LA-UR-99-2981

*Approved for public release;
distribution is unlimited.*

Title:

Composition Environment for Simulation Development

Author(s): Clay P. Booker and Greg Lacey

Submitted to:

http://lib-www.lanl.gov/la-pubs/00460093.pdf

Composition Environment for Simulation Development

An LDRD Project Implemented by
Clay P. Booker, TSA-3 and Greg Lacey, TSA-3

ABSTRACT

A prototype Composition Environment is implemented and future modifications required to make the prototype a practical tool are explored. A distributed Repository to warehouse reusable components is demonstrated, and a simple run time adapter that can attach itself to a composed simulation and execute it with little user manipulation is tested. The Composition Environment, Repository, and Run Time Adapter together allow simulations and components to be completely portable; a simulation can be composed, configured, and run anywhere. Thus, a component or simulation can be loaded on one or more notebook computers and taken to off-site locations for demonstrations or customer applications.

INTRODUCTION

The Laboratory, like many other organizations, is often faced with the situation of having to quickly respond to new or emerging problems with an analysis or technology that is based on the use of modeling and simulation techniques. This often requires the development of a new simulation design or the modification of an old, outdated simulation. Typically, one sets about designing a new simulation from the ground up, perhaps borrowing or rewriting code from other, older simulations to meet some or most of the modeling needs. Almost always, we depend on domain experts to reapply their expertise again and again to rewrite simulation models, often from scratch.

In addressing the above situation, we have set out to develop an ever growing *Repository* of reusable components embodying developed models. Further, we are developing an *Environment* in which simulations or composite components can be *composed* using those already built, tested, and trusted components. With these two tools, it will be possible to respond to an urgent demand for a new simulation or provide answers based on a simulation in a more timely fashion with far greater confidence than we have ever before experienced. Experts would no longer be called upon to spend a great deal of time reworking and recoding models they have built and rebuilt time and again; instead, they could concentrate on *configuring* the appropriate, already built model for its current application. The bulk of the expert's time could be more productively spent on understanding the problem at hand and examining alternate scenarios.

In the past, we have been hampered in communicating and interacting with our potential or actual customers because we typically either had to invite them here for a demonstration of our simulations, or we had to reduce a simulation to a series of screen shots on transparencies and talk our way through the component that we have built or propose to build. It is of great value to us and our customers that we can load a simulation on a notebook and take that fully functional simulation to the customer for demonstration. To realize that, we require a run time environment that is completely portable.

The project and tools described here have been developed to provide a completely portable environment in which components may be developed, warehoused, and combined to respond to any demand.

DESIGN ELEMENT GOALS

Some fundamental design elements are required to form a Composition Environment. These include a basis for components that endows them with the ability to arbitrate the construction (composition) of a more complex component and guarantee that such a *composed* component is properly built and that the constituent parts are compatible. A variety of visually based design elements that facilitate the composition of such components is required. These expected visual design elements include a Workspace in which to compose components from other, perhaps more primitive components, a Repository of extant components, Palettes where a Repository's available components are displayed and from which a user may select, and Configuration Editors which enable the user to customize components. A Repository must be easily accessible so that existing components are available to the user — even to a user far removed from the Repository. Finally, a Composition Environment is useless unless the simulation or components can be easily and portably executed. Thus, Run Time Adapters are required that are able to *intelligently* access the run time features of a component and execute it with little intervention by the user.

Congruous Components

Congruous Components are components constructed so that they can query each other during the composition phase and, by various means, indicate whether or not they can work together in the new component that is being composed. They actively participate in the composition process and act to prevent the composition of an ill-formed, greater component. A greater component composed from congruous components is also congruous.

For example, suppose the user wishes to construct a new component to represent a satellite from an existing set of components residing in a repository. For a simple satellite representation, the constituent components are a Sensor component which is the primary, desired function of the satellite, a Comm component which communicates with a ground station, a Motive component which calculates the motion and thus the time dependent position of the satellite, and a C2 component which integrates data from the other three components and sends appropriate signals to the ground station via the Comm component.

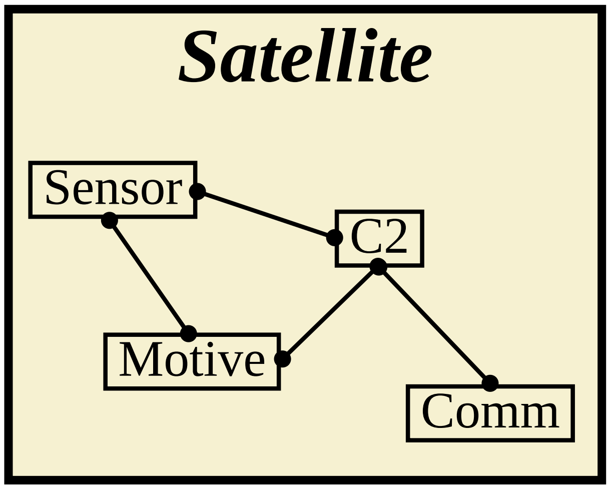


Figure 1. **Simple Satellite constructed from basic components**

The set of Motive components from which the user *might* choose for the new satellite is vast but only a very few, those dealing with orbital mechanics, are suitable for any satellite component, and the selection of the particular satellite type (for example, Geosynchronous) might further restrict the selection. If the user were to select an internal combustion engine for the Motive component, the container component which is congruous *should* reject it. Similarly, a tin-can-and-string Comm component would be rejected for inclusion. Instead of the user having to be the expert in building the satellite, the constituent components themselves should arbitrate the construction of the greater component so that no inappropriate selections *can* be made, and when all the components are put together, the resulting satellite is *guaranteed* to be functional even it is perhaps not *the best* representation of the desired satellite.

In this paradigm, the expertise resides in the components rather than necessarily in the user. Basic components are presumably designed and constructed by domain experts who also design into the components *congruous elements* so that a collection of congruous components will work together properly. This should be particularly true of *Atoms* which are components which cannot be broken down into constituent components.

Workspace and Palettes

In order to compose components, a visual manipulation environment is required. The user should have Palettes from which to select from existing components and a Workspace in which to assemble them. Such an environment is imaginatively depicted below in Figure 2.

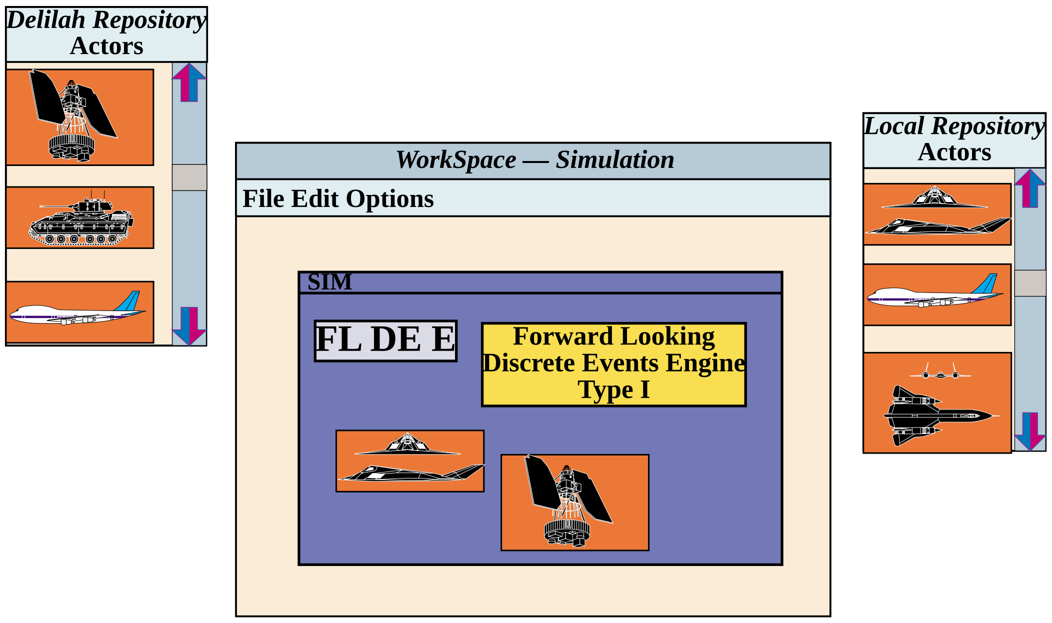


Figure 2. Workspace and Palettes.

Along with the these elements, a variety of support elements such as a clipboard and preferences selection dialogs are needed.

Repository

For the proposed Composition Environment to be of use to anyone, it must include a repository which stores already constructed components so that the user may select from a variety of components to build his new component. Further, this Repository should be distributed so that centers of domain expertise may be accessed for trusted components with which the user may construct a new component and have some confidence in its viability. Such a Repository is represented below in Figure 3. Note the provision for a user to publish his own local Repository so that other users may access it.

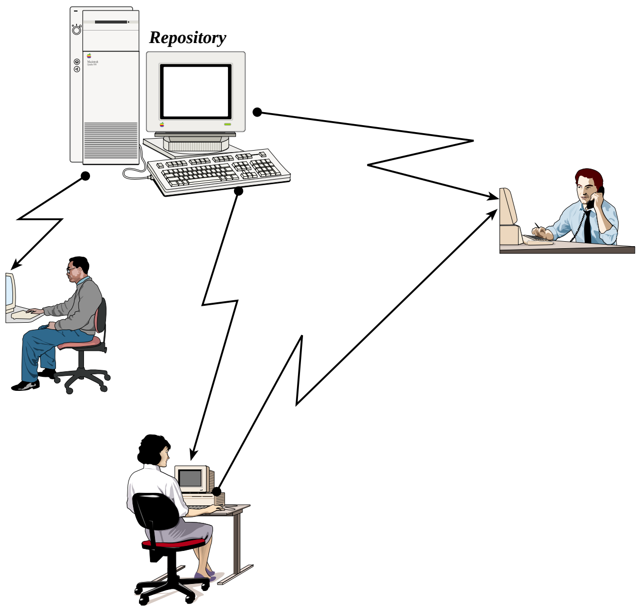
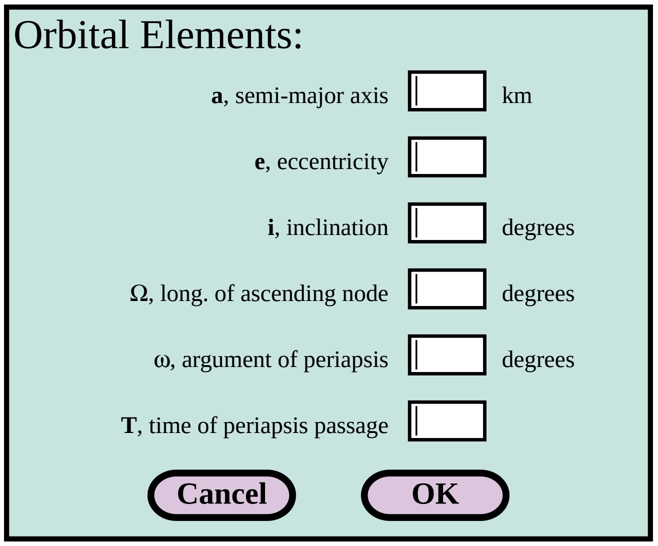


Figure 3. Distributed Repository Example.

Configuration Editors

While a component may be built to accurately represent a physical object or collection of physical objects, it still must be configured for the specific application. For example, suppose the user has selected a proper, orbital-based Motive component for his satellite; that Motive component must be *configured* with the proper orbital elements before the new satellite component can be used.



Orbital Elements:

a, semi-major axis		km
e, eccentricity		
i, inclination		degrees
Ω, long. of ascending node		degrees
ω, argument of periapsis		degrees
T, time of periapsis passage		

Figure 4. Sample Configuration Editor.

Run time

In order the execute the simulation (Note that a simulation is just a component from the stand point of the Composition Environment.), a run time adapter (RTA) is needed. The RTA must use the congruous component features to negotiate with the attached component in order to extract the appropriate interface to initiate and control the run. Accordingly, the user is only expected to provide a reference to the component he desires to run and the RTA will work with the component to set up the run and execute. For distributed simulations, the RTA works with the component to distribute the sub components on the platforms specified in the Composition Environment with the Configuration Editors; the user will not be called upon to manually distribute the components of the simulation.

BASIC IMPLEMENTATION

The basic implementation of the goals set forth in the previous sections requires the selection of a language; as explained below, the Composition Environment and the congruous component are implemented in Java. Further, the congruous component architecture must be designed and implemented along the lines established above and be compatible with a distributed Repository.

Java

In order to store and manipulate objects, an Object-Oriented language must be used. There are a variety of usable Object-Oriented languages and until recently C++ was apparently among the most popular of those for simulation design and implementation; SmallTalk has been used in some applications, but it is demonstratively slow and has failed to realize its cross-platform promise. In the last few years, Java has emerged as a language eminently suitable for simulation implementation and has gained phenomenal acceptance across the programming community — so much so that it is squeezing out support for other, older languages such as SmallTalk.

Java has many strengths and very few weaknesses that make it ideal for implementing a simulation:

- Java is truly a cross-platform language with even primitive data types completely specified across platforms [1]. Components designed and implemented on one platform are completely portable and instantly usable on a completely different platform. This can greatly facilitate the use of verified and/or validated models because once a component is verified and/or validated, it is usable without any change whatsoever on any platform.
- Persistence, the ability to write an instantiated object as a sequence of bytes to an output stream, is an aspect of the Java language; this greatly expedites the task of storing a component in a Repository and extracting it again from that Repository. As a result, an object may be *written* to an output stream which in turn may be connected to a local file, a networked Repository Server, or a networked Data Base Manager. Such a persistent object is generally called *serialized* which indicates that it is suitable for transmission over an I/O Stream. To implement the same functionality in C++ is a major task and can severely impact the basic design of the component. See Reference [2].
- Java has built-in and complete support for network communication so that implementation of the distributed Repository is vastly eased. Further, the implementation of a distributed simulation over a heterogeneous mix of computers over a network is notably simplified and enhanced; our tests of message passing between machines on a network shows that Java implementations exhibit speeds comparable to that of previous SAMSON tests which was implemented in optimized C code [3]. See References [4, 5, 6, and 7] for more information on Java's network facilities.

- Speed on many platforms with a Just-In-Time Compiler (JIT) is comparable to C++. In future releases, Sun expects speeds to approach that of C with HotSpot technology along with other improvements in the Virtual Machine.
- Java has standardized the linkage to native code (for example: C, C++ libraries) so that the incorporation of any legacy code is facilitated and is largely the same on any platform. See Reference [8].
- Threading is part of the Java language. As Java Virtual Machines mature and make better use of native Threads, Java can make effective use of multiple CPU machines no matter what the platform. See References [9 and 10].
- Java incorporates a facility called *reflection* which allows an object instance to be scrutinized at run time. With reflection virtually all aspects of an instance's implementation such as class hierarchy, methods, and field variables can be interactively discovered; methods may even be invoked without casting the object to the exact class it represents.
- The Java language supports the use of a *special* comment block, beginning with "/**" and ending with "*/", to facilitate embedded documentation of the code; this facility is called **javadoc**. Within such a comment block special javadoc tags are recognized to mark features of the code such as method arguments, method return value, and the like. When commented code is processed by javadoc, an HTML document is produced which provides good documentation of the code. The use of javadoc tags in a code is illustrated in the code listings included in the appendices; the Composition Environment is fully commented using javadoc tags.
- The Zip compressed archive has been standardized as part of the Java language specification. As a result, Java has all the facilities to create, read, and manipulate Zip archives. This facility is useful when dealing with a large number of objects which belong together or with large objects that must be transferred over a network.

Given all of these advantages and implementation of Java on a considerable number of platforms ranging from Macintosh, to Windows, to most Unix platforms, the Composition Environment is implemented entirely in pure Java.

An upcoming addition to the Java standard (1.2) is the Java Foundation Classes generally known as *Swing*. It provides a new, better, lightweight set of graphical components that are more independent of the platform on which the application runs. Swing also incorporates pluggable *look-and-feel* so that it possible to define a look-and-feel unique to the application that is *fairly* faithfully represented across platforms. Because this the intent of this study is to stay on the *forefront*, Swing is incorporated into the implementation early and is used exclusively for the visual representation. See References [11, 12, and13] for more information on Swing.

JavaBeans

Another important innovation introduced by Java is the JavaBean specification and Sun's BeanBox. The JavaBean is a reusable component architecture and the BeanBox is a *prototype* visual development environment that allows the user to combine JavaBeans to create a new, more complex Bean.

The JavaBean specification introduces many concepts that vital to the practical use of reusable components in a Composition Environment. These include:
- The JavaBean which is a specification for a reusable component.
- A specification for methods that access the attributes of a Bean.
- The use of Jars, a specialized Zip archive, to store a component. Primitive components are available from the Repository in Jars and an aggregate component is stored in a Jar.

The BeanBox exhibits many of the features essential for the Composition Environment such as:

- A Workspace in which to combine components and Palettes from with to choose components from a very *primitive* Repository.
- The Custom Editor facility to configure a component.
- A custom class loader to load objects unknown to the environment during run time, and use them as if they were part of the environment from the beginning. With a class loader, an application need *know* nothing about a candidate component; it may instantiate that component, and *discover* its attributes as needed.

Unfortunately, the BeanBox was primarily designed for the aggregation of *visual* components rather than the composition of components that may be combined to create a simulation or an application; in particular, it lacks strong support for inheritance. As a result, it lacks a variety of key elements thus making it unsuitable for composing simulations. The BeanBox's notable missing elements constitute many of the design elements described in this paper.
The JavaBean and the BeanBox are described in a variety of books; see References [14, 15, 16, and 17] for a small sample.

Congruous Components

The *Congruous Component* specification is a major, daunting design task. The methodology must be flexible enough to accommodate any desirable simulation design, but simple enough that designers can and *will* actually make use of it. With that in mind, the design implemented here consists of just two basic elements:

- A container class that may contain other containers and a single instance of a *special* class called Personality. This container class is called **GenericContainer**; see Appendix A for the listing of GenericContainer.java.
- A **Personality** interface that defines the behavior and oversees the composition of a container once it becomes part of that container. See Appendix B for a listing of Personality.java.

The combination of a GenericContainer with a single Personality constitutes a **component** in the Composition Environment described in this paper.

An empty GenericContainer has no function or usable attributes until it has been given a Personality. As illustrated in Figure 5, the fist step in creating a component is to add a Personality to a GenericContainer. Once a GenericContainer *owns* a Personality, it becomes a component albeit a undoubtedly incomplete one. However, with its Personality, the new component can inspect other components that are candidates for inclusion to define its functionality and attributes. The GenericContainer subjects all such candidate components to the scrutiny of its Personality.

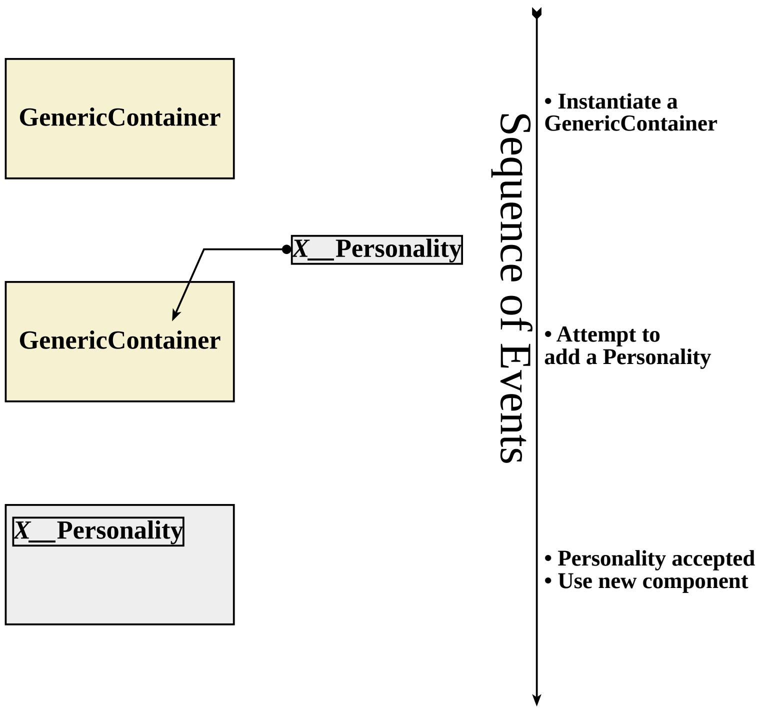


Figure 5. Transformation of a GenericContainer into a Component by the addition of a Personality.

If a GenericContainer has no Personality, it will reject all attempts to add other components to its inventory as illustrated in Figure 6.

