A GUI Configurator for Distributed Simulations in DOME

by

Twiggy G. Chan

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

February 4, 2003

Copyright 2003 Massachusetts Institute of Technology, All rights reserved.
A GUI Configurator for Distributed Simulations in DOME
by
Twiggy G. Chan

Submitted to the
Department of Electrical Engineering and Computer Science

February 4, 2003

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The DOME (Distributed Object Modeling Environment) technology enables distributed integrated simulations to be defined through an emergent process over the internet. DOME provides interoperability across a variety of software applications and supports dynamic information sharing. Processing the information generated by the integration simulations cannot be fully encapsulated by a single view of the system. Instead, multiple visualizations need to be produced to adapt the changing user needs and information availability. The DOME Configurator, a GUI builder tool, is a mass-customization solution for this need. With the Configurator, customized DOME GUIs can be rapidly configured and deployed. The architecture, implementation, and example application of the Configurator is discussed.

Thesis Supervisor: David R. Wallace
Title: Esther and Harold E. Edgerton Associate Professor of Mechanical Engineering
# Table of Contents

ABSTRACT.......................................................................................................................... 3

TABLE OF CONTENTS .......................................................................................................... 4

1. INTRODUCTION .................................................................................................................. 5

2. OVERVIEW .......................................................................................................................... 11

   FRAMEWORK ..................................................................................................................... 11
   DESIGN GOALS .................................................................................................................. 12
   RELATED WORK ................................................................................................................ 14
      Related User-Interface work ........................................................................................... 15
      JavaBeans ....................................................................................................................... 17

3. ARCHITECTURE .................................................................................................................... 22

   SUMMARY OF ARCHITECTURAL DESIGN ...................................................................... 22
   MAJOR DESIGN DECISIONS .............................................................................................. 23
      Level of Complexity ....................................................................................................... 23
      Language Choice ........................................................................................................... 26
   DESIGN SPECIFICATIONS ............................................................................................... 26
      Modes of the .................................................................................................................... 26
      Configurator ................................................................................................................... 26
      Dome Configurator Project Model ................................................................................. 28
      Persistence and Export Utilities .................................................................................... 31
      Visual Nesting Behavior ............................................................................................... 33
      Function Grouping ......................................................................................................... 34
      Bean Identification ......................................................................................................... 35
      Actions in the Configurator ............................................................................................ 35
      Interactions ..................................................................................................................... 39
      User Interface ................................................................................................................ 40

4. DEMONSTRATION .................................................................................................................. 46

   PILOT STUDY BACKGROUND ......................................................................................... 46
   CONFIGURATOR ROLE ................................................................................................... 48

DISCUSSION AND CONCLUSIONS .................................................................................. 51

   NEED FOR DOME CONFIGURATOR ............................................................................. 51
   CONTRIBUTIONS ............................................................................................................. 53
   LIMITATIONS AND FUTURE WORK .............................................................................. 55
   CONCLUSION .................................................................................................................. 57

LIST OF REFERENCES ......................................................................................................... 59
1. Introduction

A product that has been custom-made for individual user needs often provide superior performance. For instance, a tailored made suit will have a better fit than an off-the-rack version. When the product is used in business or research, the advantage of customization becomes even more apparent. A custom solution will often increase productivity, giving the user an edge above competitors. For such reasons, in the software field, the demand for custom solutions is especially strong. Often times, an organization needs a computational solution for a particular problem. If it is a simple and common problem, an off-the-shelf product might do most of what is needed but may not have every desired feature. If the problem is more unique, a solution might be stitched together from several different pieces of software. However, the best would be a well-written custom solution that specifically addresses the particular problem on hand. A good custom software solution can do everything the user specified and present that information in a way that’s useful for the particular problem.

The barrier to individual product customizations has historically been production cost [1]. Since the days of Henry Ford, products made by a labor-efficient mass-production assembly line were always several times less costly the custom-made counterparts.

Couture clothing is much more expensive than clothing produced in a factory. Buying a CD of generic web-server software is much cheaper than hiring developers to put together a special web-service. However, in some industries the cost barrier is
disappearing through advances in technology. Now it is possible for low-cost production and user-dependent customization to coexist. This coexistence is called mass-customization, defined as “a delivery process through which mass-market goods and services are individualized to satisfy a very specific customer need, at an affordable price” [2].

The mass-customization of software is possible. The development of software technology is usually a high-cost, time-consuming, and labor-intensive endeavor. Software development becomes very inefficient when the same family of code is reproduced from scratch each time someone has a problem and wants a somewhat different solution. Therefore, the key to achieving mass-customization of software is to cleanly separate commonly used features from user-specific ones. In other words, proper design abstractions enable powerful code reuse possibilities.

The important boundary between user-specific and general functionality is obvious. From the user’s viewpoint, the most important feature of a product is its user interface. If the software’s interface does not meet the needs of the user in an easy-to-understand manner, the user will probably opt for a different product or to simply do without. For example, suppose a user wishes to analyze the temporal trend of some data crunched out by the software tool. Also suppose that the software’s graphical user interface (GUI) plots the data against some other variable, say location, and the time information is only available by pointing the mouse over the data point. Although all necessary data are available, the interface of the software makes if very difficult for the user to do the temporal analysis.
In comparison, a display designed specifically to show the data’s temporal trend would facilitate the user’s work.

Edward Tufte, in his book Visual Explanations, suggests: “...clarity and excellence in thinking is very much like clarity and excellence in the display of data. When principles of design replicate principles of thought, the act of arranging information becomes an act of insight” [3]. So if the software’s core ability is to gather, process, and transfer information, the arrangement of information on the display should be done separately by the user whose expertise is in understanding the data. The information display should not be generalized by the software tool’s developers. Instead, developers should provide a mechanism for users to generate their own interfaces, or in other words, an interface configurator.

Providing an interface configurator would provide users with a mechanism to create customized interfaces. An effective interface configurator needs to accomplish two goals. First, it must provide a way for non-programmers to easily create a GUI. Secondly, the configurator should provide a straightforward process of connecting the information flow provided by the software technology to GUI. The configurator provides the means to produce fast and simple, therefore low-cost, customizations of the information technology. This promotion of code reuse is a mass-customization process.

Interface configuration tools have become increasingly popular. One prime example is National Instrument’s line of products. The company’s award-winning LabVIEW provide
scientist and engineers with a configurator as a means to create custom GUI’s that communicate with measurement and automation hardware systems. In other words, the configuration software is an off-the-shelf tool that makes it easy for NI customers to design, develop, prototype, test, and maintain their self-defined solutions. This has been wildly successful, attracting loyal customers from both big and small companies from a variety of industries. In 2000 85% of Fortune500 companies used NI products. In fact National Instruments maintains that “Customer Defined Solutions” is a central attribute that distinguishes them as a leader in computer-based and networked measurement and automation [4].

