Analysis beyond UML

Christiane Stutz, Johannes Siedersleben, Dörthe Kretschmer, Wolfgang Krug
sd&m Research
christiane.stutz@sdm.de, johannes.siedersleben@sdm.de, doerthe.kretschmer@sdm.de, wolfgang.krug@sdm.de

Abstract

In spite of being a de facto standard for analysis and design, UML has some obvious shortcomings: the UML definition is at best semi-formal, the set of result types is far too large and heterogeneous, the tool support is not satisfactory.

If this is true – how do people cope with UML? At sd&m (a medium size software company in Munich, Germany) we conducted a study of best practices concerning the analysis phase. The aim was to find out what a typical analysis documentation looks like and how UML is used or not used.

Here is the surprising result: We identified seventeen modules a typical analysis documentation consists of, that we summarize briefly in this paper. Only three out of seventeen modules use UML at all. We will try to make clear why UML is by no means satisfactory for the needs of real projects and that there is still a lot of work to do.

1. Introduction

The development of large information systems is a complex task. The software engineering discipline has suggested several software development processes like the waterfall model, the spiral model, the Rational Unified Process (RUP) or Extreme Programming (XP) to meet the challenges of system development. While these models differ in many aspects, there are always three tasks that have to be completed before implementation: requirements specification, analysis and design.

Requirements specification is mostly about gathering requirements. Analysis is focused on structuring the problem towards building a software solution. In the third step a design is developed based on the analysis results. This is also a development from informal documentation (as a result of requirements gathering) to formal documentation, consisting of design models. This paper is restricted to analysis; we consider neither requirements specification nor design. Analysis has to fill the gap between these two end points.

An analysis documentation has (at least) two kinds of different readers: the first one is the customer who wants to make sure that the specified system is suitable for his problem. The second one is the developer who has to develop a system complying with the requirements. Thus, the documentation reminds of a Janus' head with two faces: the informal face is turned to the client, the formal face is turned to the developer. Informal documentation is usually prose, while formal documentation often uses models according to the analysis paradigm that has been chosen.

Currently the object-oriented paradigm is the most popular paradigm in software engineering. Together with object orientation, the Unified Modeling Language (UML) [1][11] has become the most popular modeling language for analysis and design. But is it true that "The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing and documenting the artifacts of software systems" [11]? Do we really expect to find an analysis documentation written completely or at least mainly in UML?

In fact, UML has some obvious shortcomings:

- The set of result types is too large and heterogeneous, the definition is at best semi-formal. There are twelve diagram types in the three categories of structural, behavior and model management diagrams. In [11] we find about 90 diagram elements, still leaving interpretation of the diagrams to the user.
- Tool support is not satisfactory. There are still various tools that do not support all UML elements. Only few tools are suitable from analysis up to implementation. The reporting functions of most of the tools are not adequate. Models are not interchangeable between tools.

So how do software engineers cope with UML in their projects? Do they really use it for their every-day work? We conducted a study of best practice projects at sd&m (a medium size software company in Munich). The aim of the study was to answer the following questions:

- What does a typical analysis documentation consist of?
- What tools and models are used for the analysis documentation?
- What parts of UML are really used and which of these are helpful?
- What other modeling or documentation techniques are used?

In the example projects we found seventeen typical modules that an analysis documentation consists of.

