Developing Reusable Software Components in CAM Environments

K. Bergner[*], P. Bininda[+], A. Blessing[+], W. Daxwanger[+], T. Krenzke[+], A. Rausch[*], O. Schmid[+], M. Sihling[*]

[+] *SEKAS GmbH, Perchtinger Straße 3, 81379 München, Germany, Tel.: +49 (89) 74 81 34 -0, Fax: +49 (89) 74 81 34 -99, E-mail:{bininda|blessing|daxwanger|krenzke|schmid}@sekas.de, URL: http://www.sekas.de*

[*] *Institut für Informatik, Technische Universität München, 80290 München, Germany, Tel.: +49 (89) 28 92 26 85, FAX: +49 (89) 28 92 53 10, E-Mail: {bergner|rausch|sihling}@in.tum.de, URL: http://www4.informatik.tu-muenchen.de*

Abstract. This paper presents some of the experiences gathered in the ESSI Process Improvement Experiment SEPIOR. The goal of SEPIOR is the systematic introduction of object-oriented and componentware development methodologies in the application field of computer-aided manufacturing systems. The paper focuses on the adoption of these technologies in the software company SEKAS. At the example of the development of an alarm management component the commercial, technical, and human impacts are analyzed.

Keywords: Componentware, Reuse, CAM, Process Model, Alarm Management

1 Introduction

Recently, the componentware development paradigm has gained much attention (Szyperski 1997). On one hand, approaches like COM (Chappel 1996) or JavaBeans (SUN 1997) promise to boost the performance of application developers, creating a fast-growing market for startup companies especially in the areas of GUI design and desktop computing. At the other hand, vendors of large enterprise systems like SAP R/3 are planning to implement modular versions of formerly monolithic software systems. In this experience report, we provide an example for the commercial, technical, and human implications of componentware in the field of computer-aided manufacturing (CAM) systems. The context is provided by the corresponding department of the company SEKAS (SEKAS 1999a) specialized on this application domain.

The main purpose of CAM systems is to control the fabrication process from raw materials to final products. This task requires the coordination of a variety of different activities, among them the calculation of production schedules, the control of production lines, machines, and transport facilities, but also the management of resources and tools, the gathering and logging of machine data and the management and propagation of errors and alarms. Modern, highly automated CAM systems can do all this with no or only minimal interaction by human users.

Currently, most CAM systems are custom-built programs relying on proprietary architectures and techniques. Apart from some established real-time operating systems and program libraries, standard components in this area are hardly available. A variety of different, non-standard

control devices and low-level programming interfaces exist. We expect, however, that this will change in the medium-term due to the overall trend towards standardized interfaces. Especially the upcoming standards for real-time middleware (OMG 1999a) and production system frameworks (IPA 1998) promise to enable the creation of configurable standard components with well-defined interfaces that are reusable in different CAM applications.

In this report, we describe the considerations and experiences of SEKAS during the introduction of component-based development techniques. First, Section 2 describes the expected strategic and commercial aspects of componentware, especially with respect to the involved chances and risks. Section 3 then sketches a part of the development process for the adoption and introduction of the new techniques. Based on this, Section 4 covers a practical application of the described process with an example of the development of an alarm management component. Finally, Section 5 gives some of the lessons learned during the process, concentrating mainly on the human impacts of the adoption.

2 Strategic and Commercial Aspects

The development of CAM software is a very demanding task, as it requires knowledge in distributed system architectures, and the ability to implement fail-safe software for a variety of different real-time controllers and devices. During the last years, SEKAS has gained profound insight and experience from a variety of CAM projects. The acquired knowledge in the areas of embedded systems, real-time software, and production control systems, as well as the achieved standards of software engineering and quality management are seen as key competences of the company.

