Peppery Patterns

How to turn Java applets into Habanero hablets

Ian Chai
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue
Urbana, IL 61801
Phone: (217)367-6140
Fax: (217)367-7765
chai@uiuc.edu

Introduction

We have seen many "timeless" designs and procedures described with patterns. Many people have built OO frameworks using those designs since the Gang of Four book.

This pattern language is an exploration of a rather different use of patterns — instead of using them to *design* a framework, I am using them to teach a new user how to *use* a framework. This pattern language is a kind of user documentation or tutorial. It reveals a few differences between pattern-based representation and traditional documentation, and can serve as a model for other people to produce pattern-based framework guides.

I call these kinds of patterns *pedagogical framework patterns*. "Pedagogy" is "the art, science, or profession of teaching" [Merriam-Webster's Collegiate Dictionary] so pedagogical framework patterns are patterns that teach you how to use a particular framework.

The patterns are also available as web pages at `http://st-www.cs.uiuc.edu/~chai/writing/pepperypatterns/`. In fact, the main use of the patterns are *as* web pages.

Square brackets indicate page references; for example, "Read Me First [2]" means that "Read Me First" is found on page 2.

Why use Habanero?

You have a Java applet that you would like to make multiuser. Habanero will let a group of people run your applet at the same time and changes made by one person be immediately visible to other people. These are patterns that you may follow to convert a Java applet into a Habanero *hablet*.

Index

Read Me First

These web pages teach you how to get started writing Habanero applications.

I am assuming you already know the Java programming language. If you do not, please read the Java Tutorial[1] and get familiar with writing applets first. If you have not yet installed Habanero on your machine, go download it[2]. Play around with it to get used to its GUI. (See the user's guide[3].)

These pages are designed so that you can flip from one to another at will as you learn Habanero. However, most people will probably want to start by Habanerizing a Java Applet[3].

Each pattern usually has these sections:

What's this pattern for?
Tells you when you want to use this pattern

Background Information
What some of the things you should know before using this pattern are.

How do I use this?
This part tells you what to do.

What next?
Suggested places to go next. (This part is not always present)

---

1 http://java.sun.com/docs/books/tutorial/index.html

2 http://www.ncsa.uiuc.edu/SDG/Software/Habanero/Release/

3 http://www.ncsa.uiuc.edu/SDG/Software/Habanero/Docs/

Habanerizing a Java Applet

What's this pattern for?

You have a Java applet, and you want to turn it into a hablet.

Background Information

Typically, Java applets are single-user applications in HTML pages on the Web. The most straightforward habanerization of an applet would allow multiple users across the world-wide network to pretend they are in the same room, interacting with the same application.

For example, with the Whiteboard application, all the users can draw on the whiteboard and see what the other users have drawn as if they were all standing around the same whiteboard[4].

How do I use this?

Here is a straightforward way to habanerize an applet:

- Subclass it, and to put all the changes in the subclass [4]. That way, you do not have to mess up your original applet.
- Implement the `Wrapped` interface [5]
- Take care of event handling [6].
- Specify the default window parameters [7], so that Habanero will know how you want the window to look like when this hablet opens.
- Compile your hablet like any other Java class and install it into the Habanero environment [8].

What next?

After you are comfortable with this basic way of habanerizing an applet, you can explore some of the more advanced topics [bottom of 1].

---

4 Actually, the Whiteboard application was written from scratch as a hablet, but you can imagine that if we already had a whiteboard we could habanerize it for the same purpose.

Habanerizing by Subclassing

What's this pattern for?

One straightforward way to habanerize an applet [3] is to subclass it, and to put all the changes in the subclass. Here are some of the things you need to do when you do this.

Background Information

Habanero needs to reroute the events and send them on to other sessions. Thus, you need to find the classes in your applet that handle events and subclass them. In most applets, this is the class that is a subclass of `Applet`.

How do I use this?

So that Habanero and Java can find the files without you needing to add it to the `CLASSPATH`, you need to pick a package name and indicate it at the start of every `.java` file (including the original applet's files.) Make a subdirectory of `demos` or `tools` with the same name as the package name and put all your class files there.

Example *from the example of habanerizing TextDemo.java*. [18]

```
package HabTextDemo;
```

Along with the standard imports of

```
import java.io.*;
import java.awt.*;
import java.lang.*;
```

you need to also

```
import ncsa.habanero.*;
import ncsa.habanero.streams.*;
```

Your subclass should *extend* the applet and *implement* `Wrapped`, which consists of:

- `marshallSelf()` and `unmarshallSelf()` [5]
- `doEvent()` [6]
- `startInFrame()` [7]

Example *from the example of habanerizing TextDemo.java*. [18]

```
public class HabTextDemo extends HabTextDemo.TextDemo implements Wrapped
```

What next?

- Implement the Marshallable [5] interface.
- Take care of Event Handling [6].
- Specify the default Window parameters [7].
- Install your new hablet into your Habanero environment [8].

Implement the Marshallable Interface

What's this pattern for?

Hablets must implement the `Wrapped` interface, which includes as part of it the `Marshallable` interface. This pattern tells you how to implement the `Marshallable` interface.