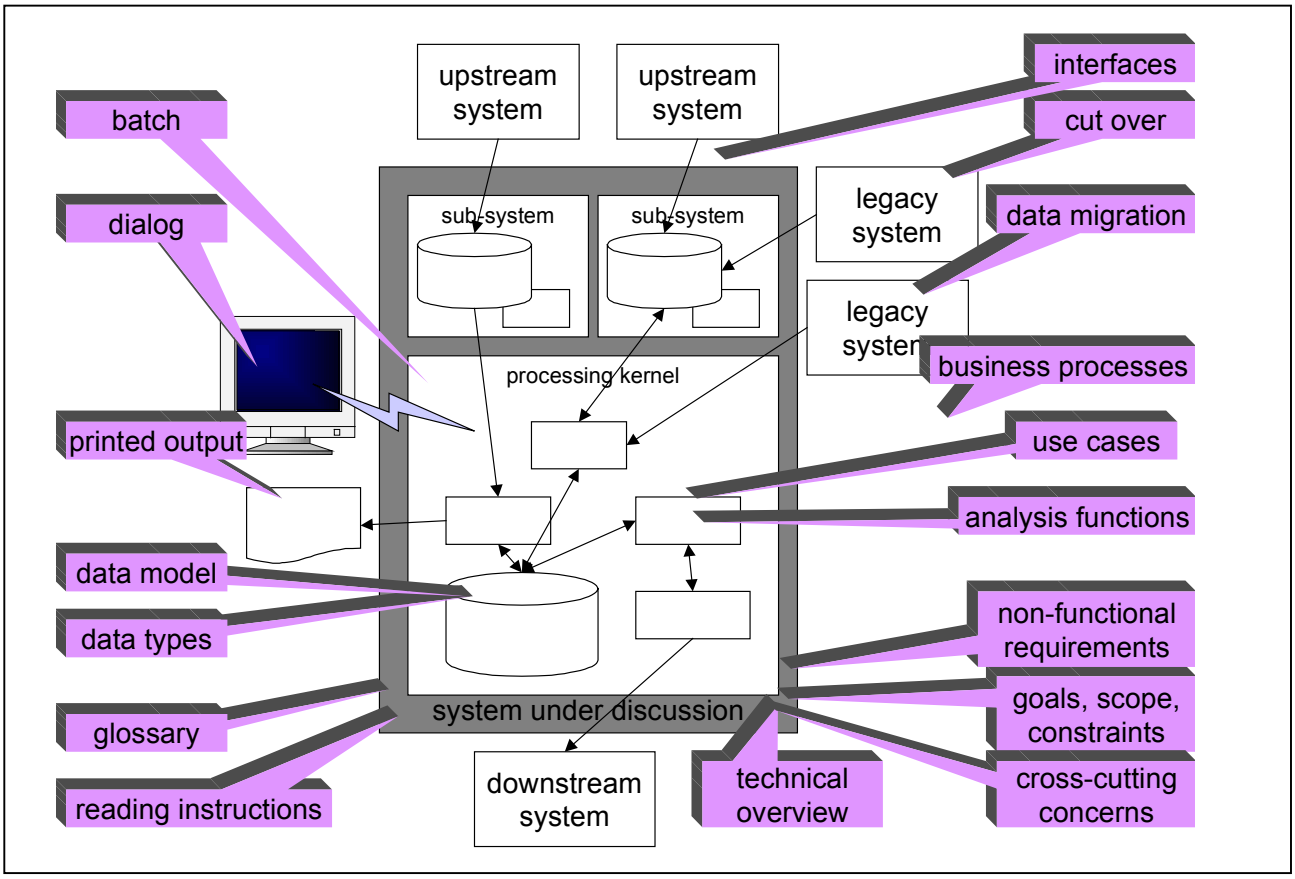

Figure 1: A typical system under discussion and the analysis modules.

Concerning UML, our illusions were shattered: Only three out of seventeen modules use UML diagrams at all. Apart from the obvious shortcomings of the UML we found another one: many of the real-world problems projects are faced with, e.g. legacy systems, interfaces to up/downstream systems, look and feel of the user interface, are not addressed by the UML diagrams. The following sections briefly describe all seventeen modules and say where and how UML was used and what other kinds of documentation were found.

2. Overview of the Analysis Modules

Figure 1 shows a rough plot of the architecture of a typical information system. Main elements of the architecture are a database, several business application modules, interfaces to up- and downstream systems, interfaces to or migration of legacy systems, a user interface, printed output. Each of these aspects has to be addressed.

