Electronic Notes in Theoretical Computer Science 65 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 12 pages

Describing Hierarchical Compositions of Java Beans with the Beanome Language

Humberto CERVANTES [1] Jean Marie FAVRE [2]

Laboratoire LSR-IMAG
220, Rue de la Chimie
Domaine Universitaire, BP53
F38041, Grenoble Cedex 9
France

Frédéric DUCLOS [3]

Dassault Systèmes
9,quai Marcel Dassault
92150 Suresnes, France

Abstract

To better understand the conceptual issues concerning hierarchical composition, that is the possibility of an assembly of components to become itself a component, we are developing a simple component model that includes a language called Beanome where this kind of compositions can be described. In this paper we will describe the Beanome language, along with details about its execution and the results we have obtained with it.

1 Introduction

We can appreciate several trends concerning CBSE. From an industrial point of view, several component technologies have appeared from which the most popular ones are Microsoft's COM [3], Sun's Enterprise Java Beans (EJB) [11] and Corba Component Model (CCM) [10]. These technologies target mainly specific domains, like for example the construction of e-business applications. On another side, the research branch focuses on numerous approaches from which we can mention in particular the Architecture Description Languages, that try to give a high level description of applications [6], and a number of

[1] Email:Humberto.Cervantes@imag.fr
[2] Email: Jean-Marie.Favre@imag.fr
[3] Email: frederic_duclos@ds-fr.com

©2002 Published by Elsevier Science B. V.

other fields each targeting more or less specific problems. There have been many trials to describe the concepts behind CBSE, either from a technical point of view [1] or from a more conceptual one [9,8].

It is interesting to point out that industrial component models contain important concepts, but unfortunately they are often obscured by abundant implementation details which makes them hard to understand, to master and to modify. These limitations have led us to develop a small component model, built on top of Java Beans [12], which intends to address in a simple way what we think are the main points of component models which are: the construction of components, their assembly, the setting of extra functional properties and their deployment.

In this paper we will limit ourselves to describe only the first two aspects, namely the construction of components and their assembly through Beanome, a simple composition language. The features of Beanome are intentionally few, the reason for this is that we want to be able to explore concepts gradually before adding more complexity. In particular we will focus to the problem of building hierarchical assemblies of components, by describing them in a simple language. We also have to point out that we want to assemble components whose size can be as small as that of a simple button, and as big as a whole application; the former constraint is not in the goals of the more complex industrial component models. Finally we want to be able to test modifications in a rapid way, and to address this specific point we have chosen our language to be interpreted during execution.

The outline of the paper is as follows : Section 2 introduces hierarchical composition and places it in a component development process. Section 3 describes with more detail the Beanome language, section 4 discusses some aspects concerning connectors. Section 5 explains the runtime environment that interprets the language. Section 6 describes current results and some measures. Finally section 7 gives a conclusion and gives possible paths to follow in the future.

2 Hierarchical composition

A process to allow the development of applications from components is outlined in the specification of EJB and CCM. This process can be roughly divided into three different phases, as shown in Figure 1.

Phase 1 concerns the construction of the components. In this phase the developer of the component joins a description of a component to artifacts that implement it. Settable properties are left to allow further adjustments to the component without the need to access its internal details.

