# Design-time Assembly of Runtime Containment Components

David H. Lorenz Predrag Petkovic
College of Computer Science
Northeastern University
Boston, MA 02115
{lorenz,predrag}@ccs.neu.edu

**Abstract**

*In this paper we lay out the design time and space of component assembly in a visual builder, and propose concrete ways of also handling context components seamlessly. We present an environment that allows Java developers to test and evaluate JavaBeans components implementing the extensible runtime containment and services protocol. We enhanced the* `BeanBox` *from the Bean Development Kit (BDK) with new visual manipulation capabilities for nested beans, so that it is possible to drag and drop, serialize and de-serialize, group and ungroup, nested beans. By means of these visual manipulations, child beans can be visually put inside context beans.*

## 1. Introduction

Glasgow [8], Sun's code-name for the new release of the JavaBeans [10] component model specification, lists the *extensible runtime containment and services protocol* [1] as one of three new capabilities added to the JavaBeans component model (the other two are the drag and drop subsystem for the Java foundation classes [2] and the JavaBeans activation framework [3]). This capability is manifested in Java 2 (JDK 1.2) in a new `java.beans.beancontext` package. Components implementing the `BeanContextChild` interface may interrogate their enclosing `BeanContext` environment to dynamically discover and employ services.

The precise details of implementation, however, remain complicated. Much of the complexity of putting the newly introduced context package into use rises from the possibility of the component to simultaneously belong to more than one of the three different logical hierarchies: a collection hierarchy, a visual hierarchy, and the new context hierarchy. Moreover, each hierarchy is governed by a different, sometimes contradictory, set of constraints and rules. For example, a `Container` cannot implement a `BeanContext` interface directly [1], but may be associated with one by implementing the `BeanContextProxy` interface, all because of a method name collision.

The `BeanContext` specification and API alone are insufficient for utilizing the new concepts quickly: the documentation is inconclusive, complicated to understand, and open to interpretation. The new `java.beans.beancontext` package is an API, not a set of guidelines, and the overall learning curve remains steep [12]. Unfortunately, the JavaBeans tradition of providing a testing environment along with the API was not followed.

Unlocking the JavaBeans API and its underlying philosophy would have been hard without the availability of the Bean Development Kit (BDK) [9], also called `BeanBox` [7] after the name of its main class. The `BeanBox` permits running JavaBeans demos, and testing your own beans. It demonstrates the functionality expected from a visual builder. It's loyal to the specification. One can determine, for example, whether a reluctant bean is really not working properly, or whether the commercial environment is at fault, by also testing the bean in the `BeanBox`. It is not a complete development tool, but rather a testing and demo tool.

What is missing is a similar lightweight demo, assembly, design, and testing environment, like the BDK, but one that also supports context-nested beans. Such an environment would serve three purposes; it would provide: (1) a vehicle for demonstrating good examples of runtime discovering of services; (2) a third-party client for testing user-made context components; and (3) a prototype (but not necessarily the Holy Grail) for visual bean-builder environments. As far as we know, none of the commercial JavaBeans builders fully support the new runtime containment and services protocol [1].

In this paper, we identify the lifetimes and visual dimensions of components at assembly and design time, we lay out the assembly-design space, and we explain what is needed in order to provide a meaningful assembly and design environment for manipulating and nesting context components.

We present an environment, which we name *ContextBox* [13], for demonstrating, testing, or prototyping a visual context bean-builder. *ContextBox* is an extension of JavaSoft's `BeanBox`. It adds to the `BeanBox` several new features that are useful for the developers and programmers working with `BeanContext` objects. *ContextBox* was created as a byproduct of our need to quickly test our own beans that implement the `BeanContext` interface, and now we wish to share it with everyone.

In Section 2, we explain the fundamental terms and concepts, and in Section 3, we then describe the additional features required for assembling context components at design-time. Section 4 describes the enhancements to the `BeanBox`: new visual manipulation capabilities, so that it is possible to drag and drop, serialize and de-serialize, group and ungroup nested beans. By means of these visual manipulations, beans can be put inside other beans. In Section 5, we conclude and talk about future work.

