

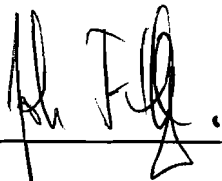
New Java Technologies and a Java-based Framework for Interactive Illustration Development

Jeffrey Evan Beall

**Department of Computer Science
Brown University**

**Submitted in partial fulfillment of the
requirements for the degree of Master of
Sciences in the Brown University
Department of Computer Science.**

August 1997

A handwritten signature in black ink, appearing to read 'John F. Hughes', is positioned above a horizontal line.

**Professor John F. Hughes
Advisor**

Table of Contents

1.0	Introduction	1
1.1	Document Overview	1
1.2	Notes to the Reader	2
2.0	Related Work	3
2.1	Overview of Commercial Web-based Technologies	3
2.2	Recent Java and Interactive Illustration Efforts at Brown	5
3.0	Java Technologies: Concepts and Discussion	7
3.1	Some General Language Features	7
3.2	Abstract Windowing Toolkit	9
3.3	JavaBeans	12
4.0	BeanStalk	14
4.1	Time Models	15
4.2	Canvas Framework	19
4.3	Lightweight Components	27
5.0	BeanStalk at Work	29
6.0	Future Work.....	35
6.1	Extended Media Support	35
6.2	JavaBeans Integration	36
6.3	Industry Developments and Their Impact	37
7.0	Conclusions	39
8.0	Acknowledgments	40
9.0	References	41
10.0	Appendix A: BeanStalk Version History	43

1.0 Introduction

For many years, researchers and educators have been investigating techniques for computer-based and computer-assisted learning. One such technique is known as the *interactive illustration*. Briefly stated, an interactive illustration is a 2D or 3D structured environment that pedagogically guides the user through a concept or set of concepts to foster exploratory learning. Their scope can range from a small responsive 2D diagram to a fully immersive and reactive 3D world. Until recently, interactive illustrations were only feasible on expensive workstations found at universities, precluding wide audiences such as high school students who could benefit from them. However, advances in consumer computing hardware and software technology are alleviating the problem, and interactive illustrations can now take advantage of commodity computing platforms [5].

One of the main thrusts of current interactive illustration research is to explore design issues in the context of the *World Wide Web* [26]. The Web, though still an emerging technology itself, already has conventions for graphic design, interaction, and navigation that must be followed in order to make effective Web-based illustrations. Another research issue is choosing the appropriate illustration development technology. The most prominent player in this realm is Java.

In two year's time, Java has gone from being a little toy on Web pages to becoming a powerful language and platform in its own right. The latest version of Java, the *JDK1.1*, now provides a full-featured set of tools for building applications on any scale. However, the standard Java libraries are not designed specifically for illustration development and can potentially be awkward to use. Also, certain higher-level design patterns that are useful in many different illustrations are not inherent in these Java libraries. A combination of tools and design patterns for interactive illustration development which build on Java's strengths would improve the development process by letting developers focus more on content instead of supporting technology. The *BeanStalk* component library and framework described in this document is such a solution.

1.1 Document Overview

This document briefly reviews the other competing Web-based technologies for

interactive illustration development as well as some recent academic Java-based projects at Brown University. Some of Java's features that are particularly relevant to interactive illustration development are then covered. Following that is a detailed discussion of the BeanStalk framework and component library. Finally, future work in BeanStalk and the future of Java as an interactive illustration platform are discussed.

1.2 Notes to the Reader

This document assumes a certain level of knowledge about Java, object-oriented programming, and user interface design. Readers should be familiar with terms such as *class*, *object*, *type*, *interface*, *inheritance*, *polymorphism*, *callback*, and *applet*, among others. One good source for learning about object-oriented techniques as applied to Java is *Java in a NutShell* by Flanagan [12].

The code throughout the document is denoted in this constant-spaced font. Classes, interfaces, and package names are denoted in this **bold** style. There are also some coding conventions used in BeanStalk and the sample code. All interfaces start with 'I' and all abstract classes start with 'A'.

2.0 Related Work

2.1 Overview of Commercial Web-based Technologies

A number of competing Web-based technologies have emerged within the past two years in addition to Java. Each has its strengths and weaknesses, and an appropriate niche in the marketplace. We will take a brief look at some of the most prominent technologies as they relate to interactive illustration development.

Shockwave

Shockwave is the Web viewer for Macromedia's *Director* multimedia production software [9]. *Director* started out as a tool for creating simple animations, but has grown over the years to become one of the most prominent multimedia development environments available today. Its biggest appeal is that it does not initially require programming knowledge to produce reasonably interesting material, although extensive scripting in Lingo, its proprietary scripting language, becomes a necessity for complex interaction and multimedia effects. Because of its initial simplicity and artist-centered user interface, *Director* has become the de facto standard for authoring CD-ROM titles and other multimedia applications.

Because of the emerging potential of the Web as a multimedia platform, Macromedia developed Shockwave as a plug-in for Web browsers which would be able to play specially processed, or "shocked," *Director* animations. This is an easy way for interactive multimedia content to be produced for the Web, although there are several important drawbacks. First, since it is a plug-in, each browser has to have the Shockwave plug-in installed before any Shockwave content can be viewed. A second problem with Shockwave's plug-in nature is that a separate plug-in has to be developed for each computer platform, which rules out viewing content on less consumer-relevant platforms such as UNIX. Another drawback is that Lingo is not a full-featured programming language and therefore does not easily support complex mathematical calculations or services such as networking. Even with all of its problems, Shockwave remains a popular platform for Web-based multimedia and is well suited to simple interactive illustrations.

Liquid Motion Pro

Liquid Motion Pro was developed by a small company called DimensionX as a way of creating Java-based multimedia content. The software would allow visual authoring of content in a style similar to Director, and would generate a Java applet as the final product. This is *Liquid Motion Pro*'s large advantage over Shockwave, in that its content could be viewed in any Java-enabled Web browser. The one drawback is that each applet generated by *Liquid Motion Pro* requires a proprietary Java-based runtime engine in order to run, increasing applet download times. This situation may soon be fixed because Microsoft recently acquired DimensionX and will be incorporating *Liquid Motion Pro* into its *DirectX* set of media libraries [25]. While Windows users will benefit, this move does not alleviate the problem for the rest of the Java platforms.

DirectAnimation

Microsoft has other multimedia content efforts underway besides *Liquid Motion Pro*. In particular, *Internet Explorer 4 (IE4)* will incorporate *DirectAnimation*, a Java- and scripting-based platform for 2D and 3D interactive content [10]. *DirectAnimation* grew out of Microsoft's failed *ActiveVRML* technology, which in turn was indirectly based on Sun's *T-Bag* interactive graphics system [11], since the same development team produced all three systems. *DirectAnimation* frames the content authoring process in terms of *time-based reactive behaviors*, where simple behaviors such as a numerical value changing over time can be composited into much larger and more complex behaviors. Since the notion of time is integral to the system, behaviors are independent of the system on which they execute. Time in *DirectAnimation* will pass at the same rate on a slow machine without graphics acceleration and on a top-of-the-line machine with a 3D accelerator board. Another key concept is all forms of media are equally important and can be easily combined. For example, a button behavior could be rendered into an image behavior, which in turn is texture-mapped onto a geometric object with motion behavior. When the image of the button on the geometric object is clicked on, the original button's action behavior would be triggered, just as though none of the other composited behaviors existed. Sound is also a type of behavior, which can be attached to geometric objects and rendered in the same way the 3D scene is rendered into a 2D image. This kind of unified media integration is extremely powerful and allows for many interesting interactive illus-

tration possibilities.

The main drawbacks to developing content in DirectAnimation include its often-cumbersome programming model, proprietary Windows-only software and hardware technology, and a lack of an authoring environment. Even though it can be programmed in Java, DirectAnimation is deeply rooted in Windows, making it less appealing for interactive illustration development since it is not cross-platform.

VRML

The *Virtual Reality Modeling Language (VRML)* started out in 1995 as a standard 3D static scene description file with the ability to attach document hyperlinks to geometry. The latest incarnation now supports behaviors such as path-based animations, object picking, and 3D sound [28]. VRML is gaining momentum as the standard for 3D on the Web, but as an interactive illustration development technology it is lacking. VRML offers very limited 2D support and defining behaviors is a complicated process. Either VRMLScript or JavaScript can be used inside the VRML files, but neither is especially powerful. Java code can be used with VRML as well, although like VRMLScript and JavaScript it is not well integrated into the rest of the VRML scene description, appearing as though it was tacked on to the specification. In general, VRML is much better suited for 3D scene description than complex interaction and therefore is not a good general-purpose platform for illustration development.

2.2 Recent Java and Interactive Illustration Efforts at Brown

A few years ago, the main platform for illustration development was the *Trim* system, an interactive 3D graphics environment developed in the Brown Computer Graphics Group. Trim relied on the 3D graphics hardware in the advanced workstations in the Computer Science department, making illustrations written with it very unportable. Those developed at Brown could only run here or at other universities with similar workstations, eliminating a vast potential audience for our work. When Java became available around the summer of 1995, as an experiment we ported some interactive illustrations of color perception that had originally been developed in Trim and had later been ported to C++. The experiment was quite successful because we had produced interactive illustrations that not only performed well in Java but were now cross-platform and available to a wide audience. Our experiences

with these color perception illustrations became the subject for a paper [5]. One of the conclusions drawn from this experience was that developing interactive illustrations in a prototyping environment such as Trim and then porting to Java was a better approach than developing in straight Java. While that may have been a valid conclusion at the time, recent developments in Java have made the proprietary prototyping environment stage unnecessary. Technologies such as *JavaBeans* [17] allow for rapid prototyping within the Java environment. JavaBeans and other related technologies will be discussed in the next chapter.