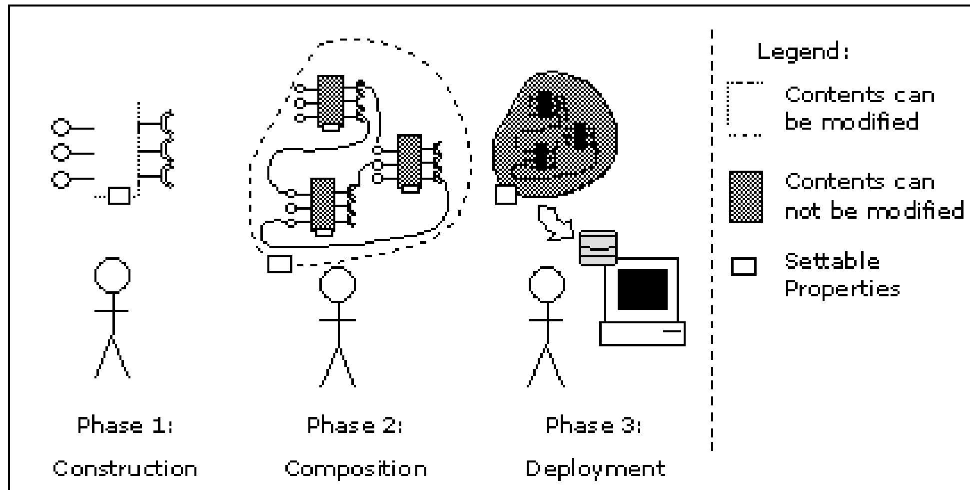

Fig. 1. Different phases of Component-Based application developpment

Phase 2 concerns the creation of the assembly of components. In this phase the assembler can't access the contents of the components anymore since for him the component has become a "black box". He can however fine tune them thanks to their properties. Since the assembly will be further used as a unit, he also leaves some properties that can be set in the next phase.

Phase 3 concerns the deployment of the assembly. The deployer activities consist in adjusting the properties of the assembly and deploying it correctly.

What distinguishes each phase is that the product it receives from the previous one can be seen as a 'black box' unit whose contents cannot be seen or modified (especially during phase 2), only some properties can be set to do some fine tuning (including setting some extra functional properties). This might be seen as an important aspect for these component models, since it allows each phase to be realized in distinct and maybe unrelated locations, and by people with different roles and skills.

However, one might wonder if the result of phase 2 can itself reenter another cycle of assembly, this time by playing the role of a "black box" component and in this way become part of a higher level assembly, in what we call a hierarchical composition. Even though this would seem to be a rather natural part of the cycle, it is not something we currently find in industrial component models [1].

One reason for this can be these models target the building of applications from a limited set of relatively big sized components, a situation where hierarchical groupings might not be indispensable. The reasons for the big size of these components might be more related to technology: the costs of life cycle management and communication between them make it impractical to build

applications composed of many small sized components who need to communicate a lot. Other difficulties that can further complicate the task of creating component hierarchies are more related to versioning and deployment aspects.

In our particular case however we want to be able to create hierarchical compositions of components that can start at a small size. We want in particular to be able to describe assemblies of Java Beans. To do this, we have to cover the two first phases described previously and since we want to create hierarchical assemblies, we must also be able to describe an assembly as if it was itself another component. It is interesting to note that this approach of building software is "bottom-up", since every assembly is a component of a higher degree of granularity.

In the next section we describe with more detail the language we have created and that allows us to explore the creation of hierarchical compositions and other concepts of component models.

3 The Beanome language

The Beanome language is composed of two main concepts: "Component Interfaces" and "Component Implementations". The first one is an external description of the components, with features that are similar to those found in the abstract component model of CCM. The second part allows to describe the implementation of the Component Interface. It can be either a binding to a Java Bean or to a static assembly of instances of other Component Implementations.

3.1 Component Interface

A Component Interface represents the external view of a Component. It is equivalent to the abstract model proposed in the specification of the Corba Component Model and described in IDL. Figure 2 shows the kinds of ports that can exist in a Component; *Facets* are named occurrences of provided interfaces while *Receptacles* are required ones. We can also see that this model allows the declaration of event related ports, *Event Sources* for events that are produced and *Event Sinks* for events that are consumed. Finally, *Properties* can also be declared.

Figure 3. shows the meta-model of the Component Interface. One must note that ports have a multiplicity attribute. This attribute, which can be either *simple* or *multiple*, allows to restrict the number of connections to a particular port (by default it is supposed to be *multiple*). The *type* attribute corresponds in our case to a Java type.