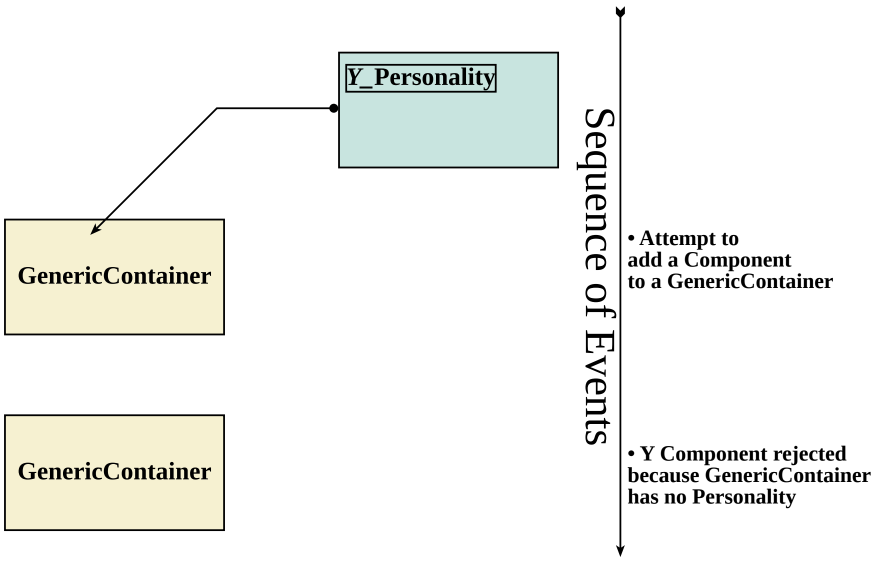

Y_Personality
GenericContainer
• Attempt to add a Component to a GenericContainer
GenericContainer
• Y Component rejected because GenericContainer has no Personality
Sequence of Events

Figure 6. Doomed attempt to add a Component to a GenericContainer.

However, when a GenericContainer has a Personality, it is possible to add other components to the GenericContainer's inventory as illustrated in Figure 7.

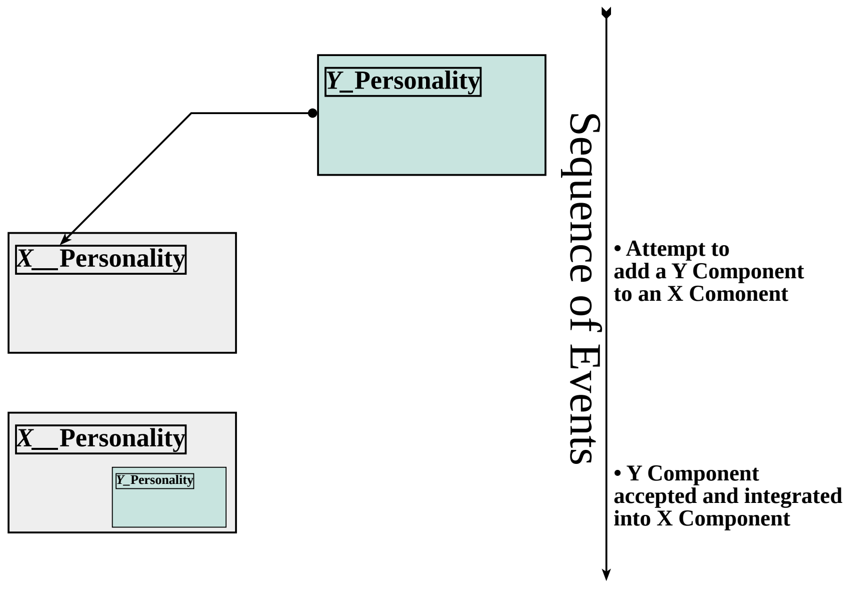

Y_Personality
X__Personality
X__Personality
Y_Personality
Sequence of Events
• Attempt to add a Y Component to an X Comonent
• Y Component accepted and integrated into X Component

Figure 7. Adding a Subordinate Component to a Component.

The process of adding a component to the GenericContainer can be a complex one. The GenericContainer does not accept the new component until it has been scrutinized by its Personality. The Personality may subject the candidate component to a variety of tests to see if the candidate is admissible for inclusion in the Personality's component; generally, it initiates this inquiry by extracting the candidate's Personality. It may then use reflection or it may subject the candidate component's Personality to a variety of queries in order to determine the candidate's suitability. The Personality may also maintain a record of past components added to its component and adapt itself; accordingly its criteria for admission may narrow and sharpen as more components are added to its component. A particular instance of a Personality can become quite sophisticated.

Once a candidate component is deemed suitable for inclusion in the GenericContainer, it is the responsibility of the Personality to *introduce* it to other, already accepted components comprising the Personality's component. The Personality may also choose to delegate some of the introduction tasks to other components. In the end, all the components comprising the Personality's component are completely connected, and upon completion of the Personality's component, comprise a fully functional component that perhaps may be executed on its own or may be included in another, greater component.

With this framework, complex components *m*ay be composed from a collection of simple components; note that in this context, a simulation or an application is just a component. In order to begin this process, there must exist a collection of components which cannot be broken down into a collection of components; these are called *Atoms*. Atoms are generally simple models that perform a single task. Both their Personality and Algorithm embody the domain expert's expertise. The Personality of an Atom declares the atomic nature of its component through its `separable()` method which returns false for an Atom. In a more complex component, constructed from other components, the domain expert's expertise is mainly embodied in the design of the complex component's Personality which will admit only certain other components meeting the expert's criteria.

An example of how a congruous component set might be designed and implemented, drawn from an existing simulation project, is given in Appendices D-F. The relationship between the classes and interfaces for the example NIC2 atom is shown in Figure 8:

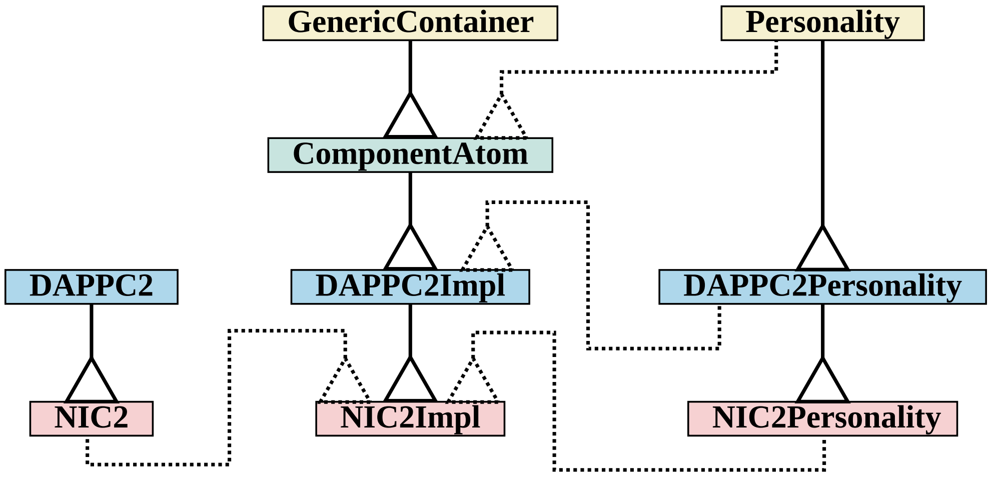


Figure 8. NIC2 makeup.

Here, ComponentAtom is provided as the basis for all atoms in this simulation as it combines GenericContainer and Personality and implements some of the basic Personality functions proper for an atom. The DAPPC2 classes provide a basis for any C2 component in the simulation; it implements the DAPPC2Personality and extends ComponentAtom providing some implementation of methods beyond ComponentAtom and germane to C2 functions. Finally, the NIC2 classes implement the functionality required for the NIPlayer's C2. It should be noted that DAPPC2Impl implements RunTimeCommands making the C2 the point of command for executing a Player in the simulation.

The NIPlayerPersonality classes (Appendix F.) show how the composed, congruous component, NIPlayer, is implemented. The NIPlayerPersonality admits <u>only</u> certain components to the composed NIPlayer. It queries any candidate component by extracting its Personality and examining it via *Reflection*. If the candidate component's Personality meets the NIPlayerPersonality's simple criteria, it is admitted to help comprise the NIPlayer; if not, it is rejected. An interesting feature of the NIPlayerPersonality is it relationship to the NIC2 described above. If the NIC2 is added before other components, the NIPlayerPersonallity subjects all subsequent candidate components to scrutiny by the NIC2 before that component may be added; if the NIC2 rejects a candidate component, the NIPlayerPersonality then rejects it as well. If the NIC2 is added after any or all other components of the NIPlayer, the NIPlayerPersonality causes the NIC2 to evaluate the already resident components. If the resident components <u>all</u> meet the NIC2's approval, then the NIC2 is added as a resident component; if the NIC2 rejects <u>any</u> of the resident components, then the NIC2 is rejected by the NIPlayerPersonality.

Jars

A Jar is simply a Zip Archive that has certain *optional, but anticipated* entries specified by the Java standard. It preserves directory structure and may be compressed so that a Jar can be efficiently transmitted over a network. The anticipated Jar entries are:

- The META-INF directory in which information about the other inhabitants of the Jar is stored. The BeanBox and similar environments expect this information to reside in the text file, MANIFEST.MF under this directory.
- A number of serialized objects which may or may not be JavaBeans. Each such object's file generally ends with the suffix, ".ser".
- A number of class files which may or may not be JavaBeans. These always carry the ".class" suffix.
- Any auxiliary files, such as image files, that may be required by the Bean.

The MANIFEST.MF *may* have the following information about other inhabitants:

- The fully-qualified name of the *file.*
- Whether or not the entry is a JavaBean; if it is, it is an object to be instantiated by the BeanBox or other such environment.
- The algorithm(s) used to digest the file.
- The digests generated by the specified algorithms

An entry may include one or as many as all the above information. A sample MANIFEST.MF is shown below in Listing 1.

```
Manifest-Version:   1.0

Name:  lanl/tsa3/simulation/actor/dapp/component/NIC2Impl.ser

Java-Bean: True

Name:  lanl/tsa3/simulation/actor/dapp/component/NIC2.class

Name:  lanl/tsa3/simulation/actor/dapp/component/NIC2Impl.class

Name:  lanl/tsa3/simulation/actor/dapp/component/NIC2Personality.class
```

Listing 1. An example MANIFEST.MF

A MANIFEST.MF entry may contain digital signature information so that some level of confidence in the validity of the object may be associated with the entry; this could be important for verification and validation concerns.

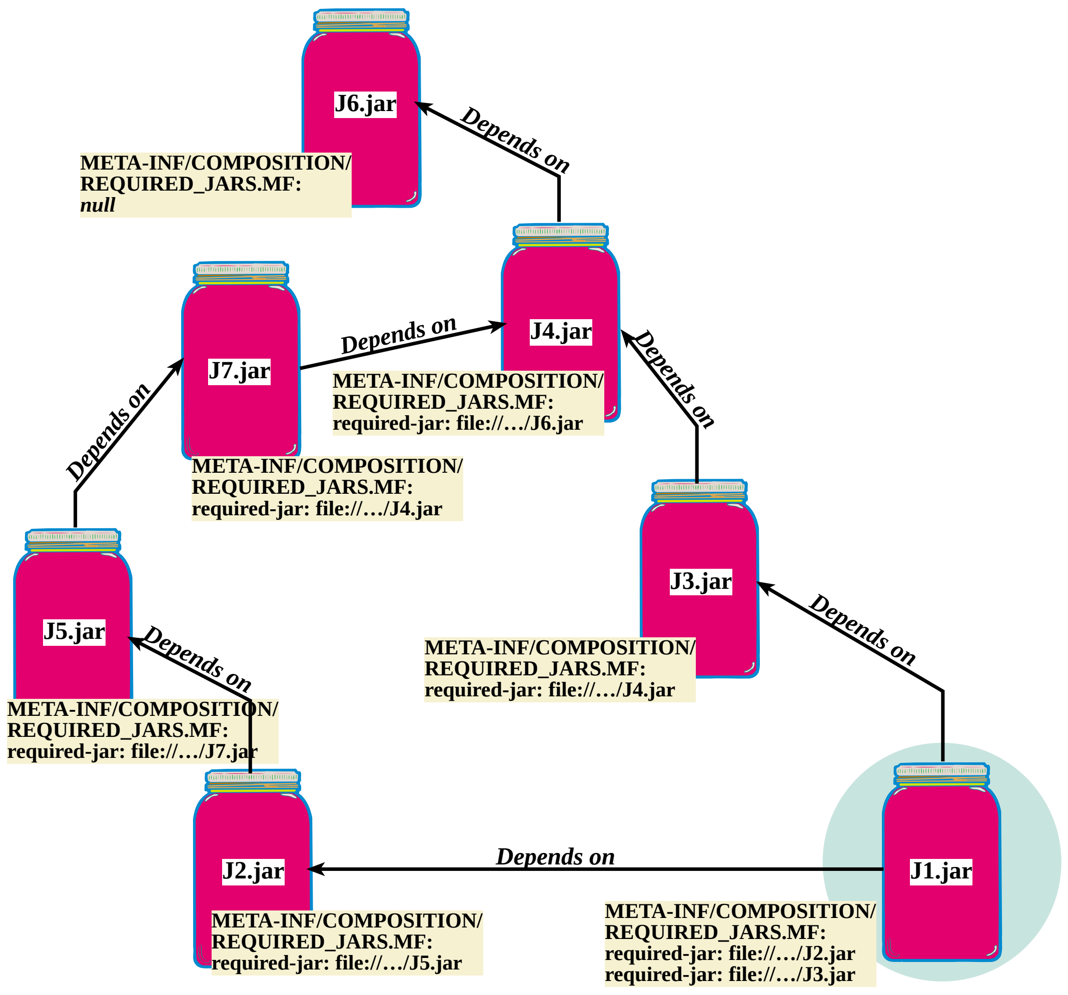
Unfortunately, the Manifest file is inadequate for use in a Composition Environment as complex and flexible as that considered here. Specifically, the Manifest file keeps no inheritance information so that a particular Jar must contain a complete set of classes to allow its bean to be instantiated. The Composition Environment described here strongly encourages the use of inheritance and adds to the META-INF portion of the Jar to carry inheritance information. It was decided that it would be imprudent to alter the MANIFEST.MF file to accommodate inheritance information because the form of the MANIFEST.MF file is still evolving and is in the hands of others. Instead a new directory is added under the META-INF directory, COMPOSITION. Under this directory a variety of files may be provided to furnish information critical to composition. At the time of this writing, only one file is expected by the Composition Environment, REQUIRED_JARS.MF. This file specifies, *in order*, the Jars that must be loaded before the current Jar is loaded. The Composition Environment then obtains the specified Jars from the Repository and examines their REQUIRED_JARS.MF. Finally, it loads the required Jars in the proper order. An example that corresponds to the MANIFEST.MF file in Listing 1 above is shown below in Listing 2; note the URL format in referencing required jars:

```
required-jar:  file://localhost/Repository/Local/.Jars/Simulation/DAPP/DAPPInfrastructure.jar
required-jar:  file://localhost/Repository/Local/.Jars/Infrastructure/Component/BOOTPManager.jar
required-jar:  file://localhost/Repository/Local/.Jars/Infrastructure/Component/EtherliteManager.jar
required-jar:  file://localhost/Repository/Local/.Jars/Part/DAPP/SensorParser.jar
```

Listing 2. REQUIRED_JARS.MF

Figure 9, below, shows how Jars may set up an inheritance hierarchy using REQUIRED_JARS.MF. For this example, the Composition Environment would load the Jars in the order:

1). **J6.jar**, 2). **J4.jar**, 3). **J7.jar**, 4). **J5.jar,** 5). **J2.jar**, 6). **J3.jar**, 7). **J1.jar** .



J6.jar
META-INF/COMPOSITION/ REQUIRED_JARS.MF: null
Depends on
J4.jar
META-INF/COMPOSITION/ REQUIRED_JARS.MF: required-jar: file://…/J6.jar
J7.jar
META-INF/COMPOSITION/ REQUIRED_JARS.MF: required-jar: file://…/J4.jar
J3.jar
META-INF/COMPOSITION/ REQUIRED_JARS.MF: required-jar: file://…/J4.jar
J5.jar
META-INF/COMPOSITION/ REQUIRED_JARS.MF: required-jar: file://…/J7.jar
J2.jar
META-INF/COMPOSITION/ REQUIRED_JARS.MF: required-jar: file://…/J5.jar
J1.jar
META-INF/COMPOSITION/ REQUIRED_JARS.MF: required-jar: file://…/J2.jar required-jar: file://…/J3.jar

Figure 9. Example inheritance hierarchy of Jars

Accordingly, a Composition Environment Jar resembles that depicted in Figure 10, below. Such a Jar maintains the META-INF/MANIFEST.MF as the Java Jar standard. It adds the META-INF/COMPOSITION directory with its mandatory file REQUIRED_JARS.MF as specified above. In the future, more information may be added to this directory; see the section **Future Work**, below. The Jar *may* contain a serialized object which is the only form of Bean recognized by the Composition Environment; this is denoted by the bean icon next to it in Figure 10. If a Jar does not contain a bean, then it undoubtedly is used for inheritance by a Jar that does include a Bean. Virtually all Composition Environment Jars will contain at least one class file corresponding to the Bean.

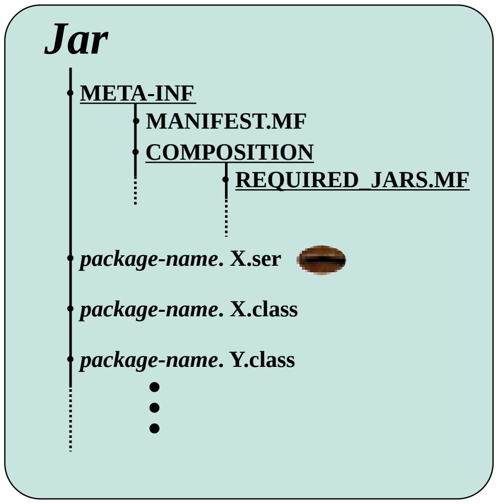


Figure 10. Composition Environment Jar structure.

Icons

Since the Composition Environment is fundamentally a *visual* setting, it is desirable to associate a icon with a given component that somehow represents the function or attributes of that component. There is provision in the JavaBeans standard for such an icon; the `BeanInfo` object has a method to fetch the Icon associated with the Bean. Unfortunately, that method returns an Image which is not a platform independent object. Further, it seems improper to *force* a developer to invent an icon for his component.

The Composition Environment provides platform independent icons using the Swing `ImageIcon` as part of the repository. The developer need only provide a String specifying the desired icon to represent the component, and the Composition Environment is responsible for searching the Repository for the required `ImageIcon` to display (See the Personality's method, `getIconName()` in Appendix B.). For example, the NIC2 component specifies its icon with the String: "`/Actor/Part/DAPP/C2/NIC2`"; the Composition Environment searches the Repository under the Icon directory, `/Icons/Actor/DAPP/C2`, for the file, "`NIC2.ser`", which is the `ImageIcon` containing the icon.

In practice this has been a clean way of associating an icon with a component, and it avoids any requirement that a component developer provide an icon as part of the component's code. In the case of a developer that refuses to address the icon question, others may make the decision external to the code and simply provide the proper icon in the Repository. It is anticipated that this feature may alleviate the incorporation of legacy code in the Repository in the future.

Tags

In order to efficiently represent the contents of a Repository, it was deemed undesirable to transfer all the Jars of a particular category to the client Composition Environment every time a Palette for that category is summoned. Moreover, a Palette represents the component simply with an icon and description. Accordingly, a compact representation for each of the residents of the Repository is required that is small enough so that network transmission times for a large number of them would not be onerous and that contains succinct information reflecting the attributes of the resident so that intelligent selections can be made from the Palette. To fill this need, the Tag file was devised.

A Tag has the following entries as illustrated in Figure 11.:

- **icon**: This is a portable icon that visually represents the component associated with the Tag. It is implemented as an Image Icon (See the Swing References.) because that class implements the only portable form of image in the Java standard. The icon conforms to a 20 by 20 pixel square.
- **icon name**: A String that references the icon in a machine independent way as described above.
- **name**: The name of the component.
- **URL**: A URL that points to the Jar containing the component. When using this field, the Composition Environment first assumes that the Jar is to be found locally and thus ignores the host part of the URL; it then searches the Local or WIP sections based on the path specified in the URL. If the specified Jar is not to be found in the Local Repository, the Composition Environment uses that path to search any user-specified preferred Repositories available on the net. Finally, if the specified Jar is not to be found there, the Composition Environment uses the host and port specification of the URL to retrieve the Jar from that Server, if active. In the future, this search sequence may be modifiable by the user.
- **Author name**: The name of the author(s) of the component referenced by the Tag.
- **Version**: A version String associated with the component.
- **Short Description**: A short description of the component.
- **Long Description**: A longer description of the component.

A Tag is realized as a Zip compressed archive so that the above heterogeneous, small objects can be kept together as a set in a single, compressed archive.

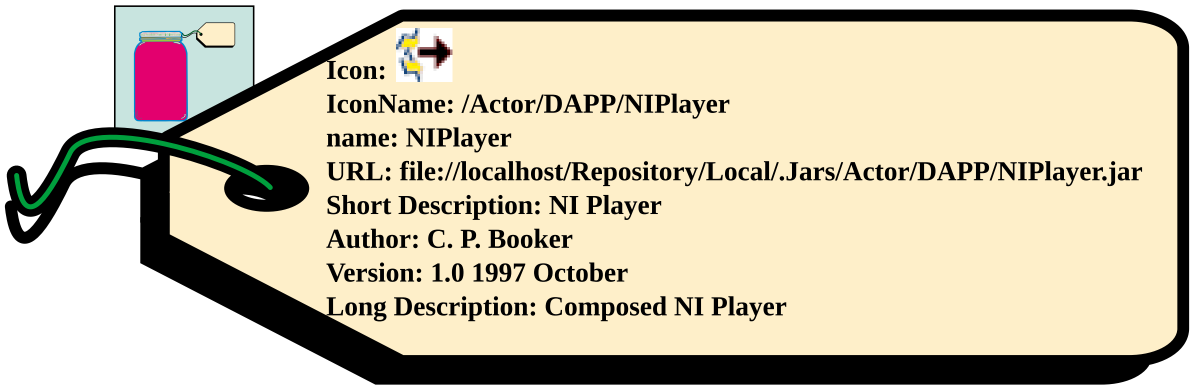

Icon:
IconName: /Actor/DAPP/NIPlayer
name: NIPlayer
URL: file://localhost/Repository/Local/.Jars/Actor/DAPP/NIPlayer.jar
Short Description: NI Player
Author: C. P. Booker
Version: 1.0 1997 October
Long Description: Composed NI Player

Figure 11. Tag Entries.

