Integration of Heterogenous Software Architectures - An Experience Report

Volker Gruhn, Ursula Wellen

University of Dortmund, Software Technology, D-44227 Dortmund, Germany
{gruhn, wellen}@helsinki.informatik.uni-dortmund.de

Key words: Software architecture, database integration, distributed objects, components, migration, software landscape

Abstract: In this article we describe our experience with a software migration project in a telecommunication company. We started from a set of heterogeneous software systems (described by rather different types of software architectures) and we defined a migration path towards an integrated software architecture. On this path several intermediate versions of the software architecture were implemented. We discuss the purpose of these intermediate versions and the problems encountered in the migration path.

1. INTRODUCTION

The experience described in this article is from a project in the telecommunication industry. In this project we analyzed the available software and recognized that several software systems were developed or purchased in a rather uncoordinated manner. In contrast to many other companies (e.g., from the insurance or finance industry), this situation was not due to software systems that were maintained and extended for decades, but was due to a rather fast set-up of software systems for the support of core business processes. The company – as a rather new competitor of the former monopolist Deutsche Telekom – had to establish these software systems in a short term. The systems were developed or purchased by departments independently from each other. Each of these departments focused on its particular problem which had to be solved as soon as possible. The

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35563-4_35](https://doi.org/10.1007/978-0-387-35563-4_35)

P. Donohoe (ed.), *Software Architecture*

© IFIP International Federation for Information Processing 1999

coordination between the departments were not as tight as it would have been in a very well-established business context.

As a consequence of this approach, software architectures of individual systems were driven by different paradigms (some are client/server architectures, others are not; some of the client/server systems are based on SQL databases as distribution paradigm, others are based on transaction processing monitors and others are based on Web servers) [Uma97]. Even worse, the functionality of software systems overlap. For example, master data about customers were administered in several systems. These data had to be reconciled in order to ensure that customers are represented in a consistent way. A mid-term goal was to avoid data redundancy. Another drawback of the initial software architecture situation was that—after a short time of operation—it turned out that higher levels of software system integration were needed in order to provide homogeneous support for the business processes (e.g., same style of user interfaces, same client platforms).

Another organizational issue to be considered was that all systems had to be available all the time (24 hours a day, 7 days a week). The risks of systems not being available after a new release varied from system to system, but, generally speaking, unavailable systems could endanger service delivery, customer satisfaction and business plans.

Another important issue was that the business processes to be supported are supposed to change over time. One particularly critical aspect was that of distribution. Even though most parts of the business processes were carried out at the company's headquarter, the future software system infrastructure should allow for a flexible distribution and allocation to new sites.

Starting from this situation, we developed a migration plan that starts from the existing software infrastructure and ends with tightly integrated software systems realized by distributed objects that support the core business processes of the company. On this path different levels of integration were (and still are) implemented.

In the experience described we put emphasis on different ways of describing the software architectures we encountered in the project. These descriptions vary from rather high-level descriptions of functionality in terms of basic building blocks, to dataflow descriptions of individual software systems, to rather technical descriptions of telecommunication infrastructure needed.

In section 2 we describe the project starting point in some more detail. Next, in section 3 we explain the migration path identified and the different software architectures on the migration path. In section 4 we sum up what problems occurred on this migration path. Section 5 puts our project-specific experience into the broader context of general work on software architecture.

Finally, section 6 concludes with pointing out which compromises were key success factors for the migration described.

2. SITUATION OF SOFTWARE ARCHITECTURES AT PROJECT START

When the project started, several departments were developing complex software systems to support their business processes. Because all these projects were carried out under high time pressure, the underlying system parts were developed and integrated without much coordination. This meant, that many software systems were developed independently of each other as stand-alone solutions without interfaces for data exchange. Some other system parts like administration systems for customer master data or provisioning systems did already exist and had to be reused and integrated into the new business context.