### 1.1. Related work

JavaSoft's `BeanBox` was used as a base because it is platform independent, supports Java 2, already had some support (although limited) for the `BeanContext` interface, and most important, because its source code was available.

Prototyping a new tool by reusing the `BeanBox` source code seems to be a common practice. Wang and MacLean [21] enhanced the `BeanBox` to support introspection features that assist in component assembly using IIOP links based on IONA OrbixWeb. Miller *et al.* created a simulation environment called JSIM [15]. The `BeanBox` source code is incorporated inside the impressive Sieve [6] collaborative visualization environment. Natarajan and Rosenblum [16] have extended the `BeanBox` to support events in the C2 architecture style [19]. Occasionally, implementation details about the internal working of the `BeanBox` are sought in the *comp.lang.java.beans* news group.

However, these are all examples of special purpose extensions. In comparison, this work may be seen as a general-purpose extension: extending the BDK to support the new standard runtime containment and services protocol. Moreover, this work also uncovers the internal working of the `BeanBox` and documents the information needed for reusing it, information that might be helpful for authors of future `BeanBox` extensions.

The BDK is platform independent, and it is lightweight with no special requirements. Its intent is testing, not production. Its approach is runnable design time (unlike other tools which build, generate code, and then run.) Thus, it reacts faster than similar tools. It is, therefore, a suitable platform for fulfilling our purpose: testing context components and demonstrating how a supportive builder works.

JavaSoft explains how to work with the `BeanBox` Kit [7]. Sun provides the source code, and encourages users to "play with it." There is some documentation within the code. However, there is no documentation about the design of the `BeanBox`. Even the projects listed above do not report a whole lot about the internal design of the `BeanBox`. Hopefully, our work can provide a jump-start to those who are attempting to reuse the BDK source code.

## 2. Understanding the assembly-design space

A seamless introduction of new features requires, first of all, understanding the design and philosophy behind existing ones. In this section we lay out the design space and define the fundamental terms. First, we explain the distinction between assembly and design time in the component's lifecycle. Then, we explain the difference between the visuality, symbolic-representation, and visibility of a component. Finally, we explain in these terms the behavior of existing visual bean-builder environments.

### 2.1. Lifetimes of a component

Components change form and appearance during their normal development and lifetime. We identify four lifetimes: (1) *compile time*—the embryonic stage of the component; (2) and (3) *assembly time* and *design time*—two periods of metamorphosis sometimes referred to as one—*building-time*; (4) *runtime*—component maturity.

The definitions of the first and last lifetimes are standard [18]:

**Compile time:** the time when the component's source code is compiled into byte code and "forgotten" thereafter. Typically done by third party component providers. That is, a component may originate from a different author in binary format and without its source code.

**Runtime:** the time when the application, which is built out of components, is executed.

The explicit distinction between assembly and design time is non-traditional, but one of importance, because often the fine line between the two environments is somewhat blurred. Since these may overlap or be interleaved, it is clearer sometimes to refer to them as *activities* rather than *times*.

**Assembly:** the activity of connecting. At assembly time the application (or component) is assembled from other (compiled) components. The activity takes place between the compilation of the components and the compilation (or serialization) of the application.

**Design:** the activity of specifying the look and feel of the application. At design time, the components' visual aspects are displayed, permitting a user-friendly mechanism for fine-tuning the application's look, feel, and behavior. In this activity, the programmer can also test the application.

### 2.2. Visuality, symbolism of components, and visibility

A component can be visual or non-visual, may or may not be associated with an icon (a symbolic image), and at times may be visible or invisible.

