Symphony: A Java-based Composition and Manipulation Framework for Distributed Legacy Resources

by

Ashish Bimalkumar Shah

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

APPROVED:

Dennis Kafura, Chairman

Calvin Ribbens

Clifford Shaffer

30th March, 1998

Blacksburg, Virginia

Symphony: A Java-based Composition and Manipulation Framework for Distributed Legacy Resources

by
Ashish Bimalkumar Shah

Committee Chairman: Dennis Kafura
Computer Science and Applications

(ABSTRACT)

A problem solving environment (PSE) provides all computational facilities necessary for solving a target class of problems efficiently. PSEs are used primarily for domain-specific problem-solving in science and engineering and aim to ease the burden of advanced scientific computing. Scientific problem solving, however, often involves the use of legacy resources which are difficult to modify or port, and may be distributed on different machines. Existing PSEs provide little support for solving such problems in a generic framework.

This thesis investigates the design of a platform-independent system that enables problem solving using legacy resources without having to modify legacy code. It presents Symphony, an open and extensible Java-based framework for composition and manipulation of distributed legacy resources. Symphony allows users to compose visually a collection of programs and data by specifying data-flow relationships among them and provides a client/server framework for transparently executing the composed application. Additionally, the framework is web-aware and helps integrate web-based resources with legacy resources. It also enables programmers to provide a graphical interface to legacy applications and to write visualization components.

Symphony uses Sun Microsystems' JavaBeans component architecture for providing components that represent legacy resources. These components can be customized and composed in any standard JavaBeans builder tool. Executable components communicate with a server, implemented using Java Remote Method Invocation mechanism, for executing remote legacy applications. Symphony enables extensibility by providing abstract components which can be extended by implementing simple interfaces. Beans implemented from the abstract beans can act as data producers, consumers or filters.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support and encouragement I received from several people during my stay at Virginia Tech. First of all, I am thankful to my adviser, Dr. Dennis Kafura, for accepting me as a student and for his constant encouragement, persistence, and excellent advice on every aspect of my work. Also, every time he saw me wavering from my decision to pursue this work, he gave me strength to believe that what I was doing was really important. I am glad I heeded his advice.

I would like to thank Dr. Calvin Ribbens and Dr. Clifford Shaffer for agreeing to serve on my committee and for giving me useful feedback on the particulars of my work. Many thanks go to Dr. Nicholas Stone, Director of the Agricultural and Natural Resources Information Systems group who supported my education and stay here through a research assistantship, for most of my terms at Virginia Tech. I would not have been able to undertake this work without these funds. I would also like to thank Dr. Marc Abrams for giving me the opportunity to work with him on the publication of the online book "WWW: Beyond the Basics". The knowledge I gained from my work on the book laid the theoretical foundation for my thesis work. Thanks to Dr. Verna Schuetz, my initial adviser in the department, for always being so nice and friendly.

I would like to acknowledge the emotional support I received from several of my friends at Virginia Tech, notably Govi, Prabhu, Raza, Hema, Mridu, Vijay, Karthik, Sunil, Mei See, Jady, Amit, Jose and Nandu. Many thanks to Siva Challa who has been a constant source of encouragement and brotherly advice. My first roommates in Blacksburg, Yash and Milap, deserve special mention for being so nice to me and for patiently answering the barrage of questions I used to fire at them.

And last but most important of all, I would like to express my deepest love and respect for my parents, brother and other family members for their constant encouragement, support and guidance. My parents have always taught me to shoot for the impossible. I hope I can fulfill their wishes some day and bring as much joy to them as they do to me just being my parents.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

A **Problem Solving Environment (PSE)** can be defined as a computer system that provides all computational facilities necessary to solve a target class of problems efficiently. The term Problem Solving Environment has a very broad meaning, possibly including word processing software, which can be viewed as a PSE for formatting documents, as well as a system for assisting engineers solving various types of partial differential equations. Some properties shared by all PSEs are that they allow a user to formulate a problem solution in a language suitable for the target class of problems and to view or assess the correctness of the solution through analysis or visualization tools [11].

Depending on the problem domain, different features are desired in a PSE. Some of these features are:

- Collaboration - Allow multiple users to simultaneously take part in the problem-solving session
- Integration - Hide heterogeneity of individual problem-solving components
- Persistence - Allow saving and reproducing of problem-solving sessions
- Distribution - Handle local as well as remote computational tasks
- Security - Provide user and server security
- Intelligence - Make automatic or semi-automatic selection of solution methods by consulting an associated knowledge base

However, the field of problem solving environments is a relatively new discipline of computer science and the general understanding of the architecture, technology and methodologies for PSEs is still immature. In fact, hardly any existing PSE or PSE-like system includes many of the features described above.

Problem solving environments have predominantly focused on science and engineering applications [1, 17]. In this thesis too, the term PSE will be interpreted with this application domain in mind. A generally accepted goal for a scientific PSE is that it should ease the burden of advanced scientific computing and should enable more people to solve problems more rapidly without requiring detailed knowledge of the underlying hardware, software, or algorithms, although knowledge about the specific problem domain addressed by the PSE is always required. The need for a PSE increases with the complexity and heterogeneity of the application.

1.1 Problem Statement

Most existing PSEs are focused on providing problem-solving facilities for narrow application domains, such as solving partial differential equations (PDEs), data visualization, numerical analysis and others [24]. These PSEs are built around software libraries which are either modified or rewritten to adapt to the architecture of the PSE. Although these PSEs function very well in their own domain, they do not attempt to provide a generic framework for solving general-purpose science and engineering problems. In practice, many such problems involve the use of legacy software which is difficult to modify and/or port and may be distributed on geographically distant machines. Existing PSEs provide little support for solving such problems within a generic framework.

The specific shortcomings of the current implementation practice for PSEs for science and engineering, that this research aims to address are as follows:

1. **Lack of support for legacy software:** Most scientific PSEs provide little support for stand-alone legacy software applications. These are applications which are run from the command-line, have limited user interaction, and communicate using specially formatted files. Support for legacy codes is extremely important because there exists millions of lines of legacy code, most of it difficult to understand and modify, yet very useful.

   There seem to be two main reasons for this drawback. First, scientific and engineering PSEs are generally built around software libraries which provide encapsulated problem-solving power for some particular problem-domain. Thus the architecture of a PSE is inextricably linked to the structure of the underlying software library. Second, PSEs are generally built for a particular platform and the PSE software is typically not platform independent. These reasons, in the context of providing support for legacy applications, basically entail modifying the application or porting it to a different platform.

   Rewriting legacy code to fit the architecture of the PSE or porting it to the platform supported by the PSE is not a feasible solution because of several problems. First,

legacy code is usually difficult to understand and modify and the cost involved in such an attempt could be quite high. Second, the underlying software or hardware facilities assumed by the application may not be available on the particular platform for which the PSE has been developed. Finally, if the performance of the legacy application has been tuned to a particular type of architecture, porting it to a different architecture may take the performance advantages away.

2. **Inability to easily compose distributed components:** Most PSEs do not allow the integration of programs and data distributed on different machines. Given a set of legacy scientific computing resources developed by a diverse group of people on different platforms (possibly located in different geographical locations) an environment is needed for constructing integrated applications out of these resources. For example, the design of a modern aircraft requires the use of numerous, perhaps tens or hundreds, separate programs; such multidisciplinary design and optimization requires a much higher level of integration than is available in existing PSEs.

   Composition of distributed resources is becoming increasingly important with the growth of the World Wide Web (WWW, Web). There are scores of applications on the Web which can be accessed at the click of a button (e.g., Java applets and servlets, CGI applications, and Web wrappers for legacy applications), but there is no single tool which can provide seamless integration of these Web-based applications with other legacy applications. There is also a need for an environment where legacy applications can be provided a graphical user interface for accepting input data and seamlessly integrated with analysis and visualization tools for processing the results.

3. **Lack of portability:** Very few existing PSEs are built around a client/server architecture and there is no clean separation of the PSE client interface from the the server-based functionality. Hence for making the PSE available on another platform, the entire PSE software must be ported, instead of just the client functionality, as for a client/server system.

   A problem solving environment is a complex system by nature and porting the entire PSE software to some other machine or even just installing a copy of the software on another machine may be a tedious task. Consider the example of Parallel ELLPACK (//ELLPACK), which is a problem solving environment for partial differential equations (PDEs) [12]. The //ELLPACK system consists of about one million lines of C, Lisp, and Fortran code. It's easy to see how complex it must be just to install a copy of the PSE on a new machine. If a system like this were to be built around a client/server architecture, only the client functionality would need to be ported to other platforms, the code for which would be only a small percent of the entire PSE software.

   These considerations, in general, limit the availability of the PSE to platforms for which they are developed and many times, to the user being present at the particular machine on which the system is installed. There is need for a PSE architecture that follows the

client/server model and where problem specification and analysis of solution can be decoupled from the task of producing a solution.

This research tries to address the above issues, either partially or completely. The specific goals for the research are outlined in the next section.

1.2 Goals

The primary goal of this research is to develop a platform-independent framework for specifying and transparently executing compositions of distributed resources, including legacy resources. This framework should provide an ability to visually compose a collection of distributed program codes, data, and visualization components by specifying data-flow relationships among them. It should also provide an ability to execute the composed application in a manner that respects the data-flow requirements of individual programs in the composition. Execution transparency in this context means that all system level operations of program execution and of moving data across geographically distributed locations must be largely transparent to the user. By employing a client/server model, the composing environment should be made independent of the underlying architecture assumed by the legacy resources being composed.

The composing environment and execution framework should also have the following additional features:

- **Extensible:** Ability to extend the framework with a minimum amount of work
- **Open:** Based on publicly available and community supported standards
- **Generic:** Independent of a specific application
- **Web-aware:** Ability to accommodate data and programs that are accessible on the Web.
- **Persistent:** Ability to save, reproduce, annotate and execute the composition at any site, regardless of when or where it was originally built
- **Secure:** Provide necessary system security required for execution of remote applications
- **Graphical:** Support for creation of graphical interfaces for soliciting input data for legacy applications from the user and support for creation of graphical components for visualizing the output data from an executable component

Although the goals for the proposed system stem from the scientific problem solving environment perspective, the system can be used for visually composing and executing any set of distributed legacy resources outside of the context of a PSE. The goal of providing a framework for solving a set of problems which are most relevant to the scientific problem solving environment community does not make the system any less general or applicable to other related domains or for purposes that are not viewed as "problem-solving." We do not contend that all of the features described above must be a part of every scientific problem solving environment; some of them facilitate extra capabilities and improve ease of use if present.

1.3 Approach

A component is a self-contained reusable software object that is not bound to a particular program or implementation. It does not constitute a complete application by itself. Cheap, personalized applications can be built by composing and customizing generic, "off-the-shelf" components. The software framework that allows composition and manipulation of these components is called a component architecture. Different components interact using standard client/server interaction models such as event notifications [15].

The goal of developing a composing environment for building collections of distributed legacy resources fits naturally with the paradigm of composing and customizing re-usable software components, where each different type of legacy resource is represented by a separate software component. The Java programming language [2] and the JavaBeans component architecture [33] were used to implement the desired system. Sun's JavaBeans specification is an Application Programming Interface (API) that enables developers to write components called **beans** in Java. Platform independence comes as an added benefit of using Java which is an architecture neutral programming language.

Since JavaBeans is an open, published API and is supported by a large number of Java development tools and Java runtime environments, beans that conform to the API can be composed and manipulated within any such beans container. Thus, the system can be used on any platform for which a Java beans container is available.

The data-flow paradigm was chosen as a way of describing relationships between components and specifying the execution sequence of related executable components. This paradigm has been popularized by visualization systems such as AVS [20] and Khoros [27]. A visual program is described as a directed graph, where each node represents an operator or function and each directed arc represents a path over which data flows. The environment provides a workspace where resource modules may be instantiated, connected, and customized to form the data-flow network.

Some PSEs also allow users to create a problem description in terms of the various tools used during the problem-solving process and corresponding data and control flow patterns which

link the tools [24]. Such an integrated collection of tools may be defined as a **meta-program**. Formally, a meta-program is a set of linked program and data components implemented as a data-flow graph that defines how each program accepts data from previous computation and produces data for further processing. Once a meta-program is built, it should be possible to ensure its structural integrity and completeness, save it for future reuse, or to execute it from the workspace.

1.4 Symphony

Symphony is a Java-based framework for composing and manipulating distributed legacy resources. The framework consists of two parts: client components that represent data, executable resources, and visualization tools which are used for composing meta-programs and the Symphony server which is needed for executing remote legacy applications. The client components are implemented as Java beans and the Symphony server is implemented as a remote object on which client beans make remote method invocations for obtaining services. Also implemented are beans that can be extended to add new types of beans to the environment by implementing simple interfaces. Utility beans, such as an annotation bean, are also implemented.

The following is a list of all the beans implemented and a short description of their purpose, along with an example of how they may be connected together into a meta-program (Figure 1.1):

- **Program Bean:** This bean represents a local or remote executable resource. Based on the location, input-output requirements, and the manner in which it can be accessed, the program bean represents two broad categories of programs: HTTP-accessible programs such as CGI scripts, and regular command-line executables.
- **File Bean:** This bean represents a local or remote data file used as an input to a program or produced as output. A file can be an HTTP accessible file, an anonymous FTP accessible file, or a private user accessible file on any machine connected to the Internet.
- **Socket Bean:** A socket bean encapsulates an input or output stream for communicating through TCP/IP sockets.
- **Standard Stream Beans:** There are three different beans representing the standard streams of a program: standard input, standard output, and standard error. A standard input bean provides a way of redirecting data into a program's standard input stream. The standard output and error beans provide means of redirecting the respective streams from a program to other beans for processing.

- **Producer Beans:** A producer bean is an abstract bean that can be extended by implementing a simple interface to define new beans types that act as producers of data. One Symphony bean that has been implemented by extending the Producer bean is a Parameters bean. The Parameters bean reads, from a URL, a textual description of the set of parameters expected by a legacy program, and creates a graphical interface to solicit those parameters from the user. The parameters entered by the user are passed onto the next bean in the sequence for further processing during execution.

- **Consumer Beans:** This is an abstract bean which is useful for implementing new beans that act as consumers of data, e.g., visualization beans and viewer beans. Symphony beans that have been implemented using this bean are a FileViewer bean which displays a text file in a window, and a WireFrame bean which reads a stream of specially formatted data and creates a rotatable 3D wireframe graph.

- **Filter Beans:** The filter bean is an abstract bean that allows the user to implement different kinds of beans for filtering the data flowing through the system. The simplest filters can be text filters analogous to the Unix filters. More complex filters include image processing filters and file format converters.

- **Annotations Bean:** This is a bean that allows the user to add annotations to the meta-program being constructed. Annotations can be added and viewed at any time.

- **Properties Bean:** This bean represents common properties such as remote host name, user name, password, etc., that are read by all other beans in the environment for customization. This is a utility bean that decreases the amount of work the user has to do for customizing the beans in a meta-program. The user customizes the properties in the properties bean and the values are propagated to all the other beans which have these properties.

Figure 1.1 shows how some of the above-mentioned beans can be composed to form a meta-program. Each Program bean can be connected to a set of input and output beans. For example, the RFPInput bean is actually a Producer bean which solicits parameters from the user during execution. Data from this bean is redirected to the standard input stream of Program1 which creates the file represented by the File1 bean when executed. Program2 takes input from the file represented by the File1 bean and also on its standard input. It creates files represented by beans File3 and File4 and the standard output from Program2 is redirected to File2. After File3 is created by the program represented by the Program2 bean it is read by the WireFrame bean which creates a 3D wireframe graph from the file data. The Annotations bean is used to add time-stamped annotations to the meta-program.

The Symphony server is a daemon process running on all host machines which serve programs to remote builder clients. The server is only needed for executing remote programs, not for accessing remote files. It is written in Java for portability. Client program beans

communicate with application servers on various hosts by making remote method invocations (RMI) on objects residing in the server. Figure 1.2 shows the general architecture of the Symphony system.

Finally, the name Symphony is representative of the fact that constructing meta-programs that provide access to distributed resources is similar to composing a complex and harmonious musical piece. The user who acts as composer, director, as well as audience, composes the musical score and, hopefully, appreciates the results.

1.5 Organization

The body of this thesis is organized as follows. Chapter 2 begins by discussing related systems and technologies and how Symphony compares or differs with these. In Chapter 3, Symphony is described from a user-perspective and targets two classes of users: those who are just interested in learning how to use the system for building and using meta-programs, and programmers who are interested in extending the set of Symphony beans by using the abstract beans. Chapter 3 also describes a real-world example of the application of Symphony for solving a science and engineering problem involving distributed legacy resources. Chapter 4 describes the implementation of Symphony in detail and targets readers who wish to extend the set of core and abstract Symphony beans. It also leads the reader through an explanation of the JavaBeans architecture and the Java RMI mechanism, which form the basic building blocks of Symphony, before delving into the details of the Symphony architecture. Chapter 5 concludes with an evaluation of Symphony in terms of its contributions to the field of scientific problem solving. It also identifies limitations and possible future work based on these limitations.

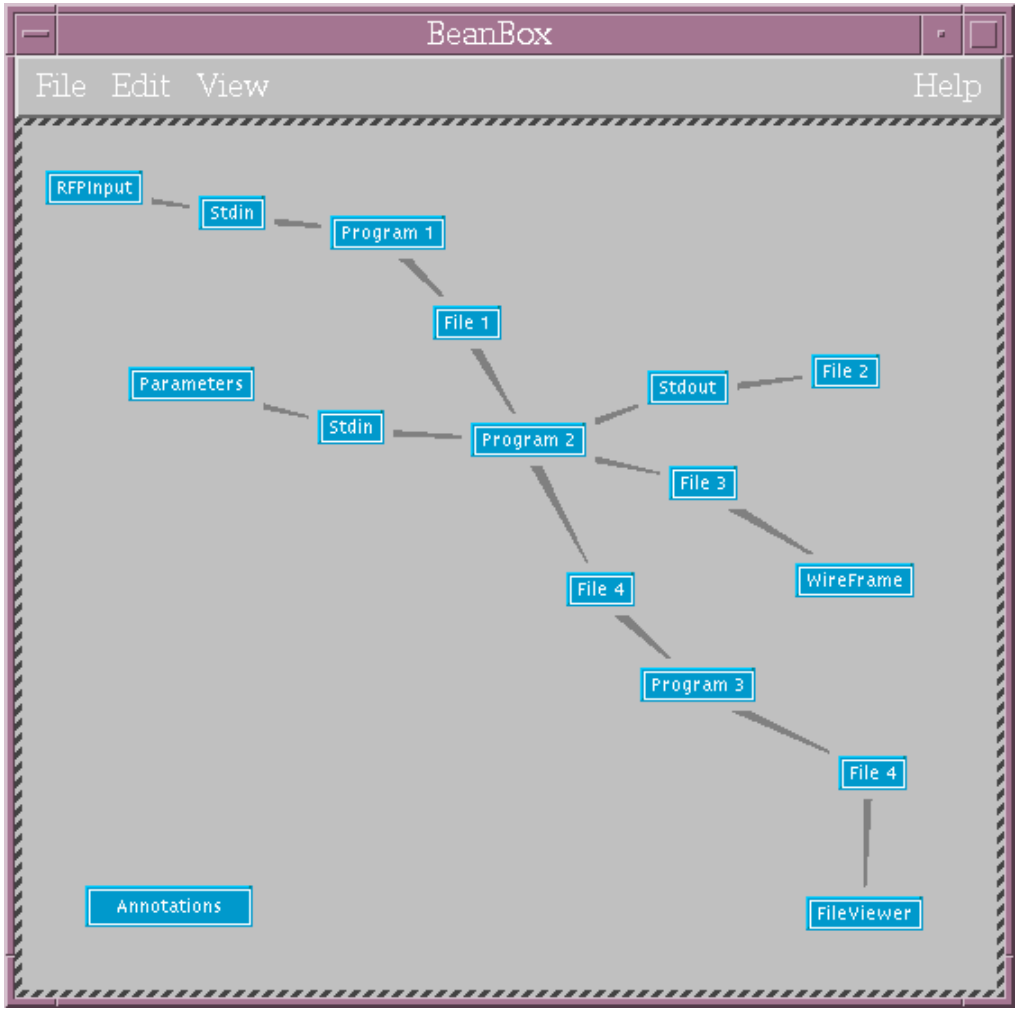

Figure 1.1: The Composing Environment for Meta-programs

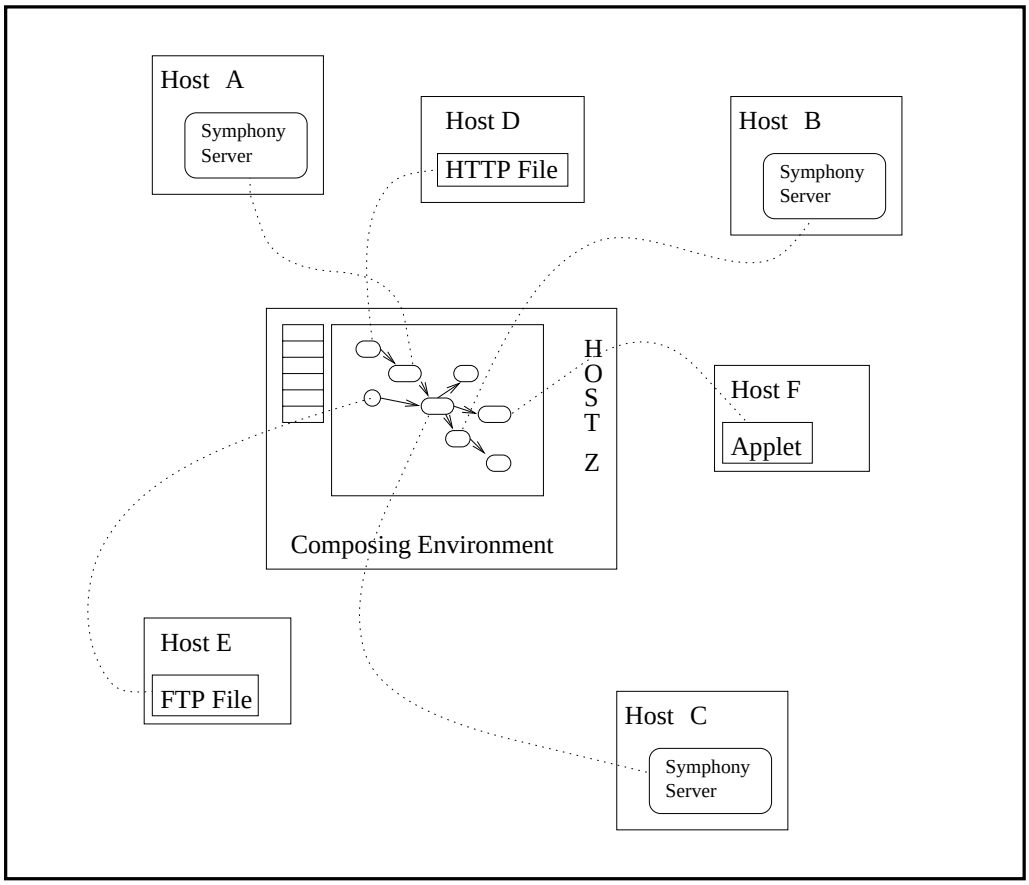

Figure 1.2: Symphony Architecture

Chapter 2

Background

This chapter reviews the numerous existing and emerging technologies related to work presented in this thesis. The chapter begins with a discussion of the Java programming language, the JavaBeans component architecture, and the Java remote method invocation mechanism, because these are the fundamental building blocks of Symphony. Since Symphony is a visual compositional system, Section 2.2 describes some existing visual programming systems that allow composition of individual components. Symphony is primarily concerned with providing seamless access to local and remote legacy resources and Section 2.3 compares and contrasts it with several systems that provide remote access to legacy resources. Finally, Section 2.4 discusses some emerging Java-based distributed computing systems that are aimed at the problem solving environments community.

2.1 Java

Sun Microsystems' Java programming language, initially popularized as an Internet programming language, has quickly transformed itself into a full-fledged computing platform [2, 35]. Java is a object-oriented, multi-threaded and architecture-neutral programming language. It achieves architecture-neutrality by introducing the Java virtual machine (JVM) that provides an additional software layer between Java programs and the underlying operating system. Java programs are compiled to bytecodes which can be interpreted by the JVM. The JVM has been ported to a wide variety of operating systems and hardware platforms and its inclusion in popular Web browsers such as Netscape Navigator and Microsoft Internet Explorer has fueled the popularity of Java and Java-based software systems. Compiled Java classes can be loaded and executed by the JVM directly from the network, as and when they are needed, provided they adhere to the Java security restrictions. This allows development of relatively small programs called applets which can be embedded in web pages and are executed inside the browser when the page is loaded. The language also facilitates develop-

ment of larger stand-alone Java applications that can be executed from the command-line. Since an applet is downloaded from an untrusted source, it runs under certain restrictions within the local machine and is prevented from doing certain system tasks such as creating or editing files on the local file system. Such restrictions do not apply to applications.

Java syntax derives from C and C++ but eliminates features which make programming in these languages a complex task, the most important being that Java does not use pointers. It also eliminates multiple inheritance, templates and operator overloading which are commonly used features in C++. It implements an automatic garbage collection mechanism which frees the developer of the burden of explicit memory management and also provides automatic array bounds checking.

The Java development kit (JDK) comes with a large number of pre-defined class libraries which provide support for a wide range of computing tasks [34]. The specific standard libraries (also termed as Application Programming Interfaces [APIs] or packages) that are important to this discussion are the ones which provide support for user-interfaces, networking, file and stream I/O, distributed computation, object serialization and reflection, and the JavaBeans component architecture.

The networking API provides support for TCP/IP sockets and stream-based access to remote files represented by Uniform Resource Locators (URLs). The Abstract Windowing Toolkit (AWT) allows the developer to create graphical user interface (GUI, UI) components and provides support for graphics objects and images. GUI components communicate using a platform-neutral event mechanism. The reflection API enables a Java program to query a Java class or object about it's structure (methods, attributes, constructors, events) at run-time [28].

Object serialization supports the encoding of objects and the objects reachable from them into a stream of bytes and the complimentary reconstruction of the object graph from the stream [29]. It can be used for storing Java objects in a persistent state and reviving them whenever necessary. It can also be used for communication via sockets. The default encoding of objects protects private and transient data and supports evolution of classes.

Symphony uses Java libraries in the following ways. Each Symphony component is a Java object that has a simple visual representation, properties defining the component, and an ability to communicate effectively with other components with which it is composed. Symphony components were developed using the JavaBeans component architecture. Also, Symphony components that represent remote legacy executable resources communicate with a server, called the Symphony server, for initiating and controlling the execution of the remote application. The Symphony server has been implemented as a remote object on which such components make remote method invocations to obtain services. The Java Remote Method Invocation (RMI) mechanism has been used for implementing the server. The next two sub-sections give an introduction to these two building blocks of Symphony: the JavaBeans component architecture and the Java RMI mechanism.

2.1.1 The JavaBeans Component Architecture

As defined by Orfali et. al, a component is a stand alone software object that is not bound to a particular program, platform, language, operating system or implementation [15]. It does not constitute a complete application by itself, but can be used to build cheap, personalized applications. Components reduce the cost and complexity of software development by enabling software reuse through write-once run-anywhere capability. Different components interact using platform-neutral, client/server interaction models such as event notifications. Component technology, by origin, is a desktop technology, whereby different applications on the desktop can access and modify data objects created by peer applications regardless of the data content and format (e.g., Microsoft Office applications). The underlying software framework that enables this functionality and provides the facilities required for it is called a component architecture.

The JavaBeans specification defines a component architecture for building portable, platform neutral software components called **beans** which can be visually manipulated in builder tools [33]. Beans are platform independent in the sense that they can be plugged into existing component architectures such as Microsoft's OLE/COM [4, 18], Apple's OpenDoc [8], and Netscape's Liveconnect [14] using standard bridges. A beans builder tool maintains a palette of beans. The user can select any bean from the palette, drop it into a workspace, modify it's appearance and behavior and define its interaction with other beans. Beans are used to compose applets or applications.

A bean is a Java class that publishes its properties, methods and events. Properties are named attributes associated with a bean which can be read or modified by calling appropriate methods on the bean. Beans can also have bound properties which are capable of notifying other objects when their value changes. Bean properties can be customized by using property editors provided by the builder tool, or through an explicit customizer, if one is provided by the bean. All public methods in a bean are exposed to the containing environment by default. Events are a way for one bean to notify other beans that something of interest has happened. Events have many different uses, a common example being that of delivering notifications of mouse and keyboard actions in window system toolkits. A bean that wishes to receive a certain type of event registers its interest with a bean that fires the event. A builder tool can introspect on the bean and discover the properties and methods it exports and the events it can generate as well as receive. Beans support introspection in two ways: by adhering to special naming conventions for the class methods and by providing an explicit bean information class.

Sun provides a Beans Development Kit (BDK) which includes a reference builder tool, called **BeanBox** [37]. The BeanBox provides a rectangular workspace in which beans instantiated from a tool box can be manipulated and composed. It allows customization of a bean's properties through standard property editors or through the bean customizer, if one is provided. It also provides support for linking beans through property binding and event notifications. The customized state of the beans in the BeanBox workspace and the connections between

them can be saved for future modification and/or use. The mechanics of these operations will be described in the next chapter. Although Symphony beans were developed and tested in the BeanBox environment, they can be used in any builder tool that supports beans. Most of the commercially available Java development environments provide support for beans. Examples include Borland's JBuilder [21], SunSoft's Java Workshop [36], and IBM's Visual Age for Java [25].

2.1.2 Java Remote Method Invocation Mechanism

A distributed object is a software component that is independent of the operating system and hardware architecture used for implementation [15]. It may be located anywhere on the network and can provide services to remote as well as local clients via method invocations. Java provides the Remote Method Invocation (RMI) API for creating and accessing distributed objects [31]. The RMI mechanism lets programmers create Java objects whose methods can be invoked from another virtual machine, potentially executing on a remote host machine. RMI is the object-oriented counterpart of remote procedure calls (RPC) in the procedural programming world. RMI uses object serialization to marshal and unmarshal parameters and return values.

RMI provides an object registry mechanism where distributed objects can be registered. Client programs contact this registry to find out what object are currently registered and obtain references to these objects for making remote method invocations. References to remote objects can also be received as arguments or return values from remote method calls.

In Symphony, the RMI mechanism is used for making remote method calls from Symphony beans to the Symphony server. The server is implemented as a remote object that publishes its services by using the RMI registry mechanism. Client beans make method calls on a reference to the server object, obtained by contacting the RMI registry on the host on which the object resides.

2.2 Visual Compositional Systems

A visual programming system allows the user to compose a program by connecting together in meaningful ways, visual representations such as drawings and icons. It provides a more natural environment for creating diagrammatic forms that are easier to build and understand and helps the user concentrate more on the problem-solving task. Visual programming has been popularized by image processing systems and systems for user interface design.

