Large Scale Object-Oriented Software-Development in a Banking Environment

An Experience Report

Dirk Bäumer, Rolf Knoll
RWG GmbH

Räpplenstraße 17
D-70191 Stuttgart, Germany

Phone: +49-711-2012-587
Fax: +49-711-2012-502
E-mail: dirk_baeumer@rwg.e-mail.com

Guido Gryczan, Heinz Züllighoven
University of Hamburg
Arbeitsbereich Softwaretechnik
Vogt-Kölln-Str. 30
D-22527 Hamburg, Germany

Phone: +49-40-54715-413
Fax: +49-40-54715-303
E-mail: zuelligh@informatik.uni-hamburg.de

Abstract

While many books have been published on object-oriented programming and design, little has been said about the overall development process. In parallel, evolutionary and participatory strategies have been discussed and used for years with variing success. We claim that combining object-oriented development with an evolutionary strategy which we call an application-oriented approach, will yield synergetic effects leading to a higher level of software quality, usability and system acceptance. This paper describes the various ingredients of our approach which are unified under a common leitmotif with matching design metaphors. A series of major industrial software projects serves as example and practical proof of the approach. We report about documents that have been produced and used within these projects and about the technical construction of the applications.

Keywords

object-oriented design, evolutionary system development, design metaphors, interactive software systems.

1. Introduction

Looking at recent and successful software projects and trying to identify the "success factors", two tendencies come to mind:

- an increased usage of object-oriented technologies and
- a change away from traditional phase-oriented life cycle models towards more evolutionary strategies which put a stress on involving end users.

But, over all, it seems that both tendencies can be found relatively separated from each other. This is somewhat amazing since object-oriented software development can be based from analysis to implementation on terms and concepts of the respective application domain. So, this type of approach to object-oriented system design will yield development documents and prototypes with features, objects and characteristics of the professional environment and language, the users are familiar with. In turn, these documents and prototypes are the prerequisites of a real integration of users into the entire development process.

We claim, that combining object-oriented design and evolutionary system development strategies with user involvement to what we call "application-orientation" will show a synergy of positive effects. Application-orientation will lead to application systems of high quality with respect to usage and software technology and a high level of user acceptance.

In this paper we will present our experiences applying an object-oriented methodology, that we and others have developed (cf. (Budde, 1992), (Bürkle 1995)), in a series of industrial software projects. In order to characterize the methodology few ideas can be seen as fundamental:

- We regard software design in general as being primarily a *learning and development process* with a strong emphasis on communication and the use of the professional language of the application domain.
- Our approach to object-orientedness centers around the terms and the daily work within an application domain and not its structural properties.
- Developing a taxonomy of the professional language in use in the application domain is a sufficient basis from which the object-oriented system design can start. Once defined, this taxonomy helps to improve communication and interaction between developers and users.
- An intelligible guideline is needed for designing and constructing interactive software systems. This guideline comprises the underlying leitmotif of *a workplace for skilled human work* and the design metaphors *Tools and Materials.*
- If the professional language of an application domain and the mutual learning processes are of major importance, than there is a need for a set of application-oriented document types which can be understood and used by all parties involved.

This approach to object-oriented software development is illustrated by experiences gained from an ongoing series of complex industrial software projects employing these principles.

2. Failure of the Conventional Approach

2.1. The Project Context

The aim of the overall GEBOS[1] project is to develop and install an integrated office system to support all tasks and services in the banking sector. The project is being conducted by a service center, the RWG in Stuttgart, responsible for some 450 banks. The banks carry out all their financial transactions via the RWG, and also draw on the RWG's central databases for information about clients, accounts, and other institutions like news agencies. The RWG currently employs a total staff of 470, the majority of

[1] The acronym Gebos stands for Genossenschaftliches Bürokommunikations- und Organisationssystem, which translates to Office Communication and Organization System for Cooperatives.

them engaged in production, marketing, and development. Some 120 persons are employed in software development, most of whom have so far been working on mainframes using Cobol, PL/I, and Assembler.

The first GEBOS project was launched in December 1989 at the instigation of the banks. Computer support was already available for certain areas of their work, but an integrated, uniform system had yet to be developed. Customer advisors, for example, were obliged to repeatedly enter and process account numbers or clients' addresses. Complex, customer-related work processes could only be carried out using a variety of different tools. And such application systems as were already in use differed greatly with respect to their user interfaces.

It was this unsatisfactory situation that led the banks to call for an integrated processing system with a uniform user interface, which would allow universal use of the data available. In the initial project phase, these requirements were to be realized in a selected application area: support for customer advisors in the investments section.

2.2. The Conventional Development Strategy

The conventional phase of the project lasted from December 1989 until July 1990. It was conducted by the RWG along the lines of a specially tailored life-cycle model (cf. (Boehm 1976), (Bürkle 1995)). The use of this life-cycle model was explicitly stipulated and had actually constituted the prescriptive basis of numerous previous projects, although these did, in practice, deviate from it. The characteristics and problems of such process models have been widely discussed, e.g., by Budde and Züllighoven (1992), and are not given further consideration here.