- **Visuality** A JavaBeans component may be visual or non-visual. A *visual component* can be displayed visually (i.e., made visible) in the final application. Visual beans are the most common kind. This is a natural consequence of the example set by Sun: the first beans were all GUI components, namely AWT components. JavaSoft even initially defined JavaBeans components to be "reusable software components that can be manipulated *visually* in a builder tool" [10]. However, beans may also be non-visual. Non-visual beans are useful for their functionality despite not having a visual appearance. For example, an adapter between two components is very useful but need not be displayed visually in the final application.

- **Symbolism** Some components are associated with a *symbolic image*, an icon, others are assigned one by the system, e.g., a label. For a JavaBeans component, the icon is specified by the bean author in the `BeanInfo` adjunct class. For a bean without an icon, the system instantiates a `Label` and uses it like an icon. The icon or label is used to display the list of available components (in the `ToolBox` window) when a `jar` is loaded. The icon or label is also used to visually display non-visual beans at assembly time.
- **Visibility** A JavaBeans component can be visible or invisible. *Visibility* is different than *visuality*. Visibility is the degree of being visible. It is possible to see visible beans. Invisible beans are hidden from the eye. Visuality, on the other hand, is the ability of being visual (at runtime). A visual bean is associated with a `java.awt.Component` object, which is displayed (at design time), regardless of whether or not it is associated with an icon. A non-visual bean is represented visually (at assembly time) by its icon, and if it does not have one, by a system generated label. But even visual beans can be at times invisible, e.g., by invoking `setVisible(false)` (at either design or runtime).[1]

### 2.3. Assembly and design environments

Different industrial JavaBeans visual bean-builder environments have taken different approaches on how to provide the user with assembly and design time control.

- **Two-window approach.** Sun's JavaStudio [11] displays two windows simultaneously. One window shows the assembly picture while the other the design of the application. The windows are fully synchronized. The assembly window shows all beans by their icon representation with in- and out-ports, permitting the users to connect an out-port to some other in-port. The design window shows only visual beans in their visual representation, and hides the connections. Thus the design window shows how the application would look at runtime. Moreover, the design window is an active running application, allowing the application to be tested immediately. For example, you can enter text into a `TextField` component, push a `Button` component, examine the reaction to mouse-movements, and so on.
- **Split-window approach.** IBM VisualAge for Java [5] displays only one window. But, within the displayed window, a dashed line denotes a special design region. You can create visual beans only in that region, and non-visual beans only outside that region. However, you can pull the connectors (wires) across the (dashed lines) borders, because the two regions are practically parts of a single window. Unlike JavaStudio, the design region is inactive. To test the application, the system must generate code, e.g., an applet, and compile and run it, a time consuming procedure.
- **Mode-toggling approach.** In Sun's `BeanBox` [9], assembly and design are one and the same. This is because the `BeanBox` tool kit is not meant to be a development tool, but rather a "proof of concept." It is a demo, testing, and prototype tool: it demonstrates the

[1]The method `java.awt.Component.setVisible(boolean b)` replaces the deprecated methods `hide()` and `show()`.

working of beans, it tests user-created beans, and it is a prototype that serves as an elaborated specification for industrial visual tools providers.
In `BeanBox` 1.1, the user can switch back and forth from a runtime to a design time view by toggling two environment options:

1. **Enable/disable design mode.** Design mode is enabled by default. When *design mode* is disabled, the `BeanBox` displays and behaves like the produced application would at runtime. This *runtime mode* is useful for testing the designed application without the extra `BeanBox` functionality as a developing tool. In disabled mode, the panel showing available beans is hidden. Selecting a component does not activate introspection, expose properties, or bring up customization panels; and all non-visual beans disappear. As a result, the application response time improves drastically.
2. **Show/hide invisible beans.** This is a misnomer,[2] as it really means show or hide *non-visual* beans. When a non-visual bean is added to the `BeanBox` container, a label with its name is displayed. Otherwise, the user would not see the newly added bean and would not be able to customize its properties or connect it with other beans. *Hiding invisible beans* makes all non-visual beans invisible (even in design mode). This is intended to provide the user with a view of how the final design would look like, and still permit fine-tuning the visual beans that remain visible.

