Comparison of JavaBeans and ActiveX – A Case Study

D. Birngruber, W. Kurschl, J. Sametinger
Johannes Kepler Universität Linz, A-4040 Linz, Austria
Email: birngruber@ssw.uni-linz.ac.at, Werner.Kurschl@acm.org, sametinger@acm.org

ABSTRACT
We have investigated the component models JavaBeans and COM/ActiveX, their support for component-based software development as well as their commonalties and differences. The main objective has been to find out the component models' usefulness in building real-world applications and to compare the underlying component models based on a concrete sample application. In this paper we briefly introduce the component models JavaBeans and COM/ActiveX, and present a simple application with two different implementations based on these models. Finally, we make a comparison of the component models.

Keywords
JavaBeans, COM/ActiveX, resource planer, comparison, case study

1 INTRODUCTION

Component-based software engineering means development of software systems by composing reusable components. Components themselves shall only be developed when not being available on the market. This may lead to reduced time to market, improved software quality, and less maintenance effort. The reusability of components has to be increased in order to realize this vision. We need global component models in order to allow the creation of global component markets. Additionally, methods, techniques, tools and process models have to be developed and adapted for the component-based creation of software systems. But before component-based software technologies can successfully contribute to improved software engineering, several questions have yet to be answered. For example: What is a component? How do we develop components? Where can we buy components? How can we compose components? How can we describe and retrieve components? These questions are not yet answered, neither from a scientific nor from a practical point of view.

For the case study presented in this paper we have investigated the component models JavaBeans by *Sun Microsystems* and COM/ActiveX by *Microsoft*, their support for component-based software development as well as their commonalties and differences. The main objective has been to find out the component models' usefulness in building real-world applications. The starting point of the case study was a JavaBeans implementation of a resource planner [1], which had been reimplemented in COM/ActiveX in order to allow a comparison of the two underlying component models based on a concrete sample application.

In Section 2 the component models JavaBeans and COM/ActiveX will briefly be introduced. In Section 3 we present the resource planner with its two different implementations. The comparison of the component models will be presented in Section 4. Conclusions will be drawn in Section 5.

2 COMPONENT MODELS

On the one hand, there are two competing component models, JavaBeans and COM/ActiveX which represent de-facto-standards and are based on proprietary developments. On the other hand, the *Object Management Group* (OMG) has developed various standards of an *Object Management Architecture* (OMA), with the *Common Object Request Broker Architecture* (CORBA) being the one known best. CORBA is intended for the communication of distributed objects that can be implemented in various programming languages. Therefore, problems are being addressed that are of importance in the context of components and component models. We will consider only JavaBeans and COM/ActiveX in this paper.

There is still no wide consensus on the definition of the term *component*. We regard the definition provided in [10] as a good starting point. In addition, we consider aspects mentioned in [6] as being important in this context. Thus, by component we mean a unit of composition with contractually specified interfaces and explicit context dependencies with appropriate documentation and a defined reuse status. We also still lack a consensus on what a *component model* really is, even though there are components available already. All we know is that such a model has to define how components present themselves to their environment and how they can be used and coupled. Subsequently, we give a short introduction to JavaBeans and COM/ActiveX, the two component models used in our case study.

COM/ActiveX
The *component object model* (*COM*) by Microsoft [2] consists of components (*coclasses*), interfaces and a mechanism to find and use components as well as various services. Components represent implementations of various interfaces. A COM interface defines a set of functions. COM objects are instantiations of components. A unique *class identifier* (CLS-ID) is assigned to every component and a unique *interface identifier* (IID) to ever interface. Interfaces are being described by the *Microsoft interface description language* (MIDL) and are independent of any programming language. COM objects can be programmed in any programming language as long as the generated object code adheres to the binary standard of COM. COM is supported by languages and tools like Visual C++, Visual J++, VisualBasic, Borland Delphi and PowerBuilder.

COM objects are provided by a COM server which may run in the same address space, being realized as a *dynamic link library* (DLL), or in a separate address space as an executable (EXE). A COM server may also run on the same or on a different machine as the client. *Distributed COM* (DCOM) is used when communication is necessary across process and/or machine boundaries.