Despite this successful history, experience has also indicated some recurring problems, among them the lack of standardized technical infrastructures. Due to this, even software realizing basic functionality had to be developed from scratch. This pertains, for example, to the implementation of several low-level communication libraries. The upcoming of standardized infrastructures, protocols, and components will make such efforts unnecessary, allowing SEKAS to accelerate system development and to concentrate on its core business. On the one hand, this may shorten the time-to-market for the customers of SEKAS. At the other hand, it may also leave more time for fulfilling customer requirements and implementing additional features. Furthermore, the use of standard infrastructures will likely have a positive impact on software quality and interoperability with other systems.

Another problematic issue is that the reuse level within SEKAS is currently very low. Essentially, every production control system was built from scratch and tailored to the actual customer requirements and technical infrastructure. Although the ability to adapt to a different technical infrastructure is seen as a strength of SEKAS, there is also consensus that a higher reuse level could raise the productivity considerably. If SEKAS succeeds in developing reusable, high-quality prefabricated components independent from a specific technical infrastructures, the effort for developing different versions of the same functionality for different infrastructures will be fundamentally lower, reducing the development costs and raising the competitiveness of SEKAS in customized software projects. To achieve this goal a clear separation between business-oriented and technical modeling has to be done, as the business-oriented model represents the part of the components independent from a specific technical infrastructure. Thus, it can be reused or even directly mapped to several infrastructures (Bergner/Rausch/Sihling 1997).

Furthermore, in the long-term this strategy may enable SEKAS to gradually transform into a component vendor, selling products on the evolving CAM component market. The shift from custom development projects to products will of course require careful preparation, as it necessitates a variety of additional capabilities and organizational measures, for example with respect to marketing and customer support.

The main risk with the sketched componentware strategy is the uncertainty about possible pay-backs for the costly development of reusable components. To cope with that risk, a careful analysis and selection of suitable components is necessary. Furthermore, most components evolve from actual projects which usually focus on special requirements due to the general lack of development resources and time. Thus, the decision to build a generic component has to be backed up by adequate funding, resources, and organizational measures.

To further evaluate the described approach, SEKAS has decided to participate in a so-called Process Improvement Experiment (PIE) of the European ESSI project (EU 1999). The specific goals of the SEPIOR experiment (SEKAS 1999b) are "to enable reuse of pre-fabricated software components by systematically introducing object-oriented technology, thus reducing development time and costs for customer-specific solutions, and to increase the quality of these solutions through reliable components forming building blocks for individual solutions". In a first step within SEPIOR existing parts of a CAM system should be refined and modeled as a reusable platform independent component. A team consisting of in-house domain experts and experts in object-oriented programming was formed. Additionally, experts in object-oriented methodologies and componentware techniques from the Technische Universität München collaborated in the SEPIOR team as consultants and coaches.

3 Process Model

In this section we present interesting aspects of an architecture-centric, iterative, incremental, and reuse-driven software design process (partly similar to the Rational Unified Process (Kruchten 1999) as successfully integrated within the SEKAS methodology. The main goal is the elaboration of a component-based architecture flexible enough to be adapted to and reused in numerous applications of a common domain. An software architecture of this kind mainly consists of two parts: a set of components and an underlying common framework gluing those components together (Broy et al. 1997). A framework provides standardized interfaces, classes and communication mechanisms and thus allows the components to interact with each other in the scope of a predefined structure (Pree 1997). Especially, components within this framework serve as a point of adaptation and configuration and can be conveniently adjusted or even be replaced without touching other components within the framework.

The presented process is divided into four essential activities:

1. Identify components and describe their functionality.
2. Specify component requirements, model component interfaces, and design the interactions between them.
3. Design the underlying, common framework.
4. Design the technical architecture and select a corresponding infrastructure.

Usually, these activities are interleaved and performed in an iterative and incremental fashion. After the design of the first component, for example, the developer might specify interaction

patterns and continue with the next component. After a couple of iterations, this process results most likely in a stable specification of components and their interfaces. Now, a first prototype containing the components and the underlying common framework is to be implemented. Finally, the concrete technical architecture and the infrastructure are selected and the prototype is re-implemented correspondingly. At this stage, non-functional requirements can be tested with the prototype.