Background Information

Since hablets can run simultaneously on many different machines, there must be some way to encapsulate all its internal state and send it over the wire so that a new session starting up in a remote location can get up to speed.

This process is called *marshalling*, so the `Marshallable` interface sets out how it does this.

This pattern assumes you have already done the things from the Subclassing the Applet [4] pattern.

How do I use this?

The `Marshallable` interface has these methods:

```
 public void marshallSelf(MarshallOutput out) throws IOException
```

and

```
 public void unmarshallSelf(MarshallInput in) throws IOException
```

`marshallSelf` writes the state of the applet to the `MarshallOutput` and `unmarshallSelf` reads the state of the applet from the `MarshallInput`.

Example *from the example of habanerizing TextDemo.java.* [18]

```
// Writes out current state for new user.
public void marshallSelf(MarshallOutput out) throws IOException
{ out.writeUTF(textArea.getText());
}

// Reads in current state for new user.
public void unmarshallSelf(MarshallInput in) throws IOException
{ textArea.setText(in.readUTF());
}
```

`MarshallOutput` extends `java.io.DataOutput`, and `MarshallInput` extends `java.io.DataInput`, so `marshallSelf()` and `unmarshallSelf()` can use any of the methods of `java.io.DataOutput`[5] and methods of `java.io.DataInput`[6] to send data to the other hablets. The example above uses `writeUTF()` and `readUTF` as the data item is a string. If the data item were an integer, it would use `writeInt()` and `readInt`, for example.

In addition to these methods, `MarshallOutput` can use `writeMarshallable()` and `MarshallInput` can use `readMarshallable` on any class that implements the `Marshallable` interface we taught you in this pattern. For example, if `thingy` is a `Marshallable` of class `ThingType`, then you can:

```
public void marshallSelf(MarshallOutput out) throws IOException
{ out.writeMarshallable(thingy);
}

public void unmarshallSelf(MarshallInput in) throws IOException
{ thingy = (ThingType) in.readMarshallable();
}
```

This means that objects that are marshallable can be easily marshalled in other objects which are marshallable.

What next?

- Take care of Event Handling [6].
- Specify the default Window parameters [7].
- Install your new hablet into your Habanero environment [8].

---

5 http://www.javasoft.com/products/jdk/1.1/docs/api/java.io.DataOutput.html

6 http://www.javasoft.com/products/jdk/1.1/docs/api/java.io.DataInput.html

(Simple) Event Handling

What's this pattern for?

This pattern tells you about how to do simple event handling in Habanero.

Background Information

A collaborative program has to inform all members of a session when an event occurs. In Java, whenever an event occurs, it calls certain methods in the applet to handle the event — for example, `action()` and `mouseDown()`. So, a hablet must redirect these events to Habanero, so it can in turn redirect them to all the copies of the hablet in a session.

This patterns assumes that you are making a hablet by subclassing an applet [4]. Also, don't forget to implement the `Wrapped` interface [5].

If you need to preserve event identity, or if you don't want to share all the events, please refer to Advanced Event Handling [9]. If you have several widgets of the same class, you might need to solve the problem of event identity.

How do I use this?

In a normal Java applet, methods like `action()` or `mouseDrag()` handle the events. Since we need to redirect the events, we need to split such methods into two pieces:

- Rewrite the methods like `action()` to forward the event to Habanero. (Unless you want to do Advanced Event Handling [9].)
- Habanero will forward the event to every hablet in the session and call each hablet's `doEvent()` method, including the hablet that originally generated the message, so that even the original hablet will carry out the event in `doEvent()`. Thus, `doEvent()` needs to do the things that the original `action()` or other event-handling method did.

Example *from the example of habanerizing TextDemo.java*. [18]

```
public boolean action (Event e, Object arg)
{ return false; }
```

Now, write `public boolean doEvent (Event evt, Object arg)` which receives the events from Habanero and processes them, similar to what `action()` used to do.

> If your actions depend upon the identity of the sender of the event, please refer to Advanced Event Handling [9]. If you don't want to share all your events, see Private Events [16].

Example *from the example of habanerizing TextDemo.java*. [18]

```
public boolean doEvent (Event evt, Object arg)
{ textArea.appendText((String)(arg) + "\n");
  textField.selectAll();
  return true;
}
```

What next?

- Specify the default Window parameters [7].
- Install your new hablet into your Habanero environment [8].

If you need to preserve event identity, or if you don't want to share all the events, please refer to Advanced Event Handling [9].

Specifying the Default Window Parameters

What's this pattern for?

This pattern tells you how to specify how the window should look like when you open the hablet in Habanero.

Background Information

Regular applets usually run in HTML pages, and the APPLET tag in HTML specifies how big the space for the applet should be.

In Habanero, there is no HTML page. Instead, the hablet runs in its own window. Thus it needs to have a method that tells it what these details are.

How do I use this?

The name of the method that does this is `startInFrame` and it has one parameter, a `MirrorFrame`. The body of `startInFrame` should set the title, size of the frame, add the hablet to it, and also call `MirrorFrame`'s `show()` method at the end, to display the window.

Example *from the example of habanerizing TextDemo.java*. [18]

```
public void startInFrame(MirrorFrame f)
{ init();
  start();

  f.setTitle("TextDemo Hablet");
  f.add("Center", this);
  f.resize(300,200);
  f.show();
}
```

What next?

