A Study of Automatic Migration of Programs Across the Java Event Models

Bharath Kumar M, R Lakshminarayanan, Y.N. Srikant
Department of Computer Science and Automation
Indian Institute of Science
{mbk,lakshn,srikant}@csa.iisc.ernet.in

Abstract

Evolution of a framework forces a change in the design of an application, which is based on the framework. The same is the case when the Java event model changed from the Inheritance model to the Event Delegation model. We summarize our experiences when attempting an automatic and elegant migration across the event models. Further, we also necessitate the need for extra documentation in patterns that will help programs evolve better.

Introduction

When Java announced a change in the event model for their AWT framework, several developers had to rewrite and reorganize large amount of code manually, to be in conformance with the new model. This motivates the need for an automated and elegant migration solution. A good solution to the migration problem should have to account for the various features which were present in the 1.0 model. Further it should also allow the migrated system to behave as if, it is designed using the 1.1 event model. In pursuit of an elegant migration which is universally applicable we have identified some design decisions that were facilitated by the 1.0 event model's semantics. Our emphasis in this paper is on the way these design decisions affect the migration mechanism. First we provide a brief summary of the event model.. . Then, we list some design decisions that were possible in the 1.0 model and their implications. Then, we discuss how a robust migration mechanism would address these issues.

A summary of the event models

The event models are essentially mechanisms for associating callbacks with events, and execution of the callbacks, when the event occurs. Java 1.0 followed the Inheritance model and Java 1.1 follows the Event Delegation model. We can view the two as patterns for event delivery.

The Java 1.0 event model

The model for event processing in version 1.0 of the AWT[1] is based on inheritance. For a program to catch and process GUI events, it must subclass GUI components and override either *action()* or *handleEvent()* methods. Whenever an event occurs, the event percolates down the inheritance hierarchy ending in a call to the event target's *action()* or *handleEvent()* method. We call the component on which the event has occurred as the event target. The event target processes the event through methods called from the *action()* or *handleEvent()* method. If the component is sure that all the processing for that event is done, it will return a "true", else it will return a "false". Returning a "true" consumes the event and it is not processed further; otherwise the event is propagated sequentially up the GUI hierarchy until either it is consumed, or the root of the hierarchy is reached. When an event is returned to the component's parent, it processes the event, and will return a "true" or a "false." The relationship between the various classes is given in Figure 1.