The MIT CADlab’s Distributed Object Modeling Environment (DOME) is a powerful simulation system that can benefit from mass-customization through an interface configurator. The goal of DOME is to enable emergent simulation systems for designing products. The technology provides the means for collaborators around-the-world to work together via communication over the internet. DOME also enables different types of software applications running on different operating systems to “talk” to each other. DOME provides very general tools to publish and subscribe to data models. DOME provides maximum freedom in the process of integrating models. The technology allows users to specify the flow of information, add scripts to modify the flow, and publish the resulting system so it can be participate in another more complex simulation [5].

One can imagine DOME as a world-wide simulation-web that allows users to join models and create distributed spreadsheet-like simulations that can span the globe. For example,
an industrial designer might publish into the DOME environment key parameters of a geometric CAD model, much like one can publish a web page in the WWW. Then an engineer, working in another location, might subscribe to parameters offered by the CAD model and link them to parameters in their own cost analysis spreadsheet. Consequently, if either the industrial designer makes changes in the CAD model or the engineer makes parameter changes in the cost model, appropriate changes are automatically triggered in the respective software applications. DOME automatically solves relationships between distributed simulations and synchronizes the execution of parametric models, regardless of what software application they are implemented in.

Because of the generality of the data model structure and flexibility in information flow, the possible applications of DOME technology are countless. Because of the undetermined nature of DOME applications, it is necessary to provide a means to quickly create custom interfaces in lieu of the generic DOME GUI. A GUI configuration tool was developed for the DOME project called the DOME Configurator.

The primary goal of the DOME Configurator project is to create a user-friendly environment that allows users to easily connect a graphical user interface to DOME simulations. The notion is that each participant in the DOME simulation can potentially have their own GUI view into the large system. The user’s view of the simulation is customized for their area of expertise and responsibilities, giving them the advantages of a custom solution. The cost of producing the custom views is kept low because the GUI’s can be easily constructed in the DOME Configurator.
In the next chapter, this document overviews similar configuration tools used in other systems. Ideas from these tools are incorporated into the design of the DOME Configurator. The architecture section then further describes the Configurator's architecture. An example demonstration of the DOME Configurator functionality is described in the fourth section. Finally, this document contains a discussion of the Configurator's role in the DOME vision.
2. Overview

Framework

The Configurator is a GUI builder for DOME simulations. The simulations are composed of data models and relationships specified as DOME objects. The simulation creators specify interfaces to sets of DOME objects that others can access. The Configurator tool enables the creation of custom GUI's for these interfaces. There could be many different interfaces to a large simulation project. The Configurator will make it easy to create a GUI for each interface, or multiple GUIs for each interface. With the Configurator, each collaborator in a project could define their own GUI to facilitate their tasks. In other words, the Configurator is a mass-customization mechanism for DOME services. The diagram below illustrates this concept.

Figure 1 Role of the Configurator in the DOME framework.
The Configurator builds GUIs that talk to DOME data models. For example, collaborators may be using DOME to design a bike. One CAD designer focuses on the geometric model of the bike frame. A mechanical engineer works on designing the gear assembly on the bike. There is also a supply manager in charge of obtaining and stocking the various parts needed in production. The CAD models, gear calculations, cost and supply spreadsheets are combined in DOME to create a system model simulation.

However, if the supply manager wants to focus on supply and cost issues, the standard DOME builder GUI would provide too much information. The data will also not be in the best format to help the manager understand the system-dependent part supply problems. The Configurator provides a solution. A GUI, customized to help the supply manager, can be inexpensively constructed. At the same time, the engineer and CAD designers can also their customized view of the same DOME simulation. All the collaborators share the same simulation solution. Because of the ability to customize interfaces rapidly, the solution can be easily adapted to meet the needs of a diverse set of users.

**Design Goals**

In order to provide mass-customized DOME interfaces, the Configurator’s design must provide four key attributes: Wide-spread availability; simple, rapid GUI construction; design flexibility; and code reuse.

First, the Configurator tool must be readily available for all potential users. This requires the software must run on most computing software. In other words, the user should be
free to work on a PC, Mac, Linux, or UNIX computer. The ability to design a custom interface should not be precluded from the lack of a computer with the right operating system. Cost is another potential roadblock in availability to users. Ideally, the Configurator and all necessary supporting software should be available free of charge to all interested parties.

The interface of the Configurator must simplify the GUI construction process. If the tool’s design confuses and discourages users, it will defeat the goal of low-cost and fast customization. The average user should not need to get special instruction in order to be able to create their custom DOME interface. In fact, the average user should not even have to know how to program. This means connecting components to function together should be a simple and understandable process. The Configurator should encapsulate DOME networking, making the integration of the subscribed models seamless. The details that are necessary to drive the interface should be hidden from the user as much as possible.

Third, the Configurator’s design must give the user as much flexibility as possible in the GUI designs. Of course, there are some trade-offs between flexibility and ease-of-use, but at the very least the user should be able to either make an applet, an application, or just another panel that can be incorporated in another GUI. The users should be allowed to control layout, specify the event flow in a very general manner, etc. The user should also be able to include their own components in the available tools and import images. If all those features are not provided in the initial implementation, the architecture of the
Configurator should make it possible for extensions to be added at a later point without affecting the compatibility with older saved projects. The source code generated by the Configurator should be readable and modifiable by the user so that it would be possible to make additions not provided by the Configurator.

In general, the Configurator should promote code reuse in order to make custom interface development as efficient as possible. A GUI designed for one project can be modified or added to for reuse in another project. The Configurator should also allow users to incorporate GUI components from outside sources. As a result of the flexibility in component reuse, the user would not have to start everything from scratch if someone else already has done the work.