Symphony users visually compose meta-programs based on the data-flow model. A meta-program is represented by a directed graph, where each node represents a program or a resource needed or created by a program and each directed arc represents a path over which

data flows. Nodes can also represent user interface and visualization components. The data-flow model is a natural co-ordination framework for manipulating meta-programs because most science and engineering computing tasks can be described as a series of steps where at each step data is transformed by operators and transferred to the next step. Much of the design for Symphony's data-flow model draws from numerous earlier data-flow based systems such as Khoros [27] and AVS [20]. This section describes two representative systems relevant to this discussion: Khoros, a visualization system that employs a data-flow mechanism for composing visual programs, and Sieve, a Java-based framework that is focused on collaborative component composition.

2.2.1 Visual Programming in Khoros

Khoros is a image processing system that provides tools for signal and surface plotting, image display and editing, image animation, geometry and volume rendering, and several other image processing tasks [27]. In image processing, pixels are often processed by a set of separate filters, each with a different convolution or image-understanding algorithm. Khoros provides a visual programming environment called Cantata for specifying the flow of data to create a program for image-processing.

In Cantata, a program is described as a directed graph where each node of the graph represents an operator or function and each arc represents a data-flow path. Each of the numerous stand-alone data processing and scientific visualization program in Khoros can be represented as nodes in the Cantata workspace. These Khoros programs can be on the local machine or on some remote machine. To create a visual program, the user places the desired programs (and control structures, if needed) on the workspace and connects them to indicate the flow of data from program to program. Such workspaces can be executed, saved, and restored later. Workspaces may also be encapsulated into stand-alone applications with a simple GUI.

Cantata extends the basic data-flow paradigm by providing flow control operators such as if/else, while, count, and trigger which provide data and control dependent program flow. This is a simple way to control order of execution when one is not already defined by the data flow. Variables may be set interactively by the user, or calculated at run-time via mathematical expressions tied to data values or control variables.

There is an event driven scheduler in Cantata, which dispatches processes in the correct sequence when the program is executed. Processes can be executed on remote machines too. The dispatcher is also responsible for determining the data transport which can be permanent (using files) or non-permanent (using sockets or streams).

Symphony borrows heavily from the Cantata data-flow architecture, but there are several differences between the two that must be noted. First, Cantata has been designed for image processing applications while Symphony is a general purpose framework that accommodates

any set of legacy resources. Second, Symphony is platform-independent and open in the sense that Symphony beans can be used in any standard bean container available for any operating system or platform. On the other hand, Cantata provides control-flow operators which are not available in Symphony, but these can be easily implemented as abstract beans. Cantata allows workspaces to be saved as stand-alone applications which can be executed from the command-line. This is not yet possible in Symphony because of the limitations of the current JavaBeans architecture. It is expected that the next version of the beans architecture code-named "Glasgow" [33], will open up a new set of capabilities for bean aggregation which enable this capability.

2.2.2 Collaborative Component Composition with Sieve

Sieve provides a JavaBeans-based shared workspace where multiple users can collaboratively add, edit, and link components to build a network of components [13]. It provides an ability for existing JavaBeans-based applications that adhere to standard beans mechanisms to be used in a collaborative manner, or to build completely new applications to take advantage of Sieve's real-time interactive collaboration. Existing beans that conform to the standard JavaBeans conventions can be directly shared across collaborating sessions through property changes - they need not be programmed specifically for collaboration. However, beans with a more explicit interaction with the Sieve framework can also be developed, where the developer has full control over how the application components are manipulated, linked, and rendered on the workspace.

Collaborators in the Sieve environment may view and manipulate the same or different parts of the shared workspace simultaneously. Sieve provides real-time information about each participants' actions and locations in the workspace to all collaborators. To aid collaboration it provides tele-pointers, which represent a remote users' mouse pointers, and a radar view of the workspace which depicts each collaborator's view of the workspace. Additionally, it provides features for annotating the workspace by using lines, arrows, text, images, and even arbitrary Java objects. These annotation objects are also shared across collaborating sessions. The state of a Sieve session is stored on the server, allowing late-joiners to be brought up-to-date. This also allows for asynchronous collaboration where collaborators working at different times can leave their work for other people to modify or review later.

Various collaborative applications have been built to use Sieve, one of which is a collaborative visualization environment (CVE). The CVE allows construction of data-flow networks from a set of modules. Modules may function as data sources, as data processors which filter and transform data in a variety of ways, or as visualization components. Source modules may read data from a wide variety of sources and data formats. Possible sources include Web-accessible files and SQL databases. The source modules hide all details specific to the actual data source from the processing modules and provide them with a consistent interface which allows data to be viewed as a two-dimensional table containing objects of

any type supported by the Java language. Each data flow module implements this interface, termed the TableView API. Source modules simply convert the raw data into a TableView representation. Processing modules read and manipulate these data and present an altered or extended table to downstream modules in the network. The resulting data-flow network uses an event mechanism to notify interested modules of changes to the data or to the configuration of the network. On receiving an event, modules can retrieve new or modified data from their source.

Symphony beans have been adapted to work in the Sieve environment so that meta-programs can be composed in a collaborative manner. Like Sieve, Symphony provides a data-flow based mechanism for composing networks of bean components. All Symphony beans conform to the standard JavaBeans conventions and can be used unmodified in Sieve. Thus, Symphony becomes an application in the Sieve environment, just like the CVE described above, with its own data flow and event mechanism. Sieve extends the capabilities provided by Symphony to a shared collaborative workspace.

2.3 Simplifying Remote Access to Legacy Resources

Legacy software applications are applications that are generally run from the command-line, have limited user interaction, and communicate using specially formatted files. The user may need to create special purpose input files before running the application, and the output files created by the application may need to be decoded or converted to alternate formats for analysis and visualization. The user may also need to write programs for converting the input and output data to the required format. For reasons mentioned in the previous chapter, it may not be economical to port these applications to platforms and machines other than the ones on which they reside and must be typically accessed remotely over the network. The problem becomes even more complex for users who must use several legacy applications distributed on heterogeneous servers to obtain a final solution.

Networked computing relies heavily on computing resources that are not locally present but are available to the user across the network. It is well suited for problem solving environments that require access to distributed, high performance, scientific computing resources. One of the most prevalent form of networked computing assumes that the processing software resides on the remote server and the user's data are sent to the server where the programs or numerical libraries operate on them. If the remote program is one that requires user interaction, mechanisms for exporting the user interface back to the local machine such as terminal emulation software and X-window based applications may be used.

This section discusses some systems that attempt to ease the burden of users who wish to use legacy applications on remote compute servers and how these systems relate to or differ from Symphony. Currently, utilizing distributed legacy computing resources is a complex and time-consuming task. Often times, the user has to go through a lengthy process of

obtaining accounts on the remote servers, logging in, setting up the required software for execution, and manually collecting the results.

2.3.1 Javamatic: Web Interface to Command-line Applications

Javamatic is a system for providing a Web-based interface to a remote command-line application [16]. It was developed at Virginia Tech by Abrams et.al. The Javamatic architecture consists of the interface client and the interface server.

The interface client generates a user interface (UI) in the form of a Java applet from a high-level description of the application and a set of UI mapping rules. The application is described by a set of commands (input parameters) grouped into logical categories. Categories can have sub-categories and the top-level categories combine to form the application category. This hierarchical arrangements of application parameters is mapped to corresponding Javamatic classes that represent UI components such as text fields, flags, and multiple choice boxes. The application can be described programatically by using the Javamatic class library. Alternatively, the description may be generated by dragging and dropping icons in a graphical editor.

The end-user interacts with the Java applet to provide the application parameters and then invokes the application. At this point, the applet contacts the Javamatic interface server running on the host machine where the legacy application resides. The server obtains the parameter string from the applet and invokes the legacy application in an independent thread. The legacy application code does not need to be changed or recompiled. But, if the application requires the parameters to be formatted in a certain manner, a wrapper script must be provided for translating the parameter string generated by the applet to the proper format. In this case, the interface server invokes the wrapper script with the parameters instead of the legacy application directly. All input data files for the legacy application must physically reside on the legacy system prior to execution of the application.

Although Javamatic does aim at providing a graphical user interface to a command-line application, it does not allow the user to specify how these parameters should be formatted when presented to the actual command-line application. The user must provide a script to convert the raw parameters obtained from this applet to the desired format. This script must also include code for moving files to the required directories before starting program execution and after completing execution. Symphony tries to eliminate a part of the functionality in this script and elevate its functionality to a graphical shell.

The biggest differences between Symphony and Javamatic are composibility and extensibility. Symphony allows integration of multiple legacy applications by composing them into a meta-program. Javamatic only provides a user interface to one legacy application at a time. The Symphony framework can be extended by adding new types of components to it. Javamatic does not provide such extensibility.

2.3.2 Web //ELLPACK: Remote Access to a Problem Solving Environment

Web //ELLPACK facilitates remote access to the //ELLPACK problem solving environment for solving partial differential equation (PDE) problems [23, 12]. //ELLPACK is a PSE for solving PDE problems on high performance computing platforms as well as a development environment for building new PDE solvers or PDE components. A GUI assists users in specifying the PDE problem and the solution algorithm and for analyzing computed solutions. Problems may be solved sequentially or in parallel on a variety of supported parallel platforms. The computed data may be imported into the solution analysis environment for visualization and analysis.

Web //ELLPACK allows remote users to access and use the //ELLPACK system. Users must obtain an account within the data space of a custom Web server where they are assigned a home directory for storing their files which can be accessed using the Web browser. Once the account has been manually created, the user may log in and use the //ELLPACK system. Users log in to their account by accessing a Web page and authenticating themselves. Once the user is logged on, the PSE can be accessed by pressing a button on the Web page. This starts a copy of the program on the remote server and the user interface is exported back to the user's local machine via the X-window system. The browser is blocked until the execution is done.

The user formulates the problem locally via the exported interface. Once the problem has been completely specified and submitted, it is solved on the server. The server may distribute the problem to other machines on the network. Once the PDE is solved, the user can view the output generated on the server or request that the solution be sent back to the local site. All the user's input and output data files are stored in their account directory which can only be accessed by using the login name and password given to the user.

There are several limitations of Web //ELLPACK. The system is limited to users having access to an X-terminal and only X-window based applications can be accommodated. Also setting up the system on the server for providing remote access requires setting up of complex access control mechanisms and policies which may be different for different systems. Symphony addresses these constraints to a large extent. It allows user to remotely execute X-window based programs as well as command-line applications which may communicate using files or through standard streams. Symphony does not assume user transparency in the sense that it does require the user to have the account information such as the user name and password to get to the required program or file. Also, Web //ELLPACK is a specialized system that provides remote access to a specific legacy application, the //ELLPACK problem solving environment. Symphony is aimed at more general-purpose science and engineering applications which can be composed of several distributed legacy applications.

2.3.3 NetSolve

NetSolve is a system that provides high-level APIs and interfaces to scientific software libraries for solving computational science problems in a reliable, fault-tolerant environment on distributed and heterogeneous computing resources [5]. It provides an environment that integrates computation, data gathering, data storage, and resource management.

Each separate numerical library on a compute server can be described using the NetSolve descriptive language in a machine-independent way. Calls are made to the actual library routines through this descriptive language interface. The description file can be compiled into an executable program on any Unix platform. In addition, NetSolve provides a Java GUI interface (an applet) which helps end users generate description files and compile them into new computational resources. The generated description file can be used in a machine-independent manner to set up new computational resources on any server.

NetSolve provides several client interfaces to the end users who wish to use the system. It provides C, Fortran, and Java APIs for programmers. It also provides a MATLAB interface and a graphical Java interface. Requests can be synchronous or asynchronous and results from asynchronous requests can be collected at a later time.

Every computational server that provides remote access to computational resources through NetSolve must run the NetSolve agent. Requests from client interfaces are first sent to a NetSolve agent which decides on which computational server the request will be executed. The agent performs load balancing by using the information contained in the request, static information about the server, network distance to the server and other parameters. Several NetSolve agents running on different sites can control and compete for the same set of resources. There is no centralized control and each agent is an independent entity which can be stopped and restarted at any time without affecting the integrity of the overall system. NetSolve provides a simple fault-tolerance mechanism. If a server becomes unreachable or fails during a computation, the computation is transparently restarted on another server as long as there is at least one server that can execute the request.

Symphony can be compared to NetSolve in the following ways. NetSolve is aimed at utilizing numerical software libraries in an effective manner, while Symphony addresses a broader set of legacy codes. NetSolve does not provide a visual interface for composing legacy resources and it's architecture is not based on data-flow between separate applications. On the other hand, NetSolve does provide a simple fault tolerance mechanism which is absent in Symphony.

2.4 Java-based Distributed Computing Frameworks

The dramatic rise in the popularity and ubiquity of the World Wide Web and the introduction of the network-centric and architecture independent Java programming language has

fueled a race for developing a Web-based framework for building distributed object-oriented applications. Some of these efforts aim at developing an infrastructure for true distributed applications and other aim at utilizing the vast pool of resources on the Web for traditional parallel computing tasks. The distributed computing model is shifting from the current distributed shared memory or LAN-based cluster of workstations model to true distributed computing with components that are distributed on to platforms that may be thousands of miles away from the computation owner's location.

This section discusses two developing architectures for Web-based distributed computing which assume a Java development environment and which are aimed at the problem solving environments community. The first, named WebWork, aims at developing a collaboratory, multi-server problem solving environment on the Internet and integrating the field of high performance computing with Web technology. The second framework proposed by Chandy et. al explores the use of Java as a means for building distributed systems that execute throughout the Internet.

2.4.1 WebFlow

WebFlow is part of an ongoing project called WebWork at Syracuse University's Northeast Parallel Architectures Center (NPAC) [10, 9]. WebWork proposes the development of high-performance applications that make use of the Internet's wealth of computing resources, by creating and utilizing compute servers throughout the Internet.

WebFlow is a general-purpose Web based visual programming environment for coarse-grained distributed computing [3]. A distributed computation is represented by a set of channel-connected Java modules. Each WebFlow module must implement a Module API. WebFlow employs a 3-tier architecture with the modules forming tier-3, a set of Java servlets that co-ordinate and manage the distributed computation in tier-2, and a data-flow based visual graph editor Java applet which provides the user interface in tier-1. Java servlets are the server-based counterparts of Java applets. These are URL-addressable Java objects that run within a Java Web Server.

The end user interacts with the WebFlow applet to create data-flow based computational graphs where nodes represent WebFlow modules and arcs represent data channels between those modules. Through the applet users can request for new modules to be initialized, request a connection between two initialized modules, and run or destroy the whole application. WebFlow management functions are handled by three URL-addressable servlets: the session manager, the module manager, and the connection manager. The applet sends each user request to the session manager running on the host from which the applet is downloaded. The session manager maintains a session object for each user and honors the user requests by calling on the services of module managers and connection managers.

Module and connection managers are run in pairs on any host machine that wishes to take

part in the computation and their services can be invoked by a session manager running on any host. A module manager handles three types of requests from the session manager, requests to initialize, execute, and destroy modules instances. Every instance of a module executes in a separate thread. In order to keep the module manager independent of the module function, each module is required to implement a Module API which includes method calls for initializing, executing, and destroying the module. A module manager can support any number of modules, and requests coming from any number of session managers.

When a module is initialized the module manager registers each of it's ports with the connection manager running on the same host. The connection manager is in charge of establishing connections between individual ports of two modules which may be running on different host machines. When a request to connect the ports of two modules is received by the connection manager, it validates the request and establishes a socket connection. If the second module is on another host, it negotiates with the connection manager on that host for making the socket connection. Each port has a type which indicates the type of data item it can send or receive. Some port types come pre-built with the system and others can be implemented by the developer.

Input and output modules that take input from the user and display the output are implemented as applets which reside on the same server as the session manager. The request to open the applet in the user's browser is forwarded by the session manager to the WebFlow applet which loads the required applet in a new window.

The biggest difference between Symphony and WebFlow is that in WebFlow problem-solving modules have to be implemented in Java and it does not provide support for executing remote legacy applications. Symphony, on the other hand, provides remote access to legacy applications without requiring any modifications to these applications.

2.4.2 The Infospheres Infrastructure

The Infospheres Infrastructure, being developed at Caltech by Chandy et. al is a distributed system framework implemented in Java that provides mechanisms for programmers to develop distributed system components from which distributed applications can be created [6, 7, 26]. It provides a variety of messaging models, including asynchronous, synchronous, and remote procedure/method calls and a variety of distributed system services, including local and global naming, object instance control, object persistence, and others.

The components in this distributed system are termed as processes. Processes are persistent communicating objects that manage interfaces and devices. A process may be in one of three possible states at any time: active, waiting, or frozen. A waiting process can be frozen and an active process can summon a frozen process. A frozen process can be moved from one location to another.

A virtual network consists of groups of communicating processes. Processes interact by

receiving and sending requests for modifying or reading state. Each process has a set of 'inboxes' through which it receives requests and a set of 'outboxes' through which it sends requests. Every inbox and outbox has an interface type associated with it which defines the types of requests it can send and receive. Each inbox and outbox has a global address which can be sent from agent to agent. Inboxes and outboxes are basically implemented as message queues and messages sent along a channel are delivered in the order sent.

An initiator process can initiate a session for accomplishing a task. A session actually represents a distributed computation. The initiator process is responsible for creating the virtual network connections. Once the task assigned to a session is accomplished the component processes can be frozen for taking part in a later session. The infrastructure provides various services needed in a distributed transaction such as checkpointing, locking, deadlock avoidance, termination detection and resource reservation.

Processes are implemented as distributed, multi-threaded Java objects called *djinn*s (pronounced *genie*). *Djinn*s have global addresses and interact under the loose control of a *djinn* master. The *djinn* master is responsible for instantiation of new *djinn*s, thawing of persistent *djinn*s, and the initial communication to a instantiated or thawed *djinn*. The communication substrate is sockets, but can be replaced with other communication systems. *Djinn*s can be frozen to be thawed later, by serializing and deserializing them. *Djinn*s have well defined interfaces and are written using the Infospheres Infrastructure Java package.

Infospheres primarily provides a framework for programmers to create distributed object oriented applications. It does not provide support for legacy applications and also does not provide a visual compositional environment for creating applications.

2.5 Comparison

This section summarizes the material in the previous sections of this chapter by providing a comparison of Symphony with other related work in terms of the set of features provided by Symphony. The specific features that are considered stem from the goals of this research outlined in Section 1.2. Table 2.1 depicts the comparison. All the features listed in this table are those provided by Symphony, but that does not mean that these are the only features provided by the related systems.

Table 2.1: Comparison of Symphony to Related Work

	Symphony	AVS	Javamatic	Web //ELLPACK	NetSolve	WebFlow	Infospheres
Platform Independent	√		√			√	√
Visual	√	√		√	√	√	
Compositional	√	√			√	√	
Distributed	√	√			√	√	√
Legacy Support	√		√				
Extensible	√	√			√	√	√
Open	√		√			√	√
Generic	√		√				√
Web-aware	√		√	√		√	
Persistent	√	√	√	√	√		√
Secure	√	√	√	√	√	√	√
Graphical	√		√	√		√	

Chapter 3

Using Symphony

This chapter provides a detailed account of how Symphony can be used for creating meta-programs from existing local and remote resources and for saving and executing the built meta-programs. A meta-program can be visually composed based on the data-flow requirements of executable components and executed in a manner that respects these requirements.

Programmers may extend the set of Symphony beans by adding new types of components without detailed knowledge of the Symphony or JavaBeans architectures. Examples of new component types that can be added are beans that provide a graphical interface for soliciting input parameters for command-line legacy applications and beans for visualizing the output data from a legacy application. This chapter also describes the procedure of implementing new Symphony beans types.

Section 3.1 defines terms used in the rest of the chapter. After describing the mechanical aspects of constructing a meta-program in Section 3.2, Section 3.3 introduces the Symphony beans and describes their use and capabilities in detail. Section 3.4 explains the logical aspects of meta-program construction and Section 3.5 describes the various operations that can be performed on a meta-program. Section 3.6 outlines the procedure of implementing new bean types and, Section 3.7 concludes with a real-world meta-program built from Symphony beans.

3.1 Terminology

This section defines several terms that will be used in all subsequent discussion. A **Symphony bean** is a customizable software component that can be linked with other Symphony beans (an in some cases, non-Symphony beans) to create meta-programs representing computations controlled by data-flow patterns.

Although Symphony beans were developed and tested using **BeanBox**, Sun's reference im-

plementation of a beans container, they can be manipulated or executed inside any JavaBeans container that conforms to the JavaBeans specification [33]. All Symphony beans conform to Sun's JavaBeans API Specification 1.01 and thus can be used in most commercially available bean containers. A **beans container** may either be a beans builder tool like BeanBox or an environment which only allows a set of serialized beans to be loaded and executed. The BeanBox allows testing, customization, linking, and serialization of beans. It can also load and recreate a set of beans saved as a serialized file.

A **client site** is the host machine where the meta-program is built or manipulated in a beans container. A **server site** is the host machine that runs the Symphony server. The Symphony server is a a Java RMI server that waits for execution requests from the Symphony beans residing on a client site.

A **local resource** is a program or file on the client-site, while a **remote resource** is a program or file on any site other then the client-site. A remote program represented by a Symphony bean has to be on a Symphony server site but a remote file does not necessarily have to be on a server site.

In general, the term **program** is used to denote any executable application program, and the term **file** is used to denote any data file.

3.2 Getting Started

This section describes the procedure for setting up the BeanBox for composing Symphony meta-programs. It also describes important BeanBox operations at a higher level in terms of the steps required for composing a meta-program [38]. When started, the BeanBox application displays three windows: The tool box, the BeanBox workspace window, and the property sheet as depicted by Fig. 3.1.

Two modifications have been made to the BeanBox provided by Sun purely for aesthetic purposes. First, the original BeanBox displays an empty property sheet when a bean that does not export any properties is selected in the workspace. In the modified BeanBox, the property sheet disappear when such a bean is selected. Second, the original BeanBox does not visually depict connections between beans, but the modified BeanBox does. This later modification is important for Symphony in order to be able to present the data-flow graph of a meta-program more naturally to the user. This, however, does not mean that the modified BeanBox has to be used for composing and executing meta-programs. Any bean builder tool can be used whether or not it provides these visual features.

There are five major steps to creating a meta-program. Symphony beans first need to be loaded into the BeanBox environment so that they are accessible from the tool box. After this, the required beans must be inserted into the workspace and customized for the resources they represent. Fourth, the beans must be linked according to the desired data-flow patterns,

and finally, the built meta-program can be verified, executed, or saved to a file for future use. This section also provided details for setting up the execution environment for executing a meta-program. Details about meta-program verification and execution are given in Section 3.5.

Loading Symphony Beans

Symphony beans are packaged in a single Java archive (jar) file. In order to use the beans they need to be loaded in the BeanBox from the jar file, which can be done in two ways. The jar file can be placed in the BeanBox's default jars directory (accessible from the directory in which the BeanBox is installed), in which case the beans are loaded automatically when the BeanBox application is started. Alternatively, the "Load Jar File" menu option of the BeanBox can be used to load the beans. Regardless of how they are loaded, if the load operation is successful, icons for Symphony beans appear in the tool box. Figure 3.1 shows Symphony beans already loaded into the environment. Notice, for example, that icons for the Program and File beans appear in the ToolBox window.

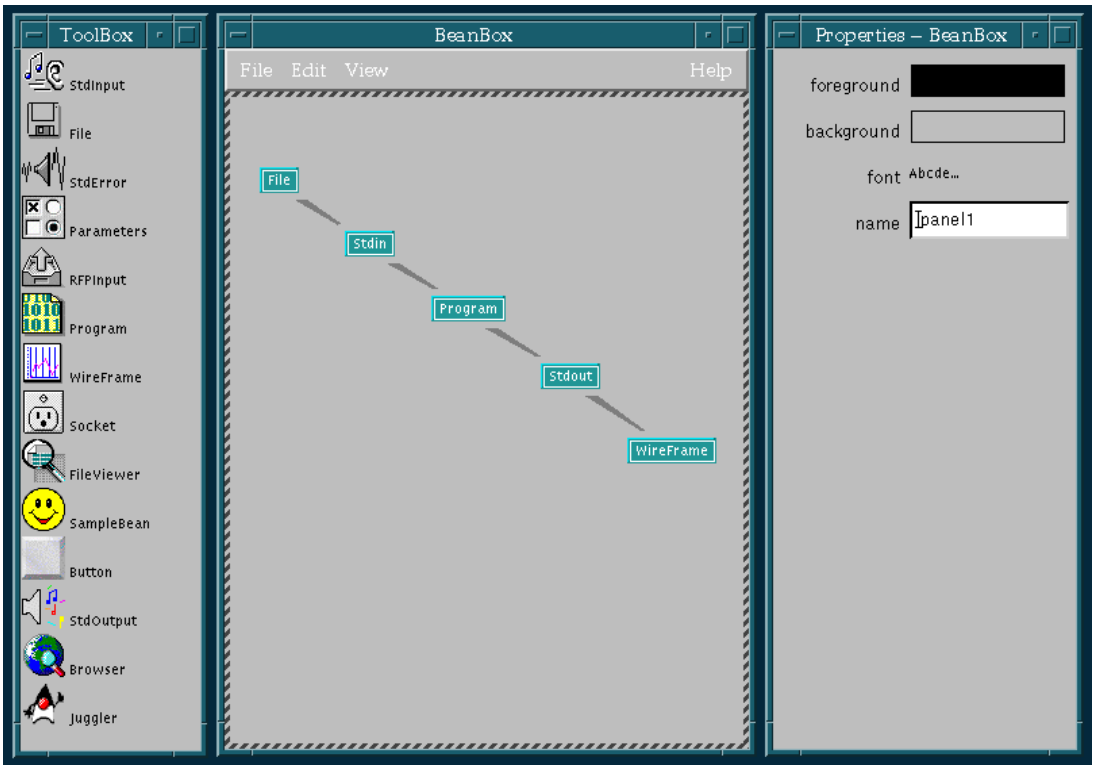

Figure 3.1: BeanBox Windows

Inserting Beans in the Workspace

A bean is inserted in the BeanBox workspace by clicking on the bean label or icon in the tool box, dragging the mouse pointer to the desired insertion point in the workspace, and clicking the left mouse button or, the insertion can be cancelled by clicking the right mouse button anywhere in the workspace window. When a bean is inserted it becomes the currently selected bean in the workspace and a hatched border appears around the been. Any other bean can be selected by clicking the left mouse button on the bean or on its perimeter. The workspace itself may be selected by left-clicking in the workspace area outside of any bean. In Figure 3.1, the workspace has been selected as indicated by the hatched border around the edge of the workspace.

Customizing Bean Properties

Every bean publishes certain properties which can be discovered by the BeanBox at run-time and customized through property editors. Default property editors are provided for simple properties represented by strings, numbers and colors, and new ones can be implemented for other property types. Property editors for all of a bean's published properties are collected together into a property sheet which is displayed when the bean is selected in the workspace. Figure 3.1 shows the property sheet for the workspace panel since the workspace is the currently selected object in the BeanBox window. This property sheet permits customization of the foreground and background colors for the workspace, the default font, and the name of the workspace panel.

Property editors are sufficient only for the simplest of property editing tasks. If more explicit customization such as property grouping or error checking is needed, the bean must provide a specialized customizer. In this case, the bean may or may not expose individual properties to the BeanBox for creating the property sheet. The customizer can be accessed by selecting the bean in the workspace and clicking on the Edit → Customize... menu item in the BeanBox.

All Symphony beans suppress property editing through property sheets and instead provide customizer dialogs. If, for example, a user selects the Program bean shown in Figure 3.1 and chooses the Edit → Customize... menu-item, the BeanBox will display the Program bean customizer shown in Figure 3.2. This customizer can be used for setting the properties of a Program bean. A detail to keep in mind while customizing Symphony beans is to always click on the "Apply Customization" button after making any changes in the customizer and before clicking on the "Done" button, otherwise the customization will not take effect. Details about the Program bean and its customization will be presented later.

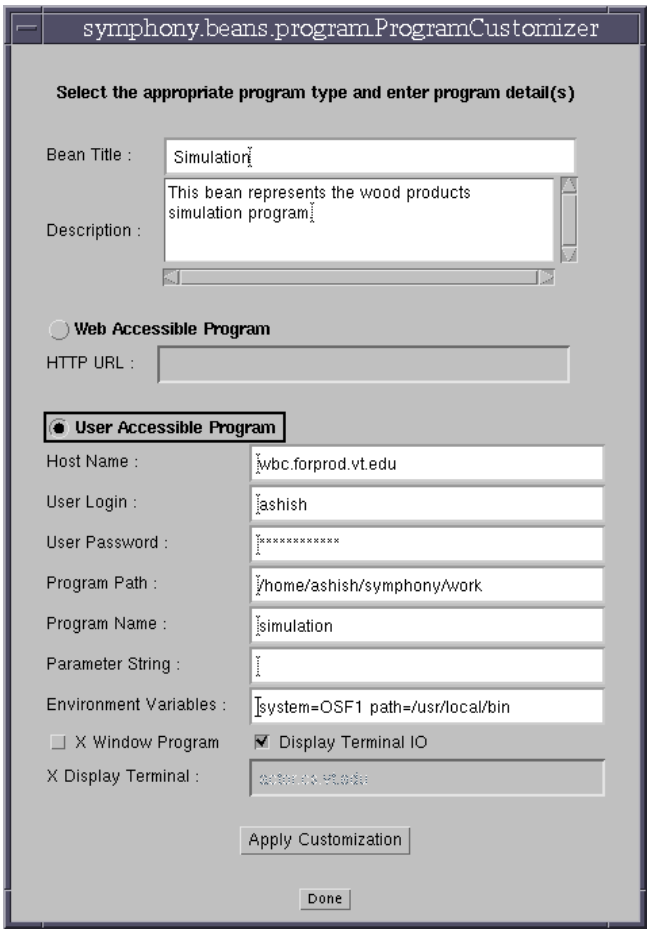

Figure 3.2: Program Bean Customizer

Linking Beans by Events

Beans can communicate through events and bound properties as explained in Section 2.1.1. The BeanBox provides a user interface for connecting beans through these two mechanisms. In Symphony, only event connections are important. Details about linking beans through bound properties can be found in the JavaBeans tutorial [38].

A particular bean can generate a variety of events depending on the bean type and purpose. For example, the most important event generated by a button bean is the action event, which is generated when the button is pushed. On the other hand, a bean which has an explicit user interface for interacting with the user may generate events corresponding to mouse and keyboard actions. When a bean is selected in the workspace, the types of events it generates appears as a sub-menu of the Edit → Events menu item. Items in this sub-menu

represent classes of events such as mouse events, keyboard events, window events, etc. Each item opens another sub-menu which shows the actual events generated, such as the mouse clicked event, mouse dragged event, key pressed event, and so on.

An event generated by a bean can be linked to a method call in another bean, such that the chosen method in the target bean is invoked automatically when the chosen event is generated in the source bean. Each Symphony bean that takes part in the data flow, except for a Consumer bean, exposes only one type of event (connection → createConnection) and every bean, except for a Producer bean, exposes only one target method (`eventSend`). In order to connect two Symphony beans in the desired data-flow graph, the connection event from a `data source` bean must be connected to the eventSend method of a `data sink` bean, by following the steps given below. It must be noted that a connection between two Symphony beans is always made in the direction of the desired data flow. Consumer and Producer beans will be presented in the next section.