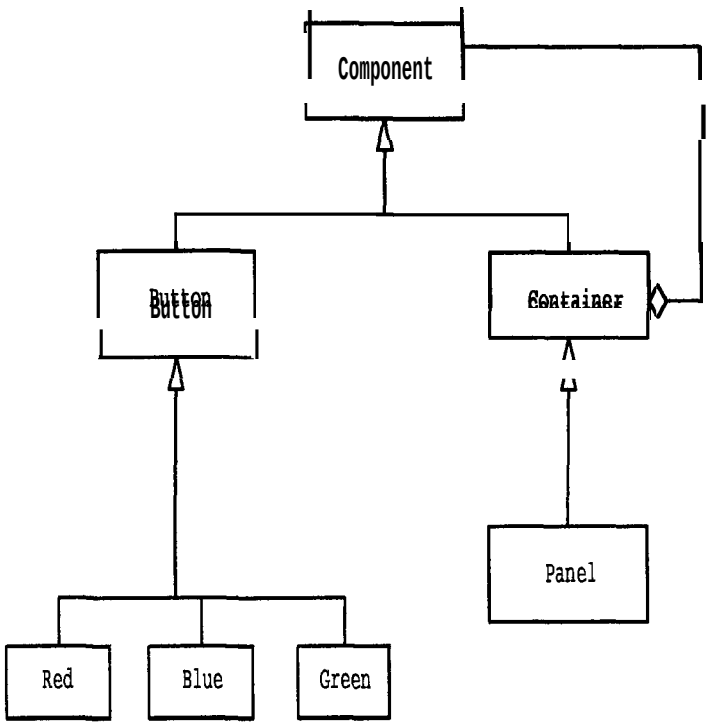

Figure 1: Class Diagram

In Figure 2 the component *Blue* is the event target, and the component Panel is its parent. When an event occurs on the component Blue, the event is delivered to it via its parent and the overriden method *handleEvent()* of Blue is called, which in turn calls its event processing methods *method1* and *method2*. When *Blue* returns a "false" the event goes to its parent *Panel* and its *handleEvent()* is called, which in turn calls its event processing method *method0*. When the event processing is very trivial, the components typically do it in the *handleEvent()* method itself. The *Green* component does it this way. The result of this model is that programs have essentially two choices for structuring their event-handling code: Each individual component can be subclassed to specifically handle its target events. The result of this is a plethora of classes. All events for an entire hierarchy (or subset thereof) can be handled by a particular container; the result is that the container's overriden *action()* or *handleEvent()* method must contain a complex conditional statement in order to process the events.

The Java 1.1 event model

While the 1.0 model works fine for small applications with simple interfaces, it does not scale well for large java applications for the following reasons:

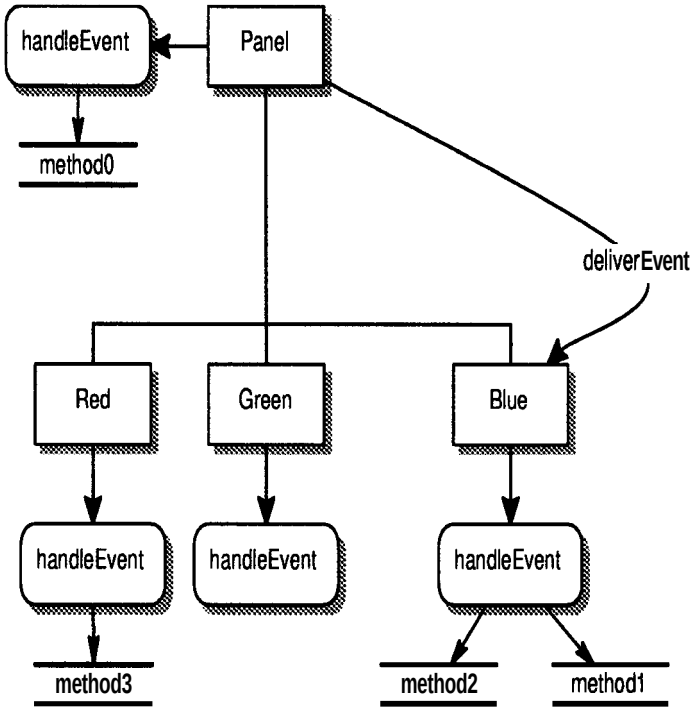

Figure **2:** Java 1.0 Event Mechanism

- The requirement to subclass a component in order to make any real use of its functionality is cumbersome to developers; subclassing should be reserved for circumstances where components are being extended in some functional or visual way.
- The inheritance model does not lend itself well to maintaining a clean separation between the application model and the GUI because application code must be integrated directly into the subclassed components at some level.
- Since all event types are filtered through the same methods, the logic to process the different event types is complex and error prone.
- There is no filtering of events. Events are always delivered to components regardless of whether the components actually handle them or not.
- This is a general performance problem, particularly with high-frequency events such as mouse moves.

The 1.1 event model was introduced to solve the above mentioned problems and to provide an efficient framework to support complex programs[2].

The following list includes the essential elements of AWT that are changed in the **1.1** event model:

- Event model – Events do not percolate up the container hierarchy as before; instead, interested listeners register themselves with AWT components, and events are multicast through *listener* interfaces.
- Event methods – A single monolithic Event class is no longer used for the delivery of all events; instead, different event classes are derived from *java.util.EventObject* (or, with respect to the AWT, *java.awt.AWTEvent*) and provide an appropriate interface to the relevant occurrence.
- Event methods – Events are no longer delivered to components through the traditional *handleEvent()* method; instead, a *processEvent()* method is used along with various associated helpers.
- Event masks – Event masks of the events that a particular component would generate are now maintained. This new approach is more efficient because a particular event will not be generated and processed if no target is listening for it. As a result, if you subclass a component, then it will, by default, not generate the usual AWT events. If you wish to receive certain events, you must explicitly flag those event types that you wish generated.
- Method names – Many methods have been renamed to achieve a more consistent interface.