Another interactive illustration research development at Brown has been the formation of a project group devoted to exploring the interactive illustration design process. Some guidelines and preliminary results are discussed in detail in honors theses resulting from project group experiences [6][26]. The essential ideas from these results are that graphic design of interactive illustrations is crucial to their overall success and that a prototyping cycle with peer review greatly benefits the design process. As was mentioned earlier, new Java technologies combined with tools geared for illustrations have enormous potential to improve the illustration production process.

Other innovative work involving Java has recently been done in Brown Computer Science. A graphics and user interface toolkit, called *Graphics Package (GP)*, has been developed over several years as the core technology for Brown's introductory Computer Science course. GP was originally developed for object-oriented Pascal [7]. It has been ported to C++ and was initially ported to Java for the color perception illustrations. The latest version is a total rewrite and is designed to completely encapsulate Java's *Abstract Windowing Toolkit (AWT)* from the students. While this approach is appropriate for a course designed to gently guide students through the basics of object-oriented programming, GP sacrifices — or at least obscures — the raw power and convenience of AWT in the process. As we will see in BeanStalk, structures can be built on top of AWT that not only make it easier to use but also make AWT and other Java technologies readily available when necessary.

3.0 Java Technologies: Concepts and Discussion

Java offers many of the things people expect in a modern programming language: objects, interfaces, run-time type information, automatic memory management, security features, threading, and exceptions — to name a few. The “core” libraries include graphics and user interface services, networking, utility data structures, and applet services. The JDK 1.1 incorporates new technology such as JavaBeans, inner classes, database connectivity, and object reflection and introspection. This palette of features makes Java a powerful and full-featured application development language, and is also well suited for interactive illustration development. We will briefly examine some of these Java features to set the stage for the description of the BeanStalk interactive illustration authoring toolkit.

3.1 Some General Language Features

Interfaces

Interfaces are one of Java’s most powerful features. An object’s type is no longer tied to its class, unlike in C++. One particularly interesting application of Java’s interfaces is the simulation of multiple inheritance. In Java, classes must inherit from exactly one class (which is the class **Object** when nothing else is specified), unlike C++. However, classes can implement arbitrarily many interfaces in addition to extending one class and it is this feature that allows for multiple inheritance to be simulated. Let’s illustrate this with an example:

```
interface IBar {
    void doBar();
}

class Bar implements IBar {
    public void doBar() {
        // some really exciting code goes here
    }
}

class Foo {
    public void doFoo() {
        // more exciting code
    }
}
```

```

    }

    class FooBar extends Foo implements IBar {
        void doBar() {
            bar_.doBar();
        }

        private IBar bar_ = new Bar();
    }

```

In this example, we have the interface **IBar** with one method and the class **Bar** which implements it. We also have the class **Foo** which does not implement anything but does define a method of its own. However, suppose we wanted to combine the functionality of **Foo** and **Bar** into one class. We cannot do this with class inheritance in Java, but interfaces provide a way around the problem. Since **IBar** is an interface, anything can implement it, including our new class **FooBar** which also extends **Foo**. To simulate the multiple inheritance of **Foo** and **Bar**, **FooBar** simply delegates the methods of **IBar** to an instance of **Bar**. This is essentially what is happening in C++'s multiple inheritance, except this approach requires explicit delegation instead of having the compiler automatically generate it.

Note that the need for this style of object composition has not gone unnoticed by other developers or Sun itself. The *Object Aggregation and Delegation Model* draft specification is currently available at the JavaSoft website which outlines the proposed addition to the Java APIs in an attempt to standardize this technique [1].

Threads

Multi-threading support has been a part of Java from its beginning and is incorporated at all levels. Threads and synchronization are language features, not just library add-ons as in C++. This makes developing threaded applications easy and straightforward, although potential pitfalls such as deadlocks have not been removed. Threads are particularly useful for lower-level constructs like asynchronous and timed callbacks, and higher-level constructs like animation.

Inner Classes

Inner classes are a new feature in the JDK 1.1. Classes can now be nested inside other classes, with the inner class given access to the outer class's members. Inner classes can hide the fact that an outer class requires an additional class to imple-

ment its necessary behavior. This fact is particularly useful in conjunction with the new AWT event model, described below. For further reading on inner classes and the related *anonymous class* construct, refer to the inner classes specification [14].

3.2 Abstract Windowing Toolkit

The Abstract Windowing Toolkit is a standard set of Java classes for graphical user interfaces (*GUI*), input devices, and graphical output. Many aspects of the initial version of AWT were confusing and inconsistent. For example, the initial version's event handling mechanisms were tightly coupled with AWT's inheritance hierarchy, forcing application code and user interface code to mix together into one confusing mess. (This led to the nickname "Awkward Windowing Toolkit.") Thanks to extensive developer feedback, the JDK 1.1 AWT now has a much improved event model which is designed to work with JavaBeans, another new Java technology in the 1.1. Some of the most prominent AWT features are discussed below.

Component Model

The majority of AWT is a set of classes that provide the standard GUI components that developers and users have come to expect in modern applications. They include buttons, sliders, windows, menus, checkboxes, lists, and so on. AWT supports two notions of hierarchy: *class inheritance* and *containment inheritance*. At the top of the AWT class inheritance is the **Component** class, from which all other components inherit. One of these subclasses is **Container**, which has the added functionality that it can contain other components. This is where the containment inheritance comes in. In the case of an applet, any given component on the screen is ultimately contained by the applet's container. Containers themselves can contain other containers, allowing for deep nesting of components. Properties such as background colors can be passed down from a parent component to a child component as well in this containment inheritance. Figure 3.1 shows the how object relationships differ between class inheritance and containment inheritance.

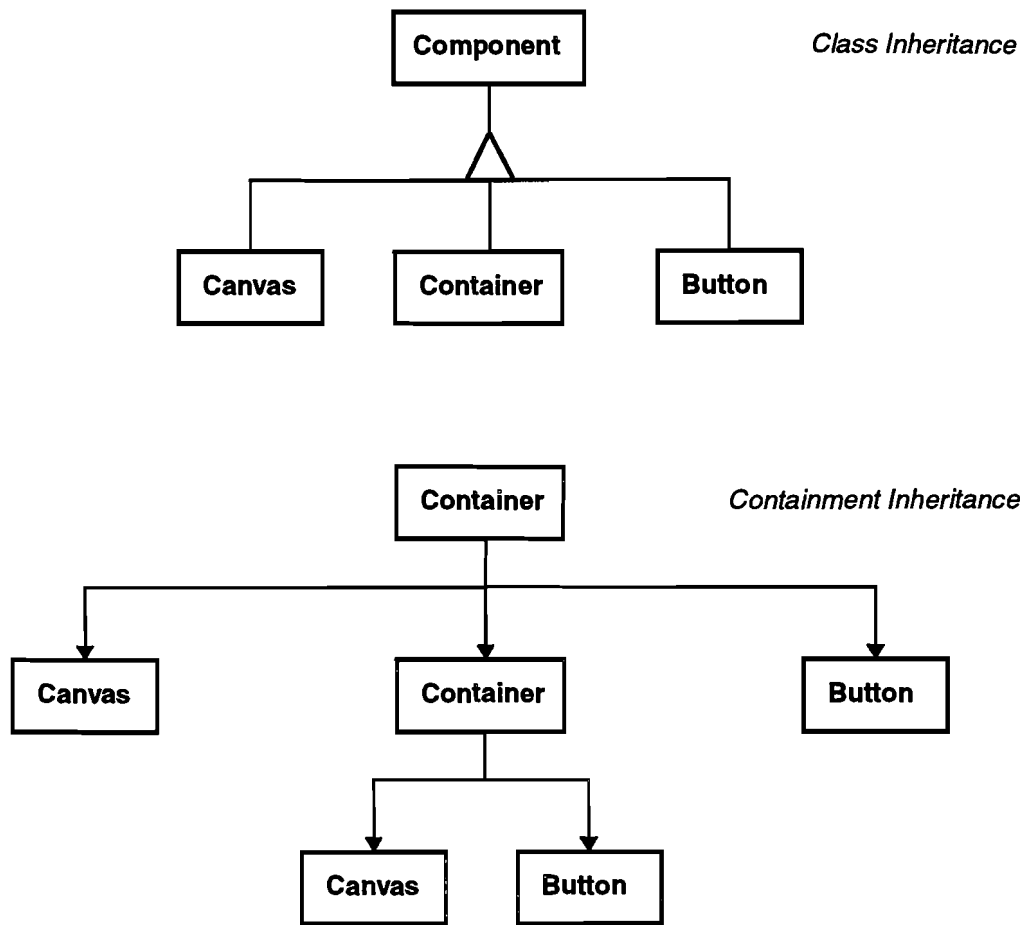


Figure 3.1: Class versus containment inheritance. In AWT, Canvas, Container, and Button inherit from Component. At runtime, Container objects can hold any Component object, including other containers.

Another aspect of the AWT component model is the mechanism behind their on-screen representation. Being inherently visual, components are required to know how to draw themselves. Therefore, each component has a “paint” method that gets called when the component needs to be drawn to the screen (or offscreen buffer). The caller of the paint method passes in a **Graphics** object, which is an abstract representation of the component’s operating system graphics context. If a developer wants a general-purpose drawing area, the paint method on the **Canvas** should be overridden. The creation of new kinds of components can also be done and is described below.