The following steps outline the process of creating the connection between the File bean and the Stdin bean shown in Figure 3.1.

1. Select the source bean (the File bean) by clicking the left mouse button on it.
2. Select the specific event under the Edit → Events menu item of the BeanBox. The BeanBox will display a rubber-banded line starting from the source bean. (Select the Edit → Events → connection → createConnection menu item)
3. Drag the mouse pointer over the target bean (the Stdin bean) and click the left mouse button.
4. The BeanBox displays a dialog box containing a list of methods in the target bean that can act as event-handlers for the selected event in the source bean. Select the desired event-handling method and click the OK button in the dialog box. (Select the `eventSend` method in the dialog box and click the OK button).

Since the implementation of the source bean knows nothing about the target method in the target bean, the BeanBox generates a standard adapter class for forwarding the event notification from the source bean to the target bean. The BeanBox also takes care of registering the adapter object with the source bean. If the adapter generation and registration are successful, an arrow depicting the connection appears between the source and target beans.

Saving the Workspace

A meta-program may be saved in a persistent state in the form of a Java serialized file. This file can be loaded into the BeanBox at a later time to reproduce the meta-program and modify or execute it.

The BeanBox uses object serialization to save and restore the current contents of the workspace (the beans in the workspace, their state, and connections) [29]. On selecting the File → Save menu item in the BeanBox, a file dialog box appears, which can be used to save the current workspace to a named file. In order to retrieve the saved beans, select the File → Load menu item and select the required file name in the file dialog that appears. The current contents of the BeanBox workspace will be replaced with the contents of the serialized file. A serialized file is machine and architecture independent and can be transported to any other site and loaded in any bean container, if support for static serialization is provided by the container.

Setting up the Execution Environment

Before a meta-program can be verified or executed, the user needs to ensure the existence of two types of server processes on remote host machines from which the meta-program accesses resources. Remote files are read or written by creating FTP connections to the host machines on which these files reside. To enable this, an FTP daemon process needs to be running on the remote host. This is not necessary for the local host because local files are accessed directly from the file system. Secondly, for executing a remote programs represented by a Program bean, the Symphony server needs to be running on the host machine on which the program resides. The Java class files for executing the Symphony server are included the Symphony_server.jar file that comes with the Symphony distribution. In order to start the Symphony server on a particular host machine, the following steps need to be taken:

- Copy the Symphony_server.jar file to the host machine on which the server is to be run.
- Unjar the file by giving the command: jar xvf Symphony_server.jar
- This will create a directory named codebase in which the server files are extracted. The jar file also contains a script named run_server.sh which can be used to run the server.
- Change directory to the codebase directory and executed the script run_server.sh. Some parameters in this script may need to be modified in order to suit the system configuration on the host machine. The README_SERVER contains comments describing the parameters that may need to be customized.

3.3 Symphony Beans

There are two important things to understand for using any set of beans that can be composed to form applications: the purpose of each bean and its connection information. The

Table 3.1: Symphony Beans Summary

Bean Type	**Purpose**
Program	Represents a local or remote executable resource which can be a non-interactive command-line executable, a command-line executables that interacts through standard streams, a X-Window based program, a local GUI-based program, or a web-accessible program such as an applet or a CGI script
File	Represents a local or remote data file that may be a Web-accessible file, an anonymous FTP file, or a private file accessible from a user account on a host machine. It can represent a file that can be read from, written to, or both.
Socket	Represents a TCP/IP socket which can be read from or written to
Stdin	Provides a way of redirecting data from a data source to a program's standard input stream.
Stdout	Provides a means for redirecting the standard output stream from a program to a bean that accepts data for processing
Stderr	Provides a means for redirecting the standard error stream from a program to a bean that accepts data for processing
Producer	Abstract bean that can be extended by implementing a simple interface to define new beans types that act as data producers, e.g., beans that extract data from remote servers and beans that provide a graphical user interface for obtaining parameters for a command-line legacy application,
Consumer	Abstract bean which is useful for implementing new beans that act as consumers of data, e.g., visualization beans and viewer beans
Filter	Abstract bean that allows the user to implement different kinds of beans for filtering the data flowing through the system, eg. sorters, image filters, file format converters, etc.
Properties	Utility bean that decreases the amount of work user has to do for customizing the beans in a meta-program
Annotation	Allows users to add off-screen, time-stamped annotations to the meta-program being constructed and to view them at any time
Browser	A simple variation of Sun's HotJava bean that is represented on the workspace by a button which when pressed, opens a new browser window.
Button	A simple button bean which can be linked to an an action in some other bean

connection information includes details about what types of connections are allowed and what each type of connection means from a functional point of view. This section introduces the various types of beans available in the Symphony framework and describes the purpose and attributes of each bean. Section 3.4 provides the connection information for all beans and describes the process of composing a meta-program that uses these beans.

Table 3.1 shows a list of all Symphony beans with a brief description of each. The set of Symphony beans has been divided into three functional categories: core beans, abstract beans and utility beans. Each of these categories of beans along with descriptions of individual beans is explained in the following sub-sections.

3.3.1 Core Beans

Beans that provide the core Symphony functionality are the Program bean, File bean, Socket bean, and standard stream beans which include the Stdin, Stdout, and Stderr beans. In order to give the reader an idea about possible ways in which these beans might be used, several sample meta-programs containing these beans are described before describing the individual beans. In further discussion, based on the context, a reference to any bean must be interpreted as a reference to the actual resource represented by it.

Figure 3.3 shows a meta-program in which File1 is redirected to the standard input read by Program1 and standard output from Program1 is redirected to File2. File3, which is created from the data stream received from the Parameters bean, is directly read by Program1 when it is executed. The implicit assumption is that File3 is a resource needed by Program1 to execute correctly, and if present will act as an input for Program1 in some unspecified manner known only to the program. The only thing Symphony guarantees is the file's creation, if needed, and presence in the required location before the program starts executing. Similarly, File4 is a file that is created by Program1 during execution in an unspecified manner known only to the program. Once Program1 finishes execution, if File4 is present in the required location, the FileViewer bean (which is actually an implementation of the Producer bean) is launched for displaying the contents of the file to the user.

Figure 3.4 shows another meta-program where data read from the "Read Socket" is used for creating File1 and the output written by the program on it's standard error stream is written to the "Output Socket".

Program Bean

A Program bean represents a local or remote executable resource. Based on the location, input-output requirements, and the manner in which it can be accessed, the Program bean can represent several types of programs.

After instantiating a Program bean in the workspace, various attributes of the bean may be

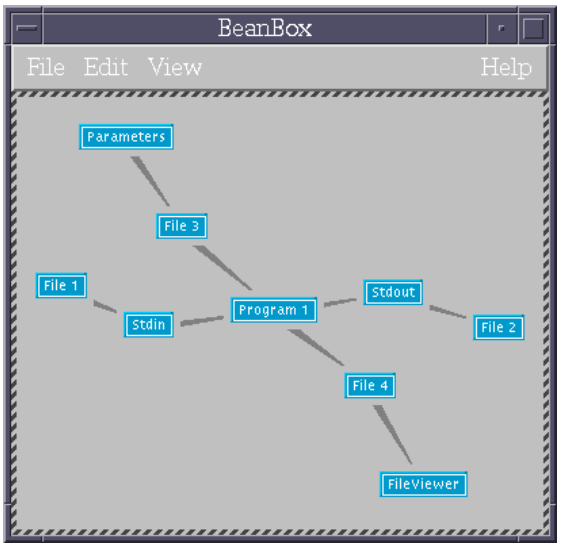

Figure 3.3: Sample Meta-Program 1

customized by using the bean customizer. Figure 3.2 shows the customizer for the Program bean. The bean title is the label given to the bean in the workspace and the description field can be used for saving a short description of the program represented by this bean. Other attributes that need to be customized depend on the type of program represented. The program types along with their execution time behavior and attributes that need to be customized for each type are listed below.

1. **Web-accessible programs:** These are programs that can be accessed from a Web browser, such as CGI scripts, applets or servlets. For this type of program, the 'Web Accessible Program' type must be selected in the customizer and a HTTP URL for the program must be entered in the URL field. During execution, the URL is loaded in a browser window which is spawned by using the Browser bean.

2. **User-accessible programs:**

   (1) **Non-GUI command-line executables:** These are programs that read input from one or more possible sources (including local or remote data files, standard input, command-line arguments and environment variables) and/or write output to one or more possible destinations (including local or remote files standard output stream, standard error stream). After selecting the 'User Accessible Program' type in the customizer, the attributes that must be customized for this type are: the host name on which the program resides, absolute path to the program and program file name, and the username and password required for authenticating

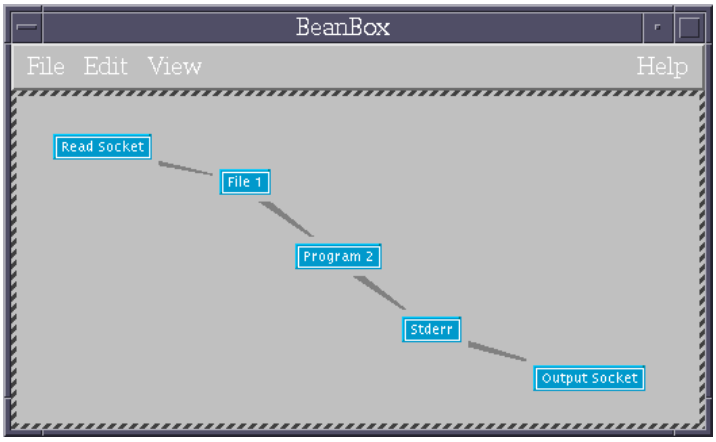

Figure 3.4: Sample Meta-Program 2

the user. Optional command line arguments and environment variables can be passed to the program by using the corresponding fields in the customizer. Environment variables must be entered as $name = value$ pairs separated by spaces.

Although it is possible to execute a local or remote program that reads from or writes to the standard terminal streams (stdin, stdout, and stderr), it is not possible to interact with such a program in a terminal-like window because the standard output stream from a program is buffered, which precludes real-time interaction through the streams. Instead, the data needed by the program on the standard input must be presented to it by the Stdin bean as shown in Figure 3.3. Data written by the program to the standard output or error streams can either be redirected to other resources by using the Stdout and Stderr beans (as in Figures 3.3 and 3.4) or displayed in a dialog box when the program is executing. If a Stdout bean is connected to the program, the dialog box that appears during execution will only display output from the standard error stream and if the Stderr bean is connected, only the standard output will be displayed. If both Stdout and Stderr beans are present, the dialog box will not be displayed. The display of this dialog box can be turned off explicitly by un-checking the "Display Terminal I/O" box in the bean customizer.

(2) **X-window based program:** A Program bean can represent a local or remote X-window based program, provided an X-server is running on the client-site. For this type of program, in addition to the customization for the previous program type, the user must provide the X-server display variable needed to export the X-display to the local site.

(3) **Local program with a GUI:** A Program bean can represent a local program which needs a GUI since it's no different than executing the program from the

command line or by clicking it's icon in the window manager. For this type of program the same set of attributes as a non-GUI command-line executable need to be customized.

The Program bean customizer provides for customizing the port attribute for a remote user-accessible program, with a default value of 9999. This port number is the standard port number used by the Symphony server and should not be modified unless there is a reason to do so, as explained in Section 3.5.

File Bean

A File bean represents an existing file containing data for a particular program, a temporary file that is created during execution, or an output file. Based on the file location and the manner in which it can be accessed, the File bean can represent several types of files.

Figure 3.5 shows the customizer for a File bean. The bean title is the label given to the bean in the workspace and the description field may be used to provide a short description of the file represented by this bean. The file types along with the attributes that need to be customized for each type are:

1. **HTTP Accessible File:** This type of file is represented by a HTTP URL and can only acts as a data source, i.e., a File bean for a URL can only have an output connections but no input connection.
2. **Anonymous FTP File:** This type of file is represented by a URL with the 'ftp' protocol identifier. Depending on the set of file access permissions on the host machine, this type of file can act as an input, output, or temporary file.
3. **Private User Accessible File:** A private file in the directory space of some user on a remote host machine is of this type. In a meta-program, it can be connected to function as an input file, output file or temporary file. Attributes that need to be customized for this type of file are: the host name on which the file resides or on which it is to be created, and the user name and password combination for authentication purpose, the absolute path to the file and the file name.
4. **Local File:** For this type of file only the absolute file path and the file name need to be customized. A local file can also function as an input file, output file or temporary file depending on how it is connected.

Socket Bean

Depending on the manner in which it is connected, a Socket bean either encapsulates an input stream obtained from a TCP/IP socket connection, or an output stream for writing

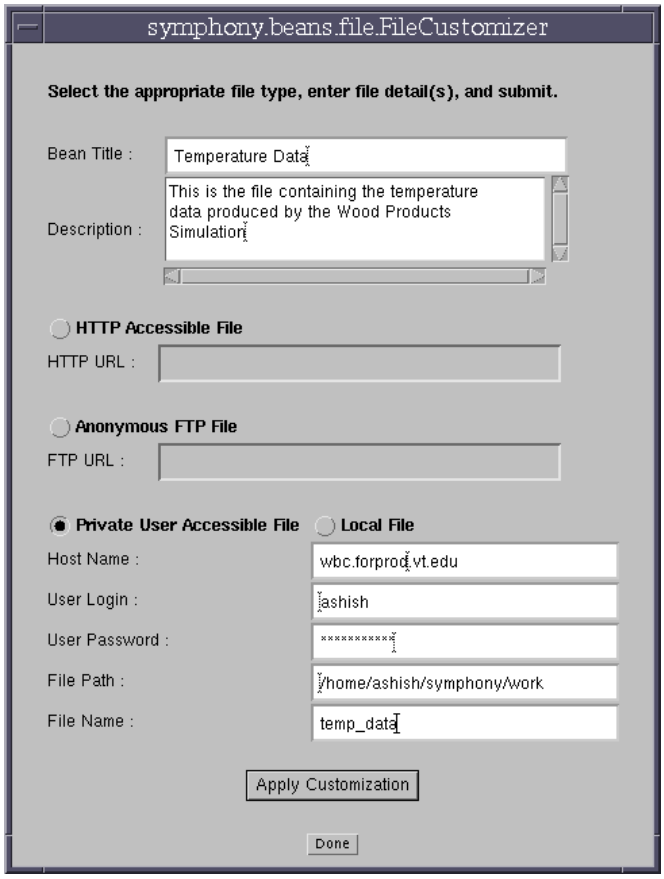

Figure 3.5: File Bean Customizer

to a socket connection. Figure 3.6 shows the customizer for a Socket bean. The title and description attributes have the same purpose as that in the Program and File beans. The only attributes that need to be customized for a Socket bean are the host name and port number for the socket.

Standard Stream Beans

The Stdin bean represents a data pipe through which data from a source bean can be redirected into the standard input stream of a program represented by a Program bean. The Stdout and Stderr beans function as pipes through which data written by a program on its standard output and error can be redirected to a sink bean. The only attributes these beans have are a title and description which can be set using the customizer.

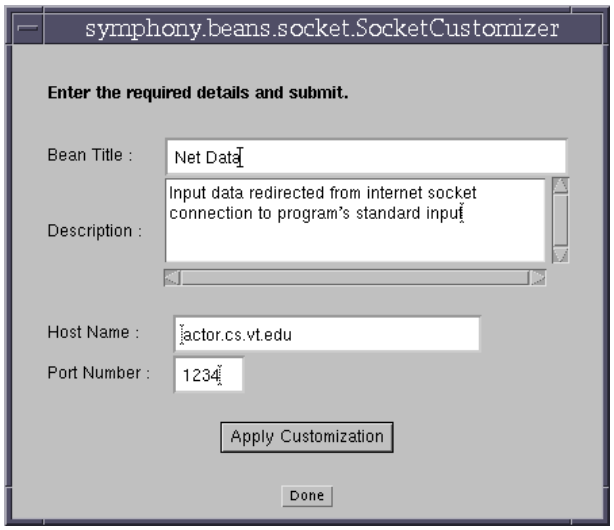

Figure 3.6: Socket Bean Customizer

3.3.2 Abstract Beans

Symphony beans present a simple interface to the user for constructing meta-programs. However, a bean implementation includes substantial machinery for creating a connection between two beans, implementing the data-flow, handling events generated during meta-program operations, and other related functions. Also, the beans have to be programmed in accordance to the JavaBeans API specification [33]. Thus, implementing a new type of Symphony bean is not a simple task for the casual Java programmer. To simplify the creation of new bean types, Symphony provides a set of abstract beans which hide the details of the Symphony event model and execution protocol. By implementing a simple interface, programmers add new bean types to the system without needing to understand the details of the Symphony or JavaBeans architecture.

In line with the data-flow model provided by Symphony, the three abstract beans included are: a data Producer bean, a data consumer bean, and a data Filter bean. The following sub-sections introduce these beans and discuss the sample beans that have been implemented by using each abstract bean. Details regarding the actual implementation of new beans from these abstract beans will be provided in the next chapter.

Producer Beans

A Producer bean is designed to act as a generic data source and can be connected on it's output side to any bean that can act as a data sink. A Producer bean can not have any

Table 3.2: Text File Format Expected by the Parameters Bean

Line 1	Program title				
Line 2	Number of parameters				
Line 3	Parameter separator character for output				
Line 4 (tab separated)	Parameter field label	Default value (opt.)	Input field length (opt.)	Display flag (opt.)	Help string (opt.)
⋮	Define other parameters with the same format as line 4				

input connections.

A sample bean that has been implemented by using the Producer bean is the **parameters bean**. The parameters bean reads from a URL a textual description of the set of parameters expected by a legacy program and creates a graphical interface to solicit those parameters from the user. The parameters entered by the user are passed onto the next bean in the sequence for further processing during execution. The required format for the text description is shown in Table 3.2. The parameter separator character is used to separate the parameter values in the output. For defining a parameter only it's label is necessary. The four other fields are optional, but if present they must be present in the order defined. The display flag field for a parameter indicates whether or not that parameter is to be solicited from the user. A value of zero indicates that the concerned parameter should always be given the default value in the output. The help string will be displayed when the user clicks the help button located beside the parameter field in the interface. Care must be taken not to split any of the field onto multiple lines and not to embed tabs in the value. In addition to the title and description attributes, the customizer for a parameters bean contains a single field to set the URL address for the description file. Figure 3.7 shows a sample input interface created by the parameters bean.

Consumer Beans

A Consumer bean is designed to act as a generic data sink and can be connected on it's input side to any bean that can act as a data source. This abstract bean is useful for writing visualization and viewer beans. Two specialized Consumer beans have been implemented:

- **File Viewer:** The file viewer bean reads a text stream obtained from it's input bean and display the text in a new window.
- **Wire Frame:** The WireFrame bean reads a specially formatted stream of data and creates a rotatable 3D wireframe graph.

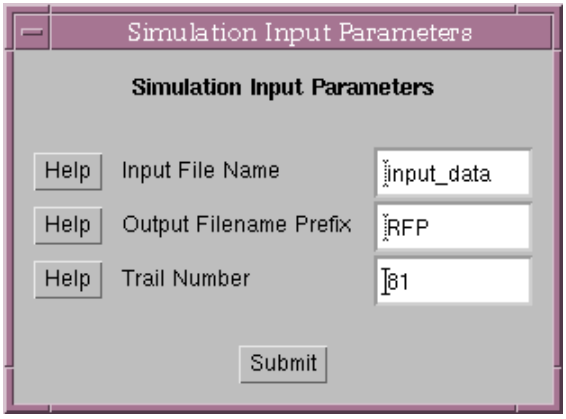

Figure 3.7: Parameters Bean Interface

Filter Beans

This abstract bean can be used to implement a bean which filters a stream of data. The simplest filters can be text filters analogous to the Unix filters. More complex filters include image processing filters and file format converters.

3.3.3 Utility Beans

Symphony provides the following four utility beans:

- **Annotation Bean:** The Annotation bean can be used to associate descriptive textual information with a meta-program. Through the customizer provided by the bean, the user can add a new annotation or view existing annotations. Annotations are stamped with the date and time when they are added. This provides a way of documenting a meta-program during it's entire life cycle.
- **Properties Bean:** This bean encapsulates initial values of common properties that may have to be customized for a Program bean and File bean while building a meta-program. Specifically, the property values that can be defined are a base URL, remote host name, login name, password, default path, and default X-display string. When a new File bean or Program bean is instantiated on the workspace, if a Properties bean is present, relevant values are initialized by reading values from the Properties bean. This saves time during customization if property values are similar across beans.
- **Browser Bean:** The Browser bean is a simple modification of the HotJava HTML Component available from Sun. On the workspace, it is represented by a button and

it has a customizer which accepts a URL. When the button is clicked, the URL set in the customizer is loaded in a separate browser window. The window also contains the 'Back' and 'Forward' buttons which emulate browser history.

- **Button Bean:** This is a simple Button bean whose action event can be connected to a method in some other bean, e.g., to the method which loads a URL in a new browser window in the Browser bean.

3.4 Composing a Meta-program

This section describes the procedure for composing a meta-program. Possible data-flow patterns are discussed and the types of connections allowed between beans are listed.

As defined in Chapter 1, a meta-program is a set of linked program and data components forming a data-flow graph that defines how each program accepts data from previous computation and produces data for further processing. In Symphony, data-flow graphs are directed acyclic graphs (DAGs) with nodes defining components for data, programs, user interaction or visualization, and with connections between nodes representing data-flow paths. Connections are always created in the direction of desired data flow with the connection source bean being the **data source** and the connection target bean being the **data sink**. A connection is an **output connection** from the point of view of the data source bean and an **input connection** as seen by the sink bean.

A meta-program is composed by placing the required components on the workspace and creating connections which represent the desired data-flow patterns. Individual components need to be customized, but the customization may be done before or after the connections are made. Bean customization and linking is described in Section 3.2.

Table 3.3 shows the pairs of valid connections where source components are represented by columns and target components by rows. An empty field in the table indicates that a component of the source type cannot be directly connected to a component of the target type. If such a connection is attempted in the BeanBox, the attempt will fail and an error message will be displayed. Those fields filled with arrows show allowed direct connections.

Table 3.4 shows the number of input and output connections required and allowed for a bean. An attempt to exceed the number of allowed input or output connections for a bean will fail and an error message will be displayed. For beans that require an exact number of input or output connections, the check for the number of connections is done when the meta-program is verified at run-time. If the check fails at that time, the verification fails with an error message.

As seen in the table, only a Program bean can have more than one input connection and only Program and File beans can have more than one output connection. Also, a Producer bean

Table 3.3: Allowed Connections

Source Bean Type									
Program	File	Socket	Stdin	Stdout	Stderr	Consumer	Producer	Filter	Target Bean Type
	$\hookrightarrow^*$		$\hookrightarrow$						Program
$\hookrightarrow^*$	$\hookrightarrow$	$\hookrightarrow$		$\hookrightarrow$	$\hookrightarrow$		$\hookrightarrow$	$\hookrightarrow$	File
	$\hookrightarrow$			$\hookrightarrow$	$\hookrightarrow$		$\hookrightarrow$	$\hookrightarrow$	Socket
	$\hookrightarrow$	$\hookrightarrow$		$\hookrightarrow$	$\hookrightarrow$		$\hookrightarrow$	$\hookrightarrow$	Stdin
$\hookrightarrow$									Stdout
$\hookrightarrow$									Stderr
									Producer
	$\hookrightarrow$	$\hookrightarrow$		$\hookrightarrow$	$\hookrightarrow$			$\hookrightarrow$	Consumer
	$\hookrightarrow$	$\hookrightarrow$		$\hookrightarrow$	$\hookrightarrow$		$\hookrightarrow$	$\hookrightarrow$	Filter

cannot have an input connection and a Consumer bean cannot have an output connection. All stream beans and the Filter bean must have exactly one input connection and one output connection. If either of the connections is missing, the anomaly will be caught during meta-program verification. Also, although it seems that a Socket bean and File bean can exist without any connections, in effect they are useless because they cannot function independent of a Program bean. However, a Program bean may still function correctly without being connected to any other beans. There are two scenarios which are not immediately apparent from the table. A program bean can have a maximum of one Stdin, Stdout or Stderr bean. A socket bean can either have an input connection or an output connection, not both.

The following list helps in interpreting the data in tables 3.3 and 3.4 and, describes the logical significance of each possible pair of connections.

- **Program Bean:** The Program bean is the most important bean around which the rest of the meta-program is built. If the executable program represented by a Program bean reads input from it's standard input, a Stdin bean must be connected as input to the Program bean. The input to the Stdin bean can come from a File bean, a Stdout/Stderr bean of the previous program in the data-flow sequence, a Socket bean, or a Producer or Filter type bean. Thus, there are several ways in which data to be redirected to a program's standard input can be obtained.

  Likewise, if the Program bean writes to its standard output (standard error) and if the data is to be consumed by another component, a Stdout (Stderr) bean must be connected as an output bean to the Program bean. The output from the Stdout (Stderr) bean can go to a File, Socket, Filter or Consumer bean, or to the Stdin bean of the another program. Thus, there are several possible destinations to which data

Table 3.4: Number of Allowed Connections

	Input		Output	
Bean Type	Exact No. Required	Maximum Allowed	Exact No. Required	Maximum Allowed
Program	None	> 0	None	> 0
File	None	1	None	> 0
Socket	None	1	None	1
Stdin	1	-	1	-
Stdout	1	-	1	-
Stderr	1	-	1	-
Producer	0	0	1	-
Consumer	1	-	0	0
Filter	1	-	1	-

from a program's standard output (standard error) can be redirected.

Besides interacting through the standard streams, the legacy program may also take input or produce output through files which may be read or created during program execution. A direct connection between a File bean and Program bean in either direction (marked by asterisks in Table 3.3), is a bit different from other types of connections because it represents a logical connection rather than a physical connection and does not involve a direct data transfer between the connected beans in the BeanBox.

A $File \rightarrow Program$ connection means that the program represented by the Program bean will try to read a file from the location defined by the File bean when it executes. If the File bean is connected to another bean on its input, the file is created using data obtained from the input connection before the program executes. But if the File bean does not have an input connection, it is only required that the file be present before the program executes.

A $Program \rightarrow File$ connection means that the program will create a file at the location represented by the File bean during execution. This type connection serves two purposes: It is used to verify that the file is actually present after the program execution is done and, if this File bean is connected on its output to some other beans, those beans will be notified of the availability of the file after the program is done.

- **File Bean:** As explained above, when a File bean is connected directly to a Program bean it represents a file that is either read or created by the program during execution. However, when a file is connected on its input to some bean other than a Program bean, it serves to store the data received on its input connection in the file represented by the bean. It can be used to store data coming from a Filter, Producer, Stdout, Stderr, Socket or File bean. When a File bean is connected to some bean other than a Program bean on its output, it acts as an input bean. A File bean can provide input

to a Stdin, Filter, Consumer, Socket File bean. A $File \rightarrow File$ connection represents a distributed file copy.

One thing to remember about the File bean is that if the bean is connected on both sides, the data passing through the File bean is always copied over to the actual file represented by the bean. The data flow in this particular case is blocking since the File bean notifies its output bean(s) of data availability only after the data from its input bean is completely received.

- **Socket Bean:** If a Socket bean has an input connection it acts as an output socket and writes data received from the input connection to the socket represented. Similarly, if it has an output connection it acts as an input socket and encapsulates an input data stream from the socket. A Socket bean cannot both be a input and and output bean at the same time.

- **Stream Beans:** Each of the stream beans must have one input connection and one output connection. The output connection for an Stdin bean can only be to a Program bean and likewise, the input connections for Stderr and Stdout beans can only be from Program beans. A Stdin bean can take input from a File, Socket, Stdout, Stderr, Producer or Filter bean. A Stdout/Stderr bean can send its output to a File, Socket, Stdin, Consumer or Filter bean.

  At run-time, a stream bean just acts like a pipe through which data flows unmodified. The data flow through the stream beans is streaming rather than blocking. The Stdin bean notifies the Program bean to start executing as soon as data is available for reading. Likewise, the Stderr and Stdout beans notify their output beans to start reading data as soon as the Program bean notifies them of the start of execution.

- **Consumer Bean:** Since the only function of a Consumer bean is to read and consume data obtained on its input connection, it must have exactly one input connection and it cannot have an output connection. When the meta-program executes, the bean starts reading data from its input connection when it receives notification that data is available.

- **Producer Bean:** Since a Producer bean is responsible only for producing data for further processing, it must have exactly one output connection. No input connection is allowed. It starts producing data when it receives notification from its output bean to do so.

- **Filter Bean:** The function of a Filter bean is to read data from its input connection, process it, and make the processed data available to its output bean for reading. Hence, it must have exactly one input connection and one output connection. The data flow through a Filter bean is streaming rather than blocking, since notification is sent to the output bean as soon as the filtering operation starts. Upon receiving the notification, the output bean can immediately start reading the available data.

3.5 Meta-Program Operations

Symphony provides three operations that can be performed on a meta-program after it has been composed: verify, execute and stop. These operation can be initiated by using the popup-menu provided by a Program bean which appears when the mouse is clicked on the bean. If there are multiple Program beans in the meta-program, any of them can be used to start an operation. Symphony provides visual feedback to the user during any meta-program operation so that the progress of the operation can be assessed at any point in time. The following sub-sections describe the mechanism used by Symphony for visual feedback and details about each individual operation.

Bean States and Visual Feedback

Every Symphony bean passes through a fixed number of states during its lifecycle. The state of a bean changes during a meta-program operation. The state changes for a bean are displayed to the user by painting the bean background with different colors for different states. Thus, the bean changes color whenever a state transition occurs. Table 3.5 lists the possible status values for a bean along with the corresponding color codes for each state.

Table 3.5: Colors Corresponding to Bean Status

Bean Status	Color
NOT_VERIFIED	Light Blue
VERIFIED	Gray
READY	Green
RUNNING	Magenta
SUSPENDED	Blue
COMPLETED	Black
ABORTED	Red

The initial status value for every bean is CREATED. Whenever the attributes of the bean are customized, the status value is changed to NOT_VERIFIED. These two states are described by the same color in the user interface for simplicity. There is no logical difference between a bean that is in state CREATED or state NOT_VERIFIED. Both status values mean that the bean must be first verified in order to perform any operation on it. The remaining status values are explained in the following sub-sections.

Verify Operation

It is possible to verify a meta-program's structural integrity and whether all beans have been customized correctly for the resources they are supposed to represent. The former includes structural integrity constraints such as those dictated by the exact number of input and output connections required for a bean, as defined in Table 3.4. Verifying the customization includes checking the validity of host names and user account information for remote resources, existence and accessibility of local and remote programs and files, and existence of remote servers needed to execute remote programs. This operation is termed the **verify operation**. If the verify operation is successful for a bean, its status value is changed to VERIFIED, indicated by the gray color. Thus, at the end of a successful verification operation, all beans in the meta-program should have a gray background color.