2.1 Initial software landscape

Before we analyzed individual software systems in detail, we examined the overall software situation. At this level, individual systems were considered as black boxes and we only looked at the relationships between the individual systems. The relationships identified were of different types:

- data exchange (pushed by the data-sending system or pulled by the data-receiving system; in each case characterized by the frequency of data exchange)
- access to persistent data
- call relationships

In the following we call this view of the overall software architecture “software landscape.” Figure 1 shows the initial landscape and some of the key business objects exchanged between systems. Some details about key business objects are discussed in section 2.2. The software landscape was taken as starting point for the migration. A vision of the future landscape defines the overall migration goal. Thus, any progress can be illustrated in terms of modifications of the software landscape.

Figure 2 describes a more technical view of the software landscape. It shows which software systems are running on which data base management systems and on which operating systems. This view of a layered architecture was very popular in the project, but it turned out to raise many misunderstandings, simply because there was no consensus on the meaning of layers and the meaning of the relationship with neighbouring layers.

Figure 1. Initial software landscape

The applications are depicted as boxes on the upper level. The communication between the several systems is realized with services of the underlying operating system, shown at the second level. The basis for communication within a LAN/WAN is TCP/IP.

Figure 2. Technical infrastructure of software landscape

The obvious deficiencies of the software landscape (redundancies, manual data exchanges) are removed during the migration described in section 3. The result is a software landscape based on distributed objects, software components, and an integrated compository (compare section 4).

2.2 Available software systems

The software landscape contains several software systems that are briefly explained below. We discuss their main purpose, their key business objects and for a few we sketch the release policy planned.

The software systems identified in the software landscape are:

- a) The **SAP/R3** modules used are SAP/FI for Finance and SAP/CO for Controlling. They deal with accounting information that is received from a provisioning system (see below). SAP/R3 impacts the vision of the future software landscape, because the structure of SAP/R3 modules and their relationships to other systems can hardly be modified. But because of the integrated data base of R3 data exchange SAP/R3 modules is no problem. Some further modules of SAP are planned to be integrated later.
- b) The administration tool for customer master data **SAM** supports marketing activities (analysis and control) for sales and marketing departments. It deals with the key business objects “sales partner” and “customer data” and does the assignment between them. These business objects are also used by a provisioning system for sales partners (PRIS) (see figure 1). SAM is developed as client/server application. It uses Oracle as RDBMS, the database access is carried out with SQL-Net. The first release of SAM is dealing with “sales partner” and is running since the middle of 1997. At the end of 1998 the second release will deal with private customers as further business objects. For the next extension of SAM, WAN-wide data access is planned.
- c) **PRIS** is a provisioning system for external sales partners. It creates accounting information for SAP/R3. It requires some data from SAM (business objects “sales partner” and “customer”). PRIS is client/server based like SAM, based on a 3-tier architecture. In a next version PRIS will administrate the business object “turnover”. It receives these information from the billing system BICOS.
- d) One further individual software system is **CURT**, an application for providing reports and statistical analysis results. It analyzes and evaluates, for example, data concerning customers’ call behaviour. The report system is based on 3-tier architecture with Oracle as the underlying database.
- e) **BICOS** is a billing system (derived from a standard billing system). Its key business objects are “overall turnover” and “call detail records.” It has an exchange interface to PRIS. BICOS data are exported as ASCII files. Interfaces to other applications were planned at the beginning of this project.

All existing software systems have their own data repository. Some of the data exchange relationships in figure 1 represent manual exchange of data.

This requires a lot of effort and it is the reason for frequent data inconsistencies.

Besides the software systems discussed, several other systems are needed. These are either developed in-house or they are purchased. In case of purchase, the goal is to focus on standard systems as far as possible.

3. THE SOFTWARE LANDSCAPE MIGRATION PATH

The overall goal of the architecture migration was to start from the existing software architecture and to finally obtain a software architecture that is properly integrated (data integrated, user interface integrated, control integrated) and easily extensible.

