Published in: Reiner Anderl, Christine Frick, Alfred Katzenbach, Joachim Rix (editors). *Proceedings of the ProSTEP Science Days 2000 "SMART Engineering"* on 13./14. September 2000 at DaimlerChrysler, Stuttgart, Germany. Pages 252-260. ISBN 3-8167-5585-2

The Requirements for a Component-based Architecture

Jochen Rütschlin

DaimlerChrysler AG, Research and Technology, P.O. Box 2360, 89013 Ulm
Email: jochen.ruetschlin@DaimlerChrysler.com

Summary

Today's enterprises are going to concentrate more and more on their core business. They do not want to implement basic functionality again and again. That's why they are using packaged software products like SAP R/3, Dassault Systemes' ENOVIA, and so on. But the introduction of such software suites also brings in elements and information that are already available in the enterprise (and should remain in the legacy systems).

In this paper, we will point out the problem on two typical architectural scenarios and give with this the motivation for a component-based architecture. Besides we propose an architecture and discuss the lacks of the current CORBA-, EJB- and COM-approaches.

Keywords: components, component model, framework, packaged software

1 Introduction

The IT (information technology) infrastructure of many enterprises currently consists of so-called *legacy* systems. These have been developed over the years and include not only internally developed software but purchased software that has been customized as well. In any case, these are poorly connected systems each of them representing a kind of "information island". This results in increased search time to find the relevant, distributed information and the problem of information loss when transferring and converting data between to systems. To avoid these disadvantages, we are working on a continuous data flow between the systems along the process chain.

In the beginning, research concentrated on the integration of data schemas only. The data of the legacy systems was integrated into a global schema by mapping the heterogeneous data models (e.g. hierarchical, relational, object-oriented, etc.) and heterogeneous structures (e.g., normalized tables versus one big table to represent the same information) into a uniform data model and a uniform representation. Standardized data models and schemas, e.g. the AP214 [ISO94] of STEP [ISO97], are very important because of their international acceptance. The common and particularly transparent access of data has been realized either by an FDBS (federated database system) [SL90] or an additional software layer called middleware.

The next step was the awareness that it is not enough to integrate the systems only via their database interfaces. A huge number of applications do not allow direct access to their databases. Examples for these systems are ERP[1] software like SAP R/3 [SAP] or ENOVIA [Enovia]. Instead, a method of an API (application programming interface) has to be used to access the required data or functionality. As a consequence, an approach for integrating systems does not only have to consider data but also functions provided by APIs.

Another trend that can be observed is the tendency towards the usage of packaged off-the-shelf software. Enterprises do not want to waste time in developing standard software/services by themselves. They are going to concentrate on the development of competitive functions (functionality representing the competence of the enterprise) and take care on their core business. To realize an interoperability between the competitive functions and the packaged software, there is also a kind of integration needed.

The best approach for this integration seems to be the use of components. Additional functions (the competitive ones mentioned above) could be plugged into the basic framework. Software pieces could be re-used instead of re-implemented, or they could be replaced by newer ones without major changes and adaptations. In such a scenario, we can also replace the integrating middleware for our legacy systems by the framework itself (respectively by lightweight wrappers which encapsulate the legacy system and provide the unified framework API).

We propose to use the concept of component models to address the two major challenges mentioned above: customization of packaged off-the-shelf software and replacement of legacy systems. These two scenarios will be described in more detail in Section 2. This will lead us then to identify the need for a framework as the architectural basis for a component model.

Afterwards, we sketch out required functionality and elements which should be provided by an appropriate architecture in Section 3. In Section 4 we consider implementation aspects and finally summarize our ideas in Section 5.

2 Application Scenarios

Illustrated by two typical scenarios, the need for component-like applications will be shown in this section. The first is the enhancement of functionality of so-called packaged software. The second scenario focuses on the modularization of software packages.

[1] ERP = enterprise resource planning.

2.1 Extending & Customizing Packaged Software

In most big enterprises, IT strategy means to concentrate on core business—the competitive functions representing the knowledge and competence of an enterprise. People do not want to waste time developing again and again basic functionality that is already implemented in many applications. Instead, they use packaged software like SAP/R3 or PeopleSoft [PS] proving this basic functionality. Although the vendor of these products offer extensions for special industrial branches they cannot cover all fields of operation. Hence, the enterprises are forced to enhance the functionality of the packaged software in some way. Imagine to build up a production line. On the one hand, business objects like bills of material, assembly plans, and inventory information are used. Those items can be managed by the packaged software. On the other hand, there is enterprise specific information that cannot be handled by the commercial software. Exactly this information increases the efficiency of the production line by, for instance, enabling specialized just-in-time processes and an optimized sequence of work items.