The boxes arranged around the architecture plot in Figure 1 indicate the analysis modules we found in the best practice analysis documentations. The modules point to that part of the system from which we get the requirements for this module. By an analysis module we mean the part of a documentation that addresses a particular problem in the analysis. It may be a separate document or a chapter in the documentation, in form of written prose or diagrams. Nevertheless we detected that there are particular forms for the documentation that have proven suitable in many projects and suggest these as a guideline in the analysis modules. Table 1 lists the modules sorted by the problems addressed in the module. The notation used in the modules is also shown. Each of the modules has a description of its goals, the contents, notations, tips and tricks. These can be found by our project engineers in the company's intranet.

Table 1: Overview of the analysis modules.

Section	Module	Notation
Functional Requirements	Business Processes	ARIS
	Use Cases	UML / Text
	Analysis Functions	Text
Data	Data Model	UML
	Data Types	Text
User Interface	Dialog	UML / Text
	Batch	Text
	Printed Output	Text
Environment	Interfaces to up/downstream systems	Text
	Data Migration	Text / Tables
	Cut Over	Text
Essentials	Goals, Scope, Constraints	Text
	Non-Functional Requirements	Text / VOLERE Shell
	Cross-Cutting concerns	Text
	Overview of the technical architecture	Text / Graphics
Miscellaneous	Reading Instructions	Text
	Glossary	Text

3. Using the Modules

All seventeen analysis modules together form a comprehensive documentation of the analysis. Depending on the size of the project you decide on which modules are relevant for your system and on the level of detail to be used in the documentation. The following points may serve as a guideline on the level of detail:

- During **problem analysis** concentrate on the business processes and the important non-functional requirements. Do a first informal shot on use cases, dialog, data, printed output, interfaces, migration and architecture.
- When doing a complete **system analysis** work on all modules. Depending on the importance of the contents of the module for the system you decide on the level of detail. E.g. for a system with a complex GUI the dialog module is done in detail while dialog patterns are sufficient for a data maintenance system.

Our approach might look like a heavy-weight, paper burden process in contrast to agile software development [2]. This is not the case. The idea is rather that of a framework in which software development can take place safely. In this framework agile software development has its place as well as any other development process. Much has been said about the pros and cons of extreme programming. In spite of many beautiful ideas extreme programming will never be an alternative to classical process models. It can be played only in small groups of not more than seven persons and needs to be managed rigorously. As projects get larger you need a heavier process (see also [2]) and have to spend more effort on documentation. The analysis modules are an instrument for documentation of results and communication between teams in large projects, let them use agile software development or a heavy-weight process.

The main problem about analysis is cutting the system into appropriate pieces, i.e. components. We are convinced that the decomposition into components from a purely application point of view is crucial for project success. The worst case would be designing a monolith: an unstructured bag of thousands of interwoven classes. This is a disaster not only from a technical point of view (beyond a given size the system won't compile) but even more from a conceptual point of view: class diagrams taking up a wall are of no use at all. Nobody understands them and errors occur frequently.

The first step for finding application components is based on a list of the use cases. These are grouped by business context indicated e.g. by having the same actor, being derived from the same business process or operating on the same data. The decomposition is checked against the data model as soon as we have a first sketch of it. Entity types within one application component are usually highly connected while there are few connections between entity types of different components.

The decomposition into components also means that for some of the modules we have one instance per component. This is the case for use cases, analysis functions, data model, dialog, batch and printed output. The other modules are completed once for the whole system only.

The following sections describe the modules and the use of UML and other notations.