Lightweight Components

In the old AWT, every component had a *peer*, which was a platform-specific implementation of the given component. For example, an AWT button under UNIX would have a Motif button peer, an AWT button under Windows95 would have a Win32 button peer, etc. Peers were created behind the scenes by AWT and were not intended to be manipulated directly. The peer architecture is why AWT is called the *Abstract Windowing Toolkit* — components are created and manipulated in the abstract, while the peers actually do the dirty work. This is also why AWT user interfaces would look different on different platforms. Each platform would have its own peer implementation, resulting in a different look-and-feel in the best case to unacceptably different functionality in the worst case. Another problem was that each peer had some amount of platform-specific resources devoted to it, which could potentially bog down the host operating system or cause other quirks. These sorts of problems, combined with the event handling mess in the JDK 1.0, caused many developers to complain.

The peer architecture still exists in the JDK 1.1 but is hidden much further from view. Peerless components, also known as *lightweight components* have been designed to replace it [4]. Lightweight components are a new addition to the JDK which behave just like their peered counterparts, except they do not have any operating system resources associated with them. Instead of having their own graphics contexts to draw in, they use the graphics context of the top-most peered, or *heavy-weight*, container in the containment hierarchy. The main advantage behind lightweight components is that they can have the same look-and-feel across platforms since they are not explicitly tied to any operating system resources. As we will see in BeanStalk, the user interfaces of interactive illustrations that use lightweight components can have much better graphic design support than if regular components were used.

Source and Listener Event Model

In addition to the component model, the JDK 1.1 overhauled the often-criticized event handling model. The following code example demonstrates the old style:

```
// --- the old way --- //
class MyButton extends Button {
    public boolean action(Event event, Object data) {
```

```
        // handle button event here and return true
    }
}
```

Having to create a subclass of **Button** for every new button behavior could become unwieldy for even moderately sized applications. While this approach was not strictly necessary, the alternate was not much better. Instead of handling the event inside the source as **MyButton** does, a subclassed component higher in the containment hierarchy could listen for events from any of its children. The problem in this case was that this parent component would need to handle the event per child and per event type, creating a huge “if-then-else” mess. This model clearly needed reworking.

The new approach decouples event handling from the event sources. AWT components such as buttons are no longer in charge of their event handling and neither are their parent containers. Event sources now notify any object that registers itself as a listener of the specific event type that the source generates. The sources know that a given object can listen to it because the object must implement an interface that identifies itself as a listener. Sources can support event multicasting, so that a single event can be delivered to arbitrarily many listeners.

For further information about the new AWT event model, refer to the documents on AWT at the Java website [3].

3.3 JavaBeans

Visual authoring environments for user interface development and prototyping have become prevalent over the past several years. Microsoft’s VisualBasic [27] has allowed many non-programmers to assemble “prefabricated components” into custom applications without much effort. VisualBasic uses the *Component Object Model (COM)*, Microsoft’s technique for providing a uniform mechanism for objects to communicate. (COM grew out of Microsoft’s *Object Linking and Embedding (OLE)* technology. Microsoft’s *ActiveX* technology is simply a wrapper around the *Distributed Component Object Model (DCOM)*.) Since VisualBasic’s components are all COM objects, there is a standard way for them to interact with each other and therefore the components can be manipulated within a visual authoring environment. The main drawbacks to COM are that it is deeply rooted in the Win32 platform and the objects themselves are stored in binary form, making them unportable across plat-

forms. Another big problem with COM is that the actual code that the visual authoring environments generate to make COM objects work together is so complicated that it is almost unreadable by human beings. Even with these problems, COM is arguably the most successful component object model to date because of the tremendous support it gets from Microsoft.

Java initially did not have a component model (although its object model is surprisingly similar to COM's). Based on extensive feedback from companies producing commercial Java development environments and the Java community at large, Sun created the *JavaBeans* specification [17]. The JavaBeans component model is very similar to COM, except that it is based entirely in Java and is therefore cross-platform. JavaBeans is the official component model for Java and has received widespread support from both development environment developers and the community at large. Many companies such as Lotus, IBM, and Netscape have pledged their full support for JavaBeans and will be releasing JavaBeans components in the near future.

Beans, as JavaBean components are also known, can be edited visually in Bean-compliant visual authoring environments and support the new AWT event model. (All AWT components are now Beans too.) To actually create a Bean from scratch, the specification defines a number of naming conventions (the specification calls them "design patterns") to which the Bean should adhere. The Bean can then be analyzed by a Bean-compliant environment to determine its *properties* and the kinds of events it supports. Properties could be colors, labels, dimensions, positions, etc., all of which can be edited through the methods named with the standard Bean naming conventions.

JavaBeans can potentially offer many useful features to interactive illustration developers. The visual authoring of GUI components could allow for rapid prototyping of illustration user interfaces. Also, prefabricated interactive illustration components could be developed and reused from one project to the next. Since a Bean does not necessarily have to be visual, these illustration Beans could be math engines, data structures, or other useful constructs.

4.0 BeanStalk

While Java has reached a level of maturity that provides sufficient tools for creating compelling interactive illustrations, using those tools can be a complex process. It is counter-productive for illustration developers to reinvent the wheel with each new project. What is needed is a framework for building interactive illustrations — something that takes care of lower-level user interface and object design issues and allows the developer to concentrate on the illustration's content. This was the major motivation behind the development of BeanStalk. BeanStalk is designed to take advantage of new Java features such as lightweight components, the JavaBeans component model, and the new event model. Another design goal was to build a lightweight layer on top of AWT so that developers can avoid AWT if they choose too, but can also have access to all of AWT's features when needed. A third goal was to create a set of JavaBeans components for illustrations. While this last goal was not met largely because of delays in commercial Beans development environments, many classes were created that adhere to the Beans specification and therefore should be easily turned into Beans at some point in the future. Finally, the overall goal was to provide the means of creating interactive illustrations without too much effort that were comparable to Director-style projects in their look and feel. For the large part, BeanStalk achieved these goals.

We will begin our overview of BeanStalk with a simple applet. It creates a shape in the applet that can be dragged around with the mouse.

```
package examples;

import stalk.canvas.*;
import stalk.canvas.geom.*;
import stalk.time.*;

import java.applet.*;
import java.awt.*;
import java.util.*;

public class SimpleApplet extends Applet {

    public void init() {

        // creates and sets up the BeanStalk canvas
        StalkCanvas canvas = new StalkCanvas();
```



```

        canvas.setSize(getSize());
        canvas.setDoubleBuffered(true);

        // adds the canvas to the applet
        add(canvas);

        // creates a canvas animator and adds the canvas to it
        CanvasAnimator animator = new CanvasAnimator();
        animator.addCanvas(canvas);
        animator.start();

        // creates and sets up a rectangle
        StalkRectangle rect = new StalkRectangle();
        rect.setFilled(true);
        rect.setSize(25, 25);
        rect.setLocation(50, 50);

        // adds the rectangle to the canvas for drawing and selection
        canvas.addDrawable(rect);
        canvas.addSelectable(rect);

        // creates the selection adapter and adds it to the canvas
        Vector selectables = canvas.getSelectables();
        ASelectAdapter adapter = new
                                DraggableSelectAdapter(selectables);
        canvas.addSelectAdapter(adapter);
    }
}

```

The four basic elements to this example are the canvas, the canvas animator, the shape, and the selection adapter. A shape is put in a canvas so that it can be drawn, which in turn is constantly repainted by the animator. The selection adapter works with the canvas and the shape to produce dragging behavior. While this applet does nothing particularly useful as an interactive illustration, it shows some of the basic relationships among BeanStalk objects.

The next several sections of the document describe the design and structure of BeanStalk in detail.

4.1 Time Models

Time is a prominent feature of BeanStalk and manifests itself in two main ways: the *animator* and the *time context*. The classes and interfaces discussed in this section are in the **stalk.time** package.

Animators

The **Animator** class provides a basic timed-callback mechanism. The class itself is abstract and must be subclassed to provide the callback. This is done by overriding the `sample()` method on **Animator**. Animators are useful for when a specific behavior has to happen repeatedly at a given time interval. Examples of this would be evaluating the state of a dynamic simulation or refreshing a drawing area.

Here are the methods of the animator:

- `getSampleRate()`
Returns the sampling rate of the animator in milliseconds.
- `isRunning()`
Returns the running status of the animator.
- `sample()`
This must be overridden to supply the animator's behavior.
- `setSampleRate(int)`
Sets the sampling rate of the animator in milliseconds.
- `start()`
Starts the animator.
- `stop()`
Stops the animator.

Time Contexts

The abstract notion of a time context takes a different approach than the animator. Whereas the animator encapsulates time as a repeating behavior, the time context encapsulates time as a state which other objects can query. Time, as a time context sees it, is a linearly increasing function that begins at zero and goes to infinity. There is no support for time before the beginning of time (e.g., any value of time less than zero). Time increases at one unit per second unless otherwise altered.

In practice, the **TimeContext** class bases its notion of time on the system clock, and cannot have an infinite time value. The class supports the following behaviors:

- `getRate()`

Returns the rate of time passing.

- `getTime()`
Returns the last sampled time value.
- `moveTimeTo(float)`
Moves current time to the specified point in time.
- `sampleTime()`
Samples the current time.
- `setRate(float)`
Sets the rate of time passing.