Four combinations are hence possible in the mode-toggling approach:

– Enable design; show non-visual: *assembly* and execution only.
– Enable design; hide non-visual: *design* and execution only.
– Disable design; show non-visual: A "read-only" view of assembly mode.
– Disable design; hide non-visual: execution only (*runtime*).

In the next section we describe how new support for the runtime containment and services protocol can be seamlessly added to the assembly and design environment.

## 3. Manipulation and representation of `BeanContext` components

The essence of the extensible runtime containment and services protocol [1] is that (`BeanContext`) beans may be placed in (and removed from) their enclosing `BeanContext` bean. The `BeanContext` hence becomes a container of objects, which not only introduces a new logical hierarchical structure, but also provides to its inhabitants a service discovery (and obtaining) protocol.

In order to test beans implementing the `BeanContext` interface in a visual development environment, the beans need to have some kind of visual representation. Without such a representation, it would be impossible to manipulate them visually at assembly and design time. Several difficulties arise. When a new `BeanContext` component (or a proxy for one) is added to the `BeanBox` there is the question of how to represent that bean visually. The visual representation can be provided either by the bean author or by the development environment. Moreover, the bean can be a `BeanContextProxy` and itself a visual composite (extends `java.awt.Container`). In that case, there are two hierarchies to maintain and a possibility for inconsistency between the two.

[2] If the bean is already invisible why the need to hide it?

The BeanBox distinguishes (when visual representation is concerned) only between visual beans (which are a kind of java.awt.Component) and non-visual ones. A non-visual bean is represented by a special label object (an instance of sun.beanbox.OurLabel), which displays the bean's name. When *hide invisible beans* mode is selected or *design mode* is *disabled*, the labels representing non-visual beans are hidden. When enhancing the BeanBox with support for runtime containment and services protocol, there is also the BeanContext or BeanContextProxy case to consider, and all possible combinations of the visual and context cases.

Every bean can have a Component or Container, and/or a BeanContext, associated with it. A bean can establish that relationship either through inheritance by extending one of BeanContext, Component or Container classes, or by being a BeanContextProxy or a BeanContextContainerProxy or a BeanContextChildComponentProxy for that object.[3]

More abstractly, in terms of the COMPOSITE design pattern [4], a *visual component* is either a *visual leaf* or a *visual composite*. A visual composite, or a visual component, is any instance of a class extending the abstract class Container, or the abstract class Component, respectively. A visual leaf is a Component that is not a Container. Being a visual composite but not a visual component is impossible, since Container extends Component, as with the design pattern. Similarly, a *context component* is either a *context leaf* or a *context composite*. From a visual perspective, however, a context leaf and a components not associated with a context are treated the same. There are therefore only 6 combinations to consider: $\{$non-visual,visual-leaf,visual-composite$\}\times\{$context-leaf,context-composite$\}$. For each we offer a different policy for the visual representation:

1. **A non-visual context leaf.** The original BeanBox covers the case of a bean which is neither a Component nor a BeanContext. At assembly time, a special label represents a non-visual bean, which is hidden at design/runtime mode. No beans can be placed inside this bean.
2. **Both a visual and a context leaf.** A bean, which is a Component but not a BeanContext, should always be represented (in assembly time, design time, and runtime) by the Component itself. This behavior would be consistent with the behavior in the original BeanBox. No beans can be placed inside this bean. It is neither a visual nor a context composite, and therefore its visual representation should be used at all times.
3. **A visual composite context leaf.** A bean, which is a Container but not a BeanContext, is a common case, typically coded by a user who did not anticipate runtime containment. This kind of visual bean should always represent itself visually. Still, there is a catch. If the user places inside this bean other beans that expect services from a runtime environment, then those beans must be added to that container and also to the runtime context of some other bean.