When the Composition Environment uses a Tag to create a Palette, it checks the local Repository for the existence of the Tag's icon according to the **icon name** specification. If the Tag's icon is not in the local Repository, it is written to the local Repository for current and future use.

At the moment the Composition Environment cannot create or edit Tags; there are external utilities to perform these functions.

Repository

The Repository design makes use of both Jars and Tags as described above. As illustrated in Figure 12 below, there are three main storage directories in the Repository. The **Local** directory is where all local Tags and Jars are stored; these are only visible to the Composition Environment running locally. Under Local, there are a variety of directories such as Actor, Simulation, and Component along with possibly others which roughly segregate the different functions of components that may be found under them. Tags are used in these directories to represent the available components. The .Jars directory may have a tree structure mirroring that of the rest of Local, and all the locally available Jars are stored here.

The tree structure under Export and WIP mirrors that of Local. The Export directory contains all the Tags that would be visible to connecting clients should the user choose to publish his Repository. These Tags might *point* to Jars under Export/.Jars or they may point to Jars under the Local directory depending on the owner's preference. For a dedicated Repository that has no anticipated local user , for example, A Server Repository, the Export directory is the only one that would be populated. The WIP directory is the *Work in Progress* directory and is available for saving or retrieving work directly from a Composition Environment's Workspace; it is only accessible from the locally executing Composition Environment.

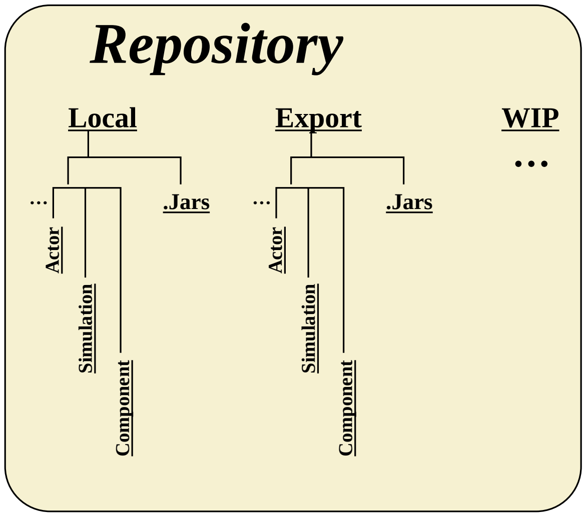


Figure 12. Structure of the Repository.
Note: Icon storage facility is not shown.

The Repository Server is implemented in Java using Remote Method Invocation for network communication. Currently this gives the Server a degree of security without employing passwords or encryption techniques. Future implementations of the Repository should incorporate more security measures.

IMPLEMENTATION OF VISUAL DESIGN ELEMENTS

The visual design elements of the implemented Composition Environment are shown and described here. These elements are designed to meet the goals brought out in a previous section. The Java Foundation Classes are used exclusively to implement these features; as a result, the ‘look and feel’ is similar on any platform. Following this section are two examples of how to use these design elements in the Composition Environment. The first example shows in detail how they are used to access the underlying structures to compose a new component from components residing in the Repository.

Main Menu

The Main Menu gives the user access to the Composition Environment’s primary tools. The first button, “New Workspace”, allows the user to summon a new Workspace in which to compose a new component; there is no restriction on the number of Workspaces that may be open. The second button, “New Palette”, allows the user to put up a Palette displaying the contents of a selected Repository; when selected, the user is queried to specify the desired Repository. The “Empty Container” button places an empty GenericContainer in the Clipboard. The user may then paste that Empty Container into any empty Workspace. Should the Clipboard become invisible, the “Show Clipboard” button will make it visible again. The “Preferences” button summons the Preferences Dialog. Finally, the Quit button, exits the Composition Environment.

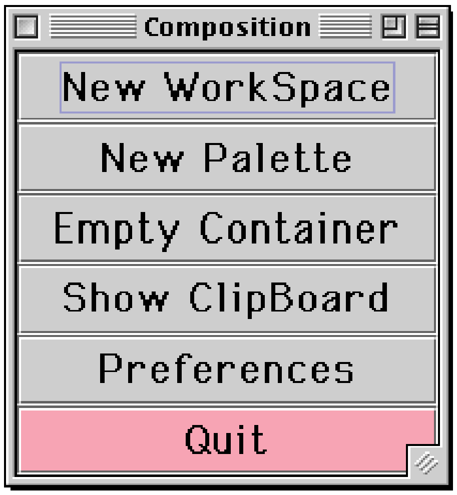


Figure 13. Main Menu

Preferences Dialog

The Preferences Dialog allows the user to define a set of preferences that survive between sessions; the preferences are stored in a file named, “Preferences”. The primary item of interest in the current incarnation of this dialog, is the “Default Repositories”; these are the Repositories to which the Composition Environment turns when it cannot find requested or required items locally. Further, when the user indicates a desire to access a Remote Repository, this is the default list that is presented. The column of buttons on the right are available to maintain the file system in the local Repository.

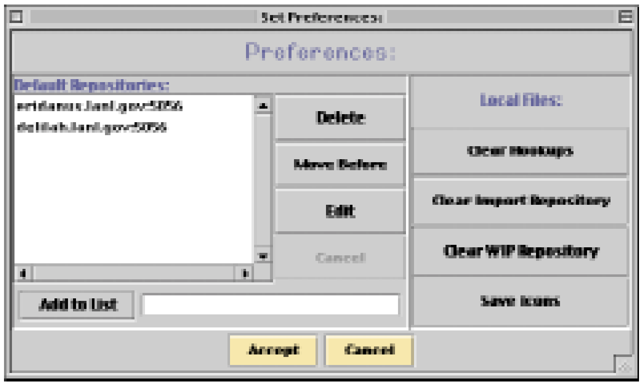


Figure 14. Preferences Menu.

Clipboard

The Clipboard is similar to a clipboard in any Operating System. A component's Tag that has been selected from a Palette is stored here *until* the component is pasted into another component or into a Workspace. The Clipboard is cleared after a component is used. Alternately, the user may clear the Clipboard at any time by using the "Clear" button a the bottom of the window.

An empty Clipboard is depicted here in Figure 15. When a component shows in the Clipboard, information from its Tag is displayed:

- The name of the Component is displayed across the top rather than "Empty" shown in the figure.
- The scrolling text area displays the long description of the component.
- The Author field displays the name of the Author.
- The Version field displays the version string associated with the component.

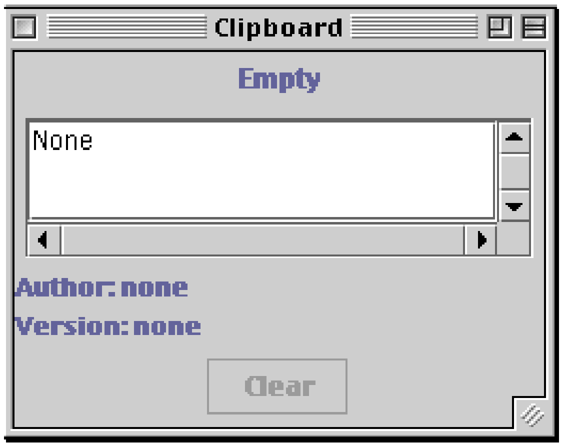


Figure 15. Clipboard

Note that when an object is selected from a Palette and *appears* in the Clipboard, the component has *not yet* been extracted from its Jar; in fact the Jar may not have been retrieved from the Repository. The Clipboard just displays information from the Tag that originated in the Palette.

WorkSpace

A Workspace is simply a visual representation of a composition area in which components may be composed. An empty Workspace will accept a component or empty container that is *dropped* on it, but it will not accept a Personality because a Personality is not a GenericContainer. Figure 16. shows three different aspects of a Workspace. The first is an empty Workspace ready to accept an empty container or a component. A component may be added to this Workspace via a Palette or from the Work in Progress (WIP) section of the Local Repository using the Workspace's Jar menu. The next Workspace, to the right, holds an Empty Container placed there from the Main Menu; in this configuration, only components or a Personality may be added to the container from a Palette. The last, bottom Workspace shown in Figure 16. depicts a Workspace that holds a component; while incomplete, the component has a Personality and thus may accept the addition of other components. Note that the component is marked as incomplete along the top of the component's display.

It should be noted that the Workspace controls the display of all components shown in it. Components have no facility to draw themselves in the Composition Environment; the drawing of a component is completely handled by the Composition Environment. As a result, a component developer need know nothing about Java's display classes to create a component. As the number of owned components exceeds the size of the display, a vertical scroll bar appears on the right hand side of the Workspace's primary component giving the user access to all owned components.

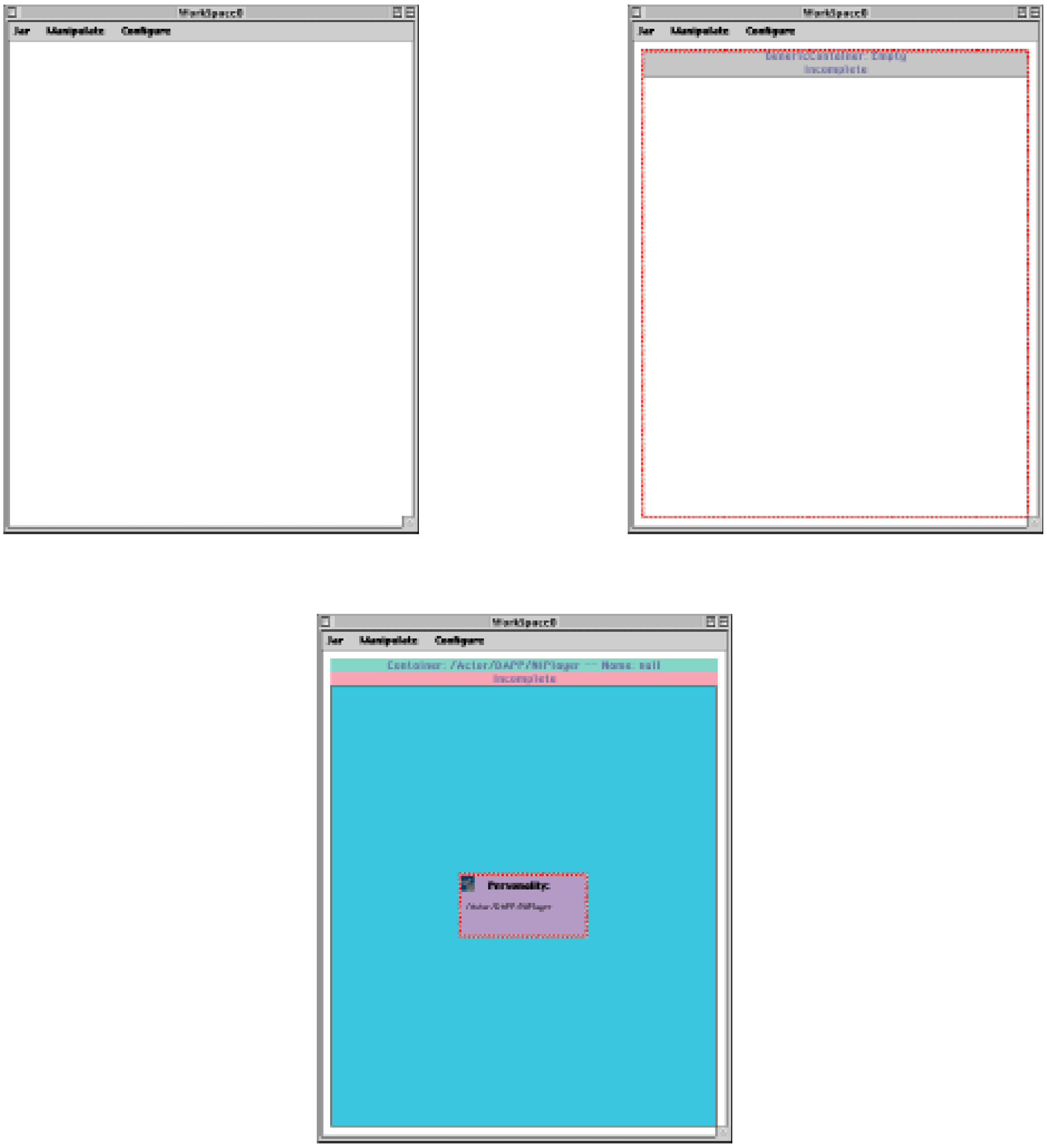


Figure 16. WorkSpaces.
The top, left WorkSpace is empty, the top, right contains an Empty GenericContainer. The bottom contains a partially composed component.

Navigating a Repository

Navigating a Repository is fairly easy within the Composition Environment. When the user selects, “New Palette” from the main menu, the dialog in Figure 17. appears allowing the user to select a Local or a Remote Repository:

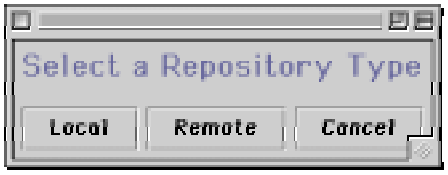
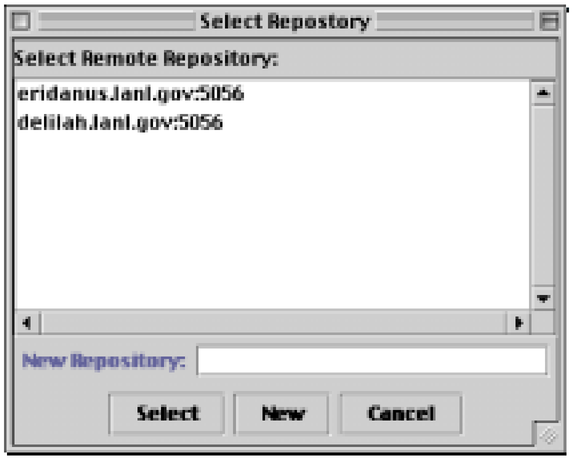


Figure 17. Selection of a Local *or* Remote Repository.

If the user chooses to access a Remote Repository, the dialog in Figure 18. appears with a default selection list from the Preferences dialog and an editable field, “New Repository” which allows the user to specify a Repository not displayed in the list. Note the Repository is specified by a machine name followed by a port number (typically 5056) with ‘:’ for a delimiter.



eridanus.lanl.gov:5056
delilah.lanl.gov:5056

Figure 18. Available Remote Repositories

A Local or Remote Repository is navigated with the dialog shown in Figure 19. In the Repository Navigator, the name of the Repository is shown at the top. The middle, scrolling field, “Available Categories”, lists the currently available categories which are actually directories in the Repository. To the right of this field is a column of buttons allowing the user to move up and down within the categories. The next scrolling field, “Current Category” shows the full name of the current category. If the button at the bottom, “Use Current Category”, is enabled, there are Tags available in the current category, and if this button is *clicked*, a Palette will be created from the Tags comprising the current category.

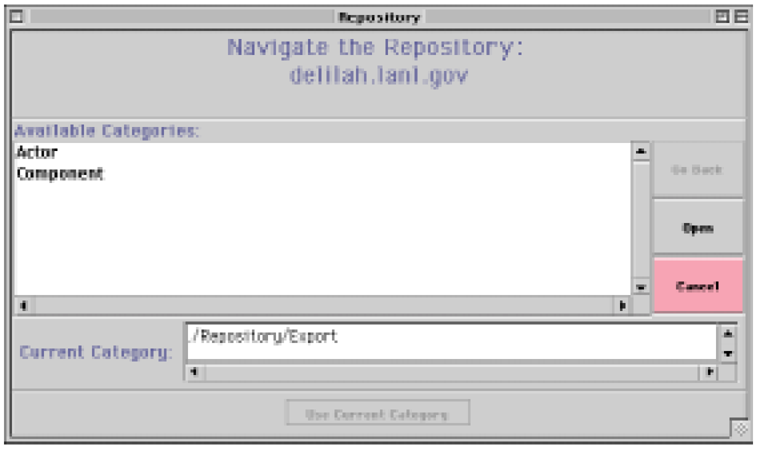


Figure 19. Repository Navigation.

Palette

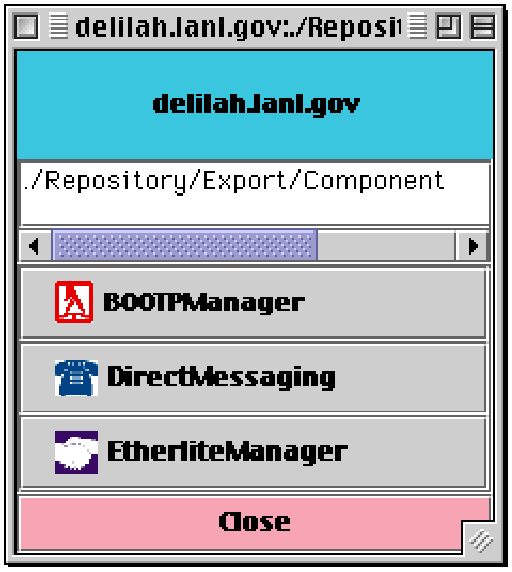


Figure 20. Example Component Palette.

A Palette depicts the components available in a selected category from a particular Repository. A Palette has the same general appearance whether it originates from a Remote or Local Repository. The top line in the Palette is the name of the Repository from which it originates. Below that, the scrolling field shows the category from which the components are drawn; it has a horizontal scroll bar because the category specification can become quite long. Under that is a series of buttons with icons. Each represents an available component in that category. The icon and name for each is taken from the corresponding component's Tag in the Repository. Should the number of components exceed a preset number, a vertical scroll bar appears to the right of the bank of component buttons allowing the user to scroll the list of components; with this feature, the Palette cannot become unmanageably tall for the user's screen. Finally, the bottom button on the Palette allows the user to close the Palette. When a Palette is closed, the Composition Environment examines all remaining links to the corresponding Repository and severs the connection to that Repository when there are no more Palettes or Browsers attached to it and when there are no more pending requests of that Repository.

EXAMPLES

In this section, two examples are presented showing how to use the Composition Environment. The first shows how to create a fairly simple component from atoms. The second, more abbreviated example shows how to create a very simple simulation using the just created component and another existing component. A following section shows how to attach a *Run Time Adapter* to the just composed simulation and execute it. All examples are taken from a simulation project which was being developed at the same time as the Composition Environment.

Construction of an Example Player

For this example, the Composition Environment is used to construct a simple Player, the NIPlayer, for a specific simulation. The NIPlayer consists of five components and an NIPlayer Personality. For this example, the desired Personality and components reside on a Remote Repository residing on a Unix machine named Delilah.

1. In the beginning:

When the Composition Environment starts-up, the user is confronted with just the Main Menu. Figure 21 shows the Composition Environment a short time after start-up; the user has already *clicked* the "New Workspace" button and arranged the windows:

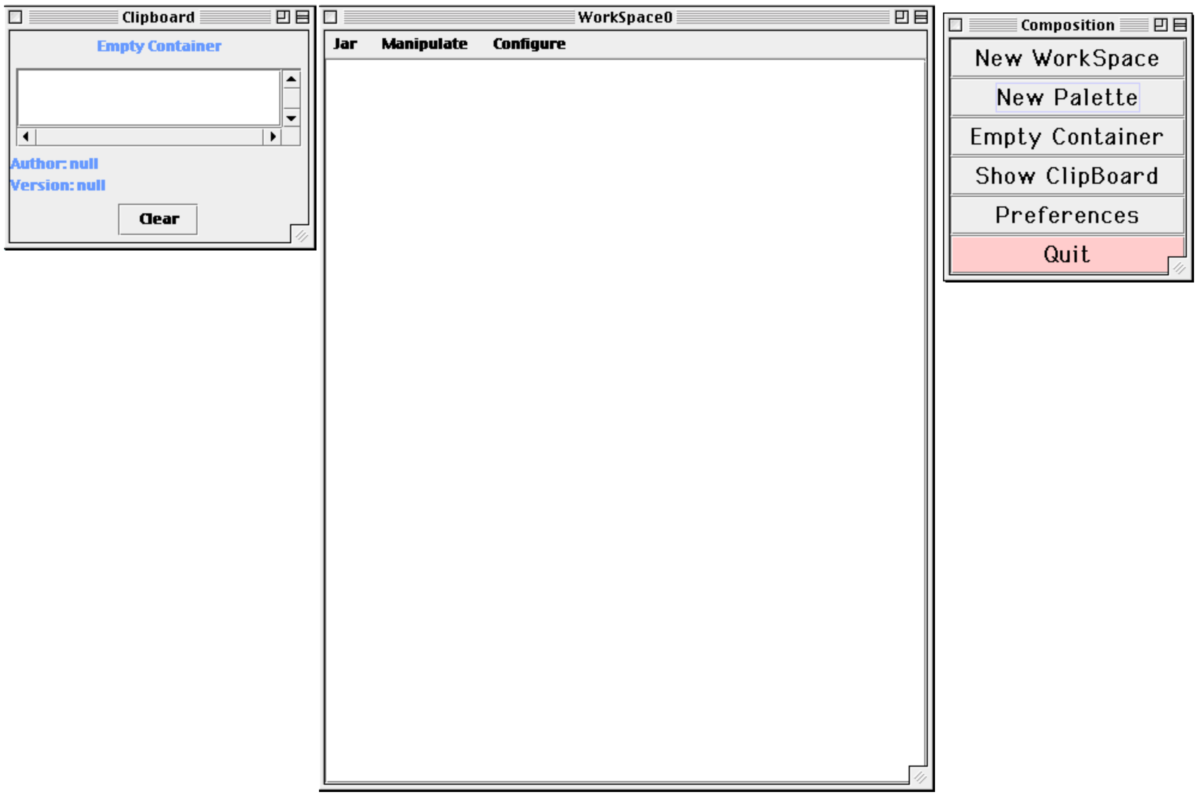


Figure 21. Ready to compose a Player.