The difference between `getTime` and `sampleTime` is subtle but important. To get the most recently sampled value of time out of the time context, the `getTime` method is called. To produce the new “current” time, the `sampleTime` method is used. This way the current time can be sampled once and then accessed as many times as necessary until the next time value is sampled. (We will see in the next section an example of where this comes into play.) If their separate behaviors were combined into one method, the caller of that method might not be able to get the same time value twice out of the time context, which would be an issue when the time context is passed around to many objects that should all get the same value of time out of the context.

Certain situations exist where other objects need to know what the current time is but should not be given arbitrary control over a time context. For these situations, a time context can be seen simply as the **ITimeContext** interface, which only supports the ability to query the current time and rate of the context. If other objects can only access the time context through this interface, then these objects are restricted from attempting to reset time to zero or performing any other unwanted modifications. The **TimeContext** class implements **ITimeContext** so it can be used when an **ITimeContext** interface is called for. (This is also another reason to have both `getTime` and `sampleTime` on the time context.) An additional advantage of having this separate interface is that classes other than **TimeContext** could conceivably implement the interface to allow time inquiries.

Using Time Contexts

Time contexts are useful in situations where a behavior implicitly performs over time, without regard the actual value of time or how often that value changes. Let's look at the example of a time-based quadratic function using a time context.

```
public float currentInterpolationValue(ITimeContext timeCtx) {  
    float currentTime = timeCtx.getTime();  
    return (currentTime * currentTime);  
}
```

If time increases at a linear rate (which it is inclined to do), the value of the function will increase quadratically. Sampled at one second the value will be one, at two seconds the value will be at four, and so on.

Now, let's alter our previous function so that it increases quadratically for the first second, and then increases linearly from then on.

```
public float currentInterpolationValue(ITimeContext timeCtx) {  
    float currentTime = timeCtx.getTime();  
    if (currentTime <= 1.0f) {  
        return (currentTime * currentTime);  
    } else {  
        return currentTime;  
    }  
}
```

Our new function now alters its return value based on the current time. Let's look at another example that bases its behavior on the current value of time.

```
public void doSomething(ITimeContext timeCtx) {  
    if (timeCtx.getTime() <= 5.0f) {  
        doSomethingBefore();  
    } else {  
        doSomethingAfter();  
    }  
}  
  
private void doSomethingBefore() { // some code... }  
private void doSomethingAfter() { // some different code... }
```

In this example, we are no longer returning a value as a function of time. Instead, the time context is used to determine what other function should be called based on the current time.

As we can see from the examples in this section, time contexts can be used whenever behavior is based on time. They're appropriate when we don't know a priori what the time value will be or how often our methods that implement behavior will be called. When the desired behavior requires the same thing to happen at very specific time intervals, such as advancing through frames of a slideshow, the animator is probably more appropriate.

Time contexts are mainly used in conjunction with BeanStalk canvases, where they provide time services to items in the canvas. (More on this later.) In general, a developer probably will not need to create one explicitly since a canvas creates one when the canvas itself is constructed. When a time context is used with a canvas, it does not need to be explicitly sampled either, so that complication is removed.

Utility Behaviors Using the Time Context

BeanStalk provides two simple time-varying behaviors in the **stalk.time.tv** package which use the time context behavior model. The first is **Envelope**, which is based on the notion of an envelope generator. (An envelope generator in this context is a function that interpolates between a series of values. The inspiration for this class came from envelope generators used in analog electronic music synthesis.) The envelope is constructed by specifying various times in the envelope and the values associated with those times. The value of the envelope is then specified by the following method:

- `getValue(ITimeContext)`
This returns the value of the envelope based on the time context.

As we can see, this is very much like the previous examples of time-based functions. **ColorEnvelope**, the other class in the package, extends the idea of the envelope to produce a new AWT color instead of just a value.

While BeanStalk only has these two time-varying behavior classes, the range of possibilities for other behaviors is enormous. For example, a whole family of time-varying interpolator functions would be very useful.

4.2 Canvas Framework

Drawing items onto the screen and allowing the user to interact with them is the essence of interactive illustrations, so much of BeanStalk is devoted to supporting

these activities. This is accomplished through a framework that handles many of the low level details of AWT while still allowing a wide range of functionality. There are three main elements to the canvas framework: the canvases themselves, items that are put into the canvas, and classes that let the user interact with the canvases.

The BeanStalk Canvases

BeanStalk offers two versions of its framework-enabled canvas: **StalkCanvas** and **LightWeightCanvas**. They both support the same core functionality and only differ in their parent class. **StalkCanvas** inherits from **java.awt.Canvas** whereas **LightWeightCanvas** inherits from **java.awt.Component**. Both exist because painting a **StalkCanvas** is more time-efficient since it has its own graphics resource, while the **LightWeightCanvas** can support a see-through background. The core functionality that these two canvases share comes from the **IStalkCanvas** interface, which both implement:

- `addDrawable(IDrawable)`
Adds a drawable item to the canvas. The item's `draw()` method will be called every time the canvas is repainted.
- `addSelectable(ISelectable)`
Adds a selectable item to the canvas.
- `addSelectAdapter(ASelectAdapter)`
Adds a select adapter to the canvas.
- `getDrawables()`
Returns the vector containing the canvas' drawable objects.
- `getSelectables()`
Returns the vector containing the canvas' selectable objects.
- `getTimeContext()`
Returns the canvas' time context.
- `removeDrawable(IDrawable)`
Removes a drawable item from the canvas.
- `removeSelectable(IDrawable)`
Removes a selectable item from the canvas.
- `removeSelectAdapter(ASelectAdapter)`
Removes a select adapter from the canvas.

- `repaint()`
Repaints the contents of the canvas.
- `setTimeContext(TimeContext)`
This sets the time context of the canvas, defining the canvas' notion of time passing.

Using the simulated multiple inheritance through delegation, both canvases contain a **StalkCanvasCore** class, which itself implements **IStalkCanvas**, and delegate to it when necessary. In addition, both classes have all of the methods of **java.awt.Component** since both ultimately inherit from it. While each also has some specific behavior such as double buffering, we will concentrate on the shared behavior throughout this section.

Canvases have a default time context when they are created, but can use any time context, allowing contexts to be shared between canvases.

The real added value of a BeanStalk canvas over a standard AWT canvas is its ability to manage the drawing and selection of graphical items. These items can be anything as long as they implement certain interfaces (described below). Items are added and removed through add/remove methods on the canvas. The entire set of drawable or selectable items can be returned at once in the form of a Java **Vector** using the `getDrawables` and `getSelectables` methods, respectively. This vector is the same vector that the canvas stores internally, providing direct access how the canvas stores the items. While this is a potentially dangerous violation of encapsulation, it allows the items to be easily manipulated through the vector's methods.

The ordering of items in these vectors is important to how they are drawn and selected. The canvas draws the items in ascending order, so the last item will be the last thing drawn into the canvas. Selection also occurs in ascending order. We will see further in the document how drawing and selection work.

Repainting the Canvas

The contents of the canvas can be painted in three ways. The most straightforward way is to simply call `repaint` on the canvas. This will draw all of the items that the canvas currently knows about. However, the time context will not be sampled, so the canvas won't show the effects of time moving forward.

A second way is to generate a **CanvasUpdateEvent** and pass it into the `updateCanvas` method on the canvas, with time being resampled in this case. This

might seem like an awkward approach, but it is designed for the source/listener event model. The canvases implement the **CanvasUpdateListener** interface so that they could be added to sources that fire the canvas update events. Currently, there are no classes in BeanStalk that are canvas update sources. Both the event and listener classes are in the **stalk.event** package.

The third way to repaint a canvas is to create a **CanvasAnimator** and add the canvas to it. The canvas animator inherits from **Animator**, and overrides the sample method to resample canvases' times contexts and repaint them. Arbitrarily many canvases can be added to the same animator, ensuring that they all get repainted synchronously.

Items in the Canvas

As was mentioned earlier, the main purpose of the BeanStalk canvases is to manage the drawing and selection of graphical items. For an item to be able to work with a canvas, it must implement some, if not all, of the following interfaces: **IDrawable**, **ISelectable**, **IProbeable**, **ISelectNotifiable**, **IProbeNotifiable**, **IPositionable**, and **ISizeable** (all of which are defined in the **stalk.canvas** package). These interfaces are described in detail below.

The IDrawable Interface

This is the most basic of the canvas interfaces. Its purpose is to provide a standard way for objects to be drawn inside a BeanStalk canvas. As far as the canvas is concerned, the item can draw itself however it chooses.

- `draw(Graphics, ITimeContext)`
The method called when the implementor needs to draw itself.

This is the sole method in the interface. The method takes a graphics context to draw into, and an interface to a time context so that time information can be used if needed. For example, if the item wanted to use a color envelope to determine its color, the draw method could ask the envelope for the color based on the time context and then set the graphics context's foreground color appropriately. Having draw take a **Graphics** object means that the implementor can use any of its extensive graphics functionality.

The draw method also provides an example of a situation where accessing some-

thing through an interface versus accessing the object directly is advantageous. The implementor of this method should not necessarily be able to change the current time of the time context object, so the **ITimeContext** interface does not allow for that.

Items are required to implement this interface in order to be drawn by a BeanStalk canvas.

The ISelectable Interface

In addition to drawing items, a BeanStalk canvas manages the items that can be selected by the user. These items must implement the **ISelectable** interface, which supports the following methods:

- `intersect(MouseEvent)`
This performs an intersection test using the information supplied in the mouse event object.
- `isSelected()`
This method returns the selection state of the object.
- `setSelected(boolean)`
This sets the selection state of the object.

