Courier Patterns

Robert Switzer*
Mathematisches Institut
Bunsenstr. 3-5
37073 Goettingen
Germany

July 24, 1998

*email: switzer@math.uni-goettingen.de

1 Courier Pattern

1.1 Intent

Define an object that encapsulates the interaction of two or three obects. Courier promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

1.2 Motivation

An object-oriented program can be complex, because a large number of classes is involved. Or it can be complex, because the interconnections between the classes are complicated. If such complicated interconnections are implemented by having the classes refer explicitly to one another, the program becomes difficult to understand and maintain but also (and almost worse) the reusability of the classes involved is severely impaired.

As an example consider the situation that a `Button` in the graphical user interface (GUI) is supposed to cause two application domain objects – say a `FileFinder` and a `TextEditor` to interact with one another. It is not unlikely that a `MenuItem` in the GUI will have the same job. It would clearly be a poor idea to have the `Button` and the `MenuItem` objects refer explicitly to the `FileFinder` and the `TextEditor` objects. This kind of thing can be achieved by subclassing the classes `Button` and `MenuItem`, which would lead to a plethora of specialized `Button` and `MenuItem` classes. It is unlikely that these specialized GUI classes would be reusable in other contexts. Even worse would be a solution that has the `Filefinder` object know that it is finding a file to be used by a particular `TextEditor`, because a `FileFinder` is prima facie quite general purpose; it is willing to find files of all sorts for use by any object that needs a file as material to work on.

The general problem is that we have a group of objects (like our FileFinder) that need to send messages to another group of objects (like our TextEditor) but we want to decouple the first group from the second as much as possible. The solution using `courier` objects will typically look as follows. As the name implies a `courier` has the task of transporting a message of some kind (e. g. the path of a file the user has chosen) from an object in the first group that we call a `producer` (e. g. the `FileFinder`) to an object in the second group that we call a `consumer` (e. g. the `TextEditor`). The courier is not interested in the contents of the message and does not transform the message in any way. The `courier` "knows" its `producer` and its `consumer`, but the only thing it knows about it's `producer` is that it is of type `Producer` and thus has a method `produce`. An analogous statement holds for the `consumer`. The `producer` does not know or care about which `courier` will transport the message it produces, and it certainly does not know or care about the `consumer` that will eventually consume it's message. The `consumer` does not know or care about the `courier` from wich it receives a message and cares even less about the `producer` from which the message originally stems. Using this pattern it is possible to connect multiple `producers` to a single `consumer` and also multiple `consumers` to a single `producer`. The pattern is extremely flexible as we will see in further examples.

In addition to the three participants mentioned so far: `courier`, `producer`, `consumer` there is almost always a fourth object involved: the `invoker`. The `invoker` is typically a widget in the user interface. The `invoker` tells the `courier` when to start the ball rolling. A `courier` object has a method `execute`. The sequence of events is as follows:

1. The `invoker` (a `Button` or `MenuItem` object) calls the `execute` method of its `Courier`.

2. The `courier` calls the `produce` method of its `producer`.

3. The `courier` object calls the `consume` method of its `consumer` and hands it the message it was given; if the producer produced nothing the message will be `null`.

1.3 Applicability

Use the Courier pattern when

- many objects communicate with many other objects, whose identity they should not know.
- a behavior that's distributed among many classes should be customizable witout extensive subclassing.
- The additional indirection implied by couriers will not significantly impair performance.
- It is not imperative that the matching of message type and recipient type (Consumer in our terminology) must be statically checked at compile time. (In the section **Implementation** we will see how to achieve dynamic type checking.)

1.4 Structure

The static class strucure of the pattern **Courier** is shown as a UML diagram in Fig. 1.

1.5 Participants

- **Invoker** (Button, MenuItem)
  - Asks the Courier to start the communication process.
- **Courier**
  - Carries messages from a ConcreteProducer to a ConcreteConsumer.
- **Producer**
  - Provides an interface for creating messages to be consumed.
- **Consumer**
  - Provides an interface for consuming messages.
- **ConcreteProducer** (FileFinder)
  - Generates the messages the Courier is to transport.
- **ConcreteConsumer** (TextEditor)
  - Accepts the messages the Courier transports.
- **Message**
  - Can be of any type. This is what the Producer produces,the Consumer consumes and the Courier transports.
- **Client**
  - Creates a Producer P, a Consumer C and a Courier c connecting P to C. Will almost certainly also create an Invoker and tell it about c.