The Java 1.1 event model, given in the Figure **3,** differs from the 1.0 event model given in Figure **2** both semantically and structurally. Here, though the event does trace the same path through the hierarchy, it is not returned to the parent after being processed. Instead every component is an observable and has a set of observers who state their interest in a particular event. Thus, a list of observers is maintained for every event in every component. When the event occurs, the message is multicast to each one of the observers. Several advantages like flexibility and dynamics of such a model have made Java switch over to this event model.

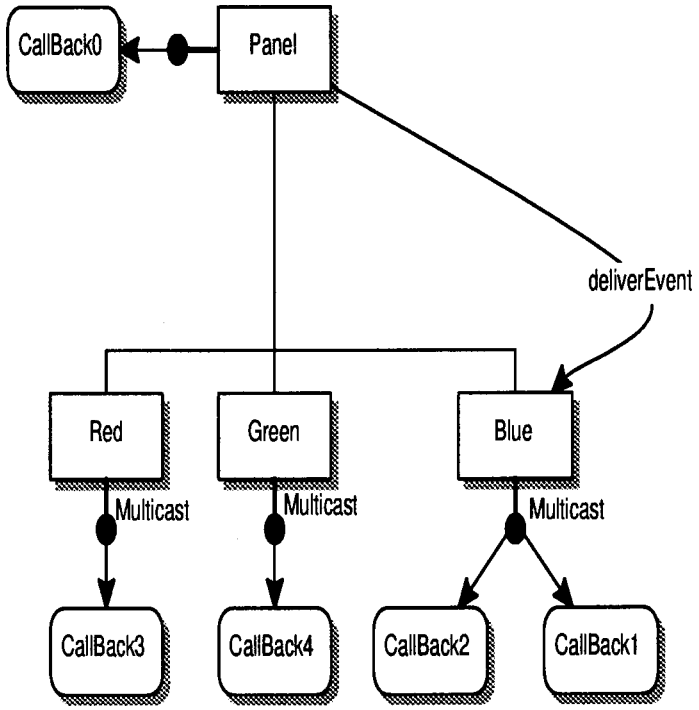

Figure **3:** Java **1.1** Event Mechanism

The Migration Problem

The difference in the event models caused incompatibility in the code and design written for different versions of Java. People were forced to change their designs manually as a consequence. First let us look at the reasons for opting for a migration to the 1.1 event model:

- The 1.1 version of AWT has some new features an application might want to use and exploit.
- The new architecture enables faster and more robust implementations of the AWT, which means that the updated application might work better.
- Support for the old API will eventually be phased out.

The solution to the migration problem has to account for the various features which were present in the 1.0 event model. Further it should also allow the migrated system to behave as if, it is designed using the 1.1 event model. This would make the components in the migrated system compatible with other components based on the 1.1 model, and allow us to leverage the new features provided by the 1.1 event model.

How should a good migration scheme be?

For a huge program that was written using the older event model, there are several places where the design and code need to be changed. Since event handling is tightly bound to a component, we might be looking at changing code potentially for each and every component in the program. Since the labor can be quite huge an effective mechanism will be required. If the mechanism can be completely automatic, well and good. In our view we could see two possible approaches.

1. Preserve the component hierarchy and simply make the old event handlers as listeners of events. This can be easily achieved by making class definitions implement the event handling listeners and changing the function signatures as required. The advantage with such an approach is that one can possibly attain perfect automatic migration. But the migration scheme does not reflect any of the rationale with which the event model itself was changed. The newer event model, essentially separates the event handlers from the event targets. But retaining the event handlers as part of the component and making them listeners makes the class bulky and also restrains potential reuse of event handling code. So, an approach that tries to separate the two (the event targets and the event handlers) will be better off on the long run.
2. Separate the event handlers from the event target to form a separate class that implements the event listener and register it as a listener with the instance of the component. Though this method complicates matters, it is still a viable alternative, because it reflects the rationale behind the newer event model. In some applications like GUI builders such a change is mandatory lest all new code will reflect the older class structure.

When we are attempting a universal migration scheme, we should also make sure that some of the design decisions that were facilitated by the constraints of the older model are still preserved. The constraints in the older model might have allowed the designer to take some liberties during design. If the migration scheme does not address them, then the correctness of the program itself might be lost. So, next we try to give some possible design decisions that were facilitated by the older model, and then see what difficulties they pose to the migration scheme.

