JBeanStudio: A Component-Oriented Visual Software Authoring System for a Problem Solving Environment – Supporting Exploratory Visualization –

Masahiro Takatsuka

School of IT, The University of Sydney, NSW 2006 AUSTRALIA
masa@it.usyd.edu.au, http://www.it.usyd.edu.au/∼masa/

Abstract. This paper discusses the benefits of a Component-Oriented visual software authoring system that provides the seamless integration of various software tools in a unified environment. It employs a visual component assembly paradigm for ease of construction, Java™and JavaBeans™component architecture for the open environment, and recursive development methods, all of which allow us to rapidly construct and share applications. Moreover, it is highly interactive and fully configurable in order to support exploratory visualization. This versatility has the potential to improve the integration of independently developed analysis tools and the dissemination of research findings.

1 Introduction

We have all witnessed the phenomenal technological progress in the fields of electronics engineering, and computer science[1][2]. This dramatic improvement has led many computer scientists and engineers to ever challenging problems involving supercomputing, parallel and distributed technologies, and much more complex algorithms that use these high-performance computing technologies. Moreover, the advances in other engineering fields such as aerospace and mechanical [3][4], has changed how massive scientific data are collected and processed. Computer scientists and Software Engineers are now in great demand for producing various types of computational tools to process and analyze these data.

1.1 Diversity of Computational Tools

Diverse groups of researchers have been building computational tools in order to support such datasets and computational demands for more sophisticated data analyzes and problem solving.

These tools are written in a variety of programming languages (C, C++, Visual Basic, FORTRAN, and Java), and also developed on different platforms (UNIX, Windows/Dos, Mac, etc.). The form of their deployment also varies.

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2659, pp. 985–994, 2003.

Some tools are distributed as programming libraries and some are deployed as standalone applications. This often constitutes a burden for the end users. If a user is fortunate enough to have all his/her tools in library form and in the same environment, the user can write his/her own code to integrate various functions provided by the tools. If some of the tools are standalone applications, which require different platforms, a user has to move in and out of different environments to execute each tool (application). Some tool producers have chosen to use cross platform programming languages like Java™ to avoid the intertwist of heterogeneous environments. However, these cross platform tools do still demand that a user integrates them at the coding level.

1.2 Need for Better Integration

Some generic mathematical and computational programming environments (such as Mathematica[1] and Matlab[2]) offer smooth integration by providing an infrastructure into which independently developed modules can easily be incorporated. Those environments, however, still require manual coding to integrate tools.

Problem Solving Environments (PSEs) have been used to address the issue presented above by providing application development systems that support the flexible deployment of analysis modules, often with a visual programming interface [5]. For instance, National Instruments' LabView[6], HP's visual programming tool: VEE[7]), OpenDX[3], IRIX Explorer[8] and AVS[9] provide a visualization-centered PSEs via visual programming interfaces. They also allow users to create their own computational/visualization modules using an application's framework. Most of the current solutions to seamless integration rely on the use of proprietary interfaces or frameworks. Independent development and deployment of tools are possible in those infrastructures but only within the particular application environment. So, for example, some analysis tools developed for AVS cannot be smoothly integrated with tools from OpenDX without modifications.

Moreover, many such environments are less flexible in configuring how data and pieces of information are exchanged. Many PSEs are interactive and configurable in the sense that a user can combine various tools in accordance with their needs. In visualization-oriented PSE, however, more flexibility should be provided at a lower level than at the level of just selecting tools and their configuration. At the lower level, a user would be able to interactively configure how data and pieces of information should be exchanged between tools. In many PSEs, tools are combined by strongly-typed functions/methods. Tools usually exchange numerical/nominal values and more complex structured data of the same type. In order to encourage interactive exploratory visualization, the process of configuring how each piece of data is mapped onto a visual variable needs

[1] http://www.wolfram.com
[2] http://www.mathworks.com
[3] http://www.opendx.org

to be provided. This is because users often do not know how best to visually represent the given data.

This project aims to provide an interactive and fully configurable visual software authoring system based on open-standards so that independently developed and deployable analysis and visualization components can be seamlessly integrated in a non-proprietary environment.