Execute Operation

Once a meta-program has been verified, it is possible to execute each of the its executable components in the sequence defined by the data-flow patterns in the meta-program. This operation is termed as the **execute operation**. Thus, for example, if Program2 depends on input data that is produced by Program1, Program2 must be executed only after data is available from Program1. There are two possibilities depending on whether the two programs are connected by files (persistent storage) or by direct data transfers (stdin and stdout streams). If data produced by Program1 is placed in persistent storage before it can be consumed by Program2, Program2 can be started only after Program1 has finished execution and Program1's output data has been completely produced. Alternatively, if persistent storage is not required, data can be streamed between the two programs and Program2 can be started immediately after Program1 is started. However, these data-flow mechanics and program activation logic are largely transparent to the user. For each Program bean, the execute operation follows the simple algorithm shown in Figure 3.8. After a successful execute operation, all beans will have the COMPLETED status indicated by the color black.

Stop Operation

It is possible to abort the execution of a meta-program by choosing the "Stop" menu-item from the Program bean popup-menu. The stop operation aborts all data transfer and program execution, closes all open network connections and files and destroys all execution threads. At the end of a stop operation, all beans are in the ABORTED status indicated by color red.

Figure 3.8: Algorithm for Execute Operation

```
initiate a verify operation if status is not VERIFIED

inform all input connections to get ready
    if there is an upstream Program bean this Program bean
    will  be suspended and an execute operation started
    for that upstream program. When the upstream Program
    bean is done it will notify this Program bean that
    it is done so that this execute operation can be
    restarted

start program execution
   if it is a user accessible program and a Stdout or Stderr
   beans is connected to this Program bean, inform it that
   data is available

wait for the program execution to complete

inform all output connections that the program is done
   if there is a downstream Program bean its execution
   will be started
```

3.6 Implementing Abstract Beans

This section describes how a Java programmer can add new bean types to the environment by extending one of the abstract beans provided by Symphony. As mentioned before, extending an abstract Symphony bean does not require any knowledge of the Symphony or JavaBeans architecture. Each abstract bean has a simple set of abstract methods that need to be implemented for creating a new bean. There are three important abstract methods that need to be implemented for each bean corresponding to each of the three meta-program operations. The following sub-sections explain the procedure in detail and give sample code to illustrate the steps.

3.6.1 Producer Beans

The Producer bean can be extended to implement a variety of beans, such as beans that solicit input data from the user, beans that obtain data from remote databases, and beans that interact with remote compute servers implemented using CORBA [19] or RMI [31]. The manner in which the data is produced or obtained by the bean is left to the discretion of the programmer. The only requirement is the data be presented in a byte stream represented by a `java.io.InputStream` object which can be read from.

Figure 3.9 shows the outline of a Java class that extends the abstract Producer class. The Producer class defines four abstract methods that need to be implemented in order to create a new Producer bean. The `getBeanTypeName()` method is a utility method which must return a descriptive name for the bean. For example, the Parameters bean returns the string "Parameters". The remaining three methods correspond to the three operations that can be performed on a meta-program. The `verify()` method is invoked during a verify operation and must be used to verify if the bean is ready for producing data. The `start()` method is invoked during an execute operation, when the bean receives notification from its output bean to start producing data. This method must initiate data production, encapsulate it in a byte stream represented by a `InputStream` object and return a reference to the object. Finally, the `stop()` method is invoked during a stop operation and must abort any activity in the bean (e.g., close connections to remote servers, destroy any active threads, close any input file or streams, close any open input dialogs, etc.)

3.6.2 Consumer Beans

A Consumer bean can be used for a variety of purposes, including data visualization and data storage. Figure 3.10 shows the outline of a Java class that extends the abstract Consumer bean. The implementation of a consumer is similar to that of a Producer bean in terms of the set of methods that need to be implemented except that the `start()` method is

Figure 3.9: Implementing a Producer Bean - NewProducer.java

```
package symphony.beans.newproducer;
import symphony.beans.producer.Producer;

public class NewProducer extends Producer {

   public NewProducer () { ... }
   public String getBeanName() { ... }
   public boolean verify() { ... }
   public InputStream start() { ... }
   public void stop() { ... }
}
```

Figure 3.10: Implementing a Consumer Bean - NewConsumer.java

```
package symphony.beans.newconsumer;
import symphony.beans.consumer.Consumer;

public class NewConsumer extends Consumer {

   public NewConsumer() { ... }
   public String getBeanName() { ... }
   public boolean verify() { ... }
   public void start(InputStream in) { ... }
   public void stop() { ... }
}
```

invoked during the execute operation when the input bean informs the Consumer bean of the availability of data. Data must be read from the `InputStream` object obtained as a parameter of the `start()` method. The purpose and use of other methods is similar to that defined for the Producer bean.

3.6.3 Filter Beans

A Filter bean can be used for implementing data filters, file format converters, data sorters, and for other tasks which require the transformation of data from one format to another format. The set of methods to be implemented for a Filter bean is very similar to that of the Consumer and Producer beans and is shown in Figure 3.11. The most important method

Figure 3.11: Implementing a Filter Bean - NewFilter.java

```
package symphony.beans.newfilter;
import symphony.beans.filter.Filter;

public class NewFilter extends Filter {

   public NewFilter () { ... }
   public String getBeanName() { ... }
   public boolean verify() { ... }
   public filter (InputStream in, OutputStream out) { ... }
   public void stop() { ... }
}
```

that must be implemented is the `filter()` method which is invoked when input data is available. In the implementation, input data must be read from the `InputStream` object passed as a parameter of the `filter()` method and filtered output data must be written to the `OutputStream` object obtained as the second parameter of the `filter()` method. In order to make the data filtering process non-blocking the `filter()` method must use a separate thread for transforming the data. The rest of the methods in the bean are similar to those in the Producer and Consumer beans.

3.7 Meta-Program Example

This section describes a sample meta-program based on the Radio Frequency Pressing (RFP) simulation developed at the Department of Wood Science and Forest Products at Virginia Tech [22]. This system simulates heat and mass transfer in wood when subject to power input by an alternating electric field. The simulation, implemented in Fortran, takes 64 input parameters as a specially formatted file, of which 11 parameters (such as thickness of wood specimen, strength and frequency of electric field, initial temperature, etc.) must be specified by the user to run the simulation; the remaining parameters have default values which may also be changed. The simulation produces output files containing temperature, pressure and moisture data for the wood specimen with respect to time. This output data can be visualized as a 3-dimensional graph. For example, the temperature inside the wood specimen at any point in time during the simulated experiment can be visualized as a 3D graph of temperature vs. position vs. time. However, the output files produced by the simulation cannot be used directly for creating the required graphs. It has to be first converted into a format that a standard 3D-graph plotting algorithm can understand. Thus, there are three major steps to running the simulation and visualizing the results:

- Obtain the input parameters from the user and create the input file
- Execute the simulation
- Convert simulation output to the required format and visualize the results

Figure 3.12 shows the meta-program constructed to execute this simulation. The screen-shot has been annotated by bean types, wherever necessary.

Figure 3.13 shows a conceptual model of the meta-program configuration. As shown in the figure, the simulation program (denoted by the Program bean labeled "Fortran Simulation" in the meta-program) resides on host machine B. The input file needed by the simulation (File bean labeled "Input Data File" in meta-program) and the produced output files also reside on host B. Although not necessary, for purpose of illustration, the script for converting the input values to a formatted file (Program bean labeled "Script1") and the script for converting the simulation output data to 3D graph data (Program bean labeled "Script2") have been placed on different host machines than the simulation program. Both Script1 and Script2 read data from the standard input stream and write output to the standard output stream. Also, for simplicity, the pressure and moisture output from the simulation has been ignored in the meta-program. Only the temperature output is considered. The RFPInput bean shown in the meta-program, created by extending the abstract Producer bean, provides a dialog box for accepting input data values for the simulation. Similarly, the WireFrame bean, created by extending the abstract Consumer bean, accepts 3D-graph data in the specified format and displays a rotatable 3D wireframe graph in a separate window. The simulation program reads its parameters from the standard input stream which are obtained from the user by using the Parameters bean (created from the abstract Producer bean) described in Section 3.3.2.

The following actions take place when the meta-program is executed from the composing environment, assuming that the execute operation is initiated from the Program bean labeled 'Script1'.

1. The Script1 Program bean asks the RFPInput bean to get ready, which in turn displays a dialog box for accepting the simulation parameters from the user. Figure 3.14 shows the dialog box. The user modifies the default parameter values and clicks the "Run Simulation" button. The RFPInput bean returns a ready status to the Script1 bean.
2. Script1 contacts the Symphony server on host A and starts execution of the Perl script for converting the parameters obtained from the RFPInput bean to the required file format. It redirects data from the RFPInput bean to the standard input stream of the remote program, which produces the file data on its standard output. Script1 also redirects the output data to the File bean labeled "Input Data File" creates data file on host B.

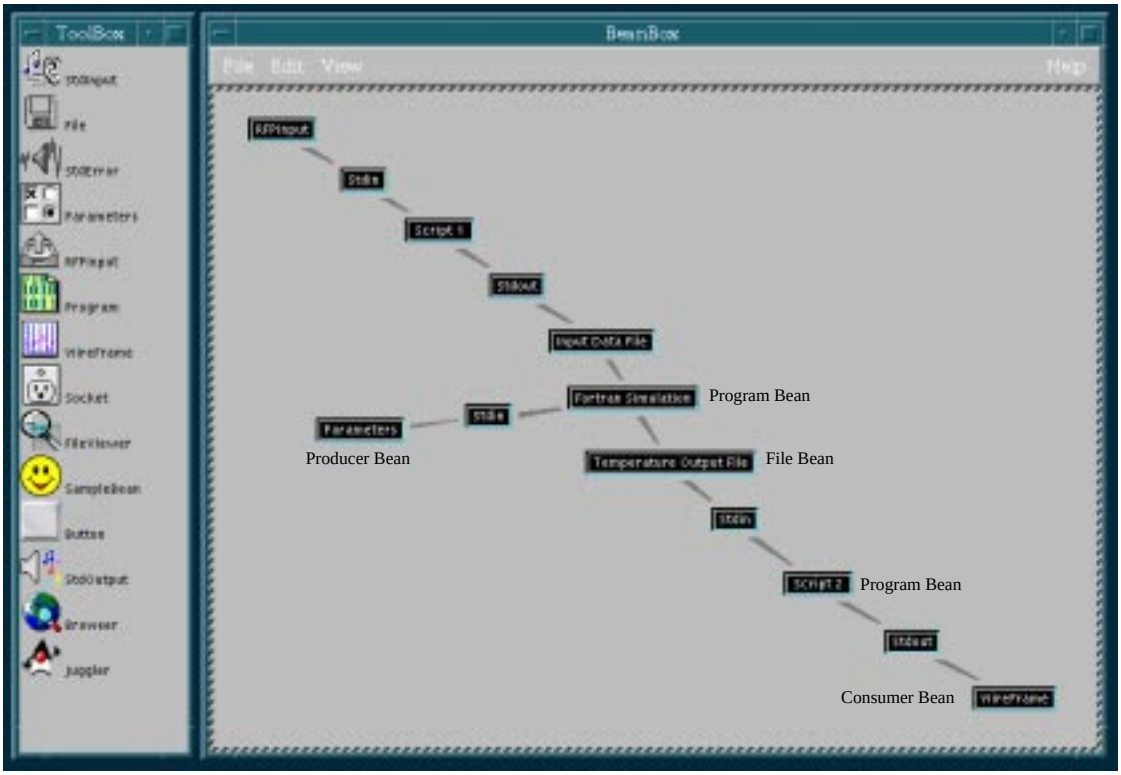

Figure 3.12: Meta-program for the Wood-based Composites RFP Simulation

3. Script1 sends a message to the "Fortran Simulation" Program bean to start execution, which in turn, asks the Parameters bean to get ready for execution. The Parameters bean reads the parameters description file from the defined URL and creates the user interface shown in Figure 3.7. The user modifies the default parameter values if needed and clicks the "Submit" button. The parameters bean returns a ready status to the Simulation bean.

4. The Simulation bean contacts the Symphony server on host B to start execution of the Fortran simulation program, and redirects the data obtained from the Parameters bean to the standard input stream of the simulation program. The simulation program writes output to the standard output stream which is displayed in a dialog box that appears when the program starts execution. The remote program creates several output files, one of which is the file defined by the File bean labeled "Temperature Output File".

5. After the simulation program is done and the output files have been completely created, the Simulation bean sends a message to the Program bean labeled "Script2" to begin execution. The Script2 bean contacts the Symphony server on host C and start execution of the Perl script for converting the temperature output data to 3D

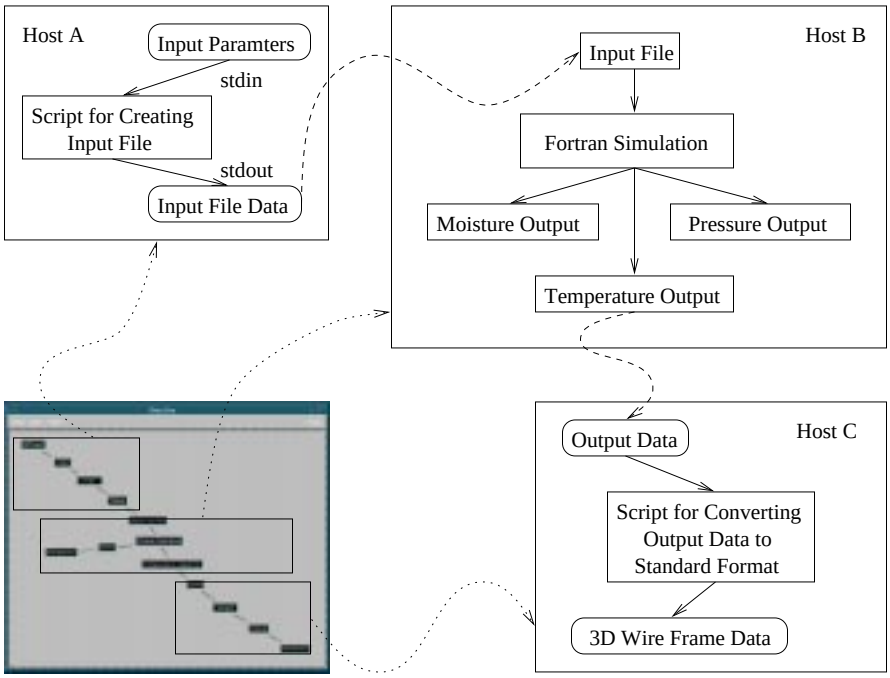

Figure 3.13: RFP Simulation Meta-Program Configuration

wireframe data. It redirects data from the "Temperature Output File" bean to the standard input of the remote script. After starting the execution of the remote script, the Script2 bean informs the WireFrame bean of the availability of data and gives it a handle to the standard output stream from the program.

6. The WireFrame bean reads the data stream obtained from the Script2 bean and displays the 3D WireFrame graph shown in Figure 3.15. This data transfer is a streaming transfer where the WireFrame bean reads data as it is being produced by the remote script. The graph can be rotated by dragging the mouse pointer in the window, with the left mouse button pressed.

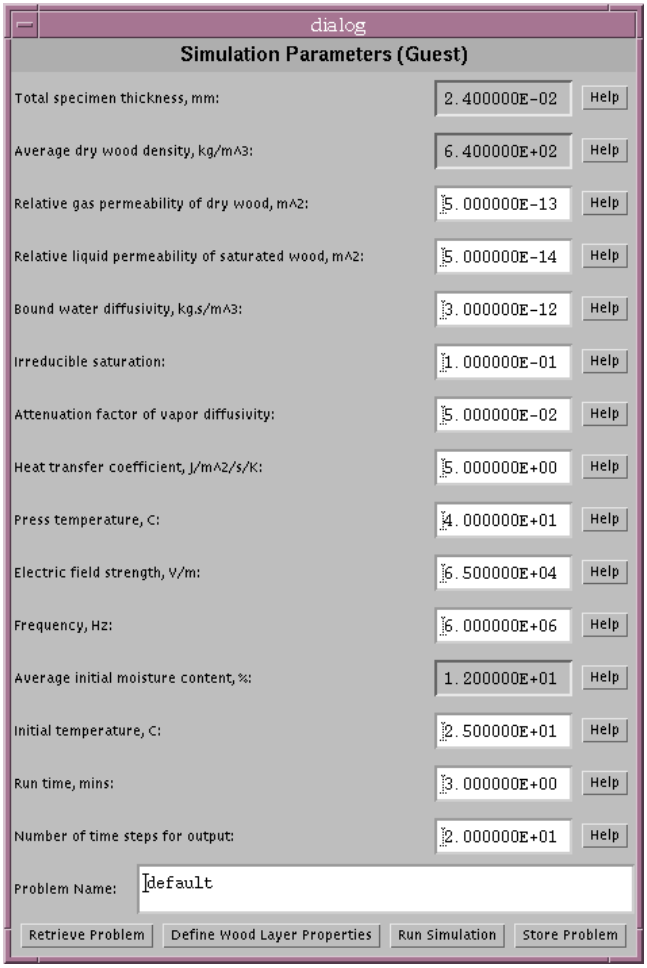

Figure 3.14: User Interface Created by the RFPInput Bean

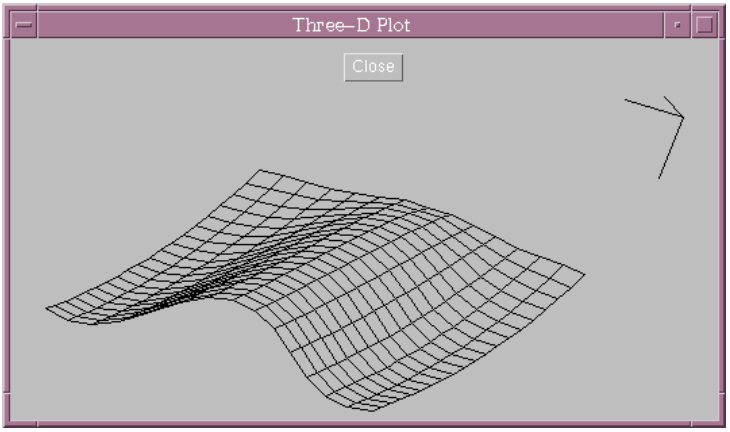

Figure 3.15: 3D Wire Frame Graph of Simulation Results

Chapter 4

Design and Implementation

This chapter discusses the design and implementation of the Symphony framework. Section 4.1 outlines the design goals which guided the implementation of the framework. Section 4.2 describes the JavaBeans architecture and presents the complete implementation of a sample bean that is a simplified Symphony bean.

All beans in a meta-program must communicate in order to carry out the various operations on the meta-program. Section 4.3 describes the building blocks of the communication framework used by Symphony beans and how the framework is used to hide the communication complexity from the user. Section 4.4 presents the communication protocol used by the beans to carry out meta-program operations defined in Chapter 3. An important part of the Symphony framework is the Symphony server that is used for executing programs on remote machines. The server has been implemented as a remote object on which client beans can make method invocations by using the Java Remote Method Invocation (RMI) mechanism. Section 4.5 explains the server implementation and provides sufficient details about the Java RMI mechanism to understand the implementation.

Finally, Section 4.6 and 4.7 describe the implementation of the Symphony beans and Section 4.8 shows how a new bean can be added to the framework and how Symphony beans were adapted for collaborative composition in Sieve [13].

4.1 Design Goals

The Symphony framework consists of a set of beans which can be connected together by the user to form meta-programs. When executed, beans interact with adjacent beans and in some cases with remote servers to carry out the operations implied by the nature of the bean and its connections with other beans.

An important requirement for any composing environment is to provide generic components

which can be connected together intuitively and which can communicate and interact naturally when executed. Thus, the biggest design challenges for Symphony were deciding the right set of components and the right ways to connect them together. Some design goals for Symphony stem from the goals set for the system itself (Section 1.2) and some were derived from accepted software engineering practices. The following list outlines the design goals and explains how they are accomplished in Symphony.

- **INTUITIVE MODEL:** The model for composing a meta-program must be intuitive for the user and easy to learn. In Symphony, a meta-program is modeled as a directed acyclic graph (DAG). Data-flow channels describe data dependencies of executable components. Executable components accept input data along input channels and push output data along the desired output channels. Thus, the meta-program can be composed intuitively in the direction of data-flow.

- **EASE OF COMPOSITION:** In addition to being intuitive, the composition process must entail a minimum amount of work for the user. This is especially important for large meta-programs. Symphony beans provide reasonable defaults for as much behavior as possible and minimize the amount of information the user has to provide in order to customize each bean. The Properties bean described in Chapter 3 was included for this purpose.

- **VISUAL FEEDBACK:** It is important to provide visual feedback representing the underlying actions being performed during any meta-program operation. This helps the user in assessing the progress of the operation, and if the operation fails it helps in tracking the point of failure so that appropriate remedial actions may be taken. This is accomplished by having well-defined states that every bean passes through or attains during each operation and by reflecting the state of a bean at any time in the user interface. Mechanically, the bean is painted with a different color for each different state.

- **PERSISTENCE:** A goal for the framework was to be able to save a composed meta-program in persistent state so that it can be reloaded at a later time for execution and modification, possibly at a different site. This was achieved by maintaining serializability of beans as defined by the JavaBeans architecture.

- **INDEPENDENCE:** The Symphony framework is independent of the composing environment, allowing the beans to be used without modification in any standard bean container. Symphony beans were designed with this goal in mind and implemented without depending on the underlying BeanBox environment in which they were tested. Another aspect of independence was to keep the beans independent of the underlying operating system at the client as well as the server sites.

- **EFFICIENCY:** As a general principle, less overhead should be imposed when accessing local resources as opposed to remote resources. This principle translates well into

the implementation of Symphony beans which can represent local as well as remote resources. For example, a remote file is accessed through the FTP mechanism, but a local file is accessed simply by using the Java classes for accessing the local file system. Also, the system uses a streaming mode of data transfer whenever possible so that unnecessary delays created due to blocking transfer can be reduced.

- **SIMPLICITY:** In order to make a data-flow based composing environment generic, the data flowing through the system must conform to a simple format that is understood by all components. In Symphony, instead of prescribing a specific data format on the data flowing through the system, all data is represented by raw byte streams. Also, for simplicity, Symphony follows a data-pull model, where the data sink bean asks the data source bean for a input data stream at the appropriate time. Data is never explicitly pushed by a source bean to a sink bean.

- **EXTENSIBILITY:** For any software system to be useful, it must ease maintainence and provide a framework which can be extended with minimum effort. Symphony addresses this requirement in three ways. First, the use of JavaBeans architecture, enhances extensibility because beans are independent software components. Secondly, Symphony provides abstract beans which can be extended by Java programmers for adding new beans to the framework with minimal knowledge of the Symphony internals or of the JavaBeans architecture. Finally, the beans themselves have been implemented so as to have minimum dependence on each other which makes it easy to add completely new beans to the existing set of core and abstract beans.

For some of the goals outlined above, the manner in which these were accomplished should be apparent from the accompanying description and from the Symphony description in Chapter 3. For others, it should become clear by the end of this chapter.

4.2 Implementing a Bean

This section describes the implementation level details of a Java bean, using as an example a simplified version of a Symphony bean, named SampleBean. A bean is a Java class that follows certain design patterns and method naming conventions to publish its properties, methods and events [33]. While a bean is not a complete application, it can be composed with other beans to form applications. A bean has several capabilities that support its use as a stand-alone, composable software component: it provides support for introspection so that a builder tool can discover its properties, methods, and events; it lets the user customize its appearance and behavior through the builder tool; it enables persistence, so that a customized bean can be saved away and reloaded later for use or modification.

Although not required, most beans in practice have a visual representation and inherit from a sub-class of the `java.awt.Component` class. To create and support the bean's user in-

Figure 4.1: A Sample Bean Implementation

```
public class SampleBean extends Panel implements Serializable,
  PropertyChangeListener {

    private String label  ;
    private int    status ;
    private Vector connectionListeners = new Vector() ;
    private PropertyChangeSupport changes = new PropertyChangeSupport(this);

    public SampleBean () {
        this ("SampleBean") ;
    }
    public SampleBean(String label) {
        setStatus(0) ;
        setLabel(label) ;
        addPropertyChangeListener(this) ;
    }

    public synchronized void addConnectionListener(ConnectionListener l)
      throws Exception {
        connectionListeners.addElement(l);
    }
    public synchronized void removeConnectionListener(ConnectionListener l)
      throws Exception {
        connectionListeners.removeElement(l);
    }
    public void eventSend (PortEvent x)
      throws EventException {
    }

    public void addPropertyChangeListener(PropertyChangeListener l) {
       changes.addPropertyChangeListener(l);
    }
    public void removePropertyChangeListener(PropertyChangeListener l) {
       changes.removePropertyChangeListener(l);
    }
    public void propertyChange(PropertyChangeEvent evt) {
       setStatus(1) ;
    }

    public void setStatus(int newStatus) { status = newStatus ; }
    public int getStatus() { return status ; }

    public void setLabel(String newLabel) {
       String oldLabel = label ;   label = newLabel ;
       changes.firePropertyChange("label", oldLabel, newLabel ) ;
    }
    public String getLabel() {
       return label ;
    }
}
```

terface the bean may override methods from the Component class (e.g., getBackground(), getPreferredSize(), paint(), etc.) and implement event handling methods for mouse, keyboard, and other GUI-related actions (which entails implementing various interfaces from the `java.awt.event` package). Figure 4.1 shows the code for the `SampleBean` class which extends the `java.awt.Panel` class. For simplicity, the code shown does not include methods for creating or managing the bean's user interface.

4.2.1 Events

Beans communicate using events. The event mechanism provides a client/server, application independent form of communication. Traditionally, events have been used for common tasks such as delivering notifications of mouse and keyboard actions in window system toolkits. The JavaBeans event mechanism derives from the event model used by the Java Abstract Windowing Toolkit (AWT) [30].

Events are a way for a source bean to notify listener beans that something of interest has happened. Events are generated by event sources and sent to event listener objects by making Java method invocations. Event handling methods for a class of events are generally grouped together into event listener interfaces that inherit from java.util.EventListener (e.g., all event handling methods for keyboard actions such as key pressed, key released, and key typed are grouped into the `java.awt.event.KeyListener` interface). The state associated with an event notification is normally encapsulated in a event state object that inherits from `java.util.EventObject` and which is passed as the single argument to the event handling method. Event handling methods can throw exceptions which are declared in the method signature. These methods must conform to the following method syntax

```
void <eventOccured> (<EventStateObjectType> evt)
     [throws <exceptionType1>, <exceptionType2>, ...] ;
```

A bean identifies itself as being interested in a particular set of events by implementing one or more event listener interfaces. Likewise, a bean identifies itself as an event source for a particular set of events by providing registration methods for interested listener beans. One or more listeners can register with a source bean to be notified about the events of a particular kind that occur in that bean. If an event listener does not directly implement a particular listener interface, an instance of an adaptor class which implements the required interface may be placed between the source and listener. In this case, instead of registering the actual listener object with the source bean, an instance of the adaptor is registered. When the event is fired, the adaptor forward the event notification to the actual listener object.

Figure 4.2 depicts the bean events mechanism. If the target object implements the XListener interface, the adaptor is not necessary. The dotted line on the event object passed from the

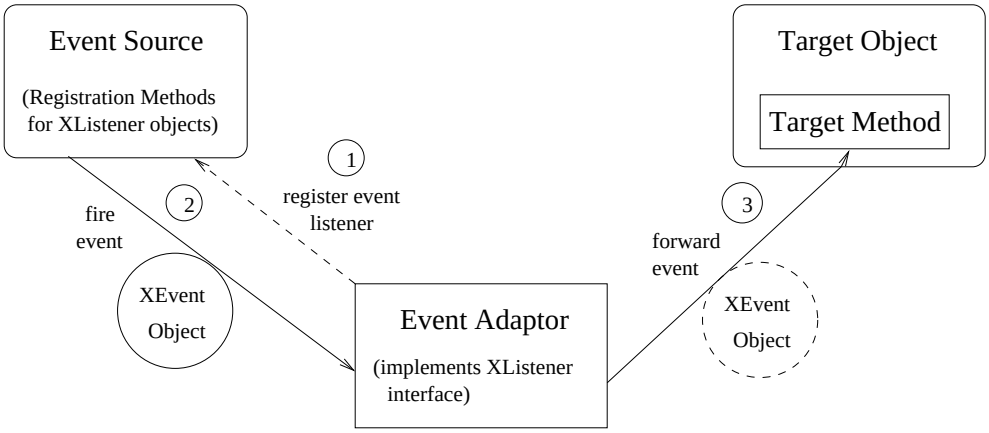

Figure 4.2: Beans Event Mechanism

adaptor to the target bean indicates that the object may or may not be forwarded depending on whether the target method accepts a parameter or not. The BeanBox discovers, at run-time, the events that can be generated by a particular bean by inspecting the listener registration methods it implements. Suppose a bean in the BeanBox workspace implements registration methods for XListener objects (`addXListener()` and `removeXListener()`), where the XListener interface contains a single target method, `sendX()`. When this bean is selected in the workspace, the BeanBox discovers that it can generate events of type X which are to be fired on XListener objects and it adds the sub-menu $X \rightarrow sendX$ under the BeanBox menu $Edit \rightarrow Events$. As described in Section 3.2, this menu-item can be used to connect the generation of the X event in the selected bean to a method invocation in some other bean.

The sample bean shown in Figure 4.1 implements registration methods for the `Connection` event and allows registration of `ConnectionListener` objects. These methods would normally have some pre-processing code for connection initialization and validation which has been removed for simplicity. Assuming that the `ConnectionListener` interface has a single method of the form

```
public void createConnection (PortEvent x) throws EventException ;
```

for firing the connection event from a source bean to a listener bean, the listener bean must implement a event handling method that accepts a `PortEvent` object as a single parameter. The `SampleBean` class implements the `eventSend()` method on which this event can be fired. By default all of a bean's public methods are exported. So the BeanBox can discover this method at run-time for presenting as a target method in the event target dialog while creating a connection between two beans in the BeanBox (see 3.2).

4.2.2 Properties

A property is a named attribute which can be read and/or modified. There are several types of properties: simple, indexed, bound, and constrained. Indexed and constrained properties are not used in Symphony. Simple properties represent single values and are defined by a pair of accessor methods. A property's name is inferred by the method names. For example, methods named `setFoo` and `getFoo` indicate a property named `foo`. It must be noted that the first alphabet of the property name is capitalized in the method name. A bound property is capable of notifying other objects when its value changes. Each time its value is changed, the property fires a `PropertyChangeEvent` which contains the property name, and old and new values of the property. A bean that wishes to implement bound properties must implement the `addPropertyChangeListener()` and `removePropertyChangeListener()` methods for registration of interested listeners. Also, a bean that wishes to receive bound property change notifications must implement the `java.beans.PropertyChangeListener` interface.

The `SampleBean` class implements one simple property (`status`) and one bound property (`label`) as seen in Figure 4.1. For supporting the `label` property, the `SampleBean` class implements registration methods for `PropertyChangeListener` objects. The JavaBeans API provides the `PropertyChangeSupport` class for managing registration of bound property listeners and for firing bound property events on registered listeners. `SampleBean` uses a `PropertyChangeSupport` object for the `label` property.