Install your new hablet into your Habanero environment [8].

Installing a New Hablet

What's this pattern for?

You have written, or were given, a new Hablet and you would like to install it into your Habanero environment.

Background Information

At this point, the Hablet should have been compiled with a Java compiler into a class. You may also have an icon, in the form of a GIF, to display in the **Tools Palette**.

How do I use this?

- Copy the GIF into `ristra/rsc/env/co_images`, if you have one. The GIF should have a size of 32x32 pixels.
- Create a directory under either `ristra/tools` or `ristra/demos` with the same name as the `package` name you selected when you subclassed the applet [3].
- Place all the `.class` files for your applet there (as well as all the `.java` files, if you so desire.)
- Create a new file in `habanero.rsc/toolsDir` that tells Habanero what the name, picture, classname, version, and, optionally, the help URL of the new tool.

Example from the example of habanerizing TextDemo.java [18].

Place `TextBoxIcon.gif` into `ristra/rsc/env/co_images` and create the file `habanero.rsc/toolsDir/HabTextDemo` which contains:

```
tool.name=Text Demo
tool.picture=TextBoxIcon.gif
tool.classname=HabTextDemo.HabTextDemo
tool.version=v0.1
```

If your hablet marshalls large chunks of data, you might want it to use its own socket to do the events, instead of using Habanero's main socket. This is so that the main socket won't be tied up when your application is transmitting its large chunk of data, for example, if you were to transfer a GIF in the whiteboard.

To do this, add:
`tool.useOwnStream=true`

If you have a web page documenting your hablet, you can add to the tool file:

`tool.help="http://url.for.the.help.pages"`

You can now run Habanero and your new tool should be available.

What next?

Try out the advanced topics [bottom of 1].

Advanced Event Handling

What's this pattern for?

Simple Event Handling [6] is not always sufficient.

Sometimes you need your hablet to react differently to widgets of the same type. Sometimes you need to pass on some events to other collaborating Hablets, while handling others locally.

Background Information

The Problem of Event Identity

In normal Java, when you do something with a widget, for example, type something in a text box or press a button, it generates an event. This event keeps a pointer to the widget that generated it, so that in the `action` method can use == to check to see which widget generated it.

In Habanero, however, that widget may be on a different machine. The pointer in the event received via Habanero actually points to a *copy* of the widget, and not the widget itself. Because of this, == will not work.

Thus, we need to have some alternative way of identifying the sender of an event.

The Problem of Private Events

Sometimes you don't want *every* event to be broadcast to collaborating hablets. Some events may be too fine-grained. For example, when drawing points in the **Whiteboard** , they did not want to send *every* point over the network, but rather, they waited until all the points for a scribble have been collected, then sent them over the network in one package.

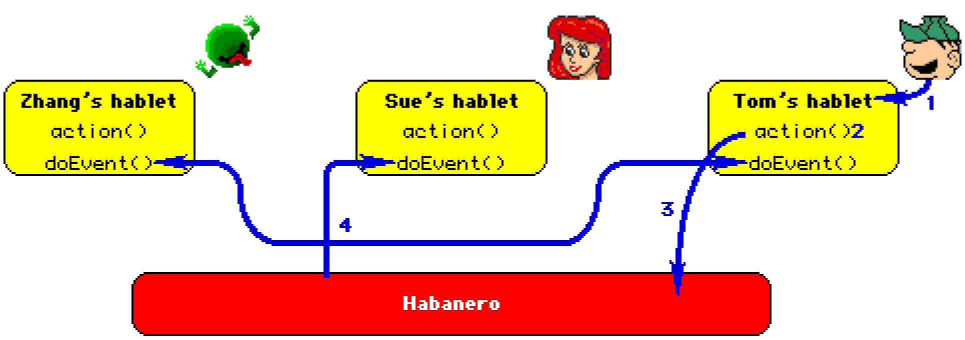
The Different Functions of `action()` and `doEvent()` in a Hablet

In a normal applet, most events are handled by `action()`. In a hablet, however, `action()` is called only on the hablet which is *local* to the event. If `action()` returns `false`, it passes the event on to Habanero. Habanero then sends it to the `doEvent()` functions on all the collaborating sessions.

For example:

Tom, Sue, and Zhang are collaborating.

- Tom does something to his hablet.
- His hablet's `action()` gets the event.
- His hablet's `action()` passes the event on to Habanero.
- Habanero hands the event to `doEvent()` on each of Tom, Sue and Zhang's hablets.



Because the local `action()` gets the event, you can place code there to do things before Habanero sees it. In fact, if you have `action()` return `true` instead of the normal `false`, Habanero will not even see and propagate that event.

How do I use this?

The Problem of Event Identity

`action()` and `doEvent()` have the same two arguments: an **event** and an **argument**. In normal Java, the **event**'s `target` has a pointer to the widget that generated that event. Both those arguments are valid in `action()`, but since Habanero is distributed, such a pointer cannot be valid, so you should ignore the **event** parameter in `doEvent()`.

Many widgets put all the information you need in the **argument**. For example, a button widgets and menu widgets put the name of the button or menu item selected there, as a string.