Two teams formulated system requirements from their own particular perspectives. One team elaborating an application-specific concept, and producing initial interface designs, while the other team specified a data and function model. The main characteristics of this design process were:

- The central aim was to automate the major work steps or working routines in the application domain.
- In order to overcome the different work processes and organizational structures found in the various banks, a major reorganization effort during the organizational implementation of the system was envisioned.
- In order to represent the working processes, the system was designed as a sequence of selection menus and fill-in screen forms.
- The traditional separation of the data and function models during technical design could only insufficiently be related to the application concepts. With both models, the strong decomposition made it difficult to preserve the relations between model and application.
- In line with the separation of data and functions the underlying design metaphor was a "window on data". This means, that the user was presented with the impression of viewing application data through various windows (implemented as database views) and having the means of directly changing these data (implemented as access functions).

- The gap between the application domain and the computer modeling process was not only evident in the different models built; it was also rooted in the thoroughly conventional organization of the project team. One team contained system analysts, not software developers. This group also included user representatives and consultants. The other team, which was responsible for modeling process control, consisted mainly of software developers. No provision was made here for establishing and maintaining a communication process between the parties concerned.
- There was no continuous cooperation between the groups involved. The cooperative design process was restricted to the so-called requirements analysis. The main group consulted here were DP organizers from banks, as the developer team preferred total to this group of persons. Although their common communication basis was of a DP-related rather than of a professional nature, it proved practically impossible to continue this cooperation during the technical phases of the project.
- Although, there was a strong urge of integrating application domain knowledge into the design process, there were little means for cooperation. The design documents were mainly technical, understandable to the developers team only. There was no adequate type of design documents for cooperation.

In April 1990, a prototype of the system was presented at RWG's own annual in-house trade fair: a menu-driven interface prototype without functionality. Its presentation failed to meet the users' expectation. The following three months were taken up with efforts to eliminate the deficiencies described above. In July 1990, however, the project management decided to abandon the project in its conventional form.

3. The Application-Oriented Development Strategy

After abandoning the conventional project, the RWG management took stock and analyzed together with the project team the reasons for this failure. It was then decided to test object-oriented development combined with an evolutionary approach. During the initial resumption period of the project in October 1990, this method was discussed with the core team, and adapted to suit the application context. Since the method with its application-oriented model formed the explicit basis of the development process for all subsequent projects, we will introduce it below.

3.1. The Setting of the Problem Space

First of all, it is important to understand the difference in the design task at hand compared to what the RWG had been doing previously. Looking at the reasons why the conventional approach had failed, it seems obvious that the development task was far too complex for the team to handle it in a traditional way. The complexity lies in two areas: the tasks at hand which have to be supported and the requirements for handling the system.

What characterizes the tasks is that they are open or not fixed in their sequence of (inter-) actions. *In addition, these tasks can be described as customer-centered, which* means that the various services and information requests of the customers are the driving force of the work place in question. The actual choice or sequence of these requests cannot be foreseen but have to be answered at one workplace by one advisor using one integrated system. Finally, the handling of the system should allow for an advisor to utilize it in a non-obtrusive way while talking with a customer.

All these requirements add up to a system which cannot implement a fixed routinized series of work processes but has to provide support for the various tasks at a workplace in a most flexible and convenient way. In the literature from areas like work Computer–Supported Cooperative Work, psychology, sociology, or epistemology, this type of work, its cultural and social importance and our failing ability to formalize it has been widely discussed (cf. (Robinson, 1993), (Suchman, 1987), (Miller, Galanter, and Pribram 1960)).

So, the key concept of our method is "support for complex tasks". These tasks demand what is called expert work done by qualified and trained persons. This type of work is what we, in line with many current economical concepts and strategies, see as the key potential of an enterprise. Utilizing this potential, an enterprise will hold or improve its position in the market. As a consequence, expert work needs to be supported and not to be rationalized. In the following we will show how to reconcile this goal with our concept of object-oriented approach.

3.2. The Fundamentals of Our Object-Oriented Approach

Object-oriented development starts with analysis. There, the fundamental decisions are made and the groundworks for the design of the future system are laid. What is *principally* overlooked or misinterpreted in analysis will rarely be detected or compensated by the other development activities. This, of course, regards basic assumptions or viewpoints and not individual requirements for a system.

Typical components of a workplace in a bank are: blank forms, memos, manuals, cheques, currencies, interest rates, folders, pens, staplers. Most of these things will be obvious to the developers, because they are tangible, others, like interest rate, are not. Still, the non-tangible things are very important "things" or objects of work at a banker's workplace. All theses things need to be there for the banker to do his or her job of e.g. client consulting.

