Independent Study Dissertation Module - CP4431 - June 1996

Coping with the problems during transition from Analysis to Design in Object-Oriented Methodologies

Olaf Merkert

MSc Student, University of Wolverhampton
O.Merkert@wlv.ac.uk

Abstract: *This paper deals with the problems that arise during the transisition from analysis to design. All impact factors like a supportive structure and contents of the analyis model or requirements of the following implementation environment are discussed. Additionally the transition process and the influences of current software engineering concepts and conditions are treated. Problem areas were identified through practical reports while solutions are taken both from practical experience papers and theoretical reports. The reader should be familiar with basic object-oriented concepts.*

Contents

1 Introduction

It is now nearly six years ago since I first got into contact with some kind of object-oriented approach to software engineering. How hard it is to master the object-oriented paradigm was one of the first things I learned, because my first lecturer behaved quite clumsy. His lessons were based on Betrand Meyer's "Object-oriented Software Construction" [Meye88]. Because this book lacks of a notation - Meyer recommends Eiffel with its pre- and postconditions as a specification method - the lecturer chose Petri-Nets to visualise every aspect of an object-oriented system. How awkward these "method" was for practical application could be seen during our one-semester lasting project. Because of his missing experience our tutor was more a burden than a help. Now, what has this story to do with this essay? It is supposed to back up my opinion, that it is not easy and definitely not trivial to use an object-oriented method. It is even harder if the method is not "pure" object-oriented, because it has to use for instance a non-object-oriented visualisation system or implementation environment. Despite the fact that problems start in the analysis - because *not everything is an object* - it seems to me that the main difficulties lie in the transition from the analysis to the design model[1]; mainly because the latter one has to reflect the specifics of the implementation environment. If this is not object-oriented, the developer has to cope with a semantic gap. I gained this experience during my last dissertation ([Merk95]) that dealt with the development of a relational database using the OMT method by James Rumbaugh et al ([Rumb93]).

[1] "*It has been learned over and over, that the translation from analysis to design is the most difficult step in producing quality systems.*" [Vayd95], p.445

The object-oriented paradigm started with oo programming in the 60s. During the 80s its concepts were put into software engineering and since the beginning of the 90s complete methods are available, which describe the whole development process. These methods often claim, that they are also applicable in a non-object-oriented environment. In this essay current reports are evaluated that describe which shortcomings during the transition from analysis to design were discovered in practise.

Possible solutions are taken from these reports and from other papers that seemed to comprehend interesting answers for the mentioned problems. Because we all know about the gap between theory and practice I mainly try to refer to practice reports more than to theoretical papers.

The first section of this paper tries to give a brief background and a short summary of theoretical concepts. The number of influences on the transition process is very extensive and therefore both macro and micro process elements are mentioned[2]. The next section contains a discussion of recommendations for the analysis model which is followed by the description of design model aspects. Subsequently the transition process from analysis to design is treated. While the both preceding chapters mainly dealt with static aspects this section contains primarily dynamic elements. Current problem areas in software development and their impact on the transition in object-oriented methods are handled in the fourth chapter, that is followed by an evaluation of popular concepts to cope with the enumerated problems.

The following abbreviations are used in the paper: **oo** = object-oriented, **OT** = object technology, **OOA** = oo analysis, **OOD** = oo design, **OOP** = oo programming, **DBMS** = database management system, **ODBMS** = oo DBMS, **PT** = prototype

2 Definition and Outline

Before the practical aspects are discussed a small amount of theory is necessary to understand what this essay is all about. Instead of providing a complete overview of the proposals of current oo methods I try to summarise important general concepts in the direction of the process, which has to be performed to get from analysis to design. I try to mention concepts that seem to be important but often overlooked. Furthermore the characteristics of analysis and design models that are relevant for the transition are specified.

2.1 The Role of Object-Orientation

> *"The increasing number of books and articles published on topics related to object technology (OT) lends credence to the fact that this discipline has come of age." [Panc95]*, p.33

Over the last three years the articles in computer magazines changed. The number of articles about the introduction of object-oriented concepts decreased in favour of reports about their practical application. This report - exceptionally - does not focus on the loads of advantages that were gained through its application instead of the usage of a classical method. I would like to describe the problems that arise and their solutions that were often not obvious and required an amount of time. According to the theme of this essay it is mainly about the transition from analysis to design, which is described in the *process* section. Which characteristics are to be implemented in the analysis model to support the transition to design and hints about how the resulting design model should look like are collected in the *models* section. Like the analysis and the design process both sections overlap in some parts.

2.2 The Process

In figure 2.1 the part of the software lifecycle that is relevant for this paper is shown. The process of the transition from an analysis to a design model is usually called design. The analysis model has to contain certain elements to support the transition. Moreover the design model has to incorporate concepts to establish the next step to implementation.

---

[2] Booch distinguishes in [Booc94] between the *micro process* that comprehends the daily development activities of a single developer or a small team and the *macro process* which is the controlling framework for the micro process. The terms are commonly known and their usage does not imply that this paper deals only with the Booch Method.

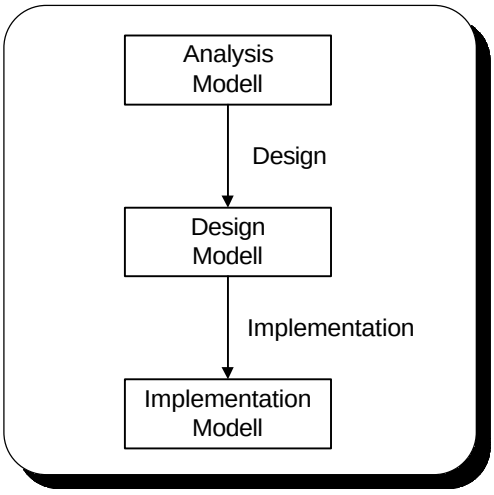

Fig 2.1 - Part of Software Lifecycle

The human creativity is an important point during both analysis and the transition to design:

> "*An interactive process for constructing models and implementing them [...] must be a framework that employs human creativity at many points.*" [Rumb96], p.14

Nevertheless practise has shown that "*a well defined process is essential*" [FaTs95]. Booch ([Booc94]) for instance is therefore describing a macro process for **determined** high level steps and a micro process for the every day development activities, which are performed with a higher degree of **freedom**. "*Many users tailor the process to their own organisations anyway*" [Rumb96]. Hence often the advantages of various oo process descriptions are combined. In the latest OMT release Rumbaugh proposes the usage of Use Cases as a first step. Booch describes the step from analysis to design as the invention of artefacts that provide the system's behaviour that was captured in the analysis model. Furthermore he stresses the importance of **traceability** that allows to prove that every required function is implemented. Another sometimes neglected aspect is the idea of **reuse** that can be realised with OT very well, because of its data&function encapsulation. Reuse concepts have to be implemented in the transition process. One of the most important questions that are discussed in common theories is: When to start the transition? Booch ([Booc94]) recommends to begin the design process not too late to avoid premature design, but early enough to avoid *analysis paralysis*[3]. He stresses that the aim of a complete and perfect analysis model is not reachable. This is also emphasised by Odell ([Odel96]) who says that an exhaustive approach is impractical and often impossible. Generally spoken has there be first an *identification* process during analysis that is the basis for the *definition* process during design ([PeCu95]).