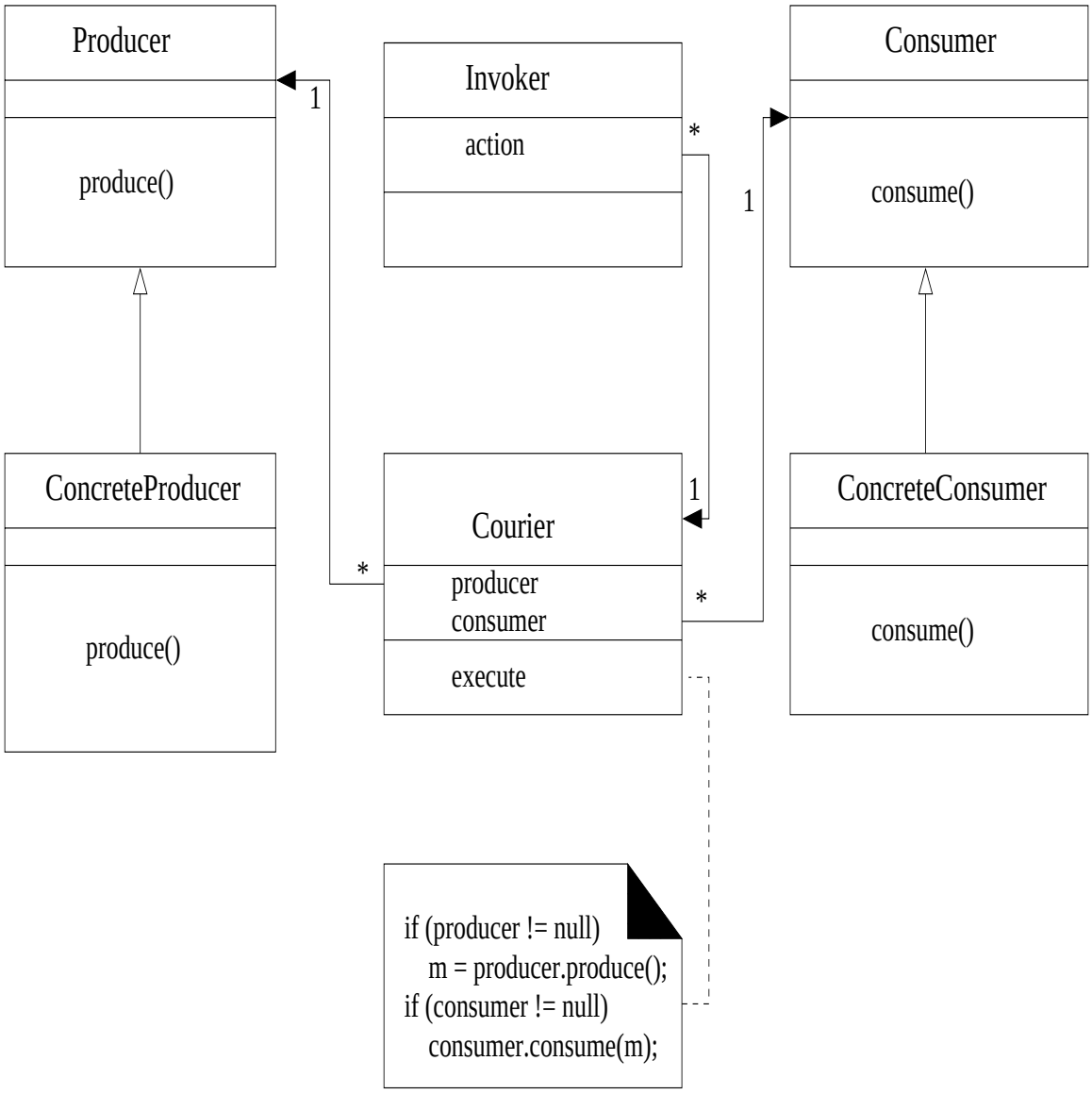

Figure 1: The static class structure

1.6 Collaborations

- Producers produce messages to be consumed by consumers they don't even know.
- Consumers consume messages from Producers they don't know.
- Couriers transport messages from Producers to Consumers without transforming the contents in any way.
- Invokers start the ball rolling by activating their couriers.