At this point, the Workspace is ready to accept a component or an Empty Container to begin the composition process. The user may load the Workspace with a partially completed component via the Workspace's Jar menu which accesses the Local Repository's WIP section or by using a Palette attached to the Local or a Remote Repository. On the other hand, the user may load the Workspace with an Empty Container from the Main Menu.

2. Creating a Component from Scratch:

Figure 22 depicts the Composition Environment after the user has dropped an "Empty Container" from the Main Menu on the Workspace.

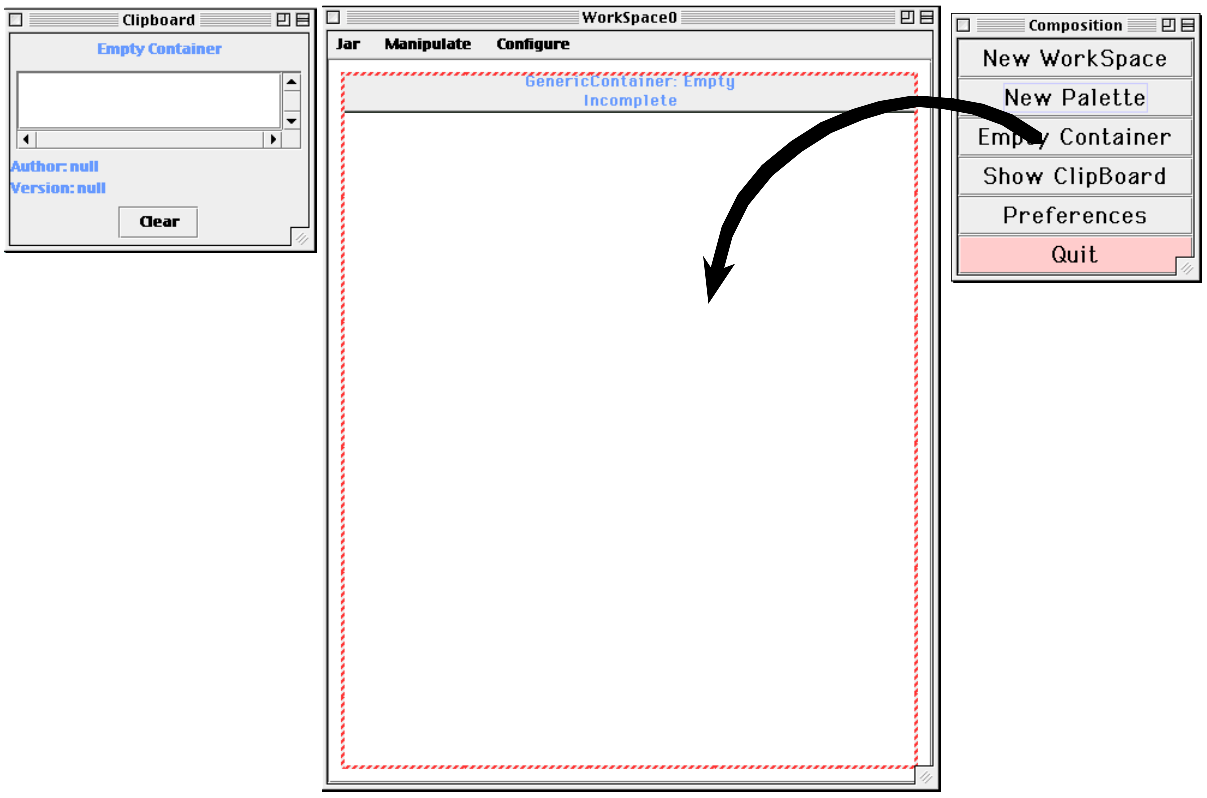


Figure 22. Instantiating an Empty GenericContainer in the Workspace.

Once the empty container is dropped on the Workspace it is instantiated. However, the Empty Container itself and any component does not possess the facility to display itself in the Composition Environment. Instead the Composition Environment itself *wraps* any given component, and that Wrapper is displayed in the Workspace. Because Wrappers are used to visually display a component, the component developer need not be concerned with the intricacies of how to display a component in the Composition Environment. The developer simply provides the required methods, typically implemented in the Personality interface, that provide general information on the component. The Composition Environment uses those methods and Reflection to create an appropriate, displayable Wrapper for the component.

3. Accessing the Repository:

Now that the Workspace contains an Empty Container, the user is ready to compose a *useful* component from components and a Personality residing in a Repository. Clicking the "New Palette" button in the Main Menu eventually leads to the "Select Repository" dialog from which the user may select a Remote Repository to access already built components. This is illustrated in Figure 23.

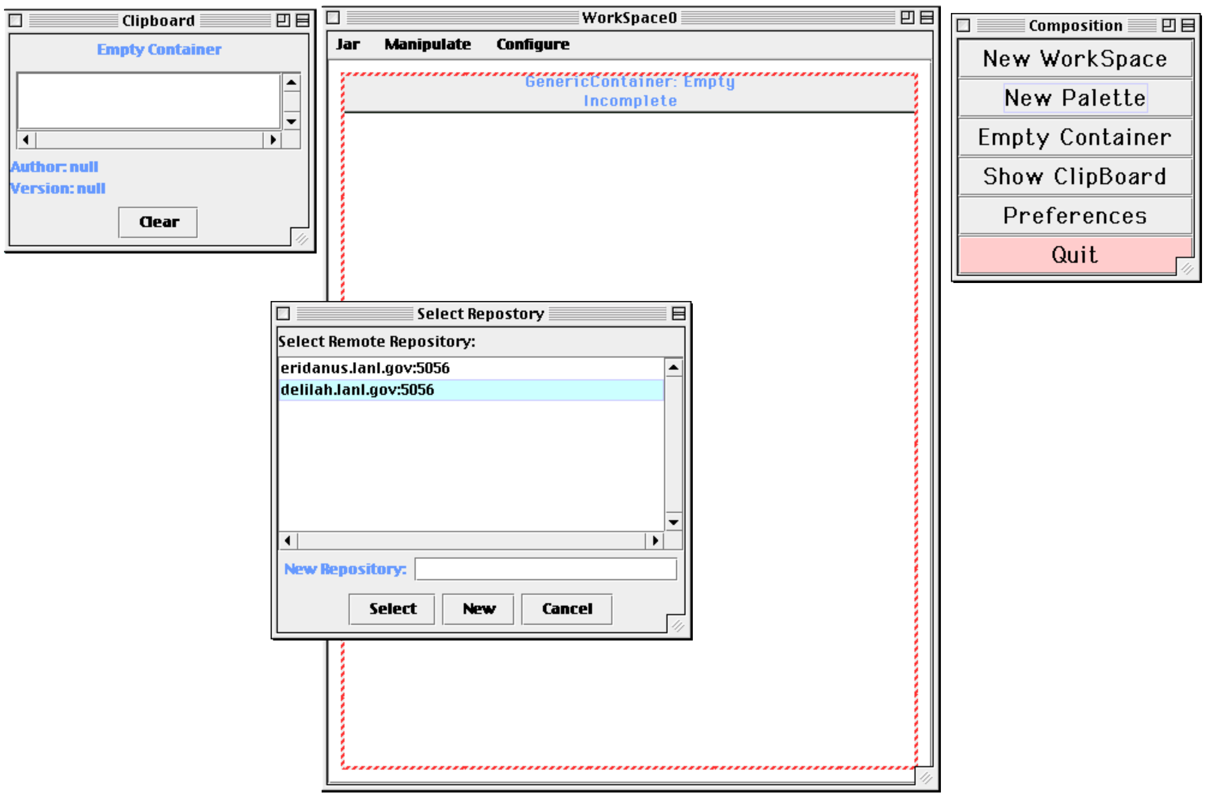


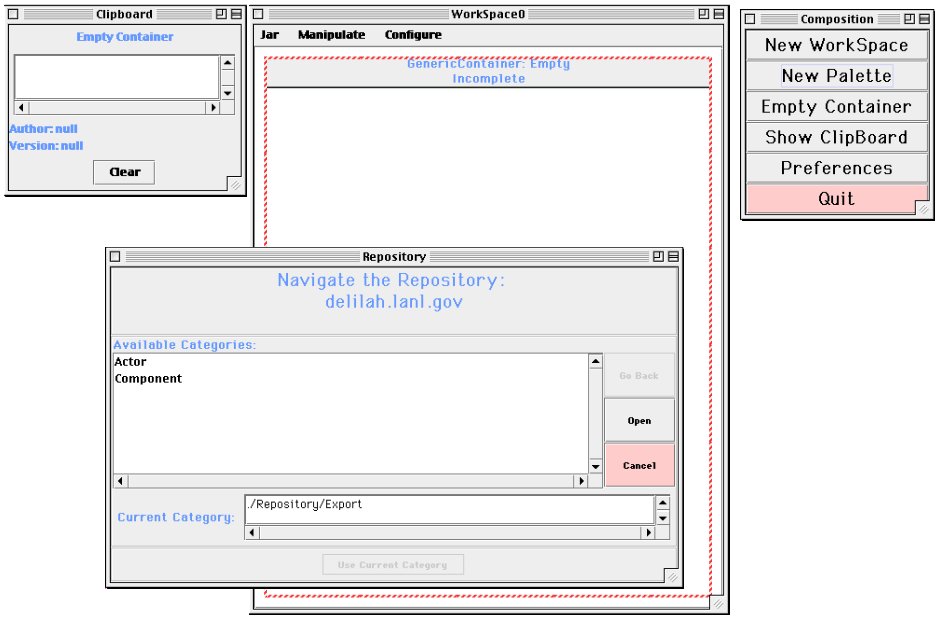
Figure 23. Selecting a remote Repository, delilah.lanl.gov .

The user's *favorite* Repositories, identified in the Preferences section, are available in the large scrolling pane. The user has the option of typing in a different Repository in the field below that. In this example, the user selects, delilah.lanl.gov:5056.

4. Navigating the Repository:

Delilah has an *active* Repository Server which connects to the client Composition Environment; In a proper production environment, there might be many such machines with Repository Servers running full-time. These machines might be in different parts of the country or even the world giving rise to a truly distributed Repository. When the server and client establish a connection, the client displays a Repository Browser as shown in

Figure 24. Now the user may select categories from within the Repository and summon Palettes to display the components available in a given category.



Author: null
Version: null

Navigate the Repository:
delilah.lanl.gov

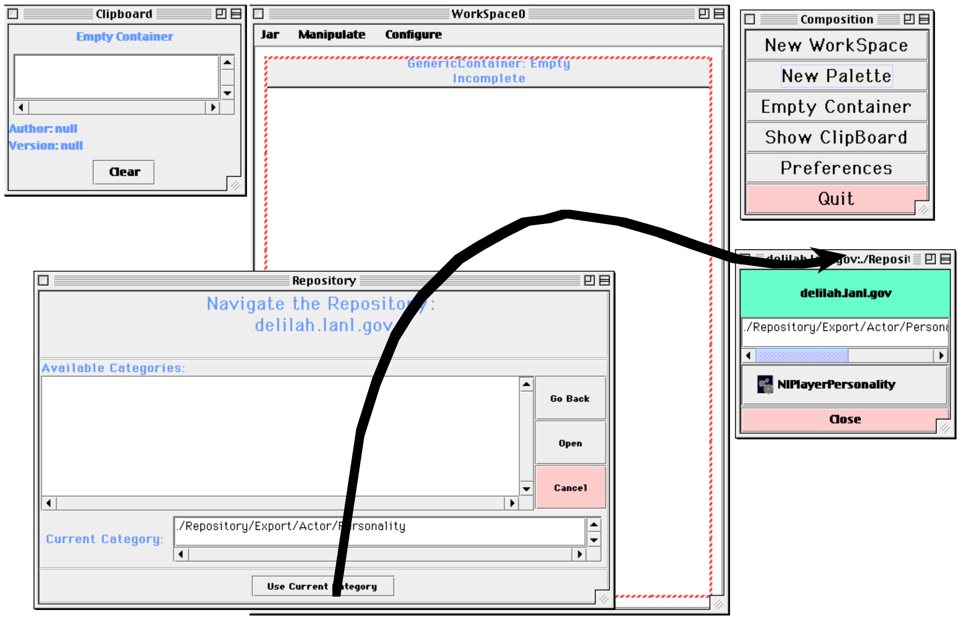
Available Categories:
Actor
Component

Current Category: ./Repository/Export

Figure 24. Navigating the remote Repository.

5. Displaying a category’s Palette:

Since the previous figure, the user has navigated into the /Actor/Personality category of the delilah.lanl.gov Repository. Once in this category, the Navigator’s “Use Current Category” button is enabled; compare the appearance of this button in figures 24 and 25.



Author: null
Version: null

Navigate the Repository:
delilah.lanl.gov

Available Categories:

Current Category: ./Repository/Export/Actor/Personality

./Repository/Export/Actor/Persona

Figure 25. Creation of a Palette from the Current Category in the remote Repository.

Clicking on the “Use Current Category” button gives rise to a series of transactions between the Repository Server and the Composition Environment client culminating in the creation of the Palette shown at the right of Figure 25. This Palette displays just 1 item, a Personality; this NIPlayerPersonalilty is suitable for a simulation Player.

6. Giving the Empty Container a Personality:

As shown in Figure 26, the user selects the NIPlayerPersonality from the Palette and *drops* it on the Empty Container in the Workspace. The dotted, black line indicates that once the Personality is selected, it appears in the Clipboard.

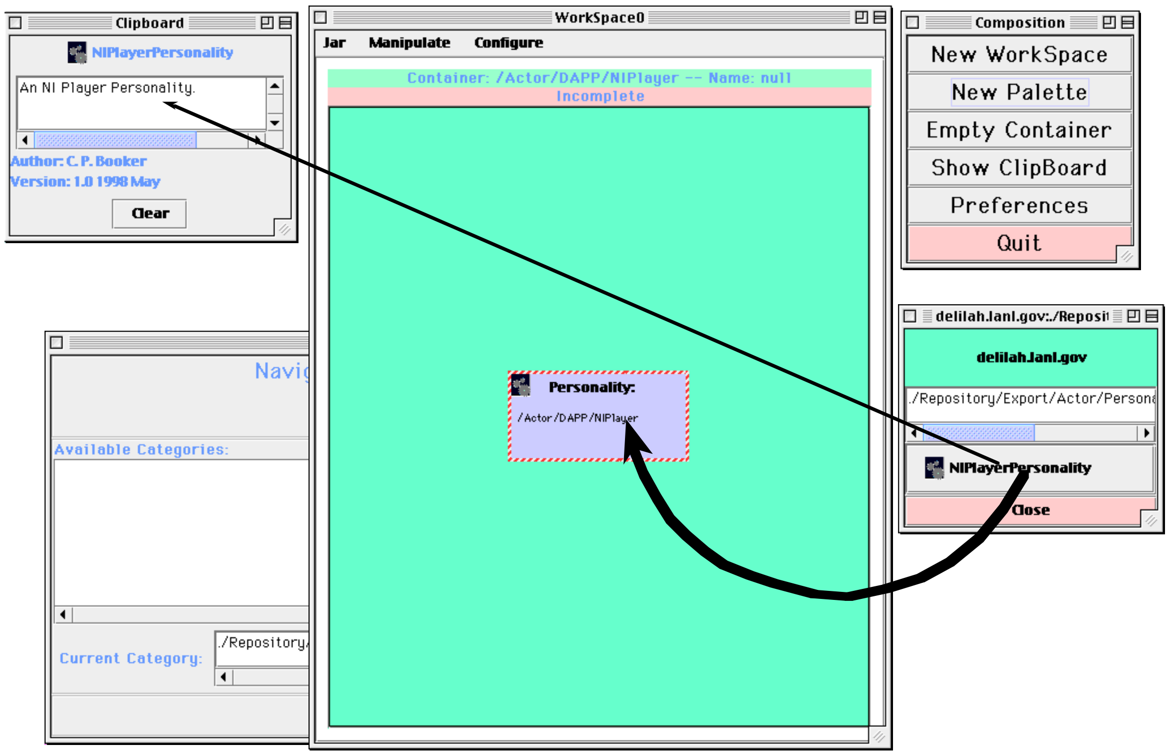


Figure 26. Dropping a component selected from the Palette on the empty GenericContainer.

Note that once the Workspace’s component has a Personality, the Wrapper representing it changes color and more information appears. The component takes on the *name* of its personality shown at the top. A second text line marks the component as *incomplete*. This is a dynamic feature; as the component is changed, its Personality is queried by the Composition Environment with respect to *completeness* of the Component. Now that the component being composed has a Personality, it is ready to consider other components as candidates for inclusion in it.

7. Completing the component:

Figure 27 combines several actions. The Repository Browser has been used a few times to acquire Palettes for two more categories; they are shown on the right. Components have been selected from these Palettes and *dropped* on the component in the Workspace; each time, the candidate component has been subjected to scrutiny by the NIPlayerPersonality before it is allowed to join with the component being composed. In each case the candidate component has been accepted. As each component is added, the Personality is queried by the Composition Environment as to *completeness* of the component; note that finally, the second line in the Workspace shows "Complete" indicating that the component is ready to use as judged by its Personality.