2.3 The Analysis Model

The analysis model is the basis for the transition to a design model and therefore must have certain qualities. How they can be achieved and where the major problem areas are in practise will be described in the following chapters. In this section the basic theoretical ideas are summarised for clarity reasons.

> "*absence of domain knowledge 'reduces the task to manipulations in a formal abstract system' and results in poor design choices*" [Panc95], p.38

The citation above, which was taken from a practitioner, shows the major intention of performing an analysis phase. The developer has to "*model the world by identifying classes and objects*" ([Booc94], p.252) on a high level to develop an understanding of the system and its environment. Therefore methods like Jacobson's ([Jaco92]) propose the construction of an analysis model just for that reason and not as a basis for a design model as it is common in other oo methods. However key abstractions - objects and their relationships - in the problem domain have to be captured. Beside the static, dynamic aspects should have to be incorporated in a conceptual model. According to [Rumb91] it should be an "*external view of the system*" that is "*understandable for the client*" because only then it can be a "*useful basis for eliciting the true requirements*". Odell stresses the importance of representing the right objects: "[...]*the analysis goes beyond just looking at information to include anything or anybody of interest to the business*" [Odel96]. If the model is too packed it will restrain the design process. Moreover it should scope and **prioritise** system design areas as a basis for iterative development. Generally the analysis model should contain **what** must be done without restrictions to the implementation.

[3] This common known term describes the state of an analysis process that does not make any - or just a small - progress because of the claim to constrcut a perfect model of the system.

2.4 The Design Model

How the system will work using the identified objects from the analysis model has to be defined during the design. The basis for the following implementation has to be created. According to Booch there are two primary products of the design: a description of the *architecture* and descriptions of *common tactical policies* ([Booc94],p.255). The latter comprehend common procedures for: error detection and handling, memory management and data storage. He stresses the importance of a distinction between physical and logical structure. Rumbaugh accentuates that the design model has to be "*reasonable efficient and practical to encode*" ([Rumb91],p.260). Furthermore he states that some parts of the analysis model can already be transferred into the design specification without further modification. Because of its higher level of detail the design model is a complementary perspective of the system. It contains a mapping of conceptual models to implementation models and incorporates technical design decisions. Moreover usability, performance and resource requirements are important considerations during design.

The output of the transition process from analysis to design - the design model - is the input for the implementation phase. For that reason the design model has to consider characteristics of the implementation environment ([Rumb91]). The aspects discussed in this paper are not language specific but rather general notions.

3 The Models

The aspects that should be included in the analysis and the design model are the contents of this section. The things that are important and which ones are not is evaluated with reference to practical experience.

The advances made in both models are an important progress indicator ([Mala95]) and hence it is necessary to understand concepts that have to be incorporated. Additionally the authors of this paper recommend to use both models to "*facilitate evaluation of design alternatives and communication **within** development teams*". Therefore narrative text should be added to enhance communication. This is especially recommended by [Caro94] for the user interface considerations. Simplicity of the models is advised in [Mala95] because otherwise the rich notations will be "*overkill for co-ordinating development across the disciplines*". A possible solution is the comprehension of models from different methods. Furthermore modifications to the notation or the invention of a new one can be helpful. How and which methods are adapted is determined by the **project requirements** ([Mala95], p.40).

In [Kaas95] interesting theoretical aspects for the usage of the *abstraction* concepts are discussed. The author stresses the importance to separate the abstraction in the analysis of a problem domain from the usage of abstraction during the design of an information system. He presents extensions for both oo and structured methods that are not further discussed here for space reasons and their missing practical relation.

3.1 Views

The lack of "*larger scale bird's eyeviews of a system*" ([Vayd95]) is often mentioned as a reason for problems in both models. It is not only that the domain expert has to navigate through the analysis documentation; but also that important design decisions can be made only by a developer if high level views are available to consider larger system parts. This "*overall view of the system*" is also demanded in [Mala95]. They stress the importance of the population of a change in one view to another which can easy be achieved with a CASE tool. Each view should highlight a particular aspects of the model.

Notions that are not mentioned in both sources but that seem to be useful for the problems of larger views are the High Level concepts in OSA ([Embl92]) that provide abstract views for static as well as dynamic modelling. Another concept described in [AlFr96] is the layering of object models based on use cases. I consider their approach to be practical relevant because the authors are involved in the development of Select CASE tool. The layering is done by using the structure shown in figure 2. Each lower layer acts as server to upper layers.

During the use case modelling the distinction between business use cases and system use cases is supported through the model because they are put into the corresponding layers. With the use case approach the traceability of required functionality is supported in different layers.

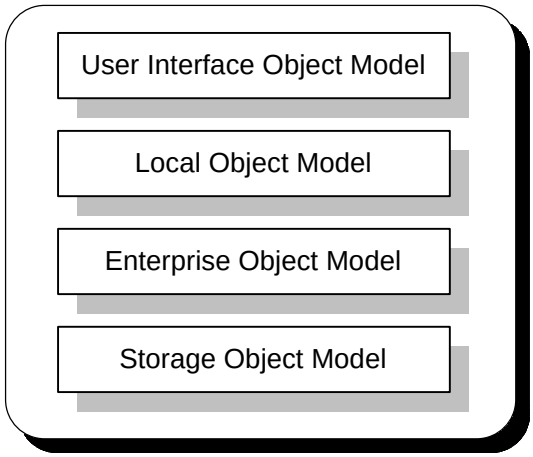

Fig. 2 - Model Layers (Source: [FrAl96])

3.2 Analysis Model

That the missing distinction between analysis and design is a big problem could be seen in larger projects as well as in smaller ones ([Vayd95]). If the constructed analysis model missed the aim to "*describe the system at a high level*" the following design process will perhaps be easier but the developed system does not fit the business needs. The domain expert and users can only understand and verify the model if it abandons implementation specific aspects. As a probable result errors in the basis for the transition may occur, because gaps and ambiguities in the requirements document are not uncovered ([Mala95]).

A missing relation between the modelling views for static and dynamic aspects results in a lack of clarity ([Vayd95]). If the designer is not able to relate the various analysis models he will not be able to develop an appropriate design model. On the other hand the misuse of existing modelling concepts during analysis can lead to a procedural system and other problems during the design. Beside too much coupling between classes instead of cohesion, inheritance hierarchies that are too deep will cause problems. They are often caused by applying more static than dynamic modelling. CASE tools are very useful to support the interrelation of the models for different aspects of the system. Furthermore they enable the user to work parallel on the diagrams to avoid biased modelling.

The authors of [Berg95] stress the importance of a "*Requirements Mutation Analysis*" to caption expected changes additionally to the functions the system should perform.