After analyzing the initial landscape of the software available it turned out that the proper integration of the existing systems would affect the architecture of the software systems available substantially. In order to ensure that the existing systems remain usable while they are prepared for integration and while they are actually integrated, it was decided to subdivide the migration path into several steps. This was meant to reduce the risks of touching working systems and to focus the effort on a few systems in each step.

Another reason for this stepwise approach was that the owners of the individual software systems had different ideas about how fast their systems had to be evolved. While some wanted to stabilize their software at first (e.g., because a version had been released recently) others wanted to implement new architectural guidelines as soon as possible (e.g., because they were about to plan a new release anyway). These goals and specific ideas had to be reconciled within a common migration path.

In the following we discuss the migration steps identified and the software architectures that were achieved after carrying out these steps.

3.1 Migration step 1: data exchange support for key software systems

In order to avoid the most error-prone inconsistencies as soon as possible, we had to ensure that at least key objects were harmonized. For this purpose, we introduced the data exchange system DEXS. An example for key object harmonization were customer master data which were provided by PRIS, SAM, and SAP independently from each other. While it was not possible to eliminate this multiple responsibility for customer master data, it was at least necessary that any system that gets knowledge of customers informs all other software systems.

This simple form of data integration based on systematic data exchange yields an architecture as sketched in figure 3.

Figure 3. Exchange of business objects between software systems and DEXS

This figure shows that the redundancy of customer master data is not eliminated, but that the exchange system ensures that all customer master data identified in one system is forwarded to all other systems concerned.

business object	master	data flow via interface							
	application	A	B	C	D	E	F	G	H
sales partner	SAM	x	x	x	x				
customer data	SAM	x	x	x	x				
call detail records	BICOS					x	x		
turnover	BICOS		x			x	x		
accounting information	PRIS							x	x

This first kind of integration is neither very ambitious from a software technology point of view nor does it reduce the effort that is spent for

managing data within several software systems, but it ensures that all customer master data is available to all software systems concerned. This first step was quite straight-forward. After identifying the common responsibilities for certain type of data and after identifying which data was produced within which business processes and by which software systems, the most difficult task was to agree on data exchange formats between applications.

At first, the data exchange system implemented only supported the exchange of customer master data between SAM and PRIS. After eight weeks of operation it had to support a further five systems (not discussed in detail in this article). Since interfaces were defined in a bilateral way (i.e., by transforming information from the internal format of system A into the internal format of system B), the effort spent for the exchange system grew exponentially with each new system to be supported. That is why the architecture of DEXS has been changed. DEXS accepts several formats of, for example, customer master data and translates them into an internal format, that can be accessed by all software systems.

Since the internal format cannot be directly processed by the software systems, it has to be translated into each of the application-specific formats. The corresponding process supported by the data exchange component is shown in figure 4.

Figure 4. Data exchange system DEXS with unified exchange format

It is described in terms of FUNSOFT nets [GG95,DG98], which are high level Petri nets. Rectangular symbols represent activities, the annotations denote the software system used. Circles represent information channels, the annotations denote the type of information exchanged.

The data exchange component DEXS gets the data in an individual output format from one of the several software systems. It translates the data into a unified format, filters it into the required input format and forwards the translated data to the concerned software system.

The effort for implementing this first step was rather low. The exchange system as sketched in figure 3 did not require any modification of the software systems at all. The exchange system DEXS as sketched in figure 4 meant to provide some additional interfaces for the software systems, but their core functionality and their core structure remained unchanged. In addition, the filter “unified to individual filter” translates the unified format into the individual formats needed. Whenever a new system has to be integrated the filter has to be extended by a translation mechanism (from the unified format into the new individual format). This modification is local and does not affect the software systems themselves.

3.2 Migration step 2: data integration of key components

While the first migration step did not remove the data redundancy that was introduced by sharing the responsibility for customer master data between SAM, PRIS, SAP and CURT, the second step aimed at avoiding data redundancy at least for key applications. In terms of software architecture this means to identify common components and to delegate the responsibility for the commonly used data to them. Figure 5 shows how a common customer component is extracted from PRIS, CURT, and SAM.