Note the clear separation of business aspects and technical activities (second and fourth respectively) in this process. This is an essential requirement as stated in Section 2.

3.1 Identify and Describe Components

Best practice for the decomposition of a system into a set of interacting components is a combination of the classic top-down and bottom-up processes. On one hand, an iterative refinement of the system's design leads to a set of components which best fit the given, specific requirements. On the other hand, a bottom-up approach allows to consider possibly standardized solutions from a component market thus emphasizing software reuse. The ideal outcome would be a decomposition into two sets of components. The first one refers to components which are to be developed from scratch as they are strongly related to the core competences of the software company. Similarly, the second set are components from third-party solution providers which are to expensive to be developed on one's own.

The most relevant information required in this activity are the core competences of the company. Results from a thorough analysis of preceding projects help in finding the dominant knowledge which makes the difference to competitors. Core competences or functionalities which have been programmed over and over again are dominant candidates for components to be developed on one's own.

Larger enterprises often concentrate these efforts in form of special "reuse departments" which canalize demands from the different departments, forwarding component requests to in-house component "vendors". This virtual in-house market facilitates the decision whether components with similar functionality are needed by several projects, thus suggesting the implementation of an abstract, configurable component. This decision requires great care - if the efforts to adapt a very generic component to the actual needs are too high, it won't be reused.

By and large, the result of this step is the "big picture" of the domain under consideration. This contains a set of components to be developed, a set of components to be bought and integrated, and a raw description of their functionality and interaction in prose or by graphical description techniques like, for instance, the Unified Modeling Language (Oesterreich 1999; OMG 1999).

3.2 Specify Component Requirements, Model Interfaces and Interactions

After the components to be developed have been identified, a detailed analysis of each component's requirements and its relations to other components is carried out. This involves modeling the interfaces of all involved components using common description techniques as well as performing walk-throughs for selected use cases. Another main goal of this step is to balance the level of abstraction of the component. If it is too high, a lot of work is needed for adaptation; if it is too low, the component is not likely to be reused in other projects. There is no common solution for this problem. It depends on the experience of the developer to find

the right level of abstraction.

The results of this activity incorporate a set of use cases a component is involved in as well as specifications of the behavior and all of its interfaces. For this reason, the usage of popular graphical description techniques as offered by the UML is common practice.

3.3 Design Framework

The activities described above result in a set of abstract, business-oriented components. For most applications, this is not sufficient – they also need some common facilities that cannot be assigned explicitly to a single business-oriented component. This pertains, for example, to foundation classes that are used by a number of cooperating components, or to base mechanisms like persistence management. Furthermore, the framework may encompass certain domain-specific guidelines that have to be observed by all components or interfaces. This step is especially critical as the framework itself is not supposed to be changed during runtime. Thus, the services needed and provided by the framework must be defined at the appropriate level of abstraction. The main result is a class diagram specifying the framework and implementation classes for the components under development.

3.4 Design Technical Architecture

The last main activity is to capture the requirements for the actual technical architecture, and to select a corresponding technical infrastructure consisting of suitable middleware components. Usually, the separation of business-oriented and technical design leads to a clear architecture, as technical details of a certain infrastructure cannot influence the business-oriented parts. Furthermore, this approach allows to reuse a certain business-oriented design for multiple technical infrastructures.

The vision of the technical architecture design is, that designers annotate the business-oriented model with technical markers, like “persistent” or “multi-threaded” and then - using a predefined mapping between marker types and technical infrastructure - generate the implementation of the system using appropriate tools. Currently, this is still a vision. But a couple of tools already got quite close to this goal. For instance, the tool AutoMate generates from class diagrams a complete implementation for a CORBA and object-oriented database environment (Bergner/Rausch/Kuhla, 1998).

4 Application Example: Alarm Management

This section demonstrates the application of the process model described in the previous section at the example of an alarm management system component (AMS). This component is currently specified and implemented at SEKAS to serve as a reusable building block in future projects.