Application Domain	*Object-Oriented Design*
Thing, Item	Object
Ways of Handling	Operation
Concept, Term	Class
Specialization, Generalization	Inheritance
Taxonomy	Class Hierarchy

Fig. 1. Concepts of the Application Domain Related to the Object-Oriented Model

Looking at a simple folder, we can identify some basic ideas. To this concrete thing ("this folder") we relate a general concept ("a folder") and a specific way of handling. We have all this in mind when we use the term "folder" and we need to have this understanding in order to identify a thing as a folder. Thus, we do not characterize a folder by its inner structure, but by giving it a *name* and describing the ways we can *work with it.*

The concrete things of everyday work are the starting point of object-oriented analysis and design. In our models, they are represented by objects. In our approach, it is obvious that objects are modeling elements which encapsulate related ways of handling and information that are meaningful in the application domain. So, this is a behavioral approach (cf. (Monarchi and Puhr 1992)) with a strong application domain focus.

Consequently, we design classes for every relevant concept in the application domain. These classes are related to each other according to the conceptual hierarchy of terms we are modelling. In our model, we can express specializations or generalizations of terms by using the inheritance mechanism between classes. As a consequence, the main structural design criterium for our programs is the *taxonomy of terms of the application domain.*

Figure 1 gives an overview of the main terms used in our approach. On the left, it shows the terms relevant for analysing an application domain. For each term there is a corresponding one on the right, where the components of object-oriented design are listed. The figure indicates the close relation between application concepts and object-oriented modelling elements.

So far, we have presented our main design idea. This does not mean, however, that there is a schematic transformation of every item, found at a workplace, into its object-oriented simulation. In the following section, we present additional guidelines that are needed to balance the modeling of the well-known things at hand with new means and opportunities of a software system.

3.3. Designing the Future System

Defining the main components of an object-oriented model is important. But in an actual software project, you will find too soon that this is not enough. The question arises, how to design the future system.

We have established a guideline for designing interactive software on graphic workstations, as this is the predominant platform for the systems, we build. This guideline has to go beyond the layout of a graphic interface, e.g. discussing the advantages of buttons over pull-down menus. It is of major importance that the developers can form a vision of the future system in use, thereby relating issues of form to the design of the system's functionality. The key to solving this problem has been recently discussed both in work psychology and in the Human Computer Interaction community (Maaß and Oberquelle1992) and will be presented in the subsequent section.

3.3.1. The Leitmotif

Characterizing the overall guideline for designing interactive application systems we use the term *leitmotif.* A leitmotif is an idea which underlies or permeates a piece of art, i.e. an artefact. Extending this notion of leitmotif, we use it in software design to characterize a basic but predominant idea that helps us to transform a model of the application domain into the design of the future system. This process deliberately pulls down the (temporal) separation between analysis and design, which other object-oriented methods still maintain: We use our growing insight into the application domain to design the software system, while the evolving vision of the future system focuses our analysis process.

Our *leitmotif* is the *well-equipped workplace for expert human work.* This could be a desk of a customer advisor in a bank or the workbench in a workshop of a craftsman. This leitmotif is detailed or illustrated by so-call design metaphors (cf. also: (Madsen, 1988); (Carroll, Mack, & Kellogg, 1988)). These design metaphors have to fit into the overall "picture". They help us to design the various components needed to equip and utilize a workplace (Budde and Züllighoven 1992).

3.3.2. Tools, Materials and Other Design Metaphors

We have chosen tools and materials as the main design metaphors. This distinction has shown to be useful beyond the realms of handcraft and is valuable for office work as well.

Frequently, we have found that people have little difficulties classifying the things they use in their office work as tools or materials. There may be discussions, whether an index card box is more like a tool or like a material, or it is noted that a pencil is a tool while we write but becomes material when we sharpen it. We will have a closer look at these problems. But in general, we can say that these design metaphors have proved conducive to systems that are both understandable and can be constructed efficiently (Budde, Christ-Neumann & Sylla, 1992).

On a software technical level, which will not be dealt with in this paper, we have elaborated a set of design patterns which can be used for technically specifying and implementing the design metaphors as software components (Riehle & Züllighoven 1995). As we have elaborated this basic conceptual distinction between tools and materials elsewhere (cf. (Bürkle, Gryczan and Züllighoven, 1995)), a short summary should be sufficient.

A *Material* is something, which in our work becomes eventually part of the result of our work. Working with materials means working with tools upon materials. Every material has its application-specific functionality and can be probed and transformed by adequate tools. Materials in a software system, however, are by no means mere passive components like the data in a database application. It is important to understand, that materials are always motivated by the work tasks of an application domain. They have an internal (hidden) structure and a state and they offer ways of handling that again strongly motivated by the application domain. Thus, an account should not have so-called generic operations like set_account_value and read_account_value, but deposit, withdraw or settle, which will change the value of the account when used.

We use *Tools* in our work to accomplish a task. They are the means of work for probing and transforming the materials at hand. Tools always show the materials worked upon and they give permanent feedback on both the way they are handled and of our progress of work. A tool thus both has a task-related functionality ("it is good for something") and a specific way of handling ("it must be easy to handle").