**Related Work**

A study of similar work is useful in obtaining ideas on how to create an effective DOME GUI Configurator. The demo examples provided by Sun Microsystems: Bean Box [6] and Bean Builder [7] showed how Java Bean technology can be harnessed to accomplish the task at hand. Lab VIEW is a good example of practical graphical programming that breaks up the potentially complex into an understandable procedure. The MIT Life-Long Kindergarten Group’s FLOGO [8] and LOGOBlocks [9] projects explore visual interfaces that literally make programming child’s play. The MIT LCS Haystack [10] project looks into ways of displaying data in a more natural and useful way for users. The NetBeans [11], an open source IDE system, has a form builder module which is also an interesting study of the use of Java Bean technology as a graphical GUI builder. Commercial IDE also provides a Java GUI builder, the Form Editor.
Related User-Interface work

In one Haystack project at MIT, a “semantically rich and uniform UI” was created to demonstrate the power of having an interface that understands what the user means and understands. The interface would create a “New Things to Look At” list for the user composed of heterogeneous objects such as email, WebPages, papers, etc. Furthermore, the user could drag an email into calendar application, simplifying the task of adding email information to the user’s schedule. Interestingly, the added usability of the interface was made possible by the decoupling of agents that interpret the information and the visuals for the information. As a result, the information handling could be dynamic while the visualizations could be reused [10]. The idea of decoupling of visuals from information flow may also prove useful in the DOME Configurator architecture.

The FLOGO and LOGOBlocks are interesting implementations of visual programming. The goal is to enable children to visually construct simple programs that connect to toys.

Figure 2 LOGOBlock interface on the left and FLOGO interface on the right.
The process of designing a toy and program is actually quite fun. The child can learn about "while" and "for" loop behavior via experimentation. The interfaces for these programs are really simple and familiar to the target audience. In the LOGOBlocks, there are literally blocks that can be dragged from the toolbox into the play area. The blocks have specially shaped connectors, like LEGOs, for inputs and outputs. A program is constructed by piecing these blocks together [9]. FLOGO is for the older child, and the visual program focuses on virtual circuit construction [8].

National Instrument’s LabVIEW product is targeted for use by scientist and engineers. The users can make virtual instruments, VI, which can be a software version of an instrument or a computer interface to some actual hardware.

Figure 3 An example GUI built in LabVIEW.
There are two main parts in building a virtual instrument: 1) Laying out the Front Panel 2) Editing the Block Diagram. The front panel is the visual side of the VI. Components are added and laid out on a panel. The block diagram expresses the behavior of the VI. The block diagram has terminals corresponding to the components located in the front panel. The designer can add non-visual components to the block diagram, perhaps a wave-form generator or some other type of tool. The designer can specify data flows between the nodes in the diagram. It is even possible to graphically construct while loops or to even add probes to the diagram for debugging the system[12]. This interface appeals to the technical users of LabVIEW. The process of creating a virtual instrument is not unlike creating a physical instrument.

**JavaBeans**

JavaBeans is component architecture for the Java application environment. The platform-neutral technology produces components that can run on any OS and can even be ported to other application environments. This is a major advance for code inter-operability and reuse. According to Sun, JavaBeans is “the only component architecture to consider” that extends Java’s “Write Once, Run Anywhere” capabilities [6].

A Java Bean is an instance of a Java class that follows the Bean API. The bean object has properties that are fields with getter and setter methods. The user of the bean can manipulate the properties using the getter and setter methods. Java supports introspection of bean defining classes. Introspection of the class dynamically determines the properties and their setter and getter classes. This information is stored in a BeanInfo object. The
bean supplier may also choose to provide a custom BeanInfo for the bean. A custom BeanInfo can provide explicit bean information and specify icons and display name for the bean. Because of beans follow a standard API, they are components that can be plugged into programs that are also aware of the Bean API [6].

The Bean Box is a very simple builder that demonstrates the first generation of JavaBeans technology. The user adds Java Bean components in the Bean Box. Components are instantiated from Java Bean classes which are found on a Toolbox palette. The classes are loaded from default and user-specified Jar files. The Bean Box’s components can be made to talk to each other within the application. The user hooks-up component properties and events. The connection creation follows the event/response model. “Hook-up” objects are dynamically created to listen to the source component and process events by invoking a call on the target component. The properties of the components can be manipulated by the user through Property Editors accessed on a component property panel. The Bean Box holder itself is basically a Java AWT panel that also operates as a bean context. The Bean Box’s components are actually wrapped in helper Wrapper objects that assist the builder functionality such as interpreting mouse movements, holding dynamic hook-up objects, etc. Not all JavaBeans are necessarily visual components, thus the Bean Box will hide non-visual components during runtime while maintaining visual proxies during design time. The Bean Box demo generates applet code. The generated Applet class constructs all the children components, initializes their properties, and creates the related hook-up objects. For persistence, the original JavaBeans depended on binary serialization [6].
On the Java2 platform, especially the 1.3 and 1.4 releases, the Java Bean technology has further matured. The Bean Builder is a simple application builder that demonstrates the second generation technology. One important new development is the dynamic event
adapter generation capabilities built-into the Java 2 platform. The event generation can now be handled much more cleanly through the Dynamic Proxy API. Another very important development is the new long-term persistence mechanism for JavaBeans. Java now only supports long-term persistence of serialization encoded with XML rather than the old binary standard. The XML encoding is advantageous because of its human-readable quality. The readability makes it easier to ensure backwards compatibility or for other applications to include parsers to interpret the builder's project. The Bean Builder also demonstrates layout management using the new javax.swing.SpringLayout. It is possible to specify layout relationships between components in an application. There are also some very good improvements in terms of the usability of the builder's interface. New Property Editors improve editing of some common component properties. There are also handy navigation tools for component properties and the architecture of the design being built. Beans can also be instantiated from any class available in the builder's Java environment, instead of having to be formally loaded as a tool in the palette. Unlike the Bean Box demo, the Bean Builder's architecture better delineates the builder's functionality from the design of the GUI being built. The bean components are not wrapped with a builder functionality object, but are directly nested together. However, the Bean Builder does not generate easy-to-modify source code. The designed root component can be serialized into an XML file and de-encoded and run by separate applications that have access to all the needed classes [7].

The NetBeans is an open-source development environment written in Java. It is a platform for developing projects in languages such as Java, HTML, XML, C/C++ etc.
3. Architecture

**Summary of Architectural Design**

There are four primary steps in a typical project undertaken by a Configurator user. In the first step, the user visually designs a GUI. Second, the user will import subscriptions of DOME Models. Next, the user specifies the interactions between all the parts in their GUI design. And finally, the user will want to distribute their design either as an application, applet, or a reusable component. The architecture of the Configurator breaks down into packages that support each of these steps. Below is a high-level diagram of the Configurator organization.

![Figure 7 Key structure of the Configurator.](image-url)
The architecture of NetBeans is modular and developers are encouraged to write commercial and non-commercial plug-ins. NetBeans can be used as an IDE to develop new projects, or as a platform to jump-start development of new applications [11]. A core NetBeans module is the Form Editor. The Form Editor is a visual builder for GUI projects. A nice feature of the Form Editor not seen in the Bean Box or Bean Builder demo is the code-generated viewer. The project’s source code is automatically generated and displayed as the designer works [13]. The documentation of the NetBeans project has been useful place to read about the practical challenges of creating a good application builder with JavaBeans.