3.3 Design Model

> "*But if you look at mature object-oriented designs, they have lots of objects in there that don't reflect anything in the original domain model...They reify abstract concepts that you didn't think of initially when you were modelling your system.*" John Vlissides in [Panc95], p.42

The citation above highlights one of the most common errors. Even if the analysis model is a representation of the real world, the design can be heaped with abstract concepts. As a result the tie that has to be maintained between design decisions and specific entries from the requirements model is loosened ([Berg95]). The analysed flexibility requirements also have to be considered in the design model because it does not come automatically with the use of an oo method. One of the ways to achieve this is to integrate architectural issues in the model ([Vayd95]). They should address both the *logical application structure* which covers the decomposition of a system into subsystems and clusters and the *physical system architecture* that contains the process allocation to processors, physical DBMS models, communication topology and protocols. Furthermore a performance modelling is essential that is only feasible if knowledge about Client/Server and DBMS issues is available.

Additionally the developer has to use advanced features of the programming language in the design model ([Vayd95]). A language that supports for instance run-time type identification enables him to use container classes more easily while exception handling leads to more robust programs. A developer has to be trained to use the advanced features of an implementation environment to produce better designs. He also has to know how they are translated into programs.

4 The Process

> "*The means used to transform analysis models to design models is the 'black hole' in most extant methodologies.*" [Vayd95], p.444

In spite of the fact that the common oo methods give some hints to perform the transition current practise reports proved additional aspects to be important. The process structure showed up to be significant as well as the application of certain design concepts and early performance

considerations. The staffing of a project and psychological considerations have to be taken into account by the management, that has to back up the design activities of the single developer.

4.1 Structure

The "**analysis paralysis**" problem, that was mentioned in the first chapter, often occurred in real world projects ([Vayd95]) especially when analysts applied OOA for the first time. Vayda stresses the option of "*overlapped, parallel and iterative development life cycles*" that allows the simultaneous execution of analysis, design, implementation and testing. He promotes a "*fountain model*" that makes the results of later lifecycle stages available for the construction of the analysis and the design models. Nevertheless the application of a distinct analysis phase is important before the transition to design starts.

For oo software development an iterative process structure has been found to be very useful. One of the many sources for this statement is [Mala95]. The authors stress the importance of "*planning development cycles based on OOA/D models*". This has to be done in an early stage together with technical issues to avoid "*a major software restructuring*" in a late phase. During analysis increments should be planned according to functionality. They have to be constituted during design. Incremental development is quite supportive for risk management because early warnings are given. The authors of [PeCu95] describe the overall approach to object-oriented analysis and design also as "*highly iterative*". Hence **further change** is implicit in the design of object-oriented systems.

4.2 Concepts for Design

The following concepts are listed in [Vayd95] as hints to support the transition from analysis to design:

- chose containers for each logical relation in the analysis model for performance and space issues
- persistent classes have to be supported by a physical database schema - especially update procedures have to be addressed
- every class in the analysis model has to be mapped - with its whole life-cycle - on a process model (architecture)
- use a GUI framework, which supports model/view/controller[4] concept
- state machines should be transferred automatically via tools into code (table driven approach or tools like lex or yacc)
- interface definition with CORBA IDL[5], because it is implementation independent

To support the later step to implementation Vayda recommends the production of precise specifications through pre- and postconditions that have to be guaranteed. Also a set of invariants for each *class*, that is true for each instance all the time, is important. These concepts can be gathered under the title **safe design** because they ensure the construction of correct and robust systems.

4.2.1 Guidelines

According to [Vayd95] design guidelines should be used that contain "*heuristics for the major software issues*". These are coupling, cohesion, modularisation, inheritance, aggregation, association, interface design, default values, canonical classes, reference counting, recompilation avoidance, communication styles and database design. Furthermore quality standards should be addressed ([Faya96]). Guidelines are necessary to guarantee uniformity, maintainability and extendibility of the constructed software product. Also in [Mala95] guidelines for the design process are described as necessary. They should be constructed from external information and extended from the experience of internal projects. These guidelines have to be set up and their application has to be guaranteed. The latter one can be done through a mentor and/or reviews. Help to determine when a model is complete enough to continue to the next activity should be included as well as evaluation criteria and concepts to support reusability ([Mala95]). An example for missing guidelines is described in [Berg95]. The absence of a directive for an internal consistency check caused the destruction of a 4000 byte region by an array overrun.

---

[4] The *Model-View-Controller* pattern is very popular in the Smalltalk environment to uncouple the internal representation of an object from its external presentation.

[5] The Object Management Group defined CORBA as a standard for objects in a distributed environment. With IDL - the interface definition language - objects in heterogenous implementation environments can interact.

4.2.2 Mentors

> "*This project confirmed the commonly accepted importance of training and mentoring.*" [Berg95], p.59

The participation of an experienced consultant is mentioned in [Vayd95] as a solution to overcome problems especially during the construction of the analysis model. But according to [Mala95] engineers that have oo expertise are not only useful for this phase of the software lifecycle. To support the design process these mentors should be seed into teams. Furthermore "*management has to provide sustained support for mentoring through planning and rewarding time required*" ([Mala95]). This was successfully done in the project described in [Berg95]. To refine the use of the oo technique is an additional function of an oo guru involved in the analysis and design process ([Faya96]). He has to clarify and extend the methodology and update current organisational standards. Because of the impact of his decisions the selection of an appropriate qualified person has a high priority.

In weekly meetings the team can get feedback from the mentor. Various experts - which apply different inspection methods - are needed to review the design from different perspectives. These are: architecture, database, communication, algorithms and GUI ([Vayd95]).

4.2.3 Reviews

> "*Without formal interim reviews, the final product will resemble a United Nations meeting - several seemingly unrelated individual elements trying to work together.*" [Faya96], p.120

Reviews with a critical inspection of design and implementation are recommended by [Berg95] to exploit the oo incremental approach. They make sure that extendibility, inheritance on the right place and design patterns were considered in the design. Standardised concepts are often misunderstood or not used on purpose. To avoid this problem, regular meetings of the developers have to be performed. Walkthrough or peer review are advised in [Faya96] as review techniques. Inspection checklists proved to be useful in ensuring that "*several independent reviewers cover all bases*". The reviews are the most cost-effective technique to ensure software quality. Especially the quality of documentation has to be examined regularly. Another advantage of reviews is the support for communication amongst teams. When different teams work on the various subsystems they have to exchange experiences and discuss change requests for interfaces ([Mala95]).

4.2.4 Change Management

The introduction of changes to the design model from a certain point in time requires not only a special procedure like "*publish/subscribe*" that is mentioned in [Vayd95]; a version control that is embedded in a configuration management tool is also necessary. While code parts are usually recognised, design documents are often not appreciated as configuration items. According to ([Fayaa96]) the following items that are typical in common oo methods are relevant for a software configuration management: requirements, class diagrams, object-interaction diagrams, object diagrams, object-hierarchy diagrams, process diagrams, object/class design and test classes. The proposed iterative development process supports changeability but according to [Vayd95], p.441: "*allow changes only via a well defined and controlled set of mechanisms*". When reuse is propagated as recommended in using OT the change management becomes even more important ([Vayd95]) because versioning has an impact on more than one application.