The typical communication among participants in the pattern **Courier** is shown as a UML sequence diagramm in Fig. 2.

1.7 Consequences

The Courier pattern has the following benefits:

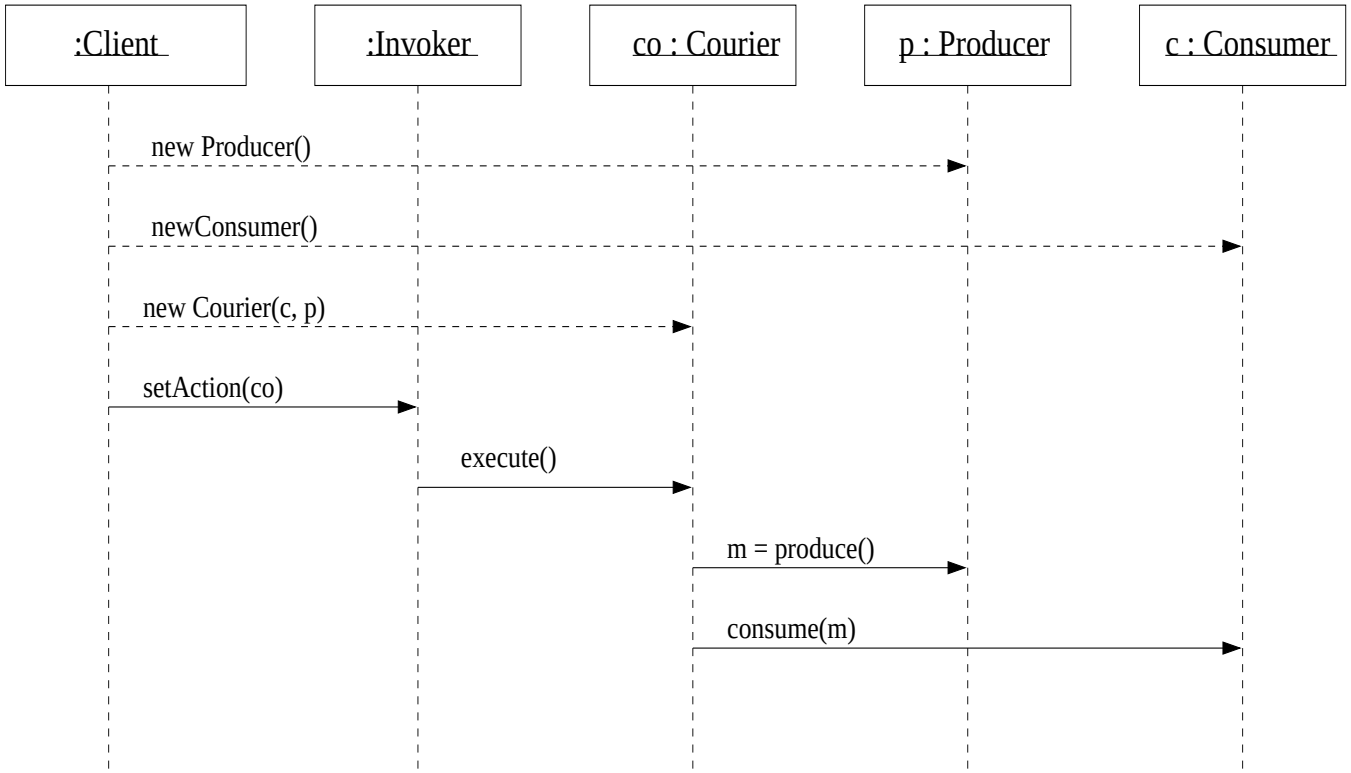

Figure 2: The communication between objects

1. It can help you achieve a maximal decoupling. With some care it should be possible to decouple the user interface from the application classes to such a degree that user interface and application can be developed completely independently – for example by two different development teams.

2. The pattern is easy to apply and shouldn't lead to complex monoliths of the sort the Mediator pattern tends to generate.