Item intersection is based on an AWT mouse event which contains the mouse's location in the canvas. An item implementing this interface must know what its intersection boundaries are in order to return whether or not it was intersected.

The selection state of an item is independent of its intersection status, although we will see later how the two are used by BeanStalk for selection. Again, the implementor is responsible for defining what it means to be selected.

The IProbeable Interface

There are occasions when an item needs to demonstrate its ability to be selectable without actually being selected. An example of this would be a draggable shape in a canvas. It might not be immediately clear that the user could select this shape to drag it, but having the shape change color when the mouse rolls over it would provide a good indication that the shape can do *something*. Probing an item provides these means. The **IProbeable** interface provides the following methods to support probing:

- `isProbed()`
Returns the item's probed status.
- `setProbed(boolean)`
Sets the item's probed status.

Items are not required to implement this interface in order to be used with a BeanStalk canvas.

The ISelectNotifiable Interface

Other objects would potentially like to know when an item in the canvas becomes selected or deselected. The **ISelectNotifiable** interface provides a standard way for listener objects to register themselves with selectable canvas items in order to be notified of selection status changes. The methods on the interface are as follows:

- `addSelectListener(SelectListener)`
Adds a listener to the select notifier.
- `notifySelect(boolean, int)`
Notifies all listeners of the selection status of the item.
- `removeSelectListener(SelectListener)`
Removes a listener to the select notifier.

The `notifySelect` method takes a selection status and any modifier keys that were relevant to the selection. The **SelectListener** and associated **SelectEvent** classes are defined in the **stalk.event** package. Since the desired notification behavior is both straightforward and invariant, implementors of this interface should not need to recreate this behavior each time. Because of this, BeanStalk provides the **SelectNotifier** class which implements **ISelectNotifiable**. Items that would like this notification behavior can simply implement **ISelectNotifiable** and delegate to an instance of **SelectNotifier**. Once again, the simulation of multiple inheritance through interfaces and delegation proves its usefulness.

Objects are not required to implement this interface in order to be used with a BeanStalk canvas.

The IProbeNotifiable Interface

The **IProbeNotifiable** interface is the probing equivalent to selection notification. It provides the following methods:

- `addProbeListener(ProbeListener)`
Adds a listener to the probe notifier.
- `notifyProbing(boolean)`
Notifies all listeners of the probe status of the item.
- `removeProbeListener(ProbeListener)`
Removes a listener to the probe notifier.

The main difference between the two interfaces is that modifiers are not used as part of probe notification. The **ProbeListener**, **ProbeEvent**, and **ProbeNotifier** perform the same functions as their selection equivalents.

Objects are not required to implement this interface in order to be used with a BeanStalk canvas.

The IPositionable Interface

This interface provides a standard way for items to be positioned inside a canvas. The methods on **IPositionable** are as follows:

- `getLocation()`
Returns the item's location.
- `setLocation(int, int)`
Another method to set the graphical item's position.
- `setLocation(Point)`
Sets the graphical item's position.

The method names and parameters match the equivalent methods on the AWT **Component** class for continuity.

Objects are not required to implement this interface in order to be used with a BeanStalk canvas.

The ISizeable Interface

This interface provides a standard way for item's dimensions to be set. The meth-

ods on **ISizeable** are as follows:

- `getSize()`
Returns the item's dimensions.
- `setSize(Dimension)`
Used to set the graphical item's dimensions.
- `setSize(int, int)`
Another method to set the graphical item's dimensions.

Like the **IPositionable** interface, the method names and parameters of this interface match the equivalent methods on the AWT **Component** class.

Objects are not required to implement this interface in order to be used with a BeanStalk canvas.

Building Objects with the BeanStalk Canvas Interfaces

The canvas interfaces described above offer broad constraints for their implementors. If all an object requires is the ability to draw itself, then it simply implements the **IDrawable** interface. On the other hand, if an object requires positioning, sizing, selection, and selection notification on top of drawing, then all the interfaces should be implemented. Since items like these are needed frequently, BeanStalk provides the **AShape** class which implements all of the above behaviors except drawing. The class must be subclassed in order to be drawn and intersected appropriately. **StalkRectangle**, **StalkOval**, **StalkCanvasImage**, **SelectVec2** are BeanStalk utility classes that all inherit from **AShape**. These classes are in the **stalk.canvas.geom** package.

The **IDrawable** and **ISelectable** interfaces are well-suited for use in the Composite design pattern [13]. For example, one object can be drawable and contain other objects that are also drawable. When the draw method of the first object is called, it can call draw on all of its drawables. Drawable and selectable objects can be arbitrarily nested using the Composite pattern, allowing for complex scene graphs in a BeanStalk canvas.

Selection Adapters

The **ISelectable** interface allows an object to be selected, but does not specify how that actually happens. This is advantageous because selectable items should

not need to know every possible way they can be selected and deselected. If they did, then each time a new way of selecting items was created every class that implemented the interface would have to augment its implementation accordingly. (The same holds true for the **IProbeable** interface.) Clearly that is an unacceptable approach.

BeanStalk handles this situation by separating the selection process from the semantics of being selected. The semantics are left to the **ISelectable** implementation, whereas *selection adapters* handle the selection process. Selection adapters are designed around the new AWT event handling model. All AWT components can register mouse and mouse motion listeners. The components notify listeners of such events as the mouse moving over a component, a mouse button being pressed, the mouse being dragged while the button is down, etc. Various methods on the AWT **MouseListener** and **MouseMotionListener** interfaces corresponding to the specific mouse action are called with an instance of **MouseEvent**. Where the mouse is and what it is doing are the building blocks of selection adapters.

ASelectAdapter is an abstract class that implements both **MouseListener** and **MouseMotionListener**. By overriding specific methods from the AWT listeners, the style of selection of selectable objects can be created. Select adapters can use the fact that a selectable object could possibly also be probeable, positionable, and sizeable. The adapters can test to see if items implement any of these interfaces by using the `instanceof` Java operator, and can modify their selection behavior accordingly.

BeanStalk provides three kinds of select adapters which all inherit from **ASelectAdapter**. **ToggleSelectAdapter** will toggle an item's selection status using mouse clicks. **RolloverSelectAdapter** will select an item that is under the mouse. **DraggableSelectAdapter** is the most complex of the adapters. It probes the selectable items when the mouse rolls over them. When the mouse button is pressed down, the items are deprobed and selected. Moving the mouse while the button is down repositions the selected items. Finally, releasing the mouse button deselects the items. To take advantage of the full capabilities of **DraggableSelectAdapter**, an object must implement **ISelectable**, **IProbeable**, and **IPositionable**.

4.3 Lightweight Components

BeanStalk provides a number of standard GUI lightweight components in the

stalk package. The main features of the package include a variety of image buttons, a generic lightweight component and container, specialized applet and panel components, and rollover help support. The specialized applet, **StalkApplet**, provides double buffering support for all lightweight components contained within it.

5.0 BeanStalk at Work

We have now seen the range of features that BeanStalk supports. In this chapter we will learn how to assemble BeanStalk classes into working applets and look at some interactive illustrations that have been developed with it.

Creating a BeanStalk Applet with Time-varying Behavior

The following BeanStalk applet creates a number of ovals that oscillate sinusoidally. The oscillation can be turned on and off using AWT buttons and adapter classes.

```
package examples;

import stalk.canvas.*;
import stalk.canvas.geom.*;
import stalk.time.*;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class TimeBasedBehaviorsApplet extends Applet {

    // the animator that repaints the canvas
    CanvasAnimator animator_;

    public void init() {

        // creates and sets up the BeanStalk canvas
        StalkCanvas canvas = new StalkCanvas();
        canvas.setSize(300, 300);
        canvas.setDoubleBuffered(true);
        canvas.setBackground(Color.black);

        // adds the canvas to the applet
        add(canvas);

        // creates a canvas animator and adds the canvas to it
        animator_ = new CanvasAnimator();
        animator_.addCanvas(canvas);
        animator_.start();

        // creates time-based animation object and adds it to canvas
```

```

SwayingShapes shapes = new SwayingShapes(canvas.getSize());
canvas.addDrawable(shapes);

// creates AWT button and adapter for starting the animator
Button startButton = new Button("Start");
startButton.addActionListener(new StartButtonAdapter());
add(startButton);

// creates AWT button and adapter for stopping animator
Button stopButton = new Button("Stop");
stopButton.addActionListener(new StopButtonAdapter());
add(stopButton);

}

// an adapter inner class used to start the canvas animator
private class StartButtonAdapter implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        animator_.start();
    }
}

// an adapter inner class used to stop the canvas animator
private class StopButtonAdapter implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        animator_.stop();
    }
}

}

/**
 * This class encapsulates the behavior of swaying shapes. The
 * shapes' positions are based on the time context, which is fed
 * into the cosine function.
 */
class SwayingShapes implements IDrawable {

    // used to know bounds for drawing shapes
    private Dimension drawingAreaDim_;

    // specifies number of shapes to be animated
    public static final int NUM_SHAPES = 30;

    // constructs the object
    public SwayingShapes(Dimension drawingAreaDim) {
        drawingAreaDim_ = drawingAreaDim;
    }
}

```



```

// draws the shapes based on time and the cosine function
public void draw(Graphics g, ITimeContext timeCtx) {

    final double phaseIncrement = Math.PI / (NUM_SHAPES / 2);
    final double currentTime = timeCtx.getTime();

    g.setColor(Color.red);

    for (int i = 0; i < NUM_SHAPES; i++) {
        int xOffset = (int) ((drawingAreaDim_.width / 3) *
                               Math.cos(currentTime +
                                           (i * phaseIncrement)));
        int x = (drawingAreaDim_.width / 2) + xOffset;
        int y = (drawingAreaDim_.height / NUM_SHAPES) * i;

        g.fillOval(x, y, 10, 10);
    }
}

```

In this example, the shapes' behavior is a function of time and the cosine, but it could be based on many other things as well. The time context provides a lot of flexibility to animating drawable items.