4.2.5 Method Refinements

Because of the importance for communication across teams, processes have to be introduced to enhance it ([Mala95]). One of these is a **walkthrough**, that should lead to a "*supplementary documentation to explain the rationale behind the model*". Unresolved issues also have to be documented. The demanded understanding of every subsystem and their interaction can be blocked by missing documentation of aspects that seemed to be obvious to the developers in the team which introduced them.

Another change can be the reordering of development steps. Instead of the production of a static object model before the construction of an operational one like prescribed in the Fusion[6] method, the authors of [Mala95] report that it showed up to be useful to do it the other way round. They also applied the object model as a

[6] The Fusion method was developed at Hewlett Packard by Coleman et al. It was published 1994 and belongs to the cathegory of second generation methods.

mechanism for conceptualising and documenting the system during design through updating it during this phase. The enhancement made by the incorporation of use cases to the requirements gathering is already recognised in current method enhancements or even new methods like the unified method ([BoRu95]). In HP projects they were also used to "*explore different patterns of object interaction*" ([Mala95]). They were represented by timeline diagrams that were used during design in spite of the fact that Fusion does not suggest that. Also Vayda ([Vayd95]) advises the production, provision and prescription of notations to overcome the mentioned shortages in popular methods.

Not to neglect is the documentation of the tailored process ([Faya96]). Any step that is required has to be documented precisely to guarantee a "*bi-directional traceability from requirements models to code*" ([Faya96],p.118). To maintain the documents and to refine the process, a process improvement group is recommended. A simple rule is the following:

> "*The software process must be defined with sufficient detail that any competent developer outside the scope of the current project could correctly answer the question: What's next?*." [Faya96], p.119

4.2.6 Change of Methods

The authors of [Losa94] try to compare oo methods through the application of Coad/Yourdon ([CoYo91]) during analysis and the use of Booch ([Booc94]) for design. Both methods were originally developed for the use in the analysis respectively the design phase and hence have advantages for these areas. To me this approach is not very useful because the notations are too different to allow an easy transition from analysis to design. This is backed up by the experience that is described in [Berg95]: one team inside a project with many teams that used OMT applied the Shlaer/Mellor method. This team had to maintain a CASE tool on their own instead of taking advantage from the central Tool Support team and to organise its own training. As a result they missed an early deadline by three months. The conclusion made be the authors is that even if another design method is superior for a subsystem, the application of a uniform method in the whole project showed up to be more useful in practise. Reasons for that are communication problems during reviews and integration difficulties when team members were moved. As a result of applying Booch to analysis and Coad/Yourdon to design the authors of [BuAd95] report a decreasing scaleability, traceability and hence maintainability.

In [BrEv94] even the usage of a structured method for analysis and the introduction of OT in the design phase is recommended to get the benefits of OOP within a mature development process. Their argument - that was discovered in smaller projects - is a clearer distinction between analysis and design phase. Although this is a recognised problem I think that giving up OOA leads to the loss of more advantages than the method mixture would bring. Especially the "cradle to grave" character of the oo paradigm is very important. After his first guided oo project a developer should be able to distinguish between analysis and design without a method break ([Vayd95]).

4.3 Performance Tuning

The extensive use of oo constructs like for instance virtual functions has a high impact on the performance of the final system ([Faya96]). Loss in performance and an increased request for memory are the side effect of the constructs that support abstraction. Thus, the design decisions have to be evaluated according to their influence on the performance. A computer resource budget that is based on the capacity of the target machine in terms of their memory, CPU performance and I/O bandwidth has to be established to give a sense of direction during design ([Faya96]). Obviously every designer must understand the overhead associated with every oo construct in terms of execution time and memory requirements. One strong influence on the latter one is code bloat ([Berg95]). The strong performance of the system described in [Berg95] was caused by the performance engineering presence from the beginning of the project. They give the following design concepts to increase the throughput of an oo system:

- The assumptions that a method makes should be moved to its caller. As this decision implies an increased performance but leads to design that is not very general it has to be documented very well.
- Put code of the callers of a method into the method to move context. A *fatter method interface* instead of a very large amount of method calls is recommended.

- To avoid the overhead caused by the construction of new objects reuse them. To achieve this, object pools have to be implemented in the design model. In these pools, objects are not constructed and initialised but rather their state is adjusted to prevent data movement.
- Templates increase code size in a large extent. Isolating some of the code in a nontemplate will reduce the executable size and hence memory requirements of the system.
- Virtual functions in abstract classes cause overhead that can be avoided through moving the function definition down to its concrete derived class.
- Multiple inheritance is acceptable at the conceptual level but usually becomes less useful at the implementation level. Thus the design model should try to avoid this construct.
- The important performance increase through caches also has to be considered during design, because objects have to be assigned to cache areas.

All in all an observation during the project described in [Berg95] was that performance problems through the usage of OT were solved through the *ingenuity* of the developers.

4.4 Staffing

The authors of [Mala95],p.37 advise a small team for analysis that creates a high level architecture. Practice has shown that small teams are most effective. After an appropriate definition of subsystems these can be developed by subsystem teams. This expanding of staffing levels is following the "*natural fan-out of work*". Projects in HP have shown that teams that were too large at the project initiation left engineers "*on the sidelines, while a select group of key architects is given responsibility*". Beside unused resources the problem of lowered group morale appeared. The following phenomenon was observed in [Berg95]: The use of OT leveraged developers at highest intelligence levels but also empowered average ones with class libraries and frameworks to do activities they normally would not be able to perform. Furthermore the authors mention that it is hard for a developer who had gone through an oo training to do a non- oo job such as porting code. As a consequence management options in distributing people will be restricted.

4.5 Psychological Aspects

> "*Good concepts and techniques do not have to be forced on people who want to maximise effectiveness. If concepts and techniques are truly good, developers will fight to be able to use them.*" [Faya96], p.121

Not only in the first project that is solved with an object-oriented method psychological aspects have to be considered; also experience developers recognise these influences. OT offers good features to cope with complexity and to allow a natural structuring of large systems. Hence OT can be used effectively especially larger projects. This implies the incorporation of many developers in often more than one team. A result is that not everyone is as motivated as the people described in the citation above. Therefore the arising problems with these teams are not only technical but also psychological because "*political issues poison the organisation's ability to work on integrated solutions*" ([Vayd95], p.440). In [Berg95] the resistance of developers who are experienced in another paradigm is mentioned. They are not enthusiastic about the advent of a new one. Other engineers did well with the oo training and asked for a faster movement to development work. To make these teams work it is very important to promote team spirit and to post the skilful people as team leaders to support their interest ([Berg95]).