4.1 Identify and Describe Components

In several workshops and design sessions at SEKAS a total of seven CAM projects were analyzed as a starting point. They resulted in the identification of some overall component candidates whose functionality was needed in multiple projects. Together, they represent a large portion of the company’s core competences. Figure 1 shows a simplified, highly abstracted

version of the identified components.

In our model, an overlaying ProductionPlanningAndControlSystem (PPCS) component deals with the commercial aspects of the respective fabrication process, for example, product pricing and customer orders. Product information for the necessary planning, scheduling, and optimization of production tasks is available from the ProductManagement component (PM). The obtained information constitutes part of the individual production plan for a certain product. Furthermore, a production plan includes a bill of materials, production steps, machine setups, and so on. The PPCS delegates the computed set of optimized tasks to the LineControlSystem component (LCS) in form of orders for production lines and machines.

The LCS initiates and subsequently controls the production for a production line order. Depending on the degree of automation, the LCS may control the machines with or without using the Resource Management System component (RMS).

All products manufactured by the LCS are managed and tracked by the PM throughout their whole lifecycle. The LCS usually not only collects the product tracking data, but gathers also machine and production data. Finally, the Alarm Management System component (AMS) serves as a kind of global service and may thus be used by any component. The AMS is used to inform users about system errors or problems occurring during the production process.

The AMS is a suitable component for demonstrating the approach introduced in the previous section due to its importance–the corresponding functionality was necessary in five of the seven analyzed project. Furthermore, to the knowledge of the authors there is currently no AMS component commercially available that fulfills the requirements comprised in the subsequent section.

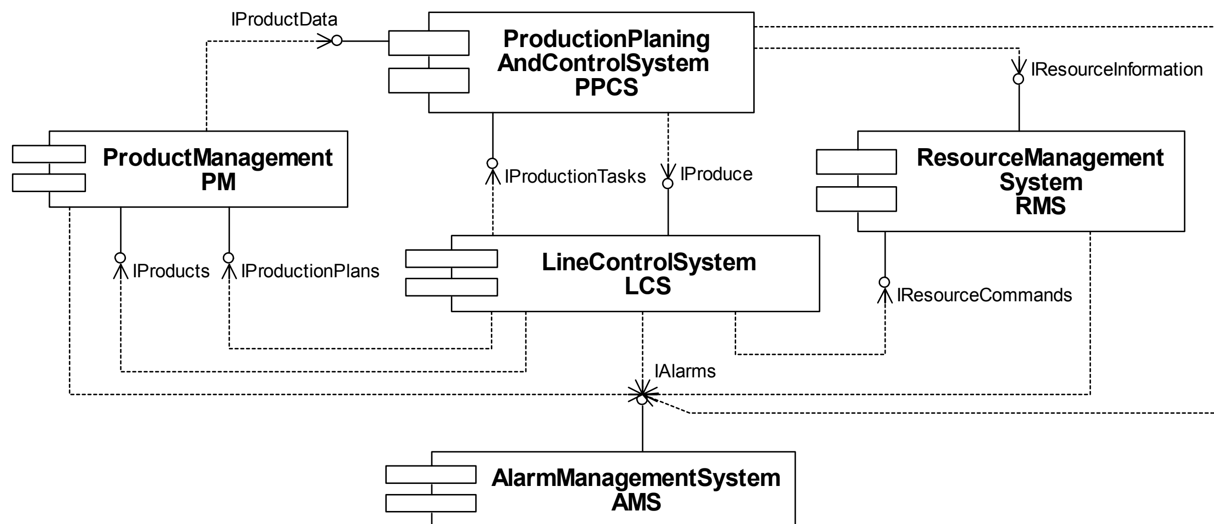

Figure 1. High-Level Component Architecture of a CAM System

4.2 Specify Component Requirements, Model Interfaces and Interactions

In order to emerge an appropriate design for a reusable AMS component, it is necessary to define the requirements:

1. *No loss of error information:* Error information sent to the AMS must not be lost. Under

all circumstances, at least a certain minimal error handling (e.g. tracing) must be ensured.