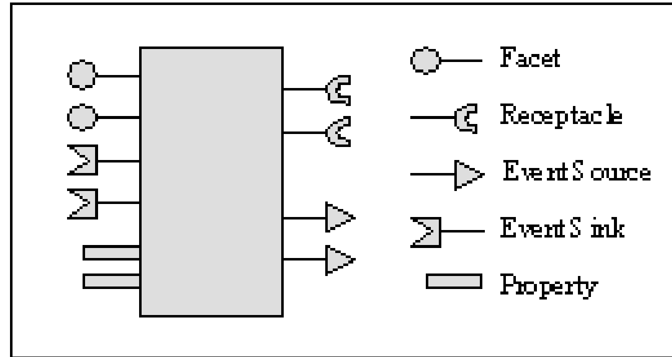

Fig. 2. The ports of a component

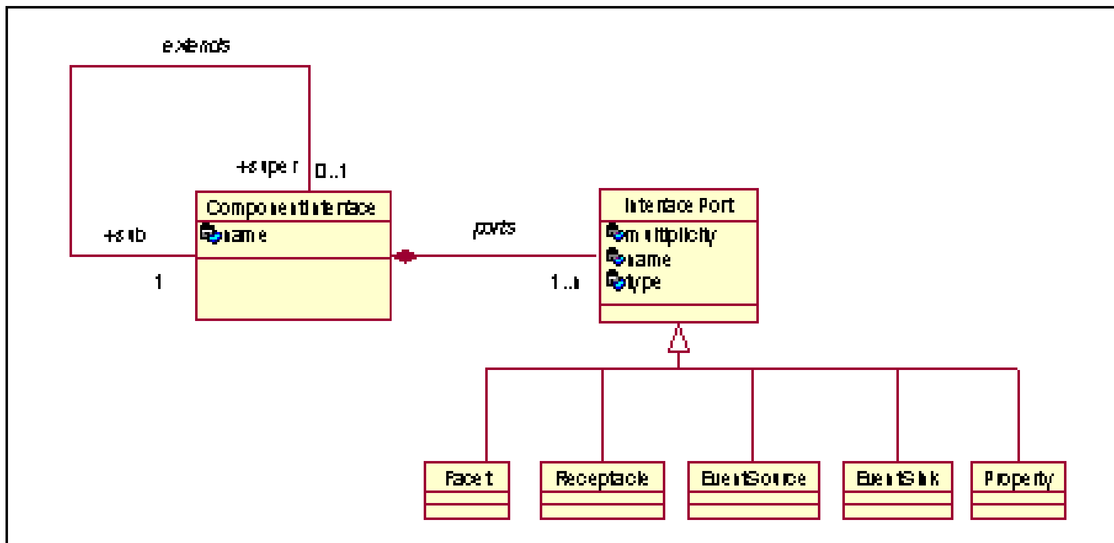

Fig. 3. Meta-Model of Component Instance

Another important aspect is that a Component Interface can *extend* another one. It must be noted that ports that are declared in a parent Component Interface and that are redeclared in a child one are simply overridden. Figure 4 shows an example of Component Interface, in this example, the component can be seen as a JComponent, an interface that allows it to be included in a Swing GUI form. This component also uses an Icon, and it can send ActionEvents. It has a String property that defines its label.

3.2 Component Implementation

Once a Component Interface has been declared, it must be implemented by one or more Component Implementations that are either native Java Beans or assemblies of connected instances of components.

The first case is the simplest one, namely when the Component Interface is implemented directly by a conventional Java Bean (we will refer to this as

```
Component Interface ButtonComponent
{
   Facets:
      JComponent jcomponent;
   Receptacles:
      simple Icon myIcon;
   EventSources:
      ActionEvent myEvent;
   Properties:
      simple String label;
}
```

Fig. 4. Example of Component Interface

being a *Native Implementation*). In this situation however, we have to follow certain rules to allow a correct match between the ports described in the Component Interface and the actual code of the Bean. These restrictions allow to 'componentize' any Java Bean, without the need of accessing or modifying its source code, this is useful for example to be able to include Swing GUI Components directly.

These rules are the following :

- All the provided facets (interfaces) must be inherited or implemented by the Bean.
- To every Receptacle must correspond a public *setter* operation.
- There can only be one Event Source or Event Sink of a particular type.
- Properties must have public setter/getter operations.

Some of these rules, like the limitation on the number of event sources/sinks of a same type arise from the characteristics of the event model of JavaBeans. Figure 5 shows a Native Implementation of the Button Component described previously, it is actually a bridge to the JButton class of the Swing library.

```
NativeImplementation MyButton of ButtonComponent in javax.swing.JButton
{
   label="ok";
}
```

Fig. 5. Example of Native Implementation

It is interesting to notice that creating implementations out of native Java Beans can be placed during the first phase of the process described in section 2.

3.3 Building components from assemblies

An assembly is created from a set of Component Implementation instances connected between each other through their ports as follows:

- Facets connect to Receptacles. One Receptacle may be linked to multiple

Facets.