Figure 5. Exchanging common subcomponents

After splitting the software systems into two parts (key functionality and commonly used customer component), the customer component is used by all three systems, thus, ensuring that only one component cares for customer master data. According to the model-view-controller paradigm, the three components can still display customer master data in the way they were used to do. Figure 5 shows how the architecture of applications is modified by extracting a commonly used customer component.

3.3 Migration step 3: control integration of key Components

The next migration step was to improve the control integration between components that were used independently before. This control integration implemented was oriented towards the business processes needed. An example process supported by the systems SAM, CURT, PRIS, SAP systems is the process “identify new customer”. Figure 6 shows an overview about this business process (again represented by a FUNSOFT net). It illustrates that first customer information is gathered by using SAM, that this information is complemented by associating the customer master data with a partner, whose provision depends on the number of customers (PRIS), that customer master data are forwarded to SAP (where debtor accounts are created), returned to SAM (where all information about the new customer is gathered) and that report information are forwarded to CURT (where the new customer is considered in the weekly “new customer report”).

As soon as the business process “identify new customer” is enacted, the supporting systems have to be called in the right order and with the right parameters. In principle, this sounds like a workflow management problem. In the given situation, however, we did it without a workflow management system, but we hard-wired the business processes to be supported. The reason for doing so was that each of the available systems covers quite large chunks of the business process. Usually, workflow management based on such powerful systems leads to gross-grained process models, which are not very expressive. In our hard-wired solution, the control component (called CTRL in figure 7) enforces business processes. It knows about the processes to be supported (parts are directly implemented as control flow dependencies within CTRL, others are expressed as process parameters in the business process model) and calls SAM, CURT, PRIS and SAP accordingly. To move to a workflow based solution following the guidelines of the workflow management coalition demands to move the complete knowledge about a business process model to the business process model representation and to keep the CTRL component completely process-neutral.

Figure 6. Business process model "Identify new customer"

Figure 7. Integration of software systems by a control component

3.4 Migration Step 4: Implementation of Components by Distributed Objects

The German telecommunications market was deregulated recently. The business of the new telecommunication companies is not completely settled yet, but it is subject to changing market conditions and to evolving organizational circumstances. This highly flexible situation demands flexible software. Existing software has to be integrated with new software, business processes may change and have to cover more and more sites. Users to be supported work with software clients running on different platforms (ranging from PCs/laptops to workstations with different operating systems). Thus, flexibility and extensibility are key requirements for the software landscape needed. Support for various and frequently changing client platforms is a concrete requirement derived from the general requirement for flexibility.

The need for flexibility, easy assembly and support of various client platforms was accepted as the general guideline for the fourth and last migration step. Thus, we chose a component-based software architecture as our vision of the migration. In this vision, functionality is provided by components that can easily be assembled, distributed to various locations, and that help to keep the software landscape flexible and extensible. In order to provide availability on various client platforms, we decided to demand that components should be embedded into web browsers. That means, functionality offered by components can easily be accessed from all platforms supported by browsers.

Obviously, the component-based vision cannot mean replacing existing systems by componentized versions immediately because this would mean spending software development effort without gaining any new functionality. This is not acceptable in a situation in which key systems still have to be developed. Instead, the vision of a component-based software landscape serves as long-term goal towards which all software development efforts are directed. Maintenance – if necessary – should try to identify components and to extract them from existing software. New development projects should deliver software components and integration into component-based architectures should be a key requirement for any software that is purchased.

The vision of a component-based software landscape cannot serve as long-term goal if it is not made concrete. New software not only has to be component-based, it also has to match the concrete component model chosen. Thus, the vision of the software landscape has to be underpinned by an implementation vision. For implementation purposes we had to choose between the COM/ActiveX component model [Cha98], the CORBA component model [MZ95] and the Java Beans/Java Enterprise Beans component model [OH98]. It was decided to accept the CORBA component model as our basis. COM/ActiveX was denied because it is a proprietary approach, Java Beans were not accepted, because it was doubted if the Java native interface was open enough to embed the variety of available software systems that were written in various programming languages. CORBA was accepted not only because of the component model, but also because of its services that provide location transparency and convenient component naming facilities.