`SampleBean` also implements the `PropertyChangeListener` interface which consists of the single method `propertyChange()`). As can be seen from the constructor of the class, a `SampleBean` object is registered with itself as a property change listener for the `label` property. Thus, a property change event is fired every time the `setLabel` method is invoked, which in turn has the effect of setting the value of the `status` property to 1.

4.2.3 Introspection

Introspection is the process of discovering the properties, events, and methods supported by a bean. Beans support introspection in two ways: by using the Java class reflection mechanism, by providing an explicit BeanInfo class. The BeanInfo class provides a way for the bean implementor to control the information that is exposed to the containing environment.

For discovering properties and events through reflection, the introspection mechanism relies on the bean to use standard method signatures. The required method signatures for automatically discovered simple properties and events are shown below:

```
// === For a simple property named <propertyName> ===
public <PropertyType> get<PropertyName>();
public void set<PropertyName> (<PropertyType> x);
```

```
// === For a class of events where <EventListenerType> ===
// === extends java.util.EventListener and where the   ===
// === <EventListenerType> name ends with 'Listener'   ===

public void add<EventListenerType>(<EventListenerType> x);
public void remove<EventListenerType>(EventListenerType> x);
```

By default all of a beans public methods are exposed as external methods which can be called from the environment.

A bean can choose to export only a subsets of its properties, events, and methods by providing an explicit bean information class which implements the `java.beans.BeanInfo` interface. This is also useful if the bean's methods do not follow the standard naming conventions for automatic introspection. The name of the BeanInfo class must be formed by adding "BeanInfo" to the bean class's name. All Symphony beans define a BeanInfo class to restrict access to exposed information and to simplify the user interface. Figure 4.3 shows the BeanInfo class for the SampleBean class. The BeanInfo classes for Symphony beans are similar to this class.

The `BeanInfo` interface provides methods which can be implemented to return descriptions of desired events, properties, and methods, and additional description about the bean. Events, properties, and methods are described by EventSetDescriptor, PropertyDescriptor, and MethodDescriptor objects, respectively. A BeanDescriptor object provides overall information about a bean, including its Java class name, its display name, etc. The BeanInfo class can choose to provide partial information about the bean and leave the rest for automatic introspection. For example, if the `getMethodDescriptors()` method returns a null, that is an indication to the introspection mechanism that it should use the default class reflection mechanism for discovering the methods of this bean. The JavaBeans API provides a utility class, `java.bean.SimpleBeanInfo`, which provides an implementation of the `BeanInfo` interface where every method simply returns a null value. This class can be extended to provide the BeanInfo class for any bean.

The class shown in Figure 4.3 prevents any properties of the `SampleBean` from being exposed to the environment. If automatic introspection of events is allowed for the SampleBean, in addition to the connection event and the property change event, event types such as mouse and keyboard events supported by the `java.awt.Panel` are also exposed. The BeanInfo class prevents this and exposes only the connection event for providing a clean user interface. Similarly, automatic introspection of methods on the SampleBean exposes the entire set of public methods in the SampleBean class and in all its parent classes (including the `Panel` and `Component` classes). Since the `java.awt.Component` class has over hundred public methods, automatic exposure of all methods would make it difficult to search for the right method during event connections. Through the BeanInfo class, a single method which can act as the target method for the connection event is exposed and all other methods are suppressed. The `getBeanDescriptor` method returns a BeanDescriptor objects which informs the environment about the bean's Customizer object. Customizers are discussed in

the next sub-section. The `getIcon` method return the Image object representing the icon which is displayed besides the bean's name in the tool box when the BeanBox application is started.

4.2.4 Customization

The bean allows customization of its properties in two ways. If it exports a set of properties, then the application builder tool uses them to construct a GUI property sheet that lists the properties and provides a property editor for each property. The user can use this property sheet to update the properties. Alternatively, if more sophisticated customization such as property grouping and validation is needed, the bean can provide its own customizer. The customizer should be an AWT component that can provide the GUI for customization of the target bean. It should also implement the `java.beans.Customizer` interface. A bean that implements a customizer must provide its own BeanInfo class. The builder tools can use the `beanInfo.getCustomizerClass()` method to locate the bean's customizer class.

A bean's customizer can be accessed from the BeanBox by using the *Edit* → *Customize...* menu-item when the bean is selected in the workspace. When this menu-item is selected, the BeanBox instantiates a new object of the bean's customizer class, puts it into a new AWT Dialog object and displays the dialog. The `java.beans.Customizer` interface contains the `setObject` method which is invoked by the BeanBox to initialize the customizer. A reference to the target bean is passed to this method as a parameter. The customizer should use this method to create the GUI and initialize its fields.

Figure 4.5 shows the customizer class for the sample bean. As required, it inherits from `java.awt.Panel` and implements the `java.beans.Customizer` interface. Since the bean's BeanInfo class (shown in Figure 4.3) suppresses any property from being exposed, the customizer is used to set the value of the bean's `label` property. Figure 4.4 shows a screen shot of the customizer dialog produced by the SampleBeanCustomizer class. In practice, you wouldn't need to implement a customizer for providing access to a a simple string property. This is just a hypothetical example. The manner in which the customizer is implemented, the user would have to press the "Apply Customization" button after typing in the new property value for the change to take effect.

4.2.5 Persistence

The builder tool can save the beans along-with their connections in persistent state using Java object serialization [29]. Serialization is the process of saving in persistent storage, the internal state of a Java object along with the associated graph of objects. Deserialization is the reverse process of constructing an object from the serialized description.

A bean is made persistent by implementing the `java.io.Serializable` interface. The

Figure 4.3: BeanInfo Class for the SampleBean

```
public class SampleBeanBeanInfo extends SimpleBeanInfo {

  private final static Class beanClass = SampleBean.class ;
  private final static Class customizerClass = SampleBeanCustomizer.class ;

  public PropertyDescriptor[] getPropertyDescriptors() {
     PropertyDescriptor rv[] = { } ;
     return rv;
  }

  public EventSetDescriptor[] getEventSetDescriptors() {
     try {
        EventSetDescriptor eDesc = new EventSetDescriptor(
                                          beanClass, "connection",
                                          ConnectionListener.class,
                                         "createConnection" );
        EventSetDescriptor[] rv = { eDesc } ;
        return rv;
     } catch (IntrospectionException e) {
        throw new Error(e.toString());
     }
  }

  public MethodDescriptor[] getMethodDescriptors () {
     try {
        Method m1 = SampleBean.class.getMethod ( "eventSend",
                    new Class[] { PortEvent.class } ) ;
        MethodDescriptor md1 = new MethodDescriptor ( m1 ) ;
        MethodDescriptor[] methods = { md1 } ;
        return methods ;
     }
     catch ( Exception e ) {
        return super.getMethodDescriptors() ;
     }
  }

  public BeanDescriptor getBeanDescriptor() {
     return new BeanDescriptor(beanClass, customizerClass);
  }

  public java.awt.Image getIcon(int iconKind) {
     if (iconKind == BeanInfo.ICON_MONO_32x32 ||
        iconKind == BeanInfo.ICON_COLOR_32x32 ) {
        java.awt.Image img = loadImage("sampleIcon.gif");
        return img;
     }
     return null;
  }
}
```

Figure 4.4: Customizer Dialog for SampleBean

SampleBean class implements the Serializable interface as shown in Figure 4.1. This is an empty interface and serves only to notify the builder tool that this bean has been marked for persistence. The fields in any instance of the bean will be saved during the serialization process. Field that are of type `transient` and `static` are not serialized. Typically, a bean should save the state of any exposed properties and relevant internal variables. It should not, however, store references to external beans.

4.2.6 Packaging

JavaBeans can be packaged in JAR (Java ARchive) files. A JAR file is a ZIP format archive file that may optionally have a Manifest file [32]. The JAR file may contain .class files, serialized beans, help files in HTML format, and resources (images, audio, text). The Manifest file describes the contents of a JAR file and can be used to indicate which classes in the JAR file constitute beans. All Symphony beans are packaged in a single JAR file named Symphony.jar which can be loaded into the BeanBox via the $File \rightarrow LoadJarFile$ menu-item.

4.3 Communication Framework

This section describes the basic building blocks of the Symphony communication framework and explains how Symphony achieves compositional generality while keeping the user interface simple.

Any bean that is part of a meta-program must have the ability to communicate with beans to which it is connected for carrying out the various run-time operations described in Section 3.5. As seen from Table 3.3, it may be possible to connect a particular type of bean as an input bean to multiple types of output beans, and conversely, it may accept input connections from multiple types of input beans. However, for a given data connection, neither the source bean nor the target bean should have to know the internal workings of each other in order to

Figure 4.5: Customizer Class for the SampleBean

```
public class SampleBeanCustomizer extends Panel
       implements Customizer, ActionListener {

   private SampleBean    bean;
   private TextComponent label;
   private Button        doneButton;
   private PropertyChangeSupport listeners = new PropertyChangeSupport(this);

   // The BeanBox calls this method to tell us what object to customize.
   // This method will always be called before the customizer is displayed,
   // so it is safe to create the customizer GUI here.

   public void setObject ( Object obj )  {

     bean = ( SampleBean ) obj ;
     this.setLayout (new BorderLayout()) ;

     Label str = new Label ("Sample Bean Label") ;
     add (str, "North") ;
     label = new TextField (bean.getLabel(), 40) ;
     add (label, "Center") ;

     doneButton = new Button ( "Apply Customization" ) ;
     doneButton.addActionListener(this);
     add(doneButton, "South");
  }

  public void addPropertyChangeListener(PropertyChangeListener l) {
     listeners.addPropertyChangeListener(l);
  }
  public void removePropertyChangeListener(PropertyChangeListener l) {
     listeners.removePropertyChangeListener(l);
  }

  public void actionPerformed (ActionEvent e) {
     if ( e.getSource() == doneButton )
        bean.setLabel (label.getText()) ;
  }
}
```

interact. This is important for maintaining simplicity of design and satisfying the JavaBeans design philosophy that a bean is an independent software component. Symphony beans were designed to satisfy this requirement. Thus, for example, a File bean can take input from any bean that can act as a data source (e.g., Stdin, Producer, Filter, etc.) without being aware of the actual type of the input bean.

In Symphony, connected beans interact through two well-defined, generic interfaces, the InputInterface and the OutputInterface. As explained in Section 3.4, connections between beans are always made in the direction of desired data flow. For any pair of connected beans, the bean from which the connection is initiated is a data source bean and the bean to which the connection is made is a data sink bean because the later reads data from the former and processes it. Any bean that can act as a data source must implement the `InputInterface` interface and any bean that acts as a data sink must implement the `OutputInterface` interface. It is important to understand the the reason for using these names for the interfaces. From the point of view of a data source bean, the data sink bean appears to be an output bean, and so the data source interacts with the data sink bean by making method calls on its `OutputInterface`. Conversely, from the point of view of a data sink bean, the source bean appears to be an input bean. Hence the sink bean interacts with the source bean by making method calls on the source bean's `InputInterface`. Thus, the source bean and the target bean can interact by making method calls on these well-known interfaces and do not have to be aware of any implementation-specific details of the peer bean. Figure 4.6 depicts the run-time communication mechanism used by two connected beans.

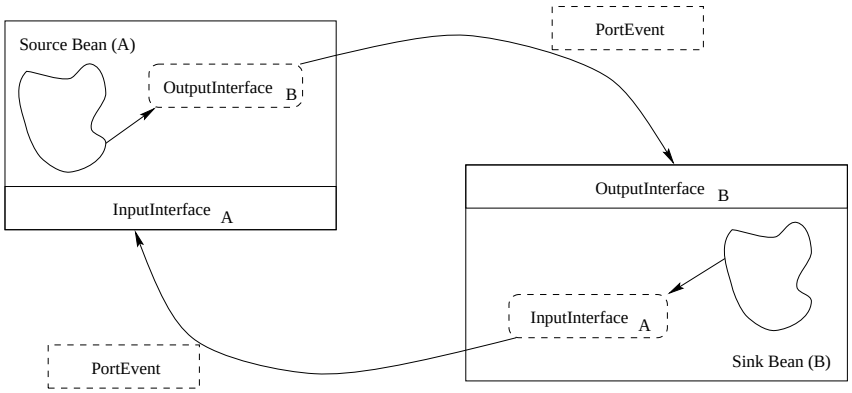

Figure 4.6: Run-time Communication Mechanism

As shown in the figure, on successful connection of a source bean **A** to a target bean **B**, bean **A** holds a reference to the `OutputInterface` of **B** and **B** holds a reference to the `InputInterface` of **A**. These interfaces contain methods that can be invoked for -

- checking the validity of the connection,

- querying the bean type and name,
- sending control events (represented by `PortEvent` objects),
- obtaining data, and
- removing the connection.

The need for each of these types of method can be explained as follows. When beans are connected, depending on the types of the source and target beans for the attempted connection, several checks need to be made to ensure that invalid connections are not allowed. For this validation, the beans being connected need to know the type of the peer bean. Control events need to be sent between beans in order to accomplish the various meta-program operations. Also, during the execute operation data sink beans need to obtain data from data source beans. Also, when a connection between two beans is removed in the BeanBox (by deleting either of the connected beans), before removing the connection, the beans need to interact in order to remove all references to the connection being removed.

4.3.1 The BaseInterface interface

Both `InputInterface` and `OutputInterface` contain some common functionality which has been elevated to the `BaseInterface` interface as shown in Figure 4.8. Figure 4.7 shows the elements of the `BaseInterface`. This common interface defines constants as well methods.

The type of a bean and its status at any time is represented by integer values picked from the set of constants defined in the `BaseInterface` for this purpose. As seen in the figure, type constants are defined for core beans as well as abstract beans as defined in Section 3.3. Constants need not be defined for the utility beans since they do not take part in the data transfer and are stand-alone beans. The `getBeanType()` and `getBeanTypeName()` methods can be used for obtaining the bean type as an integer and as a string, respectively. Thus, for a Program bean, the `getBeanType()` method would return the value `BaseInterface.PROGRAM` and the `getBeanTypeName()` method would return the string "Program". Status values are defined for the various states a bean passes through during it's lifecycle (Section 3.5). State transition diagrams that depict status changes for each bean will be given in later sections.

`BaseInterface` also defines constants for the various types of control events that can be sent in both directions during meta-program operations. Control events, represented by `PortEvent` objects, are sent by invoking the `BaseInterface.eventSend()` method. Specific details about the different event types will be discussed in Sections 4.3.5 and 4.4.

Every Symphony bean must have a label with which it is represented on the workspace. This label is set through the bean customizer. The `BaseInterface.getTitle()` method can be used to query the current label of the bean. This method is useful for generating meaningful error messages which refer to actual run-time labels of beans.

Figure 4.7: `BaseInterface`

Constants

```
// ==== Bean Type Constants ======
public static final int PROGRAM  ;
public static final int FILE     ;
public static final int SOCKET   ;
public static final int STDIN    ;
public static final int STDOUT   ;
public static final int STDERR   ;
public static final int PRODUCER ;
public static final int CONSUMER ;
public static final int FILTER   ;

// ==== Event Type constants =============
public static final int CONNECT_EVENT    ;
public static final int VERIFY_EVENT     ;
public static final int READY_EVENT      ;
public static final int START_EVENT      ;
public static final int DONE_EVENT       ;
public static final int STOP_EVENT       ;
public static final int DISCONNECT_EVENT ;

 // ==== Bean Status Codes ============
 public static final int CREATED      ;
 public static final int NOT_VERIFIED ;
 public static final int VERIFIED     ;
 public static final int READY        ;
 public static final int RUNNING      ;
 public static final int COMPLETED    ;
 public static final int ABORTED      ;
 public static final int SUSPENDED    ;

 // ==== Utility Constants ===============
 public static final String[] eventNames ;
 public static final String[] statusText ;
 public static final Color[]  beanColors ;
```

Methods

```
int    getBeanType () ;
String getBeanTypeName () ;
String getTitle () ;
void   eventSend (PortEvent x) throws EventException ;
```

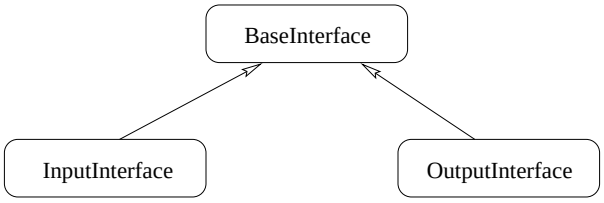

Figure 4.8: BaseInterface Inheritance Diagram

4.3.2 The InputInterface interface

The `InputInterface` defines three methods, in addition to the methods and constants defined in the `BaseInterface`. Figure 4.9 shows the methods from the `InputInterface`. The `getInputStream()` method is used by the target bean of a connection to obtain a reference to a `java.io.InputStream` object for reading data. As discussed in Section 3.5, all data-flow in Symphony is accomplished through byte streams. Regardless of the actual source or format of data, each data source bean must encapsulate the data in a `java.io.InputStream` object, a reference to which is provided to a data sink bean upon request.

The decision to use unformatted byte streams simplifies the data-flow interface between beans, but also entails extra work in the source and sink beans if the data to be transferred must has a structured format. For example, if the data is obtained as a result of a JDBC query [39] on a remote SQL database (obtained as a `java.sql.ResultSet` object), it must first be encapsulated by the source bean into a byte stream represented by an `InputStream` object. When the sink bean obtains the stream from the source bean, in order to interpret or process the data, the raw data from the `InputStream` must be converted back to a `ResultSet` object. However, for two main reasons, the approach used by Symphony generally entails less work than the opposing one of presupposing a particular format on a data stream and converting all data to this format. First, the Java serialization mechanism makes it extremely easy to convert any Java object to a raw data stream without regards to the data format and the complimentary reconstruction of the original object from the stream. Second, it is always possible to convert data in a particular format to a raw data stream, but it may not always be possible to convert it to another pre-defined format. Another important design decision is that the data-flow in Symphony always follows a data-pull model, where the sink beans asks the source for the data stream at an appropriate time. Data is never pushed by a source bean into a sink bean.

Roles of the remaining two methods, `checkConnectionTo` and `removeOutputConnection`, will be explained when the connection mechanism is explained in section 4.3.5.

Figure 4.9: `InputInterface`

Methods

```
public InputStream getInputStream (BaseInterface x) throws Exception ;
public void checkConnectionTo (OutputInterface x) throws Exception ;
public void removeOutputConnection (OutputInterface x) ;
```

Figure 4.10: `OutputInterface`

Methods

```
public OutputStream getOutputStream (BaseInterface x) throws Exception ;
public void checkConnectionFrom (InputInterface x) throws Exception ;
public void removeInputConnection (InputInterface x) ;
```

4.3.3 The OutputInterface interface

The `OutputInterface` is a mirror image of the `InputInterface`. Figure 4.10 shows the methods from the `OutputInterface`. Since Symphony beans follow a pure data-pull model, the `getOutputStream` method is currently not used. It is included in the `OutputInterface` purely for symmetry and for possible future use. Currently, all Symphony beans that implement the `OutputInterface` return a null value when this method is invoked. Again, the roles of the remaining two methods in the `OutputInterface` will be explained when the connection mechanism is explained in Section 4.3.5.

4.3.4 The PortEvent class

Symphony beans connected in a meta-program send control events to each other in order to accomplish a verify, execute, or abort operation requested by the user. Events are represented by `PortEvent` objects and are sent from a source bean by invoking the `eventSend` method on the `InputInterface` or `OutputInterface` of a target bean, depending on the event direction (Figure 4.6). A `PortEvent` object serves a dual purpose. First, it encapsulates enough information for the target bean to determine the purpose of the received event and the method for processing it. Second, it provides a means for the target bean to communicate the outcome of the event back to the source bean. Figure 4.11 shows the constructors, constants,

Figure 4.11: The `PortEvent` Class

Public Constructors

```
// ==== Input  Event Constructor ====
public PortEvent (OutputInterface oBean, Date date, int eventType) ;
// ==== Output Event Constructor ====
public PortEvent (InputInterface  iBean, Date date, int eventType) ;
```

Public Constants

```
// === Event Direction Codes ===
public static final int UP     ;
public static final int DOWN   ;

// ==== Event Return Status Codes ========
public static final int OKAY             ;
public static final int FATAL            ;
public static final int SUSPEND_YOURSELF ;
```

Public Methods

```
public Date getDate()      ;
public int  getType()      ;
public int  getDirection() ;

public void setInBean  (InputInterface in)   ;
public InputInterface   getInBean()          ;
public void setOutBean (OutputInterface out) ;
public OutputInterface  getOutBean()         ;

public void   setRetMsg (String str)    ;
public String getRetMsg ()              ;
public void   setRetStatus (int status) ;
public int    getRetStatus ()           ;
```

and methods in the `PortEvent` class. In order to understand the communication protocol for each Symphony operation, it is important to understand the functionality provided by the `PortEvent` class.

The various attributes of a `PortEvent` object are explained below.

- **Event Type** - This is the most important attribute of an event and defines the type of action that is requested by the event source. Valid event types are defined in the `BaseInterface` (4.7). The purpose and outcome of each type of event will be discussed in later sections. The event type is specified as a parameter in the `PortEvent` constructor.

- **Event Direction** - The direction of event is defined by two constants in the `PortEvent` class: UP and DOWN. When an event is sent by a data source bean to a data sink bean (i.e., in the direction of data flow) the direction is DOWN, and the direction is UP when an event is sent in the reverse direction. The direction of an event along with the event type helps in determining the type of action to be performed in the target bean. The direction value is automatically set by the constructor of the class. The first constructor shown in Figure 4.11 is used if the event is to be sent to an input bean (direction UP), and a reference to the source bean's `OutputInterface` is passed to the constructor. The second constructor is used for an output event (direction DOWN) and a reference to the source bean's `InputInterface` is passed.

- **Return Status** - Event processing in the target bean can produce one of several outcomes. The target bean communicates this information back to the source bean by using the return status attribute. This attribute can take any value from a set of constant integer values defined in the `PortEvent` class (Figure 4.11). The return status is initialized to OKAY when the `PortEvent` object is constructed. If there is a fatal error in processing the event (explained in later sections) the target bean sets the return value to FATAL. If the target bean wants the source bean to suspend all operations (explain in later sections) till further notification, the SUSPEND_YOURSELF return value is used.

- **Return Message** - The return value for an event can be qualified by a return message to be displayed to the user.

- **Date** - The date attribute has been included in the `PortEvent` class as a means of providing a time-stamp for the time when the event was generated. This attribute is not currently used by any bean, but is provided for future use. The date value is set through the constructor.

The `PortEvent` class also allows the target bean to return a reference to it's `InputInterface` or `OutputInterface` (depending on the event direction). This capability is important for the initial bean wiring when a connection is made between two beans, as will be seen in the next sub-section.

Figure 4.12: The `ConnectionListener` interface

```
public interface ConnectionListener extends java.util.EventListener {
   void createConnection (PortEvent x) throws EventException ;
}
```

4.3.5 Connection Mechanism

This section describes how the JavaBeans events mechanism explained in Section 4.2 is used to create the run-time communication structure used by Symphony beans (Figure 4.6).

While composing a Symphony meta-program, beans are connected according to the desired data-flow pattern. In order to co-ordinate the execution and data flow, Symphony beans send and receive several different types of events (Figure 4.7). If Symphony beans were to be connected strictly in the style described above, it would entail making several connections between each pair of adjacent beans. Additionally, during the execution of a meta-program components on both ends of a connection need to be able to send and receive events from each other, which means event connections would have to made both ways. Since this approach is tedious for the user and error-prone, Symphony hides most of the event machinery from the user and provides a clean, simple interface for connecting beans.

At the BeanBox level, each Symphony bean that takes part in the data flow (except for a Consumer bean) fires only one type of event (the connection event) and every bean (except for a Producer bean) exposes only one target method (eventSend). Two beans need to be connected only once, in the direction of data flow, by using this source event - target method combination, as described in Section 3.2. The Symphony connection mechanism takes care of creating the explicit run-time communication mechanism shown in Figure 4.6.

Thus, every Symphony bean that can act as a data source (i.e., as the source of a connection) implements `ConnectionListener` registration methods: `addConnectionListener` and `removeConnectionListener`. Due to this, when a source bean is selected in the workspace, the *connection* → *createConnection* menu-item appears under the *Edit* → *Events* sub-menu in the BeanBox. Figure 4.12 show the one-method `ConnectionListener` interface.

When a connection is initiated between two Symphony beans (as outlined by steps in Section 3.2), the BeanBox generates an adaptor class that implements the `ConnectionListener` interface and instantiates an object of this class. This instantiated object holds a reference to the target bean and can be used to propagate event notifications from source to target. Figure 4.13 depicts the interaction diagram for the series of actions that take place from this point onwards. The part in the figure below the dotted line represents the diagram for removing a connection between two beans, which will be discussed later. Each of the steps

shown in the diagram can be explained as follows:

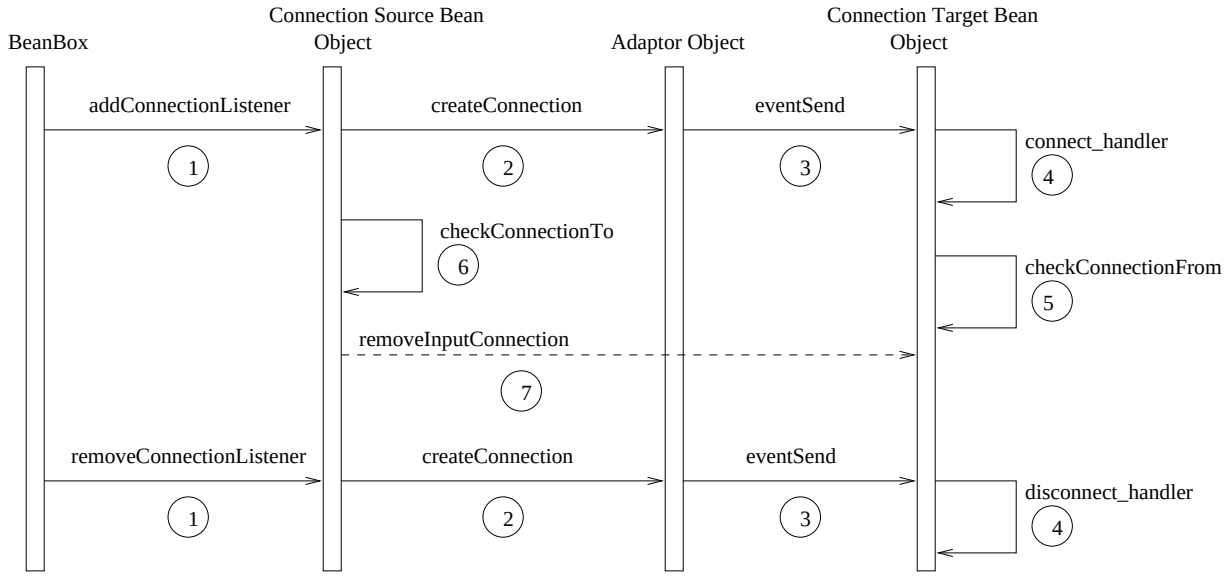

Figure 4.13: Interaction Diagrams for Connecting and Disconnecting Beans

1. The BeanBox invokes the `addConnectionListener` method of the source bean object, with a reference to the `ConnectionListener` interface of the adaptor.

2. The source bean creates a `PortEvent` object with a reference to it's `InputInterface` and of type CONNECT_EVENT (4.3.4) and invokes `ConnectionListener.create-Connection()` on the adaptor object for sending the connection event (represented by the `PortEvent` object) to the target bean.

3. The adaptor object forwards the notification by invoking the `eventSend()` method of the target bean.

4. The purpose of the `eventSend()` method is to forward a received event to the appropriate event handling method. Thus, the received connection event is forwarded to the `connect_handler()` method of the target bean.

5. The target bean checks connection validity by invoking `checkConnectionFrom()` (Section 4.3.3). If the connection is not valid, an exception is propagated back to the source bean, in which case, the `addConnectionListener` method in the source bean throws an exception which tells the BeanBox that the connection was not allowed.

   - If however, the connection is verified to be valid in the target bean, the target bean uses the `PortEvent` object received to extract a reference to the `InputInterface`

of the source bean and saves it in it's list of input beans (maintained as a vector of `InputInterface` references). It also uses the `PortEvent` object to return a reference to it's own `OutputInterface` (Section 4.3.4).

6. If an exception is not thrown in step 4, the `addConnectionListener` method of the source bean extracts the reference to the `OutputInterface` of the target bean from the `PortEvent` object. First, it checks its own list of output connections (maintained as a vector of `OutputInterface` references) to ensure that it's not already connected to the target bean. Multiple connections are not allowed between a pair of beans. If it is, an exception is thrown indicating to the BeanBox that the connection was not allowed.

   The source bean then invokes its own `checkConnectionTo()` method (Section 4.3.2) to check the validity of the connection from the source bean's point of view. If the connection is not valid, it displays an error message to the user and throws and exception which notifies the BeanBox that the connection was not allowed.

7. If an exception is thrown in step 6, the source bean invokes `removeInputConnection()` on the target bean's `OutputInterface`. This is done for removing the reference to the source bean from the target bean's list of input connections (which was added in step 5). If, however, the connection was validated, the source bean adds the extracted `OutputInterface` reference to its list of output beans, thus completing the connection structure shown in Figure 4.6.

The actions that take place when two beans are disconnected are simpler than the connection process. Two connected beans can be disconneted in the BeanBox, if the user chooses to remove either of them from the workspace. In this case, the following actions take place as shown in the interaction diagram.

1. The BeanBox invokes the `removeConnectionListener` method of the source bean object, with a reference to the `ConnectionListener` interface of the adaptor object for the connection.

2. The source bean creates a `PortEvent` object with a reference to it's `InputInterface` and of type DISCONNECT_EVENT (4.3.4). It invokes the `ConnectionListener.-createConnection()` method of the adaptor object for sending the disconnect event to the target bean.

3. The adaptor object forwards the notification by invoking the `eventSend()` method of the target bean.

4. The `eventSend()` method forwards the received event to the `disconnect_handler()` method of the target bean. The target bean uses the `PortEvent` object received to extract the reference to the `InputInterface` of the source bean and removes it from

Figure 4.14: Summary of Connection Mechanism

Method to be implemented for a source bean

```
public void addConnectionListener (ConnectionListener l) throws Exception ;
public void removeConnectionListener (ConnectionListener l) throws Exception ;
public void removeOutputConnection (OutputInterface x) ;
public void checkConnectionTo (OutputInterface x) throws Exception ;
```

Method to be implemented for a sink bean

```
public void eventSend (PortEvent) throws EventException ;
public void connect_handler (PortEvent x) throws EventException ;
public void disconnect_handler (PortEvent x) throws EventException ;
public void removeInputConnection (InputInterface x) ;
public void checkConnectionFrom (InputInterface x) throws Exception ;
```

its list of input beans. It also uses the `PortEvent` object to return a reference to it's own `OutputInterface`.

On successful delivery of the disconnect event to the target bean, the source bean extracts the reference to the `OutputInterface` of the target bean from the `PortEvent` object and removes that reference from it's list of output beans, thus removing all association between the source and target bean.