Possible Solutions to the Problem of Event Identity

- Simply call `action()` in `doEvent` if your original applet's `action()` only used the **argument** parameter [11].
- Encode all the data for the event you need as a string [12].
- Create and use message objects [14].

The Problem of Private Events

If you do not want some events broadcast to your collaborators, you can deal with them in `action()` before sending it on to Habanero. Then `doAction()` only needs to deal with those events which are shared. Here's how you do this 16].

What next?

Other advanced topics [bottom of 1].

Events: Only Using “Argument”

What’s this pattern for?

This is a simple way to solve the Problem of Event Identity [9].

Background Information

By the time Habanero calls `doEvent()`, the **event** argument’s target is lost, because it could have been on a different machine. However, if the **argument** argument is a string, Habanero is careful to transfer it intact.

How do I use this?

You can use this method if in your original applet’s `action(Event evt, Object arg)`,

- `arg` is a string
- It does not use `evt` at all.
- It does not get information from the widgets directly. (See the example below for an illustration of this.)

If your applet meets these conditions, or if you can modify your applet to meet these conditions, then you can simply put this code in your Hablet to take care of the events:

```
public boolean action(Event evt, Object arg)
{ return false;
}

public boolean doEvent(Event evt, Object arg)
{ return super.action(evt,arg);
}
```

Example

You’ll recall that in running example from the Basic Topics [bottom of 1] section, we showed you an example of habanerizing `TextDemo`. We could not use this method then because `action()` took information from the `textField` widget directly. So, if we were to simply call `action()` from `doEvent()`, we would have gotten the *local* `textField`’s contents, instead of the one transmitted from the actual generator of the event.

Why the above scenario produces the wrong result

- Fred types “Hello” in his hablet’s `textField`
- Fred’s `action()` doesn’t do anything, and it passes it on to Habanero.
- Habanero passes it on to both Fred’s and Wilma’s `doEvent()`
- Fred’s `doEvent()` and Wilma’s `doEvent` calls their respective `super.action()`s.
- Fred’s `super.action()` gets the string “Hello” from its `textField`, but Wilma’s `super.action()` gets whatever was in *Wilma’s* `textField` — hence it puts the *wrong* thing into Wilma’s `textArea`.

An alternate scenario where this would work

The applet’s `action(event, argument)` needs to get all its information from `argument` like so:

Example

```
public boolean action (Event evt, Object argument)
{ if (argument instanceof String)
      textArea.appendText((String)(argument) + "\n");
  return true;
}
```

Given this one small change (which you can put in a subclass), then this method would work.

Here’s the code for this example:

- The one small change [19].
- The hablet code [19].

What next?

If your applet does not lend itself to this solution, try looking at Encoding Events as Strings [12] or Message Objects [14]

Events: Encoding as Strings

What's this pattern for?

This is one way to solve the Problem of Event Identity [9].

Background Information

By the time Habanero calls `doEvent()`, the **event** argument's target is lost because it may be on a different machine. However, if the **argument** argument is a string, Habanero is careful to transfer it intact.

So, you can encode all the event's data that you need as a string and put that in the **argument**. This is a good solution if you have a reasonable and easily understood way of converting all the data you need to a string.

How do I use this?

We need to package up the event's data *before* sending it on to Habanero, so we need to do this in `action(evt,arg)`. To modify the `arg` that `doEvent(evt,arg)` receives, we assign what we want to the `evt.arg` in `action(evt,arg)`.

For example, if you need to know the identity of the widget that generated the event, you could put that as a key in the first character of the string, and put the rest of the data after that.

Example from the Labeled Text demo [20]

```
public boolean action(Event evt, Object arg)
{ String text = (String) arg;

  if (evt.target == f1)
  { evt.arg = "1" + text;      // See how we use this part to
    f1.selectAll();            // package all our information as
  }                            // a string in to arg before we send
  else if (evt.target == f2)   // it on to Habanero? We also take this
  { evt.arg = "2" + text;      // opportunity to do some local things
    f2.selectAll();            // like do the selectAll on the
  }                            // appropriate TextField.
  return false;
}
```

Then, on the other side, in `doEvent(evt,arg)`, you unpack the data into its various components again.

In the case of our example, we would take the first character as the key, and the rest of the string as the data. Then we reproduce the logic using the key instead of the original widget's identity.

Example from the Labeled Text demo [20]

```
public boolean doEvent (Event evt, Object arg)
{ // At this point, the arg's first character tells us which
  // TextField generated the event. So now we need to extract that
  // information and separate it from the rest of the string.

  char key = ((String) arg).charAt(0);
  String text = ((String) arg).substring(1);

  // Now we can just follow the logic from the original action(),
  // except we're using the key instead of the evt.target.

  if (key == '1')
  { outArea.appendText("Box 1: " + text + "\n");
    f1.selectAll();
  }
  else if (key == '2')
  { outArea.appendText("Box 2: " + text + "\n");
    f2.selectAll();
  }
  else // Just in case anything goes wrong
  { outArea.appendText("Error: '"+key+"' "+text+"\n");
  }
  return true;
}
```

What next?

If `arg` is already a string containing all the information you need, you can just use the argument parameter [11].

If it is not reasonable to convert all the data to a string, use a Message Object [14] instead.