Another reason for the choice of CORBA was that it was rated as the best paradigm for enabling the smooth move to a component-based and Web-integrated user interface. Using the CORBA object model provided the chance to benefit from IIOP (Internet Inter-ORB Protocol) [OH98]. This protocol helps to overcome the poor interaction facilities between client and server as provided by HTTP. While HTTP is rather restricted with respect to

parameter exchange and transfer of results from the client to the server, IIOP allows full exploitation of the advantages of exchanging objects between client and server. Since IIOP serves as standard for communication based on objects between clients and servers, the component-based software landscape can easily satisfy the requirement to provide the same user interfaces for all platforms supporting the usual web browsers. Figure 8 sketches the vision of such an architecture (called the ObjectWeb). It shows that the basic idea of flexible components managed by CORBA. In order to access these components and the CORBA services needed, the HTTP protocol between web browser and web server is complemented by IIOP.

Figure 8. Vision of the ObjectWeb

The final software landscape that results from the steps mentioned avoids data redundancies, thus ensuring higher levels of consistency. It is based on a component-based architecture of software systems, thus allowing for reuse and flexible exchange of components. It is based on web browsers, thus ensuring a uniform user interface and it uses an object request broker, thereby, implementing location transparency. Even though this software landscape has not been completely reached (some software systems are not componentized yet, some cannot be executed from within a web browser, the CORBA based integration step has still to be implemented), it is an important guideline for all software development activities. Each development project can be measured against the goal to contribute to the final software landscape.

4. PROBLEMS ENCOUNTERED IN THE MIGRATION

The migration led from a software landscape that was determined by loosely integrated software systems to a tightly integrated system that is prepared for a software architecture based on a component model [Lew98] and which is homogeneously accessible from web browsers [Uma97]. At the end of the migration process it was beyond question that the software architecture had been substantially improved with respect to data consistency, data redundancy, clarity of control dependencies, extensibility and flexibility of distribution of components. Nonetheless, during migration we encountered some problems that had to be resolved. Generally speaking, these problems did not occur for the initial version of the software landscape, but they are due to the higher level of data integration, control integration and user interface integration. In the following we discuss some of these problems:

Security: While the protection against unauthorized access to data was necessary for the initial version as well as for any of the following versions, the protection against access during data transfer was an additional problem for all versions, except of the initial one. In particular, the access to software systems from within a web browser was a major obstacle to the new architecture. In order to overcome these problems we implemented a firewall with rather strict authorization checks. In addition, software systems that are accessible from the web are double-checked before released. Unless a system is released for this purpose, it runs on a completely separated local network. Thus, the production environment accessible from the web is not integrated at all with the internal local network. This, obviously, results in some problems (how to transfer a release from one local network to the web-accessible network, which criteria have to be fulfilled before a system is released for the web-accessible network), but this rather rigid approach was the only way to allow access from the web to software systems at all.

Configuration management: While the initial version (consisting of just a set of separated software systems) did not require any coordination across the borders of applications, configuration management required more effort the tighter the systems were integrated. It turned out that the role of a configuration manager had to be established. While this role was not known as an explicit one before (each leader of a development team somehow cared for his configuration issues) the coordination of release plans for various inter-dependent software systems turned out to be a full-time job. We expect that configuration management will become easier the more the vision of a truly component-based architecture is reached, because the management of component configuration requires less coordination than the configuration

management for software architectures determined by the migration steps 2 and 3.