In the companies where no defined development process is established it is very hard to make independent workers to operate as team members ([Faya96]). They try to intensify the other member's first impression that the oo process requires "*significantly more work for small benefit*". The authors of [Mala95] mention the friction in a team caused by resistors to OOA/D methods. One result is an undermining of communication. Beside the mentioned technical disadvantages the lack of standardisation has also a negative psychological impact on the team members. If alterations to a method are performed, the team members tend to "*lose discipline and rigor*" ([Mala95]). This can be prevented by rigorous documentation of the allowed changes.

4.5.1 Confidence

A way to establish acceptance in a team is described in [Mala95],p.36. After three weeks training the developers felt too unsecure to achieve good results. But "*after the six week pilot the engineers were confident that, with ongoing mentoring*

from resident oo experts, periodic reviews with an outside consultant, and good design and programming guidelines, they were well prepared". A phase without pressure to reach targets showed that the introduced framework was usable. The required confidence of the developers was the result. Additionally the application of special design aims has to be made popular amongst the team members.

4.5.2 Rewards

A proposal to stimulate the utilisation of reuse in a company trough a reward system is mentioned in [Vayd95],p.448: "*For each line of code that is reused, its developer gets paid a nickel and the reuser gets a penny. The company where this approach was used spent about $60K and saved $430K in the first year.*"[7] The reward system proposed in [Berg95] is based on schedule and the defect performance of the individual subsystems of different developers. The lines of code (LOC) measurement that was used for the complete system was not announced to avoid drawbacks for reuse.

5 Current Problem Areas

The current software engineering world has many problem areas. For the design process the consideration of legacy systems and the construction of user friendly software are very important.

5.1 Legacy Systems

These days it is often not possible to develop systems in the green field. Legacy systems have to be implicated in the new program structure. The following strategies are mentioned in [Vayd95]: Incremental engineering, Database Conversion and Wrapper Construction. All of them often have to be considered during the transition from analysis to design. Especially the wrapper construction implies a design decision that can lead to significant disadvantages. The sometimes-recommended solution of one single wrapper for the complete system leads to the loss of maintainability, extendibility and clarity ([HoSc96]). The recommendation "*recapture the designs for legacy systems when they are to be long-lived and updated*" ([Mala95]) is therefore also a reasonable first step if an incremental replacement of the legacy system is planned. The next phase is to build capsules according to the business objects of the application domain. These capsules use various interfaces of the legacy system ([HoSc96]) and implement - if a "dumping" of the old system is planned - gradually its functionally. Thus the demand to work out and document at least the interfaces of the legacy system ([Mala95]) seems to be highly recommended.

5.2 User Interfaces

Two neglected aspects that have to be considered for the design of a software system are friendly user interfaces and interface navigation as a characteristic of large scale oo projects ([Vayd95]). That they are important for any size of project is mentioned in [Caro94]. The transition has to be "*constrained by the contexts of user activity*" that are identified during analysis beside the functional requirements. This allows an identification of problem areas for further design work. The usage scenarios have to be documented continually in the development process to accumulate and develop the attained knowledge.

A wider description for the enhancement of the transition from analysis to design through the application of *environments for design* is given by Terry Winograd. He argues for their necessity during a modern software development process in [Wino95]. The main idea is the design of software that fits the user's needs from a usability point of view and I think that his ideas have to be considered during the design process that is based upon the analysis model. Firstly he mentions the "*shift of perspective, away from what the computer does, toward the experiences of the people who use it*". Beside the functional specifications, the efficiency of usage has to be designed. Therefore the software developer has to "*move from a constructor's-eye-view to a designer's-eye-view*". Beside, the mechanisms the human situations have to be focused. Design environments are needed that support a "*broader array of representations, including different kinds of conceptual models, mock-ups, scenarios, storyboards, and prototypes*". Especially the latter ones are needed to support a new way of system design. Instead of discussing formal papers with the user, example systems are used to demonstrate the realisation of analysed requirements. They also enable the developer to "*try something out, see what it does, make changes, and try again in a tightly coupled cycle*". They enhance the iterative cycles that are

[7] That the simple reward principle is a quite bipartite concepts is discussed in [Merk96]. One section of this paper is dealing with concepts to increase the motivation of developers during maintainance.

recommended for many object-oriented methodologies and support the rapid evaluation of design decisions.

5.3 Database Systems

Nearly every software system has to deal with the problem of persistent data. That means a way is needed to provide a mechanism for storing data and retrieving it at a later point in time. This can be done with the help of a database management system (DBMS). When an object-oriented software engineering method for analysis and design is used it seems to be reasonable to store the objects in an object-oriented database system. Because of the prejudice of immaturity and a view other counter arguments existing relational systems are preferred by many companies. This is the reality as it is described by experience reports. Which impact the usage of these systems has and what possible concepts and solutions are available is summarised in this chapter.

5.3.1 The Mixture of Paradigms

The oo view of the world is a paradigm while the functional perspective is another one. Also the relational model which is mainly based on sets, their relation and manipulation can be considered to be a paradigm. Today the mixture of procedural programming languages with RDBMS is very common. Even if 4GL languages support data manipulation through procedural commands, software developers have to cope with a semantic gap between the two implementation targets. In this paper the mixture on a higher level of the software development process is examined. According to [Grah93] this process is only efficient when either a conventional method or an object-oriented one is used completely throughout the lifecycle. This opinion is also backed up by the following citation about the negative impact of hybrid programming languages ([Panc95],p.38): "*Generally, the panellists thought that hybrids encourage bad habits. By obscuring the object structure, such languages can also prolong the development process.*"

5.3.2 RDBMS Experiences

> "*It would be easier to use an object-oriented database that can utilise this design to store and access objects instead of a relational database.*" [DeFu95], p.120

The above citation states the impact that the use of an RDBMS has on the design process. With an ODBMS the designed objects could be implemented with all concepts of OO. With an RDBMS a mapping that includes the loss of convenient notions on the storage layer has to be performed to make the object data persistent. An additional problem is the missing inheritance of structured and object-based[8] programming languages that often have to be used within a DBMS environment. Code reuse that could be achieved with this technique cannot be implemented [DeFu95], p.121. The very nature of structured programming languages "*robs the object-oriented paradigm a lot of its power*" [Panc95], p.38. Nevertheless an object-oriented design can be used to trace the multiple use of code that was "cut and pasted" [Merk95].

Another problem is the application of oo analysis and design methods in an environment where relational systems were already used for data storage. "*It was difficult for the developers with a strong relational database background to design based on objects instead of database tables.*" [BuAd95], p.73 If a developer has the RDBMS target still in mind it will be hard for him to analyse and design according to oo concepts. A resistance towards OT which is based on the perception that it incorporates an additional mapping effort can be the result.

5.3.3 RDBMS Solutions

Low level clues how to map the classes of an object-oriented design to relational tables are given in the literature (for instance [Rumb91]). That a RDBMS can be used in many different ways with objects and that the commonly known mappings can be performed nearly automatically is described in [Chun95]. The following integration concepts have to be considered during design at least as possible design decisions:

- **Services-centric approach:** To avoid the necessary mapping in an environment that uses the RDBMS as a storage for object data (object repository) this approach exploits stored procedures. During design, well-defined entry points for these services have to be defined.