2. *Asynchronous error handling:* The process producing errors may neither be blocked nor may it suffer from any overhead imposed by error management.
3. *Freely configurable error handling:* Error handlers may be configured to handle a certain error or even a whole class of errors at a specific error level. Moreover, the definition of escalation strategies is possible: when the handling of an error at a certain error and escalation level fails, the systems switches to an increased escalation level with another error handling strategy (possibly employing different handlers). Dynamic configuration allows for a simple integration of new alarms with new handling strategies even at runtime.
4. *Platform independence:* The AMS will be used by a variety of different systems, each even running on a different platform. To achieve reusability, the AMS must be designed platform-independent.
5. *Transparent API:* The AMS interface should be shallow in order to be easily connectable to specific systems. Essentially, an interface for sending errors is sufficient–no further knowledge about the configuration and the possible error handling should be necessary.
6. *Error classification:* The AMS must allow for a context-sensitive classification of an error according to its severity. Furthermore, a reclassification in a different context must be possible.

The architecture of the AMS as shown in Fig. 2 reflects these requirements. The AMS handles errors asynchronously to the external system according to a configured strategy. Thereby, one or more handlers may be instructed to handle the error (for example, handlers for error tracing, printing, displaying error information on a pager, and so on). As there may be several AMS components, each serving another manufacturing area, it is possible to pass error information between different AMS components. A configuration facility may be used to configure the AMS with the specific error handling strategies.

The AMS component only offers two interfaces: one to connect the external system with the AMS (*handle*) and one to configure the alarm handling strategies with the help of a configuration tool (*configure*). The external system, acting as a client of the AMS, sends the produced errors to the AMS by using the *handle* interface. Optionally, the external system is able to log the error by itself, using the interface to a Logging component. The AMS instructs the different device handlers by using the appropriate device handler interface.

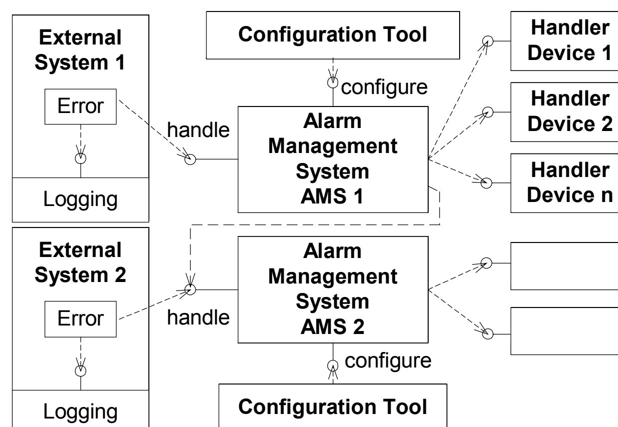

Figure 2: Architecture of the AMS

4.3 Design Framework

A CAM system is by its nature a distributed system. There are components running on a wide range of platforms distributed throughout a factory. The platforms include, for example, embedded real-time systems on production machines, real-time systems for production control, database systems, and Windows workstations.

Every component in the CAM system is a potential client of an alarm management system, since every component can encounter events that need attention. However, it is not desirable to have alarm handling done locally on every machine. An AMS should be manageable as a single instance and should marshal access to central resources such as beepers and phone lines.

Therefore, we need a practicable framework for the clients of the AMS, which is a part of the overall framework gluing together the components of a CAM system. Figure 2 reflects the system dependent framework of the AMS which can be utilized by any external system. An external system produces an error object and has the possibility to log the error information to a file. This error object may then be given to the AMS under use of the handle interface. As described above, the AMS then handles the error asynchronously to the external system.