The scope of extending packaged software ranges from adding some simple if-clause-branchings up to plugging in big components with the power of application-sized software. The first case can still be overcome with built-in programming languages like ABAP (advanced business application programming) in the case of SAP/R3 [Man98], whereas on the other end of our functionality scope a component architecture with an open API is needed.

2.2 Modularization of Software Products

The architecture in Figure 1 shows a typical Web application infrastructure. The user interacts by means of a browser with a portal providing a personalized front-end environment. The portal software includes an AS (application server) ①, a Web server realizing the GUI (graphical user interface) ②, an LDAP[2] directory for storing the user profiles ③, and the portal functionality (personalization and channeling) itself ④. In order to run the AS, another Web server ⑤, another LDAP directory ⑥ as well as a database ⑦ (for storing the configuration values and authorization information) are needed. Moreover, there might be independent Web servers ⑧ offering additional information to the portal or legacy databases ⑨ connected to the client via the AS.

[2] = lightweight directory access protocol. "This is a software protocol for enabling anyone to locate organizations, individuals, and other resources such as files and devices in a network, whether on the Internet or on a corporate intranet. LDAP is a "lightweight" (smaller amount of code) version of DAP (directory access protocol), which is part of X.500, a standard for directory services in a network.." (Source: http://www.whatis.com/ldap.htm)

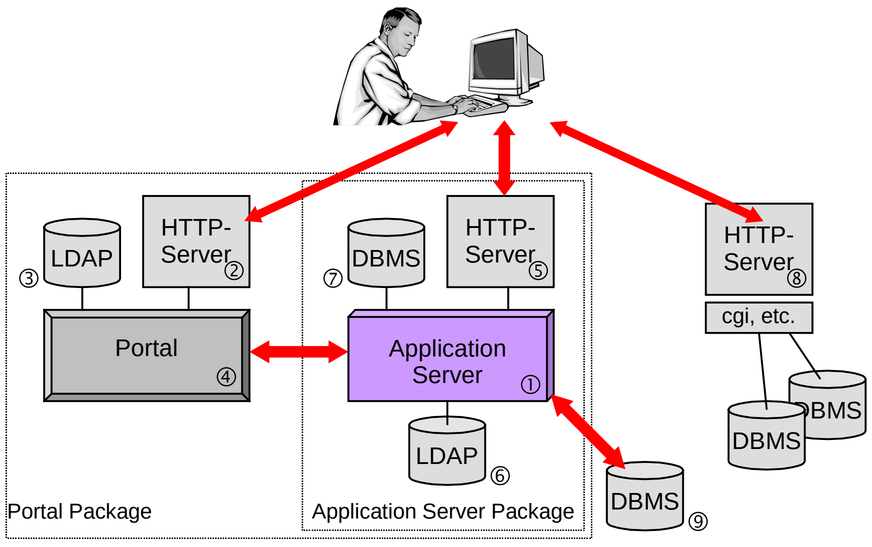

Figure 1: A typical Web application infrastructure.

Obviously, buying and running single software packages like an AS or a portal brings in software products and information that is already available in the enterprise's IT infrastructure. If, for instance, we have a look at the security aspects in this scenario, we can realize that the required information is stored redundantly in different places. Each application brings in its individual security model and security database, although in most cases authentication and authorization information is already stored somewhere in the system. Web resources on simple Web servers are protected by "htaccess" files[3] and a file based user/group database; security information of the portal is stored in the LDAP directory; the AS holds its authentication information in another LDAP directory and the authorization information for the Web resources is kept in a relational database. Finally, information stored in a database is protected by authorization information which is also stored within the database—the authorization information is mostly based on the user's operating system identity. In such a heterogeneous scenario, it is rather difficult to build a single sign-on mechanism, not to mention the unification of the authorization procedure.

Summarizing the discussion of the two application scenarios, the main disadvantage of today's IT solutions is the missing extensibility of packaged software to cover the whole functionality needed by the enterprise. In addition, we have to deal

[3] These are files in the directory structure of your Web resources holding the authorization information for specific HTML files.

with high redundancy of software products and stored information. In the next section, we will introduce our ideas for a component framework architecture overcoming these shortcomings.

3 Architecture Requirements

In the following, we will develop the functionality needed to build a component framework which is shown in Figure 2.