Design Decisions facilitated by the Inheritance Model

Adherence to an event model provided by a framework imposes a set of constraints. For example, in the inheritance model, the name of the callback which is called when a particular event occurs is fixed, so also are the parameters to the function. One has to inherit from a component class to write the event handler. At the same time, adherence to an event model also facilitates some design decisions that are based on the constraints provided by the model. Here we list some possible design decisions that are facilitated by the Inheritance Event Model.

1. The event handler operates in the context of the event target. Since, in the inheritance model, to define an event handler, we specialize the component, the event handler method has access to all the protected and public members of the component, and might use those methods during its own execution.
2. Actual needs to specialize a component might be coupled with the response to the event. There might be a case where a component requires a different look in the application. But since it is the only component which looks like that, we can as well code its event responses in the same class itself.
3. Post Conditions/ Post Processing of Events in the Container. In the Inheritance model once an event has been handled by the component who was responsible for the event, the component reserves an option to return "false" and thus make the default event handler of the container to handle the event. This facilitates the designer to impose some post conditions common to all components in the container, through the default event handler in the container.

   For example, suppose that we need to validate the contents of all the text boxes in a frame, (say to be legitimate words from a dictionary), then we can write the code for validation in the Container's event handler (say for keypress). Thus each time the contents of the text boxes changes, we can validate the contents. This design decision seems viable considering that no additional function calls need to be made from inside the components' event handlers. Given that the programs' designer knows how the event model works, coding such logic in the container's event handler is perfectly fine. Similarly, if we have a lot of components with same be-

havior in a container, then the common behavior (or response to events) can be coded in the container's event handlers. For those components where the behavior is different, we can simply "return (true).''

A scheme for migration

Upon close analysis of the two event models from an architectural perspective, we can notice a transition from the Document-View pattern to the Model View Controller pattern[5]. So, it only follows that a good migration scheme should reflect that rationale. So, as far **as** possible, the transformation should try to separate the event handlers from the components. This approach is in contrast with the transformation suggested by the official document[3], which suggests the components to implement the event listeners and register the methods **as** callbacks. Though the latter case is easier to achieve, on the long run, we lose out on the advantages of decoupling. Moreover, the program will still be left with several specialized components which are simply implementing event handlers. So, first we introduce a simple migration where we separate the event handlers from the component.

A Simple Case

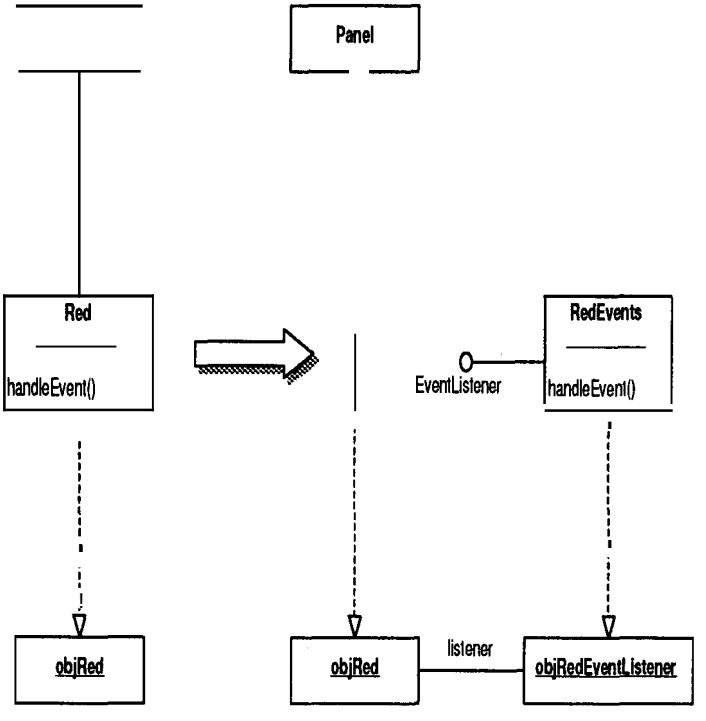

Figure **4:** A simple transformation

Figure **4** gives an example of one such transformation. This transformation can be applied to simple cases where the following conditions hold:

- The component subclass was created only to define event responses.(No other functions are defined there)
- The event responses do not access any protected members of the component class.
- The event responses do not "return(false)"