Figure 6 Form Editing in the NetBeans IDE.
component such as a plotter. With this in mind, should resemble LabVIEW more than it resembles an IDE such as NetBeans. The number of different tasks a user needs to perform should be kept to minimum. At the same time, the Configurator should provide a robust and customizable toolbox so that users have the flexibility to make the additions needed in their specific project.

To reduce complexity, the Configurator focuses on facilitating the design of the most common type of user project. So instead of providing mechanisms to create projects with multiple windows and menus with complicated action events, the Configurator focuses on panel design. For example, both the NetBeans Form Editor and the Bean Builder expect users to construct application starting from top-most container level, the Swing JFrame. Although this allows the user greater flexibility in their design, it complicates the design process. The user might get confused when looking at the project tree. First the user will have to know what a JFrame and JLayeredPane are. Additionally, they might wonder why there is more than one JPanel and JLayeredPane under the JFrame root. The user might not know where in the tree they should instantiate non-visual components.

Hiding the Swing structure reduces some of the user confusion. The Configurator provides the users with just two options: 1) Design a panel 2) Design a panel with multiple layers. After designing their GUI, the user can choose to export the project as an Application, Applet, or component that can be embedded in a future project. For most users, this simplification will not hinder what they want to accomplish. Besides making the Configurator easier to understand and use, the simplification encourages users to
The tool box and property customizer are fundamental project creation tools, supporting the component based structure of user designs. Future versions of the Configurator may include a tree view and a code editor. The workspaces each help the user accomplish one step of the interface design process. Additional workspaces can be added in the future if desired. Removing or replacing a workspace will not affect the operability of the other packages. The project holder acts as a container for the project object, holding dynamic design information. The project object itself is exportable and can be fully functional outside of the Configurator. The basic types of projects are the DomePanel and DomeLayeredPane. The Configurator application is managed by control environment that provides action definitions, window management, and a menu bar.

**Major Design Decisions**

**Level of Complexity**

Central to the Configurator is the GUI. The usability of Configurator’s GUI is important. Usability depends on how easy it is for the DOME user to figure out how put together a GUI from some library of components. By identifying and characterizing the most common type of usage, it is possible to incorporate user-friendly qualities in the Configurator. The hypothesis is that the typical GUI design will be a simple panel with some buttons, labels, text boxes, some images, and maybe a special data-display.
design panels that can be reused. Instead of deciding between creating an applet or application at the start, users are led to create panels that can be used in both ways.

Figure 8 Realm of project types.

The drawback of concentrating on user design of panels or layered panes is reduced flexibility. The user would not be able to create an application with menu bars. There is also no support for multi-window management applications. Furthermore, the panel simplification makes the assumption that all designed projects are visible. Creating a non-visible component would be outside the scope of the Configurator’s tool set. Perhaps, in actuality the hypothesis that most users would just need simple GUI’s is wrong. Thus, the DOME Configurator should have an architecture that can be extended to support building of more complicated applications. In the future, to provide users with more IDE capabilities, the Configurator could be restructured as a module in the open-source Net Beans project. This process would not be difficult because the controller follows a standard Java action definition pattern to control application events.
Language Choice

The DOME Configurator is written in Java and generates Java applications, applets, and components. Java is used because once compiled, the Java binaries can run on any platform. This means users can simply download the freely available Java Runtime Environment to run DOME software. Furthermore, Java’s WebStart technology makes it easy to provide software and updates online. Each time you mention a new technology there should be a citation.

The Configurator’s purpose also calls for a component-based architecture. The GUI building is done by gathering components and hooking them together for a particular functionality. As discussed, in Java this means the Configurator uses Bean technology. The demos, Bean Box and Bean Builder, provided by Sun served as guides in creating the DOME Configurator.

Design Specifications

Modes of the Configurator

The modes of the Configurator reflect the stage of the design project. Therefore, the states are: No Project, Initialization, Visual Design, DOME Subscription, Interaction Design, Testing/Export/Saving, and Closing Project. Currently, only one project can be
opened at a time, a simplification for now. It is possible to extend to more than one open project later.

![Configurator modes of operation](image)

**Figure 9** Configurator modes of operation.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Project (Dormant)</td>
<td>User can only create a new project, open an old project, or exit the Configurator. No other actions are possible.</td>
</tr>
<tr>
<td>Initialization</td>
<td>User is creating a new project. Wizards walk the user through the process of creating a new project. The user can choose to either design a single-layer panel or a multi-layer panel. Note in a multi-layer design panel, the user can choose to have only one layer. The implementation would be that in the single-layer panel project, the root GUI container is a subclass of a JPanel. In the multi-layer panel project, the root visual container is a subclass JLayeredPane with a single JPanel as the main container for each layer. A default project setting choice is provided so that the user can skip walking through the wizard.</td>
</tr>
<tr>
<td>Dome Configurator Project Model</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
</tr>
</tbody>
</table>

There are two general layers. There is an overall wrapper class called the ConfiguratorProjectHolder. This class provides an interface between the underlying user project and the rest of the Configurator. The wrapper class holds helpful information such as the event hookups that have been created between beans in the user project, tables mapping the name of the object to the object, etc. The ConfiguratorProjectHolder also manages the clipboard necessary in the designing of the project.
The user project, the root object that is being constructed, is a visual container component. For current Configurator purposes, the classes of user projects are sub-classes of JPanel and JLayeredPane that implement the DomeSubscriber, FunctionSupport, and VisualRoot interfaces. Implementing the DomeSubscriber interface means that the panels support DOME subscriptions. The FunctionSupport interface provides methods to add interactions to the components of the project. The VisualRoot interface provides a method to get a list of all the visual descendents of the project root.

The projects implement the DomeSubscriber by creating a DomeSubscriptionServiceContext. This bean context contains DOME data object beans such as a simulation interface and its parameters and relations.

![Diagram of the DomeSubscriptionServiceContext functionality.](image)

Figure 10 Diagram of the DomeSubscriptionServiceContext functionality.
These beans interact with a DomeClientService. The DomeClientService class implements the DOME Client API. The DomeClientService instance is generated by a service provider, the DomeServiceProvider. The DomeServiceProvider class implements the DOME subscription process. The interface includes methods to start and stop subscription to the Dome models. The interface provides methods to add, remove, and query the project's list of subscriptions.