On the other hand the Courier pattern has the following drawbacks:

1. The high degree of indirection involved in the Courier pattern may lead to a certain degradation of efficiency. Having the Invoker call the `produce` method in the Producer directly and the `consume` method of the Consumer directly would undoubtedly improve efficiency but at the price of the disadvantages listed at the beginning of this section. On the other hand this interaction Invoker-Producer-Consumer is right at the user interface, where the decisive speed factor is the reaction time of the human user. There inefficient software seldom hurts.

2. In order to use the Courier pattern the application classes involved will have to subclass `Producer` or `Consumer`, which may seem a nuisance. In languages that support multiple inheritance this isn't too serious. In Java you can make `Producer` and `Consumer` into interfaces and let your application classes implement one or the other of these interfaces.

3. A courier can connect only two colleagues, whereas a mediator can connect arbitrarily many. See, however, the `Implementation` section.

1.8 Implementation

1. We can make the way a consumer reacts to a message variable by passing not only the message but also a message type to the method `consume`. The message type could take values defined as (static) constants in the class of the consumer. Thus a `TextEditor` could be induced to execute a `BlockCopy`, a `BlockCut` or a `BlockPaste` depending on the message type; in all three cases the actual message could be `null`, since it plays no role in this example. Here a `Producer` would also be superfluous. Three different `Couriers` could connect three different `Invokers` with one and the same `Consumer`; depending on which `MenuItem` the user chose the `TextEditor` would carry out one of its three operations. If we choose this path, then the constructor for the class `Courier` will accept an argument of type `int` that specifies the message type the courier will always pass to its `Consumer`. Used in this way a courier is just a variation on the **Command** pattern [Gamma+95].

2. In [Gamma+97] Gamma and Helm describe a Courier pattern that has great similarity to the one described here in many points. One significant difference, however, is that [Gamma+97] suggests providing a suitable message type for every consumer (the consumers are called "recipients" in that paper); this provides greater type safety since a consumer (recipient) is then guaranteed to understand any message it receives. [ICE94] achieved the same degree of (runtime) type safety by giving consumers a boolean function

   ```
   boolean validMessageType(Object m, int type);
   ```

   that a courier can call before passing its consumer a message. If the consumer rejects the message the courier should raise an uncaught exception: 'uncaught' because this kind of thing represents a logical error on the part of the programmer, who has obviously tried to pair up a producer and a consumer that don't fit together. An uncaught exception will cause the program to abort with a suitable error message during the testing phase. This is not as good as the static type checking provideed by pairing message types to consumer types but it serves the purpose.

3. It might be worth considering (if only for reasons of symmetry) letting the `produce` method accept an argument

   ```
   int messageType
   ```

   so that it could produce different messages for different couriers.

4. A courier as previously described can only initiate a single communication between colleagues `Producer` and `Consumer`. In many situations this will not be enough; The Mediator [Gamma+95] is capable of managing a much more complex set of actions. But in some cases this complex set of actions reduces to a sequence of simple communications `Producer` to `Consumer`.

   Example. Let us illustrate this situation by revisiting the example that [Gamma+95] uses for motivating the **Mediator** pattern. We have a dialog box with an entry field, a list box and several buttons that may be deactivated depending on the selection in the entry field. The sequence of simple communications involved here can be described as follows:

   (a) The list box as Producer produces its selection.

(b) The entry field as Consumer consumes the selection as the text that it displays.

(c) Several Couriers with the entry field as Producer activate or deactivate some buttons (their Consumers) depending on the selection. Here filter consumers (see 5. below) can be used to decide whether activation or deactivation is the right choice.

We can chain an arbitrary number of Couriers together by letting each Courier have a `successor`. The last step in the `execute` method of a Courier is to check whether its `successor` is non-null. If so, the `execute` method of the `successor` is called. (See the sample code below.)

5. In many applications it seems too restrictive to forbid the couriers to work in some appropriate fashion on the messages they are transporting. Shouldn't we allow them to transform their messages or at least check that the message fulfills some minimal requirements? But there are at least two arguments against giving in to this temptation:

   (a) The couriers are very easy to understand as long as they merely transport the message. It weould be a pity to sacrifice this clarity.