4.4 Design Technical Architecture

As stated above, various components throughout a factory must be able to access the AMS. To enable communication, an appropriate infrastructure has to be established. The traditional approach within most CAM systems is to use TCP/IP sockets with a proprietary protocol in order to handle the communication between the components. This approach implies high development efforts and does not encourage reuse. The alternative is to use a standard infrastructure technology. We have chosen CORBA, because it is platform and language independent and widely available. Consequently, the AMS component offers its functionality via CORBA interfaces to its clients. The communication of the AMS component with its device handlers also employs CORBA.

CORBA was chosen over DCOM because it is an open standard with implementations available for all relevant platforms, while DCOM is primarily available on the Windows platform. In addition, production control and industrial applications are a traditional domain of CORBA, while DCOM focuses primarily on desktop components. For a detailed comparison of DCOM and CORBA see (Chung et al. 1999).

As stated previously, the CORBA interface of the AMS is language independent. Therefore, clients can communicate with the AMS regardless of the language in which it is implemented. However, as error handling is an integral part of the implementation of any component, language specific extensions of the framework are necessary. These framework extensions will be made available for C++ and Java, which allow clients to handle errors and to trace information with minimal programming effort.

The anticipated performance bottleneck of the AMS is the network communication between the client component and the AMS on the one hand, and between AMS and the alarm devices on the other hand. Therefore, the efficiency of the programming language used to implement the AMS is not an major issue. Consequently, we have chosen to implement the AMS in Java although we expect inferior performance compared to other programming languages such as C++. With Java, we expect a significantly shorter development time than in C++, especially

since memory management and the integration with CORBA are much less complicated.

5 Lessons Learned

5.1 Human Impacts

As mentioned above, the first step towards component development was the domain analysis of the CAM domain. Although the participating staff at SEKAS already had basic training in OO analysis and design, it was not easy to adapt this rather abstract knowledge to the concrete request to do a domain analysis. The training has to be complemented by suitable guidelines for the development process. Otherwise, the developers will be unsure where to start the analysis and whether the analysis covers all critical sections later on. Coaching from OO and componentware experts should be employed until the practical use of the learned techniques is adopted by the team members.

For the domain analysis it is necessary to have experts for the problem domain and experts for OO techniques. According to our experience, they do not need to be the same people.

We also learned that the ideal size of an analysis team is three to five members. One or two members eventually forget critical details, more than five members sometimes linger in endless discussions, drifting away to technical details instead of concentrating on the domain problems. We also discovered that it is necessary to cut such discussions from time to time, and to restart them with a smaller team. These results have an organizational impact on the planning of the person power for future design activities.

5.2 Technical Impacts

During the whole design and implementation phase of the components a CASE-Tool with UML-support and sophisticated code generation was used. We think it is not practical to make the design without such a tool. It is also necessary to use the tool during implementation, because an iterative development cycle is only possible if the way from design to implementation and back to design is feasible. That implies that the tool supports good synchronization mechanisms between the source code and the design model and good code generation possibilities. In fact, code generation and synchronization really shortens the implementation time and makes the design more robust, because the design is not hidden in source code and so the developer is prevented from destroying good design during an implementation enthusiasm phase.

We also found out, that a CASE-Tool using UML helps very much in providing documentation and keeping it up to date. The reason for this is, that the UML diagrams are easy to understand and are kept up to date, using the synchronization techniques of our tool.

During design and development we experienced that the more time we spent to design the important parts, the less time was needed for implementation and the more readable and simple was the emerging source code.

However, it is not necessary to first design every detail of the component and then proceed to implementation. On the contrary, it is sometimes necessary to make a detailed design of the important parts and a very rough design of the less important parts, to implement the system prototypically, and to try whether the design works. Then one can go back to design and spec-

ify the missing parts. If it is found that there are conceptual errors in the design, it will cost less time to make corrections according to this approach.

5.3 Result Measurement

The assessment of the degree of reusability is done based on a so-called base line project. The base line project comprises selected parts from an actual project of a representative SEKAS customer from the CAM environment. The customer project included the integration of distinct manufacturing lines, transport, logistics and test into an continuous production process. Customer acceptance was reached in May 1999.