Events: Message Objects

What’s this pattern for?

This is one way to solve the Problem of Event Identity [9].

Background Information

By the time Habanero calls `doEvent()`, the **event** argument’s target is lost because it may be on a different machine.

If the **argument** argument is a string, Habanero is careful to transfer it intact. However, sometimes it does not make sense to always be encoding everything as a string — as Java is an Object-Oriented language, we would often like to pass objects across Habanero directly.

How do I use this?

An object that wants to be passed to other hablets via Habanero must Implement the Marshallable Interface [5]. The one exception is strings — Habanero takes care of strings as a special case [12]. Your main hablet then can use this marshallable object to package up event data to pass on to the others.

Implementing the message object

The message object must implement `marshallSelf()` and `unmarshallSelf()`, as described in the pattern on implementing the Marshallable Interface [12].

Example from Hchat.Message [23]

```
// Writes out current state
public void marshallSelf (MarshallOutput out) throws IOException
{ out.writeUTF(strMessage);
  out.writeInt(nValue);
}

// Reads in current state
public void unmarshallSelf (MarshallInput in) throws IOException
{ strMessage = in.readUTF();
  nValue = in.readInt();
}
```

It also needs to have a constructor which the hablet uses to give it data.

Example from Hchat.Message [23]

```
public Message (String str, int n)
{ strMessage = str;
  nValue = n;
}
```

Finally, it needs accessor methods so that the hablet on the other end can retrieve the information.

Example from Hchat.Message [23]

```
public int getField ()
{ return nValue;
}

public String getMessage ()
{ return strMessage;
}
```

Using the message object

The calling hablet's `action(event,argument)` needs to package the information up into the message object using the methods it provides, and assign it to the `event.arg`. Then when it (`action()`) returns `false`, Habanero takes over and sends the event on to all the collaborating hablets.

Example from Hchat.Message [23]

```
public boolean action (Event e, Object arg)
{ if (arg.getClass().getName().equals ("java.lang.String"))
  { if (e.target == in1)
      { e.arg = new Message (in1.getText(), 1); }
    else if (e.target == in2)
      { e.arg = new Message (in2.getText(), 2); }
  }
  return false;
}
```

What next?

Other advanced topics [bottom of 1].

Private Events

What's this pattern for?

This pattern tells you how to let the local hablet process some of the events locally and not broadcast them over Habanero.

Background Information

Sometimes you don't want *every* event to be broadcast to collaborating hablets. Some events may be too fine-grained. For example, when drawing points in the **Whiteboard** , they did not want to send *every* point over the network, but rather, they waited until all the points for a scribble have been collected, then sent them over the network in one package.

How do I use this?

If you do not want some events broadcast to your collaborators, you can deal with them in `action()` (or various other event-handling routines like `mouseDown()` or `mouseDrag()`) before sending it on to Habanero. Then `doAction()` only needs to deal with the shared events.

Example: `PrivateEventDemo.java [22].`

This applet lets the user type as much as they want in the `TextArea`. Only when they press the **Send** button does it make a public event and send the data therein to the collaborators.

Here is the crucial code: in `action()` it checks to see if the event came from a button — ie. a public event — and if so, it sends it on. Otherwise, it keeps it to itself.

```
public boolean action(Event evt, Object arg)
{ if (evt.target instanceof Button)
  // If it is a button, it is a public event and we need to send
  // the data on by putting it in the evt.arg and then returning
  // false.
  { evt.arg = pad.getText();
    return false;
  }
  else
  // If we had needed to do any processing of private events,
  // we would have done it here before we returned true.
    return true;
}
```

Then, in `doEvent()` it knows it was a public event, so it simply takes the data and puts it into the `TextArea pad`.

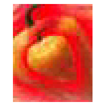
```
public boolean doEvent(Event evt, Object arg)
{ pad.setText((String) arg);
  return true;
}
```

What next?

To learn about more sophisticated means of sending event data to other hablets, read Advanced Event Handling [9].

Other advanced topics [bottom of 1].

Epilogue: Lessons Learned



The biggest lesson is that you cannot write good framework documentation on the first try. This is not surprising to the Pattern community since we are always submitting our patterns to writers' workshops to get them reviewed.

However, when writing documentation for *teaching* frameworks, not only do you need to have experts in writers' workshops read and critique your patterns, you need to also test them out on your end-users — the very kind of people who will be learning how to use the framework from your patterns. You need to watch them use your patterns and note what trips them up, then correct it.

People, unlike computers, are notoriously bad at following step-by-step instructions. People are more motivated when there is something they themselves want to do driving them. So as you write your patterns, whenever possible, make them random-access. Provide an index with suggestive titles so that they can pick which patterns to read based on what they want to do. Sometimes this is not possible, because of preconditions. In those cases, make sure your patterns point them back to what they needed to do first.

Application programmers — the consumers of your framework patterns — are often not familiar with the concept of patterns. So you need to introduce them to the idea. However, do *not* label your introduction "Introduction." I tried that, and *nobody* read them. I changed the name to "Read Me" and almost *everybody* read them!