Interactive Illustrations

Several interactive illustrations were developed this summer that used BeanStalk. Most were done as part of the *Web-based Academic Resources Project (WARP)*, while one was done for a course on computational geometry.

A main goal of the WARP projects was to produce graphically intensive interactive illustrations that were very user-friendly. To achieve this, the WARP applets made extensive use of BeanStalk's lightweight components. One of the illustrations demonstrates the concept of flipbook animation. It uses both the lightweight compo-

nents and the canvas framework to accomplish this.

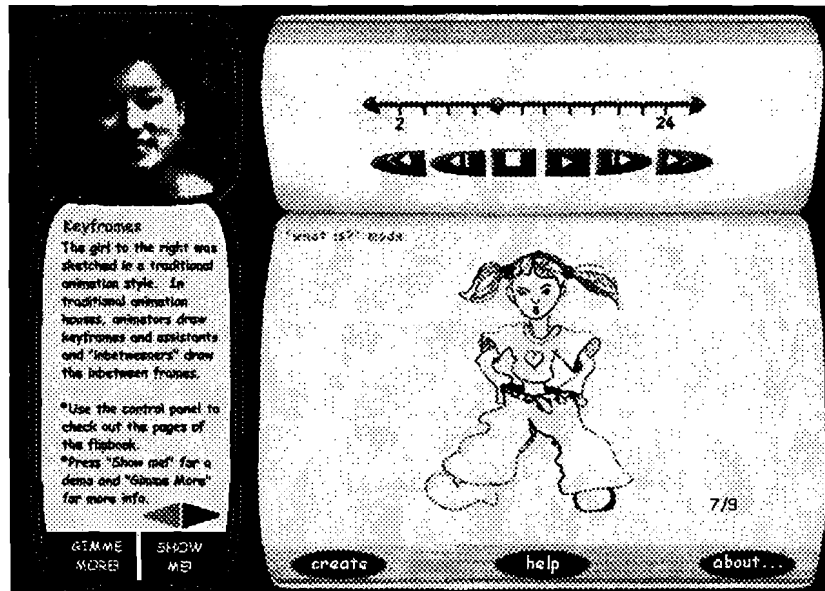


Figure 5.1: The flipbook WARP interactive illustration. The buttons and framerate control are lightweight components, while the animation of the girl is handled by a BeanStalk canvas and an associated animator.

Another WARP applet was designed to show how the halftoning process worked. Like the flipbook interactive illustration, it used lightweight components for most of the user interface. In a few cases the applet required components that were not in BeanStalk but were in AWT. The two sets of components worked together without

any problems.

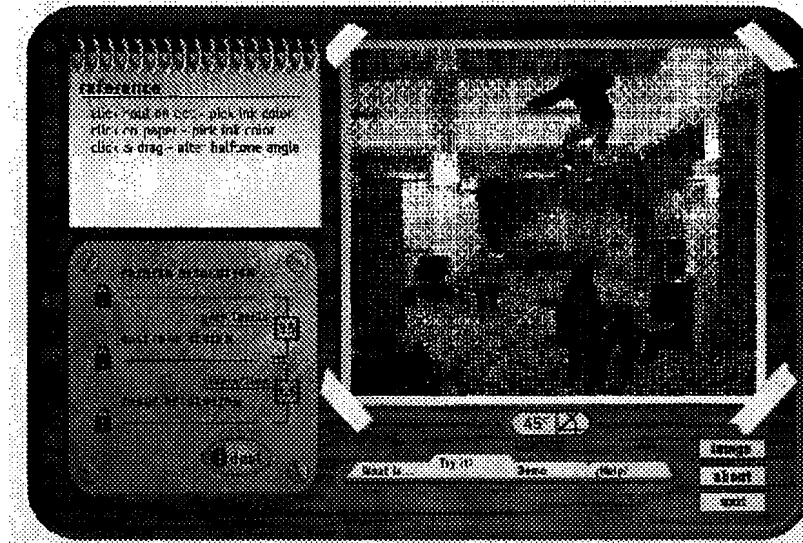
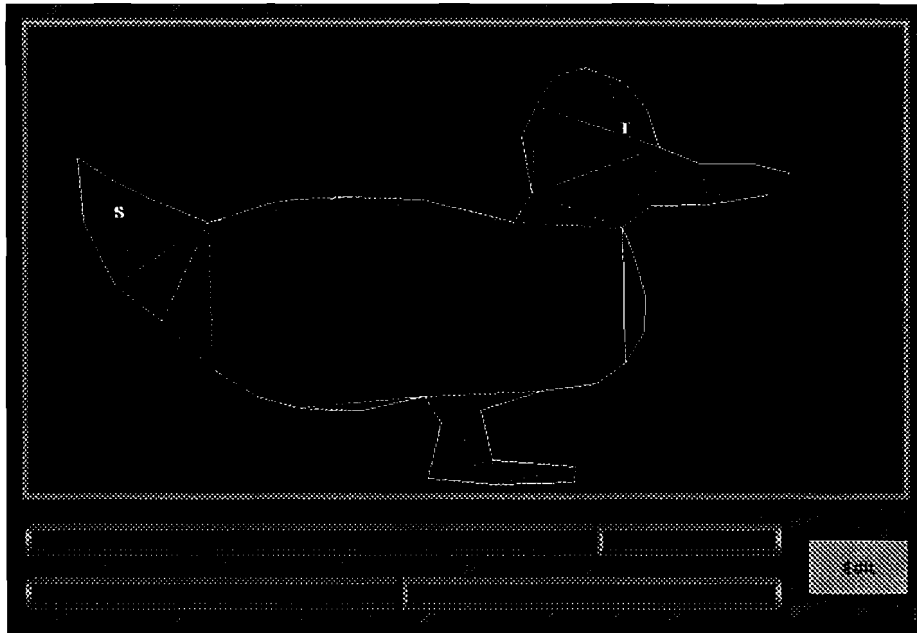


Figure 5.2: The WARP applet illustrating halftoning. The buttons and folder tabs are BeanStalk components, while the text input areas for the numbers are AWT components. The halftoned image is displayed in a BeanStalk canvas.

The interactive illustration for the computational geometry course is designed to demonstrate an algorithm that finds the shortest path between two points inside a polygon that stays within the polygon's boundary. The illustration uses time contexts as animation playback engine. After the user has created the polygon and placed the points within it, the entire algorithm is executed, creating and storing for later playback the various graphical elements that show the algorithm in action. Each one of these elements knows when in the course of the algorithm it was created so that it can know when to draw itself. When the animation playback occurs, these elements, which all implement **IDrawable**, show themselves at the appropriate time based on the current time context. The algorithm playback can be controlled by bars at the bottom of the applet. Clicking and dragging on a bar will move time around based on the bar's position. When the bar is not being dragged, it moves from left to right indicating the passing of time. Both the bar and the animation ele-

ments are using the same time context to produce their behavior.



*Figure 5.3: The shortest-path illustration. This image shows a polygon (in the shape of a duck) being triangulated. The edges' color is based on a **ColorEnvelope**, which in turn is based on a time context. The bars at the bottom show the progress through the animation. Clicking and dragging on them corresponds to moving the current time of the animation.*

These examples show BeanStalk's potential to enable the production of interactive illustrations with a strong focus on graphic design and provide some ideas for future uses of BeanStalk. Further interactive illustration development will reveal new ways of using the various classes as well as inevitable pitfalls to BeanStalk's approach.

6.0 Future Work

Developments in the world of Java are constantly taking place and BeanStalk has the potential to grow along with these changes. This chapter looks at some of the most prominent developments in the Java landscape and their impact on BeanStalk.

6.1 Extended Media Support

The upcoming *Java Media and Communications* libraries will bring much needed media support and integration to Java. These libraries include enhanced 2D capabilities, 3D support, video and audio playback, and enhanced sound support [23]. BeanStalk will immediately benefit from these libraries because of its close ties with AWT.

Java 2D is the new library for extended graphics support in Java [18] and was co-developed with Adobe, the industry leader in 2D rendering and imaging. New features include alpha channels, color depth resolution, new color models, additional graphics primitives, image filters, image compositing with alpha, anti-aliasing, spline-based paths, clipping against paths, and stroke-based fonts, among others. One of the key classes Java 2D provides is the **Graphics2D** class, which inherits from AWT's **Graphics** class. This new class has methods which support many of the features described above. To avoid having to rewrite AWT (and break a lot of code in the process), the Java 2D designers decided that none of the methods for painting components would change. The `paint` method would still have a **Graphics** object as a parameter. The trick is that `paint` will actually now be receiving a **Graphics2D** object, which works because it is a subclass of **Graphics**. Components aware of Java 2D can simply downcast the graphics context into one of type **Graphics2D**, allowing access to all of its new features. Because BeanStalk lightweight components and the **IDrawable** interface use the **Graphics** object, they can easily be reworked to use **Graphics2D**, resulting in much more graphically complex interactive illustrations.