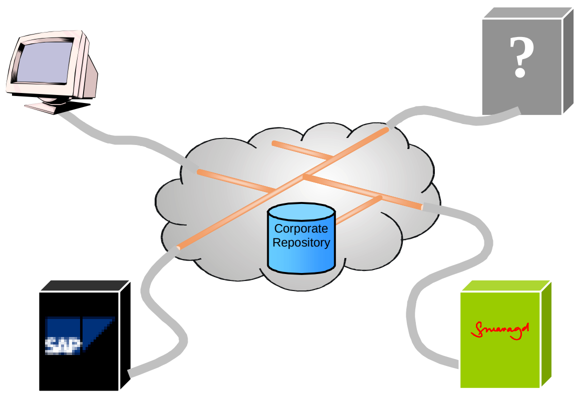

Figure 2: Component Architecture Framework.

The core of our proposed component architecture is a communication infrastructure to which the components could be plugged in easily. During the plugging-in process the component registers itself to the framework. A corporate repository (CD) storing this registration data also contains additional information about all systems (software and hardware) of the enterprise. The idea is, that components looking for a certain functionality can get and analyze the semantic information of all components registered at the CD. Together with the corresponding syntactic interface description, the found method can directly be called. For example, an engineer is browsing the bill of material in his EDM (engineering data management) system and realizes that a certain part is not released. His EDM system certainly offers the releasing functionality in the GUI, but does not have an implementation on its own. So his application is contacting the CD and looking for an appropriate component offering the desired functionality on the relevant part of the global data schema. The found component is “started” with identification of the part which has to be released. As result it returns a status message (e.g. “part sucessfully released”) to the EDM system.

When speaking about a component, we mean simple clients, applications like Smaragd[4], or large software packages like SAP/R3. In Figure 3 we can see such a component from its functional view.

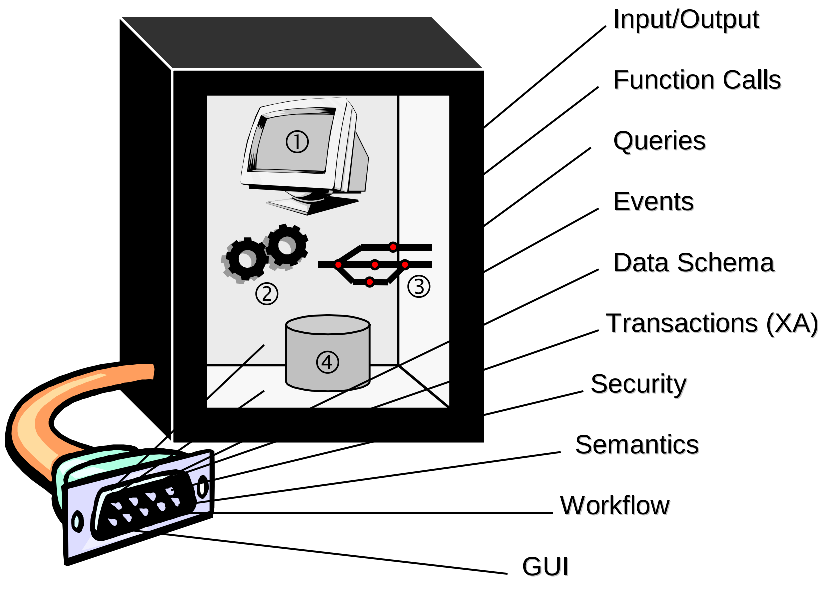

Figure 3: A component with its interfaces.

The component by itself acts as an independent application. With this it has a GUI ①, implements some business logic ②, possibly has an internal workflow ③, and encloses some data ④.

Basically, a component processes information. It needs interfaces for the data exchange (*input/output*) and for the call of its functionality (*call interface*). Moreover the input/output functionality—which essentially is a routing of parameters only—could be extended to an interface for submitting search-inquiries, based on SQL[5] for example (*query interface*).

By supporting asynchronous communication, we give the component the possibility to react on unforeseen events or to set off events by itself (*event interface*). The component processes a part of the (assumed) global schema and thus encapsulates its data. To keep the data consistent, we are offering a transaction interface with the 2-phase-commit protocol (*XA interface*).

[4] Smaragd is a product from SDRC to maintain product structure linked together with the corresponding CAD (computer aided design) files.

[5] = structured query language [ANSI99]. Query language to access data stored in a relational database system.

The *security interface* offers functions e.g. to exchange authentication information between components in order to realize a single sign-on or to cipher/decipher input/output data. Furthermore, the interface should be able to process authorization information of the underlying global schema.