2 JBeanStudio

JBeanStudio is a component-oriented software authoring system. Users utilize its visual environment to rapidly wire components together into a useful application. It is called an *authoring* system because it does not generate code but rather generates running software on the fly. The application is always "live" while it is designed, so a design can always be changed as a consequence of using the constructed application and evaluating the results produced. Such seamless integration of design and runtime environments enhances productivity since it allows rapid adaptation of the program in accordance with new analysis requirements. It fully utilizes the unique characteristic of a component:

> "... a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." Szyperski and Pfister, 1997[10]

The feature *independent deployment*, or more specifically, *runtime deployment* is the key feature which needs to be exploited. If components could be imported into a system (which an end user is using) at the runtime, the user would not need to go through the coding and compiling processes in order to enjoy the advantages of the components. Consequently, it allows highly interactive and customizable program development. This is important since there are often as yet no rules as to how best to connect or configure components in order to perform the best problem solving tasks.

JBeanStudio offers:

1. Ease of use, but with the capability to construct complex analysis applications,
2. Rapid development and modification of applications, minimizing programming requirements,
3. Support for sharing and exchanging of developed applications.

The first two features allow many computational and other scientists to focus on actual problems and not on the underlying programming logic that provides these tools and facilitates their dynamic interactions. They are therefore free to fully utilize their domain knowledge. The third feature is even more important for peer review (replication of research results) and dissemination of research findings[11].

2.1 System Architecture

JBeanStudio is built around the well-established, open standard Java™programming language[12] and JavaBeans™component architecture[13] forming a layered model as shown in Figure 1. There are a few technologies which can be used to build component-oriented systems such as OMG's CORBA and Microsoft's .NET. However, Java has been accepted in various fields due

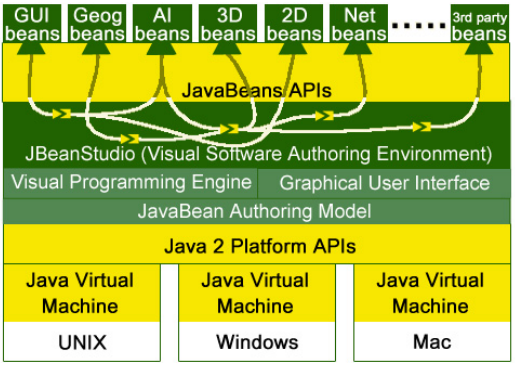


Fig. 1. System Architecture

to its many benefits, such as cross platform support (the bottom layer in Figure 1) and ease of use, as evidenced by the growing transition from C++ to Java. JavaBeans technology provides a standard Application Programming Interface(API) to make reusable components in the Java programming language. The combination of Java and JavaBeans provides better support for Component-Oriented Programming (COP). It is known that C++ does not directly support the concept of COP[14].

A program in JBeanStudio is constructed from building blocks - beans - (see the connections made at the layer of the visual authoring environment in Figure 1). By combining many components, more complex programs can be developed. As long as beans are created according to the JavaBeans standard, one can assume that they will work together with other beans created by different software vendors or individuals (see the top layer of Figure 1).

There are few applications similar to JBeanStudio. Sun Microsystems' BeanBox and BeanBuilder [15] are examples utilizing standard Java and JavaBeans component architecture. However, they are only a reference implementation to illustrate the possibilities of visual programming using Java and JavaBeans component architecture. They present various shortcomings to be addressed in order to fully support exploratory visualization and probably data analyses. Some of those problems are:

1. information/data exchanges between components are strictly via Java's event model. This means a component has to pass some pieces of information in the form of an event object. This implies that a special event needs to be created for each information which needs to be passed.
2. only subclasses of a particular Java component (java.awt.Component) can fully utilize the Java's event model in such an environment. In other words, if an event originates a Java software component, which is not a subclass of the java.awt.Component, the system fails to properly recognize the event dispatcher and listener relationships.
3. when a pieces of information is passed onto another component, the data type has to match between a sender and a receiver. For example, an integer value cannot be sent to a method (function) which takes a real number as its argument.
4. all events are handled by a single thread. It is almost impossible to create an application with loops.

Other visualization-oriented visual programming environments like AVS, IRIS Explorer and OpenDX provide excellent Problem Solving Environments and programming-based tools like Visualization Tool Kit (VTK)[16], but each of them forms its own proprietary environment making it difficult for components from different environments to work together.