Convergence towards commonly used basis systems: The systems already available at the start of the migration, and the planned systems, used different database management systems and user interface management systems. Sometimes they were based on certain types of middleware (ranging from distributed database systems to message-oriented middleware systems). The vision of the ObjectWeb allowed details of database management systems to be abstracted away as long as they were accessible from an object broker following the CORBA architecture [MZ95] (which is true for all database management systems used). User interface issues were sorted out by demanding browser-compatible interfaces, which was accepted for all systems (even though it was not implemented immediately). Thus, the most important remaining problem was that of the middleware system used. Here, the choice of CORBA was subject to discussions, but it was finally accepted. The implementation efforts for moving systems to a CORBA-based architecture varied from system to system, but it has to be admitted, that efforts were quite substantial in a few cases (e.g., for the PRIS system, which was based on a distributed database system and on the SQLNet protocol from Oracle).

Common development plan / focusing of efforts: As long as software systems were developed independently by departments or autonomous development teams, project planning was rather easy and straightforward. With a higher level of integration additional dependencies on the release requirements of other systems, quality management plans, and management goals became obvious. We are convinced that these dependencies did exist right from the beginning, but that they were hidden. That is why we believe that it was useful to make them explicit, even though it increased the management overhead.

Common architecture model: The initially available software systems not only differed with respect to the platforms used and the ways they were subdivided into pieces, but also with respect to the architecture model chosen for describing software systems. Single teams had agreed on a vocabulary and on certain types of diagrams they used efficiently to communicate internally. They used layered models of software architecture, they used hierarchies of modules, and some used UML class and package diagrams. Some teams provided architecture information that showed how to associate software pieces with operating system processes, and explicitly defined communication protocols between operating system processes. Others just provided high-level package diagrams (which could serve as a starting point for a work breakdown structure, but not as a sufficient architecture description). A tighter integrated software landscape demanded

that teams had to communicate across team borders (in order to define requirements for components they wanted to use, in order to specify interface of services they wanted to use and in order to agree on exchange formats and communication protocols between their systems). In the project we introduced a distinction between an application architecture view (data flow between functional modules, identification of key business objects), a software architecture view (modules, object types, call relationships) and a technical architecture view (distribution issues, telecommunication infrastructure needed). The use of a predefined set of architectural diagrams and the decision to use only these diagrams was cumbersome on the one hand because it meant to give up certain individual kinds of architectural descriptions. On the other hand it helped to overcome misunderstandings and useless arguments concerning the style of architecture descriptions. To sum up, the use of a common vocabulary (represented in terms of a few types of diagrams) and a clear – albeit still informal – description of its semantics turned out to be of substantial benefit with respect to a homogeneous description of the software landscape.

Management of development problems that had already started:

While it was easy to ensure that new projects adhered to the software landscape and to the new architectural guidelines, it was rather difficult to ensure that project teams, which were in the middle of their projects, at least try to consider as much of landscape and guidelines as possible. In fact, we did not succeed in transferring our vision of the ObjectWeb into these kind of projects. This is all the more annoying since we spent some effort on doing so (in terms of walkthroughs and review discussions). Our conclusion from this experience is that a new vision should only be applied to new projects and that it is hardly possible to establish new architectural guidelines after project start.

5. RELATED WORK

The research in the roles and purposes of software architecture in general [SG96, PW92, Gar95], in architecture description languages [MT97, RMR98], and in patterns of software architecture [Sha95] seems to be well ahead of the industrial practice of software architecture design. While the research on software architecture and architecture description languages describes well-founded approaches to the specification of software architectures, software architectures in industrial practice seem to be determined by vague and inconsistent descriptions. Even worse, it seems as if we are rather far away from any kind of industrial de facto standard for describing software architectures. We believe that the benefits of explicit and

precise architecture descriptions can at best be conveyed in the context of smooth software migrations. Here it is possible to immediately reduce costs (of extensions, maintenance and restructuring) and to improve quality of systems delivered [BJN98]. Based on these benefits it is possible to ensure the use of appropriate architecture descriptions.

The problem of describing architectures of distributed systems in industrial practice is even harder. Once again there is some research foundation [MDE95], but little consensus on how to describe distribution issues. Again, it should be demonstrated which benefits are to be gained by using homogeneous architecture descriptions.