   Even if it were possible to somehow manage this split behavior, it would still be too confusing to work with. Therefore, in the extended BeanBox version, every bean that is a Container but not a BeanContext is automatically associated with a new BeanContext. Then, every bean added to that container is also added to its associated BeanContext, which propagates the environment services according to the protocol.
4. **A non-visual context composite.** This is a kind of BeanContext bean that has no visual

[3] Component and Container are classes from the java.awt package. The others are all classes from the java.beans.beancontext package.

representation. It should be represented by a special kind of `Container` (e.g., `TransparentPanel`). In assembly time, the user can place inside this bean other beans, and in design/runtime the container should become "transparent", i.e., become itself invisible but leave the contained components visible.

5. **A visual leaf context composite.** A bean can be both a `Component` and a `BeanContext`. However, this is an unnatural case that probably ought to be disallowed. It is unnatural because the bean seems to have a contradictory behavior, i.e., as a leaf (`Component`) in the visual containment hierarchy, and as a composite, a collection (`BeanContext`), in the `BeanContext` containment hierarchy.
   A possible work-around is to represent such beans at assembly time by a special kind of `Container` (e.g., `OurPanel`, analogous to `OurLabel`), which will allow the user to visually put in it child beans, and at design/runtime by the `Component` associated with the bean. Beans placed inside this bean should be added to its associated container and also to its `BeanContext`.
6. **Both a visual and a context composite.** This is a bean that is a `Container` and a `BeanContext`. It is the simplest case. The bean should always represent itself, and there are no additional problems. Beans placed inside the component should be added to both the `Container` and to the `BeanContext`, either by the bean itself or by the environment.

## 4. `BeanBox` enhancements

To demonstrate the approach described in the previous section for design-time assembly of `BeanContext` components, we enhanced JavaSoft's `BeanBox` from BDK 1.1 [9] with the new visual manipulation capabilities for nested beans. Focusing on the most salient features we list here 6 visual enhancements: hierarchical visual containment, hierarchical context containment, drag and drop capabilities, de-serializing, grouping and ungrouping nested beans, and removing wrappers. By means of these visual manipulations, beans can be put inside context beans.

### 4.1. Hierarchical visual containment

It is difficult to imagine the context hierarchy unless a corresponding visual containment hierarchy displays it. Therefore, the first step is to add the ability to display nesting of visual containers.

The `BeanBox` class is a visual composite; consequently, when adding new beans, one can place the new bean inside the panel. In the standard `BeanBox`, however, it is impossible to place a new bean inside another visual composite, which might be itself inside the `BeanBox` and displayed on the screen. By fixing this discrimination, our extension permits new beans to be created and placed inside any AWT container. As a result, one can form nested visual components.

### 4.2. Extensible runtime containment and services protocol support

Once the support for presenting a visual hierarchy is in place, the next step is to support the runtime containment protocol. Sun's `BeanBox` already has a limited support for the extensible containment and services protocol [1]. The `BeanBox` implements the `BeanContextProxy` interface. A new bean added to the `BeanBox` container is automatically added also to the `BeanContext` peer, the `BeanContext` for which the `BeanBox` container is a proxy. Now that we have added the ability to place beans inside any visual composite, we need also to take care of adding the new bean to its *innermost* bean context.

The innermost context of a bean is not necessarily the BeanBox, but rather a BeanContext associated with the container to which the bean is added. Unfortunately, the BeanBox is totally ignorant of the existence of any other BeanContext component in the environment.

As a result, one can only test beans against the BeanContext for which the BeanBox object is a proxy.[4] While this capability is useful, it is very limited. In particular, one can test one's services beans but one cannot test one's own BeanContext beans.

This second extension provides users with the possibility of testing their own BeanContext beans. When a bean is created inside a nested container, the innermost context is located, and the bean is added to it.