In order to get a component with a specific class identifier at run-time, the procedure *CoCreateInstance* is called. (All procedure names of the COM library start with *Co* which stands for COM.) CoCreateInstance requires a class identifier and an interface identifier as parameters. It creates a new instantiation of the specified class and returns an interface of the type required. The client always receives a pointer to the requested interface, but never a pointer to the component itself. The *registry* is used in order to find a COM object with a certain class identifier. The registry determines which server is capable of creating a specific object. When necessary, the specific server is being started. For in-process servers, i.e., servers running in the same address space than the client, the appropriate dynamic link library is loaded. Out-of-process servers, i.e., servers running in separate address spaces, will simply be started as applications. The started COM server contains a separate class factory object for each class in order to create instances of that class.

Every COM object implements the interface *IUnknown*, that defines the methods *QueryInterface*, *AddRef* and *Release*. QueryInterface allows the client to determine at run-time, whether a component provides a certain interface, and, if available, to request that interface. The other two methods are used to administer memory resources by using *reference counting*. AddRef increments a counter for each requested interface. Whenever a client does not need an interface any longer, it can release it. Thus, COM is able to stop a server and release resources that are no longer needed as soon as the counter decrements to zero.

COM has been extended by concepts and mechanisms for distributed computing. ActiveX controls serve as server components. They combine automation services and compound documentation services, thus, creating flexible relationships between containers and servers. ActiveX controls are embedded servers that can be activated within a component. They use the automation service in order to provide its methods and properties to the container. The container also uses the automation service in order to receive event notifications from the control. This allows, among other things, to integrate ActiveX controls into Internet applications, i.e., they can be embedded into HTML pages.

JavaBeans
JavaBeans has been introduced in 1997 and represents the component model based on Java [8]. A *bean* is a reusable software component that can be visualized and interactively manipulated in builder tools. Builders can range from simple layouting tools to extensive, visual, component-based programming environments. JavaBeans supports the following concepts:

- *properties*: Properties can be set interactively by the user and be used for the communication among beans. A distinction is made among *standard properties*, *indexed properties* (property arrays), *bound properties* (properties signaling their modification) and *constrained properties* (modifications of these properties can be vetoed by other components).

- *events*: Events are used as a communication vehicle among beans.

JavaBeans is based on Java's object model, which allows the usage of classes and objects in a well-known manner. Interfaces to beans are defined by a set of methods that define properties and events of beans. Interfaces of beans may be deducted from interfaces of Java classes, which can be done by means of the reflection and introspection mechanism. However, interfaces of beans have to adhere to certain conventions, called patterns in JavaBeans terminology. For example, properties are defined by a pair of setter- and getter-methods, a property *name* is defined by two methods *setName* and *getName*, presuming they have the right parameters. Rather than adhering to naming conventions a bean can also be defined by an additional BeanInfo class providing meta information about the bean.

A bean can be represented by a single simple Java class. A bean can also comprise many Java classes as well as resources like images and videos. Complex beans are usually stored in and distributed by means of compressed archives, i.e., JAR files. Java's serialization mechanism is used to make bean instantiations persistent. At run-time, Java's garbage collection is used to get rid of any bean instantiations that are not in use anymore.

JavaBeans had not been designed for distributed systems. However, beans may communicate across process boundaries, e.g., over the Internet, by means of Java's *remote method invocation* (RMI) mechanism. RMI provides a transparent communication vehicle among objects and, thus, beans, that run on different *virtual machines* (JVM). Services needed for distributed JavaBeans architectures have been considered in *Enterprise JavaBeans* (EJB) [9], which has not been available when conducting the case study described in this paper. Further details about JavaBeans and closely related technologies can be found in [8].

3 CASE STUDY

A resource planning tool can be used for various purposes. We have developed such a tool especially for the coordination of breaks and their supervision in schools. However, since we have kept extensibility in mind, the tool could easily be adapted to other domains. This section briefly describes the requirements on the tool and a sample scenario. Subsequently the two implementations in JavaBeans and COM/ActiveX will be presented.

