The Command class for the Java AWT

Jan Newmarch,

University of Canberra
PO Box 1, Belconnen
ACT 2616, Australia
email: jan@ise.canberrra.edu.au
Phone: +61 6 201 2422

Abstract

Java is a relatively new OO language, with a set of libraries used to extend behaviour. The AWT library allows GUI programming using a particular event-driven model. This paper describes an alternative way of handling events which is based on the `Command` class of the book "Design Patterns" by Gamma, Helm, Johnson and Vlissides. The `Command` class is built on top of the standard AWT event handling. The paper discusses implementation aspects of this alternative event model for the Java AWT. While this can be built reasonably easily, there are some issues in language and environment that are not completely satisfactory.

Introduction

Java [Arnold and Gosling] is a relatively new object-oriented language that is becoming very popular for a variety of reasons. It offers platform independence, the ability to be shipped around the network, multimedia effects, as well as being a simple and clean language.

Java is defined at two levels: the language specification ties down syntax and to a large extent semantics [Goslinga]. In addition, there is a large set of ‘‘standard’’ libraries [Goslingb]. While these have not not been formalised to the same extent as the language - and indeed are still evolving - the majority of applications will use them. The extent to which these libraries conform to and present good patterns will affect the acceptability of the language as a whole.

One of the sets of libraries, the AWT (Abstract Windowing Toolkit), allows programming using graphical user interfaces. This is built upon an event model. This library has suffered a fair amount of criticism, and in particular the event model used has proven to be difficult to understand, poorly structured and full of bugs [Newmarch96a, Newmarch96b]. The model is discussed in more detail later.

The book ‘‘Design Patterns’’ [Gamma] discusses a number of patterns, and sketches implementations for languages such as C++. It is of course true of patterns that a complete implementation cannot usually be given except in the context of a particular application or application domain. In general, such complete implementations would be quite specialised and not of wide interest. However, in some cases the application domain may be wide enough to make an implementation useful for a large number of problems, and in such cases the existence of an implementation would be quite useful.

This paper discusses how the different event model - the `Command` class - proposed by Gamma *et al* [Gamma] can be built on top of the Java AWT event model. This removes a large number of the problems of the AWT model, but not completely. While some features of the language aid in implementing the `Command` class in Java, other features have worked against it and lead to some inelegancies.

Event models

Systems using graphical user interfaces generate events in response to user actions. For example, clicking on a `Button` will usually generate a ''Button Click'' event or similar. Applications have these events forwarded to them in some manner and must respond in an *event driven* way to these events.

The event loop which catches and dispatches events may be explicitly visible and under application control, or may be hidden in some way. For example, Xlib [Scheifler] allows an application to sit in an event loop, catching each event and branching on event type. Within each branch application-specific code is invoked. Microsoft Windows acts in a similar way. This is a strictly procedural approach.

Motif and other Xt-based toolkits [Asente and Swick] allow application-specific code to be attached to *callback* functions. This is more OO in approach in that it attaches application code to the application objects. The event loop is still there, in the function call `XtAppMainLoop()`, but this hides the details of event dispatch. Applications need only worry about setting up callback functions. However the callback method does suffer flaws in that it tightly couples GUI objects to application code. This makes it harder to separate view from model. It also makes code organisation quite messy, with application code being easily mixed up with GUI code.

The Command class approach has been used in systems such as Interviews [Linton]. It attempts to divorce GUI code from application code by placing the application code in separate command objects, where each GUI object has a command object installed. The GUI object invokes suitable methods in the command object when events of interest affect the GUI object. The separation of GUI and application code allows GUI elements to be changed without having to make corresponding changes in the application code.

The AWT event model

The AWT toolkit is an OO toolkit - how does it fare for event processing? While it is a superbly crafted toolkit in many ways, it seems to have adopted a weak OO approach in event processing.

There is a simple level of understanding and programming the AWT event model. This level leads to various programming styles, none of which are particularly ''good'' OO. This is discussed in the rest of this section. There is also a great deal of complexity going on between the different layers of the AWT, and this is discussed in a later section.

Graphical user objects consist of containers and primitive objects, based on an application defined windows hierarchy. For example a `Frame` will be a toplevel AWT window which may hold a container such as `Panel` which may contain a primitive object such as `Button`.