Tools and materials are not the only design metaphors we use. Obviously, tools and materials need a place to be put and stored. This is were the *environment* metaphor comes in. A familiar example of an environment for application software is the so-called electronic desktop of many workstation systems. There we spread our materials and tools ready for use or we store them according to our own principles and working habits.

It has also proved useful to introduce the design metaphor of an *automaton*. Automata encapsulate automatic processes which will run for a considerable time without human interaction. They realize programmable work routines or mechanical procedures which can be specified with a minimum of context information and external control. Once set, an automaton will run and produce a predefined result. In a bank office, examples of automata are print spoolers, mail routers, or an automaton for transferring banker's orders.

Our design goal is no simple simulation of the physical characteristics of tools, materials and automata as we find them in the "real" world. We try to identify the conceptual characteristics of these things and their contribution to the goals and intentions we try to achieve in working. In other words: an adequate collection of flexible reactive components will support a user in dealing with his or her tasks effectively and efficiently. So, as a result of an evolutionary process you will find on an electronic desktop tools and "small" automata which will offer new means and perspectives on materials.

This new or complementary software components will eventually change the work situation. But even if new and different workplaces evolve, these new software components should never be out of place or extraneous elements, but should become in time as familiar as the other objects and means of work.

3.4. The Evolutionary Process

So far, we have looked at the design of application-oriented product using the concepts of object-oriented software development. As said in the introduction, there is the further complimentary dimension encompassing the organization, execution, and management of the development process itself (Floyd, 1987).

We have said that understanding the work relations of a given application domain is central to the evolutionary design process. One major obstacle, we have identified, is the gap between the developers' and the users' worlds. The key to a solution is the complementarity of object-oriented design of the product and the evolutionary system design process. Key issues are the intertwining of analysis, design, and evaluation activities (Swartout and Balzer, 1978), and the explicit consideration of communication and learning processes (Floyd, 1992).

As a consequence, two essential elements of our evolutionary design process are:

- We base our development documents on the professional language of the application domain and continue to develop them further.

- We view the design of an application system as part of a continuous learning process between all the parties involved on the basis of prototypes.

What we hope to achieve thereby is the evolution of a common project language, based on the professional language of the application domain. All the design documents, discussed among the groups involved, use the terminology and forms of representation familiar to the application domain. This design process is driven by these application-oriented document types and prototyping (cf. Budde et al., 1992).
The original methodology comprised a small set of application-oriented documents (as described in (Bürkle, Gryczan and Züllighoven, 1995)), namely scenarios, system visions and glossaries (cf. (Rubin and Goldberg (1992)). In the subsequent experience report we will report on how and why we have refined these basic concepts.

4. The GEBOS Project

Having outlined so far the application-oriented development methodology adopted by the GEBOS project, we return to our experience report. In table 1 we summarize the main project stages of the first project and then we report on the development process.

End	Project Stage
Feb.1991	Development of a HyperCard presentation prototype. First sketch of the future system presented to management and project team. No end users involved.
Apr. 1991	First prototype combining handling and functionality presented at the RWG trade fair. It met broad acceptance by banking management and users. RWG management decided to continue project.
Aug. 1991	Working group of software team and end users of 15 banks established. Evaluation of extended prototype.
Mar. 1992	Redesign of the prototype in line with the increased technical competence of the team. Additional application-oriented requirements by the working group integrated. Pilot system installed at 2 banks.
Dec. 1992	Functional enhancement and partial redesign of prototype in close cooperation with future end users. More pilot installations at banks.
Jun. 1993	Official shipping of the system.
Jun. 1993 - now	Due to wide acceptance, RWG management decides to launch 4 more projects for counter services, bond trading, loans and self-services.

Tab. 1. The main project stages

4.1. Refining the Application-Oriented Document-Types

4.1.1. Scenarios

Our major document type for capturing the findings of the analysis was the scenario (cf. with different meanings (Rubin and Goldberg, 1992), (Rumbaugh et al.,1991), (Sharble and Cohen, 1993)). We use scenarios for describing the *current* situation at a workplace.

Based on interviews and discussions at various workplaces, developers have written short pieces of prose, covering work tasks, work routines and typical situations of everyday work. This has been done using the professional language of the application domain. It is crucial that these scenarios, once written by the developers, are then discussed with the experts of the application domain.

Very soon in the development process it became clear, that the load of scenarios had to be structured, in order to maintain the information captured. Thus, we developed a taxonomy of scenarios (needless to say, that this taxonomy makes use of object-oriented techniques). Figure 2 shows this taxanomy.

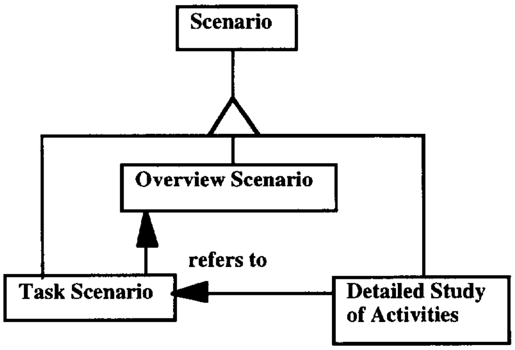


Fig. 2. Taxanomy of Scenarios

- Overview Scenarios provide (what's in a name?) an overview of tasks at a specific workplace.