The implementations were done in two separate projects. The JavaBeans project was done before the COM/ActiveX project. Design and architecture had been defined in the JavaBeans project and were available to the team of the COM/ActiveX project, see [1]. The overall architecture was designed independently of a specific component technology. The suitability of a component model to develop a given architecture was part of the investigation.

Requirements

There are three different types of users: the person responsible for the resource plan, the person creating the resource plan, and the person involved in the plan (i.e., the person supervising a break). The system has to administer information about supervisors, i.e., name, dates and times not to be assigned to that person, kind of employment, etc. We distinguish between regular employees and auxiliary employees, because this influences the maximum number of breaks a person may supervise. Based on this information, a resource plan has to be created that assigns two to three people to each break and picks one of these persons, which has to be a regular employee, to be responsible for the supervision.

Each resource plan has a life cycle that is described by three states: *development*, *verification*, and *released*. Released plans indicate the date when they will become effective. They remain valid until another released plan becomes effective. Only the person responsible for the resource plan may change its status.

Resource plans have to be accessible by various people in order to allow their flexible creation and modification. Supervisors may read plans that have the states *verification* or *released*. Also, they may modify information about their person, e.g., times when they must not be assigned to supervise. All data is kept on a server and may be accessed from clients with modern user interfaces.

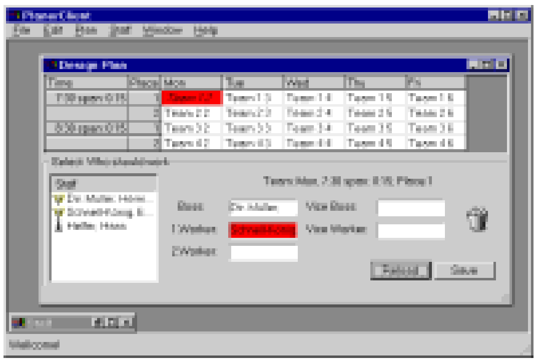
Sample Scenario

Mrs. Miller is responsible for creating a resource plan of a certain school. First she has to input all information about potential supervisors. She delegates this task to her secretary, Mrs. Gandy, who thus becomes the creator of the resource plan. Mrs. Gandy inputs all information known to her; e.g., she excludes dates when she knows people will be out of town. Mr. Graham, a teacher at the school and thus a potential supervisor, adds additional excluding dates not yet known to Mrs. Gandy. Mrs. Gandy then creates a new resource plan that is under *development* and assigns teams of two to three people to all the breaks. Then Mrs. Miller inspects the plan, makes minor corrections and modifications, and modifies its state to *verification*, i.e., she gives public notice that the resource plan is to be checked by teachers. Any wishes for modification have to be reported to Mrs. Gandy, who collects all the wishes and hands them over to Mrs. Miller. Mrs. Miller makes the final decision on any assignments, checks whether there are any conflicts, and finally releases the plan. Conflicts occur when a person is assigned to supervise a break even though he is not available at that time. Conflicts have to be shown by the resource planer.

Implementation

In this section we will describe the architecture of the resource planer. We have chosen an architecture of four layers, i.e., the data layer, the access layer, the application layer, and the user interfaces layer. They will be described subsequently.

- *data layer*
  The data layer describes the application specific data model with necessary conditions to guarantee integrity and consistency. The data layer is independent of the application in order to make the components reusable for similar applications. Additionally, the data layer does not have any references to any of the other layers.
- *access layer*
  The access layer serves as a communication vehicle between the data layer and the application layer. It provides various views and ways of modification to the data layer. The client/server architecture as well as mechanisms for manipulation and synchronization are modeled in this layer. The access layer should have references to the data layer only. However, we had to make exceptions in a few cases.



Time	Place	Mon	Tue	Wed	Thu	Fri
7:30 upto 8:15	1	Team 2.2	Team 1.3	Team 1.4	Team 1.5	Team 1.6
	2	Team 2.2	Team 2.3	Team 2.4	Team 2.5	Team 2.6
9:30 upto 9:15	1	Team 3.2	Team 3.3	Team 3.4	Team 3.5	Team 3.6
	2	Team 4.2	Team 4.3	Team 4.4	Team 4.5	Team 4.6

Team Mon, 7:30 upto 8:15, Place 1

Boss: Dr. Muller
1.Worker: Schnell-König

Figure 1: Client of Resource Planer

- *application layer*
  The application layer implements all transactions for the realization of system behavior as well as the management of the user interface.
- *user interface layer*
  The user interface layer contains all graphical user interface elements and their layout, see Figure 1.