Events are generated by the user performing actions such as clicking the mouse within a `Button`. The event loop is completely hidden from the programmer in the AWT. An event will be generated in a manner that is invisible to the object. Suffice it to say (for now) that upon occurrence of a user action an event is generated and the method `postEvent()` is called for the object the event occurred in.

An object's event handler can choose to pass the event or block it. The default event processing is that each object passes the event by sending it to its parent in the window hierarchy. This leads to a Chain of Responsibility: the `Button` passes the event, then the `Panel` passes the event, and finally the `Frame` passes it. This chain can be broken by a GUI object choosing to block sending it to a parent.

An object can override the default event handler. This allows it to perform various activities, in addition to deciding whether or not to pass the event on to a parent.

This leads to two common ways of handling events within the standard AWT: leave them all to the top-level `Frame` or `Applet` to manage from `handleEvent` (or some of the convenience methods), or to subclass each object that generates an event and let them manage their own events.

The first of these leads to application code lying in the `Frame` (or `Applet`), and the `Frame` (or `Applet`) must be aware of all the details (such as object names or the objects themselves) for all of the GUI elements. This gives a very rigid and inflexible design. For example, typical code looks like

```
public boolean handleEvent(Event evt) {
    if (evt.id == Event.ACTION_EVENT) {
        if (evt.target == Button1)
            // handle Button1
        else if (evt.target == Button2)
            // handle Button2
        // etc
    } else if (evt.id == Event.KEY_PRESS) {
        if (evt.target == TextArea1)
            // handle TextArea1
        // etc
    }
}
```

While this works ok for applications with a small number of GUI objects, it rapidly becomes unmanageable for large numbers.

The second method leads to a large number of ''thin'' classes each with a single method that carries the application code. While this is better OO, it still ties the application code fairly closely to the GUI objects. For example, for each Button in the application one must subclass it in this manner:

```
class MyButton extends Button {

    // duplicate constructor
    MyButton(String label) {
        super(label);
    }

    public boolean action(Event evt, Object what) {
        // code to handle mouse click
    }
```

```
    // no other methods!
}
```

The Command class model

The Command class model separates GUI objects from application behaviour by placing application behaviour in subclasses of `Command` objects. For example, when the application wishes a file to be saved it should call on the `FileSaveCommand` object to perform this action, instead of making GUI objects such as a `Frame` or a `MenuItem` do this.

Each Command object has a method `execute()` that can be invoked to perform this application-specific code.

A GUI object does not perform application-specific code itself. What it does is to ‘‘install’’ a Command object. When an event of interest to the GUI object occurs it invokes the execute method of its Command object.

This avoids both of the problems mentioned earlier. Firstly, the amount of detailed GUI knowledge in some toplevel object is completely removed, so that changing the GUI components can be done much more flexibly. Secondly, the problem of ‘‘thin’’ GUI elements that only exist to implement application code is removed, being replaced by application specific objects.

This allows Command objects to be written more or less independently of GUI objects. The implementation of both the GUI code and the application code can then be varied independently as long as they use the same Command objects.

The awtCommand Package

We have implemented a package on top of the AWT event model to bring the Command event model to Java. This section describes the elements and use of this package.

The Command class defines one abstract method `execute()`. This could be implemented either as an abstract class or as a Java interface. (An interface defines only abstract methods and can be multiply inherited by a class that must supply an implementation for all of the abstract methods.) An application will be expected to have a fairly complex class structure of its own. An interface allows the Java ‘‘multiple inheritance’’ model to work well here, so Command is defined as an interface.

Each graphical object has a set of events that it will handle. For example a `List` object will generate `LIST_SELECT`, `LIST_DESELECT` and `ACTION_EVENT` events. There will be a (possibly) different Command object used to handle each of these. The `LIST_SELECT` event will be handled by a `selectCommand` object, the `ACTION_EVENT` event will be handled by an `actionCommand` object, etc.

The awtCommand package subclasses all of the relevant AWT classes. Each class is prefixed with ‘C’ (really, the prefix should be ‘Command’ but that is too verbose). So `CList` is a subclass of `List`, `CFrame` is a subclass of `Frame` etc. Each of these classes has additional methods over the parent class to allow a `Command` object to be attached. These methods have names based on the event-types that they handle.