- Event Sinks connect to Event Sources. An Event Source can send events to multiple Sinks.
- Properties are filled with a value.

To be able to connect two ports, the only restriction is that their types match. This means that a Receptacle whose type is a can hold any Facet whose type is either a or any of a's subtypes.

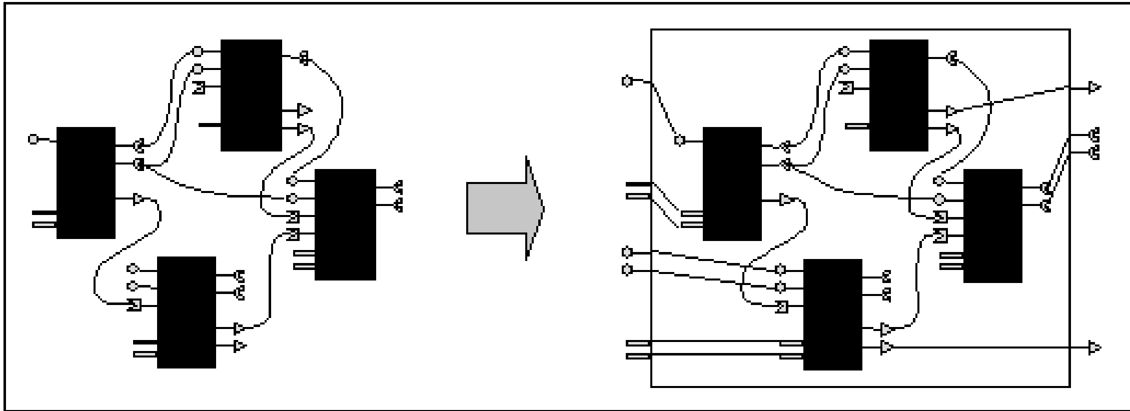

Fig. 6. An assembly becomes a Component Implementation

The creation of an assembly can be located during the second phase of the process described in section 2. In our case however, once the assembly has been created, it can itself become the implementation of a Component Interface. To do this we must link ports belonging to the elements of the assembly to equivalent ports in the Component Interface we want to implement (Figure 6), and in this way, internal ports are *externalized.* Ports are said to be equivalent if their kind (that is *Facet, Receptacle, etc.*), their type and their multiplicity are equal. We must note that all the ports of the Component Interface must be connected to internal ports.

Figure 7 shows an example of assembly. In this example we can see two instances of myButton, the Native Implementation described previously. The events produced by this button are sent to theLed, a component that displays a small LED that can switch between two states. The three components are added in a ContainerPanel. Finally, the jcomponent port of the ContainerPanel is externalized to another port of the same name to allow the component to be itself added in another Swing Component. In this way, LEDwithButtons can now be used in other assemblies.

By repeating the process of building each time higher level assemblies, we eventually end up with a component that is in itself an application. The hierarchical composition that implements it is a static vision of how components are connected initially.

```
Component Implementation LEDwithButtons of LEDComponent
{
   jcomponent <- apanel.jcomponent;

   Instance apanel of ContainerPanel
   {
      layoutType="BoxLayout";
      children=theLed.jcomponent,onButton.jcomponent,offButton.jcomponent;
   }

   Instance onButton of myButton
   {
      label="On";
      myEvent->theLed.turnOn;
   }

   Instance offButton of myButton
   {
      label="Off";
      myEvent->theLed.turnOff;
   }

   Instance theLed of GreenAndRedLED
   {
   }
}
```

Fig. 7. Example of Component Implementation

4 About connectors

In the Beanome language there is not a concept of complex connectors, that is connectors that allow to overcome mismatches in the description of components and that act as “glue” [6]. There is however a real need for particular constructions allowing for example to create a bridge between an Event Source and an Interface. This is described as an “adaptor” in the Java Beans specification [12].

A syntactical construction that hides the concept of a simple adaptor has been proposed in [2], this construction allows the calling of a method when an event is fired. This solution is useful because it solves the most typical problem that can be encountered in a Java Bean, that is when the Bean doesn’t implement a Listener interface for a particular event, but this is still limited since it doesn’t allow for example to send another event as a consequence of a successful call to the method.