[8] An object-based programming language supports data abstraction and classes while an object-oriented one also provides inheritance and polymorphism.

- **Database broker layer:** Arisen from the needs to map objects in a large scale, this idea promotes the construction of a broker which automates the mapping of complex hierarchical data in a flat relational system. The construction of that layer during design is considered to be very complicated. A similar functionality can be implemented in an **impedance mismatch resolver** that works with various relational systems. With this concept the oo design is DBMS system independent.
- **C++ class library framework:** Details of the persistence implementations are hidden from clients that use the persistent classes while transaction management and concurrency control has to be considered during design.
- **SQL-3:** With the latest version of the SQL standard that is currently just available as a draft the degree of semantic content of the stored data should be increased. This is achieved by features for defining new data types and operations on these types. Beside multiple inheritance and overloading, object identity is described as a concept that has to be implemented.

The concepts presented in [Klei95] are dealing with oo software support for that mapping and a distributed model that addresses these mappings in a larger environment. The described enterprise object framework supports the design process through its clear distinction between "*user interface, business objects and the database server*". That business policies can be implemented as methods instead of storing them as form definitions or database procedures enables the developer to use this policies and constructed objects can be "*subclassed and re-used*". To solve the problem of "*matching static, two-dimensional data structures with the extensive flexibility afforded by objects*" the framework provides a conceptual bridge trough a modelling tool that supports the developer in mapping his design model to an appropriate entity/relationship model.

5.3.4 ODBMS

That the ODBMS application is probably the most suitable solution can be seen in the summary of the presentation of the two IBM employees David Parkhill and Bill Friedmann ([Chun95], p.166). After the development of half a dozen commercial applications with OT and an RDBMS they state these systems are inadequate to "*meet the needs of enterprise object applications*". I support this thesis because only ODBMS offer today a complete exploitation of object-oriented concepts. With CORBA, designs for distributed systems can be constructed in a real *implementation independent* way. This standard will soon be supported by common ODBMS like ITASCA and GEMSTONE that are also the only oo data storage environments that store object data and operations while other products still follow the approach to put methods in applications.

6 Modern Concepts

Certain support elements for the design process that are quite popular these days where not covered adequately in common oo methods. They are important for the flexibility and agility through the modelling of real world concepts that are the aims of OT application. Besides their advantages for the transition process they comprehend some dangers. The benefits and pitfalls of possible solutions are summarised in this section.

6.1 Reuse

> "*In the best case, the results of a domain analysis may lead us to discover that we do not need to develop any new software, but can reuse or adapt existing frameworks.*" [Booc94],p.253

Today's designs should incorporate reuse to an extend that is as large as possible. Reuse implies a lower rate of errors but on the other hand it requires special steps during the construction of a design model. "*Many people still associate OOD with automatic reusability. We found that just because we used OOT, that did not make the code reusable"* [Faya96]. No matter if logical models or implemented components will be reused, a **facilitator** is necessary ([Vayd95]) to control the process and to act as a final authority. He has to be an expert in the application domain, because he has to understand the multiple software projects within the domain ([Faya96]). The authors of [Mala95] report that several divisions within HP today are using OOA/D to successfully develop domain specific frameworks for product families. These frameworks consist of both concepts and components and have to be well documented with OOA/D models. The "*use of a professional documentation writer to provide high quality documentation*" is recommended in [Mala95]. Furthermore the authors propose the

application of colour-coding to show design with-reuse. Legacy components and corporate as well as commercial library components can be highlighted in a *design-with-reuse view*.

I support the distinction between logical and physical reuse that is combined in a framework. The first one can be characterised by analysis and design **patterns** while the second one can be captured with the term **components**.

6.1.1 Patterns

Many of the problems that were solved by the production of software can be attributed to the same solution strategy ([HeSc93]). Therefore reuse starts already during analysis to identify known structures that can be mapped in a defined way to design constructs. This "*collection of received wisdom based on past experience" ([Rumb96*]) is currently developed and maintained by the "the pattern movement"[9] and is an important component of a software engineering method. Currently design patterns that have to be chosen by a skilful designer are the state of the art[10]. Nevertheless design patterns can be applied to support the design process in various ways. They are an eminent concept to cope with complexity during conversation. With one term classes, relationships between them and their behaviour, as well as the interaction in-between the pattern, are referenced. In [Berg95] this is called a "*language of discourse between developers*". Beside the time saving aspect during normal design reviews the learning curve is reduced when developers are moved to a different team to work on another subsystem.

6.1.2 Components

[Vayd95] recommends to examine each design class for reuse by either "*aggregation or inheritance from an existing set of in-house and externally produced classes*". This describes the starting point for reuse: the consideration of existing components during the design process. At the beginning this should be just external class libraries for GUI, container, persistence and communication. The second step is the design of reusable classes. They have to be truly general, widely reusable, documented and distributable ([Vayd95]). To achieve this, Vayda recommends to establish the infrastructure for reuse very early. The documentation of produced components has to be maintained according to guidelines. Alan Nugent states in [Panc95], p.37: "*That's what enabled hardware reuse- not necessarily the ability to make the components, but the ability to document them well.*" He considers software ICs not to be good because of the missing documentation that is completely there for every hardware IC: "*exactly performance characteristics, the physical requirements and every single operational characteristic it had*". Information about "*object reliability, performance, or resource utilisation*" is considered to be necessary. In [Faya96] the importance of encapsulation is stressed because *"if too many internal details are visible, designers will be less likely to use the part"*.

In [Appe96] requirements for externally or internally produced components are given. They arose from an oo project and were originally stated for modules but are also valid for the reuse of classes. A developer has to know them when he chooses a component for his design or when he designs components that should be reused later.

- The class must have an explicit interface, that can be used without knowledge about its internals. This interface is the basis for a reuse decision.
- The interface is clearly defined in terms of syntax and semantic.
- The class is responsible for the realisation of a well defined task that is easy to understand.
- Cohesion (internal relations/interactions) must be high while coupling (external) must be low.
- If a change in requirements occurs, the change of the class must be possible without much overhead.

Furthermore a pragmatic procedure is described that covers steps for a development *with reuse* and *for reuse*. The main idea is the development of a problem area model for the first application that should lead to reusable components. This model has to be developed through an OOA,

[9] The book [Gamm94] contains a huge collection of patterns that were identified during oo projects. Online information can be found at
`http://st-www-cs-uiuc.edu/users/patterns/patterns.html`

[10] A formal method approach that uses Djikstras guarded command language during the mapping of an analysis to design model is maybe a first step in the direction of an automated transition. But this is only feasible if enough logical standard constructs on a high level are available.

OOD and OOP phase. Afterwards the analysis, design and programming steps are performed for a special model that is a subset of the problem area model. The development process for another system in the same problem area can then make use of the components of the first one. These were designed using an oo typical bottom-up procedure that supports a higher level of reusability than a procedural top-down approach.