The FunctionSupport provides methods to obtain an un-nested list of all the components in the project. In implementation, the FunctionSupport interface is provided by a FunctionSupportContainer. The FunctionSupportContainer is a bean context. All the components added to the project are placed in this bean context. For example, a visual component such as a button can have to be a child of the FunctionSupportContainer. It may or may not be a visual descendent of the root component, DomePanel or DomeLayeredPane. If the button is not a descendent of the root visual component, the button is not visible on the GUI but can still interact with other project components. Non-visual objects that assist in the functionality of the design are found only as beans in the FunctionSupportContainer. DomeSubscriberParameters are also contained by this bean context.

The VisualRoot interface only defines one method: a method to get a list of all the descendents of the visible root object. So, in implementation would build this list by recursion down the component tree structure using the getChildren() calls.
Currently, the two possible classes of user projects are DomePanel and DomeLayeredPane. Classes of user projects can be added later if support for a greater variety of design projects. For example, if supporting designs of stand-alone frames is desired, we might add a DomeFrame class. Note that future user project classes do not have implement DomeSubscriber or FunctionSupport, but certain design processes will not be available to the user without those interfaces in the user project. It is a good idea to make the sources of DomePanel and DomeLayeredPane classes available for modification/extension by the user. Note, to be clear Configurator projects are required to be visual container objects. Currently the Configurator supports only having one project open.

Persistence and Export Utilities

Project Saving

To save a project so that it can be worked on at a later time, the Configurator writes the ConfiguratorProjectHolder as an XML file. The Configurator can open a saved project by simply decoding the file. This is because Java now supports persistence via XML encoding. Using Java persistence means that it is best if the objects added to the user process should follow bean specifications. A DefaultPersistenceDelegate attempts to generate the correct expressions from information gather from standard Bean Introspection. If the DefaultPersistenceDelegate cannot properly write out the state of a class instance, then a PersistenceDelegate should be provided for the class. The
SpringLayoutPersistenceDelegate is necessary to produce correct XML encoding of Swing containers using the SpringLayout layout manager.

**Component Reuse**

The user can export their design for future reused as a component in other Configurator projects. In the export process, the Configurator writes the user project root, DomePanel or DomeLayeredPane, out as an XML file. The Configurator provides tools that can read components that have been encapsulated in XML, and adds the reconstructed component to the new project. Below is an illustration of the component reuse cycle.

![Component Reuse Cycle](image)

**Figure 11** Component reuse cycle.

**Export to Application or Applet**

The Configurator generates applications by generating a simple Java class. The generated class is extends JFrame. The application’s main method decodes the DomePanel or DomeLayeredPane instance from an XML file and adds the resulting object to the contentPane. The result is then set to visible. If necessary, the user can further customize their application by had by using the API of the root object. The DomePanel and
DomeLayeredPane interface allows access to all the subcomponents of the project. The export to Applet works in a similar manner except the template extends a JApplet class. An improvement would be to build Java WebStart support into the template. For the future, could look into using the archiver package to generate full Java Source Code instead of depending on the XML encoding. Problem with the archive package is that it only works with Java 1.3. Also, there is not a great deal of benefit is avoiding XML persistence usage since XML is human readable as well.

**Visual Nesting Behavior**

To facilitate component reuse, the Configurator supports nesting of visual components. An old project can be incorporated into the new design and the user can make seamless modifications to the new design. Of course the old project would have to add its DOME subscriptions and other functionality support beans added properly to the new project as well to make this nesting function correctly.

![Analyse Regional Data by Quarter](image)

**Figure 12 Nesting visual components.**
The ability to make seamless modification means that users can visually access any visible component in the tree. For example, let an instance of a DomePanel be the root container. Users can add a button to the root container, but they can also create a smaller panel within the DomePanel and instead add the button to the child panel. Thus the visual child of a root can be a container that has children that are containers or just visual components nodes. Ideally in visual nesting support, the designer can select a great-grandchild of the root container, and modify as if it were direct child component. The main hurdle in the implementation is that mouse motion interpretation is challenging and it becomes necessary to specifically almost every type of component layout.

**Function Grouping**

Sometimes, specifying a particular behavior in the GUI requires grouping of components. Take for example the usage of button groups. Collecting selectable buttons in a button group produces the behavior that only one button of that group can be selected at any time. It would be difficult to specify this functionality in any other way. Or, as another example, the designer might want to group some components into a list. Currently, the Configurator provides only a few basic grouping tools, such as grouping into a button group or creating a list of objects for a JList, JComboBox, or JSpinner. In the future, to support a wide variety of possible GUI functionalities, it will be good to support nesting of these functionality groups. The LabVIEW diagram builder allows users to group components to create while loops and other type of abstract behavior. With extended
grouping ability, the ability to create multiple visual references to the same object might also become necessary.

**Bean Identification**

Every bean added to the project is given a unique name. The ability to identify a specific object by a unique name assists both the user and the Configurator in the design process. If possible, the unique name of the bean is equivalent to its name property, if such a property exists. The name property exists in all subclasses of the java.awt.component class. The Configurator provides a text box in which the user can change the selected bean’s name to one’s liking. Of course, the uniqueness of the new name is checked before it is applied. The unique naming is handled by a UniqueNameManager object that is bound to the project holder.