Instead of adding to the language a limited set of syntactic constructions to express only the most common situations, we have decided to employ components to create connectors. The advantage is that this approach can allow the definition of a series of different reusable connectors for specific purposes, like for example event demultiplexing, another type of adaptor that is necessary. Finally, since an assembly can be seen as a component, once we have ‘surrounded’ a component with a series of adaptors, they can become together a new reusable component.

5 Runtime environment

The execution of a Beanome component is realized with the help of a runtime environment that manages the lifecycle of the components. It interprets the language and consequently locates, loads, instantiates and connects the physical objects that form an application, while also keeping a representation of the elements that form the hierarchical composition. Once the composition has been created and after the native objects that form it are connected between each other, the runtime environment used to interpret the language can eventually disappear.

This can be an interesting feature because of the fact that native objects connected in this way become independent of the runtime environment. This may be seen as an implementation of the *Third Party Binding* pattern described in [5], the runtime being the third party that binds the objects.

Another possibility however is to allow some of the native pieces of code to be aware of the runtime environment by giving them access to it. Since the runtime keeps the information that represents the hierarchical composition, it is possible to create in this way a reflection mechanism at the level of the components for the native objects that implement the components.

Therefore, these native objects become "runtime aware", and they can benefit of a number of services provided by the runtime environment. An example of a service is *navigation*, that allows a native object to ask the runtime environment for a particular port of a component, this is the equivalent to the *QueryInterface* mechanism of the COM model, or to the *getFacet* of CCM. Another advantage of this "runtime awareness" is that the initial graph that represents the internal structure of a component can evolve as a result of the interaction between the native objects and the runtime environment. This allows the runtime environment to keep not only a static description of an assembly corresponding to its initial state, but to have a representation that is synchronized with the evolution of the application during execution.

6 Experience with Beanome

To put the Beanome language to the test, we have created an software environment to allow the assembly of Beanome components that itself is built by using this approach (Figure 8). This practical method has brought us good benefits since it provides immediate feedback about the needs and issues of the language.

The software environment we have created allows to show a visual representation of Components. In the image above, we can see on the left a graph representing a Component Implementation that is formed by several