### 4.3. Drag and drop

After adding the first two features, the enhanced BeanBox has already become a useful testing tool. However, there are still several problems that make the developer's life harder than ought to be. What if a bean, in its initial state, cannot be placed directly into some BeanContext, but could be if it were customized? There is no way to customize a bean before creating it. One can place a bean in the BeanBox panel and customize it there. However, so far there is no support for moving beans from one container into another.

Now that we may have more than one container, it is natural to want to be able to move items from one context to another. The BDK supports moving, but because the assumption is that there is only one container, namely the BeanBox, moving is interpreted as just changing the position in the same container. The fourth extension adds support for drag-and-drop moving that is sensitive to the underlying container's context. Recall that the BeanBox is a BeanContextProxy and hence both a visual and a context composite. With the drag-and-drop capability one can customize the bean in the BeanBox and then drag it into some other container.

Another reason for a drag-and-drop feature is to save development time. If a user by mistake places a bean in the wrong container, the only way to undo it in the standard BeanBox is to delete (cut) the bean and add a new one again. With the new drag-and-drop capability, the user can move the bean into the correct container.

### 4.4. De-serialization

In the BeanBox, there is a feature, named *Serialize bean*, that allows the user to dump a bean's state to a file. There isn't, however, any way to read back in the bean's state from the saved file (to de-serialize a bean). It is interesting that many more complicated features were implemented (e.g., making an applet from the BeanBox), but not this simple one. We have added this capability.

### 4.5. Grouping and ungrouping

There is no way to manipulate subcomponents of beans (e.g., customize, hook events up, bind properties), not even if the bean is a Container. For example, a bean that extends Panel and contains a button would not permit the programmer to select the button. As a result, one cannot change the button's label or color, or connect its ActionEvent to some other bean.

The fifth enhancement is the ability to manipulate subcomponents by ungrouping the bean. Ungrouping beans allows the programmer to separately manipulate each subcomponent in that bean.

[4] The BeanBox is provided with two examples showing the working of the BeanContext: the MethodTracing and the InfoBus services, but these cannot be changed.

The opposite operation is grouping. Once the bean and its subcomponents are customized, the user can re-group the beans. Grouping also disables any additional objects that are used internally by the `BeanBox` for manipulating beans (like wrappers).

### 4.6. Removing `Wrappers` from the visual hierarchy

The last enhancement is crucial for managing AWT and `BeanContext` hierarchies, since beans could be sensitive to their enclosing `BeanContext` environment (and contrariwise). In the standard BDK, every bean is automatically placed in a `Wrapper` (a kind of `Panel`) before being added to the `BeanBox`. The only function of the wrapper objects in the visual presentation of beans is to draw a hashed bar around selected ones. The wrapper, however, can confuse a `BeanContext` to reject an otherwise acceptable bean. As a result, certain context-sensitive beans can be tested only by removing the additional layer of wrappers.

## 5. Conclusions and future work

In this paper we had two objectives:

1. To present a better understanding of the design time and space of component assembly in a visual builder.
2. To present the new *ContextBox* tool, which enhances the `BeanBox` from Sun by adding the ability to also manipulate beans implementing the `BeanContext` interface.

The first objective is useful in understanding other software component models [14, 17, 19, 20], while the second is strongly related to the JavaBeans technology. Design and implementation issues, which were omitted due to space limitation, can be found in further detail in [13]. These may be of help to others who wish to create enhancements of their own.

Our *ContextBox* is a strict extension of the `BeanBox`. However, during our exhaustive use of the `BeanBox`, we have occasionally encountered some odd system behavior. Only those which were surely caused by bugs and could not be interpreted as undocumented features have been fixed in the *ContextBox* version. As with any software program, *ContextBox* is far from perfect. There is always room for improvement, both in the existing features and in the new added features.