3 Component Wiring and Dataflow

There are two types of dataflow models common to the above mentioned systems: 1) event-driven and 2) demand-driven. Most visual programming based systems use event-driven dataflow. This approach naturally follows the cause-effect paradigm and provides a very natural component wiring mechanism to users [17]. The demand-driven approach is often used when precise program control of how to execute branching and conditional execution is needed[18].

JBeanStudio aims to provide a more complete visual software authoring system using a hybrid approach (event-driven demand execution). This approach addresses the previously mentioned shortcomings by providing:

1. five types of information passing mechanisms - using 1) a standard Java event object, 2) event object's accessor method, 3) event dispatcher's accessor methods, 4) accessor methods of another object, which is not an event dispatcher, and 5) a constant value. This means that the event receiver uses the event as a simple trigger to obtain extra pieces of information from various sources.
2. a generic event handling - any JavaBean component can dispatch and receive events. A component does not need to be a subclass of a particular object or to use proprietary interfaces. This means any component can send and receive a message via Java's event model.
3. a data converter manager - it runs in the background and automatically converts one data type to another when an appropriate data converting

module is present in the system. It provides a basic set of data converter modules but a user can register any kind of customized data converters at the run-time in order to support custom data types.

4. multi-threading - each event dispatch-receive connection is executed in a separate thread. Hence, the execution of a loop and branching control will not halt or delay executions of other parts of an application.

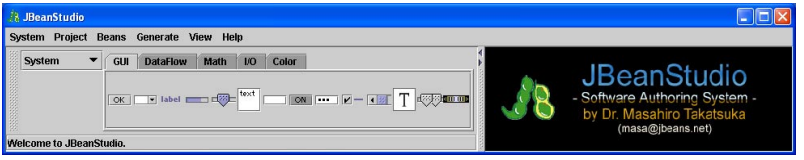
4 Interactively Building a Visualization Application

JBeanStudio was designed to unify component-assembly and runtime environments. When JBeanStudio is first launched, it presents its main window with various menus and bundled components (see Figure 2(a)) along with an empty design and GUI windows.

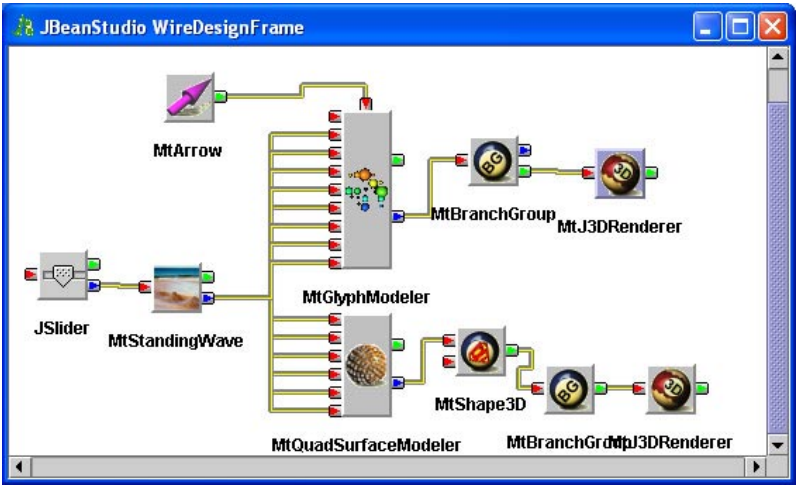
Each component is stored in a folder of a component palette. A user can customize and create folders and palettes as well as load palettes from XML files. This feature is useful when customized components and/or groups of components need to be shared among colleagues. It also allows a user to import any JavaBeans components developed by third parties.

In JBeanStudio, a user designs an application by dragging components from the palettes and dropping them onto either the design or GUI window. Once components are placed in the design they can be customized and wired together as shown in Figure 2(b). Due to the visual assembly mechanism, no programming is required; in fact code writing is not supported (the user can write the code for beans outside of JBeanStudio). This makes it different from other development tools in which a user has to write some code to produce software components, applets or applications; hence our users (especially non-programmers) should find development less burdensome. The design of the application is saved using XML. This application is then ready to be distributed over the net. Since the whole design information is included in the saved design file, the other users can improve and/or customize the application further to suit their own needs.