4. Functional requirements

Functional requirements have been the focus of analysis methods since the early days of software

engineering. There are three analysis modules dealing with functional requirements: *Business Processes*, *Use Cases* and *Analysis Functions.*

4.1. Business Processes

A business process describes the activities within the business and how they relate to and interact with the resources in the business to achieve a goal for the process. Workflow runs through different parts of the organization, crossing software product boundaries. The UML notation for business processes is an activity diagram. This is also the diagram suggested by the Rational Unified Process (RUP) [8]. The diagram elements used for processes are activities, states, objects for the data and swimlanes for the organizational structure. We do not use UML for business process modeling, because this notation has the following drawbacks:

- The use of activity diagrams for business processes is not strictly defined. Hence you have to find your own specific guidelines or use extensions such as suggested in [6].
- The granularity of objects is too fine-grained for business processes. Use stereotypes such as <<document>> or <<information>> on the objects to delimit them from objects in the object oriented sense.
- Especially when analyzing existing processes it is helpful to show the systems used. Again you have to customize the UML with your own stereotypes.

Our favorite notation for business processes is the ARIS method [10]. In this method a process is modeled as a sequence of activities (called functions in ARIS) and events. Inputs and outputs are attached to the actions by two symbols: one symbol depicts an entity type, another one a so-called “data cluster”. As during process modeling you are not (yet) talking about entities, it is valuable to have an element for logical data unit as the data cluster. ARIS also provides symbols for organizational units and systems.

As described here, we use the term *business process* for a larger process, we do not use the term “business use case” (as e.g. [3]). The term *use case* is used in the sense of an “application use case”, i.e. for an actor interacting with the system under discussion to complete a given task. You find the use cases from the business process as a part of an activity, one activity as a whole or several activities together.

4.2. Use Cases

For use case definitions UML provides the use case diagram with actors, use cases, extend and uses relationships. While use case diagrams are suitable to give an overview of the requirements, they are not sufficient for a use case specification and need to be supplemented with a textual description. For this description our module provides a Word template, because this is more comfortable than text documentation in a CASE tool. The structure of the template is similar to that suggested in many books (e.g. [7], [3]) containing the subjects preconditions, actors, normal flow, variances, fault scenarios and postconditions. Depending on the level of detail of the specification, this template is customized by the projects. Templates like this are part of many of the other modules, too.

Remember that the use cases are the major source for finding application components and that each component requires an instance of this module. The same applies for the analysis functions that are introduced in the next section.

4.3. Analysis Functions

An analysis function is used for the description of complex processing and computation. The description of a function should contain pre- and postconditions, effects and messages, performance aspects and a description or algorithm for the problem solution. There are two reasons why we suggest not to put the analysis function in the use cases:

- The use case description will be more compact and easier to read.
- The functions will not get lost in the use case descriptions, they are easier to find for the developer.

UML does not provide any elements for analysis functions, in compliance with OOA it provides methods for objects only. Please note that we see a difference between analysis functions and methods in a class. The analysis functions are a source for determining the methods you will need, in fact one step in an analysis function may be a method. Still, analysis functions do not have the same granularity as methods and it is not yet necessary nor advisable to assign the analysis functions to classes. As we will also discuss in the data section, during analysis the focus is on understanding the customer's business. The structure you find during analysis is a structure from a business point of view, not from a technical point of view. As UML has no elements for analysis functions, the functions are specified in a template-based text document in this module.

5. Data Requirements

As the functional requirements, so have the data requirements been the focus of analysis methods. There are two analysis modules in this section: *Data Model* and *Data Types*.

5.1. Data Model

During analysis we use a data model for the specification of the business data. This model contains the major entity types, their attributes and associations. As UML is a standard notation and CASE tools are widely spread, we suggest to use the UML class model as a notation for the data model.

In OOA the class model is the heart of the analysis. The basic elements of OO are the classes with attributes and methods, interfaces, instantiation of objects, object identity, generalization and polymorphism. Which of these elements do we need in analysis?

For a data model we need classes, attributes and associations. Generalization and aggregation have proven useful. We recommend to provide a *data model*, without methods. Keeping in mind that the goal of the analysis is a proper description of the requirements that is readable for users and developers, it is not relevant to specify the functions as methods for a class, the definition of a function does as well. As the focus is on understanding the customer's business, the data model has a lot of highly connected entity types and is usually very complex. We structure this model by decomposing it into application components where classes are highly connected within a component and few connections exist between the components. There is an instance of this module for each of the components.