The effect of reuse is assessed by comparing the effort needed for the original development of the base line project (delivered to the customer) and the effort needed for the redefinition of the base line project using the newly constructed components and their framework. The goal is to reduce development effort by at least 20%. An analogous comparison is drawn regarding warranty and maintenance efforts. These should be reduced from 5% of the original development costs to 2.5 %.

The introduced measurement avoids the need for prototypical projects, but takes some time to get a statistical relevant result for maintenance efforts.

The expected commercial impact cannot be consolidated at the moment, but will be assessed at the end of the project. However, the authors are confident in the success of SEPIOR, because the lessons learned so far are more than positive.

6 Conclusions

Although the process improvement experiment SEPIOR is not completed the time this paper is written, some preliminary conclusions can be drawn. The conclusions mainly summarize the technical and human aspects in introducing component based software development.

Despite the initial expense of introducing component based development, the reuse aspect permanently spreads at SEKAS, indicating the rising acceptance of these techniques and methodologies. One of the major impediments is to create continuous stimuli to foster the development of reusable components on the one hand, and to emphasize the deployment of the components on the other hand. The primary risk of management activities inciting these stimuli is that developers overshoot. A natural balance between developing reusable components and specifically customized modules or prototypes has to be found. This premises a skilled developer, who not only shows technical excellence and experience but also possesses common sense. Thus, in the authors opinion the management activities should emphasize on continuous professional technical and social training of the developing staff instead of financial or non-financial incentives.

The major technical issue is the development of a framework for components both during component development and component deployment. This topic is most important for component vendors. The component framework has to include aspects such as communication, configuration, persistency and distribution. Additionally, a framework has to comprise testing and debugging aspects that even work with no introspection possibilities based on the code of the components.

References

Bergner, K.; Rausch, A.; Kuhla, K.: Schnelle Schichten – Transparenter Zugriff auf ODBMS über CORBA. iX No. 11, heise Verlag, 1998.

Bergner, K.; Rausch, A.; Sihling, M.: Using UML for Modeling a Distributed Java Application. Technische Universität München, technical report TUM-I9735, 1997.

Broy M.; Denert E., Renzel K., Schmidt M.: Software Architectures and Design Patterns in Business Applications. Technische Universität München, technical report TUM-I9746, November 1997.

Chappel, D.:: Understanding ActiveX and OLE. Microsoft Press, Redmond 1996.

Chung, P. E.; et al.: DCOM and CORBA Side by Side, Step by Step, Layer by Layer, http://www.bell-labs.com/user/emerald/dcom_corba/Paper.html, Download 1999-10-06

EU (Ed.): Homepage of the European Systems and Software Initiative, http://www.cordis.lu/esprit/src/stessi.htm, 1999.

IPA (ed.): Computer Integrated Manufacturing Framework. Fraunhofer Institut für Produktionstechnik und Automatisierung, http://semi-tf-cim-framework.ipa.fhg.de/, Download 1998-12.

Kruchten, P.: The Rational Unified Process – An introduction. Addison Wesley Ltd., May 1999.

Oesterreich, B.: Objekt-Orientierte Softwareentwicklung mit der Unified Modeling Language. Oldenburg Verlag, 1997.

SEKAS GmbH (ed.): SEKAS GmbH Homepage, http://www.sekas.de/,1999.

SEKAS GmbH (ed.): Homepage SEPIOR project, http://www.sekas.de/english/research.html, 1999.

Szyperski C.: Component Software – Beyond Object-Oriented Programming. Addison Wesley Ltd., 1997.

OMG (ed.): Real Time CORBA Specification,. Object Management Group, 1999.

OMG (ed.): Unified Modeling Language Specification, Version 1.3, Object Management Group, http://www.omg.org/, 1999.

Pree, W.: Komponenten-basierte Softwareentwicklung mit Frameworks. dpunkt Verlag, Heidelberg, 1997.

Sun Microsystems (ed..): JavaBeans: JavaBeans API Specification 1.01. Sun Microsystems, Mountain View 1997.