Appendix

HabTextDemo.java

```
// This is the simple demo of habanerizing the TextDemo from the Java
// Tutorial.

package HabTextDemo;

import TextDemo;
import ncsa.habanero.*;
import ncsa.habanero.streams.*;
import java.io.*;
import java.awt.*;
import java.lang.*;

public class HabTextDemo extends HabTextDemo.TextDemo implements Wrapped
{
  // Override the super's action() so that the action is passed on
  // to Habanero.
  public boolean action (Event e, Object arg)
  { return false;
  }

  // Sets up the window and calls super's init. Sets the size of the
  // window like what the <applet> tag does in HTML.
  public void startInFrame(MirrorFrame f)
  { init();
    start();

    f.setTitle("TextDemo Hablet");
    f.add("Center", this);
    f.resize(300,200);
    f.show();
  }

  // Writes out current state for new user.
  public void marshallSelf(MarshallOutput out) throws IOException
  { out.writeUTF(textArea.getText());
  }

  // Reads in current state for new user.
  public void unmarshallSelf(MarshallInput in) throws IOException
  { textArea.setText(in.readUTF());
  }

  // Like a distributed action(), this receives events, which may
  // be from a remote app. Note how it parallels the action() in
  // TextDemo.
  public boolean doEvent (Event evt, Object arg)
  { textArea.appendText((String)(arg) + "\n");
    textField.selectAll();
    return true;
  }
```

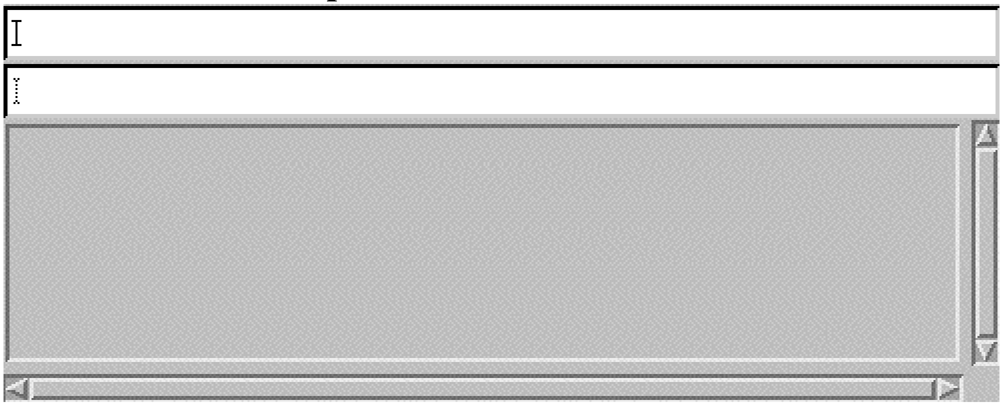
TextDemoSub.java

```
// This implements an alternate action() which uses the argument rather
// than directly accessing textField.

package HabTextDemo;

import java.awt.*;
import java.applet.Applet;

public class TextDemoSub extends HabTextDemo.TextDemo
{ public boolean action (Event evt, Object argument)
  { textArea.appendText((String)(argument) + "\n");
    return true;
} }
```

HabTextDemo2.java

```
// This is the simple demo of habanerizing the TextDemo from the Java
// Tutorial, Version II. This is the "Use only argument parameter"
// version.

package HabTextDemo;

import TextDemo;
import ncsa.habanero.*;
import ncsa.habanero.streams.*;
import java.io.*;
import java.awt.*;
import java.lang.*;

public class HabTextDemo extends HabTextDemo.TextDemoSub implements Wrapped
{
  // Override the super's action() so that the action is passed on
  // to Habanero.
  public boolean action (Event e, Object arg)
  { return false;
  }

  // Sets up the window and calls super's init. Sets the size of the
  // window like what the <applet> tag does in HTML.
  public void startInFrame(MirrorFrame f)
  { init();
    start();

    f.setTitle("TextDemo Hablet");
    f.add("Center", this);
    f.resize(300,200);
    f.show();
  }

  // Writes out current state for new user.
  public void marshallSelf(MarshallOutput out) throws IOException
  { out.writeUTF(textArea.getText());
  }

  // Reads in current state for new user.
  public void unmarshallSelf(MarshallInput in) throws IOException
  { textArea.setText(in.readUTF());
  }

  // Like a distributed action(), this receives events, which may
  // be from a remote app. Note how it parallels the action() in
  // TextDemo.
  public boolean doEvent (Event evt, Object arg)
  { return super.action(evt,arg);
  }
}
```