  What was needed was the overall "picture", i.e. an overview of which people (or better "roles") deal how often with what tasks at which work places. It proved to be essential that these descriptions focus on the task and its goal and not on a temporal sequence of work activities. So this type of scenario helps to provide an overall understanding of the kind of tasks related to a person or a workplace.

- A Task Scenario describes an individual task with respect to its nature and the way to cope with it.

  The style of this scenario types is that of a script or a little "episode". Alternatives at decision points are hinted at, but sparsely. Frequently we had to decide whether in a scenario alternatives ways of dealing with a specific situation had to be described, or, whether different ways of dealing with a task should be described in different scenarios. A rule of thumb is, that simple decisions can be understood in one scenario, whereas different levels of alternatives are unsuitable to be captured in one scenario.

- A Detailed Study of Activities describes – not surprisingly – in detail, how a specific task or activity is handled.

  Most often this level of description has to be encountered only for activities that occur frequently in very different tasks. As an example consider how a form is found in a folder. By describing this activity on a very detailed level it gets clear, what the concept of a folder, i.e. the specific way of dealing with it, means in the application domain. On the other hand, this activity needs not to be described in detail in each Task Scenario, where, among other activities a form has to be found in a folder.

In the context of the Projects these kinds of scenarios have not only proven to be a helpful means for developers to increase their understanding of the application domain, but also they serve for the user management incorporated in the process as job profiles. Figure 3 shows an excerpt from a task scenario.

> The advisor then fetches his customer advice file, looks for the required product in the index , and opens the file at the desired point. In addition to the customer advice file, there is a form file in which standard forms (e.g., contracts with third parties) are deposited, and a specimen file in which completion guides for contracts and code sheets are deposited.

Fig. 3. Excerpt from a task scenario

4.1.2. System Visions

While scenarios are mainly used to document the work situation as is, we have used system visions to capture the design of the future system. They document the "vision" which evolves within the software team prior to the construction of prototypes. As was the case with scenarios, a single type of system visions proofed to be to coarse. System visions were written on different levels of details reaching from an overview of the future work situation in general to a detailed description of the anticipated system use for a specific work task. Basically, we distinguish between three different kinds of visions, that are further specialised (cf. Figure 4).

- General visions: They provide an overview of the systems functionality and how this functionality is embedded into its technical environment. More specific, an overview vision describes, to what extend tasks described in scenarios will be dealt with support of the future system.
- Procedural Visions: Whereas scenarios describe how tasks are performed actually, Procedural Visions describe how tasks should be performed in the future, with the aid of tools, automatons, and materials. Application Orentied PVs are written at an early stage of the visionary process. Their main task is to

serve as a means of discussion to shape the ideas that are more specifically described in other visions, namely Technical PVs and Handling Visions. To start with the latter, Handling Visions describe how tasks (usually described in Task Scenarios) are performed in the future. Technical PVs use textual description as well as CRC Cards to envision a first idea of the class structure of system components. Here, first decisions are made, where and how functionality should be implemented, i.e. as support for qualified work (which needs tools) or as formalized routine, which is implemented using automatons. PVs refer to

- Component Visions, where functionality of components is decribed independant of tasks. Here first sketches of tool-interfaces are drawn, the functionality of automatons and materials are described.

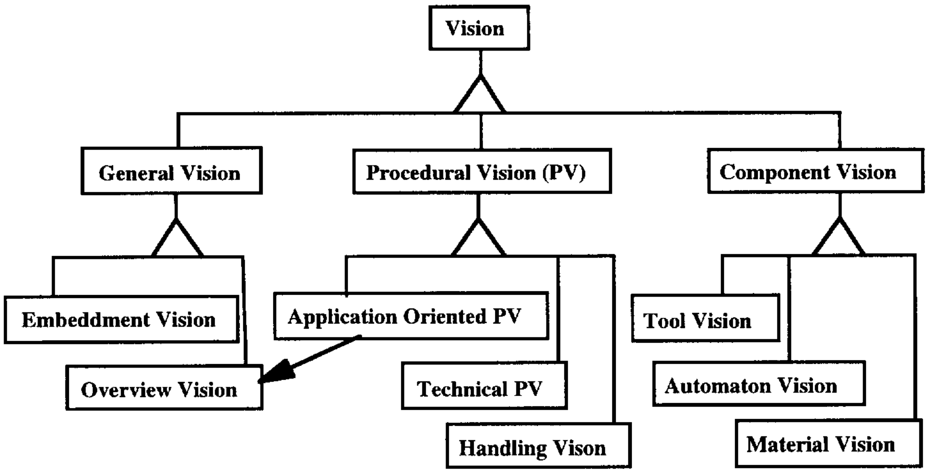


Fig. 4. Taxanomy of Visions

This taxanomy of Visions emerged from the needs of the projects we consulted. Having this taxonomy, it is not said, that each kind of vision has to be produced in every project. We stress, that these documents primarily serve as means of communication among developers. An example of a handling vision is given in Figure 5.