It goes without saying that the chosen architecture does have an influence on the degree of reusability of available components. It is useful to check and perhaps modify the architecture according to the availability of reusable components.

Implementation in JavaBeans

The *user interface layer* of the client is based on predefined bean components which provide only rudimentary functionality. Therefore, additional application independent user interface components had to be developed. For the application itself various components were developed as Java classes in a first step. Then they had been extended to JavaBeans components without much effort.

For the *application layer* most of the needed functionality had to be implemented by hand in order to compose components of the user interface layer, the access layer and the data layer. Components of the user interface had to be enriched with code in order to provide the right behavior of the system based on user input. Similar to the user interface layer, the application layer had been developed in a prototypical manner.

The architecture of the *access layer* had been finished in a single step. The reason for this efficient implementation process lies in the fact that we were able to map the architecture to the mechanism for *remote method invocations* (RMI). RMI is perfectly suited for the object model of Java. Therefore, we only had to implement an additional service for the distribution of events across process boundaries (remote event service).

The *data layer* consists of domain-specific components that are used by components of the other layers. The architecture of this layer had been redesigned several times. A redesign comprises the activities of design, component search, component adaptation, partial implementation for a prototypical evaluation, and refinement of the design. The reason for several redesign cycles was based on the goal of having the entire system available as a reusable component as well as components especially from the data layer and the application layer. We believe that the number of redesigns stems from the fact, that it is a difficult task to develop components designed for later reuse.

A description of the components goes beyond the scope of this paper. Further details of the implementation can be found in [1].

Implementation in COM/ActiveX
COM/ActiveX components may be implemented in several programming languages. For evaluation purposes we decided to implement the server in C++ and the client in Java.

The implementation of the client *user interface layer* is based both on graphical user interface components of the *Windows Foundation Classes* available under Visual J++ and on ActiveX controls available from various vendors.

Similar to the JavaBeans implementation, the *application layer* contains mostly components for the composition of components from the user interface layer and the data layer.

Components of the user interface layer and the application layer are part of the client. Components of the data layer on the server side are being accessed by means of the *access layer*. For this layer there were no developments necessary, because the functionality of DCOM was sufficient for our purposes.

The data layer contains domain-specific COM objects that provide interfaces to the other layers.

4 COMPARISON

We will compare the COM/ActiveX and the JavaBeans component platforms based on the experiences in the case study. Additionally, we will discuss problems we identified with these component platforms.

Learning curve
The time needed to become familiar with the COM/ActiveX component platform was significantly higher compared to the time needed for the JavaBeans platform. The project members needed four to six months until they were able to build and use COM/ActiveX components implemented in C++. This seems to be rather typical for such projects according to [4]. The time needed to become familiar with JavaBeans was about 2 weeks. The difference in the learning curve has several reasons:

- The project members were used to design and code object-oriented programs. In the COM world the term object does exist, but with a different meaning compared to the traditional object-oriented model. The JavaBeans model is based on the traditional object-oriented model [5].
- Many books deal with COM/ActiveX programming but with different quality. This increased the effort for comparison and selection of the right literature.
- The project members had no experience in MS-Windows programming. Therefore, they had to learn the *Microsoft Interface Definition Language* (MIDL), different frameworks and libraries like MFC, WFC and ATL. Extensive frameworks and libraries make it hard for a COM/ActiveX novice to master the complexity. However, they allow to build complex COM components faster and more easily than building them from scratch as soon as the main parts of the frameworks and libraries are well understood.
- The project members had experience in Java programming.

Résumé: We regard JavaBeans to be superior to COM/ActiveX with regard to the learning curve.