Figure 4.14 shows a list of methods that must be implemented by the source and sink beans in order to support the mechanisms described above for connecting and disconnecting two Symphony beans.

4.4 Communication Protocol

As described in Chapter 3, Symphony enables a user to perform three operations on a meta-program: verify, execute and stop. Although the command for initiating an operation is applied to a single program bean, all operations are applied to all beans in the meta-program. Depending on the type of operation to be performed, all beans must be instructed to execute their part of the operation in the correct order. Although, order of execution is not important for the verify and stop operations, all beans still need to be notified to perform verification or to abort execution. This is done by employing a distributed communication protocol that is based on the event passing framework described in the previous section.

Table 4.1: Protocol for Verify and Stop Operations

	Verify/Stop Event		
Bean	Send	Receive	Propagate
Program	√	√	√
File		√	√
Socket		√	
Stdin		√	√
Stdout		√	√
Stderr		√	√
Filter		√	√
Producer		√	
Consumer		√	

In addition to CONNECT_EVENT and DISCONNECT_EVENT, `BaseInterface` defines five more events (VERIFY_EVENT, READY_EVENT, START_EVENT, DONE_EVENT, STOP_EVENT). The verify event is used for carrying out the verify operation, the stop event for the stop operation, and the other three events are used during the execute operation. Recall that an event is represented by a `PortEvent` object that has an event type (Section 4.3.4). The event is sent from a source bean to a target bean by invoking the `eventSend` method on the `InputInterface` or `OutputInterface` of the target bean depending on the direction of the event. The following sub-sections outline the communication protocol used for all three meta-program operations.

There are two modes of event flow: event generation and event propagation. Generally, events are generated (initiated) only by Program beans. When an event is received by a particular bean it may be propagated to other connected beans depending on the type of the event and the type of bean receiving it. An event is propagated only to beans other than the one from which it was received.

4.4.1 Verify Operation

When the user instructs a Program bean to verify the meta-program, the Program bean generates a verify event (VERIFY_EVENT in Figure 4.7) and sends it to all the beans connected to it. Table 4.1 depicts the communication protocol used during the verify operation. For each bean type, the table shows whether the bean can send, receive, and/or propagate the verify event. As can be seen from the table, only the Program bean generates a verify event. All other beans in the meta-program receive this event and propagate it, if appropriate. The Producer, Consumer, and Socket beans receive but do not propagate the verify event since a Producer can only have one output connection, a Consumer bean can have only one input connection, and a Socket bean can have either one input connection or one

Figure 4.15: Algorithm Used During Verify Operation

```
if (bean status is RUNNING)
   return with an error message

verify local (this bean's) connection information and
customized properties for the resource represented

if (verified)
   if (this is a program bean)
      propagate verify event to all other connections
      if (verification of connected beans did not return a error)
         set bean status to  VERIFIED
      return
   else
      set bean status to  VERIFIED
      propagate verify event to other connections, if any
      return
else
   return with an error message
```

output connection.

The basic algorithm followed by a bean when a verify event is generated or received is shown in Figure 4.15. As can be seen from the algorithm, if verification fails for a particular bean, it does not propagate the verify event to other connected beans and the verification process stops. A bean that fails its verification returns a FATAL return value (Section 4.3.4) and an appropriate error message. Also, while propagating or sending the verify event to all connections, if a FATAL return value is received from one connected beans, the event is not sent to the remaining connections.

During a verify, no data transfer or program execution is carried out. The operation only checks the existence and accessibility of all local and remote resources, network state, and the propriety of all the connections and property customizations. The specific checks that are made for each bean depend on the bean type and are described in later sections.

4.4.2 Stop Operation

The stop operation aborts the execution of all executing programs, all current data transfer channels, and closing all open files and network connections. When the user instructs a Program bean to stop the meta-program, the bean generates a stop event (STOP_EVENT in Figure 4.7) and sends it to all the beans connected to it. The communication protocol

Figure 4.16: Algorithm Used During Stop Operation

```
depending on what type of bean this is, stop any executing
  program, close all files and network connections, stop all
  ongoing data transfer, destroy any active threads, and close
  any open dialog boxes

propagate the stop event to all other connections
set bean status to ABORTED
return
```

used for the stop operation is exactly similar to that used for the verify operation, and is depicted by Table 4.1. Again, the Program bean which receives the user notification to stop the meta-program, generates a stop event, which is received by all other beans in the meta-program. Also, for the reason outlined before, the Producer, Consumer, and Socket beans only receive the event, but do not propagate it.

The basic algorithm followed by a bean when a stop event is generated or received is shown in Figure 4.16. The specific details about the actions performed in each individual bean depend on the bean type and are described in later sections. The major difference in the stop algorithm from that of the verification algorithm is that a stop operation never fails. Regardless of the state of any bean, any current activity in the bean is aborted and the stop notification is propagated if necessary.

4.4.3 Execute Operation

The execute operation for a meta-program is initiated when the user chooses the 'Start' option from a Program bean's popup menu. Although the command is given to one Program bean, it is meant to execute all programs in the meta-program in the sequence that satisfies the data-flow requirements for each program. The execute operation for an individual Program bean ensures that all Program beans that need to execute earlier in the sequence are executed successfully before this program can start executing, and that when this program is done executing, all programs later in the sequence are instructed to start executing. This distributed protocol, when applied to each Program bean, causes the execution of the meta-program as a whole. The steps that must happen during the execute operation of an individual Program bean can be conceptually outlined as follows:

1. The meta-program must be completely verified, before any program can be executed

Table 4.2: Protocol for the Execute Operation

Bean	Ready Event			Start Event			Done Event		
	S	R	P	S	R	P	S	R	P
Program	√	√		√			√	√	
File		√	√					√	√
Socket		√						√	√
Stdin		√	√		√				
Stdout					√		√	√	
Stderr					√		√	√	
Filter		√	√					√	√
Producer		√							
Consumer								√	

S = Can Send (Initiate), R = Can Receive, P = Propagates the received event

2. All input resources for a program represented by a Program bean must be present and ready for reading before the program can start executing.

3. The execution of a program can only be started if all upstream programs (i.e., the program which precede it in the data-flow) have executed successfully.

4. If the program takes input on the standard input, input must be redirected to it from the proper source, and if it writes output to the standard output or error, the output must be redirected to the appropriate destination.

5. All output resources must be notified and checked for presence after the program execution is done.

6. Once the execution of a particular program is done, if there are any downstream programs (i.e., programs which depend on this program for data) they must be instructed to start executing.

Steps 2 and 3 above, are accomplished by using the ready event (READY_EVENT in Figure 4.7), step 4, is accomplished by using the start event (START_EVENT in Figure 4.7), and steps 5 and 6 by using the done event (DONE_EVENT in Figure 4.7). Figure 4.17 shows the algorithm used by the Program bean when either the user instructs it to start program execution, or when it receives a notification from an upstream or downstream Program bean to start execution.

Table 4.2 shows the communication protocol for the execute operation. The actions taken by each type of bean when a ready, start, or done event is received will be described in later sections.

Figure 4.17: Algorithm used by Program Bean for Meta-Program Execution

```
obtain execution rights
if (execution rights could not be obtained)
   program is either already executing or suspended
   so return with an error message

if (status is not VERIFIED)
   initiate a verify operation to verify the entire meta-program
if (still not VERIFIED)
   return with error message

send ready event to all input beans
   if (there is an upstream program)
      this program is suspended till that program finishes execution
if (ready event failed)
   initiate a stop operation to abort meta-program execution
   return with an error message

if (program is Web-accessible)
   load the program URL in a browser window

else if (program is User-accessible)
   try to start execution of the actual program represented
   if (start fails)
      initiate a stop operation to abort meta-program execution
      return with an error message

   display terminal for showing program output, if necessary
   if (standard I/O beans are connected)
      send start event
      if (start event fails)
         initiate a stop operation to abort meta-program execution
         return with an error message

   wait for the program execution to complete
end

send done event to all the output beans
if (there are any downstream programs from this program)
   this done event will have the effect of starting their execution too

if (done event failed)
    initiate a stop operation to abort meta-program execution
    return with an error message

release execution rights
```

4.5 The Symphony Server

There are three types of legacy resources which can be represented by the core Symphony beans: input data resources, output data resources, executable resources. The respective beans need to read, write, or execute the resource they represent when the meta-program is executed. Input and output legacy resources are generally represented by files which are read from or written to by using standard HTTP or FTP mechanisms. Managing executable resources is more complicated. These are programs which must be run on the machines on which they reside at the appropriate time, with the appropriate command line parameters and environment variables. The meta-program may also dictate manipulation of the remote programs standard I/O streams.

The Program bean represents a local or remote executable program which may be Web-accessible (such as applet or CGI script) or accessible through a user account on the host machine on which the program resides. Web-accessible programs can be executed by simply loading the URL for the program in a web browser. On the other hand, when a private user-accessible program is to be executed, four things must occur:

- The Program bean must contact a server running on the remote host and provide it with security information for authentication.
- If authenticated, the Program bean must provide the server with the program details such as the absolute path to the program, program name, optional command-line parameters and environment variables, and ask the server to execute the program.
- The bean must obtain a handle to the remote program's standard input stream for writing to it, if necessary.
- The bean must obtain handles to the remote program's standard output and error streams for reading from them, if necessary.
- The bean must wait for the remote program to finish execution before closing the streams and releasing resources.

The Program bean communicates with the Symphony server for carrying out the steps outlined above. Instead of using a custom Socket based protocol for communicating with the server, the server has been designed as a remote object whose services can he invoked by simple method calls. This frees the client from engaging in an application-level protocol for encoding and decoding messages. The Java Remote Method Invocation (RMI) mechanism is used for this purpose.

4.5.1 Executing Native Applications from Java

The `java.lang.Runtime` class provides a way of executing a non-Java native program from a Java program and accessing to the program's standard I/O streams. It also allows the specification of command-line parameters and environment variables for the program. The `Runtime.exec()` method has the following signature:

```
public Process exec (String cmd, String[] envp)
       throws IOException;
```

The `exec()` method returns an instance of the `java.lang.Process` class. This instance can be used to control the native process and access to the process's standard I/O streams. The Process object can also be used to check the exit value from the program and to stop it while it's running, if necessary.

4.5.2 Implementing the Symphony Server

This sub-section explains the fundamentals of the Java RMI mechanism and describes the implementation of the Symphony server using RMI [31].

Traditionally, socket-based communication has been used in designing client/server systems. Sockets require the client and server to engage in an application-level protocol for encoding and decoding messages exchanged between them. The design of such protocols is cumbersome and error-prone because the socket provides only an unstructured byte stream with no support for coping with incompatible data representation across different hardware architectures.

An alternative is Remote Procedure Calls (RPC) which uses an external representation to encode the arguments and return values of a procedure invocation that is executed remotely. Because RPC is a procedure-based mechanism, it is not well suited for distributed object-oriented systems, where communication between program-level objects residing in different address spaces is needed. Remote Method Invocation is an extension of the RPC mechanism for object-oriented systems.

A Java remote object is an object whose methods can be invoked from another Java virtual machine potentially running on a different host. A method invocation on a remote object has the same syntax as a method invocation on a local object. The arguments and return values of a remote invocation can be any Java type that is serializable. A reference to a remote object can also be passed as an argument or returned as a result in any remote method invocation. Local objects are passed by copy and remote objects by reference. For local objects, only non-static and non-transient fields are copied.

The ProgramServer Interface

A remote object's implementation is separated from its interface description. The remote object is described by one or more remote interfaces that inherit from the `java.rmi.Remote` interface. Client always access the remote object through remote interfaces, never directly through the implementation. Also, a remote object passed as an argument or return value must be declared as the remote interface, not as the implementation class. All methods in a remote interface must throw the `java.rmi.RemoteException` exception. Clients using remote objects must be prepared to handle this exception. A remote invocation may fail with little or no information about the cause of the failure.

The Symphony server is implemented as a remote object. It provides a single method which can be invoked from remote client beans as shown in Figure 4.18. The client defines the parameters of an executable program on the server machine in the form of a `ProgramDefinition` object. A partial implementation of the `ProgramDefinition` class is shown in Figure 4.18. The client creates a `ProgramDefinition` object with the appropriate parameters and invokes the `getProcess()` method of the `ProgramServer` interface. The second parameter of the `getProcess()` method represents a unique identifier for the Java virtual machine within which the client is executing. This parameter is not used currently, but provided for future use.

As described in the previous section, a native application is executed by invoking the `Runtime.exec()` method which returns a `Process` object. This object can be used to manipulate the underlying system process. When the program is executed on the server machine, such a Process object will be created in the Java virtual machine running on the server. Since an object created in the Java VM on the server can not be accessed directly by the client, it must be exported as a remote object on which the client can make remote method invocations. The `RemoteProcess` interface is a remote interface that provides all the methods that can be invoked on a `Process` object (Figure 4.18). The class that implements this remote interface (`RemoteProcessImpl`) encapsulates a `Process` object and forwards remote method invocations on the interface to the actual `Process`. When a client invokes the `ProgramServer.getProcess()` method, a `RemoteProcessImpl` object is instantiated and a reference to its `RemoteProcess` interface is returned to the client. Creating a `RemoteProcessImpl` object does not automatically start the execution of the program. The client must invoke the `RemoteProcess.start()` method to execute the program.

Program Server Implementation

Figure 4.19 shows the implementation of the `ProgramServer` interface that, like most other remote objects, extends the `java.rmi.server.UnicastRemoteObject` class. `getProcess()` first extracts information from the `ProgramDefinition` object to authenticate the client bean's request. If authentication is successful, it extracts more information and creates the

Figure 4.18: `ProgramServer`, `ProgramDefinition` and `RemoteProcess`

```
public interface ProgramServer extends java.rmi.Remote {
   public RemoteProcess getProcess (ProgramDefinition pdef, VMID vmId)
      throws java.rmi.RemoteException, java.io.IOException ;
}
```

```
public class ProgramDefinition
       implements java.io.Serializable {

   String  host     ;  // Name of host machine on which program resides
   String  path     ;  // Absolute path to the program file on the host
   String  file     ;  // File name
   String  login    ;  // User login
   String  password ;  // User password for authentication
   String  paramStr ;  // Command-line parameters string
   String[] envArr  ;  // Array of environment variables
   boolean xFlag    ;  // True if program is a X-window program
   String  xDisplay ;  // X-server display string for a X-windw program

   public ProgramDefinition ( String newHost, String newName,
          String newPath, String newLogin, String newPass,
          String newStr,  String[] envAry, boolean x, String newTerm ) {
      host     = newHost ;      file  = newName   ;
      path     = newPath ;      login = newLogin  ;
      password = newPass ;      paramStr = newStr ;
      envArr   = envAry  ;      xFlag    = x      ;
      xDisplay = newTerm ;
   }

   // === Accessor methods for properties defined above ===
   ...
}
```

```
public interface RemoteProcess extends java.rmi.Remote {

   public void start()                throws RemoteException ;
   public void stop()                 throws RemoteException ;
   public void setCommand(String cmd) throws RemoteException ;
   public String getCommand()         throws RemoteException

   public RemoteOutputStream getOutputStream() throws RemoteException ;
   public RemoteInputStream  getInputStream()  throws RemoteException ;
   public RemoteInputStream  getErrorStream()  throws RemoteException ;

   public int waitFor()   throws RemoteException, InterruptedException ;
   public int exitValue() throws RemoteException ;
   public void destroy()  throws RemoteException ;
}
```

command-line string for executing the native application. A `RemoteProcessImpl` object is constructed with this string and the environment variable string as parameters. As a reminder, instantiating the `RemoteProcessImpl` object does not start the actual program defined by the command-line string. Finally, a reference to the created object, cast as a `RemoteProcess`, is returned to the client. The client can use this reference to make remote method invocations for starting the host program when it deems appropriate and for obtaining remote references to it's standard I/O streams (Figure 4.18). The implementation of the remote stream objects will be described in the next section.

Obtaining a ProgramServer Reference

In order to invoke the `getProcess` method on a `ProgramServer` implementation running on a particular host, the client must have a reference to the remote object. The Java RMI mechanism provides a simple bootstrap technique for obtaining this reference. Server objects running on a particular host can be registered with a URL-based registry, also running on the same host machine. The `java.rmi.Naming` class provides methods to bind an instance of a remote object to a URL and register the binding. It also provides methods for remote clients to contact a registry on a particular host and ask for a reference to a remote object by providing a URL.

As shown in Figure 4.19, the implementation of a server object that acts as the initial point of contact for clients generally contains a `main()` method. This method must install a new security manager, create one or more instances of the remote object for providing service, and register the instance(s) with the registry. The security manager guarantees that classes loaded from the network do not perform "sensitive" operations on the server. Also, every instance of the remote object must be given a different string name. The server implementation class is executed once as a Java application in order to initialize the server objects and register them with the registry. The RMI registry must be started as a daemon process before any server objects are initialized by running the `rmiregistry` application from the command-line. An optional command line parameter, which specifies the port number on which the registry listens for requests, can be given when the application is started.

Figure 4.20 shows how a reference to the ProgramServer object can be obtained by a remote client bean and how this reference is used to invoke a remote method. The `lookup()` method of the `Naming` class takes a URL with the "rmi" protocol identifier as a parameter. This URL must include the host name for the RMI registry to be contacted, the port number on which the registry is configured to listen for requests, and the programmatic name of the server object as registered with the registry (`ProgramServer` in Figures 4.19 and 4.20). The `Naming.lookup()` method returns a reference to a `java.lang.Object` which can be cast to the required type. Once the reference is obtained, method calls can be made on the remote object like any other local method call. It must, however, be noted that a remote method can always throw the `java.rmi.RemoteException` which must be caught or declared in the

Figure 4.19: The ProgramServer Implementation

```
public class ProgramServerImpl extends UnicastRemoteObject
       implements ProgramServer {

   private String name;
   public ProgramServerImpl (String s) throws RemoteException {
      super () ;
      name = s ;
   }

   public synchronized RemoteProcess getProcess (ProgramDefinition pdef,
                       VMID vmId) throws java.rmi.RemoteException {

      // == Authenticate the request against the local FTP daemon ==
      ....

      // == Build the command string ====
      String command = pdef.getPath() + "/" + pdef.getFile() + " " ;
      if ( pdef.getXFlag() && ( pdef.getXDisplay().length() != 0 ))
           command += "-d " + pdef.getXDisplay() + " " ;
      command += pdef.getParamStr() ;

      // == Create the RemoteProcess object ==
      RemoteProcess rp = new RemoteProcessImpl (command, pdef.getEnvArr());
      if (rp == null)
         throw new RemoteException ("Remote Process Creation Failed");
      return rp ;
   }

   public static void main (String args[])
      throws Exception {

      // Create and install a security manager
      System.setSecurityManager(new RMISecurityManager());

      try {
        hostname = InetAddress.getLocalHost().getHostName() ;
      } catch (java.net.UnknownHostException ex) {
        throw new Exception ("Local host name can't be found!") ;
      }

      try {
        ProgramServerImpl obj = new ProgramServerImpl ("ProgramServer");
        Naming.rebind ("//" + hostname + ":9999" + "/ProgramServer", obj);
      } catch (Exception e) {
        e.printStackTrace();
      }
   }
}
```

Figure 4.20: Obtaining a ProgramServer Reference

```
ProgramServer ps = (ProgramServer)
       Naming.lookup( "rmi://" + host + ":"+ 9999 + "/ProgramServer" );

ProgramDefinition pdef = new ProgramDefinition ( ... );
VMID vmId = new VMID ();
RemoteProcess rp = (RemoteProcess) ps.getProcess (pdef, vmId);
```

`throws` clause of the calling method.

Stub and Skeleton Classes

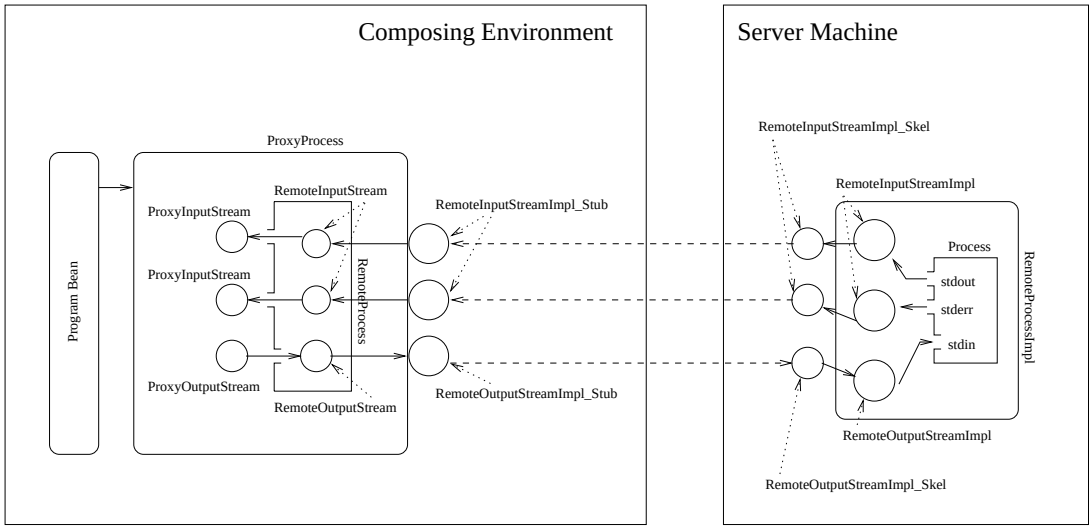

Figure 4.21: Exporting Remote Streams

Once a remote object has been implemented, stub and skeleton classes need to be generated for supporting remote method invocations on the object. On the client side, a reference to a remote object is actually a reference to a local stub object. The stub is an implementation of the remote interfaces of the object and forwards the invocation requests to the server. Similar to a stub on the client side is a skeleton object on the server side which is responsible for receiving client calls and forwarding them to the actual object implementation. The return value follows the reverse path. The stub and the skeleton are also responsible for serialization and de-serialization of method parameters and the return value for transmission.

The Java classes that implement the remote object server and the client do not have to be aware of the stub and skeleton classes. These classes are generated automatically from the remote object implementation by using the RMI compiler. Thus, for the ProgramServerImpl class, the RMI compiler generates two classes: ProgramServerImpl_Stub.class and ProgramServerImpl_Skel.class.

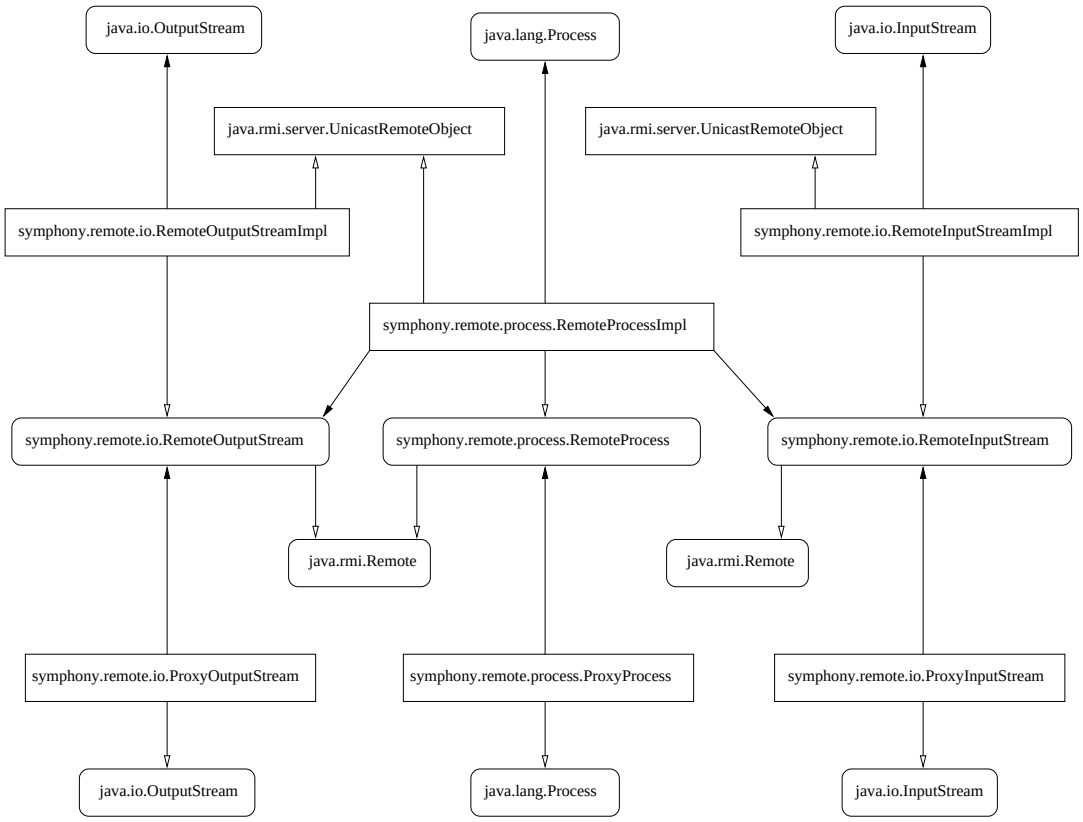

Figure 4.22: Inheritance and Association Diagram for RMI Beans

4.5.3 Accessing Remote Data Streams

As explained at the beginning of this section, a Program bean needs access to the standard I/O streams of the actual program executing on the local or remote host machine. Since these streams on the host machine, represented by the `java.io.InputStream` and `java.io.OutputStream` classes, can not be accessed directly by the client bean, some special mechanism is need for the bean to interact with them. A possible solution is to have a socket-based server on the host machine for each stream with which the bean can interact for sending or receiving data from the stream. For reasons mentioned previously, the

socket-based approach is awkward and difficult to program and understand. Instead, Symphony exports the streams on the server to the client as remote objects on which the bean can make regular method calls as on local objects. Symphony takes the convenience offered by RMI one step further and encapsulate the remote stream objects on the client side in local objects which are of type `java.io.InputStream` and `java.io.OutputStream`. Thus, the bean naturally interacts with standard Java object types and does not have to know anything about RMI or remote streams. It simply obtains references to these streams as local objects. Figure 4.21 shows a conceptual model of this approach. On the server machine, the `java.io.OutputStream` object obtained for writing to the standard input of the process is encapsulated in a `RemoteOutputStreamImpl` object. The `RemoteProcess` reference exported back to the client by the `ProgramServer` can be used to access this remote stream implementation through the `RemoteOutputStream` remote interface (Figure 4.18). In the composing environment on the client side, this `RemoteOutputStream` reference is encapsulated in a `ProxyOutputStream` which directly inherits from `java.io.OutputStream` and overrides its methods. A similar mechanism is used for the programs standard output and error represented by input streams which can be read from.

4.5.4 Local vs. Remote Transparency

One of the design goals was to minimize the overhead for accessing local resources. If a program is executed on the same machine as the composing environment, it is not efficient to employ the RMI mechanism used for remote programs. Instead, a local `Process` object is created and its streams can be accessed directly, thus avoiding the several levels of indirection required for remote streams.

However, the Program bean does not have to aware of whether the process is executing locally or remotely. This is achieved by using the `ProxyProcess` class. When the bean wishes to execute the program represented, it collects all the program details in a `ProgramDefinition` object and instantiates a `ProxyProcess` object. The constructor of the `ProxyProcess` class checks to see whether the request is for a local program or a remote program. If it is for a remote program it contacts the RMI registry on the remote system and obtains a reference to the `ProgramServer` running on that host. It then uses the `ProgramServer.getProcess()` method to obtain a reference to a `RemoteProcess` object (Figure 4.20). Again, it must be remembered that obtaining a `RemoteProcess` reference does not actually start the remote program on the host machine. The `RemoteProcess` reference is used to start the remote program and obtain references to its standard I/O streams (as explained in the previous section) only when the Program bean invokes the `start()` method on the `ProxyProcess` object. If, however, the request from the Program bean is for executing a local program, the only thing that is done in the `ProxyProcess` constructor is to validate the local program's absolute path and file name and whether it's accessible by the current user. The program, again, is actually started when the bean invokes the `ProxyProcess.start()` method.

Another advantage of using the `ProxyProcess` class is that it can enable transparent communication with remote servers which are implemented using a raw socket-based transport rather than RMI. Although such a server has not been currently implemented, it may be needed for executing programs on host platforms which do not have support for Java or for the Java RMI mechanism. The Program bean does not have to be aware of the underlying differences in the communication mechanisms used.

Another view of the communication mechanism is given in Figure 4.22 that shows the inheritance and association relationships between the various entities involved in the RMI implementation of remote processes and remote streams. A hollow arrow indicates inheritance or implementation of an interface, and a solid arrow indicates that the source object holds a reference to or encapsulates the target object.

4.6 Implementation of Core Symphony Beans

The set of Symphony beans currently includes six beans that form the core of the Symphony framework: Program, File, Socket, Stardard Input, Standard Output, and Standard Error beans. These beans contain significant common functionality such as that needed for managing connections and events. Some of the beans also share similar properties. The common features of these core beans have been elevated to an abstract class named `BasicBean`. This section describes the `BasicBean` class and the implementation of the core beans that inherit from this class.

4.6.1 The BasicBean Abstract Class

Figure 4.23 shows the inheritance structure for the core Symphony beans all of which inherit from the `BasicBean` class. Arrows in the diagram represent either inheritance or implementation of an interface depending on the type of the object pointed to. As can be seen from the figure, the `BasicBean` class extends the `java.awt.Panel` class and provides event handling methods for mouse events by implementing the `MouseListener` and `MouseMotionListener` interfaces. It also implements the `Serializable` interface which serves to mark a `BasicBean` object as serializable. Further, it implements the interfaces, `InputInterface` and `OutputInterface`, required for beans that can act as data sources as well as sinks, as explained in Section 4.3, and the `PropertyChangeListener` interface for beans that wish to listen for property change events.