6.2 CASE Support

The extensive usage of a CASE tool is controversial. While the authors of [Faya96] state that even if the tools are often expensive they are essential, for the oo process in [Berg95] their use is only recommended for *capturing*, *checking* and *communicating* the results of an analysis or design session. The authors of the latter source report that during their project the classical whiteboard was the most useful because of the fundamentally social and creative character of the design process. It was not possible to keep up with the flow of ideas while capturing them in the CASE tool. As a result the team was slowed down because of tasks like opening windows or saving documents. Nevertheless the investment in a tool is an important message of the management. It demonstrates its commitment to the technology and the project ([Faya96]). In [BuAd95] the productivity increase with the use of the Objectory CASE tool is mentioned. Despite the learning time it was all in all more productive than a simple drawing tool.

Also method creators like Rumbaugh recommend in their latest papers the usage of tools. Important is "*a single internal model that captures all of the important design information*" [Rumb96], p.15. This provides a single repository that can be shared between design applications. Through a framework integration data can also be shared between life-cycle tools to support management aspects.

6.2.1 Problems

In [Mala95] the current state of tools is considered to be not sufficient. General problems like non-intuitive usage and bugs are mentioned as well as using model integration in the whole spectrum from low to high end tools. The authors of the paper demand the following features: flexibility, integration, team support, reverse engineering, reverse traceability and support for reuse.

6.2.2 Tool Selection

An indicator for the selection of a wrong tool is that it ends up as a drawing tool ([Berg95]). To avoid this the right tool has to be identified before its purchase. [Vayd95] recommends a careful evaluation, because repository features, reverse engineering and code generation are more important than the simple drawing support for a lot of notations. Other authors stress the flexibility requirements ([Faya96]). They demand an enforcement of OOT with tools that catch the faults of developers who are used to a structured or no programming technique during the model construction. Reverse engineering and automated code generation are low pay-off features and hence should not be decisively for the selection of the tool ([Faya96]). They are just marginally useful. The consistent application of the graphical rules, checking between diagrams and a data dictionary are more important.

6.3 Prototyping

All commonly known types of prototypes have an impact on the transition from analysis to design [WeSc95]. *Exploratory prototypes* are very useful to support the development and evaluation of usability features. Technical design decisions can be made after the construction of *experimental prototypes* that should especially be used for critical aspects like the usage of a new DBMS or communication subsystem. During a RAD development process the *evolutionary prototypes* support the incremental character of common oo methods. Instead of the both first mentioned types they are not thrown away but refined.

The influence of an implemented systempart on the design is also described in [WeSc95]. The authors suggest to use a spiral model[11] for every time box[12] cycle during development. With this approach each time box contains OOA, OOD and OOP activity. As a result the implemented parts can have an impact on the design in the next iteration.

---

[11] In contradiction to the waterfall model that requires the isolation of every phase in the development cycle the spiral model promotes the iterative application of each phase in the commonly known sequence. It was defined by Boehm in [Boeh88].

[12] A way to control the Rapid Application Development (RAD) development process is the usage of time boxing. For each time box a delivery date is defined that has to be kept.

6.3.1 Advantages

Despite the fact that it is a quite old article I would like to refer to [Grah91] because of its practise orientation. Graham describes a prototype as a specification that *amplifies communication* between designers, when evolutionary PT is applied. A prototype shows how the world *could* be on the basis of a analysis model that models how it *is*. This supports the construction of visions that improve the modelled business system. Graham talks about the "*grave philosophical error of assuming that the two groups [users and developers] share the same perception of the world and what is relevant in it*". The developer has to design the system's behaviour that is percepted by the user in a different way than it appears to the outer world. Prototyping is a bridge to simulate the behaviour of the internal design concepts and to verify them. Furthermore it improves reuse because nowadays a prototype can often be made up out of existing parts.

Finally prototyping increases job satisfaction through its support for incremental development. The team that works on the prototype is not only a design but also an implementation team that sees the results of their design work at an early stage. The motivation to refine this executable systempart is higher than the motivation to work on "just" a design model.

6.3.2 Requirements

It is important to perform and document analysis and design before any kind of PT is implemented [WeSc95]. Hence the application of the PT technique in an organisation has to be integrated in a *framework*[13] that describes its utilisation. Timeboxing and the spiral model are often mentioned in the literature as a possible solution. When evolutionary prototyping is applied *reviews* are very important, because designs that are appropriate for the prototype may not be appropriate for the full scale system [Grah91].

According to [Faya96] a prototyping expert is required amongst a development team to develop a software product rapidly. This is valid even for a semi-formal oo project. His activities must be separated from those of the other developers because he has to "*develop a software product rapidly and correctly without necessarily following any specific techniques*" ([Faya96],p.115). After he has evaluated the efficiency of different designs he communicates the design to software developers.

6.4 Formal Methods

Despite the fact that they promise a better analysis model and a reduced amount of errors during the transition from analysis to design, formal methods were not mentioned in any of the experience reports about OT that I evaluated. Various object-oriented extensions for one of the most common formal methods called Z are available, but they were not used in any of the larger projects. A probable explanation is their need for experts and a longer and thus more expensive development time. Nevertheless I would like to summarise the theoretical paper [Eldr95] which describes how the oo programming language Eiffel facilitates the link between formal methods and OT and how it supports the reduction of errors during the step from analysis to design. Although the paper is theoretical the mentioned concepts contain a lot of ideas that are practical relevant.

The author of Eiffel, Betrand Meyer, described his involvement in the development of Z as an important influence on Eiffel features that improve quality, correctness of the software and a more efficient development process. Because it is a programming languages it can be easier learned by a software engineer than a formal specification language. Nevertheless it contains assertions that increase reliability. These assertions can be defined during analysis and design with the expressive programming language. An integration of business object description into Eiffel specifications is described in [WaNe95]. Their notation is a possible way to reduce the lack of clarity because of the code language, that is a general major disadvantage of formal methods. The most important concept is the idea of *assertion guided program construction*. A program can be proved by a proof that is integrated in the construction process. Assertions provide a debugging support and they are the basis for exceptions. Furthermore reuse is supported through a better description of components.

The problem of formal methods is their lack of clarity for a user. Nevertheless they can be a way to produce analysis models that are more complete if they are integrated into other methods. A possible way is the formal specification of methods in a common oo technique. With Eiffel an oo

[13] *Framework* in this case means the definition and application of standard procedures as well as guidelines to support the developers.

implementation language is provided that can be used for specification during analysis and design. These can be implemented without a syntactical change and hence are less erroneous.

7 Conclusion

The areas that were discussed in this papers are quite wide. Nevertheless every developer should to be aware of the various impact factors on the transition from analysis to design, to avoid problems during the system development process. Software engineering practitioners observed new problems but also additional solutions during their current practise. Even if the mentioned solutions are not adaptable or not considered to be good at least the problem descriptions should be known.