In order to associate `Command` objects to be associated with awtCommand objects, there is a method to set the `Command` object for each event type. For example, `CList` has additional methods

```
setSelectCommand(Command c)
setDeselectCommand(Command c)
setActionCommand(Command c)
```

When an event occurs for which a `Command` object has been registered, the awtCommand package invokes the method

```
execute(Object target, Event evt, Object what)
```

of the `Command` object. The actual `Command` object will be a class which contains the application code in the `execute` method. Although the Command pattern allows decoupling of GUI and application code, it is not possible to completely separate them. For example, the constructor for a Command object may be used to pass in information about the GUI objects it will use as Observers.

In addition, runtime information may be needed by the Command object. To allow this, the `execute` method is called with three parameters

- `Object target`
  This is the object that called the `execute` method *e.g.* a `CList` object.
- `Event evt`
  This is the event that caused this method to be run. Note that the target object of the event may be different if event handling was deferred to a parent.
- `Object what`
  This is an additional parameter that depends on the target and on the event. For example, for a `CButton` with the `Command` object being the `actionCommand`, `what` is the `CButton`'s label. For a `CList` with the `Command` object being the `selectCommand`, `what` is the index of the item in the `CList` that was selected.

These allow a sufficient amount of runtime information to be available if needed.

Example 1

This is a trivial example that puts a `CButton` in a `Frame` (a `CFrame` would do just as well). When the `CButton` is activated, the `execute` method of the associated `Command` object is run. This justs prints some information.

```
import java.awt.*;
import java.awtCommand.*;

public class HelloWorld extends Frame {

    public static void main(String argv[]) {
        new HelloWorld().show();
    }

    HelloWorld() {
        CButton b = new CButton("Hello World");
        b.setActionCommand(new HelloCommand());
        add("Center", b);
        resize(100, 100);
```

```
        }
    }

    class HelloCommand implements Command {
        public void execute(Object target, Event evt, Object what) {
            System.out.println(target + " " + evt);
        }
    }
```

Example 2

This example is a little less trivial, and shows how a single `Command` object can be used for two distinct user actions.

The application shows a list of colors next to a label. When one of the colors is selected the label's foreground is changed to that color.

Items in the list can be chosen in two ways: by selection with the left mouse button or by using the up/down keys to traverse the list with selection by pressing the Return key. These generate different events: mouse selection generates a LIST_SELECT while pressing Return generates an ACTION_EVENT. A single `Command` is used to handle both of these by using it as the `actionCommand` and as the `selectCommand`.

When the `execute` method runs, the `what` field is used to find the selected data. This may be of different types, though. From the ACTION_EVENT, the value is the item selected as a `String`. From the LIST_SELECT the value is the index of the item selected as an `Integer`. A runtime test distinguishes between them.

The Command object does not need to know any of the details of the GUI interface. All it needs to do is to call a method `setColor()` of some suitable GUI object, and that will look after the GUI-side details. No other details of the object are required. This is best done in Java terms by defining an interface `Colorable` with one abstract method `setColor()`. An object implementing this interface is passed into the Command object's constructor for later use.

```
import java.awt.*;
import java.awtCommand.*;

interface Colorable {
    public void setColor(Color c);
}

class ColorList extends CFrame implements Colorable {
    final Color colors[] = {Color.red, Color.blue, Color.green};
    final String colorLabels[] = {"red", "blue", "green"};
    Label label;

    public static void main(String argv[]) {
```

```
        new ColorList().show();
    }

    ColorList() {
        // a CList showing the color choices
        CList list = new CList();
        for (int n = 0; n < colors.length; n++)
            list.addItem(colorLabels[n]);

        // a Command invoked on button select and Return key
        ColorCommand command = new ColorCommand(this, colorLabels);
        list.setSelectCommand(command);
        list.setActionCommand(command);

        label = new Label("Hello World");

        // set geometry
        add("West", list);
        add("Center", label);
        resize(300, 100);
    }

    public void setColor(String color) {
        int n;

        // search for Color matching string
        for (n = 0; n < colorLabels.length; n++) {
            if (colorLabels[n].equals(color)) {
                break;
            }
        }
        label.setForeground(colors[n]);
    }
} // end ColorList

class ColorCommand implements Command {
    Colorable app;
    String colorLabels[];

    // Constructor stores local info
    ColorCommand(Colorable app, String colorLabels[]) {
        this.app = app;
        this.colorLabels = colorLabels;
    }

    public void execute(Object target, Event evt, Object what) {

        if (what instanceof String) {
            // got here as actionCommand
            String color = (String) what;

            app.setColor(color);
        } else {
            // got here as selectCommand
            int index = ((Integer) what).intValue();

            app.setColor(colorLabels[index]);
        }
    }
} // end ColorCommand
```