Development environment
The acquisition of a proper development environment or builder tool for developing JavaBeans was a time consuming process at the beginning of the implementation phase. The market for JavaBeans development environments was still immature and proper tools with the desired functionality were missing. The market has changed and now various tools

from different vendors offer JavaBeans support. The tools offer different functionality and quality which increases the effort for comparison and selection. A recommendation for a specific tool is problematic, because new versions of the tools with changed functionality and quality are appearing quickly on the market. The chosen products for the JavaBeans development were IBM VisualAge and Symantec VisualCafe.

Different vendors offer development environments for COM/ActiveX development. We decided to pick tools from the palette offered by Microsoft because this seemed to best guarantee conformance with the COM platform. The tools had many similarities which eased the change from one programming language and tool to another. The chosen products were Visual J++, Visual C++ and VisualBasic.

Résumé: We regard COM/ActiveX to be superior to JavaBeans with regard to the development environments.

Implementation

The implementation phase of both the JavaBeans and the COM versions consisted of several design, redesign and implementation cycles. When building the COM version the cycle included the search for ActiveX components. According to [11] there are much more ActiveX controls available on the market than JavaBeans components. In the meantime the number of available JavaBeans may have increased but is still far from the number of available ActiveX controls.

The development teams encountered several implementation relevant problems when building the COM and the JavaBean version. Problems encountered when developing COM components include:

- *Reference counting*: COM uses reference counting rather than automatic garbage collection. A COM server can be shut down when the clients have released all their references to COM interfaces provided by the server. The server depends on a careful reference handling by the client which can be more error prone when the number of used interfaces increases. Different programming languages offer different utilities for this problem like *smart interface pointers* in C++ or the automatic reference counting and releasing of the Microsoft JVM.
- *MIDL*: The Microsoft Interface Definition Language offers different data types. However, they cannot be used in every *COM aware* programming language. Therefore, using the *wrong* data types for an interface definition can exclude a programming language from implementing the interface. An overview and a least common denominator among C++, VisualBasic and Java is given in [2].
- *DCOM*: Client/server applications based on DCOM are not able to run on a regular Windows 95 operating system. However, Microsoft offers a special DCOM extension for Windows 95 which has to be installed separately. We recommend to run DCOM servers on Windows NT because not all servers work with the Windows 95 DCOM extension.

Problems encountered when developing the JavaBeans components include:

- *Icons*: Many builder tools use icons to visually represent JavaBeans. This requires that a JavaBean is delivered with its BeanInfo class. According to the JavaBeans specification [7] a JavaBean can but has neither to provide a BeanInfo class nor an icon. It turned out that building a BeanInfo class without tool support is a tedious task, especially during the implementation phase when the interfaces of the beans still get modified.
- *Names*: Typically builder tools allow the visual manipulation of JavaBeans. Some builder tools use names to identify different components of the same type. These names are used for building a visual representation of the wiring code. The JavaBeans specification does not mention such names. The base class for all visual components has a *name* property but not all builder tools, e.g., Symantec Cafe, actually use it. Manipulating JavaBeans with different builder tools can cause problems because the visual representation of the manipulation may not be displayed.
- *Bean container*: The original beans specifications did not reflect on containment. Hence, JavaBeans often form a logical containment hierarchy. The Bean extension *Glasgow* offers such a functionality but for our case study Glasgow was still in the specification process. Currently Glasgow is

available but not all development environments have it integrated already.

- *Code generation*: Builder tools generate code based on visual manipulations. The way how and where code is generated has not been defined. Different tools use different strategies to generate the wiring and customization code. As a consequence, it is nearly impossible to change the development environment during the development phase. However, a finished JavaBean may be used with different builder tools.

Résumé: ActiveX is superior to JavaBeans with regard to the implementation phase. ActiveX components can be developed with different programming languages including Java. The COM/ActiveX platform offers more functionality and there are more components on the market than for JavaBeans.

Documentation

The Java source code contains special tags in the comment of classes and methods in order to allow the generation of HTML documentation. The tags are extracted with a special tool (javadoc) which is delivered both in form of an executable binary and in source code form with the JDK. The documentation of the class libraries within the JDK was produced with javadoc. However, special features of a bean like properties and events cannot be documented with this tool. The source code of the tool can be modified to support project specific documentation but this requires extra effort. The tool itself turned out to be memory intensive especially for large documentation. COM/ActiveX does not offer a documentation tool like javadoc. But COM/ActiveX components written in Java can be documented the same way as JavaBeans.