The most important aspect is the necessity of skilled employees. If the developers do not know enough about the application of object technology and if not at least one experienced person is involved in a project, problems will not only occur during the design process. Nevertheless the transition from analysis to design is one of the anchor processes during the development. If its importance or the high probability for errors during its execution is underestimated, the complete system construction will fail.

Finally, after all the important concepts about the application of the object-oriented paradigm, we have to consider that we are not able to reach one of the most important provisions: The biases that are there because of our formal education have to be left behind in order to apply object technology in the most efficient way. Ted Kahn describes the prerequisites that should be there as follows: "*In fact, one of the kids wrote the program for me that I used as the basis for my dissertation research. He was 16 when he did that.*" [Mest96], p.26

8 Bibliography

I decided to use the first four Letters of an author with the year of publication as a reference from the text to the bibliography. If a book has two authors the first two letters of their names are used. More than two authors lead to the usage of the first one as reference which is similar to the et al notation.

[Appe96] *Phasenübergreifende Wiederverwendung durch den Einsatz von oo-Konzepten* - W. Appelfeller - OBJEKTspektrum 1/96 - SIGS Conferences 1996

[Berg95] *Lessons learned from the OS/400 OO Project* - W. Berg, M. Cline, M. Girou - Communications of the ACM, October 1995/Vol.38 No.10, pp.54-64

[Boeh88] A spiral model of software development and enhancement - B. Boehm - IEEE Computer, May 1988, S.61-72

[Booc94] *Object-oriented Analysis and Design with Applications* - G. Booch - Benjamin Cummings 1994

[BoRu95] *Unified Method for Object-Oriented Development* - Documentation Set Version 0.8 - G. Booch, J. Rumbaugh - Rational Software Corporation 1995

[BrEv94] *OO oversold - Those objects of obscure desire* - T. Bryant, A. Evans - Information and Software Technology 1994 36(1), p.35-42

[BuAd95] *Applying Object-Oriented Software Engineering Methods to the Development of Call Center Software: A Case Study* - J. Burgett, S. Adam - OOPS Messenger, pp.72-75 / OOPSLA '95 - ACM 1995

[Caro94] *Making Use: A Design Representation* - J. Caroll - Communications of the ACM, December 1994/Vol.37 No.12, pp.29-35

[Chun95] *Objects and Relational Databases* - J. Chung, Y. Lin, D. Chang - OOPS Messenger, pp.164-169 / OOPSLA '95 - ACM 1995

[CoYo91] *Object-Oriented Analysis*, 2nd edn. - P. Coad, E. Yourdon - Prentice Hall 1991

[DeLu95] *Using an object-oriented approach to the development of a relational database application system* - P. Deng, C. Fuhr - Information & Management 29 1995, pp.107-121

[Eldr95] *Facilitating the Link between Software Engineering Practice and Formal Methods* - G. Eldrige - Formal Reasoning in Software Development, October 1995

[Embl92] *Object-Oriented Systems Analysis:* A Model-Driven Approach - D. Embley, B. Kurtz, S. Woodfield - Englewood Cliffs, NJ: Yourdon Press 1992

[FaTs95] *Object-Oriented Experiences* - M. Fayad, W. Tsai - Communications of the ACM, October 1995, pp.51-53

[Faya96] *Transition to Object-Oriented Software Development* - M. Fayad, W. Tsai, M. Fulghum - Communications of the ACM, February 1996/Vol.39 No.2, pp.108-121

[FrAl95] *A Use-Case Approach to Layering Object Models* - S. Frost, P. Allen - ROAD January-February 1996 - SIGS Publications 1996

[Gamm94] *Design Patterns - Elements of Reusable Object-Oriented Software* - R. Gamma, R. Helm, R. Johnson, J. Vlissides - Addison Wesley 1994

[Grah91] *Structured prototyping for requirements specification in expert systems and conventional IT projects* - I. Graham - Computing & Control Engineering Journal, March 1991, pp.82-89

[Grah93] *Object-Oriented Methods* - I. Graham - Addison Wesley 1993

[HeSc93] *Wiederverwendung bei der Softwareerstellung für betriebliche Informationssysteme* - H. Heß, A. Scheer Information Management, Mai 1993

[HoSc96] *Sanfter Übergang der zahlreichen Altsysteme in die neue objektorientierte Welt* - F. Hoffmann, T. Scharf - Datenbank Focus 2/96, pp.8-14 (Objekt Focus) - it Verlag 1996

[Jaco92] *Object-Oriented Software Engineering - A Use Case Driven Approach* - I. Jacobson, M Christerson, P. Jonsson, G. Övergaard - Addison Wesley 1992

[Kaas95] *Abstraction and concretizing in information systems and problem domains: Implications for system descriptions* - J. Kaasbøll - Conference paper for Information System Concepts 1995, Marburg Germany

[Klei95] *Enterprise Objects Framework* - C. Kleissner - Proceedings of SIGMOD '95, pp.455-459

[Losa94] *Object-oriented methodologies of Coad and Yourdon and Booch: comparison of graphical notations* - Losavio, Matteo, Schlienger: Information and Software Technology 36(8) 1994, pp.503-514

[Mala95] *Lessons from the Experiences of Leading-Edge Object Technology Projects in Hewlett-Packard* - R. Malan, D. Coleman, R. Letsinger - Proceedings of OOPSLA '95, pp.33-45, ACM 1995

[Merk95] Diplomarbeit: *Entwurf einer Datenbank zur Dokumentation und Planung der Klärschlammentsorgungswirtschaft des Ertfverbandes* - O. Merkert 1995

[Merk96] *Softwarewartung: Situationsanalyse und Entwicklungsmöglichkeiten* - O. Merkert, Januar 1996

[Mest96] *It's Child's Play* - R. Mestel - New Scientist, pp.24-27, 13. April 1996

[Meye88] *Object-oriented Software Construction* - B. Meyer - Prentice Hall 1988

[Panc95] *The Promise and the Cost of Object Technology: A Five-Year Forecast* - C. Pancake - Communications of the ACM, pp.33-49, October 1995

[PeCu95] *Object-Oriented Analysis and Design: Realism or Impression?* - D. Pei, C. Cutone - Information Systems Management Winter 1995, pp.54-60

[Rumb91] *Object-Oriented Modeling and Design* - J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen - Prentice Hall 1991

[Rumb96] *To form a more perfect union: Unifying the OMT and Booch methods* - J. Rumbaugh - Journal of Object-Oriented Programming, January 1996, pp.14-18

[Vayd95] *Lessons From the Battlefield* - T. Vayda - Proceedings of OOPSLA '95, pp.439-452, ACM 1995

[WaNe95] *Seamless object-oriented software architectur: Analysis and Design of relyable systems* - K. Walden, J. Nerson - Prentice Hall 1995

[WeSc95] *RAD-OO - Ein Vorgehenesmodell für objektorientierte Software-Entwicklung* - W. Weibel, P. Schorn - ObjectSpektrum 6/95, pp.72-76

[Wino95] *Environments for Design* - T. Winograd - Communications of the ACM, pp.65-74, June 1995