**Actions in the Configurator**

Actions classes are created for each type of event triggered by the user. Listeners and triggers register the actions related to their purpose. This consolidation makes it easier to construct multiple buttons or menu items that have the same purpose. It also makes it easier to turn actions on and off. There are two basic superclasses of all the actions in the Configurator. The base Delegate Action that forwards all events of the type to a single ActionListener. The StateChangeAction is a subclass that also keeps the state, selected or not, of the action. The StateChangeAction is useful for triggers that might be represented as checkboxes or radio buttons.
Table 2 List of actions in the Configurator application.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewAction</td>
<td>This action is only enabled when the Configurator is in the NoProject state. The NewAction means the user has requested to start a new project.</td>
</tr>
<tr>
<td>OpenAction</td>
<td>This action triggers the Configurator to bring up an OpenFile Dialog. If a valid file is found, all the proper workspaces are initialized. This action is only enabled when the Configurator is in the NoProject state.</td>
</tr>
<tr>
<td>SaveAction</td>
<td>When this action is triggered, a dialog requesting a file name and location is started. The current project is then saved with an XMLencoder.</td>
</tr>
<tr>
<td>ExportApplication</td>
<td>This starts a dialog asking the user to supply the application name and location of where the resulting files should be placed.</td>
</tr>
<tr>
<td>ExportApplet</td>
<td>This starts a dialog asking the user to supply the application name and location of where the resulting files should be placed.</td>
</tr>
<tr>
<td>ExportComponent</td>
<td>This starts a dialog asking the user to supply the component name and location of where the resulting files should be placed.</td>
</tr>
<tr>
<td>CloseAction</td>
<td>Closes the current project. First it will also ask the user if they wish to save the current project. The state of the Configurator</td>
</tr>
</tbody>
</table>
goes to ClosingProject mode. This action will close all the windows except for the window containing the MainMenuBar. The state of the Configurator reverts to NoProject mode.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExitAction</td>
<td>The user is requesting that the Configurator closes. This is triggered from the menu bar or by closing the window containing the menu bar. Again, the user is asked if they wish to save the current project before the exit action finishes.</td>
</tr>
<tr>
<td>TestAction</td>
<td>This action occurs when the user requests that the current project is tested as an application. A new frame is created with running the current project in its content pane, as an exported application would do.</td>
</tr>
<tr>
<td>AddAction</td>
<td>Adds a new bean into the workspace from the Tool Box.</td>
</tr>
<tr>
<td>CutAction</td>
<td>Some bean or interaction has been removed, perhaps temporarily, from the project. The removed is placed in the appropriate clipboards. Thus clipboards listen to these types of actions.</td>
</tr>
<tr>
<td>CopyAction</td>
<td>A bean or interaction has been copied. The original remains where it is in the project. But there is a reference to the “copied” object so that a clone can be made when paste is called.</td>
</tr>
<tr>
<td>StartPasteAction</td>
<td>A paste is started by a clipboard. The paste is done on the last workspace in focus.</td>
</tr>
<tr>
<td>DonePasteAction</td>
<td>The paste has been completed in the workspace. Listeners can...?</td>
</tr>
<tr>
<td>Action</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>undo their paste dependent states.</td>
<td></td>
</tr>
<tr>
<td><strong>DeleteAction</strong></td>
<td>A bean or interaction is removed and cannot be recovered.</td>
</tr>
<tr>
<td><strong>LoadJarAction</strong></td>
<td>Starts the process of loading a jar file as a palette on the toolbox.</td>
</tr>
<tr>
<td><strong>&quot;Window&quot;ShowAction</strong></td>
<td>The user can request that this window be brought to the top of the desktop.</td>
</tr>
<tr>
<td><strong>&quot;Window&quot;FrontAction</strong></td>
<td>The user has requested the about information.</td>
</tr>
<tr>
<td><strong>GroupAction</strong></td>
<td>The selected bean has been changed. Selected bean has to exist in the project, therefore objects in clipboard cannot be the currently selected bean. The Configurator keeps track of the previously selected bean and the currently selected bean. The selected bean should be selectable in the workspace currently in focus.</td>
</tr>
<tr>
<td><strong>AboutAction</strong></td>
<td>The user has requested help information.</td>
</tr>
<tr>
<td><strong>StartSubscriptionAction</strong></td>
<td>Indicates that the user wants to activate a subscription to a DOME service.</td>
</tr>
<tr>
<td><strong>StopSubscriptionAction</strong></td>
<td>Indicates that the user wants to deactivate a subscription to a DOME service.</td>
</tr>
</tbody>
</table>
Interactions

When creating a custom GUI, the designer will define interactions between the components in the design. The Configurator facilitates the following types of hookups. The user defines the proper interaction using the interaction workspace. Additionally, the interaction workspace provides tables that allow users to see the hookups that have been made.

Table 3 Types of user defined interactions.

<table>
<thead>
<tr>
<th>Interaction Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property Connection</td>
<td>The user can also specify that a property of bean 1 is set to bean2.</td>
</tr>
<tr>
<td></td>
<td>For instance, the JList's model can be set to the List Model of the user's choice.</td>
</tr>
<tr>
<td>Event Connection</td>
<td>Using the new Java Action Proxy API, it is possible to dynamically create event listeners. Thus the Configurator user can specify that bean 1's setX() method is called when an action occurs on any bean in the project. The arguments to the method</td>
</tr>
</tbody>
</table>
User Interface

MainMenuBar

The main menu bar contains a menu of general project controls, such as new, open, save, close, export, and exit. Also located on the main menu bar are menus that are specific to a process in the design project. A menu with editing functions such as cut, copy, paste, and delete is also available. Others include a menu for the component palette and a menu for managing event hookups. The user can also control which windows/accessories are showing or hidden. The Configurator also includes a help menu.

Workspaces

The most important windows in the Configurator’s GUI are the workspaces. There is a workspace for each process of the user’s project development. Therefore, there is a workspace to support the visual design of the user GUI, another workspace to add DOME subscriptions to the project, and yet another to let the user specify the functionality of their design. Each workspace listens to changes and events to the Dome GUI.

In general, a workspace is a JFrame with a toolbar that provides controls for the workspace. There also might be an info bar that provides current information about the
process in the workspace. The workspace will hold a view of the user project. The view can either be held in a JInternalFrame or directly in the contentPane of the workspace.

The project-GUI layout view uses a JInternalFrame. This helps make it visually clear where the user's project panel starts and where the Configurator's GUI ends. This workspace also provides controls to layout visual components. The user can select the layout manager desired for the panel or layer. If the user changes the layout after having added components, all the added components are removed to a clipboard. The user can reinstate the components using the new type of layout. The way the mouse behaves or what mouse motion means depends mostly on the type of layout being used. An overlay GlassPane is added to the JFrame and intercepts mouse motions. The mouse motions are interpreted according to the layout. Therefore, supporting the SpringLayout and the TableLayout necessitates SpringLayoutGlassPane and a TableLayoutGlassPane. True support of component nesting will require GlassPanes for every type of LayoutManager.

The process of visually laying out a project depends on the type of root visual container. The layout design workspace of a DomeLayeredPane is a little different from the layout design workspace of a DomePanel. The workspace in DomeLayeredPane allows users to add and delete layers. Furthermore, the workspace provides tools to switch between layers or to set whether the bottom layers are visible while the current layer is being designed. These features are unavailable in the DomePanel's layout workspace.
In the workspace that supports DOME subscription, there is a browser panel for the models available for subscription, another panel that displays the tree structure of the currently selected model, and another panel of the added DOME model subscriptions. Finally, there is also a workspace that allows the user to design the functionality of their GUI. This workspace contains a panel with icons of the visual components of the GUI. The user can also add helper, non-visual, components to their project by adding them to this panel. Controls are available in this workspace to create interactions between the components of the project. There are also tools that display the hookups that have already been made between the components.