Labeled Text Example



HabLabeledText.java

```
// HabLabeledText
// Demonstrates "Encoding Event Data as Strings"

package LabeledTextDemo;

import LabeledTextDemo.*;
import ncsa.habanero.*;
import ncsa.habanero.streams.*;
import java.io.*;
import java.awt.*;
import java.applet.*;

public class HabLabeledText extends LabeledTextDemo.LabeledText implements Wrapped
{
  public boolean action(Event evt, Object arg)
  { String text = (String) arg;

    if (evt.target == f1)
    { evt.arg = "1" + text;  // See how we use this part to
      f1.selectAll();    // package all our information as
    }             // a string in to arg before we send
    else if (evt.target == f2)   // it on to Habanero? We also take this
    { evt.arg = "2" + text;  // opportunity to do some local things
      f2.selectAll();    // like do the selectAll on the
    }             // appropriate TextField.
    return false;
  }

  public boolean doEvent (Event evt, Object arg)
  { // At this point, the arg's first character tells us which
    // TextField generated the event. So now we need to extract that
    // information and separate it from the rest of the string.

    char key = ((String) arg).charAt(0);
    String text = ((String) arg).substring(1);

    // Now we can just follow the logic from the original action(),
    // except we're using the key instead of the evt.target.

    if (key == '1')
    { outArea.appendText("Box 1: " + text + "\n");
      f1.selectAll();
    }
    else if (key == '2')
    { outArea.appendText("Box 2: " + text + "\n");
      f2.selectAll();
    }
    else      // Just in case anything goes wrong
    { outArea.appendText("Error: '"+key+"' "+text+"\n");
    }
    return true;
  }

  // Writes out current state for new user.
  public void marshallSelf(MarshallOutput out) throws IOException
  { out.writeUTF(outArea.getText());
  }
```

```
  // Reads in current state for new user.
  public void unmarshallSelf(MarshallInput in) throws IOException
  { outArea.setText(in.readUTF());
  }

  // Sets up the window and calls super's init. Sets the size of the
  // window like what the <applet> tag does in HTML.
  public void startInFrame(MirrorFrame f)
  { init();
    start();

    f.setTitle("TextDemo Hablet");
    f.add("Center", this);
    f.resize(300,400);
    f.show();
  }
}
```

LabeledText.java

```
// LabeledText
// Labels what you type depending on the box you type it in.

package LabeledTextDemo;

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

public class LabeledText extends Applet
{ TextField f1, f2;
  TextArea outArea;

  public void init()
  { f1 = new TextField(20);
    f2 = new TextField(20);
    outArea = new TextArea(5, 20);
    outArea.setEditable(false);

    //Add Components to the Applet.
    GridBagLayout gridBag = new GridBagLayout();
    setLayout(gridBag);
    GridBagConstraints c = new GridBagConstraints();
    c.gridwidth = GridBagConstraints.REMAINDER;

    c.fill = GridBagConstraints.HORIZONTAL;
    gridBag.setConstraints(f1, c);
    add(f1);
    gridBag.setConstraints(f2, c);
    add(f2);

    c.fill = GridBagConstraints.BOTH;
    c.weightx = 1.0;
    c.weighty = 1.0;
    gridBag.setConstraints(outArea, c);
    add(outArea);

    validate();
  }

  public boolean action(Event evt, Object arg)
  { String text = (String) arg;

    if (evt.target == f1)
    { outArea.appendText("Box 1: " + text + "\n");
      f1.selectAll();
    }
    else if (evt.target == f2)
    { outArea.appendText("Box 2: " + text + "\n");
      f2.selectAll();
    }
    else       // Just in case anything goes wrong
    { outArea.appendText("Error: "+text+"\n"); };
    return true;
  }
}
```

PrivateEventDemo

```java
// Private Event Demo
// This lets you type away, and not send the data until you press the
// send button.

package PrivateEventDemo;

import ncsa.habanero.*;
import ncsa.habanero.streams.*;
import java.io.*;
import java.awt.*;
import java.applet.Applet;

public class PrivateEventDemo extends Applet implements Wrapped
{ TextArea pad;
  Button sendButton;

  // Here we check to see if the event is the "send" button. If it
  // is, we pass it on, else, we handle the event locally (which
  // is actually to do nothing since we're just letting the user
  // continue to type in the TextArea pad.
  public boolean action(Event evt, Object arg)
  { if (evt.target instanceof Button)
    // If it is a button, it is a public event and we need to send
    // the data on by putting it in the evt.arg and then returning
    // false.
    { evt.arg = pad.getText();
      return false;
    }
    else
    // If we had needed to do any processing of private events,
    // we would have done it here before we returned true.
      return true;
  }

  // Here we received the public event and have to handle it
  // (which in this simple example is just simply to put the
  // data in the pad.)
  public boolean doEvent(Event evt, Object arg)
  { pad.setText((String) arg);
    return true;
  }

  public void init()
  { pad = new TextArea(5, 20);
    sendButton = new Button();
    sendButton.setLabel("Send");

    //Add Components to the Applet.
    GridBagLayout gridBag = new GridBagLayout();
    setLayout(gridBag);
    GridBagConstraints c = new GridBagConstraints();
    c.gridwidth = GridBagConstraints.REMAINDER;

    c.fill = GridBagConstraints.HORIZONTAL;
    gridBag.setConstraints(sendButton, c);
    add(sendButton);

    c.fill = GridBagConstraints.BOTH;
    c.weightx = 1.0;
    c.weighty = 1.0;
    gridBag.setConstraints(pad, c);
    add(pad);

    validate();
  }

  public void startInFrame(MirrorFrame f)
  { init();
    start();

    f.setTitle("Private Event Demo Hablet");
    f.add("Center", this);
    f.resize(300,300);
    f.show();
  }

  public void marshallSelf(MarshallOutput out)
throws IOException
  { out.writeUTF(pad.getText());
  }

  public void unmarshallSelf(MarshallInput in)
throws IOException
  { pad.setText(in.readUTF());
  }
```

Hchat Example

This example consists of `Message.java`, `SimpleChat.java`, and `Hchat.java`.