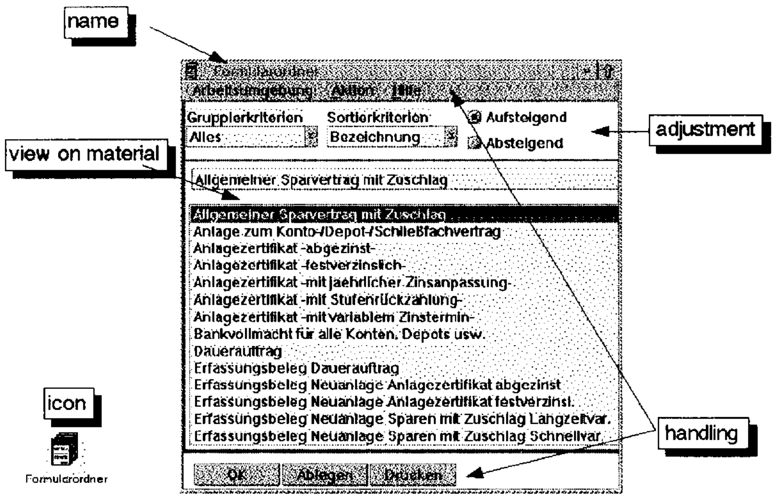
The advisor then opens his customer advice file by double-clicking the mouse, selects first of all the product from a visible table of contents, then selects the desired variant from a second table of contents by double-clicking the mouse again, thus causing the corresponding sales help facility to be displayed as well.

Fig. 5. Excerpt from a system vision.

4.1.3. Prototypes

System visions are the basis for the development of *prototypes*. Prototypes are preliminary versions of the future system which represent selected features. They are the fundamental for discussing design decisions between all parties involved and they are the most effective vehicle for the various learning processes. Different kinds of prototypes are needed according to the problems at hand (cf. (Lichter et al. 94)), e.g. presentation prototypes that give a quick first sketch of the future handling, functional prototypes showing both the user interface and selected functionality or "breadboard" prototypes that implement important design alternatives which are then discussed within the software team.

Naturally, prototyping is not restricted to object-oriented design. But what makes prototyping so attractive for our application-oriented approach is first of all its strong support to evolutionary design. As the mutual understanding of the design task evolves, so do the different prototypes. They always maintain the strong link between the application-oriented design metaphors and the architecture of the underlying software. So, when the user speaks of handling a customer advice folder and a money order, the developer will find matching tools and materials classes of the software. Figure 6 shows a screen shot of a browser tool for folders.



Allgemeiner Sparvertrag mit Zuschlag

Allgemeiner Sparvertrag mit Zuschlag
Anlage zum Konto-/Depot-/Schließfachvertrag
Anlagezertifikat -abgezinst-
Anlagezertifikat -festverzinslich-
Anlagezertifikat -mit jaehrlicher Zinsanpassung-
Anlagezertifikat -mit Stufenrückzahlung-
Anlagezertifikat -mit variablem Zinstermin-
Bankvollmacht für alle Konten, Depots usw.
Dauerauftrag
Erfassungsbeleg Dauerauftrag
Erfassungsbeleg Neuanlage Anlagezertifikat abgezinst
Erfassungsbeleg Neuanlage Anlagezertifikat festverzinsl.
Erfassungsbeleg Neuanlage Sparen mit Zuschlag Langzeitvar.
Erfassungsbeleg Neuanlage Sparen mit Zuschlag Schnellvar.

Fig. 6. A Browser Tool for Folders

4.2. Using the Application-Oriented Document-Types

In this section we summarize the integration of the application-oriented document types into the object-oriented development process as meanwhile established at the RWG for the GEBOS project family.

Every project starts by establishing an integrated working group of the development team and representatives of about 15 banks. All bank representatives are application domain experts and potential end users of the future system. Within this working group (qualitative) interviews are conducted, leading to the documents, just mentioned. Usually, most efforts in the initial stages of a project are put into writing scenarios and glossary entries and evaluating them in feedback cycles. In this way, *233 scenarios* and more than *1800 glossary* entries have been developed so far. They provide the basis for the subsequent user manuals of the various system components.

The design of the future system evolves in parallel, based on this documents. The central document type of the early design stages being the system visions written by the software team. While these visions emphasize the functionality of the future tools and materials, the design of their handling is supported by presentation prototypes, constructed with the help of an interface builder. Both system visions and presentation prototypes are used for discussing the design of application features and the handling within the working group. Questions of interface layout are of minor concern. The GEBOS projects have built *406 system visions* so far. During further development, system vision relate functional requirements to technical design decisions.

When the mutual understanding within the working group has stabilized, then the first functional prototype is built. Functional prototypes are used to design and test the actual support of work tasks. Beyond a certain point of maturity, a functional prototype is developed evolutionary into the system version that is shipped to users.