Complexities of the new AWT Event Model

One of the confusing points in using the AWT event model, and one of the things that complicates adding a Command class layer is that there are really three AWT event models, with a gradual change from one to another. The full details of them are given in [Newmarch96b].

The old model treated events as read-only, although nothing in the implementation enforced this. It was expected that as soon as a GUI object handled an event, it would block it from passing to its parents. This is a standard Chain of Responsibilty.

The new model was introduced to allow ‘‘filtering’’ of events by Text classes. For example, an UpperCaseTextEntry object would translate all characters typed into uppercase, while a Password object would suppress display while maintaining a record of characters typed. The new model allows such filtering in two ways

- An event may be modified to change fields such as the character value.
- An event which is blocked never reaches the native GUI object.

The new event model allows changes to some event fields. It also allows events to be discarded so that - in effect - they never occurred.

The awkward part is that this has not been fully implemented, so that some events are handled purely by the new model, buts others by a mixture of new and old models. Presently, KEY events are subject to this filtering process, but no others are. The most extreme case that I have come across so far is that of handling `<Return>` in the single-line text editor `TextField`. When typed by the user it is initially treated as an ordinary character. This can be changed to another character, or blocked from affecting the native GUI object. Neither of these should be done, and it must be passed through unchanged. The reason for this is that when it reaches the native GUI object it is used to generate an ACTION_EVENT which can be passed or blocked without any further effect on the native object.

This is a real mess.

New Model Effect on Command Implementation

In implementing the Command class, attention must also be paid to the Chain of Responsibility in passing on events, and the fact that this can make a difference to native GUI behaviour in some cases (the KEY events).

An earlier version of the awtCommand package was based on the old AWT event model, and in this as soon as an event was dealt with by a Command object the event was blocked. This followed exactly the Chain of Responsibility pattern as well as the Command pattern.

The possibilities under the new model are

- Don’t pass on events after handling them. This is incorrect for KEY events, but makes no difference for other events.
- Pass on KEY events always, but don’t pass on other events after handling them.

- Always pass on events. This removes the possibility of blocking KEY events.
- Allow the Command object to decide whether or not to pass on events.

The first choice will result in Text objects never getting KEY events, which is clearly wrong behaviour. The second is inconsistent in that it gives two Chain of Responsibility models, one which stops on handling and one which doesn't. The third is only wrong in one case. The fourth choice is the only one that allows full and correct use of the new AWT event model. Despite this, the third has been used in the current implementation of the awtCommand package. That is, all events are passed on to the parent, whether handled or not.

The Command class is intended to reduce coupling between application and GUI. If a Command object had to decide whether or not an event should be passed on or blocked based on the *type* of the event then it would increase coupling. In addition, if we look at examples where blocking of events is needed, they tend to be situations where the Command class is insufficient anyway.

The major example of blocking events from reaching native GUI objects is complete suppression of input for password entry. The object should suppress display of characters typed, but should maintain a copy of all characters typed. The basic object required here is a HiddenTextField object, with actual checking of password validity being performed by application code. If we concentrate on the HiddenTextField then I would argue that this is really a new GUI object which does not involve application code. To see this, consider the method `getText()`. For the standard AWT `TextField` this returns the text actually showing. For the HiddenTextField it would need to return the string collected but not showing. So it would need to be a subclass of `TextField` which not only overrides the event handling but also `getText()`.

The problem of passing on events or blocking them is best answered at the AWT level by adding a HiddenTextField object that subclasses the standard `TextField`. The Command object can then be added by subclassing this just like the other objects.

This solution is not completely satisfactory. It could, for example, be broken by Sun migrating more objects to this new model. Difficulties in actually doing this make it seem unlikely to be a problem.