   (b) The manner in which a message should be 'handled' tends to be application-specific. Until now the couriers were completely application-neutral and they ought to stay that way!

   Examples.

   (a) In the first section we had the example of a `FileFinder` that produces a path for a file to be consumed by a `TextEditor`; shouldn't we at least check that the `TextEditor` will have read permission for the file before passing it the path? This looks like an ideal task for an "active courier". But careful! We are in danger of generating a whole slew of courier classes each of which handles its messages differently and mostly in an application-specific fashion. Let's stop and think again.

   (b) A courier connecting the entry field in a dialog box with a button that should be activated or deactivated depending on the text in the entry field might want to inspect the message (the text from the entry field) and set its message type accordingly (see 4. above).

   Let's just make the Consumer do any necessary filtering. But what about the *real* Consumer (the `TextEditor` in our example)? We can give each `Filter` a partner of type `Consumer` to which it passes its filtered message. Since `Filter` will subclass `Consumer`, we can stick together as many filters as we want to an arbitrarily long pipe in the well-known Unix fashion. We simply have to make the second `Filter` the partner of the first and so on.

   For example in addition to checking that the `TextEditor` will have read permission for the path we pass it we might also want to look inside the file to make sure it contains ASCII text. Then we would place an `AsciiChecker` between the `PermissionChecker` and the `TextEditor`.

6. The idea with the filters and particularly the PermissionChecker raises a problem that still needs a solution. What should a Filter do, if for some reason it can't filter its message properly?

In addition to the Filters we define a new subcategory of Consumers, the Error Handlers. Each Filter gets two partners of type `Consumer`: one of them is the next link in the filter chain, the other one is an error handler to be called upon if the filter process somehow goes wrong. The message passed to the error handler somehow describes what went wrong so that the user can be appropriately informed. An error handler might pop up a dialog box with a text supplied as message.

1.9 Sample Code

We will illustrate the use of the `Courier` pattern by giving a partial implementation for the communication between a user and a `TextEditor`. As programming language we will use Java, which is in the process of replacing C++ as lingua franca in the OO community. C++ programmers will have no difficulty in understanding the Java code.

As example we take the case of the `FileFinder` and the `TextEditor` as originally described. Let us suppose we already have classes `FileFinder` and `TextEditor` that we don't want to touch; we begin with some basic interfaces:

```
    public interface Producer
    {
        public Object produce();
    }
/************************************************/

    public interface Consumer
    {
        public void consume(Object message, int messageType);
    }
/************************************************/

    public interface Command
    {
        public void execute();
    }
```

Note for C++ programmers: Java interfaces are like C++ abstract classes that have no data members and in which all methods are pure virtual. Interfaces must be subclassed to provide an implementation.

And now our class `Courier`. As the reader will see, we have already incorporated the idea with message types as well as the possibility of chaining – both described in the previous section.

```
    public class Courier implements Command
    {
         private int      type;
         private Producer producer;
         private Consumer consumer;
         private Courier  successor;

         public Courier(Producer prod,Consumer cons,int type)
```

```
        // constructor
    {
        producer  = prod;
        consumer  = cons;
        successor = null;
        this.type = type;
    }

    public void setSuccessor(Courier aCourier)
    {
        successor = aCourier;
    }

    public void execute()
    {
        Object m = null;

        if (producer != null)
           m = producer.produce();

        if (consumer != null)
            consumer.consume(m, type);

        if (successor != null)
            successor.execute();
    }
}
```

We follow with a minimalistic class `Button` to show how an invoker might look.

```
// This class is supposed to illustrate a typical invoker.

public class Button
{
    protected String  label;
    protected Command action;

    public Button(String label) { this.label = label;}
    public void setAction (Command a) { action = a;}

    public void press()
    {
        changeToPressedAppearance();
    }

    public void release()
    {
        if (action != null)
            action.execute;
        restoreUnpressedAppearance();
```

```
        }
        // Undoubtedly dozens of other methods.
    }
```