Not until design we assign services to the components (according to the use cases). Methods are used only in design.

How about the other object-oriented elements? In analysis it is obvious that entities are identified by one or more of their attributes, we do not need object identity. Interfaces are not necessary before design. Instantiation and polymorphism are purely technical, there is no place for these in analysis.

Summing up, the class model created during analysis looks a lot like an entity-relationship model. The UML notation is our preferred notation. Nevertheless, the notation for ER models is also suitable.

5.2. Data Types

UML does not provide special elements for data types. We see four kinds of data types: elementary data types, ranges, enumerations and structures. If you use UML in this module, use stereotypes such as <<range>>, <<enumeration>> and <<structure>> in your class model. Still, a condensed description of the data types is needed for the developers. When using a CASE tool we extract the data types with a homemade script. As UML is not helpful, we rather recommend a textual description in this module. We provide a Word template where data types are specified in tables corresponding to their kind.

Please note that the specification of data types should be done only in the advanced analysis phase bordering on design.

6. User Interface

There are different ways of interaction between the user and the system: The obvious one is via screen dialogs, specified in the *Dialog* module, the other are via *Printed Output* or *Batch* programs.

6.1. Dialogs

Two aspects have to be analyzed for dialogs: the static of the dialogs, that is what the screens will look like, and the dynamic aspect, i.e. what is the sequence of the different screens.

For the screen layout, UML has no special elements. Various tools have been invented by our projects, ranging from paperwork through stencils for drawing software to Visual Basic or HTML. It depends on the customer and on the system which of these methods is appropriate, so we leave this open in the module. Nevertheless it is always necessary to supplement the layout with text descriptions of the screen's contents referring the data model. We provide a text template for this part in our module.

The UML state diagram is used as a notation for an interaction diagram (IAD) as suggested by [4] for the dynamic aspects of dialogs. The elements of an IAD are states, transitions, actions, a unique initial state and a unique end state. In this module, the UML state diagram is a good notation. Another UML diagram for dynamic aspects in analysis is the sequence diagram. The problem with sequence diagrams is that these are unreadable for the client once the dialogs get complex. As dialogs in sequence diagrams correspond to classes there is also a risk with these diagrams to get into design too early. We therefore advise against using sequence diagrams in this module.

There are also several other ways for the specification of dynamic aspects. The low-tech way is to do it manually (in conjunction with paperwork for the layout): screens are painted on paper, user and developer play

through the different screens. The high-tech way is programming a prototype e.g. in Visual Basic.

One part of the dialog specification is also a style guide. We have filed the style guide in the cross-cutting concerns module, because it is significant for the system as a whole, while the contents in this module are component-specific.

6.2. Printed Output

Printed outputs are neglected in UML. With printed output we mean all kinds of serial letters, lists and reports the system will produce. In this analysis module, we specify the layout, the used tools, the quantity of the output and all kinds of organizational and technical requirements up to which printer will be used. Tools for the specification are reporting tools and mainly office software.

6.3. Batch

As for many of the other modules, there is no concept for batch programs in the UML. With batch we mean a program running without user interaction. Batch programs are identified in the use cases. However, for a batch program the use case description is not sufficient, as there are special aspects due to running without user interaction. Batch programs run anytime, especially at night and therefore need robust error handling.

To meet these requirements, we suggest a text documentation in the module, using the following structure: preconditions, input, output, data volume, robustness, influences on run time, embedding in work flow management system, interfaces, fault tolerance. A Word template with this structure is found in the batch module.

7. Project Essentials

UML is focused on a detailed level of analysis like class models. Aspects concerning the system as a whole and the project environment are neglected. Nevertheless these are essential aspects for the project success that are addressed in four modules: *Goals, Scope, Constraints (GSC)*, *Non-Functional Requirements*, *Cross-Cutting Concerns* and *Overview of the Technical Architecture*.