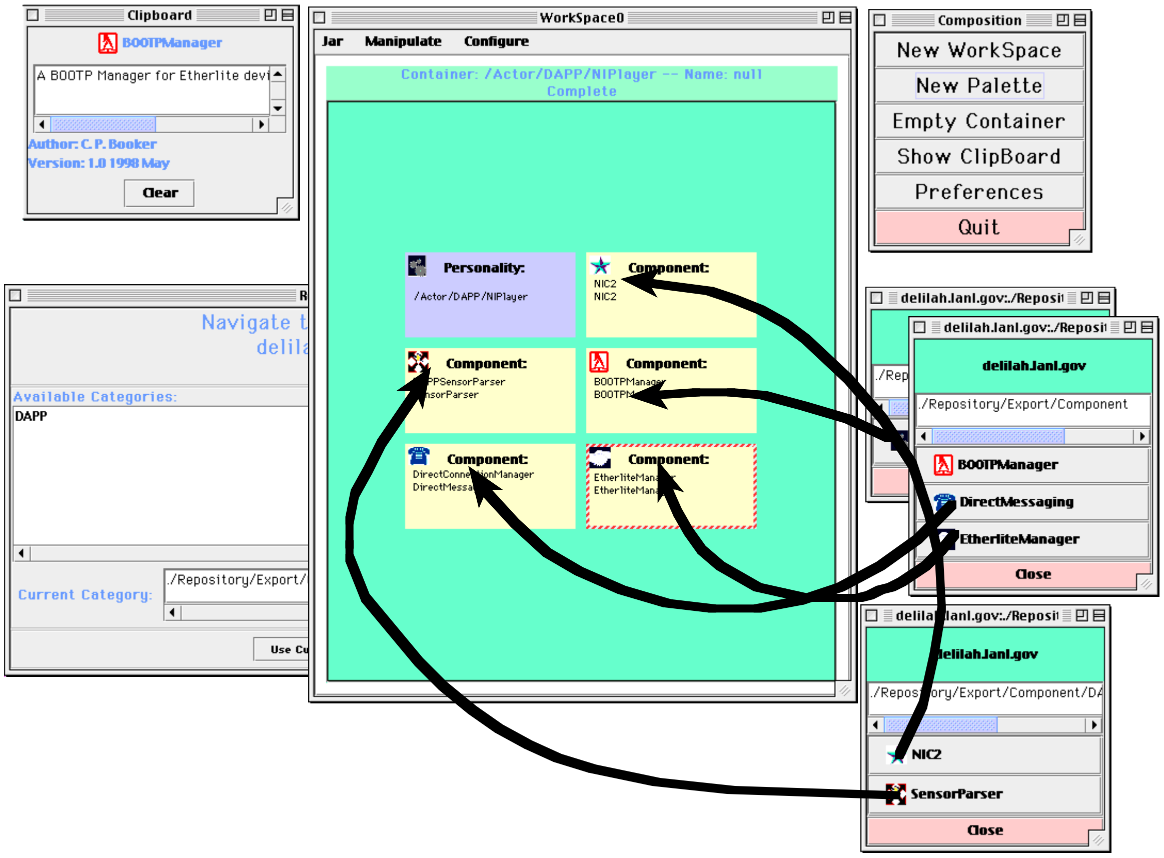
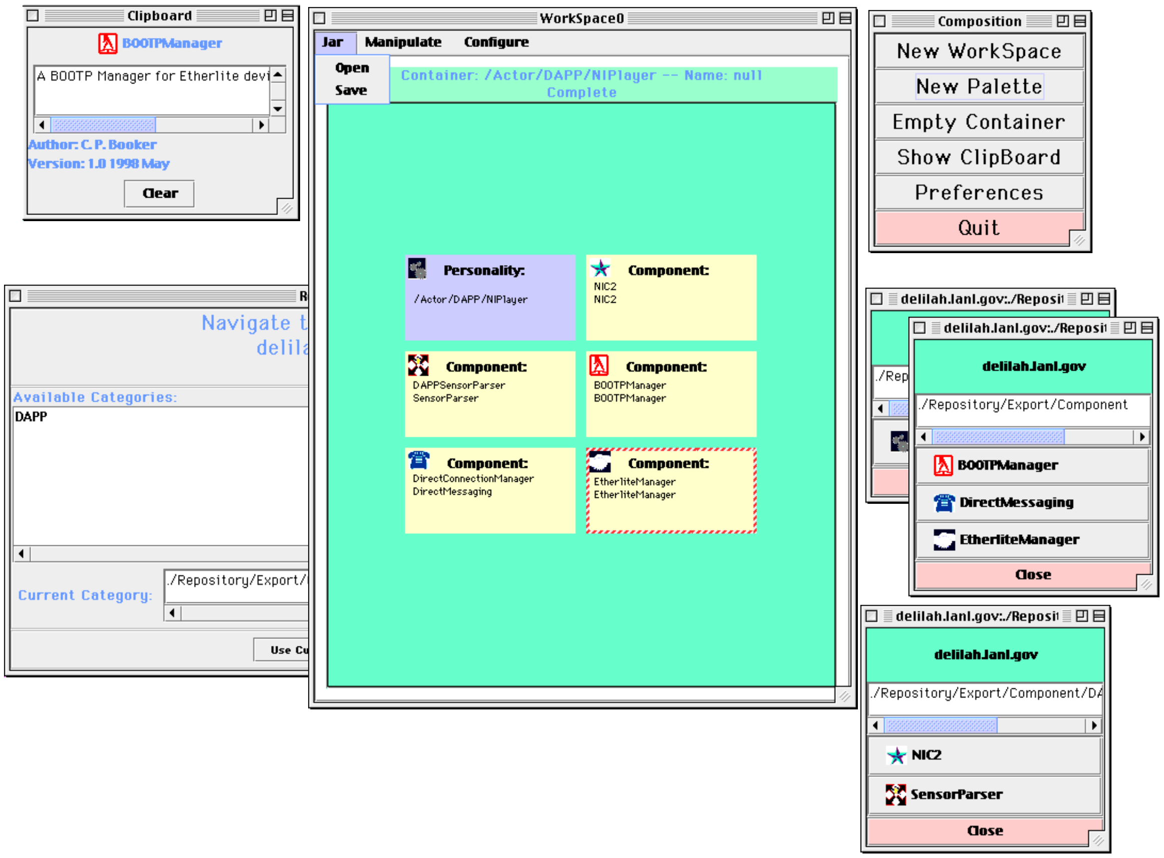


Figure 27. Dropping the rest of the required components on the evolving new component, NIPlayer.

8. Saving the new component:

Finally, as shown in Figure 28 the completed, composed component is saved via the Workspace File menu to the WIP section of the Local Repository.



A BOOTP Manager for Etherlite devi

Author: C. P. Booker

Version: 1.0 1998 May

Container: /Actor/DAPP/NIPlayer -- Name: null
Complete

/Actor/DAPP/NIPlayer

NIC2
NIC2

DAPPSensorParser
SensorParser

BOOTPManager
BOOTPManager

DirectConnectionManager
DirectMessaging

EtherliteManager
EtherliteManager

./Repository/Export/

./Repository/Export/Component

./Repository/Export/Component/DA

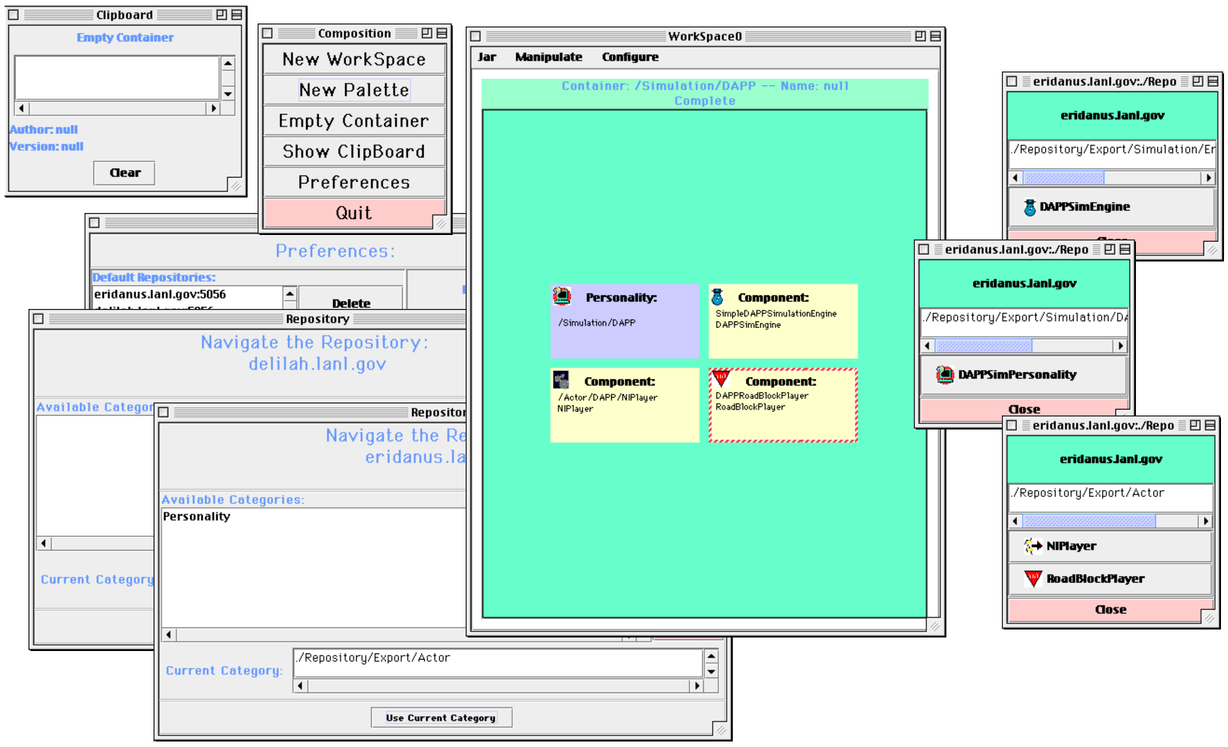
Figure 28. Saving the completed new component, NIPlayer.
Note the Workspace's Jar Menu.

At this time, components stored in the WIP directory can be manipulated with a set of external utilities to move the component's Jar into a more accessible area of the Repository such as the Local or Export directory. These utilities are used to create a Tag for the component, and the Tag is placed in the Repository so that the component may be accessed. In the future, these utility functions will be integrated into the Composition Environment.

Composition of a simple Simulation.

In the previous section, a Player is composed from preexisting components residing in a Remote Repository. It is stored in the WIP section of the Local Repository. Using tools external to the Composition Environment, that Jar is moved into a Remote Repository and given a corresponding Tag.

Figure 29 shows a *composite* screen shot of how the Composition Environment is used to compose a simulation using the just composed NIPlayer and an existing Player, RoadBlockPlayer.



Author: null
Version: null

Default Repositories:
eridanus.lanl.gov:5056

Navigate the Repository: delilah.lanl.gov

Navigate the Repository: eridanus.lanl.gov

Available Categories: Personality

Current Category: ./Repository/Export/Actor

Container: /Simulation/DAPP -- Name: null
Complete

Personality: /Simulation/DAPP

Component: SimpleDAPPSimulationEngine DAPPSimEngine

Component: /Actor/DAPP/NIPlayer NIPlayer

Component: DAPPRoadBlockPlayer RoadBlockPlayer

./Repository/Export/Simulation/En

./Repository/Export/Simulation/DA

./Repository/Export/Actor

Figure 29. Construction of a *simple* Simulation in the Composition Environment.

Here, <u>two</u> Remote Repositories are accessed simultaneously to build the simulation; there is no hard restriction on the number of Remote Repositories that may be open in a session. The simulation is then built from a single Personality and three components. Two of the components are the two aforementioned Players, and the remaining component is a simulation engine that links together the Players and puts them through the various initialization phases, communication hook-up phase, and executes the simulation. The use of the simulation engine with a Run Time Adapter to allow the user to *run* the simulation is delineated below.

RUN-TIME ADAPTERS

Once an executable component such as a Simulation or even a Player in a distributed simulation has been *composed*, the next step is to execute it. As illustrated in Figure 30, it is not important as to the nature of the executable component; it simply needs to be executable; it must have a run-time interface that can be made available to the Run-Time Adapter. The run time interface must have sufficient methods to control an execution; the runnable interface and associated Personality that has been in use as of the time of this writing is listed in Appendix G; the available methods are:

- **initialize** has a an int argument, phase; this allows a simulation designer to have a number of discrete initialization steps.

- **step** is not used yet. In the future, it may allow a time wise simulation to advance one time step. It may have little or no meaning in a discrete-events simulation.
- **run** is currently used to run the simulation rather than step.
- **pause** is not currently used. It may allow a simulation to pause.
- **stop** stops a simulation.
- **terminate** terminates a simulation. It embodies stop but is more final.

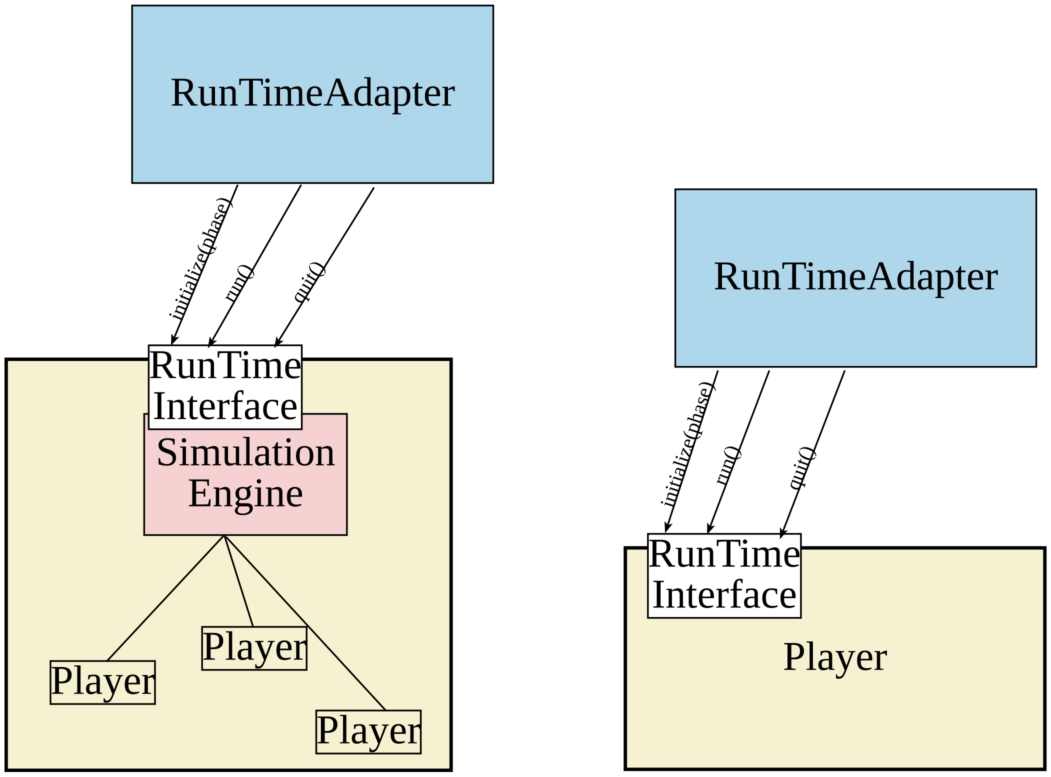


Figure 30. A RunTimeAdapter may command any *runnable* component such as a simulation (left) or an individual Player (right).

Run-Time Adapters are simple pieces of code that *attach* to an executable component and execute it. Figure 31 shows the steps in involved in attaching a Run-Time Adapter to a composed, runnable component; note that these steps are performed by the Run-Time Adapter itself.

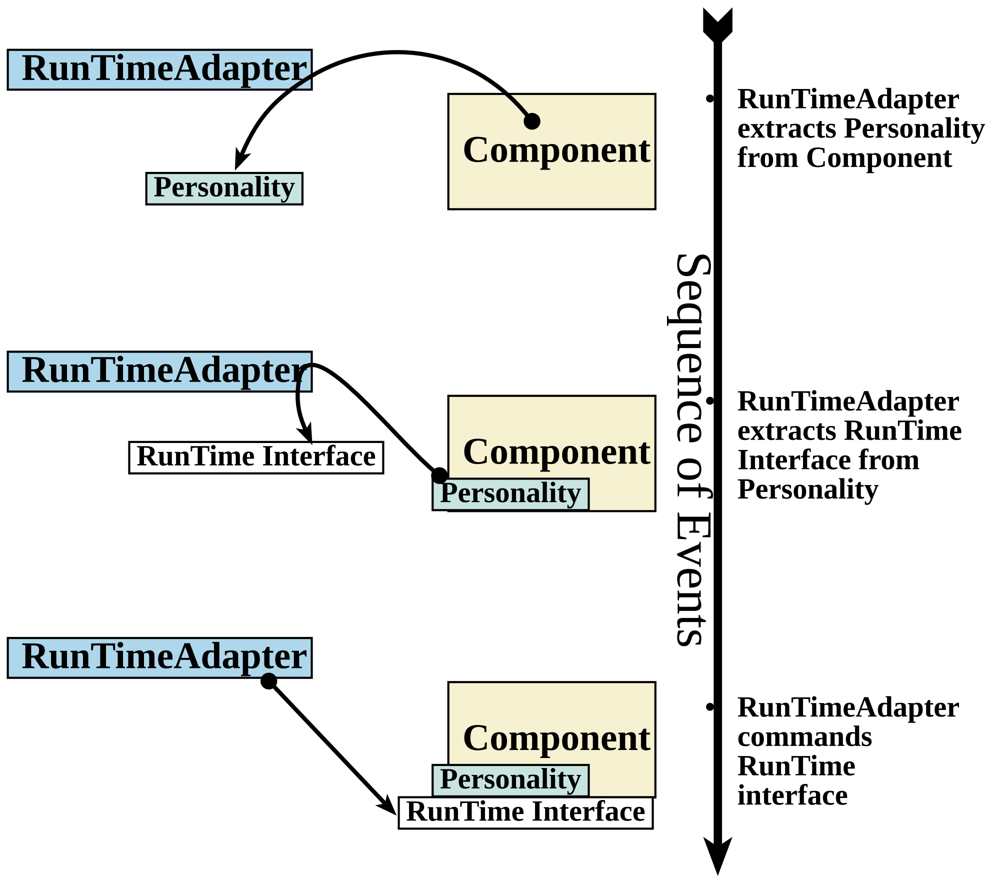
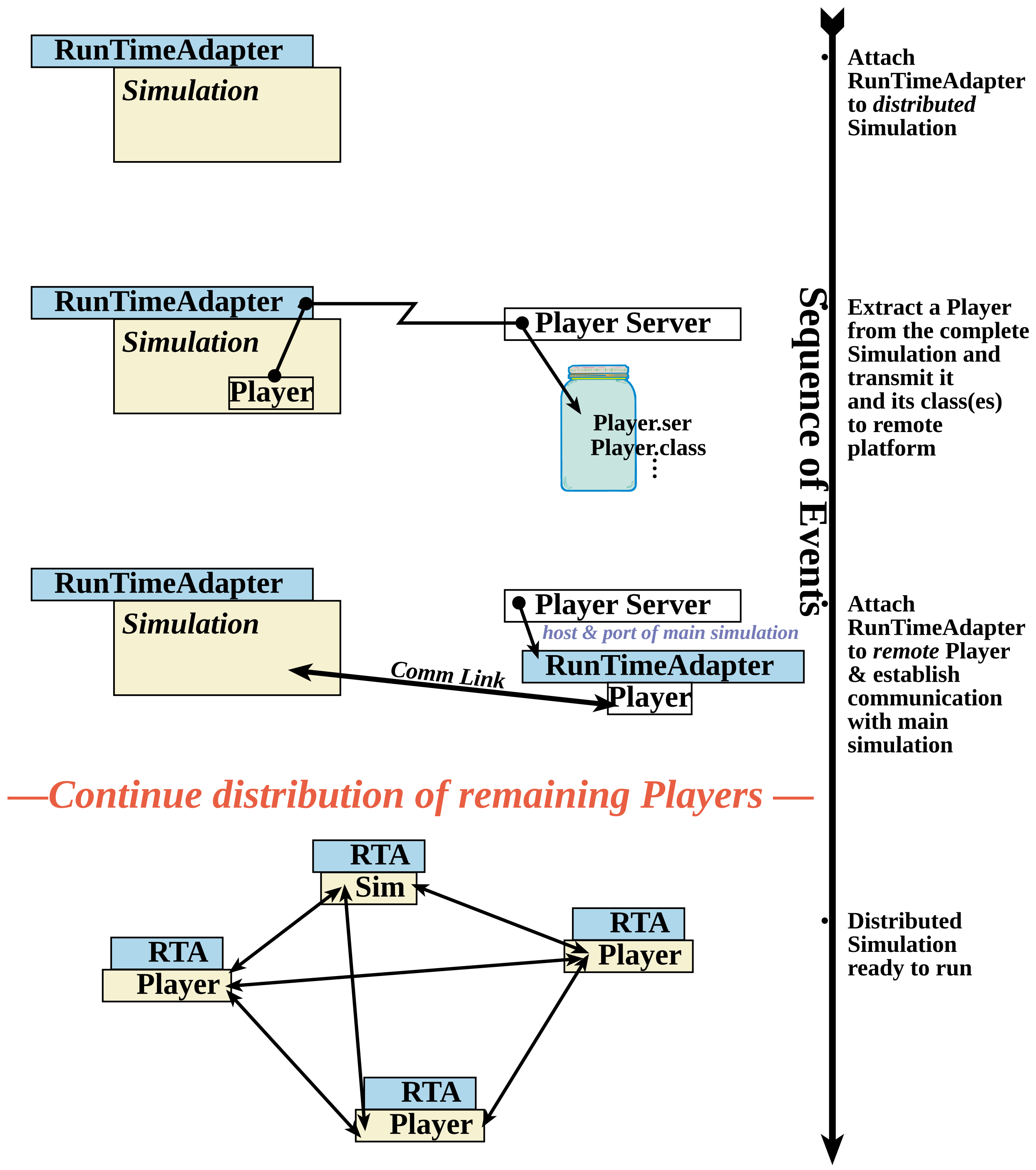


Figure 31. How a RunTimeAdapter attaches itself to a runnable component.

An example Run-Time Adapter that attaches to the simulation composed in the previous section is listed in Appendix H. Note that the user need only specify the name of the Jar in which the executable component resides and the name of the bean to instantiate in order to actually run the simulation. In the future, the requirement to specify the name of the bean could be dropped; it could be named in the Jar's manifest.

Run-Time Adapters are simple to design and implement on single platforms as illustrated by the code listed in Appendix H. A Run-Time Adapter designed to manage a *distributed* simulation is more interesting and is a topic of current study. Figure 32 illustrates the essential functions of such a Run-Time Adapter for a distributed simulation.



RunTimeAdapter
Simulation
Attach RunTimeAdapter to *distributed* Simulation
RunTimeAdapter
Simulation
Player
Player Server
Player.ser
Player.class
Sequence of Events
Extract a Player from the complete Simulation and transmit it and its class(es) to remote platform
RunTimeAdapter
Simulation
Player Server
host & port of main simulation
RunTimeAdapter
Player
Comm Link
Attach RunTimeAdapter to *remote* Player & establish communication with main simulation
—Continue distribution of remaining Players —
RTA
Sim
RTA
Player
RTA
Player
RTA
Player
Distributed Simulation ready to run

Figure 32. Example of how to distribute a Simulation created in the Composition Environment.

There are some interesting features in Figure 32 briefly addressed here:

- The first step, as usual, is to *attach* the Run-Time Adapter to the simulation. As mentioned before, this can be as simple as specifying the name of the Jar in which the simulation resides.
- The next step is to distribute the simulation among the various available platforms. The figure shows this process involving the Run-Time Adapter, but it is possible to design the simulation engine such that the primary Run-Time Adapter is *ignorant* of the distributed nature of the run.
- It is assumed that the remote platform has some means of receiving the executable component that is to run on it and save that component to a file. In the figure, some sort of Player Server is postulated to be running on the remote platform, and it is able to set-up a run locally.
- The Player Server attaches a local Run-Time Adapter to the Player that it has received and passes vital information on to it such as the host name and port number for the simulation engine.
- The remote Player then establishes a link to the simulation engine.
- This process continues for all the Players in a simulation.
- Finally, the distributed simulation can go through the initialization steps and then execute.