On the organizational level the task of the software team is supported by a common architecture group (cf. also Jacobson, 1992). The members of this group work as consultants to the individual GEBOS projects. Their major group tasks, however, are to identify emerging general concepts and integrating them as new technical components of the various class libraries, and encouraging the regular use of these libraries in their projects. The long-term goal of this activity is to arrive at an application framework for the banking business.

4.3 The Technical Framework

Classes and frameworks developed in the Gebos-Projects are structured on different levels. On the lowest level a Kernel-Cluster was developed. Its responsobility is to

- encapsulate operating system specific services (including services of the graphics library),
- implement a Meta Object Prototcol, and
- implement commonly needed data structures like different kinds of containers.

The kernel cluster does not include bank-specific classes. Table 2 summarizes the classes of the Kernel-Cluster.

These classes are used by all projects, to be more specific, by classes of the Gebos-Framework. The Gebos-Framework comprises all classes, that can be shared among different projects, i.e. these classes contain domain specific, but not application specific concepts.

The Framework is divided into a horizontal and a vertical component. The horizontal component divides the framework into sub-frameworks, which are responsible for a specific functionality. Examples for this kind of sub-frameworks are: electronic forms, domain values, and classes for the construction of interactive tools.

Parts	*Total*
Meta Object Protocol	31
Containers	60
Interface to Operating System	17
Garbage Collection & Tracetool	4
Late Creation	42
Interface to Graphics Library(CommonView)	60
	214

Tab. 2. Classes of the Kernel-Cluster

Each horizontal component is divided vertically into a concept- and a basic implementation part. They are responsible for:

- abstract protocols offered by the sub-frameworks and implemented patterns (for example the observer pattern (Gamma, 1995). This part of the framework is the concept part;
- basic implementations of the abstract protocols. Projects can use these implementations as is, but are also allowed to redefine the basic implementations. It is not recommended to change the protocols. This part of the framework is the basic implementation part.

Table 3 summarizes the sub frameworks implemented so far.

Sub-Frameworks	*Concept*	*Base*	*Total*
Domain Values	18	67	85
Test Environment for Domain Values	11	40	51
Converter for Domain Values	4	14	18
Container	9	23	32
Database Interface	8	20	28
Tool construction	41	162	203
Aspect classes	43	0	43
Interface to bank specific hardware	24	39	63
Interface for the host-system	23	38	61
Special interfaces for host transactions	0	66	66
Global Tools	0	105	105
Electronic Forms (incl. Tools)	20	70	90
Customer, Account, Product (incl. Tools)	13	46	59
debits	7	29	36
Offline-Framework	0	6	6
Bank, Employee	0	20	20
Total	221	745	966

Tab. 3. Sub frameworks of the Gebos-Projects

The basic implementation part of the frameworks is used to construct application specific tools, materials and automatons in the projects. Table 4 provides an overview of these projects and how many classes of which kind have been implemented there.

Project	*Materials*	*Tools*	*Automatons*	*Total*
Administrators Workplace	17	10	0	27
Credit	72	36	27	135
Database (Administration)	40	0	25	65
Ejour (Over night clean up)	0	2	0	2
Sending of Offline-Transactions	0	2	0	2
Offline Data-Management	0	2	0	2
Investment	34	10	6	50
Self-Service	19	58	7	84
Shares	42	66	1	109
Teller-Banking	30	58	26	114
Total	254	244	92	590

Tab. 4. Project specific classes

There is a clear tendency, that the number of project specific classes to design is decreasing from project to project. On the other hand this "return on investment" has to be paid by an increased work effort in maintaining and enhancing the structure of the sub frameworks. For that purpose the RWG has installed an "architecture group" which is responsible for adapting sub frameworks to project needs (cf. paragraph 4.2).

5. Evaluation of the Object-Oriented Approach

After more than 4 years of GEBOS projects and various projects completed or under way at other institutions, it seems clear that major application systems can be developed using the application-oriented approach presented in this paper. The GEBOS system has been shipped so far to more than 130 banks and is successfully used at more than 1600 workplaces.

What is recognized by both management and the teams, is the high degree of user acceptance combined with a level of software quality way beyond standard.

The intense cooperation between developers and users has encouraged the RWG to reengineer their organization according to the increased demands for user involvement, feedback cycles and smaller but faster software projects.

The tools and materials design metaphors have proved constructive in the design of complex interactive systems with high flexibility and changeability. These metaphors are the key to seamlessly linking models and structures of professional languages of an application domain via class design to the actual program code.

The benefits of the design metaphors are twofold. The similarity between technical and application-oriented concepts allow the designers to relate technical components to the work tasks, which are supported thereby. On the other hand, metaphors provide a new abstract level of speaking about design – they enhance communication and are part of an evolving design language.