Message.java

```
package Hchat;

import java.lang.*;
import java.io.*;
import ncsa.habanero.Marshallable;
import ncsa.habanero.streams.*;

public class Message implements Marshallable
{
  private String strMessage;
  private int nValue;

  public Message (String str, int n)
  { strMessage = str;
    nValue = n;
  }

  public Message ()
  { strMessage = null;
    nValue = -1;
  }

  public int getField ()
  { return nValue;
  }

  public String getMessage ()
  { return strMessage;
  }

  // Writes out current state
  public void marshallSelf (MarshallOutput out) throws IOException
  { out.writeUTF(strMessage);
    out.writeInt(nValue);
  }

  // Reads in current state
  public void unmarshallSelf (MarshallInput in) throws IOException
  { strMessage = in.readUTF();
    nValue = in.readInt();
  }
}
```

Hchat.java

```
package Hchat;

import Hchat.*;
import ncsa.habanero.*;
import ncsa.habanero.streams.*;
import java.io.*;
import java.awt.*;
import java.lang.*;

public class Hchat extends SimpleChat implements Wrapped
{
  // Overrides the super's action so that Habanero can call it manually.
  public boolean action (Event e, Object arg)
  {
    if (arg.getClass().getName().equals ("java.lang.String")) {
      if (e.target == in1)
    {
      e.arg = new Message (in1.getText(), 1);
    }
      else if (e.target == in2)
    {
      e.arg = new Message (in2.getText(), 2);
    }
    }
    return false;
  }

  // Sets up the window and calls super's init.
  public void startInFrame(MirrorFrame f)
  { init();
    start();

    f.setTitle("Chat Hablet");
    f.add("Center",this);
    f.resize(500,300);
    f.show();
  }

  // Writes out current state for new user.
  public void marshallSelf (MarshallOutput out) throws IOException
  { out.writeUTF(out1.getText());
    out.writeUTF(out2.getText());
  }

  // Reads in current state for new user.
  public void unmarshallSelf (MarshallInput in) throws IOException
  {
    if (out1 == null)
      init();
    out1.setText(in.readUTF());
    out2.setText(in.readUTF());
  }

  // Like a distributed action().
  // It receives events from remote app.
  public boolean doEvent (Event evt, Object arg)
  { TextField tf;

    if (arg instanceof Message)
    {
      switch (((Message)arg).getField())
      { case 1:
          out1.appendText(((Message)arg).getMessage() + "\n");
          in1.setText("");
          return true;
        case 2:
          out2.appendText(((Message)arg).getMessage() + "\n");
          in2.setText("");
          return true;
    default:
      out1.appendText("Didn't get a cases but did get class\n");
          out2.appendText("arg.getClass().getName() is \"" +
            arg.getClass().getName() + "\"\n");
      return true;
      }
    }
    if (arg.getClass().getName().equals("java.lang.String")) {
      out1.appendText("Ignoring " + arg);
```

```
      return true;
    }

    out1.appendText("Got an unrecognized event\n");
    out2.appendText("evt.target.getClass().getName() is \"" +
      evt.target.getClass().getName() + "\"\n");
    out2.appendText("arg.getClass().getName() is \"" +
      arg.getClass().getName() + "\"\n");
    return true; //false;
  }
}
```

SimpleChat.java

```
package Hchat;

import java.awt.*;
import java.applet.Applet;

public class SimpleChat extends Applet
{ TextField in1, in2;
  TextArea out1, out2;
  Label l1, l2;

  public void init()
  { // Prepare components
    in1 = new TextField(40); in2 = new TextField(40);
    out1 = new TextArea(60,40); out1.setEditable(false);
    out2 = new TextArea(60,40); out2.setEditable(false);
    l1 = new Label(); l1.setText("User 1 type here:");
    l2 = new Label(); l2.setText("User 2 type here:");

    // Add components
    GridBagLayout gb = new GridBagLayout();
    GridBagConstraints c = new GridBagConstraints();
    setLayout(gb);

    c.fill = GridBagConstraints.BOTH;

    c.weightx=1.0; gb.setConstraints(l1,c);
    add(l1);

    c.gridwidth=GridBagConstraints.REMAINDER;
    gb.setConstraints(l2,c); add(l2);

    c.gridwidth=GridBagConstraints.RELATIVE;
    gb.setConstraints(in1,c); add(in1);

    c.gridwidth=GridBagConstraints.REMAINDER;
    gb.setConstraints(in2,c); add(in2);

    c.weighty = 2.0;
    c.gridwidth=GridBagConstraints.RELATIVE;
    gb.setConstraints(out1,c); add(out1);

    c.gridwidth=GridBagConstraints.REMAINDER;
    gb.setConstraints(out2,c); add(out2);

    validate();
  }

  public boolean action(Event e, Object arg)
  {
    if (e.target == in1)
    { out1.appendText(in1.getText());
      out1.appendText("\n");
      in1.setText("");
      return true;
    }
    if (e.target == in2)
    { out2.appendText(in2.getText());
      out2.appendText("\n");
      in2.setText("");
      return true;
    }
    return false;
} }
```