Clearly, Run-Time Adapters are straight forward to design and construct. in the future, they should be an integral part of the Composition Environment so that the user need only instruct the Composition Environment to execute the specified component, and the Composition Environment takes care of all the necessary steps.

FUTURE WORK

Much has been accomplished in this study with good success. However, some of the necessary elements, identified from the beginning, have been left undone because of time and monetary constraints. Other important features have been identified as the study proceeded and are now deemed to be highly desirable for eventual development. The important features proposed for continued support are briefly discussed here:

Configuration Editors

As indicated earlier, Configuration Editors were always anticipated to be an integral part of the Composition Environment. After all, putting together a collection of components to create a new, more useful component is only half the task of composition. The new component must be *localized* and otherwise configured to be of any use in a simulation. For example, suppose the user chooses to compose a tank for use in a simulation. Once the tank has been composed *to be* a particular type of tank, the user must then *localize* the tank — give it an initial location for it to participate in the simulation as illustrated in Figure 33.

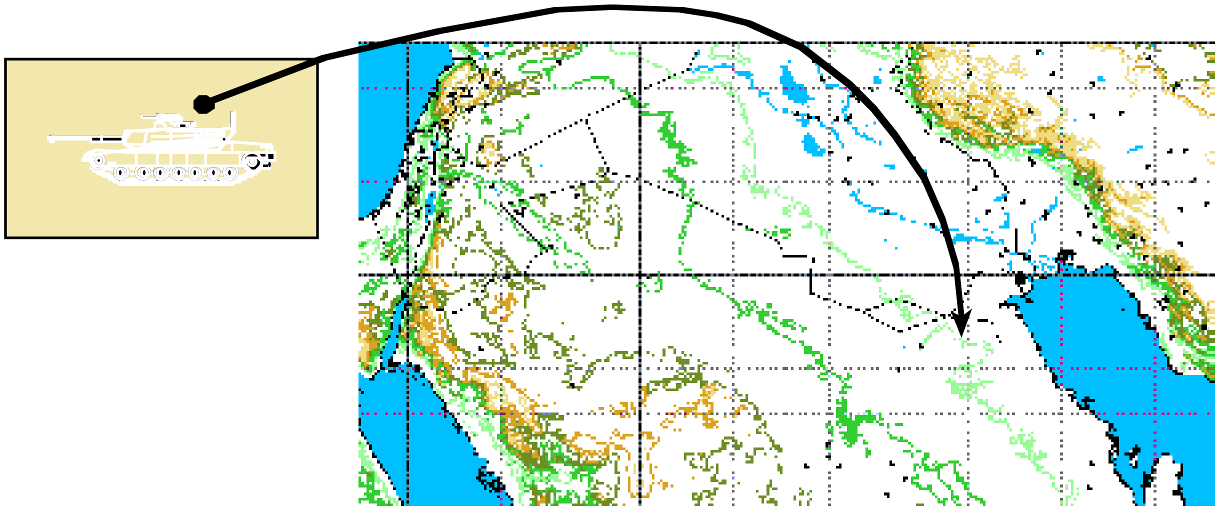


Figure 33. Localization of a component.

Other configurable parameters might include the fuel load and the amount and type of ammunition on board. All of these tasks would be accomplished by Configuration Editors *in* the Composition Environment as indicated in Figure 34.

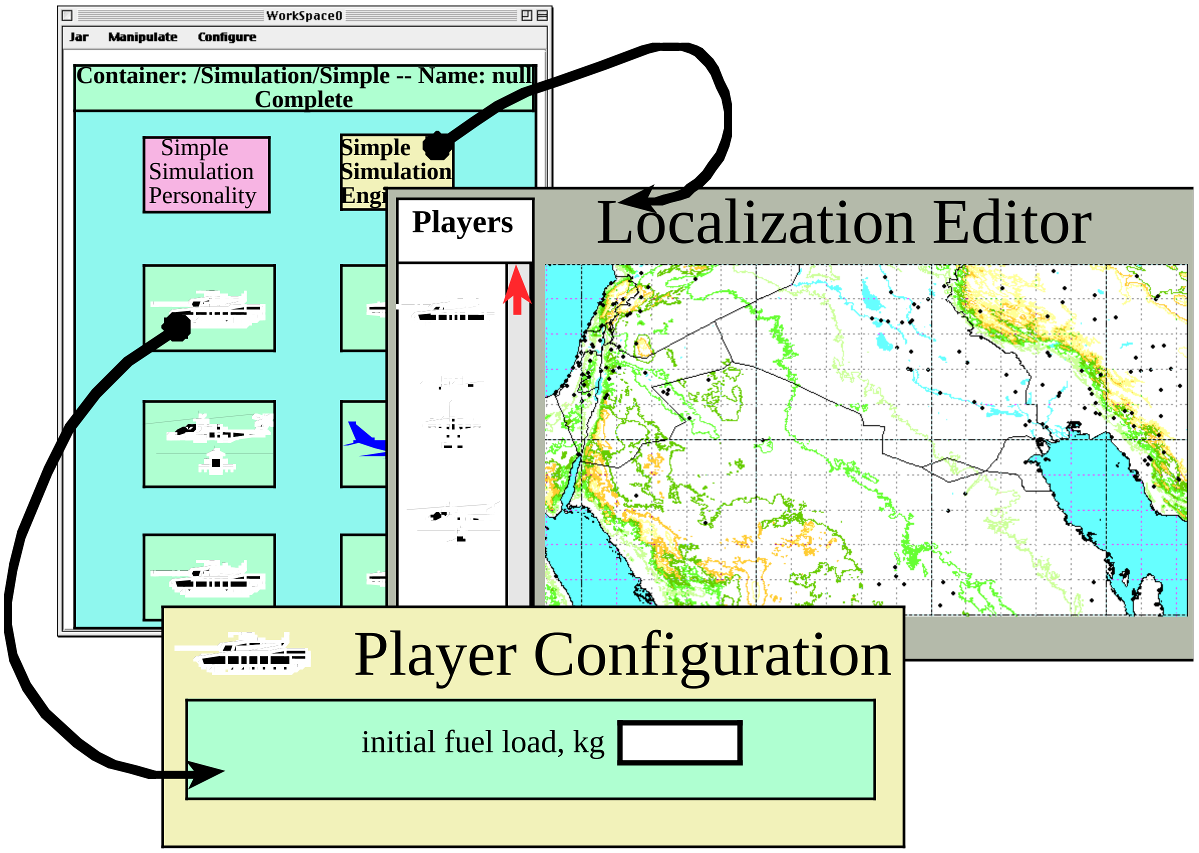


Figure 34. Conceptional use of Configuration Editors.

For those familiar with Sun’s BeanBox, the Configuration Editors resemble the Custom Editors available in there. In fact the JavaBean’s specification provides for Custom Editors. However, as with icons, this built-in structure was deemed to be insufficient for the Composition Environment.

Configuration Editors, like icons, are envisioned as being residents of the Repository. Because they are not integral to the component, Configuration Editors can evolve independently of components; advances in Configuration Editors need not impact components at all. The designer of a component need specify an appropriate *type* of Configuration Editor for configuration in only a generalized manner; as with icons, it is specified by a descriptive String. The Composition Environment can then offer the user a list of candidate Configuration Editors that are appropriate for the selected component. This also means that a component designer need not be familiar with the exact Configuration Editors available he need not be concerned about specifying an outdated editor. By specifying an editor type, the Composition Environment may consult a dictionary and provide the user with a selection of applicable editors. Further, the Composition Environment can examine the component via Reflection and query to determine what parts of an editor are applicable. We are also examining generic ways of linking a given Configuration Editor to a component so that a larger set of Configuration Editors is available for a given component; it is expected that the BeanBox method of compiling specialized adapters on-the-fly, will be used.

We are also examining the use of Configuration Editors outside of the Composition Environment. For example, it would be useful to have a set of such Editors *or viewers* available during the execution of some simulation and certainly to assist in post-processing simulation data.

Editing Components

At this writing, the Composition Environment *only* builds components from other components. However, as the Composition Environment comes into common use, it is anticipated that the Repository will be populated with composite components that could be used in other than their intended application *if only they could be slightly changed.* For example, an Aircraft Player intended for use in a monolithic (single platform) simulation *could* be used in a distributed simulation if only its communication component were *replaced* by a new communication component suitable for distributed communication.

This is not a demanding modification of the Composition Environment; it is relatively straight forward to implement. It is a more *interesting* change for the Personality paradigm. The Personality must be able to *disconnect* the removed component from the other constituents of the composed component and introduce the new one. We are confident that once this process has been explored, it can be mostly embodied in a parent Personality class and will be relatively simple for the end developer to implement in his own Personality.

Consider an actual case. While the SAMSON and JWARS projects were under development, it was deemed desirable to employ some of the same Players in both simulations. While SAMSON is a real-time, distributed simulation and JWARS is a discrete-events, monolithic simulation, the overwhelming bulk of such a Player’s coding is identical; the main difference is the way that a SAMSON Player references and communicates with another Player compared to that of JWARS. In short, the difference between a Player in one simulation compared to a mostly identical Player in the other simulation, is the communication part. To use a Player from one simulation in the other, a developer was forced to edit the relevant sections of code, recompile and relink; if the Player changed, the developer was forced to repeat the editing, compile, and link. Clearly, if the Player-to-Player communication part was isolated in a reusable component, a Player could be transplanted from one simulation to another simply by *replacing* that single communication component, and that task should be an especially simple and quick one in the Composition Environment.

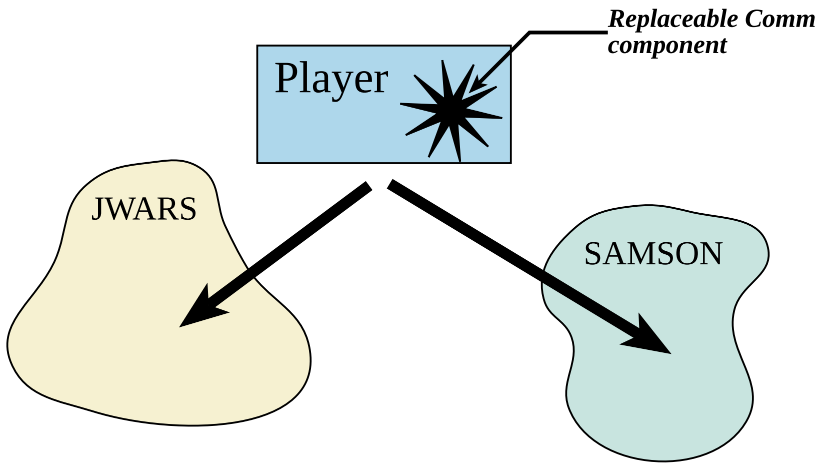


Figure 35. A Component (Player in this case) with a *replaceable* Comm component may be used in a variety of simulations with a simple change.

Trainable Personality

It is apparent from the Congruous Component discussion that a considerable amount of design work can be invested in the implementation of a specific Personality. A component's Personality is responsible for accepting and rejecting candidate, constituent components and writing the code to perform this task can be a burden on the developer. To alleviate this encumbrance, it has been suggested that a *trainable* Personality could be developed. To train such a Personality, it could be instantiated in a Workspace and *acceptable* components could be *dropped* on it; as each component is dropped on it, the Personality adapts to expect and allow only certain types of components. Further Configuration Editors can be used on such a trainable Personality to fine tune it. This promises to be an interesting and fruitful area of research.

Repository Management

In the examples, it was mentioned that the Player composed in the Composition Environment was moved, via utilities, into the Repository and *then* used to compose a Simulation. These utilities manipulate Jars and create Tags from those Jars with user assistance. With small effort these utilities can be integrated into the Composition Environment so that when a component is composed in one Workspace it becomes available almost immediately from the Repository for use with other components. This functionality can also extend to the user's Export Repository enabling a user to *publish* new or revised components quickly.

Legacy Code

It is assumed that there is a considerable store of legacy code that users might wish to continue to use in new, composed simulations. This is particularly true because Java is a relatively *new* language, and it is anticipated that few models have been implemented in Java as of this writing. Because Java offers so many advantages and the Composition Environment can greatly improve the ability to create and warehouse simulation components and simulations themselves, it is desirable, in the long run, to convert existing models to Java and thus take full advantage of the Composition Environment's power. However, in the short term, it is possible to make use of legacy code in the Composition Environment.

Java has a well defined interface to native code on any platform; as a result, a Java head-end can be provided to *wrap* any native code, and that head-end would be the same on any platform. Native code could be provided for one *or more* platforms and included in the component's Jar. The

META-INF/COMPOSITION directory could host a new manifest file dealing with native code issues; it could indicate what native code is in the Jar and the type of platform on which it will run. During the composition phase, this information from the Jar can be communicated to the component's Personality and that could be used in composing and configuring the component or enclosing components. In preparing for execution, the Personality could provide information on suitable platforms and guard against the choice of a platform for execution contrary to that dictated by the native code.

Expanded Manifest Information

Currently, a Composition Environment Jar contains simple Composition information that specifies any Jars that must be loaded before the current one. Just above, it has been proposed to expand this information to include information on native code required for the component. To facilitate the design of distributed simulations, it is proposed that the Composition Manifest section of a Jar also include a mapping of component to class files. If this information is present, then when a component is transferred to a remote platform for execution, only those class byte-codes required for that component need be transmitted with the Bean.

CONCLUSIONS

In this study, we have successfully implemented a prototype Composition Environment with a distributed Repository. We have demonstrated the feasibility of a design environment where the components themselves can participate in and even mediate in the composition of a greater, composed component. It is clear that such expertise incorporated in the component give rise to an environment where the user, not being a domain expert in the models he wishes to use, can be guaranteed that the components he employs to construct a new, complete composite component *will* function together and properly execute. Unfortunately, this is not to say that the composed component will be the best component for the job or even yield the proper solution. Yet, sufficient guidance *could* be embedded in the components so that the inexpert user *could* in fact construct a component that is not only guaranteed to perform, but carries with it great assurance that it will meet the user's needs.

Further, it has become clear that the requirements necessary to build such congruous components can largely be built into the very parent classes necessary to participate in the Composition Environment; as a result, a domain expert can primarily concentrate on the model design and how it might interact with other models rather than being concerned with extraneous coding required to fit into a particular simulation. Beyond this, we have shown with the above examples, that it is credible to maintain a distributed Repository of components that are available to build new, composite components that, in short order, can be used to solve a particular problem or answer a specific question.

Finally, the Composition Environment, Repository, and Run time environment developed and demonstrated here meet the requirements set forth at the beginning of this paper. All of them run on any machine with a recent version of the Java Virtual Machine (JVM) installed on it, and JVMs are available, free of charge, for most computers from PC to Macintosh to Unix. With the features described here *and* with the proposed enhancements, we can take a sample of our simulation and domain expertise anywhere in the world and provide a comprehensive *live* demonstration of our work. Moreover, we can begin to compile a Repository of components that embody a variety of physical models, and as demands are placed on us for new simulations or simply for answers to new questions, we can use the components in the Repository to quickly and accurately respond.

Recommendations

In previous sections, several areas were highlighted as worthy of continued development. Indeed, some were identified as vital to make a viable Composition Manager. The suggested order of new development is:

- Work on Configuration Editors should be given top priority because the Composition Environment is pretty much crippled without them.
- A collection of generalized Configuration Editors that may be used by a variety of components across simulations should quickly follow.
- The Repository should be populated with useful, *interesting* components from existing simulations which will provide the basis of future simulations. These include.
  - SAMSON (distributed)
  - JWARS (discrete events)
  - any current project that can be refashioned to fit into the Composition Environment.

APPENDICES

A. GenericContainer.java listing.

```
package lanl.tsa3.simulation.infrastructure;

import java.awt.*;
import java.awt.event.*;
import java.beans.BeanInfo;
import java.io.*;
import java.util.Enumeration;

import com.objectspace.jgl.SList;

/**
 * GenericContainer is a Container that may be manipulated in the Composition
 * Environment. A GenericContainer represents a Reusable Component (or in Java
 * parlance, bean). A GenericContainer will not accept 'ownership' of other
 * GenericContainers until it has been given a Personality.
 * @see lanl.tsa3.simulation.composition.Personality
 *
 * @version 1.0 1997 December
 * @author C. P. Booker
 */

public class GenericContainer implements Serializable
 {
  protected Personality personality = null;
  protected SList       beans       = null;

  /**
   * Construct an empty GenericContainer.
   */

  public GenericContainer() { }

  /**
   * Attempt to add a bean.
   * @param bean The object to be added. It may be either a Personality or a
   * GenericContainer.
   * @return true if the bean was successfully added.
   */
  public boolean addBean( Object bean )
   {
    if ( (bean instanceof Personality) && !(bean instanceof GenericContainer) )
     {
      if ( personality != null ) return false;

      personality = (Personality) bean;
      return true;
     }

    if ( personality == null ) return false;
    if ( bean instanceof GenericContainer )
     {
      GenericContainer item = (GenericContainer) bean;
      if ( !(personality.addGenericContainer( item )) ) return false;

      if ( beans == null ) beans = new SList();
      beans.add( item );

      return true;
     }

    return false;
   }

  /**
   * Attempt to remove a bean from the GenericContainer.
   * @param bean The GenericContainer to be removed.
   * @return true if the removal was successful.
   */
  public boolean removeBean( Object bean )
   {
    if ( personality == null ) return false;

    if ( !( bean instanceof GenericContainer ) ) return false;

    if ( !( personality.removeGenericContainer( (GenericContainer) bean ) ) ) return false;

    if ( beans == null ) return false;

    beans.remove( bean );

    return true;
   }

  /**
   * Get a BeanInfo suitable for this GenericContainer.
   * @return A suitable BeanInfo.
   */
  public BeanInfo getBeanInfo()
   {
    if ( personality == null ) return null;

    return personality.getBeanInfo();
   }

  /**
   * Get the Personality for this GenericContainer.
   * @return This GenericContainer's Personality.
   */
  public Personality getPersonality() { return personality; }

  /**
   * Get an array containing the contents of this GenericContainer.
```

```
   * @return an array of GenericContainers which are contained in this GenericContainer.
   */
  public GenericContainer[] getBeans()
   {
    if ( beans == null ) return null;

    GenericContainer inventory[] = new GenericContainer[ beans.size() ];
    int              i = 0;
    Enumeration slist = beans.elements();
    while( slist.hasMoreElements() )
     {
      GenericContainer item = (GenericContainer) slist.nextElement();

      inventory[i++] = item;
     }

    return inventory;
   }

  /**
   * Make an in-depth copy of this bean. Note that references to objects outside
   * of the GenericContainer are not copied.
   * @return An in-depth copy of this bean.
   */
  public GenericContainer duplicate()
   {
    GenericContainer myClone = new GenericContainer();

    if ( personality == null ) return myClone;

    myClone.addBean( personality.duplicatePersonality() );

    Enumeration inventory = beans.elements();
    while( inventory.hasMoreElements() )
     {
      GenericContainer bean    = (GenericContainer) inventory.nextElement();
      GenericContainer newBean = bean.duplicate();

      if ( !( myClone.addBean( bean ) ) ) continue; // Should throw exception!
     }

    return myClone;
   }
 }
```

B. Personality.java listing.

```
package lanl.tsa3.simulation.infrastructure;

import java.beans.BeanInfo;

/**
 * The Personality Interface is implemented by an object that
 * creates a personality for an GenericContainer. Note that the
 * method names here are designed not to interfere with methods
 * of GenericContainer; this is to accommodate Atoms (GenericContainers
 * that cannot be broken down into constituent GenericContainers) which
 * have the option of descending from GenericContainer and implementing the
 * Personality Interface.
 * @see lanl.tsa3.simulation.composition.GenericContainer
 *
 * @version 1.0 1997 December
 * @author C. P. Booker
 */

public interface Personality
 {
  /**
   * queries whether or not the Personality will veto the
   * candidate GenericContainer.
   * @param bean A GenericContainer that could be added to the Personality's
   * GenericContainer.
   * @return true if the Personality accepts the candidate bean.
   */
  public boolean inspectGenericContainer ( GenericContainer bean );

  /**
   * constitues an attempt to add a GenericContainer to another GenericContainer.
   * The Personality may veto the add attempt by returning false.
   * @param bean The candidate GenericContainer to be added to the Personality's
   * GenericContainer.
   * @return true if the bean is acceptable for addition.
   */
  public boolean addGenericContainer     ( GenericContainer bean );

  /**
   * attempts to remove a given bean from those known to the Personality.
   * @param bean The GenericContainer to be removed.
   * @return true if the Personality recognizes the bean and if it was successfully removed.
   */
  public boolean removeGenericContainer  ( GenericContainer bean );

  /**
   * returns a BeanInfo suitable for the GenericContainer
   * with which it is associated.
   * @return A BeanInfo suitable for the GenericContainer
   */
  public BeanInfo getBeanInfo();
  /**
```

```
     * Create a simple copy of this Personality. That is the configuration is copied,
     * but references to any GenericContainer beans are not copied.
     * @return A simple copy of this Personality.
     */
    public Personality duplicatePersonality();

    /**
     * Specity the type of Personality.
     * @return A String identifying the Personality Type.
     */
    public String getPersonalityType();

    /**
     * Indicate whether or not the Personality is separable from its GenericContainer.
     * An Atom (a GenericContainer that cannot be divided any farther) has an inseparable
     * Personality.
     * @see lanl.tsa3.simulation.composition.GenericContainer
     * @return true if separable ( not atomic )
     */
    public boolean separable();

    /**
     * Indicate a "name" for the GenericContainer associated with this Personality.
     * @return The name associated with this Personality and its GenericContainer.
     */
    public String getName();

    /**
     * Set the name for this Personality and its associated GenericContainer.
     * @param name The name for this Personality.
     */
    public void setName( String name );

    /**
     * Get the fully qualified name of the Icon associated with this Personality.
     * @return the fully qualified name of the Icon associated with this Personality.
     * null if no Icon is specified.
     */
    public String getIconName();

    /**
     * Set the fully qualified name of the Icon associated with this Personality.
     * @param iconName The fully qualified name of the Icon associated with this Personality.
     * null if no Icon is specified.
     */
    public void setIconName( String iconName );

    /**
     * Get the completeness status of the associated container.
     * @return true if the associated container is complete.
     */
    public boolean isComplete();
}
```