7.1. Goals, Scope, Constraints (GSC)

The contents of this module lay the foundations of the project. Hence they have to be collected in requirements specification or at the very beginning of analysis. A checklist for the contents in this module is found e.g. in the VOLERE Requirements Template [9], section product constraints: purpose of the product, client, customer and other stakeholders, users, relevant facts, assumptions and constraints.

The form of this module is written prose. Not only are there no UML models for it, it is also an entry point for the analysis documentation and a management summary. Prose is the form most suitable for this purpose.

7.2. Non-Functional Requirements

Even though non-functional requirements are a vital part of any analysis, there is no corresponding UML diagram. The VOLERE Template introduced in [9] is a useful checklist of non-functional requirements. For the documentation a scheme as the VOLERE shell suggested in [9] is suitable. The structure of the shell contains: a number, description, rationale, source, fit criterion, customer satisfaction / dissatisfaction, dependencies, conflicts. Hence again, text is the first choice in this module and a template is provided. Alternatives are requirement management tools or individual data bases.

This module is a container for all non-functional requirements. However there are some non-functional requirements needing a more detailed analysis that is written as a concept of its own. These are found in the cross-cutting concerns module introduced in the next section. It is left to the user of the analysis modules to decide on the module where he puts his requirements. As long as all requirements are concerned the modules serve their purpose.

7.3. Cross-Cutting Concerns

The OOA is focused on objects, thus taking a detailed view of the system. While we find some diagrams with an overview character in the UML (e.g. the components diagram), there still are no models for overall concepts.

Every analysis finds some cross-cutting concerns. Typical examples for these are style guide, authorization or data history. While different concepts are suitable for different projects, there are typical questions that have to be answered. In this module you find a checklist of relevant themes, writing instructions and best practice examples for the most popular concepts.

7.4. Technical Architecture

This is rather a reference to results of the technical group than part of the analysis document. It turns out however that in order to understand the analysis document, the client needs a rough picture of what his system will look like technically.

The two diagrams for a system overview found in UML are the components diagram and the deployment diagram. Nevertheless these do not give a condensed overview of the technical architecture of the system and its environment as a whole.

In this module we give an overview of how the new system is embedded in the existing applications, which application components there are, interfaces to up- and downstream systems and technical decisions that have been made. A graphical overview of all these aspects is part of the documentation. Instead of a UML diagram we recommend colored graphics with individual semantics supplemented with text explanations.

While this is not the core of analysis, this is still an important result and essential for the further success of the project.

8. Environment

Up to now, we have taken into account our own software only. In this section we dare looking beyond and find another three modules: *Interfaces to Up/Downstream Systems*, *Data Migration* and *Cut Over*.

8.1. Interfaces to Up/Downstream Systems

In UML an interface is a set of methods that are publicly available. However, the level of these interfaces is far too detailed for the analysis and the relevant aspects are not addressed. Among the relevant aspects on an interface description are: business background, online / offline, performance, exchanged data, data transformations and protocols. As this is very different from UML interfaces we again provide text templates in this module.

8.2. Data Migration

Most of the time the new system replaces a legacy system. Usually we have to take over the old data. In the data migration module we specify how the old data model and the new data model fit together and which conversions have to be made.

In UML this corresponds to a mapping between two class models using dependencies between classes and attributes, respectively. However, dependencies between attributes are not supported by most tools, though the UML standard allows them. Moreover mapping diagrams are very complex. It is easier and more efficient to use tables for the mapping than using UML. A typical mapping table has five columns: old entity type name, old attribute name, transformation rules and comments, new entity type, new attribute name.