Some ideas for future enhancements are to allow the user to disconnect connected beans, to provide more complex event hookups, and to visually show connections during design time. The first enhancement would save a lot of testing time. Wrongly connected beans would not have to be erased and connected all over again. Instead, only wrong connections could be undone. The second enhancement could be very handy for more complicated tests, e.g., when the user loses track which beans are connected to where, in the middle of the connecting process. All these features are not necessarily related to the `BeanContext` beans, but are also very important for making the BDK more accessible and thus useful.

The `BeanContext` specification leaves a few issues open. For example, when a `BeanContextChild` is associated with a `Component` and a `BeanContext` with a `Container`, it is not clear how they should interact. The specification gives three possible models and is therefore not conclusive enough. This makes the life of a tool designer difficult. The *ContextBox*, as a prototype, can be of help here too.

**Acknowledgment** We thank Mitchell Wand and William Clinger for their valuable comments.

## References


[1] L. Cable. Extensible runtime and services protocol for JavaBeans. Version 1.0, JavaSoft, Dec. 3 1998. ftp://ftp.javasoft.com/docs/jdk1.2/beancontext.ps.

[2] L. P. G. Cable. Proposal for a drag and drop subsystem for the Java foundation classes. Final Draft 0.96 of the API for JDK 1.2, JavaSoft, Aug. 24 1999. http://java.sun.com/beans/glasgow/-dnd.ps.

[3] B. Calder and B. Shannon. JavaBeans activation framework specification. Version 1.0a, JavaSoft, May 11 1999. http://java.sun.com/beans/glasgow/JAF-1.0.ps.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[5] IBM. VisualAge for Java. http://www.ibm.com/software/ad/vajava/.

[6] P. L. Isenhour, J. B. Begole, W. Heagy, and C. Shaffer. Sieve: A Java-based collaborative visualization environment. In *Late Breaking Hot Topics Proceedings, IEEE Visualization'97*, pages 13–16, Phoenix, AZ, Oct. 1997.

[7] JavaSoft. BeanBox. http://www.javasoft.com/beans/software/beanbox.html.

[8] JavaSoft. JavaBeans Glasgow specification. http://java.sun.com/beans/glasgow.

[9] JavaSoft. BDK 1.1, November 1997. http://java.sun.com/beans/software/-bdk_download.html.

[10] JavaSoft. JavaBeans API specification. Version 1.01, Sun Microsystems, Mountain View, Calif., July 24 1997. http://www.javasoft.com/beans/docs/spec.htm.

[11] JavaSoft. JavaStudio 1.0, March 1999. http://java.sun.com/studio.

[12] O. Kluyt. Unlocking the BeanContext API. Java Developer Connection, May 1999. http://developer.java.sun.com/developer/technicalArticles/Beans/-BeanContext.

[13] D. H. Lorenz and P. Petkovic. ContextBox: A BeanBox environment for design-time assembly of BeanContext components. Technical Report NU-CCS-99-04, College of Computer Science, Northeastern University, Boston, MA 02115, Nov. 1999.

[14] Microsoft. DCOM architecture. White paper, Redmond, Wash., 1997.

[15] J. A. Miller, Y. Ge, and J. Tao. Component-based simulation environment: JSIM as a case study using Java beans. In *WSC '98: Proceedings of 1988 conference on Winter simulation*, pages 373–382, 1998.

[16] R. Natarajan and D. S.Rosenblum. Merging component models and architectural styles. In *Proceedings of the third international workshop on Software architecture*, pages 109–111, Orlando, FL, Nov.1-5 1998.

[17] J. Siegel. CORBA fundamentals and programming. John Wiley & Sons, New York, 1996. OMG specifications http://www.omg.org.

[18] C. Szyperski. *Component-Oriented Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[19] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. W. Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions of Software Engineering*, pages 390–406, June 1996.

[20] A. Thomas. Enterprise JavaBeans server component model for Java. Prepared for Sun Microsystems by Patricia Seybold Group, 1997.

[21] G. Wang and H. MacLean. Architectural components and object-oriented implementations. A position paper presented at the 1998 International Workshop on Component-Based Software Engineering, ICSE, 1998.