. If the process of adding details to a model can be reversed, so that early analysis models can be regained from a late model or the final program, we call this model reversible.

The static architecture model of OMT, BON or other methods is reversible. This has been shown in several CASE-tools. The dynamic models of these methods, though, are not reversible. Nevertheless, this is very well possible. The key to reversibility in dynamic models lies in strictly sticking to object-oriented concepts and using these to their full extend.

When looking at a piece of code of an object-oriented program (we assume a properly object-oriented language, like Eiffel) there are not very many concepts. Basic instructions like assignments, branch instructions like if and case instructions, loops and calls to other methods. In Eiffel for example, even the addition of two numbers is implemented as a method call to the routine add. The key concept in object-oriented programs is the delegation. The delegation relation is what we model in our task oriented model TOM.

Due to its conceptual integrity, TOM is reversible. It can be reconstructed from code. Leaving away all details of a program, what is left is just the basic model of TOM introduced in section 4.3. But also most of the details can be included in TOM. The parameters of calls represent dataflow as discussed in section 4.5. The information expressing in which order and under which conditions which delegations should be affected, the branch and loop statements, can be directly translated to regular expressions for conditional constraints, introduced in section 4.6. Calls to routines dealing with the outside world, a read-statement for example define the system border from section 4.4. Information, that is too detailed can be filtered out. The boolean expressions of a branch or loop and calls to library methods like add are left away.

Thus TOM is fully reversible. At the same time, this shows how well TOM serves as a specification for the program. Once TOM has been fully developed, only few more details need to be added to achieve the final program.

5.3 Seamlessness

Waldén and Nerson point out the importance of seamlessness in a development process in [Waldén/Nerson 95]. We agree with this but we have found, that not all models used today support a seamless development process. Static architecture models, as in OMT or BON can usually be considered seamless.

Dynamic models, though, have up to date only served to analyze systems and to encounter information needed for the static model. They do not serve to model details and specify software components. They do not help the implementers as well as needed. They are not seamless.

We have tried to develop a new method that is seamless in all its parts. All its models should be seamless, but also the seams between the models and the program as well as the seams among the different models should be minimized.

SAM is adopted from other methods, mainly from BON. It readily inherits the property of seamlessness.

The difficulty was to design a seamless dynamic model. TOM can be used in very early analysis. It then helps to break down a problem into its subproblems. It can be used during design, when details as dataflow and conditional constraints have to be modeled. For the implementation phase it can serve in two ways. It gives a specification of the software component and it can be generated from the existing code, to represent the actual state of the project. Thus, TOM is a seamless dynamic model.

This way also the gap between model and software becomes very little. With SAM and TOM very precise specifications of a program can be developed. On the other hand SAM and TOM can be used, grace to reversibility, as views upon the existing software.

Even the seams among the models of MODERN disappear. The models SAM and TOM can be developed hand in hand. Methods from existing classes in SAM serve as candidates for subtasks in TOM. In TOM needed methods and classes can be identified. Also, a task from TOM and a method from SAM can be attached to each other. This way there is a semantical link between the two models. If information is changed in one, it can consistently be represented in the other.

5.4 Summary

In the last sections, we have presented the MODERN method and its two model SAM and TOM. In this section, we want to come back to the criteria for models introduced in section 2.1.

The static architecture model SAM is adopted from BON. So the criteria are all answered as positive as for BON. Only for the criteria consistency, where BON had its only encountered weak spot, we can now point out an improvement. SAM can, assuming appropriate CASE-tool support, be kept consistent with TOM; any relevant information that is changed in SAM will be shown in TOM consistently and vice versa.