The *Java Media Player (JMP)* APIs supports playback services of time-based media such as audio and video [24]. BeanStalk's time context model was loosely based on early versions of the JMP's notion of a *timebase*, which encapsulates the

system clock and supports time playback synchronization among players. While JMP's time model is far more complex than BeanStalk's, the two can be made to work together. The easiest way would be to create a time context wrapper around a JMP timebase. When something accesses the `getTime` method on **ITimeContext**, the wrapper implementation could get the current time out of the timebase. This would let time-varying behaviors be synchronized with JMP media playback. For example, an interactive illustration about the dot product could play a video segment that narrates the effects of the dot product while a BeanStalk animation could graphically show the effects. If the video were stopped and rewound, the animation would stop and be reset to its beginning in correspondence with the video. Since the JMP video player is itself a subclass of **Component**, it can be placed inside the applet just as if it were a lightweight component or standard AWT component. This kind of tight integration of BeanStalk and JMP's advanced media services could produce a new level of engaging interactive illustration content.

The most anticipated Java media library is *Java 3D*. It would give developers a standardized, cross-platform way of performing 3D rendering, maintaining a scene graph, and producing reactive behaviors [19]. These tools have enormous potential for Java-based interactive illustrations, because content can now enter the 3D realm without having to worry about the machine used for viewing (which is a major issue with *DirectAnimation*). It is currently unclear how tightly integrated BeanStalk and Java 3D could be. Java 3D does not support the kind of mixed-media integration that *DirectAnimation* does, so it would not be possible to integrate a BeanStalk canvas into a Java 3D virtual environment. Experimenting with both will be the only way to tell how well the two will work together.

6.2 JavaBeans Integration

BeanStalk's inspiration came from the promise of rapid prototyping of interactive illustrations using JavaBeans in a visual authoring environment — hence the *Bean* in BeanStalk. The idea was to provide a Bean component library especially tailored to the needs of illustration developers so that they could focus on content creation instead of software engineering busywork. While key players in the industry were quick to endorse the JavaBeans component model, none of them at the time of this writing have yet produced a solid, shippable product that fully supports Beans as they are described in the JavaBeans specification. Because of this, BeanStalk was

not able to take on the Bean features that were originally intended for it. Still, JavaBeans conventions were used in BeanStalk wherever possible with the hope that someday they could be turned into Bean components. Once commercial JavaBeans development environments become mature enough to allow developers to create, test, and package Beans without ever having to resort to a command line tool, then BeanStalk can begin to reveal its true nature. For the time being, interactive illustration developers will have to use it in the traditional, non-visual approach to application development.

6.3 Industry Developments and Their Impact

Lightweight components and JavaBeans are simple enough conceptually and not even that innovative, but they could potentially change the way future generations of applications are developed and viewed. If applications could be easily assembled and have a consistent look-and-feel on any given hardware platform, then the industry playing field could really be leveled. Developers and users would no longer be tied to any given operating system and would immediately know how to use applications because they would look the same regardless of what they ran on. This is Sun's vision of the future and they are producing the *Java Foundation Classes (JFC)* to deliver on it [22].

JFC is a pure-Java layer that sits on top of AWT, making extensive use of its lightweight component architecture. It provides many high-level components that AWT itself does not, such as image buttons, tree views, and tab bars, to name a few. All JFC components will also be Beans, making them available for visual authoring. In many ways JFC's goals are similar to BeanStalk's. JFC completely encompasses BeanStalk's lightweight components, although the canvas framework is still unique to BeanStalk. JFC will fully use Java 2D for enhanced graphical effects in the same way that BeanStalk could. Since JFC will be incorporated into the JDK 1.2, the duplicated features in BeanStalk will become unnecessary and counter-productive for illustration developers to use. JFC and the remaining aspects of BeanStalk have potential to be a really powerful set of tools for illustration development, especially when combined with a JavaBeans visual authoring environment.

The JFC faces tough competition because Microsoft is not about to stand by and let Sun make the Windows platform irrelevant. Since the Java platform is increasingly taking on the role of an operating system itself, Java and the JFC could very

easily lure developers (and therefore users) away from Windows. Microsoft's answer to this problem has four aspects. First, they have chosen not to support the JFC and other key libraries in the current and future JDKs, claiming that their licensing agreements lets them do so. This is bad news for anyone who wants to use JFC components because those applications are not guaranteed to run on all platforms. The second aspect is that Microsoft is developing its own pure-Java higher-level component library call the *Application Foundation Classes (AFC)*. Now developers are going to have to choose which completing set of components to base their applications on. This is further complicated by the third aspect, which is that Microsoft *will* include AFC with every copy of Internet Explorer 4 (IE4) that it ships. Since IE4 will soon ship with every copy of Windows98 and every copy of the Mac OS [2], the once-unified Java platform will become fragmented into Microsoft and non-Microsoft camps. The final aspect of Microsoft's plan is to lock Java developers into the Win32 API platform by allowing developers to make direct system calls from inside the Java virtual machine. This technology, called *J/Direct*, completely bypasses all Java class libraries, making the Java code totally unportable [16]. Microsoft's plan to turn Java into a proprietary Windows technology is thorough and *very* effective.

Some might argue that all the current Java developments are forcing Java to rapidly evolve to a new level of maturity — something that everyone will benefit from. While that may be true, the current situation presents a severe problem for all Java developers, including those creating interactive illustrations. Do they embrace the JFC and rule out a huge portion of the potential audience who will be using IE4? Do they embrace AFC, or even J/Direct? Or do they ignore all the potential benefits of the new libraries and write their own custom layers on top of AWT and other basic Java libraries? There are no satisfactory answers to any of these questions.

7.0 Conclusions

Even with its uncertain future, Java is still the best platform for illustration development because of its extensive features and guaranteed audience. The open question is which Java-based technologies are going to prove to be most useful for the development community. The answer will likely be decided within a year's time based on industry support and developer response.

BeanStalk is an example of how current Java technologies can be applied to make the illustration authoring process easier and produce more compelling content. BeanStalk's reliance on AWT is both an asset and a liability. It can take advantage of AWT's numerous features but requires the developer to have a solid understanding of how AWT works. Therefore it is not ideal for introductory students who do not have much experience with object-oriented program design. Visual authoring environments that support JavaBeans may alleviate this problem by abstracting away low-level AWT.

Ultimately, the most important part of interactive illustration development is independent of the technology involved. Experience has shown that attention to graphic design and other user interface issues early in the design process affects the final outcome at least as much as the technology used to implement the illustration. No amount of amazing technology can save a poorly designed interactive illustration.

8.0 Acknowledgments

I'd like to thank John Hughes for advising this project and seeing it through its completion, Anne Spalter and Andy van Dam for providing useful guidance and encouragement, Christine Waggoner, Scott Klemmer, and Mike Legrand for being helpful beta testers and feedback providers, Jeff White and Rosemary Simpson for proof-reading, other members of the Brown Computer Graphics Group for conversations that contributed to the outcome of this project, and my parents for being the main sponsors of this work.

9.0 References

- [1] *Aggregation and Delegation Specification* (draft). <http://splash.javasoft.com/beans/glasgow.html>, Sun Microsystems, 1997.
- [2] *Agreement Between Microsoft and Apple* (press release). <http://product.info.apple.com/pr/press.releases/1997/q4/970806.pr.rel.microsoft.html>, Apple Computer, 1997.
- [3] *AWT Enhancements in JDK1.1*. <http://java.sun.com/products/jdk/1.1/docs/guide/awt/designspec/index.html>, Sun Microsystems, 1997.
- [4] *AWT Lightweight UI Framework*. <http://www.javasoft.com/products/jdk/1.1/docs/guide/awt/designspec/lightweights.html>, Sun Microsystems, 1997.
- [5] Beall, J., Doppelt, A. and Hughes, J. "Developing and Interactive Illustration: Using Java and the Web to Make It Worthwhile," in *3D and the Internet: Information, Images, and Interaction*. Edited by Earnshaw, R., and Vince, J., Academic Press, London, 1997.
- [6] Becker, S. *Educational Interactive Illustrations*. <http://www.cs.brown.edu/research/graphics/research/illus/thesis/home.html>, Computer Science Honors Thesis, Brown University, 1997.
- [7] Conner, D., Niguidula, D. and van Dam, A. *Object-Oriented Pascal: A Graphical Approach*. Addison-Wesley, Reading, MA, 1995.
- [8] *Current Interactive Illustration Efforts in the Brown Computer Graphics Group*, <http://www.cs.brown.edu/research/graphics/research/illus/>, Brown University, 1997.
- [9] *Director*, <http://www.macromedia.com/software/director/>, Macromedia, 1997.
- [10] *Direct Animation*. <http://www.microsoft.com/msdn/sdk/inetsdk/help/dxmedia/jaxa/>, Microsoft, 1997.
- [11] Elliott, C., Schechter, G., Yeung, R. and Abi-Ezzi, S. "T-BAG: A High Level Framework for Interactive, Animated 3D Graphics Applications." *SIGGRAPH 94 Proceedings*, pp. 421-434, 1994.
- [12] Flanagan, D. *Java in a Nutshell, Second Edition*. O'Reilly & Associates, Inc., Cambridge, MA, 1997.
- [13] Gamma, H., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.
- [14] *Inner Classes Specification*. <http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html>, Sun Microsystems, 1997.
- [15] *Introduction to Application Foundation Classes*. <http://www.microsoft.com/java/pre-sdk/afc/article1.htm>, Microsoft, 1997.
- [16] *J/Direct*. <http://www.microsoft.com/sitebuilder/features/jdirect.asp>, Microsoft, 1997.
- [17] *JavaBeans Specification*, <http://splash.javasoft.com/beans/spec.html>, Sun Microsystems, 1997.