In the Java 1.0 model, the class *Red* [Fig. **4**] is a subclass of *Panel* and defines a set of event responses, *objRed* is an instance of Red, which is used in the program. We separate the event responses from the class *Red* and build a separate class called *RedEvents* which implements an *EventListener* interface. Since, the event responses are removed from the component class, there is no need to subclass *Panel* and hence *Panel* directly instantiates *objRed*. *objRedEventListener* is an instance of *RedEvents* and is registered **as** a listener with *objRed*. All necessary name changes like changes in the function signature can be easily made. The transformation ensures that the classes and objects conform to the Java 1.1 event model.

Effect of the design decisions

Actual applications might have exploited the design decisions facilitated by the Inheritance model and hence a straight forward migration may not be possible. i.e. the three conditions stated in the previous section may not hold good. Hence it is necessary to study how the design decisions affect the migration scheme. We should also keep in mind that it is more important to provide a correct migration than an elegant one. So, when it is impossible to accommodate both correctness and elegance, we opt for correctness. We will study each case one by one and see how the migration scheme changes in each case, at times compromising on elegance.

- When the event handler assumes the knowledge of the component, we may find calls to member functions from inside the event handlers. Since, the migration scheme suggested separates the event handlers from the component, we will face accessibility problems. When only public members are accessed, we can change the way the access is made, by specifying a reference to the object. (This is possible because every event handler is sent a reference to the target object as parameter) But when a protected member is accessed, the event handler cannot make a reference to it through the object. So clearly two cases arise.

  1. When only public members are accessed: Identify the members and give appropriate reference to them through the target object.
  2. When at least one protected member is accessed, separating the event handlers from the component will not work. So, it's best to retain the hierarchy, and make the component itself implement the event listener.

  Figure 5 shows the transformation for case 1. Since *countComponents()* is a public member (defined in Container, the parent of Panel), the separation won't cause problems. Figure 6, is the case, where the event handler accesses a protected member function *processEvent()* (defined in Container), separation will render the function inaccessible. So, the component is made to implement the event listener and register itself **as** a listener of the events.