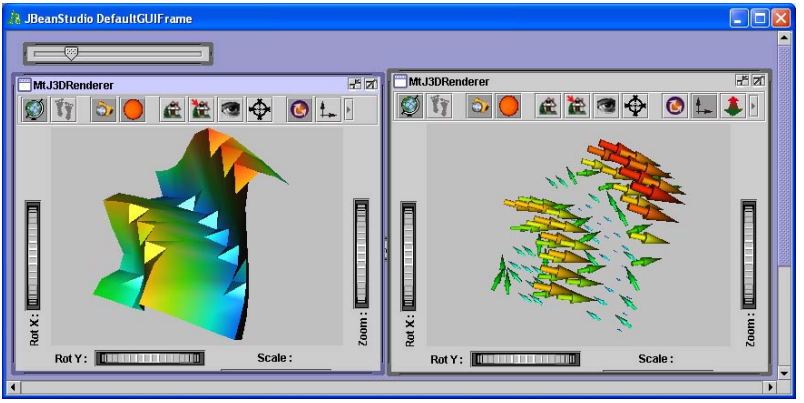
As described before, JBeanStudio integrates the environments for building and running a program and an application is always running while it is being designed (see Figure 2(c)). Hence, a user can adopt the design in it as a final working program. However, the designer of the program might need to package the design into an applet or a standalone application as a customized product for other users. Alternatively, the user might want to package the design into a JavaBeans component for inclusion in a larger application. To support creation of Java beans, applets and applications, it provides functionality that does all of the source code generation and compiling work. This is the only function in JBeanStudio involving code generation. It automatically generates Java source code for the designed application, compiles it and packages it up into an archived file for distribution.



(a) Components are organized in the palettes and folders.



(b) Components are wired together in the design panel.



(c) GUIs of components are displayed in this panel while an application is designed and executed.

Fig. 2. JBeanStudio's Main, Design and GUI panels.

4.1 Example: Exploratory Visualization

The example shown in Figure 2 illustrates how JBeanStudio provides an interactive exploratory visualization system. The application in the example simply visualizes a dynamically changing standing wave. The component named MtStandingWave generates a standing wave in an $M \times N$ grid format according to the value of a slider bar (named JSlider). Upon generating a new standing wave, MtStandingWave sends an event to other components (MtGlyphModeler and MtGridSurfaceModeler). When those two components receive the event, they invoke methods from other data sources. MtGridSurfaceModeler, for example, has six input methods, each of which is used to compute a grid surface with colors (x and y dimensions, (x, y, z) coordinates and color of each grid). When those input methods receive a branched out event from MtStandingWave, MtGridSurfaceModeler component invokes various methods in MtStandingWave to retrieve various data for each input method. After obtaining all the information needed, MtGridSurfaceModeler generates a synthetic grid surface and send it to a 3-D renderer component. It should be noted that the source of the demanded information does not have to be the event source. In other words, MtGridSurfaceModeler can invoke a method to obtain, say, the color information for each grid from a different component. In JBeanStudio, the event can be used as a data transfer mechanism as well as just a triggering signal.

MtGlyphModeler receives the same event from MtStandingWave. It, however, places glyphs specified by a glyph source (MtArrow component in this case) at each grid point of the standing wave. Moreover, the size, color and orientation of each arrow can be controlled by some numerical values. In the example, color and z (height) values of the wave were fed into MtGlyphModeler's method from MtStandingWave in order to compute the orientation of arrow glyphs. The generated surface and the arrow glyphs are shown in Figure 2(c). While this simple visualization system is displaying the dynamically created surface and a set of arrow glyphs, one can re-configure how events are passed around and how demanded pieces of information are obtained. In other words, while a user is observing the produced images, the user can re-assign various numerical values to different visual variables (orientation of arrow, size of cone head and body, color, etc.) in order to obtain the best possible visual representation of his/her data.

5 Limitations and Future Works

JBeanStudio provides a visual programming environment and the flexibility and interactivity of being able to build an exploratory visualization system. However, since all data/information exchanges need to be visually specified, a novice user, who might have no experience in programming, might find JBeanStudio still difficult to use (program). However, the aim of the very first stage of JBeanStudio development is to provide a core visual authoring environment, which can be used to build a further higher-level framework. Hence, a team of users can develop a framework which defines a basic information exchange mechanism so that the

connections that need to be made between multiple software components can be minimized. The approach of building an extra framework on top of JBeanStudio would be useful if a set of components needs to be closely coordinated.