	SAM	TOM	**MODERN**
Expressiveness	**high**	**high**	**high**
Static Relation	yes		yes
Delegation		yes	yes
Contracting	yes		yes
Dataflow		yes	yes
Conditional Constraints		yes	yes
Clustering	yes		yes
System border		yes	yes
Systemevents		yes	yes
States			option.
Modernity	**high**	**high**	**high**
Concept. Integrity	high	high	high
Seamlessness	high	high	high
Reversibility	yes	yes	yes
Usability	**high**	**high**	**high**
Simplicity	high	high	high
Scalability	yes	yes	yes
Effectivity	high	high	high
Consistency	with TOM	with SAM	yes

Table 3: Discussion of MODERN

Our dynamic model TOM has to be discussed in more detail. Expressiveness can now be regarded as high. The delegation relation, not modeled in other dynamic models can now be expressed. Dataflow and conditional constraints, excluded in other dynamic models can be expressed strictly using object-oriented concepts. A proper system border and system events can be identified and modeled. The system is not anymore only examined for scenarios, but for the entire system. Nevertheless we consider scenarios as very important and helpful. They should be examined in analysis to better understand the problem. But one should not stop here. Information from scenarios can be collected in TOM. In TOM, then, the entire system can be modeled. As in BON we propose to use state diagrams as extension where needed.

Development process concepts are supported in a very modern way. Conceptual integrity is ensured. Only object oriented concepts are used, even used exhaustively. Our model TOM is truly seamless. It can be used for very early analysis as well as in latest design. It can even be reconstructed from code for maintenance or representation. Different from all other dynamic models, TOM is reversible.

Usability is high. It is easy to read and easy to learn. TOM can be scaled from small to huge problems. A great improvement is achieved in effectivity (TOM directly serves as implementation specification) and consistency (SAM and TOM are semantically linked).

5.5 A Comparison

In this section we will compare the three methods examined in this paper. Overall MODERN improves the development process in all three classes of criteria. Expressiveness as well as modernity and usability are raised. With MODERN all concepts expressed in other methods can be expressed as well. In addition, the delegation relation can be expressed, dataflow can be modeled using the object-oriented mechanism, message passing, conditional constraints are included at fine grain size and a proper system border can be identified.

Our approach supports modern development process concepts. Compared to other methods conceptual integrity, the strict use of object-oriented concepts, is maintained in the most proper way. While OMT introduces concepts not conforming to the object-oriented paradigm and BON does not make use of object-orientation exhaustively, in MODERN all aspects are modeled using object-oriented concepts. Seamlessness can be improved. OMT was not considered seamless, due to the underlying waterfall lifecycle model. In BON only the static but not the dynamic model was considered seamless. In MODERN both models support a seamless development process. In addition, the seams among the models and towards program code disappear. Reversibility, in both other methods as well as in all others, was only possible for the static model Our method is the first to be reversible in all its parts.

Due to the strict use of object-oriented concepts, effectivity was raised. Everything expressed in our models can directly be transformed into program code. Our dynamic model is not restricted to scenarios but can model the entire system. Finally, while other models can not support automatic consistency, this can be fulfilled in our method, grace to the semantic link between our models.

	OMT	**BON**	**MODERN**
Expressiveness	**middle**	**middle**	**high**

Static Relation	yes	yes	yes
Delegation	no	no	yes
Contracting	no	yes	yes
Dataflow	coarse	no	yes
Conditional Constraints	partial	partial	yes
Clustering	yes	yes	yes
System border	no	no	yes
Systemevents	yes	yes	yes
States	yes	option.	option.
Modernity	**low**	**middle**	**high**
Concept. Integrity	low	middle	high
Seamlessness	low	middle	high
Reversibility	no	no	yes
Usability	**middle**	**middle**	**high**
Simplicity	middle	high	high
Scalability	yes	yes	yes
Effectivity	middle	middle	high
Consistency	no	no	yes

Table 4: Comparing OMT, BON and MODERN

6 Current Status

The MODERN method was developed based on the experience from the DOCASE project [Mühlhäuser 93] and the VDAB project [Gellersen 95].

We are using MODERN in connection with the programming language Eiffel. This has several reasons. First of all, we are dealing with software engineering. Eiffel seems to be the language most appropriate for this. Secondly, one of our main aims is reversibility. For this we need a proper object-oriented language. With the C-part of C++ we would encounter many problems here. Eiffel on the other hand fulfills our needs.

On the base of MODERN a specialized method for interactive systems has been developed, called MEMFIS. For interactive systems, it is important to identify the exact interface between the user and the system. Within the model TOM, a system border can be identified. The interface between user and system is a specialization of this system border. Based on this model, we have developed a method to build interactive software [Gellersen et al. 95].