8.3. Cut Over

Once we are done with programming there comes the great moment when we go live. As this has more of an organizational character than an analysis character, it is not addressed in UML. Nevertheless, the planning must be done during the analysis phase. In this module we use prose to describe the process we have decided on: the big bang strategy where we switch to the new software in one big step at once or maybe rather a parallel processing with the old system. Note that you may also find new requirements like interfaces with legacy systems in this module.

9. Miscellaneous

After all there are two more modules for the convenience of the reader: *Reading Instructions* and *Glossary*.

The reading instructions advise the different species of readers – like management, end users, developers – which parts of the specification to read. In the glossary we find explanations for application specific vocabulary. These modules are part of the informal face of the analysis documentation. No UML diagrams are used.

10. Mastering the Analysis

The result of the analysis consists of the documents for the analysis modules. In a small project these are put together in one document, each analysis module being one chapter. In a large project, however, it is hard to handle a document of that size, you need a master document and several documents for the different analysis modules. For OOA usually CASE tools are used. Yet UML models are found only in three of the modules: use cases, data model and dialogs. Hence it does not make sense to have a CASE tool as the master of the analysis documentation while only a minor part of the documentation is done with it. We recommend working document centered. Writing text permits a better view on the context of the problem for the writer as well as for the reader. Still, the text is more or less structured depending on the module and diagrams are used where adequate.

Concerning the technical aspect of handling analysis documentation, there are one or more document files. One of them, usually a MS Word document, is the master of the documentation referring all other documents and the CASE tool results. Sophisticated tools have been developed to handle the documents in large projects. A highlight among these is a set of scripts using a Rational Rose model as input and generating a MS Word document for the classes and data types and

MS Excel tables for data mapping. Finally a Word dictionary containing class and method name was created to keep Word and Rose consistent. Still, these are homemade tools and the tool support for a document centered analysis is still poor.

11. Conclusion

After all, we are left with a real mess: Most of our modules are text based, and the most heavily used tool is Word. This strongly confirms an assertion put forward by E. Denert in 1993: Software is written, not drawn [5]. Software is mainly text and so is its description. One picture is worth ten thousand words but 1000 pictures do not replace 100 pages of carefully written text. There is an obvious problem: MS Word (or any other Editor) is not made for managing large semi-formal documents.
We believe that there is still a long way to go. We need a system of rules which governs our analysis documents and a system of manual or semi-formal transformations helping us to get from informal documents to formal ones. A first – and rather simple – step will be to replace Word templates by XML schemata.
The analysis modules have already proven useful in our every-days work, accelerating analysis. In our future work we will conduct a similar study to find construction modules, i.e. modules for a design documentation. We are convinced to find a lot of more formal documents and models in this phase. These will be consistent with the analysis modules, some of them might even be generated. What looked like a paper-burden process in the beginning will finally turn out to accelerate our projects.

12. References

[1] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1999.

[2] A. Cockburn, *Agile Software Development*, Addison-Wesley, Boston, 2001.

[3] A. Cockburn: *Writing Effective Use Cases*, Addison-Wesley, Boston, 2001.

[4] E. Denert, *Software-Engineering*, Springer, Berlin, 1991.

[5] E. Denert, "Dokumentenorientierte Software-Entwicklung", *Informatik Spektrum* vol. 16, pp. 159-164, Springer 1993.

[6] H.-E. Erikson, M. Penker, *Business Modeling with UML*, OMG Press, John Wiley & Sons, Inc., New York, 2000.

[7] I. Jacobson, M. Christerson, P. Johnson and F. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Wokingham, England, 1992.

[8] P. Kruchten, *The Rational Unified Process*, Addison-Wesley, Reading, MA, 1998.

[9] S. Robertson, J. Robertson, *Mastering the Requirements Process*, Addison-Wesley, London, 1999.

[10] A.-W. Scheer, ARIS, Business Process Frameworks, Springer, Heidelberg, 1998.

[11] *Unified Modeling Language (UML), version 1.4*, Object Management Group (OMG), http://www.omg.org/technology/documents/formal/uml.htm, January 21st, 2002.