- [18] *Java2D White Paper*. <ftp://java.sun.com/docs/java-media/java2dwp.ps>, Sun Microsystems, 1997.
- [19] *Java3D API Specification*. <http://java.sun.com/products/java-media/3D/forDevelopers/3Dguide/j3dTOC.doc.html>, Sun Microsystems, 1997.
- [20] *Java Development Kit 1.0.2*, <http://www.javasoft.com/products/jdk/1.0.2/>, Sun Microsystems, 1996.
- [21] *Java Development Kit 1.1.3*, <http://www.javasoft.com/products/jdk/1.1/>, Sun Microsystems, 1997.
- [22] *Java Foundation Classes*, <http://java.sun.com/products/jfc/index.html>, Sun Microsystems, 1997.
- [23] *Java Media and Communications Home Page*, <http://java.sun.com/products/java-media/>, Sun Microsystems, 1997.
- [24] *Java Media Player Guide*, <http://java.sun.com/products/java-media/jmf/forDevelopers/playerguide/>, Sun Microsystems, 1997.
- [25] *Liquid Motion Pro White Paper*. <http://www.microsoft.com/dimensionx/lm/info/whitepaper.html>, Microsoft, 1997.
- [26] Trychin, S. *Interactive Illustration Style Guides*. <http://www.cs.brown.edu/research/graphics/research/illus/Style-Guide.html>, Computer Science Honors Thesis, Brown University, 1997.
- [27] *VisualBasic 5.0 Professional Edition*, http://www.microsoft.com/products/prodref/195_ov.htm, Microsoft, 1997.
- [28] *VRML97 Specification*. <http://www.vrml.org/Specifications/VRML97/DIS/>, VRML Consortium, 1997.

10.0 Appendix A: BeanStalk Version History

Version 1.07 — August 10, 1997

- Added the notion of a “time window” to **AShape** and modified all its subclasses to use it.
- Added support for modifier keys to **SelectEvent**, **ISelectable**, and **SelectNotifier**.
- Modified **AShape** and all of its children to support selection with modifiers.
- Modified **CanvasAnimator** so that would stopping it would freeze time on all of the registered canvases’ time contexts. Starting again would return the time contexts’ rates to their original values.
- Changed the name of **ISizable** to **ISizeable**.

Version 1.06 — July 31, 1997

- Changed **StalkApplet** so that it could create **StalkImage** objects directly.
- Tweaked parameters and behavior of **Envelope** and **ColorEnvelope**.
- Changed constructors of **SelectNotifier** and **ProbeNotifier** to take an object representing the source of the select and probe events, respectively.
- Changed name of **ButtonSelectAdapter** to **ToggleSelectAdapter**.
- Changed name of **IntersectSelectAdapter** to **RolloverSelectAdapter**.
- Added **StalkCanvasImage** class to the **stalk.canvas.geom** package.
- Fixed **HelpDisplay** so that the help message actually appears under the mouse like it’s supposed to.

Version 1.05 — July 21, 1997

- Added selection policies to **ASelectAdapter** and updated **DraggableSelectAdapter** to support them.
- Added methods to add and remove select adapters to canvases. These make it unnecessary to call the AWT methods for adding/removing listeners directly, although that’s certainly still supported.
- Added **HelpDisplay** and **HelpInfo** to the **stalk** package. These two classes provide a means to do rollover help on all components in an applet. If all the applet’s components are lightweight, then the rollover help can appear directly on top of the components.

Version 1.04 — July 17, 1997

- Added **LightWeightCanvas**, a lightweight-component equivalent of **StalkCan-**

vas, to BeanStalk. In place of double-buffering, this canvas supports transparency which allows the component behind the canvas to show through when this feature is turned on. Dirty painting is still available, but only has meaning when transparency is turned off.

- Created the **IStalkCanvas** interface to represent common functionality among canvases. Also created **StalkCanvasCore** which implements the interface. Both **StalkCanvas** and **LightWeightCanvas** implement the interface and contain a core, delegating to it when appropriate. This is Java's alternate to multiple inheritance.
- Made **CanvasAnimator** responsible for sampling time instead of **StalkCanvas** (or its lightweight sibling). This means that if a canvas is repainted by something other than an animator (e.g., AWT when a window is exposed), it won't resample time too. Also, made **StalkCanvas.canvasUpdate()** sample time before it refreshes the canvas.
- Added the **IProbeable** interface to the **stalk.canvas** package. This allows a standard way to graphical items in BeanStalk canvases to support "rollover" functionality. (The spelling of this interface may leave something to be desired, but hey, I was following convention.)
- Added the **ISizable** interface to the **stalk.canvas** package. This allows a standard way to graphical items in BeanStalk canvases to be resized.
- Extended **DraggableSelectAdapter** to work with probable items.
- Added a **IProbeNotifiable** and **ProbeNotifier** to the **stalk.canvas** package. These are the rollover equivalents to **ISelectNotifiable** and **SelectNotifier**.
- Added more classes to the **stalk.canvas.geom** package. In particular, the **AShape** class provides many convenient behaviors one would like in a shape used in a BeanStalk canvas.
- Returned **StalkApplet** to the library. All applets using BeanStalk and require double buffering should use this instead of **java.applet.Applet**.
- Added **StalkPanel** as a heavyweight container with double-buffering and dirty painting capabilities. This class is designed to be used in place of a **StalkApplet** in a Java application.
- Removed double-buffering capabilities from **StalkContainer** since that functionality will now rest in the top-most component in the applet — namely **StalkApplet**. This class is now useful for when you need a generic lightweight container. All classes that previously inherited from this should inherit from **java.awt.Container** instead because there is no gain in inheriting from it anymore.

Version 1.03 — July 15, 1997

- Added **stalk.canvas.geom** and **stalk.time.tv** packages for utility **StalkCanvas** geometry and time-varying behaviors, respectively.
- Fixed bug in **StalkImage** where the component had to be added immediately after setting the image or else a **NullPointerException** would occur. (Thanks to Scott for tracking that one down!) Also fixed similar problem in **AImageButton**.

- Added **IPositionable** interface to `stalk.canvas` to provide a standard way for items to be positioned in a **StalkCanvas**.
- Added **RolloverToggleButton** so that the button will highlight when the mouse rolls over it.
- Reworked selection notification mechanisms. Before, something that inherited from **ASelectAdapter** would be responsible for keeping track of all currently selected and deselected items and broadcasting the changes. The new approach is to have a **ISelectNotifiable** interface that an item can implement if it wants to notify a listener of a change in its select status. In addition, the **SelectNotifier** (which implements **ISelectNotifiable**) provides standard selection notification mechanisms that an item can dispatch to when needed. Finally, the **SelectEvent** has been modified to reflect these changes.

Version 1.02 — July 9, 1997

- Commented all BeanStalk classes following javadoc conventions.
- Removed **StalkContainer.getTimeContext()** and **StalkContainer.setI-TimeContext()**. These methods are more appropriate for **StalkComponent**, where they already exist. On **StalkComponent**, removed the 'I' from those method names.
- Made **AImageButton.getImages()** and **AImageButton.setImages()** operate on the actual image arrays instead of clones of them, as they did before.
- Changed the method names to add/remove/get drawable objects on **StalkCanvas**.
- Added support to add/remove/get **ISelectable** items on **StalkCanvas**.
- Added classes in the `stalk.canvas` and `stalk.event` packages to support selection mechanisms for **ISelectable** objects.
- Changed functionality of **TimeContext** so that changing the rate of time would only affect how quickly time would pass in the future instead of scaling all of time by the rate.
- Replaced **TimeContext.reset()** with **TimeContext.moveTimeTo(float time)** while maintaining the functionality of the old method (by passing in 0.0f).
- Made **StalkCanvas.getDrawables()** return the actual vector instead of a clone of it, as it did before.
- Performed additional code reworking that does not affect functionality or APIs.

Version 1.01 — July 7, 1997

- Changed name of **SmartCanvas** to **StalkCanvas**.
- Added functionality to **StalkCanvas** so that the buffer can be made to not clear itself before painting begins. (Called "dirty painting" in code and comments.) Works with double-buffering too.
- Made **ImageButton** fire events of type **java.awt.ActionEvent** instead of **But-**

tonEvent.

- Removed **ButtonEvent** and **ButtonEventListener** from **stalk.event** package.
- Removed **StalkApplet** from BeanStalk library.
- Created **AImageButton** which is an abstract class that all image buttons inherit from. All classes that inherit from it use **java.awt.ActionEvent** for event notification.
- Created **ToggleButton** which inherits from **AImageButton**.
- Created **RadioButton** which inherits from **ToggleButton**.
- Created **RadioButtonContainer** from **StalkContainer**. Works in conjunction with **RadioButtonManager**.
- Created **RadioButtonManager** to control radio-button interaction behavior for objects of type **RadioButton**. Works in conjunction with **RadioButtonContainer**.
- Added some preliminary comments for a few classes. Full comments will appear in next version.

Version 1.0 — June 30, 1997

- Released first version of BeanStalk packages, based on “alpha” versions of classes.