The next few classes illustrate how we could use the pattern **Courier** to connect a FileFinder to a TextEditor.

```
    public class FileFinderAsProducer extends FileFinder
                                      implements Producer
    {
        public Object produce()
        {
            return theUsersChoice();
        }
    }
/*****************************************************/

    class UnrecognizedMessageException extends RuntimeException
    {}

    public class TextEditorAsConsumer extends TextEditor
                                      implements Consumer
    {
        public final static int BlockCopy  = 0;
        public final static int BlockCut   = 1;
        public final static int BlockPaste = 2;
        public final static int LoadText   = 3;

        public void consume(Object message,int messageType)
        {
            /* In the first three cases the message is */
            /* ignored; it's probably null anyway.     */

            switch(messageType)
            {
            case BlockCopy:
                copyBlock();
                break;
            case BlockCut:
                cutBlock();
                break;
            case BlockPaste:
                pasteBlock();
                break;
            case LoadText:
                String path = (String) message;
                loadText(path);
                break;
            default:
                throw new UnrecognizedMessageException();
```

```
        }
    }
}
```

Here we are obviously assuming that the class `FileFinder` has a method

```
String theUsersChoice()
```

Therefore the downcast used in `TextEditorAsConsumer` will be perfectly safe.

In just the same way we are assuming the class `TextEditor` already had methods

```
private void copyBlock()
private void cutBlock()
private void pasteBlock()
private void loadText(String path)
```

We finish our editor example with a rudimentary client to show how all these classes and interfaces could be used.

```
public class SimpleClient
{
    FileFinderAsProducer finder = new FileFinderAsProducer();
    TextEditorAsConsumer editor = new TextEditorAsConsumer();
    Courier              loader = new Courier(editor,
                                              finder,
                                              editor.LoadText
                                             );
    Courier              copier = new Courier(editor,
                                             null,
                                             editor.BlockCopy
                                            );
    Button               load_button = new Button("Load File");
    Button               copy_button = new Button("Copy Block");

    load_button.setAction(loader);
    copy_button.setAction(copier);

    // and so on ...
}
```

And now we come to the filter classes proposed in the section. **Implementation**. We give the general class Filter as well as a subclass PermissionChecker to illustrate the typical use of filters. This is a version that incorporates the idea with the error handlers.

```
public abstract class Filter implements Consumer
{
    private Consumer   partner;
    protected Consumer errorHandler;
    protected Object   filteredMessage;
    protected int      filteredType;

    public Filter(Consumer thePartner,
```

```
                    Consumer theErrorHandler)
        {
            partner      = thePartner;
            errorHandler = theErrorHandler;
        }

        public void setPartner(Consumer thePartner)
        {
            partner = thePartner;
        }

        public void setErrorHandler(Consumer theErrorHandler)
        {
            errorHandler = theErrorHandler;
        }

        public void consume(Object message,int messageType)
        {
            if (filter(message, messageType))
            {
                if (partner != null)
                    partner.consume(filteredMessage,
                                    filteredType);
            }
            else if (errorHandler != null)
                    errorHandler.consume(filteredMessage,
                                         filteredType);

        }

        protected abstract boolean filter(Object message,
                                          int messageType);
        // This method must be suitably redefined in each subclass.
        // It should not forget to set filteredType as needed.
        // It should return true if the filtering was successful.
    }
/*************************************************************/

    //File : PermissionChecker.java

    public class PermissionChecker extends Filter
    {
        private String permissionString;
        /*
         * This string may contain any of the characters
         * e, f, r or w in any order. Their meaning is
         * e : Does the file described by the path exist?
         * f : Does the path designate an ordinary file
         *     as opposed to a subdirectory?
```

```
 * r : Does this process have read permission
 *     for this file?
 * w : Does this process have write permission
 *     for this file?
 */

public PermissionChecker(Consumer thePartner,
                         Consumer theErrorHandler)
    { super(thePartner, theErrorHandler); }

public void setPermissionString(String perms)
{
    permissionString = perms;
}

protected boolean filter(Object message,int messageType)
{
    String path = (String) message;
// Check the permissions described by permissionString
// for path.

    if (path fails the existence test)
    {
         filteredType    = ...;
         filteredMessage = "The file " + path +
                           " does not exist.";
         return false;
    }
    else if (path fails the ordinary file test)
    {
        filteredType    = ...;
        filteredMessage = path +
                          " does not specify a file.";
        return false;
    }
    else if (path fails the read test)
    {
        filteredType    = ...;
        filteredMessage =
            "You do not have read permission for " + path;
        return false;
    }
    else if (path fails the write test)
    {
        filteredType    = ...;
        filteredMessage =
             "You do not have write permission for " + path;
        return false;
    }
```

```
            else
            {
                filteredType    = messageType;
                filteredMessage = path;
                return true;
            }
        }
    }
/*************************************************************/

    public class ErrorHandler implements Consumer
    {
        public void consume(Object message,int messageType)
        {
            String text = (String) message;

            // Display a dialog box with text as content.
        }
    }
```