Moreover, it currently does not have any component layout algorithms. Therefore, the design can be very cluttered and hard to read once it gets large and complicated. The use of an automatic clustered graph layout algorithm [19] is now under investigation in order to improve the user interface of JBeanStudio.

6 Conclusion

This paper describes the use of JBeanStudio as one example utilizing an open component-oriented infrastructure that provides seamless integration of various computational tools. It offers an easy-to-use, visual assembly environment that supports rapid construction of sophisticated software. This allows scientists and engineers to concentrate on solving their domain problems rather than dealing with programming.

JBeanStudio employs Java-based component architecture. Any JavaBeans components can be wired together in this environment to form a larger application. Those components can be independently developed without being concerned with the integration environment (JBeanStudio). Moreover, it is capable of automatically producing a JavaBeans component (which can also be a Java applet or an application) from a design. This allows a user to create a reusable and scalable system as well as to distribute and share the created design on the Web. The flexibility of being able to customize an application and the seamless transition from a desktop data analysis/visualization program to a Web application allows scientists to freely apply and experiment with their knowledge in scientific analyses and share results instantly and accurately.

References

1. Ceruzzi, P.E.: A History of Modern Computing. MIT Press, Cambridge, Mass. (1998)
2. Geppert, L.: The 100-million transistor IC. IEEE Spectrum **36** (1999) 22–24
3. NASA: TERRA: The EOS flagship. http://terra.nasa.gov (1999)
4. European Space Agency (ESA): Envisat. http://envisat.esa.int (2002)
5. Rice, J.R., Boisvert, R.F.: From scientific software libraries to problem-solving environments. IEEE Computational Science and Engineering 3 (1996) 44–53
6. Baroth, E.C., Hartsough, C., Johnsen, L., McGregor, J., Powell-Meeks, M., Walsh, A., Wells, G., Chazanoff, S., Brunzie, T.: A survey of data acquisition and analysis software tools, part. Evaluation Engineering Magazine (1993) 128–140
7. Hewlett Packard: VEE: Visual engineering environment. technical data 5091–1142EN, Hewlett Packard (1991)
8. Numerical Algorithms Group (NAG): Iris explorer user's guide. http://www.nag.co.uk/visual/IE/iecbb/DOC/html/nt-ieug5-0.htm (2000)
9. Systems, A.V.: Information visualization : Visual interfaces for decision support systems. http://www.avs.com/ (2002)

10. Szyperski, C., Pfister, C.: Workshop on component-oriented programming, summary. In Mühlhäuser, M., ed.: Special Issues in Object-Oriented Programming – ECOOP96 Workshop Reader. dpunkt Verlag, Heidelberg (1997)
11. Takatsuka, M., Gahegan, M.: Sharing exploratory geospatial analysis and decision making using GeoVISTA studio: From a desctop to the web. the Journal of Geographical Information and Decision Analysis (JGIDA) 5 (2001) 129–139
12. Joy, B., Steele, G., Gosling, J., Bracha, G.: The Java^{TM} Language Specification. Second edition edn. The Java Series. Addison-Wesley Pub Co. (2000)
13. Sun Microsystems Inc.: The javabeans^{TM} 1.01 specification (1997)
14. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. ACM Press and Addison-Wesley, New York (1999)
15. Microsystems, S.: BeanBox and BeanBuilder. http://java.sun.com/products/javabeans/software/ (2002)
16. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit: An Object-Oriented Approach to 3-D Graphics. 2nd edn. Prentice Hall PTR (1998)
17. Abram, G., Treinish, L.A.: An extended data-flow architecture for data analysis and visualization. In: Proceedings of the 1996 IBM Visualization Data Explorer Symposium, CA, IBM, IBM Corporation (1996) http://www.research.ibm.com/dx/proceedings/proc.htm.
18. Schroeder, W.J., Martin, K.M., Lorensen, W.E.: The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In: Proceedigns of Seventh Annual IEEE Visualization '96, IEEE Computer Society, IEEE Computer Society (1996) 93–100
19. Pulo, K., Takatsuka, M.: Inclusion tree layout convention: An empirical investigation. In Pattison, T., Thomas, B., eds.: Volume 24 – Information Visualization 2003, Australian Computer Society, Australian Computer Society (2003) 27–35