C. ComponentAtom.java listing.

```
package lanl.tsa3.simulation.component;

import lanl.tsa3.simulation.infrastructure.*;

/**
 * ComponentAtom is a convenience class simplifying the construction of Atoms.
 * Any Atom need only extend ComponentAtom and override any methods which need
 * to be made more specific.
 * Note: ComponentAtom is for Atoms constructed by implementing Personality. An
 *       Atom need not implement Personality but may own one.
 *
 * @version 1.0 1998 April
 * @author C. P. Booker
 */
public class ComponentAtom extends GenericContainer implements Personality
 {
  protected String name;
  protected String iconName        = "/Part/Generic";
  protected String personalityType = "Component";



  /**
   * Construct a ComponentAtom. Note that GenericContainer.personality is set
   * to "this".
   */
  public ComponentAtom()
   { personality = this; }

  // ***** Generic Personality Methods *****
  /**
   * queries whether or not the Personality will veto the
   * candidate GenericContainer. Since MessageConnectionManager is an Atom, all
   * are rejected.
   * @param bean A GenericContainer that could be added to the Personality's
   * GenericContainer.
   * @return true if the Personality accepts the candidate bean.
   */
  public boolean inspectGenericContainer ( GenericContainer bean ) { return false; }

  /**
   * constitues an attempt to add a GenericContainer to another GenericContainer.
   * The Personality may veto the add attempt by returning false. Since
   * MessageConnectionManager is an Atom, all
   * are rejected.
   * @param bean The candidate GenericContainer to be added to the Personality's
   * GenericContainer.
   * @return true if the bean is acceptable for addition.
   */
  public boolean addGenericContainer     ( GenericContainer bean ) { return false; }

  /**
   * attempts to remove a given bean from those known to the Personality.
   * @param bean The GenericContainer to be removed. Since MessageConnectionManager is an
   * Atom, all
   * attempts are rejected.
   * @return true if the Personality recognizes the bean and if it was successfully removed.
   */
  public boolean removeGenericContainer  ( GenericContainer bean ) { return false; }

  /**
   * Create a simple copy of this Personality. That is the configuration is copied,
   * but references to any GenericContainer beans are not copied. This is an Atom so
   * null is returned.
   * @return A simple copy of this Personality.
   */
  public Personality duplicatePersonality() { return null; }

  /**
   * Specity the type of Personality.
   * @return A String identifying the Personality Type.
   */
  public String getPersonalityType() { return personalityType; }

  /**
   * Indicate whether or not the Personality is separable from its GenericContainer.
   * An Atom (a GenericContainer that cannot be divided any farther) has an inseparable
   * Personality. Since this is an Atom, false is always returned.
   * @see lanl.tsa3.simulation.composition.GenericContainer
   * @return true if separable ( not atomic )
   */
  public boolean separable() { return false; }

  /**
   * Indicate a "name" for the GenericContainer associated with this Personality.
   * @return The name associated with this Personality and its GenericContainer.
   */
  public String getName() { return name; }

  /**
   * Set the name for this Personality and its associated GenericContainer.
   * @param name The name for this Personality.
   */
  public void setName( String name ) { this.name = name; }

  /**
   * Get the fully qualified name of the Icon associated with this Personality.
   * @return the fully qualified name of the Icon associated with this Personality.
   * null if no Icon is specified.
   */
  public String getIconName() { return iconName; }

  /**
   * Set the fully qualified name of the Icon associated with this Personality.
   * @param iconName The fully qualified name of the Icon associated with this Personality.
   * null if no Icon is specified.
   */
  public void setIconName( String iconName ) { this.iconName = iconName; }
```

```
 /**
  * Get the completeness status of the associated container.
  * @return true if the associated container is complete.
  */
 public boolean isComplete() { return true; }
}
```

D. DAPPC2.java, DAPPC2Personality.java, and DAPPC2Impl.java listings.

DAPPC2.java

```
package lanl.tsa3.simulation.actor.dapp.component;

import java.util.*;

import com.objectspace.jgl.SList;

import lanl.tsa3.simulation.actor.dapp.*;
import lanl.tsa3.simulation.actor.dapp.messaging.message.*;
import lanl.tsa3.simulation.actor.dapp.messaging.messageinterpreter.*;
import lanl.tsa3.simulation.actor.dapp.player.*;
import lanl.tsa3.simulation.component.*;
import lanl.tsa3.simulation.identification.*;
import lanl.tsa3.simulation.infrastructure.*;
import lanl.tsa3.simulation.messaging.*;
import lanl.tsa3.simulation.messaging.message.*;
import lanl.tsa3.simulation.runnable.*;

/**
 * DAPPC2 is provided as the interface to any DAPP Player's C2.
 *
 * @version 1.0 1998 May
 * @author C. P. Booker
 */
public interface DAPPC2 extends RunTimeCommands, RegisterIdentification
 {
 }
```

DAPPC2Personality.java

```
package lanl.tsa3.simulation.actor.dapp.component;

import lanl.tsa3.simulation.actor.dapp.*;
import lanl.tsa3.simulation.infrastructure.*;
import lanl.tsa3.simulation.messaging.*;
import lanl.tsa3.simulation.runnable.*;

/**
 * The base Personality for all DAPP Players.
 *
 * @version 1.0 1998 May
 * @author C. P. Booker
 */
public interface DAPPC2Personality extends DAPPPlayerPersonality
 {
  /**
   * Get the MessageReceiver from this Player.
   * @param receiver The MessageReceiver of the requestor.
   * @param requestor The name of the requestor.
   * @return The MessageReceiver from this Player.
   */
  public MessageReceiver getMessageReceiver( MessageReceiver receiver, String requestor );
  /**
   * Instruct the DAPPPlayer to connect to another DAPPPlayer.
   * @param otherPlayer The GenericContainer comprising the other Player.
   */
  public void connectToDAPPPlayer( GenericContainer otherPlayer );

  /**
   * Inspect a MessageConnectionManager component.
   * @param messageConnectionManager The candidate MesssageConnectionManager.
   * @return true if the candidate is acceptable.
   */
  public boolean inspectMessageConnectionManager( MessageConnectionManager
                                                  messageConnectionManager );

  /**
   * Attempt to add a MessageConnectionManager component.
   * @param messageConnectionManager The candidate MesssageConnectionManager.
   * Use a null to clear the current choice.
   * @return true if the candidate was accepted.
   */
  public boolean setMessageConnectionManager( MessageConnectionManager
                                              messageConnectionManager );

  /**
   * Get the reference to the extant MessageConnectionManager, if any.
   * @return a reference to the attached MessageConnectionManager.
   */
  public MessageConnectionManager getMessageConnectionManager();
 }
```

DAPPC2Impl.java

```
package lanl.tsa3.simulation.actor.dapp.component;

import java.util.*;

import com.objectspace.jgl.SList;

import lanl.tsa3.simulation.actor.dapp.*;
import lanl.tsa3.simulation.actor.dapp.messaging.message.*;
import lanl.tsa3.simulation.actor.dapp.messaging.messageinterpreter.*;
import lanl.tsa3.simulation.actor.dapp.player.*;
import lanl.tsa3.simulation.component.*;
import lanl.tsa3.simulation.identification.*;
import lanl.tsa3.simulation.infrastructure.*;
import lanl.tsa3.simulation.messaging.*;
import lanl.tsa3.simulation.messaging.message.*;
import lanl.tsa3.simulation.runnable.*;

/**
 * DAPPC2Impl is provided as a base class for any DAPP Player's C2.
 *
 * @version 1.0 1998 May
 * @author C. P. Booker
 */
public abstract class DAPPC2Impl extends ComponentAtom implements DAPPC2Personality,
                                                                  RunTimeCommands,
                                                                  RegisterIdentification
 {
  protected String                    name;
  protected DAPPPlayer.DAPPPlayerType myType                  = null;
  protected Message                   myIdentificationMessage = null;
  protected MessageConnectionManager  connectionManager;
  protected SList                     connectedPlayers        = null;
  protected MessageInterpreter        messageIntrepreter      = null;
  protected PlayerID                  thisPlayerID;
  protected boolean                   active                  = true;
  protected double                    currentTime             = 0.;
  protected long                      timeOffSet              = 0;
  protected TimerThread               timer                   = null;
  protected long                      timeBetweenUpdates      = 1000;
  protected SList                     locationListeners       = new SList();

  class TimerThread extends lanl.tsa3.simulation.utility.SimpleTimerThread
   {
    public TimerThread( long sleepTime ) { super( sleepTime ); }
    public void onExpiredTime() { update(); }
   }

  /**
   * Create a new DAPPC2Impl.
   */
  public DAPPC2Impl()
   {
    thisPlayerID = new PlayerID();
    thisPlayerID.playerName = name;

    currentTime = 0.;

    // For the parent GenericContainer:
    personality     = this;
    iconName        = "/Part/DAPP/C2/Generic";
    personalityType = "DAPPC2";
   }

  /**
   * Get the completeness status of the associated container.
   * @return true if the associated container is complete.
   */
  public boolean isComplete()
   {
    if ( connectionManager == null ) return false;

    return true;
   }

  // ***** DAPPPC2Personality methods *****
  /**
   * Get the MessageReceiver from this Player. Override this
   * method if more detailed tracking of connections is desired. Note, if this method
   * is used, then messageIntrepreter should be instantiated (probably in the child's
   * constructor) before it is called.
   * @param receiver The MessageReceiver of the requestor.
   * @param requestor The name of the requestor.
   * @return The MessageReceiver from this Player.
   */
  public MessageReceiver getMessageReceiver( MessageReceiver receiver, String requestor )
   {
    PlayerID candidatePlayer = connectionManager.acceptConnection( receiver, null, -1,
                                         requestor, thisPlayerID, messageIntrepreter );
    if ( candidatePlayer  == null ) return null;
    if ( connectedPlayers == null ) connectedPlayers = new SList();
    connectedPlayers.add( candidatePlayer );

    return candidatePlayer.receiver;
   }

  /**
   * Instruct the DAPPPlayer to connect to another DAPPPlayer. Override this
   * method if more detailed tracking of connections is desired, but that is NOT
   * suggested for future changes to the connection method. Note, if this method
   * is used, then messageIntrepreter should be instantiated (probably in the child's
   * constructor) before it is called.
   * @param otherPlayer The GenericContainer comprising the other Player.
   */
  public void connectToDAPPPlayer( GenericContainer otherPlayer )
   {
    System.out.println( "DAPPPlayer.connectToDAPPPlayer: otherPlayer = " + otherPlayer );
    PlayerID candidatePlayer = connectionManager.establishConnection( otherPlayer, null, -1,
                                                      thisPlayerID, messageIntrepreter );
    System.out.println( "     candidatePlayer = " + candidatePlayer );
    if ( candidatePlayer  == null ) return;
    if ( connectedPlayers == null ) connectedPlayers = new SList();
    connectedPlayers.add( candidatePlayer );

    //IdentificationMessage message = new IdentificationMessage( name, myType, true );
    System.out.println( "     sender = " + candidatePlayer.sender );
    candidatePlayer.sender.sendMessage( candidatePlayer, myIdentificationMessage );
    System.out.println( "    Sent identification message." );
   }

  /**
   * Inspect a MessageConnectionManager component.
   * @param messageConnectionManager The candidate MesssageConnectionManager.
   * @return true if the candidate is acceptable.
   */
  public boolean inspectMessageConnectionManager( MessageConnectionManager
                                                  messageConnectionManager )
   {
    if ( connectionManager != null ) return false;
    // Right now only accept direct connections:
    if ( messageConnectionManager instanceof DirectConnMgrPersonality ) return true;
```

```
    return false;
   }

 /**
   * Attempt to add a MessageConnectionManager component.
   * @param messageConnectionManager The candidate MesssageConnectionManager.
   * Use a null to clear the current choice.
   * @return true if the candidate was accepted.
   */
  public boolean setMessageConnectionManager( MessageConnectionManager
                                              messageConnectionManager )
   {
    if ( inspectMessageConnectionManager( messageConnectionManager ) )
      {
       connectionManager = messageConnectionManager;
       //if ( beans == null ) beans = new SList();
       //beans.add( (GenericContainer) messageConnectionManager );
       return true;
      }
     else return false;
   }

  /**
   * Get the reference to the extant MessageConnectionManager, if any.
   * @return a reference to the attached MessageConnectionManager.
   */
  public MessageConnectionManager getMessageConnectionManager()
   { return connectionManager; }

  // ***** RunnablePersonality Methods *****
  /**
   * Get the RunTimeCommand interface for this Reusable Component.
   * @return The RunTimeCommand interface for this Reusable Component.
   */
  public RunTimeCommands getRunTimeCommands() { return this; }

  // ***** RegisterIdentification method *****
  /**
   * Receive identification information from another Player. Note: Override
   * this method for a specific Player.
   * @param sender The PlayerID identifying the sender.
   * @param registerName The name of the Sender.
   * @param type The sender's type.
   * @param rsvp If true an IdentificationMessage is requested in return.
   */
  public void registerIdentification( PlayerID sender, String registerName,
                                      DAPPPlayer.DAPPPlayerType type, boolean rsvp )
   {
    // This Player does not care about others and so takes not action.
    /*
    System.out.println( "DAPPPlayer received RegisterIdentification: \n" +
                        "       sender = " + sender + "\n" +
                        "       name   = " + registerName + "\n" +
                        "       type   = " + type );
    */
    if ( registerName != null ) sender.playerName = registerName;
    if ( rsvp )
     {
      IdentificationMessage message = new IdentificationMessage( name, myType, false );
      //System.out.println( "     sender = " + sender.sender );
      sender.sender.sendMessage( sender, message );
      //System.out.println( "    Sent identification message." );
     }
   }

  // ***** RunTimeCommands Methods *****
  /**
   * Initialize a Reusable Component. Override to implement.
   * @param phase An int specifying the phase of initialization.
   */
  public void initialize( int phase )
   {}

  /**
   * Force the Reusable Component to take a step. Override to implement.
   */
  public void step() {}

  /**
   * Force the Reusable Component to pause. Override to implement.
   */
  public void pause() {}

  /**
   * Allow the Reusable Component to run. Override to implement.
   */
  public void run()
   {
    active = true;
    timeOffSet = System.currentTimeMillis();
    currentTime = 0.;
    update();
   }

  /**
   * Force the Reusable Component to stop. Override to implement.
   */
  public void stop()
   {
    active = false;
    if ( timer != null ) timer.interrupt();
   }

  /**
   * Force the Reusable Component to terminate. Override to implement.
   */
  public void terminate()
   {
    if ( connectionManager != null ) connectionManager.shutDown();
   }

  /**
   * For time-step based Players, a way to advance in time. Override to implement.
   */
  protected void update()
   {
    if ( !active ) return;
    long tempTime = System.currentTimeMillis() - timeOffSet;
    if ( tempTime > 0 ) currentTime = ((double) tempTime)/1000.;
    timer = new TimerThread( timeBetweenUpdates );
   }
 }
```

E. NIC2.java, NIC2Personality.java, and NIC2Impl.java listings.

NIC2.java

```
package lanl.tsa3.simulation.actor.dapp.component;

import java.util.*;

import com.objectspace.jgl.SList;

import lanl.tsa3.simulation.actor.dapp.*;
import lanl.tsa3.simulation.actor.dapp.messaging.message.*;
import lanl.tsa3.simulation.actor.dapp.messaging.messageinterpreter.*;
import lanl.tsa3.simulation.actor.dapp.player.*;
import lanl.tsa3.simulation.component.*;
import lanl.tsa3.simulation.identification.*;
import lanl.tsa3.simulation.infrastructure.*;
import lanl.tsa3.simulation.messaging.*;
import lanl.tsa3.simulation.messaging.message.*;
import lanl.tsa3.simulation.runnable.*;

/**
 * NIC2 is provided as a base interface for an NI Player's C2.
 *
 * @version 1.0 1998 May
 * @author C. P. Booker
 */
public interface NIC2 extends DAPPC2, RegisterSensorLocListener
 {
 }
```

NIC2Personality.java

```
package lanl.tsa3.simulation.actor.dapp.component;

import lanl.tsa3.simulation.infrastructure.*;
import lanl.tsa3.simulation.messaging.*;
import lanl.tsa3.simulation.runnable.*;

/**
 * The Personality for all NI C2s.
 *
 * @version 1.0 1998 May
 * @author C. P. Booker
 */
public interface NIC2Personality extends DAPPC2Personality
 {
  /**
   * Get the interface to the NIC2.
   * @return The interface to the NIC2.
   */
  public NIC2 getNIC2();
 }
```