The search for components is a tedious task because there are no appropriate tools for browsing and retrieving. This is unacceptable because the components offered on the Internet cannot be searched in a common way. The documentation of both component models should have foreseen information about functional and non functional requirements like resource usage, capacity, latency, performance and reliability.

Résumé: The documentation quality for both component models needs major improvements. There is no appropriate tool support for browsing and retrieval.

Security

The use of components within HTML documents over the Internet requires security mechanisms. Attacks could harm the client machine as well as the user. Possible aggressions include:

- *Integrity and consistency of data and processes*
  through the deletion of data on the hard disk or through the abnormal termination of processes.
- *Availability of resources*
  through the creation of processes with high scheduling priority or high memory usage.
- *Privacy of user or company*
  through publishing and sending private data to unauthorized persons.
- *Annoyance of users*
  through playing noise or through displaying unwanted images.

Such attacks can be performed in several ways. Especially dynamically loaded components over the Internet can be what is called *Trojan horses*. Trojan horses perform an official task – like chess playing with the user – and an unofficial, vulnerable task – like reading the hard disk of the user and sending private data.

The security model of Java offers a mechanism to prevent external or untrusted code to perform system critical operations – like accessing the user's hard disk [3]. In contrast, ActiveX components which are dynamically loaded over the Internet are not restricted in their functionality.

Résumé: JavaBeans is superior to COM/ActiveX with regard to security on the Internet.

Distribution

RMI is one possible communication mechanism for distributed Java programs. It is an *application programming interface* included in the JDK. Remote objects are accessed via their interfaces which allows a homogenous integration of local and remote method invocations. RMI does not offer services like an event distribution service. We implemented such an event distribution service with low effort. RMI and CORBA can be integrated. Therefore, CORBA services can also be used. The distribution mechanism of DCOM has offered enough functionality in order to realize our distributed client/server architecture.

Résumé: Both component models offer an acceptable communication mechanism for distributed systems.

Language Independence
An advantage of COM/ActiveX is its programming language independence. Components can be developed in the preferred programming language which allows the developer to choose her preferred programming language and paradigm. The interfaces of a COM component are described with MIDL. The usable data types are restricted to the available programming language – MIDL mapping. Normally JavaBeans are written in Java and the code is compiled to Java byte code. JavaBeans can, in principal, be written with other languages as long as the compiler compiles the programmed code into Java byte code. As far as we know a byte code compiler for other languages is not commercially available yet.

Résumé: COM/ActiveX is superior to JavaBeans with regard to language independence.

Platform Independence
One important aspect of the popularity of Java is its operating system independence. This independence is still not reached at all points and a final test of software systems on at least Unix and Windows platforms is required. COM/ActiveX has been built for one platform, i.e., Microsoft Windows. But COM has been ported to UNIX operating systems like Sun Solaris, DEC Alpha (Software AG, EntireX). The developed and reused COM/ActiveX components were not tested on the ported COM platforms.

Résumé: JavaBeans is superior to COM/ActiveX with regard to platform independence.

Summary
An overview of the comparison is given in Table 1 which shows a well-balanced picture for both component models. The strengths of JavaBeans are the short learning curve, the security model (referred to as the *sandbox model*) and its platform independence. The strengths of COM/ActiveX are the provided development environments, the big number of available components and the independence of programming languages.

Category	**COM/ ActiveX**	**Java-Beans**
Learning Curve	-	+
Development environment	+	-
Implementation	+	o
Documentation	-	-
Security	-	+
Distribution	+	+
Language independence	+	-
Platform independence	o	+

Table 1: Summary of the Comparison

Documentation should be improved for both models by providing appropriate browsing tools, retrieval tools and documentation techniques. Both component models allow to build distributed software systems with an acceptable amount of design and programming effort.