To support the lookup for a particular functionality, the component must offer a description of its *semantics*. With this description only, the component can be guaranteed exchangeable with other components that have exactly the same functionality.

Besides, the component maybe contains a sub-*workflow*, so the entry- and exit-points of the workflow must be accessible through the component's interface.

Finally, a component should offer a presentation for the *GUI*. Components/applications could then be used from other applications. Examples are a TWAIN driver from the Windows operating system, whose dialog window can be accessed from different graphic tools, or the Adobe Acrobat reader, which displays a selected PDF file from the internet directly via the web browser.

4 Design and Implementation Aspects

In order to build such a component framework we consider already existing component models for fulfilling our requirements. Most important are the "three big commercial" ones, namely Microsoft's COM+ (component object model, [Pla99]), Sun's EJBs (enterprise JavaBeans, [EJB]), and CORBA (common object request broker architecture, [OMG]) of the Object Management Group (OMG). These architectures focus nearly on the same characteristics: reusability, distribution, and interoperability. But beneath these advantages, they only support a more or less technical realization of communication. Imagine a (possibly distributed) pool with components of the same functionality. The client software can choose an instance of this pool depending on the current distribution and load of the system. In such a scenario, the mentioned models show some weaknesses, because they only describe the signature of functions and components. Since a semantic description is missing, no automatic choice and communication can occur with other components. Because of the same reasons there is no possibility to verify that a replaced component offers exactly the same functionality than the original one. Comparable to the AP214, a commonly defined semantics could help to exactly specify the functionality of a component.

Concerning the granularity of the commercial component models, they do not match our requirements either. When speaking about a component, we consider a stand-alone application like a product data management system or a CAD application. EJBs, in contrast, aim at the encapsulation of smaller units like business objects.

Aside from the problems described so far, each component model has its peculiarity: COM+ is limited to the Windows platform, EJBs are dependent on the Java programming language, and CORBA cannot deal efficiently with data-intensive operations [Sel00].

Considering all the aspects mentioned so far, a more comprehensive component model is necessary. Our intention is not to develop yet another component model but to realize a solution based on existing technologies.

5 Conclusions and Outlook

In this paper, we have worked out the requirements for a component-based architecture. After motivating the need for such an architecture by the problematic of packaged software and the shortage of modularization of software products, we have proposed first steps to achieve a component-based architecture. The paper closes with some remarks regarding the three commercial component models (CORBA, EJB, COM+) and their deficiency for such an architecture.

Our next steps include the definition of a global security model, since it represents an important element of a component model as described in this paper. Moreover, we are looking for an appropriate communication infrastructure for the framework. In addition, we will continue to have a look at the commercial models considering their evolution w.r.t. our requirements.

6 References

[EJB] Sun Microsystems (2000). Enterprise JavaBeans. http://java.sun.com/products/ejb/

[Enovia] Enovia Corporation (2000). ENOVIAVPM. http://www.enovia.com/solutions/ html/edesvpmoverview.htm

[ISO94] ISO 10303 (1994). Industrial Automation Systems and Integration – Product Data Representation and Exchange – Part 1: “Overview and Fundamental Principles”, International Standard.

[ISO97] ISO CD 10303 (1997). Industrial Automation Systems and Integration – Product Data Representation and Exchange – Part 214: “Core Data for Automative Mechanical Design Processes”, Committee Draft.

[ANSI99] American National Standards Institute, Inc. (1999). Database Languages – SQL – Part 2. Foundation (SQL/Foundation). ANSI/ISO/IEC9075-2-1999. In: American National Standard for Information Technology, approved Dec-1999.

[Man98] Hans Dieter Mann (1998). ABAP/4 Sprachreferenz. Feldkirchen: Franzis. ISBN 3772356842.

[OMG] OMG (1999). *CORBA Components – Joint Revised Submission*. OMG TC document 99-02-05. http://www.omg.org/cgi-bin/doc?orbos/99-02-05

[Pla99] David S. Platt (June 1999). *Understanding COM+*. Redmond, Washington: Microsoft Press. ISBN 0-73560-666-8.

[PS] PeopleSoft, Inc. http://www.peoplesoft.com/

[SAP] SAP AG (2000). SAP R/3.
http://www.sap.com/solutions/r3/

[Sel00] Jürgen Sellentin (2000). *Datenversorgung komponentenbasierter Informationssysteme*. Informationstechnologien für die Praxis. Berlin · Heidelberg · New York [a.o.]: Springer-Verlag. ISBN 3-540-67728-3.

[SL90] Amit P. Sheth, James A. Larson (1990). Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22 (3): 183-236.