NIC2Impl.java

```
package lanl.tsa3.simulation.actor.dapp.component;

import java.util.*;

import com.objectspace.jgl.SList;

import lanl.tsa3.simulation.actor.dapp.*;
import lanl.tsa3.simulation.actor.dapp.messaging.message.*;
import lanl.tsa3.simulation.actor.dapp.messaging.messageinterpreter.*;
import lanl.tsa3.simulation.actor.dapp.player.*;
import lanl.tsa3.simulation.component.*;
import lanl.tsa3.simulation.component.hardware.*;
import lanl.tsa3.simulation.identification.*;
import lanl.tsa3.simulation.infrastructure.*;
import lanl.tsa3.simulation.messaging.*;
import lanl.tsa3.simulation.messaging.message.*;
import lanl.tsa3.simulation.runnable.*;

/**
 * NIC2 is the NI Player's C2.
 *
 * @version 1.0 1998 May
 * @author C. P. Booker
 */
public class NIC2Impl extends DAPPC2Impl implements NIC2Personality,
                                                    NIC2,
                                                    RegisterIdentification,
                                                    RegisterSensorLocListener
 {
  protected SList                     locationListeners = new SList();
  protected BOOTPManager              bootpManager      = null;
  protected EtherliteManager          etherliteManager  = null;
  protected EtherliteSlave            etherliteSlave    = null;

  /**
   * Create a new NIC2. Note: in the constructor, some super field variable
   * are set:
   * 1. super.messageIntrepreter is set to the NIC2
   * 2. super.myType is set to the NIPLAYERTYPE
   * 3. super.iconName is specified
   * 4. super.myIdentificationMessage is created
   */
  public NIC2Impl()
   {
    super();
    myType                  = DAPPPlayer.NIPLAYERTYPE;
    iconName                = "/Part/DAPP/C2/NIC2";
    messageIntrepreter      = this;
    myIdentificationMessage = new IdentificationMessage( name, myType, true );
    personalityType         = "NIC2";
   }

  /**
   * Get the completeness status of the associated container.
   * @return true if the associated container is complete.
   */
  public boolean isComplete()
   {
    if ( !super.isComplete() ) return false;

    if ( bootpManager == null ) return false;
    if ( etherliteManager == null ) return false;
    if ( etherliteSlave == null ) return false;

    return true;
   }

  // ***** RunTimeCommands Methods *****
  /**
   * Initialize a Reusable Component. Override to implement. Note that 2
   * initializtion steps are required.
   * @param phase An int specifying the phase of initialization.
   */
  public void initialize( int phase )
   {
    if ( phase == 0 ) bootpManager.startBOOTP();
    if ( phase == 1 )
     {
      String ipControlledDevice = bootpManager.getControlledDevice();
      etherliteManager.connectTo( ipControlledDevice, 1, etherliteSlave );
     }
   }

  /**
   * Force the Reusable Component to pause. Override to implement.
   */
  public void pause() {}

  /**
   * Connect to the Sensors and parse data.
   */
  public void run()
   {
    etherliteManager.run();
    etherliteSlave.run();
   }

  /**
   * Force the Reusable Component to stop. Override to implement.
   */
  public void stop() {}

  /**
   * Force the Reusable Component to terminate. Override to implement.
   */

  public void terminate()
   {
    System.out.println( "NIC2 terminating..." );
    super.terminate();
    if ( bootpManager != null )     bootpManager.shutDown();
    if ( etherliteManager != null ) etherliteManager.shutDown();
    if ( etherliteSlave   != null ) etherliteSlave.shutDown();
   }

  // ***** RegisterSensorLocListener *****
  /**
   * Request that the sender be placed on
   * an announcement list for all Sensor Location Messages.
   * @param requestor The id of the DAPPPlayer desiring to receive all Sensor
   * Not Responding Messages.
   * @param registerName The name of the requestor.
   */
  public void registerSensorLocListener( PlayerID requestor, String registerName )
   {
    System.out.println( "NIPlayer.registerSensorLocListener: otherPlayer = " + requestor );
    //if ( locationListeners == null ) locationListeners = new SList();
    locationListeners.add( requestor ); // Should check for duplicates.
   }

  /**
   * Get the RunTimeCommand interface for this Reusable Component.
   * @return The RunTimeCommand interface for this Reusable Component.
   */
  public RunTimeCommands getRunTimeCommands() { return this; }

  /**
   * Get the interface to the NIC2.
   * @return The interface to the NIC2.
   */
  public NIC2 getNIC2() { return this; }

  /**
   * queries whether or not the Personality will veto the
   * candidate GenericContainer. This NIC2 accepts only:
   *  - BOOTPManager
   *  - EtherliteManager
   *  - SensorParser
   * @param bean A GenericContainer that could be added to the Personality's
   * GenericContainer.
   * @return true if the Personality accepts the candidate bean.
   */
  public boolean inspectGenericContainer ( GenericContainer bean )
   {
    Personality personality = bean.getPersonality();
    if ( personality instanceof MsgConnectMgrPersonality )
     {
      MsgConnectMgrPersonality temp = (MsgConnectMgrPersonality) personality;
      return super.inspectMessageConnectionManager( temp.getMessageConnectionManager() );
     }
    if ( personality instanceof BOOTPManagerPersonality )
     {
      if ( bootpManager != null ) return false;
      BOOTPManagerPersonality tempP = (BOOTPManagerPersonality) personality;
      return true;
     }
    if ( personality instanceof EtherliteMgrPersonality )
     {
      if ( etherliteManager != null ) return false;
      EtherliteMgrPersonality tempP = (EtherliteMgrPersonality) personality;
      return true;
     }
    if ( personality instanceof SensorParserPersonality )
     {
      if ( etherliteSlave != null ) return false;
      SensorParserPersonality tempP = (SensorParserPersonality) personality;
      return true;
     }
    return false;
   }

  /**
   * Add acandidate GenericContainer. This NIC2 accepts only:
   *  - BOOTPManager
   *  - EtherliteManager
   *  - SensorParser
   * @param bean A GenericContainer that could be added to the Personality's
   * GenericContainer.
   * @return true if the Personality accepts the candidate bean.
   */
  public boolean addGenericContainer ( GenericContainer bean )
   {
    if ( inspectGenericContainer( bean ) )
     {
      Personality personality = bean.getPersonality();
      if ( personality instanceof BOOTPManagerPersonality )
       {
        BOOTPManagerPersonality tempP = (BOOTPManagerPersonality) personality;
        bootpManager = tempP.getBOOTPManager();
        //if ( beans == null ) beans = new SList();
        //beans.add( bean );
        return true;
       }
      if ( personality instanceof EtherliteMgrPersonality )
       {
        EtherliteMgrPersonality tempP = (EtherliteMgrPersonality) personality;
        etherliteManager = tempP.getEtherliteManager();
        //if ( beans == null ) beans = new SList();
        //beans.add( bean );
        return true;
       }
      if ( personality instanceof SensorParserPersonality )
       {
        SensorParserPersonality tempP = (SensorParserPersonality) personality;
        etherliteSlave = tempP.getEtherliteSlave();
        etherliteSlave.setLocationListeners( locationListeners );
```

```
        //if ( beans == null ) beans = new SList();
        //beans.add( bean );
        return true;
       }
      if ( personality instanceof MessageConnectionManager )
       {
        MsgConnectMgrPersonality temp = (MsgConnectMgrPersonality) personality;
        super.setMessageConnectionManager( temp.getMessageConnectionManager() );
        return true;
       }
     }
    return false;
   }

  // ***** RegisterIdentification method *****
  /**
   * Receive identification information from another Player.
   * @param sender The PlayerID identifying the sender.
   * @param registerName The name of the Sender.
   * @param type The sender's type.
   * @param rsvp If true an IdentificationMessage is requested in return.
   */
  public void registerIdentification( PlayerID sender, String registerName,
                                    DAPPPlayer.DAPPPlayerType type, boolean rsvp )
   {
    // This Player does not care about others and so takes not action.
    System.out.println( "NIPlayer received RegisterIdentification: \n" +
                        "       sender = " + sender + "\n" +
                        "       name   = " + registerName + "\n" +
                        "       type   = " + type );
    if ( registerName != null ) sender.playerName = registerName;
    if ( rsvp )
     {
      IdentificationMessage message = new IdentificationMessage( name, myType, false );
      System.out.println( "     sender = " + sender.sender );
      sender.sender.sendMessage( sender, message );
      System.out.println( "    Sent identification message." );
     }
   }
 }
```

F. NIPlayerPersonality.java *and* NIPlayerPersImpl.java

```
package lanl.tsa3.simulation.actor.dapp.player;

import lanl.tsa3.simulation.actor.dapp.*;
import lanl.tsa3.simulation.infrastructure.*;
import lanl.tsa3.simulation.messaging.MessageReceiver;
import lanl.tsa3.simulation.runnable.*;

/**
 * The Personality for NIPlayers.
 *
 * @version 1.0 1998 May
 * @author C. P. Booker
 */
public interface NIPlayerPersonality extends DAPPPlayerPersonality
 {
 }




package lanl.tsa3.simulation.actor.dapp.player;

import com.objectspace.jgl.SList;

import lanl.tsa3.simulation.actor.dapp.*;
import lanl.tsa3.simulation.actor.dapp.component.*;
import lanl.tsa3.simulation.actor.dapp.messaging.messageinterpreter.*;
import lanl.tsa3.simulation.actor.dapp.messaging.message.*;
import lanl.tsa3.simulation.component.hardware.*;
import lanl.tsa3.simulation.identification.*;
import lanl.tsa3.simulation.infrastructure.*;
import lanl.tsa3.simulation.messaging.*;
import lanl.tsa3.simulation.runnable.*;

/**
 * The Network Interface Player parses digital multi-slice Frame data, converts
 * it into the corresponding Message, and distributes it to any Player that has
 * registered as being interested in the specified data.
 *
 * @version 1.0 1998 April
 * @author C. P. Booker
 */
public class NIPlayerPersImpl extends DAPPPlayerPersImpl implements java.io.Serializable,
                                                                    NIPlayerPersonality,
                                                                    RegisterSensorLocListener
 {
  protected GenericContainer niC2Container              = null;
  protected NIC2             niC2                       = null;
  protected GenericContainer bootpManagerContainer      = null;
  protected GenericContainer etherliteManagerContainer  = null;
  protected GenericContainer sensorParserContainer      = null;
  protected GenericContainer messagingManager           = null;
protected RunTimeCommands  runTimeCommands            = null;

/**
 * Construct the NIPlayer.
 */
public NIPlayerPersImpl()
 {
  super();
  personalityType = "/Actor/DAPP/NIPlayer";
  iconName        = "/Actor/DAPP/Player/NIPlayer";
 }

/**
 * Set the name for this Personality and its associated GenericContainer.
 * @param name The name for this Personality.
 */
public void setName( String name )
 {
  super.setName( name );
  if ( niC2Container != null ) niC2Container.getPersonality().setName( name );
 }

/**
 * Get the completeness status of the associated container.
 * @return true if the associated container is complete.
 */
public boolean isComplete()
 {
  if ( !super.isComplete() ) return false;

  Personality personality;

  if ( niC2Container == null ) return false;
  personality = niC2Container.getPersonality();
   if ( !personality.isComplete() ) return false;
  if ( bootpManagerContainer == null ) return false;
  personality = bootpManagerContainer.getPersonality();
   if ( !personality.isComplete() ) return false;
  if ( etherliteManagerContainer == null ) return false;
  personality = etherliteManagerContainer.getPersonality();
   if ( !personality.isComplete() ) return false;
  if ( sensorParserContainer == null ) return false;
  personality = sensorParserContainer.getPersonality();
   if ( !personality.isComplete() ) return false;

  return true;
 }

/**
 * queries whether or not the Personality will veto the
 * candidate GenericContainer.
 * @param bean A GenericContainer that could be added to the Personality's
 * GenericContainer.
 * @return true if the Personality accepts the candidate bean.
 */
public boolean inspectGenericContainer ( GenericContainer bean )
 {
  Personality personality = bean.getPersonality();
  if ( personality instanceof MsgConnectMgrPersonality )
   {
    if ( messagingManager != null ) return false;
    MsgConnectMgrPersonality temp = (MsgConnectMgrPersonality) personality;
    if ( niC2Container == null ) return true;
    DAPPC2Personality tempC2P = (DAPPC2Personality) niC2Container.getPersonality();
    return tempC2P.inspectMessageConnectionManager( temp.getMessageConnectionManager() );
   }

  if ( personality instanceof BOOTPManagerPersonality )
   {
    if ( bootpManagerContainer != null ) return false;
    if ( niC2Container == null ) return true;
    return niC2Container.getPersonality().inspectGenericContainer( bean );
   }

  if ( personality instanceof EtherliteMgrPersonality )
   {
    if ( etherliteManagerContainer != null ) return false;
    if ( niC2Container == null ) return true;
    return niC2Container.getPersonality().inspectGenericContainer( bean );
   }

  if ( personality instanceof SensorParserPersonality )
   {
    if ( sensorParserContainer != null ) return false;
    if ( niC2Container == null ) return true;
    return niC2Container.getPersonality().inspectGenericContainer( bean );
   }

  if ( personality instanceof NIC2Personality  )
   {
    if ( niC2Container != null ) return false;
    // Check extant owned components against the candidate C2. If any fail, then
    //  this C2 is not acceptable:
    boolean candidateTest = true;
    if ( bootpManagerContainer != null )     candidateTest =
                      personality.inspectGenericContainer( bootpManagerContainer );
    if ( !candidateTest ) return false;
    if ( etherliteManagerContainer != null ) candidateTest =
                      personality.inspectGenericContainer( etherliteManagerContainer );
    if ( !candidateTest ) return false;
    if ( sensorParserContainer != null )     candidateTest =
                      personality.inspectGenericContainer( sensorParserContainer );
    if ( !candidateTest ) return false;
    return true;
   }

  return false;
 }
```

```
/**
 * constitues an attempt to add a GenericContainer to another GenericContainer.
 * The Personality may veto the add attempt by returning false.
 * @param bean The candidate GenericContainer to be added to the Personality's
 * GenericContainer.
 * @return true if the bean is acceptable for addition.
 */
public boolean addGenericContainer     ( GenericContainer bean )
 {
  if ( inspectGenericContainer( bean ) )
    { // bean is acceptable, hook it up:
     Personality personality = bean.getPersonality();
     if ( personality instanceof NIC2Personality )
      {
       NIC2Personality tempPersonality = (NIC2Personality) personality;
       runTimeCommands = tempPersonality.getRunTimeCommands();
       niC2Container   = bean;
       niC2            = tempPersonality.getNIC2();

       if ( bootpManagerContainer != null )
         niC2Container.addBean( bootpManagerContainer );
       if ( etherliteManagerContainer != null )
         niC2Container.addBean( etherliteManagerContainer );
       if ( sensorParserContainer != null )
         niC2Container.addBean( sensorParserContainer );
       if ( messagingManager != null )
         niC2Container.addBean( messagingManager );
       return true;
      }
     if ( personality instanceof BOOTPManagerPersonality )
      {
       if ( niC2Container != null ) niC2Container.addBean( bean );
       bootpManagerContainer = bean;
       return true;
      }

     if ( personality instanceof EtherliteMgrPersonality )
      {
       if ( niC2Container != null ) niC2Container.addBean( bean );
       etherliteManagerContainer = bean;
       return true;
      }

     if ( personality instanceof SensorParserPersonality )
      {
       if ( niC2Container != null ) niC2Container.addBean( bean );
       sensorParserContainer = bean;
       return true;
      }

     if ( personality instanceof MessageConnectionManager )
      {
       if ( niC2Container != null ) niC2Container.addBean( bean );
       messagingManager = bean;
       return true;
      }
    }
  return false;
 }

// ***** RunTimeCommands Methods *****
/**
 * Initialize a Reusable Component. Override to implement. Note that 2
 * initializtion steps are required.
 * @param phase An int specifying the phase of initialization.
 */
public void initialize( int phase )
 { if ( runTimeCommands != null ) runTimeCommands.initialize( phase ); }
/**
 * Connect to the Sensors and parse data.
 */
public void run()
 { if ( runTimeCommands != null ) runTimeCommands.run(); }

// ***** RegisterIdentification method *****
/**
 * Receive identification information from another Player.
 * @param sender The PlayerID identifying the sender.
 * @param registerName The name of the Sender.
 * @param type The sender's type.
 * @param rsvp If true an IdentificationMessage is requested in return.
 */
public void registerIdentification( PlayerID sender, String registerName,
                                    DAPPPlayer.DAPPPlayerType type, boolean rsvp )
 { if ( niC2 != null ) niC2.registerIdentification( sender, registerName, type, rsvp ); }

// ***** RegisterSensorLocListener *****
/**
 * Request that the sender be placed on
 * an announcement list for all Sensor Location Messages.
 * @param requestor The id of the DAPPPlayer desiring to receive all Sensor
 * Not Responding Messages.
 * @param registerName The name of the requestor.
 */
public void registerSensorLocListener( PlayerID requestor, String registerName )
 { if ( niC2 != null ) niC2.registerSensorLocListener( requestor, registerName ); }

/**
 * Get the MessageReceiver from this Player.
 * @param receiver The MessageReceiver of the requestor.
 * @param requestor The name of the requestor.
 * @return The MessageReceiver from this Player.
 */
public MessageReceiver getMessageReceiver( MessageReceiver receiver, String requestor )
 {
  if ( niC2Container == null ) return null;
  DAPPC2Personality tempP = (DAPPC2Personality) niC2Container.getPersonality();
  return tempP.getMessageReceiver( receiver, requestor );
```

```
   }

  /**
   * Get the RunTimeCommand interface for this Reusable Component.
   * @return The RunTimeCommand interface for this Reusable Component.
   */
  public RunTimeCommands getRunTimeCommands()
   {
    if ( runTimeCommands != null ) return runTimeCommands;
    return (RunTimeCommands) this;
   }

  /**
   * Instruct the DAPPPlayer to connect to another DAPPPlayer. Override this
   * method if more detailed tracking of connections is desired, but that is NOT
   * suggested for future changes to the connection method. Note, if this method
   * is used, then messageIntrepreter should be instantiated (probably in the child's
   * constructor) before it is called.
   * @param otherPlayer The GenericContainer comprising the other Player.
   */
  public void connectToDAPPPlayer( GenericContainer otherPlayer )
   {
    if ( niC2Container != null )
     {
      NIC2Personality tempP = (NIC2Personality) niC2Container.getPersonality();
      tempP.connectToDAPPPlayer( otherPlayer );
      return;
     }
    super.connectToDAPPPlayer( otherPlayer );
   }
 }
```

G. RunnablePersonality.java *and* RunTimeCommands.java

```
package lanl.tsa3.simulation.runnable;

import lanl.tsa3.simulation.infrastructure.*;


/**
 * RunnablePersonality is the base Personality for all Reusable Components
 * that are runnable (ie. have Run Time Commands).
 *
 * @version 1.0 1998 April
 * @author C. P. Booker
 */
public interface RunnablePersonality extends Personality
 {
  /**
   * Get the RunTimeCommand interface for this Reusable Component.
   * @return The RunTimeCommand interface for this Reusable Component.
   */
  public RunTimeCommands getRunTimeCommands();
 }
```

package lanl.tsa3.simulation.runnable;

```
/**
 * The RunTimeCommands interface provides access to the Run Time methods
 * of a Simulation, Actor, etc.
 *
 * @version 1.0 1998 April
 * @author C. P. Booker
 */
public interface RunTimeCommands
 {
  /**
   * Initialize a Reusable Component.
   * @param phase An int specifying the phase of initialization.
   */
  public void initialize( int phase );

  /**
   * Force the Reusable Component to take a step.
   */
  public void step();

  /**
   * Force the Reusable Component to pause.
   */
  public void pause();

  /**
   * Allow the Reusable Component to run.
  */
 public void run();

 /**
  * Force the Reusable Component to stop.
  */
 public void stop();

 /**
  * Force the Reusable Component to terminate.
  */
 public void terminate();
}
```

H. RunTimeAdaptor.java

```java
package lanl.tsa3.simulation.runnable;

import java.io.*;
import java.util.zip.*;

import lanl.tsa3.simulation.infrastructure.*;

public class RunTimeAdaptor
 {
  protected RunTimeCommands runTime = null;

  class TimerThread extends Thread
   {
    protected long    sleepTime = 1; // ms

    TimerThread( long sleepTime )
     {
      this.sleepTime = sleepTime;
      start();
     }

    public void run()
     {
      try
       {
        synchronized( this ) { wait( sleepTime ); }
       } catch( Exception e )
          { System.out.println( "*** Error: unable to sleep." ); }

      stopRunning();
     }
   }

  public static void main( String[] args )
   {
    RunTimeAdaptor runTimeAdaptor = new RunTimeAdaptor( args[0], args[1] );
   }

  public RunTimeAdaptor( String beanName, String jarName )
   {
    System.out.println( "Starting the RunTimeAdaptor..." );
    GenericContainer bean = readBean( beanName, jarName );
    if ( bean == null ) return;

    Personality personality = bean.getPersonality();
    System.out.println( "RunTimeAdaptor: using bean = " + bean + " => " + personality );
    RunnablePersonality runnable = (RunnablePersonality) personality;
    runTime = runnable.getRunTimeCommands();

    System.out.println( "Found RunTimeCommands: " + runTime );

    runTime.initialize( -1 );
  runTime.run();

  TimerThread timer = new TimerThread( 21000 );

  System.out.println( "Letting Timer Thread run..." );
 }

public GenericContainer readBean( String beanName, String jarName )
 {
       byte buffer[] = new byte[1024];
  try
   {
    ZipInputStream zis = new ZipInputStream(
                          new BufferedInputStream(
                           new FileInputStream( jarName ) ) );
    ZipEntry ent = null;
         while ( (ent = zis.getNextEntry()) != null )
          {
           String name = ent.getName();
                /* the object we're loading */
                ByteArrayOutputStream baos = new ByteArrayOutputStream();

                /* NOTE: We don't know the size of an entry until
                   we've reached the end of one because of
                   compression. This means we can't just do a get size
                   and read in the entry.
                */
                for (;;)
                 {
                  int len = zis.read(buffer);
                  if (len < 0) break;
                  baos.write(buffer, 0, len);
                 }

                if ( !name.equals( beanName ) ) continue;

                ObjectInputStream ois = new ObjectInputStream(
                                         new ByteArrayInputStream( baos.toByteArray() ) );
                Object candidateBean = ois.readObject();
                ois.close();
                System.out.println( "RunTimeAdaptor.readBean: found object = " + candidateBean
                                 );
                return (GenericContainer) candidateBean;
          }
   } catch( Exception e )
      {
       System.out.println( "$$$ Error: Unable to read bean = " + beanName +
                           " from Jar = " + jarName + " e = " + e );
       e.printStackTrace();
      }
  return null;
 }

public void stopRunning()
```

```
  { runTime.terminate(); }
}
```

REFERENCES

1. David Flanagan, *Java in a Nutshell* (Sebastopol, CA: O’Reilly & Associates, Inc., 1997).
2. Bruce Eckel, *Thinking in Java* (Upper Saddle River, NJ: Prentice Hall PTR, 1998).
3. Los Alamos National Laboratory, *Samson User’s Manual* (Los Alamos, NM: Los Alamos National Laboratory, 1996).
4. Elliotte Rusty Harold, *Java Neetwork Programming* (Sebastopol, CA: O’Reilly & Associates, Inc., 1997).
5. Merlin and Conrad Hughes, Michael Shoffner, and Maria Winslow, *Java Network Programming*, (Greenwich, CT: Manning Publications, Co. 1996).
6. Prashant Sridharan, *Advanced Java Networking* (Upper Saddle River, NJ: Prentice Hall PTR, 1997).
7. Troy Bryan Downing, *Java RMI: Remote Method Invocation* (Foster City, CA: IDG Books Worldwide, Inc., 1998).
8. Rob Gordon, *Essential JNI: Java Native Interface* (Upper Saddle River, NJ: Prentice Hall PTR, 1998).
9. Scott Oaks and Henry Wong, *Java Threads* (Sebastopol, CA: O’Reilly & Associates, Inc., 1997).
10. Doug Lea, *Concurrent Programming in Java: Design Principles and Patterns* (Reading, MA: Addison-Wesley Publishing Co., 1997).
11. Steven Gutz, *Up to Speed with Swing: User interfaces with Hava Foundation Classes* (Greenwich, CT: Manning Publications Co., 1998).
12. Daniel I. Joshi and Pavel A. Vorobiev, *JFC: Java Foundation Classes* (Foster City, CA: IDG Books Worldwide, Inc., 1998).
13. Matthew T. Nelson, *Java Foundation Classes* (New York, NY: McGraw-Hill, 1998).
14. Michael Morrison, Randy Weems, Peter Coffee, and Jack Leong, *How to Program JavaBeans* (Emeryville, CA: Ziff-Davis Press, 1997).
15. Elliotte Rusty Harold, *JavaBeans* (Foster City, CA: IDG Books Worldwide, 1998).
16. Dan Brookshier, *JavaBeans* (Indianapolis, IN: New Riders Publishing, 1997).
17. Laurence Vanhelsuwe, *Mastering JavaBeans* (Alameda, CA: SYBEX Inc., 1997).