Figure 4.24 shows the methods implemented by the `BasicBean` class. As can be seen from the figure, `BasicBean` defines five abstract methods. These abstract methods, when implemented by a derived class, define how the derived class objects respond to the basic types of events. Also, `BasicBean` does not implement all methods from `InputInterface` and `OutputInterface`. Thus, all beans that inherit from `BasicBean` must implement the re-

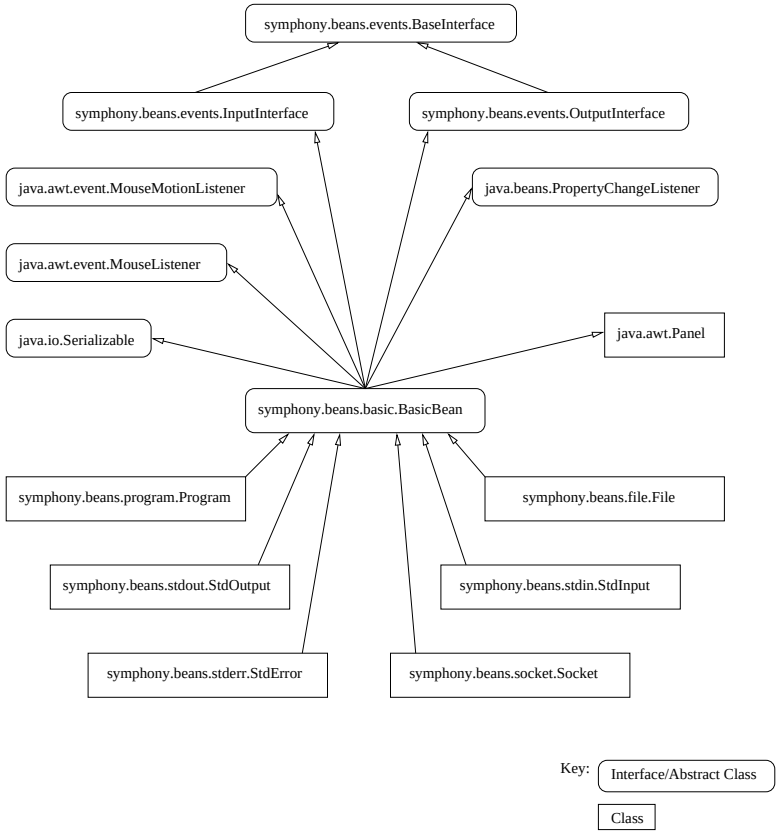

Figure 4.23: Inheritance Structure for Core Beans

maining methods from these interfaces and also the abstract methods defined by `BasicBean`. Figure 4.25 shows all the methods that need to be implemented in order to provide an implementation for the `BasicBean` class. The various interfaces and methods implemented by the `BasicBean` class can be sub-divided into: user interface methods, property methods, connection and event handling methods and dialog methods, as explained in the following sub-sections.

User-Interface Methods

The `BasicBean` class inherits from the `java.awt.Panel` class. All core beans have a user interface which displays the bean title in a colored box with a border. The Program and File beans have additional popup menus that appear when the user clicks on either of these bean types. Since mouse events need to be captured and processed for creating the popup menu, `BasicBean` implements the `MouseListener` and `MouseMotionListener` interfaces from the `java.awt.event` package. The class also provides other methods for supporting the user

Figure 4.24: The `BasicBean` Class

```
// ===== User Interface Methods =====

// Methods from the java.awt.event.MouseListener and
// java.awt.event.MouseMotionListener interfaces,
// methods to get/set background color, get/set foreground
// color, paint, minimumSize, etc.

// ===== Property Methods =====

public void addPropertyChangeListener(PropertyChangeListener l) ;
public void removePropertyChangeListener(PropertyChangeListener l) ;
public void propertyChange(PropertyChangeEvent evt) ;
// Accessor methods for properties (getX()/setX(Xtype)) -
// title (String), status (int), host (String), port (int)
// fileName (String), filePath (String), login (String),
// password (String), description (String)

// ===== Connection and Event Handling Methods ======

public synchronized void addConnectionListener (ConnectionListener l)
   throws Exception ;
public synchronized void removeConnectionListener (ConnectionListener l)
   throws Exception ;
public void removeOutputConnection (OutputInterface x) ;

public void eventSend          (PortEvent x) throws EventException ;
public void connect_handler    (PortEvent x) throws EventException ;
public void disconnect_handler (PortEvent x) throws EventException ;
public void removeInputConnection  (InputInterface x) ;

public boolean outputAction   (int callType) ;
public boolean inputAction    (int callType) ;
public boolean propagateEvent (PortEvent x, int pdir) ;

public abstract void verify_handler (PortEvent x) ;
public abstract void stop_handler   (PortEvent x) ;
public abstract void ready_handler  (PortEvent x) ;
public abstract void start_handler  (PortEvent x) throws EventException ;
public abstract void done_handler   (PortEvent x) ;

public synchronized void incrementSuspensionCount() ;
public synchronized void decrementSuspensionCount() ;
public synchronized boolean isSuspended () ;

// ===== Dialog Methods ====

private Frame getBeanBoxFrame() ;
public void msgBox (String title, String msg) ;
public void errBox (String msg) ;
```

interface such as methods that get and set the background and foreground color, return the size of the bean on the workspace, and paint the bean interface. The `getBackground()` method which is responsible for returning the background color of the bean is particularly interesting because it returns the color value depending on the current status of the bean (Figure 4.7). It uses the color constants defined by the `beanColor` array in `BaseInterface`, for obtaining the color value corresponding to a status value.

Property Methods

The `BasicBean` defines a set of properties (Section 4.2.2), some of which are used by all core beans and others are common to more than one bean. All properties supported by Symphony beans are bound properties such that a property change can be bound to a method invocation in a bean or in the environment. The `BasicBean` class implements registration methods for `PropertyChangeListener` objects. The bound property mechanism is used for a specific purpose in Symphony. Properties customized for a bean are verified when the user performs the 'Verfiy' operation on the meta-program. On verification, the bean status is set to VERIFIED (Figure 4.7). If, after verification, the user changes the value of any "sensitive" property, a property which affects the overall meta-program configuration, the bean status must be changed to NOT_VERIFIED and an execute operation must not be allowed unless the bean properties are verified again. For example, if the host name property is changed for a Program bean that represents a remote user-accessible program, the Program bean needs to be verified again in order to ensure that the Symphony server is running on the newly defined location and that the program exists and is accessible. Since all bean properties are bound properties, property changes can be caught using the bound property change mechanism. For this, every Symphony bean implements the `PropertyChangeListener` interface and registers with itself as a property change listener at instantiation. Thus, every time a property value is changed, the `PropertyChangeListener.propertyChange()` method will be invoked. This method checks to see if a "sensitive" property has been changed and if so, sets the bean status to NOT_VERIFIED.

Every property is defined by property read and write methods (Section 4.2.2). As described in Section 4.2.4, properties of a Symphony bean are set by using the bean customizer. For any customizer, relevant property read methods are invoked when the customizer is initialized and the relevant property write methods are invoked when the user pushes the "Apply Customization" button in the customizer.

The properties implemented by `BasicBean` are:

- **Bean Title:** This is the title with which the bean is represented on the workspace. It must be noted that the `getTitle()` method, in addition to being an accessor method for the title property, is also required as part of the `BaseInterface`.
- **Bean Description:** This property allows the user to include a short description with

the bean.

- **Bean Status:** This is an important property which represents the state of a bean at any point in time. A bean passes through several states during it's lifecycle (Figure 4.7). The `setStatus()` method is also responsible for providing the visual feedback to the user during any meta-program operation. After changing the status value, it invokes the `getBackround()` method which in turn sets the background color of the bean to represent the current status.

- **Host Name:** This property is used by the Program, File, and Socket beans. For Program and File beans it defines the host machine on which the corresponding program or file resides. For a Socket bean, it defines the host name for the socket connection.

- **Port Number:** This property is also used by the Program, File, and Socket beans. For a Program bean, it is used for a user-accessible program type and defines the RMI registry port number to contact for a remote program. For a File bean, it specifies the FTP daemon port number for accessing a private user-accessible file. For a Socket bean, it's the socket port number.

- **User Name and Password:** The user name and password properties are used by the Program bean and File bean for specifying the authentication information for a remote program or file.

- **File Path and Name:** These two properties are used by the Program and File bean to specify the absolute path to and the name of the represented program or file.

Since one design goal was to keep everything serializable, the `BasicBean` class implements the `Serializable` interface (see Section 4.2.5). All the above properties being non-static and non-transient, when the BeanBox serializes a core bean, all property values that are part of the `BasicBean` class are also saved.

Connection and Event Handling Methods

All core beans follow the connection and communication model discussed in Section 4.3 and can act both as a data source as well as a data sink as can be seen from Table 3.3. Thus, each core bean must implement the set of methods shown in Figure 4.14 for supporting the connection mechanism. The `BasicBean` class implements all but two methods from this list of methods. The two methods not implemented are `checkConnectionTo()` and `checkConnectionFrom()` since these are bean specific methods and have to be implemented by each core bean separately. The implemented methods follow the exact same logic as explained by the interaction diagram in Figure 4.13 and the associated description in Section 4.3.5.

Any bean that inherits from `BasicBean` can receive seven different types of events (Figure 4.7), including the connect and disconnect events. The `eventSend` method redirects the received event to the appropriate event handling method. If an event of unknown type is received, the `eventSend` method ignores it. Event handling methods for events other than the connect and disconnect events are defined as abstract methods by the `BasicBean` class, since the task to be performed for each of these events is different for different beans. All core beans must implement these abstract methods and provide appropriate behavior for processing each event type.

`BasicBean` implements three utility methods: for sending an event to all input beans (`inputAction`), for sending an event to all output beans (`outputAction`), and for propagating an event received from some other bean (`propagateEvent`) to all input and/or output beans. Event propagation means that the event is sent to all requested beans (i.e., input and/or output) except to the bean from which the event was received. These methods use the `BaseInterface.eventSend()` method for sending events to each individual bean. As described in Section 4.3.4, every event has a return status and a return message which indicate the outcome of sending the event. Each possible return status is handled by the above three methods as follows:

- **OKAY**: As described in Section 4.3.4, this return status indicates that the event was successful in achieving the desired effect in the target bean and it should be send to the remaining input and/or output connections.
- **FATAL**: This means that an irreparable problem was encountered in the target bean while processing the event. In this case, the `inputAction` and `outputAction` methods simply display the return message to the user and notify the calling method that the event failed by returning a false value. The `propagateEvent` method, on the other hand, copies over the return value and message to the original event it had received for propagating. This is important because it channels the original error message all the way back to the event source (which in effect is either the `inputAction()` method or the `outputAction()` method) where it is displayed to the user. In each of these methods, if an event send or propagation operation fails, the event is not sent to the remaining connections in the list.
- **SUSPEND_YOURSELF**: This return value is received during an execute operation for a Program bean (Figure 4.17) when the bean sends a READY_EVENT along an input path that contains another Program bean. Whenever a Program bean receives the READY_EVENT from a downstream Program bean it realizes that it must start execution before the requesting program can start. However, depending on the receiving programs type and data connections, it may take a long time to actually produce the data needed by the requesting program. Thus, the receiving program first returns a SUSPEND_YOURSELF return status to indicate that the requesting program should suspend all operations until it is notified and only then begins its own execute operation. When the data needed by the requesting program and produced by the receiving

program is available, the requesting program is notified by sending a DONE_EVENT, which is when it comes out of suspension and continues its own execute operation.

Mechanically, these steps translate into the following method calls. If this is a Program bean, the `inputAction()` method on receiving a SUSPEND_YOURSELF return status invokes the `incrementSuspensionCount()` method. It then suspends all operation until it receives a false return value from the `isSuspended()` method, by executing a `wait()` inside a while loop. The `isSuspended()` method checks the suspension count and return true if the count is greater than zero, otherwise it returns false. When data is available from the upstream program and when a DONE_EVENT is received, the Program bean will invoke the `decrementSuspensionCount()` method and notify the waiting thread which can then resume execution if the event count has reached zero.

The SUSPEND_YOURSELF return status can never be received from an output event and thus by the `outputAction()` method. Also, if received by the `propagateEvent()` method during event propagation to an input connection, only the status of the propagating bean is set to SUSPENDED.

Dialog Methods

All beans need to display certain status and error messages to the user at various times during the composition and execution process. The `BasicBean` class provides a convenient mechanism for doing this. The `msgBox()` and `errBox()` methods accept a message string and create a dialog box that displays the message. The `msgBox()` method accepts an additional parameters which can be used for giving a custom title to the dialog box. These methods use the `MyMessageDialog` and `MyErrorDialog` classes, defined in the same package as the `BasicBean` class, for creating the dialog box. A reference to a `java.awt.Frame` object is needed for creating a modal dialog box. An interesting point to note is that this reference must be obtained in a manner independent of the BeanBox container. Since any bean container will always have an independent frame that contains the workspace, a reference to the frame can be obtained by searching upwards in the containment chain of AWT Container objects starting from the bean itself. This is exactly what is done by the `getBeanBoxFrame()` method.

4.6.2 Extending BasicBean

This section builds upon the description of the `BasicBean` class in the previous section and describes the implementation of the core beans in general. Figure 4.25 shows the methods implemented by a core bean. It must implement the methods from `InputInterface` and `OutputInterface` which are not implemented by `BasicBean` as well as the abstract event handling methods defined by `BasicBean`. Additionally, if a core bean has a GUI (such as an additional popup menu) that differs from the simple GUI provided by `BasicBean`, it must

Figure 4.25: Methods to be Implemented by all Core Beans

```
public CoreBeanName extends BasicBean implements Seriazable {

    // === Methods from the InputInterface and OutputInterface
    // === left unimplemented by BasicBean
    int    getBeanType () ;
    String getBeanTypeName () ;

    public InputStream getInputStream (BaseInterface x) throws Exception ;
    public void checkConnectionTo (OutputInterface x) throws Exception ;

    public OutputStream getOutputStream (BaseInterface x) throws Exception ;
    public void checkConnectionFrom (InputInterface x) throws Exception ;

    // === Abstract Event handling methods from BasicBean
    public abstract void verify_handler (PortEvent x) ;
    public abstract void stop_handler   (PortEvent x) ;
    public abstract void ready_handler  (PortEvent x) ;
    public abstract void start_handler  (PortEvent x)
       throws EventException ;
    public abstract void done_handler   (PortEvent x) ;

    // === Overridde methods from the java.awt.event.MouseListener and
    // === java.awt.event.MouseMotionListener interfaces for providing
    // === a new GUI
    ...

    // === Methods for creating a GUI for the bean
    ...

    // === Accessor methods for new properties defined in the core bean ===
    ...

    // === Other bean specific methods
    ...
}
```

define methods to create this GUI, and also override relevant mouse event handling methods defined in the `BasicBean` class. A core bean can define its own properties by implementing property accessor methods.

Every core bean separately implements the `Serializable` interface. Every bean has a BeanInfo class and a Customizer class as described in Section 4.2. The BeanInfo class for every core bean is very similar to the SampleBeanBeanInfo class show in Figure 4.3. All core beans expose a single method `eventSend`, a single event `connection`, and define their own icon image and customizers. A customizer class for a bean has the same framework as that of the SampleBeanCustomizer shown in Figure 4.5, except that the customizer includes fields for customizing all properties used by the bean.

From the methods shown in Figure 4.25, some of the methods have a similar implementation for each core bean and hence can be explained as follows:

- `getBeanType`: For all core beans this method returns an appropriate integer value from the set of bean type values defined in `BaseInterface` (Figure 4.7). The Program bean, for example, returns the `BaseInterface.PROGRAM` value.
- `getBeanTypeName`: This method returns the bean type as a string value. Thus for a Program bean it returns the string "Program".
- `getOutputStream`: As described in Section 4.3.3, this method just returns a null value for every core bean.
- `checkConnectionFrom`: This method first checks to see if the input bean requesting the connection is of the valid type as shown by Table 3.3. For beans other than a Program bean, this method throws an exception if there is already an input connection to the bean and for a Socket bean it throws an exception if there is already an output connection. Recall that, a Socket bean can either have an input connection or an output connection, not both. However, an additional check is performed for the Program bean. If the input bean requesting the connection is a Stdin bean, a check is made to ensure that there is not already a Stdin bean that is connected to this Program bean. If there is, then the connection is not allowed.
- `checkConnectionTo`: For all beans other than a Program bean or a File bean, this method throws an exception if there is already an output connection for the bean. For a Socket bean an exception is thrown even if there is an input connection. An additional check is performed for a Program bean where if the target bean is a Stdout (Stderr) bean, and if a connection already exists to a Stdout bean from this Program bean, the new connection is not allowed.

The following sub-sections describe the implementation of the individual core beans. Following things are explained for each bean: new properties implemented by the bean, properties

used from the `BasicBean` class, event handling methods, `getInputStream()` method and the GUI differences from `BasicBean`.

4.6.3 Program Bean

The Program bean uses all properties defined in the `BasicBean` class and defines some additional properties: program type, HTTP URL, parameter string, environment variable strings, X-terminal display string, X-windowing program flag and display terminal I/O flag. All these properties can be collectively customized by using the Program bean customizer.

The Program bean provides a popup menu with three choices (Start, Stop, Verify) which is created in the bean constructor. The bean overrides the `mouseClicked` method from `BasicBean` for displaying the popup menu. It implements the ActionListener interface and registers with itself for listening to action events generated when the user selects a menu-item from the popup menu. The actionPerformed method from the ActionListener interface redirects the user request to the appropriate method in the Program bean: verify_handler() for the verify operation, start() method for the execute operation, and stop_handler() for the stop operation.

The method implemented by the Program bean for carrying out meta-program operations and for handling events are described below:

- `verify_handler`: The following algorithm shows the list of things that are verified for a Program bean when it generates or receives the verify event.

```
if (program type = web accessible)
   invalid if stream beans are connected
   invalid if url doesn't begin with http
     (it must be remembered that currently there is not way of
      verifying that the remote program exists without actually trying
      to load the contents of the URL. That approach is not taken
      because for a CGI program it effectively amount to executing the
      program which may not always be desirable)

if (program type = user accessible)
   create a ProgramDefinition object and instantiate
      a ProxyProcess object. The ProxyProcess constructor will contact
      the ProgramServer on the remote host and invoke the getProcess
      method on it. The getProcess method will first use the login
      information to authenticate the user an then try to verify that
      that program file is accessible by creating a File object and
      checking the permissions.
```

- `start`: The actions that take place when this method is invoked are outlined by the algorithm shown in Figure 4.17. The `start()` method is invoked under 3 conditions:

1. When the user chooses the Start menu-option in the Program bean popup menu
2. When a ready event is received from a downstream bean (ready_handler)
3. When a done event is received from an upstream bean (done_handler)

This method starts the execute operation for this program by creating a new execution thread which carries out the algorithm shown in Figure 4.17.

- `ready_handler`: This method is invoked either when a ready event is received from a downstream Program bean or from the execution thread started by the `start()` method. In the first case, it starts the execute operation for this Program bean by invoking the `start()` method and in the second case, it sends a ready event to all input beans. Sending the input event to all input beans may have an implicit side-effect of suspending the execution thread if there is a upstream Program bean that needs to execute before this program, as explained in the description of the event handling methods for the `BasicBean` class. This suspended thread will be resumed when a done event is received from the upstream Program bean.

- `start_handler`: This method is invoked by the execution thread during an execute operation for sending the start event to any connected standard stream beans. The standard input bean on receiving the start event obtains the input stream from its input bean and saves it, and the Stdout and Stderr beans obtain the respective streams from this Program bean and send done events to their respective output beans to signal that data is available.

- `done_handler`: This method can be invoked in two scenarios: by the execution thread during an execute operation for sending done events to output beans after program execution has finished, or when a done event is received from an input bean. A done event, received from an upstream Program bean, can either be in response to a ready event that was sent to it by this bean (in which case the suspended thread of this bean must be woken up) or it can be a unsolicited done event in which an execute operation must be started for this program by invoking the `start()` method.

- `stop_handler`: This method is invoked either when the user chooses the Stop menu option from the Program bean popup menu or when a stop event is received from some other bean. If the status is RUNNING and program type is user accessible, this method stops the active program, closes any open dialog box, kills the data transfer and execution threads, and sends/propagates the stop event to all connections.

- `getInputStream`: If the status of the bean is RUNNING and program type is user-accessible, this method returns a reference to the InputStream for the active program's standard output if the invoking bean is a Stdout bean, and a reference to the InputStream for the program's standard error, if the invoking bean is a Stderr bean.

4.6.4 File Bean

The File bean uses all properties defined in the `BasicBean` class and defines some additional properties: file type, HTTP URL and FTP URL.

The File bean provides a popup menu with the single item Show which, if clicked, loads the file in a browser window for a file which is defined by a URL. Currently, nothing is done for other file types. The bean overrides the `mouseClicked` method from `BasicBean` for catching mouse clicks. It implements the ActionListener interface and registers with itself for listening to action events generated when the user selects the Show option from the popup menu. The actionPerformed method which is invoked when the action event is generated invokes the `showFile()` method which actually loads a URL-accessible file in a browser window.

The method implemented by the File bean for carrying out handling events received during meta-program operations are described below:

- `verify_handler`: The following algorithm shows the list of things that are verified for a File bean when it receives the verify event.

```
if (file type = web accessible)
   invalid if (there is an input connection)
   invalid if URL doesn't begin with http
   try to create a connection to the URL

if (file type = ftp accessible)
   invalid if url doesn't begin with ftp
   try to create the connection to the URL

if (file type = user accessible)
   create a FTP client for the required host machine
   try to login with the given username and password
   try changing directory to the given directory
   if (there is a input connection)
      try getting a stream to the file
   if (there is an output connection)
      you cant really do anything because if you try to
      create an output stream to the file, you will effectively
      delete the existing file on the machine if there is one and
      you may not always want to do that

if (file type = local file)
   if (there is an input connection)
      try and get an input stream to the local file using FileInputStream
   if (there are output connections)
      check file path, name, and permissions by using the File class
```

  If verification is successful, status is set to VERIFIED and the verify event is propagated.

- `ready_handler`: If there is no input connection for the File bean, this method just sets the status to COMPLETED and returns. If there is an input connection, however, the file represented by this bean needs to be created/updated when the ready event is received. Thus data must be obtained from the input bean for which the ready event is first propagated to the it. Successful propagation means that data is available and an input stream can be obtained from the input bean.

  – If the input bean is a Program bean nothing needs to be done because the program will automatically create the file when it executes on receiving the ready event. In this case and in the more general case when there is a Program bean in the input path upto the File bean, the event propagation will return a SUSPEND_YOURSELF return status. Here, the ready_handler should just return without any further processing assuming that the file will be created/updataed automatically, or when the done event is received from the upstream program, whatever the case may be.

  Once the data is available, this method obtains the input stream from the input bean, creates the output stream to the file, and writes over the input data obtained from the input stream to the file stream. This data transfer is blocking and is not done in a separate thread because the entire file needs to be written before it can be termed as ready for reading by the output bean. Status is set to RUNNING before the data transfer is started and set to COMPLETED when it's done. The exact method of obtaining the output stream for writing to the file depends on the file type. A `sun.net.ftp.FTPClient` object is created for an anonymous FTP-accessible file or a user-accessible file, which is then used to obtain the output stream to the file. For a local file, a `java.io.FileOutputStream` object is created with the file path and name as parameters. It must be noted that an HTTP-accessible file is not allowed to act as an output file.

- `start_handler`: This is just a dummy method since the start event is never received by a File bean.

- `done_handler`: The done event is received by a File bean when data is available from an upstream program. If this File bean is directly connected to a Program bean on input, the event is received only after the program has finished execution and created the file represented implicitly. In this case, only the existence of the file is verified and the done event is propagated to the output connections, if any. If the input bean is not a Program bean, an input stream is obtained from it, an output stream is created to the file represented, and the input data is written to the file stream. Again, this data transfer is blocking and the method of obtaining the output stream is dependent on the file type, just as in the ready_handler. The status is set to RUNNING before starting the data transfer and set to COMPLETED when the transfer is done. The done event is propagated to output connections, if any, after the transfer is done and the file has been completely updated/created.

- `stop_handler`: On receiving a stop event any active data transfer is aborted by closing the input and output data streams, the status is set to ABORTED, and the stop event is propagated.

- `getInputStream`: This method is invoked for a File bean that has an output connection when the output bean wishes to read data from the file represented. This method presupposes that the file has already been created/updated for this run of the meta-program by a preceeding ready or done event. The method used for actually obtaining an InputStream to the file depends on the file type. A URLConnection object is created for an HTTP-accessible and anonymous FTP-accessible file which is used to obtain the input stream. A `sun.net.ftp.FTPClient` client object is used for a remote user-accessible file and a `java.io.FileInputStream` object is created for a local file.

4.6.5 Socket Bean

The Socket bean does not define any new properties but uses the host, port, status, title and description properties from `BasicBean`.

Recall that a Socket bean can either be a data source or data sink, not both. The operations performed in the event handling methods for a Socket bean depend on the whether it represents a input socket or an output socket. However, it must be remebered that a event received by a Socket bean never needs to be propagaged since it has only one connection. The event handling methods are described below:

- `verify_handler`: For a Socket bean, depending on whether the bean is a source bean or a sink bean, verification only consists of trying to create a socket connection to the specified host and port. The connection is not used for reading or writing any data and is closed once it has been verified that the connection can be made. The socket information is represented by the host name and port number properties which can be set using the socket customizer. Status is set to VERIFIED on successful verification.

- `ready_handler`: The ready event is received by an input Socket bean during an execute operation. Since all beans must attain the COMPLETED status when the meta-program is done executing, this method just sets the status to COMPLETED if the current status is VERIFIED. It does not perform any data transfer.

- `start_handler`: This method is never invoked for a Socket bean since it never receives a start event.

- `done_handler`: The done event is received by an output Socket bean during an execute operation, when data is available from the input bean. Since Symphony follows a data pull model, this method is responsible for obtaining the data from the input bean and writing it to the output socket stream. The transfer is done in a separate thread which

just copies over the data obtained from the input bean to the socket stream. Before the thread is started, the status is set to RUNNING and it is set to COMPLETED by the thread once the transfer is done.

- `stop_handler`: This method closes the appropriate stream connection (input socket stream or output socket stream), if open and sets the status to ABORTED.
- `getInputStream`: For an input Socket bean, this method creates the actual socket connection and returns an InputStream for this connection for the output bean to read from.

A read call on a InputStream obtained from the `Socket.getInputStream()` method can block if data is not available on the socket. Since the InputStream object obtained in this manner is returned directly to the output bean, this blocking behavior may not always be desirable. Hence, to get around this problem, the socket read time-out is set to 1 second by using the `Socket.setSoTimeout()` method.

4.6.6 Stream Beans

The stream beans represent the standard streams with which a legacy process interacts during execution: standard input (stdin), standard output (stdout), and standard error (stderr). The Stdin bean only serves to redirect input from its source to the program's standard input stream and the Stdout and Stderr beans redirect output on the program's respective streams to their respective output beans. The stream beans do not implement any new properties and only use the status, title and description properties from the set of properties defined by `BasicBean`. Also, these beans do not add anything to the simple GUI provided by `BasicBean`.

It must be rememebered that the input bean for a Stdin bean may be any data source bean, but the input bean for a Stdout bean and a Stderr bean is always the Program bean. The event handling methods implemented by the stream beans are described below:

- `verify_handler`: For all the stream beans, the only thing that is checked during verification is whether or not the bean is connected on both sides; verification fails if it is not. If verification is successful, status is set to VERIFIED and the event is propagated.
- `ready_handler`: The ready event is simply propagated to the input bean and if the propagation is successful, sets status to READY.
- `start_handler`: For the Stdin bean, this method simply saves a reference to a InputStream object obtained from its input bean by invoking the `getInputStream()` method

on the input bean. For the Stdout (Stderr) bean, the start event does two things: first, it obtains the InputStream representing the standard output (error) stream from the Program bean, second, it sends a done event to its output bean indicating that data is available. Sending the done event in the start_handler itself instead of waiting for the done event from the Program bean enables a streaming transfer of data so that the beans downstream from the Stdout (Stderr) bean can start processing the data as soon as its available.

- `done_handler`: For the Stdin bean this method forwards the done event to the Program bean and if the event is successful sets the status to COMPLETED. For the Stdout and Stderr beans it just sets the status to COMPLETED and returns, since the done event is already sent to the output bean during the start event.
- `stop_handler`: The input stream obtained during the start event is closed if it's open, the status is set to ABORTED and the stop event is propagated.
- `getInputStream`: For all the stream beans, this method returns the input stream reference obtained from the input bean (which is the Program bean in case of Stdout and Stderr and any data source in case of stdin) during the start event.

4.7 Abstract Beans

This section describes the implementation of the abstract beans. Symphony provides three abstract beans: Producer, Consumer, Filter. The abstract Filter bean is an extension of the `BasicBean` class, similar to the core beans. Although the Producer and Consumer beans do not inherit from this base class, their implementation follows a similar model. The following sub-sections outline the implementation of each of these beans.

4.7.1 Producer Bean

Figure 4.26 show the inheritance diagram for the abstract `Producer` class. As can be seen from the diagram, the `Producer` class includes similar functionality to the `BasicBean` class. Since a Producer bean can only act as a data source, it only needs to implement `InputInterface` for satisfying the Symphony communication mechanism. The `Producer` class inherits from the `java.awt.Panel` class and provides methods to handle mouse events by implementing the `MouseListener` and `MouseMotionListener` interfaces. For serializability, it implements the `Serializable` interface and for listening to bound property changes, it implements the `PropertyChangeListener` interface.

Figure 4.27 shows the methods implemented by the `Producer` class. As can be seen from the figure, the user interface, dialog and property methods are similar to those in the `BasicBean`

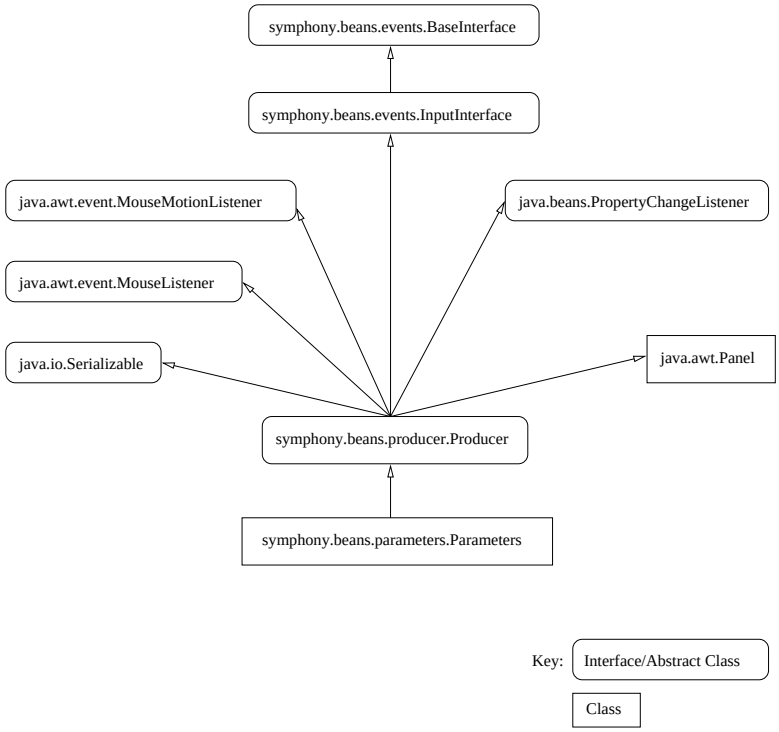

Figure 4.26: Inheritance Structure for Producer Beans

class, the only difference being that only three properties are defined: status, title and description. It implements the methods needed by a source bean to support connections as shown in Figure 4.14. The implementation of these methods follows the same logic as explained in Section 4.3.5. Also as described in Section 3.6, it defines four abstract methods that must be implemented in order to extend the `Producer` class to implement a Producer bean.