6. CONCLUSION

We believe that the use of architecture descriptions should be embedded into a software architecture method, that explains what has to be done when and for what purpose. In the project discussed in this paper we introduced such a – still rather coarse-grained – method that is set up around the term of a software landscape. Even though this remains rather vague for the time being, we think that it allows to start where industrial projects are and to successively move towards more precise and expressive descriptions of software architectures. In particular, this not very ambitious approach helped to obtain architecture descriptions (expressed as software landscapes) that were amenable to discussions with different levels of management. Generally speaking, the approach to establish a common architectural model for the software systems of a telecommunication company demands the careful definition of a migration path that compromises between two goals:

1. fast migration to a state-of-the-art software architecture
2. smooth migration that ensures 100% availability of working systems

The compromise between these two goals must ensure that any effort spent on evolving or replacing software systems must bring the software architecture closer to the vision of the final software landscape.

We believe that our experience is not only typical with respect to the intermediate migration steps taken, but also with respect to the final vision of our migration. The current trends towards component-based software development on the one hand and towards components being executed under the control of a web browser on the other hand naturally lead to the vision of the ObjectWeb.

REFERENCES

- [BJN98] K. Bohrer, V. Johnson, A. Nilsson, B. Rubin (1998). "Business Process Components for Distributed Object Applications" *Communications of the ACM* 41(6), 43-48.
- [Cha96] D. Chappell (1996). *Understanding ActiveX and OLE* Redmond, Washington, US: Microsoft Press.
- [DG98] W. Deiters, V. Gruhn (1998). "Process Management in Practice - Applying the FUNSOFT Net Approach to Large Scale Processes" *Special Issue on Process Technology / Automated Software Engineering* 5, 7-25.
- [Gar95] D. Garlan (1995). "First International Workshop on Architectures of Software Systems: Workshop Summary" *ACM SIGSOFT Software Engineering Notes* 20.
- [GG95] G. Graw, V. Gruhn (1995). "Process Management in-the-Many" *Software Process Technology - Proceedings of the 4th European Software Process Modeling Workshop*, W. Schäfer (editor) Noordwijkerhout, Netherlands: Springer 163-178. appeared as Lecture Notes in Computer Science 913.
- [Lew98] S.M. Lewandowski (1998). "Frameworks for Component-Based Client/Server Computing" *ACM Computing Surveys* 30(1), 3-27.
- [MDE95] J. Magee, N. Dulay, S. Eisenbach, J. Kramer (1995). *Specifying Distributed Software Architectures* European Software Engineering Conference, Barcelona, Spain, 137-153, appeared as LNCS 989.
- [MT97] N. Medvidovic, R.N. Taylor (1997). *A Framework for Classifying Architecture Description Languages* European Software Engineering Conference, Zurich, Switzerland, 60-76, appeared as LNCS 1301.
- [MZ95] T.J. Mowbray, R. Zahavi (1995). *The Essential CORBA: Systems Integration Using Distributed Objects* Toronto, Canada: Wiley.
- [OH98] R. Orfali, D. Harkey (1998). *Client/Server Programming with JAVA and CORBA* Toronto, Canada: Wiley.
- [PW92] D.E. Perry, A.L. Wolf (1992). "Foundations for the Study of Software Architectures" *ACM SIGSOFT Software Engineering Notes* 17(4).
- [RMR98] J.E. Robbins, N. Medvidovic, D.F. Redmiles, D.S. Rosenblum (1998). *Integrating Architecture Description Languages with a Standard Design Method* International Conference on Software Engineering, Kyoto, Japan.
- [SG96] M. Shaw, D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline* Prentice-Hall.
- [Sha95] M. Shaw (1995). *Patterns of Software Architectures*, in: *Pattern Languages of Program Design* edited by J.O. Coplien and D.C. Schmidt, Addison-Wesley.
- [Uma97] A. Umar (1997). *Object-Oriented Client/Server Internet Environments* New Jersey, US: Prentice-Hall PTR.