Both component models allow the creation of software components with different granularity, i.e., from small grained visual components to complex components. Large distributed software systems often require different services like transaction, message queuing, load balancing, directory services, etc. These services are available for COM/ActiveX. For the Java world many of these services are now available, too, but some are still in the specification process and not available yet.

The comparison presented in this paper can help to make a decision for one component model but other relevant project dependent circumstances have to be considered as well. Such circumstances can cover the target platforms, integration with existing systems, and performance. The strengths of JavaBeans include its simplicity and platform independence. The number of available components and the language independence are considered to be the main advantages of COM/ActiveX. Documentation of available components rarely covers non-functional requirements like reliability, fault tolerance, correctness, security and performance. The question is: How suitable are poorly documented components off the shelf? Some projects do not allow the reuse of external components, like projects for nuclear power plants or for security systems and the number of available components may not influence the decision process.

This comparison does not cover all scientific relevant categories, e.g., a discussion about the object models. The basis of the comparison was a qualitative approach and not a quantitative measurement.

5 CONCLUSION

There are many similarities between COM/ActiveX and JavaBeans. A clear winner in the race of component-based programming cannot be determined. In case someone had to choose between the two component models, several factors should be taken into account. JavaBeans has advantages in that it is a simple model with a low learning curve. But it lacks a wide selection of reusable components. The availability of components is a definite plus for COM/ActiveX.

For components that may be considered for potential reuse, there is almost never information about the quality of the component, i.e., it is hard to say whether the component is reliable, correct, efficient, secure, etc. So even for high quality components it may be a better solution to invest in an own development. This situation may make the number of currently available components for a component model less significant for various projects.

The number of available services like *transaction server, message queuing* or *name service* may be crucial for the development of commercial, distributed systems. COM/ActiveX offers a wide range of such services. For JavaBeans various specifications of such services are available and implementations will hopefully follow soon.

For the development of components for distributed systems, especially when used in the Internet, the Java security model can be used, for example, to restrict access to local resources. However, this model cannot avoid unwanted use of resources like starting many threads. COM objects enjoy full privileges, being able to access the file system without any restrictions or to shut down the computer. Therefore, we would refrain from using COM/ ActiveX in the Internet domain.

Most personal computers run under Microsoft Windows. COM/ActiveX performs significantly better on this platform than the interpreted byte code of JavaBeans. This fact makes the choice of COM/ActiveX obvious in many situations. However, in heterogeneous environments, e.g., when using servers on both the UNIX and the Windows platform, the use of JavaBeans becomes interesting again, even though COM/ActiveX has been ported to UNIX environments.

It is impossible to be clearly in favor of either COM/ActiveX or JavaBeans. It is necessary to precisely be aware of a project's goal as well as to closely follow new developments in the arena of component models. This paper provides a decision basis for practitioners as well as for scientists by showing what we believe are the major strengths and encountered problems during a development project.

6 REFERENCES

[1] Dietrich Birngruber, Werner Kurschl, Gustav Pomberger, *Komponentenbasierte Softwareentwicklung – Ergebnisse einer Fallstudie*, Bericht, Institut für Wirtschaftsinformatik, Johannes Kepler Universität Linz, 1997.

[2] Don Box, *Essential COM*, Addison-Wesley, 1998.

[3] James Gosling, Henry McGilton. *The Java Programming Language*, OOPSLA '95 Tutorial Notes, Austin, TX, 1995.

[4] David J. Kruglinski, *Insight Visual C++, V5.0*, Microsoft Press, 1997.

[5] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[6] Johannes Sametinger, Software Engineering with Reusable Components, Springer-Verlag, 1997.

[7] Sun Microsystems, *JavaBeans API Specification*, 1997, see http://java.sun.com/beans/docs/spec.html

[8] Sun Microsystems, *JavaBeans Documentation*, 1999, see http://java.sun.com/beans/docs/index.html

[9] Sun Microsystems, Enterprise *JavaBeans Technology*, 1999, see http://java.sun.com/products/ejb/

[10] Clemens Szyperski, *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

[11] Clemens Szyperski, *Emerging component software technologies – a strategic comparison*, Software Concepts & Tools, Vol. 19, No.1, 1998.