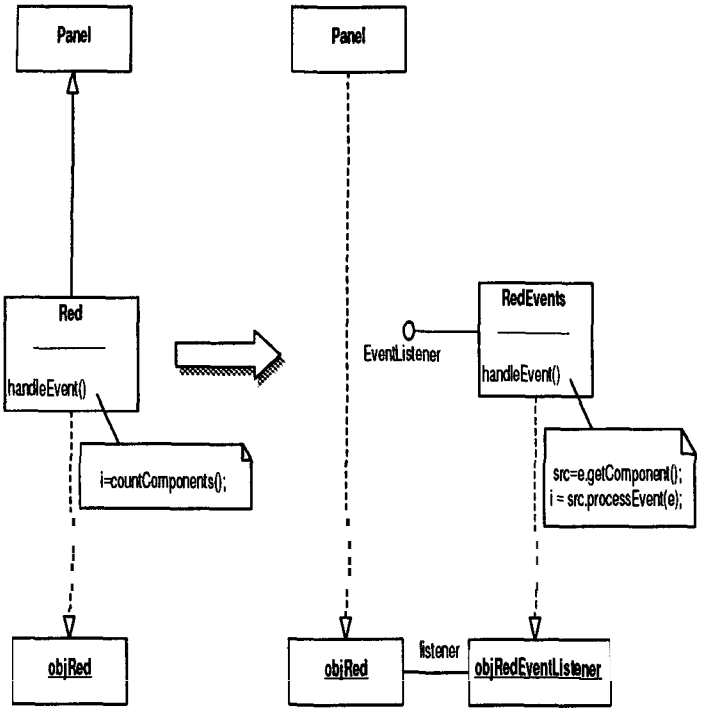

Figure **5:** When only public members are accessed

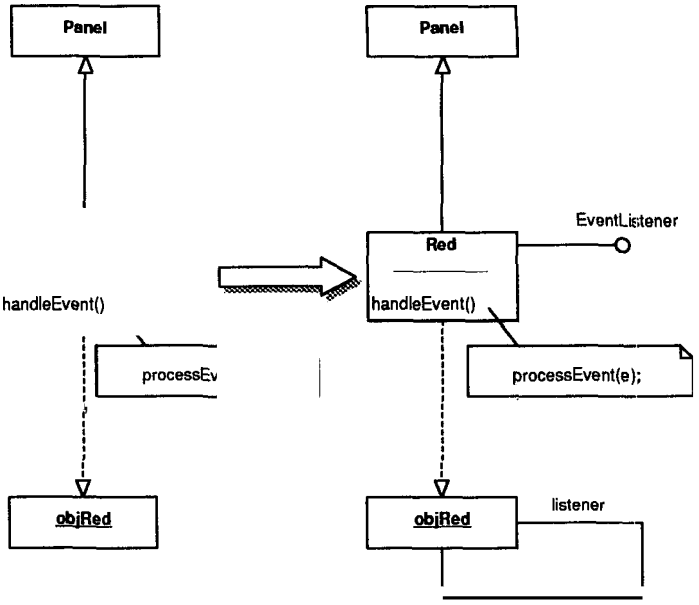

Figure **6:** When protected members are accessed

- When the event handlers are written along with the specialized class, after the separation of the event handler, we should still retain the specialized component class. The event handlers can he separated into a separate class (see Figure **7)** only if no protected, or private members are accessed from the event handlers. Otherwise, the transformation will be similar to Figure **6.**

  In the Figure **7,** in Red, the *pint()* method is redefined. So, we'll have to retain the class. However *eventhandler()* can be separated.

- Accommodating Post Conditions/ Post processing in the container is relatively tougher than the previous two cases. It is perhaps here that the two models differ grossly. The Java 1.1 model does not allow events percolating back up the hierarchy, thus disabling containers to process events. In the earlier model a component could pass the event back to its container (parent) by using "return(false)" in its event handlers. In the new model, event handlers cannot return anything. So, to allow parents to handle events we should manually pass the event to them.

  Upon close analysis, we can notice that the behavior of the events 'percolating up', is a manifestation of the Chain of Responsibility design pattern[4]. So, it has to be manually implemented in the transformation so **as** to accommodate the design decision.

  The Component class provides the *dispatchEvent()* method through which we can explicitly ask the component to handle the event. *dispatchEvent()* notifies the occurrence of the event to its event listeners. This method is redefined in the Container class, where it puts additional logic like locating the event target among its children and then calling the *dispatchEvent()* method of the event target, so as to eventually deliver the event to its target. In our case, the event target has already handled the event, and we wish to explicitly call the parents' event handlers (to emulate the "return(false)" case). If we call the parent's *dispatchEvent()* method, then the method delivers the event back to the component, This would result in a recursive loop. So, we wish to deliver the event to the container **as** if it were a component (thus causing only the container's listeners to be notified). For this, we explicitly cast the container to a component type and make the call. This ensures that the event is passed back up the hierarchy so that the post conditions/post processing can be applied. This is exactly the Chain of Responsibility pattern where we pass the event (responsibility) up the hierarchy so that other components can handle it.

  In Figure 8, we separate the old event handlers to form a class RedListener, which also implements the Eventlistener. The new event handlers, act **as** proxies which conditionally make explicit calls to the parent's event dispatcher. Note that we can separate the old event handlers into a new class only if none of them access any protected member of the component. Otherwise, we will have to preserve the hierarchy and make the component itself implement the event listeners.

  It is important to note that the parent too can make explicit calls to its parent. So, the change *is* made to every component in the hierarchy (wherever required).