The interaction of workspaces with the other parts of the Configurator is modular. Therefore, it should be easy to create new workspace types or substitutes. If say, we want a different workspace. Workspaces also seemingly communicate with each other. However the communication is not direct. Workspaces make changes to the user’s project. In turn, the workspaces also have to listen to changes to the user project made by other workspaces.

The Configurator attempts to make the last selected component of prior focused workspace the selected component to the workspace that is given the new focus. Of course this is not always possible since workspaces don’t always show the same set of components. For example, if the user cuts a component in the visual design workspace, the change is also shown in the function designer. The removed component should be
found in the clipboard of the visual designer and in the component clipboard of the function designer.

Furthermore, all the interactions that involve the removed component are also removed to the function designer’s function clipboard. To facilitate location of these interactions, hash maps are kept with component name keys and values being a vector of all the interactions they are involved in. The interactions are represented by objects. The interaction representative classes have a valid/not-valid boolean state. When a component is moved to the clipboard, the interactions are invalidated. Component just has to toggle the valid and not-valid property. All the interaction references are not removed until the interaction is truly deleted.

A bean paste while in the function designer is different from pasting in the visual design workspace. A paste in the function design workspace means the component is not visible, not added as a child of a visual container. But the component exists in the project as a potential functionality helper. However, the visual designer’s clipboard will still show that the component is still available for addition into the visual layout.

The user can hide or show the each workspace. There is only one instance of a particular workspace for the open project. Not all workspaces apply to every type of project. In fact, workspaces are sometimes customized for a particular type of user project. Furthermore, the state of the Configurator depends on which workspace is in focus.
**Clipboards**

Clipboards are associated with particular workspaces. There is clipboard for the visual design workspace. The functionality designer actually has two clipboards: one for components and one for interactions. The DOME subscriber has a clipboard too.

**Property Sheet**

Because the Configurator projects are based on Java Beans, a PropertySheet window allows users to view and set the properties of the component beans. The PropertySheet window allows the user to select the type of properties they want to see. The PropertySheet uses PropertyEditors to let users change the value of the property. Some default editors for the most common classes of properties are included. However, it is expected that bean distributions include custom property editors for non-standard property classes.

**Tool Box**

The component toolbox is also an important part of the Configurator's GUI. The toolbox allows the user to choose components to add to the project design. There are three possible ways to add a component to the project. The toolbox provides the user interface for these three methods. First, there are palettes of components available for addition to the project. These palettes contain handy buttons with icons representing the type of component. The frequently used components are available on the palettes. The user can also add to these palettes by loading jar files of beans. Secondly, components can be instantiate from any class in the class path. The toolbox provides a textbox where the user
can request a specific component that can be instantiated in the current classpath. Finally, a component can be instantiate from an xml file. The toolbox tool tip information which provides a brief explanation of each tool.
4. Demonstration

The Configurator was created to support integrated simulations in the DOME environment. This section illustrates the role the Configurator plays. In a pilot study with Ford Motor Company, Abrahamson et al. demonstrated that DOME can be used to build an emergent system simulation of product design [5]. In this demonstration of the Configurator, simple GUIs for the system simulation are created.

**Pilot Study Background**

The study at Ford constructed a full door glass drop design for a car. Models in three different internal organizations and two suppliers were integrated to create the system simulation of a glass-drop. The complete simulation involved 21 models in different third party applications and the coordination of some 1500 shared design parameters.

![Figure 13 Car door design activities.](image)
One performance parameter in the design of a system is the velocity of the glass-drop in the car door. To do this calculation, a spreadsheet was created by a Ford engineer. This spreadsheet calculation was the basis of one of the DOME models in the pilot study. Another model was created from the geometric representation developed by a CAD designer. Yet, there was a third spreadsheet based model that calculated the cost of using various parts in the design.

The first step of building the simulation was for model owners to build and deploy parametric interfaces to their application. By doing so, other participants can interact with their model simulation over the Internet. Below shows the examples of how these interfaces look in the third-generation DOME builder.

Figure 14 Examples DOME models used in the glass drop study.

Once deployed, other collaborators may subscribe to the parameters exposed by the interfaces. Collaborators may add relationships between individual models, creating new integrated simulations.
**Configurator Role**

As DOME model interfaces become available, GUIs for them might be needed to facilitate user tasks. This is where the Configurator comes in. The Configurator is used to quickly generate the GUIs. Thus, user interface development is no longer a bottle-neck in the emergent development process.

To illustrate this, a couple simple GUIs were constructed in the Configurator for the glass-drop system model. The first GUI includes graphics from the CAD model as well as calculations from the velocity simulations. The picture below shows the Configurator being used to create this GUI.

Figure 15 Building a GUI in the Configurator.
This GUI is helpful for users who just want to monitor the effects of key design parameters on velocity calculations as the integrated system is being simulated. A shot of the running application is shown above.

Figure 16 Running the velocity GUI.

The second GUI created focused on helping users understand optimization issues. The graphical display of a genetic optimization object, implemented to automatically search for models states that best meet design goals is included in this GUI [14].
Constructing and deploying these two GUI's was a simple process with the Configurator.

After locating all the necessary parts, the work using the Configurator took less than an hour.
Discussion and Conclusions

Need for DOME Configurator

The purpose of the DOME project is to create tools to promote collaborative and emergent simulations. Most application studies have focused in product design, although DOME really is a general technology. It may be possible to apply DOME to areas such as interactive simulation games, financial modeling, or in eco-system analysis. Stephen Kraines has used DOME in chemical engineering studies in Tokyo [15]. The DOME core is powerful because it makes solving and understanding complex systems much faster than traditional methods. In some cases DOME can perform simulations that are previously intractable. In studies done at Ford and Polaroid [5], DOME usage could reduce the time of performing design assessments that usually required weeks or months in traditional environments to a matter of seconds.

Because a key benefit of DOME technology is the ability to reduce the time needed to accomplish milestones, it is in some sense a Rapid Application Development (RAD) tool. One part in providing RAD is to offer a GUI builder, or the Configurator. Because DOME based simulations are developed through an emergent process, when a particular GUI is needed cannot be pre-determined, the specifics of the GUI might be in flux, etc. As a result, it is important that GUI creation does not become the bottleneck in the process. Without the ability to quickly build GUIs, the emergent and rapid adaptation qualities of the DOME simulation would be hampered.
In integrated system modeling, problem oriented GUIs are often necessary. At EMPA Swiss Federal Laboratories for Materials Testing and Research, Markus Koschenz [16] found that developing a GUI builder for an integrated mathematical solver executing different batch applications was very useful. Instead of programming, in Koschenz’s demonstration, GUI building was almost like working on a piece of paper with handy numbers, text, graphs, and pictures available. The below is an example of GUI for a thermal materials model.