We are constantly validating our method. Currently, the method is tested in a medium scale project in the automobile industry. It is also further investigated in practical courses at university. Also a prototype of a CASE-tool is currently been developed. MODERN is a very appropriate method for a CASE-tool. Only two models are used. These two are a very precise specification of the software system under development. Both models are fully reversible. Both models can be kept consistent both with each other as well as with the program. We are developing a development environment including a powerful Drag and Drop mechanism as it is used in Eiffel Bench and Eiffel Base [Eiffel 95]. In future, we plan to construct a development method for distributed and concurrent systems.

7 Conclusion

In this paper we have discussed object-oriented models. We examined the two methods OMT and BON and their models representatively for existing methods. This discussion has shown, that static models are very elaborate. It has also shown that dynamic models have severe weaknesses. Major problems were encountered as lack of reversibility, lack of consistency between models, restriction to scenarios and not enough specification of how methods are to be implemented.

We presented a new approach, the MODERN method. Its static architecture model SAM is reused from other methods. Its dynamic model, the task oriented model TOM, on the other side, is a new approach. In TOM the delegation relation is examined and modeled. TOM is simple, yet highly expressive. It strictly makes use of object-oriented concepts and does this exhaustively. It supports a seamless development process and it is fully reversible.

Overall, our approach improves expressiveness and usability of object-oriented methods. The development process is entirely seamless. With our method MOdeling and DEsign can be done with a fully Reversible Notation - MODERN.

References

Berard 1992. *A Comparison of Object-Oriented Development Methodologies*, Gaitherburg, Maryland: Berard Software Engineering.

Booch, G 1994, *Object-oriented Analysis and Design with Applications*. Redwood City, California: Benjamin Cummings.

Coad, P. and Yourdon, E 1991a. *Object-Oriented Analysis, 2nd Ed.* Englewood Cliffs, New Jersey: Prentice Hall.

Coad, P. and Yourdon, E 1991b. *Object-Oriented Design*, Englewood Cliffs, New Jersey: Prentice Hall.

Coleman, D et al. 1994. *Object Oriented Development - The Fusion Method*, Englewood Cliffs, New Jersey: Prentice Hall.

Eiffel 1995. *ISE Eiffel: The Environment*, ISE Technical Report TR-EI-39/IE.

Gellersen, H.W. 1995. *Support of User Interface Desing Aspects in a Framework for Distributed Cooperative Applications*. In Taylor, R. (Eds.) Software Engineering and Human-Computer Interaction, Lecture Notes in Computer Science 896, Springer Verlag, 1995.

Gellersen, H.W., Boger, M. Bonnet, T. and Hirschmann, M. 1995. *A Toolkit and a Method for Building Modality Abstraction into Interactive Software*, Technical Report. Submitted for publication.

Harel, D 1987. *Statecharts: a visual formalism for complex systems*. Science of Computer Programming 8 (1987): pp. 231-274.

Jacobson, I. 1992. *Object-Oriented Software Engineering - A Use Case Driven Approach*, Wokingham, England: Addison-Wesley.

Meyer, B. 1988. *Object-Oriented Software Construction*. Englewood Cliffs, New Jersey: Prenice Hall.

Meyer, B 1991. *Eiffel: The Language*, Englewood Cliffs, New Jersey: Prentice Hall.

Mühlhäuser, M., Gerteis, W., and Heuser, L. 1993. *DOCASE: A Methodic Approach to Distributed Object-Oriented Programming*. CACM 36, 9 (Sept. 1993), pp. 127-138.

Rumbaugh, J. 1991. *Object-Oriented Modelling and Design*. Englewood Cliffs, New Jersey: Prentice Hall.

Stein, W. *1994. Objektorientierte Analysemethoden: Vergleich, Bewertung, Auswahl.* BI Wissenschaftsverlag.

Van den Goor, G., Hong, S., Brinkkemper, S. 1992 *A Comparison of Six Object-Oriented Analysis ans Design Methods.* Center of Telematics and Information Technology, University of Twente, Netherlands, and Computer Information Systems Department, Georgia State University, Atlanta, USA.

Waldén, K. and Nerson, J.M. 1995. *Seamless Object-Oriented Software Architecture*, Englewood Cliffs, New Jersey: Prentice Hall.