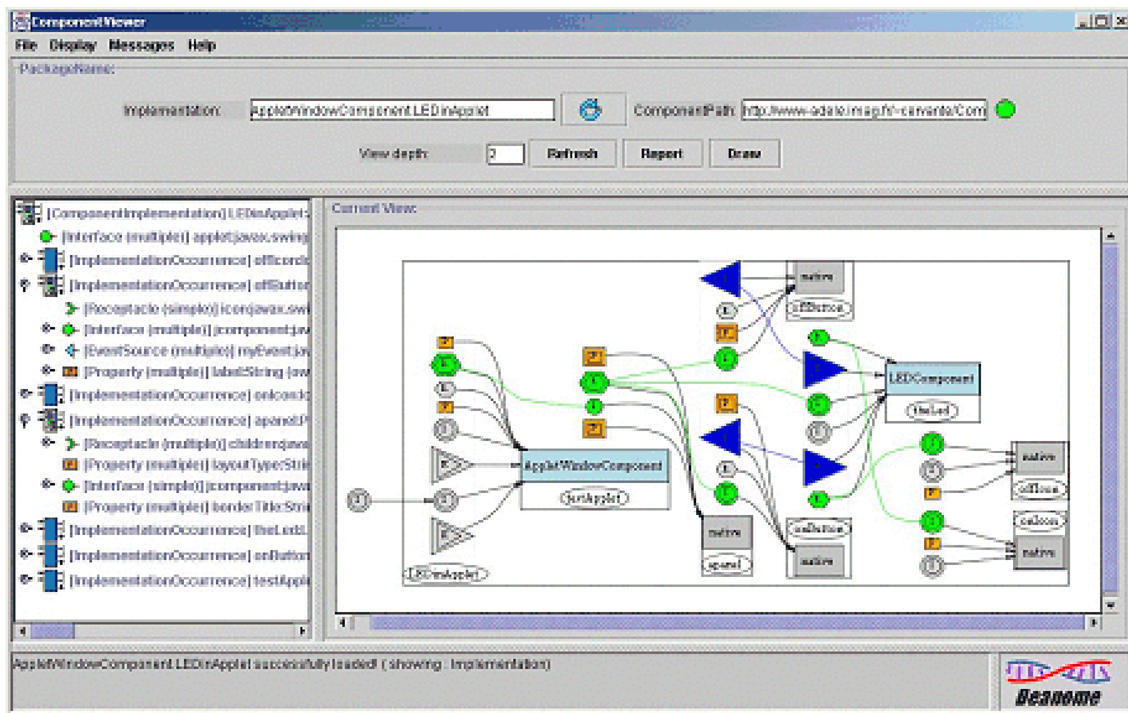

Fig. 8. A software environment built using the Beanome language

instances of components connected between each other. Triangles pointing towards their component correspond to event sinks, their opposites correspond to Event Sources. Circles represent Facets, hexagons represent Receptacles and rectangles to Properties. We can also appreciate that this assembly externalizes one Facet.

It is interesting to look at some numbers that can give us an idea of how this simple environment is built:

- Component Interfaces: 30
- Component Implementations: 36
- Component Implementation instances: 135
- Native Objects: 159

The amount if instances of Component Implementation may be surprising if it is compared with the number of Component Implementations that exist, however we must recall that our model allows the assembly of small sized components, which are often instantiated many times. There is also a difference between the number of Component Implementation instances and that of Native Objects. This can be explained by the fact that Component Implementations that are assemblies are not directly binded to any native object, this also allows us to see that there are 24 instances of Component Compositions.

Our application is still relatively simple, and these numbers may vary with time. It is however interesting to be able to quantify Components and their instances.

7 Conclusions and Future Work

In this paper we have introduced the Beanome language, a simple language that allows to describe hierarchical assemblies of components whose size can start a something as simple as a Button. We have also explained the way it is interpreted to build a graph of native objects connected with each other while keeping a representation of the hierarchy of components thus allowing for a reflection mechanism at the component level. Finally we described a software environment we have created with the language and we have given some measures that give us an idea about the quantities of components and of their instances that can exist in an application.

Currently we are still exploring several aspects linked to the hierarchical characteristic. In particular we are studying how packaging should be realized. Another aspect we are studying is the possibility of creating abstract implementations, similar to the "Plans" described in [2] or to the "Composition Patterns" in [4], that could later be completed either by introducing a concept of *refinement* to the component Implementations or interactively during execution. However, this might be complicated for several reasons, starting with the fact that we want the details of an assembly to remain hidden once it reenters the assembly phase, but refining is difficult without having access to the assembly details [7]. Another reason is that this might create unwanted dependencies between components. Work is in progress to try to give an answer to these questions.

Finally, other members of our research team are studying other important aspects of component models, in particular there is work in progress to find ways to include non functional aspects to the components, and also on solving deployment issues. We hope that all these ideas will ultimately converge in a complete yet simple component model.

References

[1] Bachmann, Bass, Buhman, Comella-Dorda, Long, Robert, Seacord, and Wallnau. Volume II : Technical Concepts Of Component Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, The Software Engineering Institute (SEI), May 2000.

[2] Dietrich Birngruber and Markus Hof. Using Plans for Specifying Preconfigured Bean Sets. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS)*, 2000.

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

[4] Francisco Curbera, Sanjiva Weerawarana, and Mathew J. Duftler. On Component Composition Languages. In *Proceedings of the* 5th *International Workshop on Component Oriented Programming (WCOP)*, 2000.

[5] Philip Eskelin. Component Interaction Patterns. In *Proceedings of the Conference on the Pattern Languages of Programs (PLoP '99)*, August 1999.

[6] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural Description of Component-Based Systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, New York, NY, 2000.

[7] John Lamping. Typing the Specialization Interface. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 201–214, October 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number10.

[8] Theo Dirk Meijler and Oscar Nierstrasz. Beyond Objects: Components. In M. P. Papazoglou and G. Schlageter, editors, *Cooperative Information Systems: Current Trends and Directions*, pages 49–78. Academic Press, November 1997.

[9] Betrand Meyer. The Significance of Components. http://www.sdmagazine.com/beyond/, November 1999.

[10] Object Management Group. *OMG, CORBA Component: Joint Revised Submission*, August 1999.

[11] Sun. *Entreprise Java Beans.* http://java.sun.com/products/ejb/.

[12] Sun Microsystems. *Java Beans Specification*, version 1.01 edition, 1997. http://java.sun.com/products/javabeans/.