Since a Producer bean can be a part of a meta-program, it can also receive events during meta-program operations. The purpose of the `Producer` class is to hide all details regarding the Symphony architecture. Thus, it implements event handling methods for all event types that can be received by a Producer bean and if needed invokes one of the abstract methods it defines. A Producer bean can receive three types of events from the set of events events defined in `BaseInterface`: VERIFY_EVENT, READY_EVENT, STOP_EVENT. The `eventSend()` method redirects the received event to the appropriate event handling method: `verify_handler()`, `ready_handler()` and `stop_handler()`. If an event of unknown type is received, it is ignored by the `eventSend()` method. The `verify_handler()` method invokes the abstract `verify()` method so that the inherited class can perform verification of its resources as described in Section 3.6. The `ready_handler()` method invokes the abstract `start()` method, obtains the `InputStream` object from it and saves a refer-

ence to it. This reference is returned when the output bean invokes the `getInputStream()` method. The `stop_handler()` method invokes the abstract `stop()` method in order to abort execution any activity that may have been initiated by invoking the `start()` method.

The `Producer` class also implements the remaining methods from `InputInterface`: the `getBeanType()` method which returns the type `BaseInterface.PRODUCER` and the method `getBeanTypeName()` which invokes the `getBeanName()` method for obtaining the actual name of the implemented bean.

4.7.2 Consumer Bean

Figure 4.28 show the inheritance diagram for the abstract `Consumer` class. Like the Producer bean the Consumer bean also inherits from the `java.awt.Panel` class. Since a Consumer bean can only acts as a data sink, it needs to implement `OutputInterface`, but not `InputInterface`. All other interfaces implemented by the `Consumer` class are similar to that of the `Producer` class.

Figure 4.29 shows the methods implemented by the `Consumer` class. User interface, dialog and property methods are similar to those in the `BasicBean` class, and three properties are defined: status, title and description. The `Consumer` class implements all the methods needed to be implemented by a sink bean, as shown in Figure 4.14. The implementation of these methods follows the same logic as explained in Section 4.3.5. It defines four abstract methods that must be implemented in order to extend the `Consumer` class to implement a Consumer bean.

Just like the Producer bean, the purpose of the Consumer bean is to hide the details of the Symphony architecture, and hence it implements event handlers for the three types of events (other than connect and disconnect events) that can be received be received by a Consumer bean. The `verify_handler()` method which is invoked when a verify event is received invokes the abstract `verify()` method. The `done_handler()` method is invoked when a done event is received. It obtains an `InputStream` from the input bean by invoking its `InputInterface.getInputStream()` method and uses it to call the abstract `start()` method for starting data consumption. The `stop_handler()` method invokes the abstract `stop()` method in order to abort execution any activity that may have been initiated by invoking the `start()` method.

The `Consumeer` class also implements the remaining methods from `OutputInterface`. The `getBeanType()` returns the type `BaseInterface.CONSUMER` and `getBeanTypeName()` returns the string obtained by invoking the abstract `getBeanName()` method. A null value is returned by the `getOutputStream()` method.

Figure 4.27: Methods implemented by the abstract `Producer` class

```
// ===== User Interface Methods (similar to BasicBean) =====
// ===== Property Methods (similar to BasicBean) =====
         // Propperties defined: title, status, description
// ===== Dialog Methods (similar to BasicBean) =====

// ===== Connection and Event Handling Methods ======

public synchronized void addConnectionListener (ConnectionListener l)
   throws Exception ;
public synchronized void removeConnectionListener (ConnectionListener l)
   throws Exception ;
public void removeOutputConnection (OutputInterface x) ;
public void checkConnectionTo (OutputInterface x) throws Exception ;

public void eventSend (PortEvent x) throws EventException ;
public void verify_handler (PortEvent x) ;
public void stop_handler   (PortEvent x) ;
public void ready_handler  (PortEvent x) ;

// ===== Other Methods from InputInterface =====

public int getBeanType() ;
public int getBeanTypeName() ;
public InputStream getInputStream (BaseInterface x) ;

// ===== Abstract Methods to be Implemented by Actual Beans ====

public abstract boolean verify() throws Exception ;
public abstract InputStream start() throws Exception ;
public abstract void stop() throws Exception ;
public abstract String getBeanName() ;
```

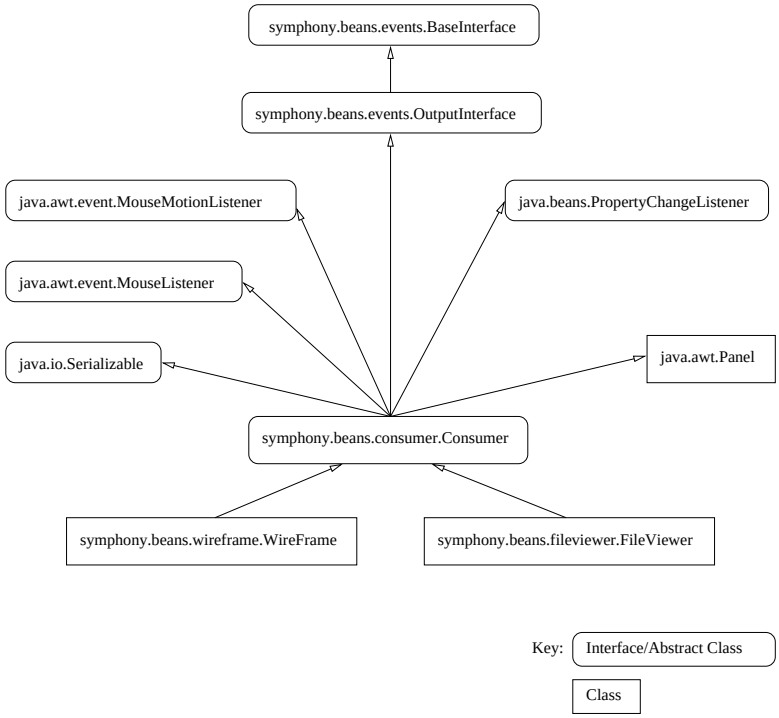

Figure 4.28: Inheritance Structure for Consumer Beans

4.7.3 Filter Bean

The abstract `Filter` class inherits from the abstract `BasicBean` class. However, it implements all the methods shown in Figure 4.25 and defines the four abstract methods as described in Section 3.6. The `Filter` class does not define any new properties or GUI methods apart from what it derives from `BasicBean`. The Filter bean uses `PipedInputStream` and `PipedOutputStream` objects from the `java.io` package, for stream connections. The methods implemented by the `Filter` class can be explained as follows:

- `getBeanType`: Returns the type `BaseInterface.FILTER`
- `getBeanTypeName`: Invokes the abstract `getBeanName()` method
- `getOutputStream`: Returns a null value
- `verify_handler`: Invoked when a verify event is received, it invokes the abstract `verify()` method and propagates the event if verification is successful
- `ready_handler`: Invoked when the ready event is received from the output connection and only propagates the received event to the input connection.

Figure 4.29: Methods implemented by the abstract `Consumer` class

```
// ===== User Interface Methods (similar to BasicBean) =====
// ===== Property Methods (similar to BasicBean) =====
         // Propperties defined: title, status, description
// ===== Dialog Methods (similar to BasicBean) =====

// ===== Connection and Event Handling Methods ======

public void connect_handler    (PortEvent x) throws EventException ;
public void disconnect_handler (PortEvent x) throws EventException ;
public void eventSend (PortEvent x) throws EventException ;
public void removeInputConnection (OutputInterface x) ;
public void checkConnectionFrom (InputInterface x) throws Exception ;

public void verify_handler (PortEvent x) ;
public void stop_handler   (PortEvent x) ;
public void done_handler  (PortEvent x) ;

// ===== Methods from OutputInterface =====

public int getBeanType() ;
public int getBeanTypeName() ;
public OutputStream getOutputStream (BaseInterface x) ;

// ===== Abstract Methods to be Implemented by Actual Beans ====

public abstract boolean verify() throws Exception ;
public abstract start( InputStream in ) throws Exception ;
public abstract void stop() throws Exception ;
public abstract String getBeanName() ;
```

- `start_handler`: Just a dummy method which will never be called because the start event is sent only to standard stream beans
- `done_handler`: Instantiates `PipedInputStream` and `PipedOutputStream` objects and connects them using the `connect()` methods of the respective classes. It then creates a new thread for performing the data transfer and in the thread invoked the `filter()` method with the InputStream object obtained from the input bean and the PipedOutputStream object as parameters The filter method will read input data from the input stream and write filtered data to the output stream.
- `stop_handler`: Stops all data transfer and kill any transfer threads.
- `checkConnectionTo`: Rejects the connection if there already is one output connection
- `checkConnectionFrom`: Rejects connections from invalid types of input beans and rejects the connection if there already is one input connection
- `getInputStream`: Returns a reference to the `PipedInputStream` object created by the ready_handler or the done_handler method. The data read from this piped input stream will represent the data written by the filter() method to the piped output stream.

4.8 Extending and Adapting the Framework

This section describes the step by step process of creating a completely new core or abstract bean for Symphony and also explains how Symphony beans were adapted to be used in the Sieve environment

4.8.1 Adding a New Bean to the Enviroment

All the Symphony source files are under the 'src/symphony' directory in the Symphony_src.tar file. Following are the steps that must be followed in order to create a new Symphony bean:

- Say the name of the new bean to be created is NewBean. Create a new directory named 'newbean' under the 'src/symphony/beans/' directory.
- Create the bean source file, named 'NewBean.java' in this new directory. This bean can either inherit from the BasicBean class, or from one of the abstract beans, or can be a totally new bean. If it inherits from the BasicBean, use one of the core bean's source file to make this new bean. If it inherits from an abstract bean use the source files of one of the beans implemented from the correct type of abstract beans. If it is a totally new type of bean, use the SampleBean files from the 'src/symphony/beans/sample' directory.

- If this bean is not being created from an abstract bean, the type of this bean needs to be added to the 'src/symphony/events/BaseInterface.java' file. If this bean will be used to connect as input bean to other beans, the bean type needs to be added in the isConnectionAllowedFrom() method of every possible target bean.
- Create a BeanInfo class for the new bean. The SampleBeanBeanInfo class can be used as a starting point for this.
- Create a customizer for the new bean. The SampleCustomizer.java program can be used as a starting point for this.
- If the new bean is not an abstract bean, add the entry for the bean in the Symphony Manifest file in the 'symphony' directory.
- Create a new makefile for the new bean. A makefile for one of the existing beans can be used for this. Add the name of the new makefile to the list of makefiles in the 'makefile' file. Execute the 'make' command in order to compile the class files for the newly created beans and add them to the Symphony.jar file.

4.8.2 Collaborative Composition in Sieve

This section describes how the Symphony beans were adapted for collaborative composition the Sieve [13] environment. As described in Section 2.2.2, Sieve provides a JavaBeans-based shared workspace where multiple users can collaboratively add, edit, and link components to build a network of components. It provides an ability for existing JavaBeans-based applications, like Symphony, that adhere to standard beans mechanisms to be used in a collaborative manner. In Sieve, collaboration is achieved by broadcasting changes in bound properties of a bean.

Since all properties supported by Symphony beans are bound properties, these beans can be imported unmodified in the Sieve environment. This allows meta-programs to be composed in a collaborative manner. Like Sieve, Symphony provides a data-flow based mechanism for composing networks of bean components. The mechanism of linking beans in Sieve, however, is different from the regular BeanBox mechanism described in Section 3.2. In Sieve, instead of going through the menu-based mechanism and the generation of adapter classes as in BeanBox, the user just clicks and connects two beans. A right click on the border of the source bean starts a link which can be completed by clicking on the border of the target bean. Sieve instantiates a new link object for each new link. This link object is responsible for creating the communication structure between the two beans. Thus, to be able to connect two Symphony beans together in Sieve and to create the communication framework described in Section 4.3, a new link class was created (`InputInterfaceOutputInterfaceLink`). Objects of this class, when instantiated, invoke the appropriate methods in the source and target beans to create the required communication structure.

Chapter 5

Conclusions and Future Work

This chapter summarizes the capabilities and contributions of the Symphony framework, enumerates its limitations, and provides directions for possible future work based on the Symphony framework.

5.1 Conclusions

Symphony attempts to address the problems outlined in Section 1.1 for scientific and engineering problem-solving tasks that involve the use of distributed legacy resources. It allows users to visually compose legacy programs and data distributed on different machines by specifying the data-flow relationships among them. These programs and data can be used without any modifications. It also allows transparent execution of the composed application in a manner that respects the data-flow requirements of executable components. Execution transparency means that all system-level operations of program execution and of moving data across geographically distributed locations are largely transparent to the user.

Symphony has been implemented as a set of Java beans which can be customized and composed in a beans builder tool. It provides six core beans for representing legacy resources: Program, File, Socket, Stdin, Stdout and Stderr. These beans can be used to build meta-programs based on the data-flow patterns between executable components. The Program bean communicates with the Symphony server for executing remote applications. The server has been implemented by using the Java Remote Method Invocation (RMI) mechanism.

Other goals for the system identified in Section 1.1 were for it to be: extensible, open, generic, web-aware, persistent, secure and graphical.

Symphony enables extensibility by providing abstract beans which can be extended to add new bean types to the environment. Three abstract beans have been provided: the Producer bean, the Consumer bean, and the Filter bean. These beans can be used to implement

components that act as data producers, consumers, or filters respectively. The abstract beans do not define the manner in which data is obtained, consumed or transformed, which provides sufficient flexibility to programmers who wish to implement new bean types.

Use of standard JavaBeans mechanisms makes the Symphony framework open in the sense that the set of beans can be used in any standard JavaBeans container. This has enabled the application of Symphony components in Sieve, a JavaBeans-based collaborative workspace, where multiple users can collaborate on composing a Symphony application in real-time.

Although, work on Symphony was initiated from the perspective of science and engineering applications, the system is sufficiently generic to be used for visually composing and executing any set of distributed resources outside of this context.

The Program and File beans allow the user to specify web-accessible resources. Thus web-based resources can be effectively composed with legacy resources. The composed application can also be saved to persistent storage and reloaded later for user or modification. Symphony also provides basic security in terms of username/password authentication for protected resources.

The Producer abstract bean can be used to implement new bean types that provide graphical interfaces to legacy applications as illustrated the Parameters bean. Similarly, the Consumer abstract beans can be used to create visualization components for viewing results. This has been illustrated by implementing the FileViewer and WireFrame beans.

5.1.1 Limitations

This sub-section outlines limitations of the Symphony framework. Some of these limitations stem from the limitations of the current JavaBeans architecture and some from the time constraints faced during the design and development of Symphony. Given more time and research, most of the limitations listed below can be addressed in a satisfactory manner by employing the means described in the next section.

- Although, Symphony does allow the user to store meta-programs in persistent storage for modification and use at a later time, there is currently no mechanism where a meta-program can be saved as an aggregate bean which can be loaded back into the bean container as an individual bean.
- The data-flow model provided by Symphony does not support explicit control operators. Also, data-flow loops are not supported.
- The Symphony data routing mechanism is not efficient in cases where data needs to be transferred between remote sites, since all data must flow through the composing environment regardless of its source and destination.

- Assuming that a web browser with support for beans was available, an ability to encapsulate Symphony meta-programs as applets which can be accessed and executed from web browsers would make them accessible to a wide variety of platforms and hardware architectures without any prior setup.
- The simplistic security mechanism used by Symphony is not correct for scenarios where remote access must be provided to certain applications without requiring authentication. Also, authentication is currently carried out through the FTP mechanism and hence an FTP daemon needs to be executing on all host machines from which private user-accessible resources are read or written to.
- Symphony does not allows transparent execution of applications on remote compute servers. The onus of setting up the execution is on the meta-program designer, not on the application developer. This is a trade-off because even though it places more burden on the designer, it provides more flexibility.
- Symphony does not provide fault-tolerance.

5.2 Future Work

This section lists several areas of future work that have been identified during the course of using Symphony to create meta-programs and by reasoning about how Symphony could be applied to more challenging problems.

Customization

The convenience afforded by the Properties bean in easing customization of meta-programs can be improved by including additional functionality in the customizers for the Program, File and Socket beans. Currently the rule is that if a properties bean has been instantiated in the workspace, any new bean that is instantiated thereafter is given default property values which reflect the values set in the Properties bean. However the Properties bean has no effect on the beans that already exist in the workspace when it is instantiated. An improved strategy would also allow existing beans in the workspace to use the values set in the Properties bean. customization. This involves adding a checkbox for each property field in the customizer which, if checked, indicates that the default value from the Properties bean must be used for this particular property and not the value set in the bean customizer.

Aggregation

There should be a way to aggregate a composed meta-program into a independent bean which can be manipulated as a single bean like any other bean. This aggregate can have

it's own representation on the workspace and it may be expanded to see and manipulate the constituent beans of the meta-program. An aggregate bean can also have other properties of its own which represent the properties of the meta-program as a whole.

The Properties bean could be used to define the properties of such an aggregate bean and if the customization functionality described in the previous sub-section is available, all beans in the meta-program could be customized to use properties only from a Properties bean associated with the aggregate. More research is needed to determine what properties and operations a meta-program should have as a whole (e.g., how would an aggregate be executed).

Data Routing Efficiency

During meta-program execution, all data is currently routed through the BeanBox for ease of development and testing. In some cases this is not efficient as it involves two or more copy operations to transfer the data from the source to destination instead of a single copy. For example, if data from a program's standard output is supposed to go to a file on the same host machine as the program, the remote data stream obtained by the program bean is passed on to the file bean which in turn creates a remote FTP stream to the file. Thus in this case, instead of making a direct connection between the program's standard output stream and the file which are on the same machine, the data has to flow through the beans in the BeanBox. This can be very inefficient if large amounts of data needs to be transferred. One solution to this problem is to decouple the data copying operations from the data control operations, transferring the copy responsibility to a server-side entity. In this case the beans would instruct server-side entities to perform the required data transfer at the right time and provide them with enough information about the source and destination of the data to be routed. A particularly appealing approach would be to use agent-based communication where beans can send agents to remote servers for performing the required data transfers and program executions. The agents would in turn notify the beans of the completion or failure of the data transfer or execution tasks. Thus, the beans would only be responsible for controlling the data flow, and not actually doing the data transfers.

Control-Flow and Loop Control

The connection mechanism in Symphony only allows data-flow connections. There is currently, no way to depict a control connection between beans instead of a data connection. It would be useful to be able to specify control connections between program beans even if there is no data dependency between programs, such that when one program finishes, it automatically triggers execution of the next one. Such a capability would be useful in several scenarios such as for sequencing the execution of concurrent programs executed on the same machine to avoid overload, or to force error-prone programs to execute first in order

to avoid unnecessary computation if they fail. Although, this can currently be simulated by using an additional file bean connected as an output bean for the first program and an input bean for the second program, a cleaner way of depicting these type of connections is needed. In addition, a control connection would look different than a data connection in the user-interface to more clearly express the meaning of the connection. An alternate would be to create a trigger bean with the sole function of triggering the next program when the preceding program is done.

Also, currently Symphony beans cannot handle loops in the meta-program. Any operation on a meta-program with loops could result in a deadlock of the beans in the loop because Symphony operations currently do not have a time-stamp or a sequence identifier. If each operation could be given an identifier that would remain valid for one execution, this identifier could be used for events passed during a meta-program operation and if an event for a future operation is received, the bean can queue it for later action. If an event for a past operation is received the bean will ignore the event.

Security

Symphony currently provides a minimal level of security which involves username and password authentication for remote resources. Explicit security features could be included as part of the Symphony execution environment. Security can be addressed at three levels:

- **Client-side security:** Currently it is not possible to package a Symphony meta-program as an applet because of the restrictions of the applet security model. If however, it were possible to package the beans as an applet and sign the applet with a digital signature, the applet could be run from within a browser with enhanced privileges.
- **Server-side security:** For some applications, the simplistic username/password security can become cumbersome, especially if more than one person needs to use a meta-program. There must be a mechanism which allows execution of certain programs on a server machine without requiring login authentication. An example of such a mechanism would be to use a dictionary approach where every execution request is authenticated against a program dictionary. Execution is allowed if an only if the request matches an entry in the dictionary. Specialized mechanisms could be used for managing archiving of results and periodic cleanup of directories. Also, currently for executing a program on a remote server using the Symphony server, the program must have execute permissions for the user under whose account the Symphony server was executed. Another copy of the Symphony server needs to be executed from another user's account if a program owned by that user needs to be accessed.
- **Communication security:** The RMI communication mechanism can be made secure by employing a Secure Sockets based transport that uses the SSL protocol. RMI allows

the programmer to replace the default RMI socket-based transport with any other transport mechanism.

New Types of Abstract Beans

Several new types of abstract beans can be added to enrich the set of beans provided by Symphony. Abstract beans provide a convenient mechanism for extending the set of Symphony beans by implementing simple interfaces. One idea for a new abstract bean is a Gather bean that takes data from several input sources, processes the input data, and provides a single output stream to a single output bean. The bean would let the programmer specify the number of inputs and the manner in which the inputs are to be processed to obtain the output. This bean could allow concurrent execution along its input paths. The Gather bean could be complemented with a Scatter bean with the opposite functionality. Figure 5.1 depicts possible examples of Gather and Scatter beans.

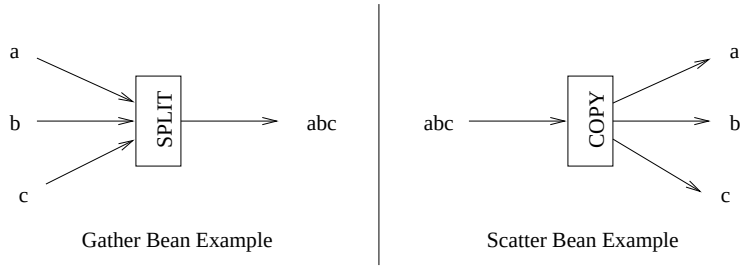

Figure 5.1: Examples of New Abstract Beans

Another useful set of abstract beans would be beans that represent control-flow operators. Currently, Symphony follows a pure data-flow model where there is no provision for specifying control flow except for what is implicit in the data flow. It would be useful to have control-flow operators such as those available in Khoros [27]. Examples include, if-then-else operators and loop control operators.

Middleware Layer

Symphony uses a two layer architecture where the client layer interacts directly with servers that access resources. While reviewing the various types of work that is related to Symphony as described in Chapter 2, it was found that a middle-ware layer that sits between the client and server can be useful in several ways:

- **Accessibility:** It would be very useful if a meta-program could be packaged as an applet, in which case a browser supporting beans could be used to view, customize,

and execute the meta-program. Currently this idea is severely restricted by the applet security model because the beans in a meta-program may need to access resources from a variety of host machines and from the local machine. This problem can be solved if there is a centralized server on the host machine from which the applet is served, which can be contacted by the interface beans first for any transaction with another host machine. For example, for accessing an file on a host machine other than the server machine for the applet, the data stream can be routed through the server machine instead of creating a direct connection to the required host for reading or writing the file. This, of course wouldn't be efficient in some cases, but it can provide wider accessibility to a meta-program built by some one for use by others.

- **Fault Tolerance:** Introduction of a middle tier which is responsible for taking execution requests from the interface beans and passing them onto servers for performing the actual operations can enable fault-tolerance. This concept is similar to that used by NetSolve where a NetSolve agent accepts requests from interface clients and passes them onto NetSolve servers. Fault tolerance is implemented using a simple mechanism which restarts a failed or disconnected application.

This idea was not pursued in Symphony because it became evident only during the later stages of development. However, a middleware layer can provide a complimentary set of functionalities to those currently provided by Symphony.

User Interface Issues

It might be useful to collectively visualize common properties of beans in a meta-program. For example, if every socket, program, and file bean can be annotated on-screen with the host name for each resource it can help in accessing the overall meta-program structure and understanding the data-flow relationships. This is very useful for a person trying to use a meta-program built by some one else or a designer of a meta-program trying to insure that all necessary customization (in this case for host machines) has been done.

Although it is possible currently to add annotations to a meta-program by using the annotations bean, such annotations are not visible in the workspace. The annotations bean must be used to view as well as add annotations. A new type of bean which allows adding of on-screen annotations and associating them with meta-program beans can be useful for documenting the meta-program. Sieve, for example, provides users with a capability to annotate the workspace by using lines, arrows and text [13]. Another functionality that could be useful is the ability to embed URLs in annotations which when clicked would open the URLs in a browser window. The Browser bean can be used for this purpose.

REFERENCES

[1] John Ambrosiano, Steve Fines and Mladen Vouk. Problem-Solving Environments in the Year 2000 and Beyond, 1995.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, MA, 1996.

[3] Dimple Bhatia, Vanco Burzevski, Maja Camuseva, Geoffrey Fox, Wojtek Furmanski and Girish Premchandran. Webflow - a visual programming paradigm for web/java based coarse grain distributed computing. *Concurrency: Practice and Experience*, pages 555–577, June 1997.

[4] Kraig Brockschmidt. *Inside OLE*. Microsoft Presss, 1995.

[5] Henri Casanova, Jack Dongarra and Keith Moore. Network Enabled Solvers and the NetSolve Project. *SIAM News, Society for Industrial and Applied Mathematics*, January-February 1998.

[6] K. Mani Chandy, Adam Rifkin, Paolo Sivilotti, Jacob Mandelson, Matthew Richardson, Wesley Tanaka and Luke Weisman. A world-wide distributed system using java and the internet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 11–18, August 1996.

[7] K. Mani Chandy, Joseph Kiniry, Adam Rifkin and Daniel Zimmerman. A Framework for Structured Distributed Object Computing. Technical Report 256-80, 1997.

[8] Jesse Feiler and Anthony Meadow. *Essential OpenDoc*. Addison Wesley, MA, 1996.

[9] Geoffrey Fox and Wojtek Furmanski. Towards web/java-based high performance distributed computing - an evolving virtual machine. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 308–317, August 1996.

[10] Geoffrey Fox and Wojtek Furmanski. Petaops and exaops: Supercomputing on the web. *IEEE Internet Computing*, pages 38–46, 1997.

[11] Efstratois Gallopoulos, Elias Houstis and John Rice. Computer as a Thinker/Doer: Problem Solving Environments for Computational Sciences. *IEEE Computational Science and Engineering*, pages 11–23, 1994.

[12] E. Houstis, J. Rice, S. Weerawarana, A. Catlin, P. Papachiou, K.-W. Wang and M. Gaitatzes. Parallel ELLPACK: A Problem-Solving Environment for PDE Based Applications on Multicomputer Platforms. *ACM Transactions on Mathematical Software*, (To Appear) 1998.

[13] Philip Isenhour. Sieve: A Java-Based Framework for Collaborative Component Composition. Master's thesis, Virginia Tech, Blacksburg, VA, 1998.

[14] Lori Leonardo. *Using Netscape Liveconnect*. QUE Education and Training, 1997.

[15] Robert Orfali, Dan Harkey and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley and Sons, New York, NY, 1996.

[16] Constantinos Phanouriou and Marc Abrams. Transforming Command-line Driven Systems to Web Applications. In *Proceedings of the Sixth International World Wide Web Conference*, April 1997.

[17] John Rice and Ronald Boisvert. From Scientific Software Libraries to Problem-Solving Environments. *IEEE Computational Science and Engineering*, pages 44–53, 1996.

[18] Dale Rogerson. *Inside COM*. Microsoft Press, 1997.

[19] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley and Sons, New York, NY, 1996.

[20] Craig Upson, Thomas Faulhaber, David Kamins, Davin Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The Application Visualization System: a Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, pages 30–42, July 1989.

[21] Borland International, Inc. Borland JBuilder Home Page. URL: http://www.borland.-com/jbuilder, 1998.

[22] Department of Computer Science, Virginia Tech. Research in Problem Solving Environments at Virginia Tech. URL: http://www.cs.vt.edu/ pse/, 1998.

[23] Department of Computer Sciences, Purdue University. Web//Ellpack: A Networked Computing Service on the World Wide Web. Technical Report CSD TR-96-011, 1996.

[24] Department of Computer Sciences, Purdue University. Problem Solving Environments. URL: http://www.cs.purdue.edu/research/cse/pses/, 1998.

[25] IBM Corporation. IBM Visual Age for Java. URL: http://www.software.ibm.com/-ad/vajava/, 1998.

[26] The Infospheres Research Group. The Infospheres Infrastructure User Guide. URL: http://www.infospheres.caltech.edu/, 1998.

[27] Khoral Research, Inc. What is khoros? URL: http://www.khoral.com/khoros/-whatis.html, 1998.

[28] Sun Microsystems, Inc. Java Core Reflection API and Specification. URL: http://www.-javasoft.com/products/jdk/1.1/docs/guide/reflection/spec/java-reflectionTOC.doc.-html, 1997.

[29] Sun Microsystems, Inc. Java Object Serialization Specification. URL: http://www.-javasoft.com/products/jdk/1.1/docs/guide/serialization/spec/serialTOC.doc.html, 1997.

[30] Sun Microsystems, Inc. Java AWT: Delegation Event Model. URL: http://www.-javasoft.com/products/jdk/1.1/docs/guide/awt/designspec/events.html, 1997.

[31] Sun Microsystems, Inc. Java Remote Method Invocation Specification. URL: http://-www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html, 1997.

[32] Sun Microsystems, Inc. Manifest Format. URL: http://www.javasoft.com/products/-jdk/1.1/docs/guide/jar/manifest.html, 1997.

[33] Sun Microsystems, Inc. The JavaBeans (tm) API Specification. URL: http://www.-javasoft.com/beans/docs/beans.101.pdf, 1997.

[34] Sun Microsystems, Inc. Java Development Kit 1.1.x. URL: http://www.javasoft.com/-products/jdk/1.1/, 1998.

[35] Sun Microsystems, Inc. Java Home Page. URL: http://www.javasoft.com/, 1998.

[36] Sun Microsystems, Inc. Java Workshop 2.0. URL: http://www.sun.com/workshop/-java, 1998.

[37] Sun Microsystems, Inc. The Beans Development Kit. URL: http://www.javasoft.com-/beans/software/bdk_download.html, 1998.

[38] Sun Microsystems, Inc. The JavaBeans (tm) Tutorial. URL: http://www.javasoft.-com/beans/docs/Tutorial-Nov97.pdf, 1998.

[39] Sun Microsystems, Inc. The JDBC Database Access API. URL: http://www.javasoft.-com/products/jdbc/, 1998.

VITA

Ashish Bimalkumar Shah was born in Ahmedabad, India on December 18, 1973. After finishing high school at the St. Xavier's Higher Secondary School, he began his undergraduate studies at L.D. College of Engineering, Ahmedabad, in August 1991. As an undergraduate he undertook a one year project at the Space Applications Center of the Indian Space Research Organization where he developed a mathematical computation and visualization tool for the Windows environment. He received his Bachelor of Engineering degree in Computer Engineering from Gujarat University in November 1995, and was awarded a Gold Medal for securing the top position in the graduating class.

He joined the Computer Science department at Virginia Tech for graduate studies in Spring 1996. While at Virginia Tech he has worked as a part-time software developer at the Agricultural and Natural Resources Information Systems (AGNIS) laboratory of the College of Agriculture and Life Sciences. At AGNIS he developed web-based systems for on-line interactive quizzing and on-line event publishing which are being used by several faculty members and academic departments at Virginia Tech.

He served as the President of the Indian Students Association at Virginia Tech for the 1997-98 term. He will be graduating with a Master of Science degree in Computer Science and Applications in May 1998 and moving to Redmond, WA for starting a full-time job with Microsoft Corporation. At Microsoft, he will be joining the Microsoft Exchange Server team as a software design engineer.