Within each awtCommand object an event handler takes the AWT event and invokes a Command object if possible. For example, In a `CList`, selection of an item is handled by

```
public boolean action(Event evt, Object what) {
    if (actionCommand != null)
        actionCommand.execute(this, evt, what);
    return false;
}
```

Comments on code reuse

The Java design allows single inheritance but not full multiple inheritance. The interface mechanism does not allow code reuse but instead enforces specification implementation. There is multiple inheritance for interfaces, but as all methods are abstract this avoids the problems of full multiple inheritance.

There are several instances of code duplication in the awtCommand caused by these mechanisms, which are very definitively not code reuse. These occur in the common code used in CCanvas, CDialog, CFrame, CPanel and CWindow. They all handle a large common set of events.

The awt toolkit does not use code duplication. How this is done is intriguing: (under X) they all create their native window windows by calling a C function `awt_canvas_create` that creates a DrawingArea and installs event handlers for events. Code is shared by calling a shared function that lives in its own file, canvas.c. Code sharing is performed at the native level by calling a common global function. This of course is not possible at the Java level.

In the awtCommand toolkit, `CCanvas` is a subclass of `Canvas`, `CDialog` is a subclass of `Dialog`, etc. They can only inherit methods from their superclasses. But they all need to implement a large number of methods such as `mouseUp`, `scrollAbsolute`, `action`, etc, as well as the corresponding methods to set these command objects.

Multiple inheritance would certainly be a means of sharing code. This is a rather heavyweight mechanism, though. A simple `#include` mechanism would be more than adequate to solve the code sharing problems encountered in building this toolkit. To avoid complex dependencies possible in the C include mechanism, such include's should not be nested.

An alternative way is to use an associated object, much like the AWT toolkit uses peer objects. Jean-Michel Leon sent me code to do this, and I would have used it apart from one factor: it places the documentation in the peer object, and this would make it hard to find using the automatic documenting system used for Java libraries. This points to another small problem with Java: the documentation system is unable to track through optimisation mechanisms.

The problem of a good solution to code-sharing deserves more investigation than is possible here.

Status

This library is version 2.0, released in May 1996. It uses the AWT JDK 1.0.2 release. This software is available free of charge (and free of warranty) by anonymous ftp as a tar gzip file from ftp://ftp.canberra.edu.au/pub/motif/command/command.2.0.tar.gz

Conclusion

The AWT event model can have the Command pattern layered on top without too much trouble. However, the event model currently used in the AWT does have inconsistencies that lead to some problems in the Command pattern.

Nevertheless, the Command pattern does resolve major problems in using AWT events and leads to more maintainable programs with better separation of GUI and application code.

Java has a weak multiple inheritance model in that multiple implementations cannot be inherited, only multiple specifications. Some cases within this implementation require code sharing in addition to inheritance from a GUI object. This is not possible at present, so code is duplicated. Better mechanisms are needed to avoid this.

References

[Arnold and Gosling] K. Arnold and J. Gosling ‘‘The Java Programming Language’’, Addison-Wesley, 1996

[Asente and Swick] P. J. Asente and R. R. Swick ‘‘X Window System Toolkit’’, Digital Press, 1990

[Gamma] E. Gamma, R. Helm, R. Johnson and J. Vlissides ‘‘Design Patterns’’, Addison-Wesley, 1995

[Goslinga] J. Gosling, B. Joy and G. Steele ‘‘ The Java Language Specification’’, Addison-Wesley, 1996

[Goslingb] J. Gosling, F. Yellin and the Java Team, ‘‘The Java Application Progamming Interface’’, vols 1 and 2, Addison-Wesley, 1996

[Linton] M. Linton, P. Calder, J. Interrante, S. Tang, J. Vlissides ‘‘InterViews Reference Manual’’, CLS, Stanford University, 1992

[Newmarch96a] J. D. Newmarch, ‘‘The Java AWT: Events’’, X Advisor, February 1996, http://landru.unx.com/DD/advisor/TOC/v2n2.shtml

[Newmarch96b] J. D. Newmarch, ‘‘The Java AWT: The New Event Model’’ X Advisor, June 1996, http://landru.unx.com/DD/advisor/TOC/v2n6.shtml

[Scheifler] R. W. Scheifler, J. Gettys and R. Newman ‘‘X Window System - C Library and Protocol Reference’’, Digital Press, 1988