Thus we believe, that the right choice of design metaphors, in our case tools and materials among others, is a prerequisite for the design and continuous further development of large application systems. The projects, we are reporting on, can serve as an empirical basis for the systematic use of metaphors as demanded by (Madsen, 1995). But, unlike Madsen, we do not believe, that using metaphors is a benefit in itself. Not every metaphor (cf. his examples of "TV-broadcasting" and "a meeting place") seems fit for bridging the gap between application domain and technical models. This bridge, we feel, is the key to building high quality and highly accepted software systems.

Acknowledgement. The GEBOS projects would not have been successful without the work of Karl-Heinz Sylla and Reinhard Budde who cooperate in developing the methodology presented here and helped substantially in establishing the early stages of the application-oriented project phase.

References

Boehm, B. W. (1976). Software engineering. *IEEE Transactions on Computers.* 25(12), 1226–1241.

Budde, R., Christ-Neumann, M.-L. , & Sylla, K. H. (1992). Tools and materials, an analysis and design metaphor. *Proceedings of the TOOLS 7,* 135–148, Englewood Cliffs, NJ: Prentice-Hall.

Budde, R., Kautz, K., Kuhlenkamp, K., & Züllighoven, H. (1992) *Prototyping – An approach to evolutionary system development.* Berlin: Springer.

Budde, R., & Züllighoven, H. (1992). Software tools in a programming workshop. In C. Floyd, H. Züllighoven, R. Budde, & R. Keil-Slawik (Eds.), *Software development and reality construction* (pp. 252–268). Berlin: Springer.

Bürkle, U., Gryczan, G., & Züllighoven, H. (1995). Object-Oriented System Development in a Banking Project: Methodology, Experience, and Conclusions. Human Computer Interaction 10 (1995) 2 & 3, pp. 293 - 336

Carroll, J. M., Mack, R. L., & Kellogg, W. A. (1988). Interface metaphors and user interface design. In M. Helander (Ed.), *Handbook of Human-Computer Interaction,* 1 pp. 283–307.

Carroll, J. M., & Rosson, M. B. (1990). Human computer interaction scenarios as design representation. *Proceedings of the Hawaii International Conference on System Sciences,* 555-561. Los Alamitos, CA: IEEE Computer Society Press.

Floyd, C. (1987). Outline of a paradigm change in software engineering. In G. Bjerknes, P. Ehn, & M. Kyng (Eds.), *Computers and democracy – a Scandinavian challenge* (pp. 191–210). Aldershot, England: Avebury.

Floyd, C. (1992). Software development as reality construction. In C. Floyd, H. Züllighoven, R. Budde, & R. Keil-Slawik (Eds.), *Software development and reality construction.* (pp. 86–100). Berlin: Springer.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995): Design Patterns - Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison-Wesley.

Jacobson, I. (1992). *Object-oriented software engineering – A use case driven approach.* Reading: Addison-Wesley.

Lichter, H., Schneider-Hufschmidt, M., & Züllighoven, H. (1995). Prototyping in industrial software projects. Bridging the Gap Between Theory and Practice. *IEEE Tranasactions on Software Engineering,* 20(11), (pp-825-832).

Maaß, S., & Oberquelle, H. (1992). Perspectives and Metaphors for Human-Computer-Interaction. In C. Floyd, H. Züllighoven, R. Budde, & R. Keil-Slawik (Eds.), *Software development and reality construction* (pp. 233–251). Berlin: Springer.

Madsen, K. H. (1988). Breakthrough by breakdown: Metaphors and structured domains. In H. K. Klein, & K. Kumar (Eds.), *Information systems development for human progress in organizations* . Amsterdam: North-Holland.

Madsen, K.-H. (1995). A Guide to Metaphorical Design. *Communications of the ACM, 37*(12), 57–62.

Miller, G., Galanter, E., and Pribram, K. (1960). Plans and the Structure of Behavior. New York, NY: Holt, Rinehart and Winston.

Monarchi, D. E.; Puhr:, G. I.: A research typology for object-oriented analysis and design. Communications of the ACM, 35 (1992) 9 S. 35 - 47.

Riehle, D., Züllighhoven, H(1995). A Pattern Language for Tool Construction and Integration Based on the Tools & Materials Metaphor. In: J.D. Coplien & D.C. Schmidt (EDS.), *Pattern Languages of Programs.* Addison-Wesley, to appear.

Robinson, M. (1993): *Design for unanticipated use.*In: G. de Michelis, C. Simone, K. Schmidt (Eds.) Proc. ECSCW '93. 187-202.

Rubin, K. S., & Goldberg, A. (1992). Object behavior analysis. *Communications of the ACM, 35*(9), 48–62.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-oriented modeling and design.* Englewood Cliffs, NJ: Prentice-Hall.

Sharble, R.C. & Cohen, S.S. (1993). The Object-Oriented Brewery. Software EWngineering Notes, 18(2), 60–73.

Suchman, L. (1987): Plans and Situated Actions; Cambridge University Press, 1987.

Swartout, W., & Balzer, R. (1978). On the inevitable intertwining of specification and implementation. *Communications of the ACM, 25*(7), 438-440.