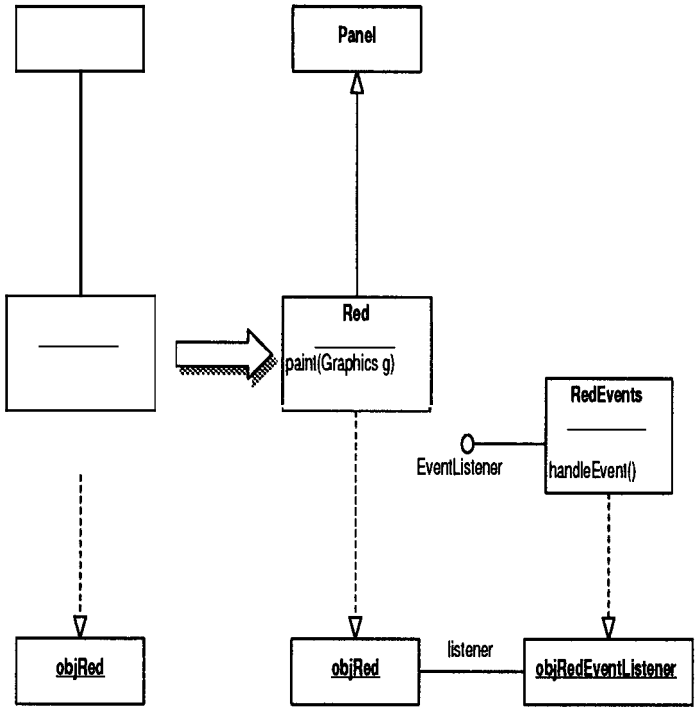

Figure 7 Retaining specialized components

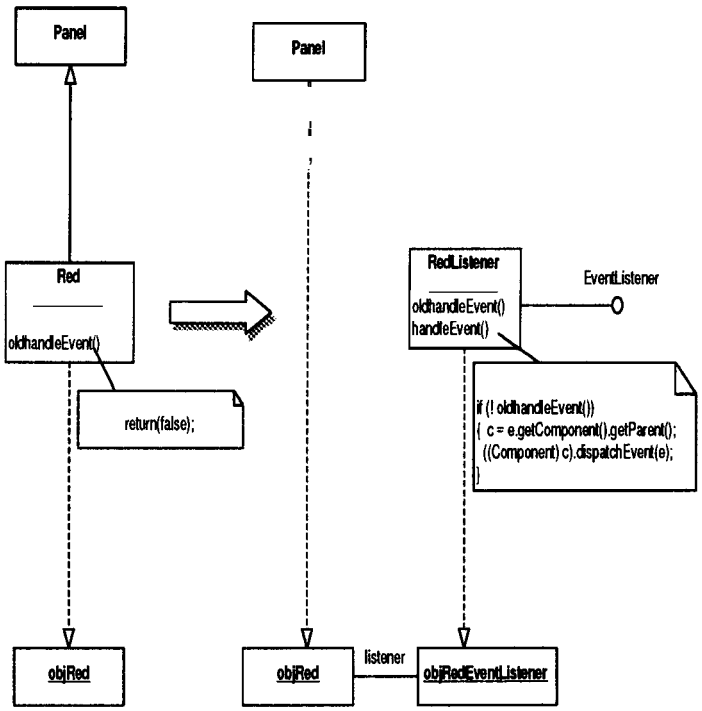

Figure 8: Implementation of Chain of Responsibility

Conclusions

The change in the framework has introduced a very interesting problem. The problem of change/ replacement/ evolution of patterns, and its impact on the design of the application. In the context of the Java's event model, the implications of the change is so pervasive that an elegant migration scheme is not possible. We have shown the various cases that arise due the change in the framework, and implications of the design decisions facilitated by the old model. Further, we have provided a migration scheme that takes into account the various cases.

From a general software design perspective, there are several lessons that can be learned from this exercise of finding a universal migration scheme.

1. When Java went for a change in the event model, with a rationale of separating the event handlers from the components, we feel they should have spent some thought on providing migration schemes that make the resulting code reflect structure of the new model. For this some work was required on accommodating design decisions that were facilitated by the earlier model.

2. This problem also brings up the ramifications caused by pattern replacement. We can look at the problem of migration as simply a case where the Inheritance Model pattern is replaced by Event Delegation Model pattern. The impact of the replacement of a pattern by another is not straight forward. So, when a pattern is documented we need to document possible factors that would exploit the constraints in a pattern. This documentation would help the evolution of the program.

References

[1] Zukowski, John(1997). *Java AWT Reference.* O'Reilly Publications.

[2] *The AWT in 1.0 and 1.1.*, http://www.javasoft.com/products/jdk/awt/index.html

[3] *How to convert programs to the 1.1 AWT API.*, **http://www.javasoft.com/products/jdk/1.**1/docs/guide/HowToUpgrade.html

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object Oriented Design.* Addison-Wesley, 1995.

[5] Buschmann, F., et. al., *Pattern Oriented Software Architecture: A System of Patterns,* Addison-Wesley, 1996.