![GUI for a thermal materials model](image)

**Figure 19 GUI demos from EMPA.**

Likewise, users also often need to create special graphical interfaces to systems created with DOME. For example, in Ferara’s thesis DOME connected components were used to construct applets for use in website to support mass-customization of bicycles [17]. This application of DOME needed an interface that was unlike the a-priori design simulation experiments. Thus, it was realized that it was necessary to build a user tool to support DOME GUI construction. In the demo in the previous chapter, two GUIs were created for
the car door glass system model. The two GUIs supported different types of activity. The first GUI with the CAD geometry and engineering parameters facilitated simple monitoring the effect of design choices on the average drop velocity. The second GUI created included a tool that enabled users to find optimizations in the model. Provided that there is a library of useful GUI components, putting these GUIs together was not difficult in the Configurator.

**Contributions**

By enabling rapid GUI construction, the Configurator enhances the accessibility of DOME models. Better accessibility promotes the collaborative quality of the technology. With the ability to rapidly develop custom client interfaces, the accessibility of DOME models are increased because they are not limited to set interfaces that may only be usable to a small set of potential users. In the example GUIs, there could be two people looking at the simulation. One may only care about the velocity of the glass drop, while the other may be interesting in overall global characteristics of the system. By having two GUIs to the same simulation model, both users can be satisfied. Since, the Configurator is a visual programming tool, so users do not really have to understand how to write code in order to create their own interface. Thus, accessibility to DOME is not limited to those who know Java and have studied the API for DOME subscribing. The Configurator helps make DOME accessible to non-programmers. In the future, the Configurator may adopt the Java Internationalization to support other languages and promote global collaboration.
Multiple views are often very helpful in certain projects. For example, in a collaborative and emergent design process, it is important to be able to make visualizations of data relationships. Without this ability, the system understanding becomes intractable and problems are difficult to troubleshoot. In an emergent process, the data relationships are not predetermined and cannot always be generalized for all simulations. So it is not possible for a pre-made, general viewer to sufficiently display the nuances of the information being generated. Perhaps a new view can help in uncovering new insights about the system. Just as there are human specialists to solve parts of the simulation, custom views are needed to help simplify the complexity of the system. Usually, building a custom viewer application from scratch would take a good deal of time. The Configurator makes mass-customization of views possible, providing DOME users with the multiple views of the simulation.

Because the Configurator promotes the reuse of pre-made graphical components it makes it easier to combine information from many sources in the created GUIs. DOME simulations pull together information from different sources, MATLAB, CAD system, Excel, etc. The ability to create multiple views to display that information, combining graphics from different systems further extends DOME as an interoperability solution.

The Configurator and the GUI it produces works seamlessly with DOME technology. Network subscription service to DOME models is internally provided for in the Configurator. This is essential since DOME revolves around a web based communication.
The Configurator makes it easy to develop GUls that access DOME simulation services over the internet. A company can easily construct and distribute interfaces to their service to their clients, or the clients themselves can build their own custom interface to their supplier’s interface. As the parameters in the subscribed simulations alter, the changes automatically appear in the Configurator environment. GUI components connected to DOME parameters also are automatically made aware of the changes. Additionally, with the Configurator it is easy to update GUls as the simulation system is modified or as user needs become clearer. Seamless integration with DOME technology promotes the development of a world-wide simulation-web. It makes it possible for service providers to meet the need of many more clients because creating custom interfaces has been made more efficient.

**Limitations and Future Work**

The current Configurator supports a limited level of applications. GUI designers can use the Configurator to create simple applets or applications. The designs in the Configurator can be saved as components and reused later on. The nesting ability encourages component library development and further speeds future GUI development. However, the application areas of the Configurator are limited in complexity. In a trade off with usability, the Configurator currently does not explicitly support complex projects that require multi-window management or animation. If users show a need for more complicated functionality, the Configurator should be repackaged as a module in a full blown IDE such as NetBeans.
Currently, standard tool library in the Configurator is not very large. The abilities of the Configurator would be greatly enhanced by the addition of additional component beans. For instance, it would be very nice to have beans that could display, in real-time, CAD generated graphics and being able to update as the CAD model changes. Perhaps even some beans that provide simple geometry manipulations could be made.

Another interesting idea would be to create beans for the Configurator library that filters information for the users. The user would connect the filter to a collection of DOME parameters. The filter would observe the updates in the parameters, but only pass the information on to a visual component when the values fit within design specified constraints.

Libraries of beans that answer a category of simulation queries would probably be useful, and should be developed in the future. For example, perhaps a common question would be: What is the relationship between x and y? A bean that can plot the temporal trends of the x and y might help solve that question. Perhaps seeing the relative proportions of the two quantities in a pie chart might be elucidating. As the availability of pre-made GUI components grows, the usability of the Configurator would improve.

A tree viewer that shows the structure of the GUI being designed would be a nice addition. This would help users understand the component nesting in their project. The alternative view of the project would give users another way to select and move items.
Enhanced visual programming capabilities, such as the ability to define loops, simple functions, data flows with boolean operators, or debugging capabilities might be an interesting addition. The simple event and property hookups currently work and are easy to understand. But the expanded interaction vocabulary might make interaction definition more powerful and easier to do. Other visual programming tools that are not Java based have implemented the extended interaction set successfully. It may be possible to create such an abstraction with the bean based architecture.

A future code editor feature would also enhance the DOME Configurator, as Java savvy users can then add their own custom code. This would offer users the flexibility to create GUIs with special capabilities, while at the same time having the Configurator generate the basic structure of the GUI.

**Conclusion**

In the Information Age, the quantity of data people have access to has grown phenomenally. This is in part due to technologies like DOME that encourage and facilitate the sharing of information and the development of new data and ideas. As the diversity and size of the information set a person has to process increases, information visualization capabilities becomes increasingly important. Without good visualizations, the insights in the information can be easily missed by the human observer. DOME has the potential to model complex systems, collaborating the data and analysis capabilities of multiple sources. But the influx of information generated by DOME simulations cannot be visualized by a single, or even a few standard visual tools. Flexibility in the
ways information is expressed to users must be available. The Configurator addresses this problem with mass-customization of DOME enabled GUIs. Java technology was used to create a user-friendly visual programming environment, enabling the construction of component-based DOME GUIs. The Configurator reduces the time and resources needed to create a GUI, making it possible for almost every collaborator to have their own visual of the system.
List of References