The method `consume` in our class `Filter` is an example of the **Template Method** pattern [Gamma+95]. Subclasses need only redefine the method `filter`.

1.10 Related Patterns

The **Courier** pattern is closely related to the **Mediator** pattern [Gamma+95]. The Mediator can mediate communication between arbutrarily many colleagues, whereas the Courier can prima facie only mediate between two colleagues (but see the section **Implementation** for more on this). It is also like the **Observer** pattern [Gamma+95] in that it permits an object, typically the consumer, to react to a change in a second object, which in this case would be simultaneously invoker and producer. If the courier is used in this way we have the same problem as with the observer: unless the producer knows explicitly what type of consumer is observing it, it can't know what information to put into the message. This problem could be solved by having the producer put a reference `this` to itself into the message and letting the consumer call appropriate methods of the producer to get the information it needs (i. e. 'pulling' instead of 'pushing'). But then again this solution would increase the coupling between producer and consumer.

It would be a mistake to overlook the similarities (at least in the objectives) between the **Courier** pattern and the design patterns built into the event modell for Java in the JDK 1.1. In the JDK 1.1 there are "event sources" that produce or "fire" events and there are "event listeners" that report their interest in being informed about certain types of events. The classes that represent potential event sources (typically AWT components) have methods with names like `addXXXListener` and `removeXXXListener` where `XXX` represents the kind of event a listener might be interested in: thus there are methods

```
    void addActionListener(ActionListener l);
    void addMouseListener(MouseListener l);
    void addWindowListener(WindowListener l);
    void removeActionListener(ActionListener l);
    void removeMouseListener(MouseListener l);
```

```
void removeWindowListener(WindowListener l);
```

A "broadcast source" is willing to register arbitrarily many listeners. When such a source needs to fire an event of a given type it informs all listeners that have registered an interest in this type of event. For this purpose there are interfaces

```
ActionListener
MouseListener
WindowListener
```

and so forth that potential listeners can implement. The interface `ActionListener` declares a method

```
void actionPerformed(ActionEvent e);
```

that is called by a source that fires an `ActionEvent` In the same vein the interface `MouseListener` declares methods

```
void mousePressed(MouseEvent e);
void mouseReleased(MouseEvent e);
void mouseClicked(MouseEvent e);
void mouseEntered(MouseEvent e);
void mouseExited(MouseEvent e);
```

Clearly this design pattern achieves some of the goals we wanted to reach with our **Courier** pattern. The event source as producer of events "knows" who its consumers (listeners) are in the sense that it keeps a list of references to them; but it does not even know what their types are except that the types must implement the appropriate interfaces. On the other hand the consumer (listener) must explicitly know the source of the events it wants to listen for, in order to be able to call the `addXXXListener` method for that source.

1.11 Known Uses

ICE (Interface Classes for Eiffel) [ICE94] a port (or migration?) of InterViews [LCITV92] from C++ to Eiffel introduced the class `COURIER` and used it extensively to avoid the C++-typical callbacks.

The Unidraw graphical editing framework [VL90] uses this technique to dispatch commands through its user interface.

References

[Gamma+95] E. Gamma,R. Helm,R. Johnson, and J. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addisdon-Wesley, 1995.

[Gamma+97] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, The Courier Pattern, *Dr. Dobb's Journal*, 1997.

[LCITV92] M. Linton, P. Calder, J. Interrante, S. Tang, and J. Vlissides *InterViews Reference Manual*(3.1 ed.) Stanford CA: CSL, Stanford University, 1992.

[ICE94] The Programmer's Manual *The ICE Library*, SwisSoft, 1994.

[VL90] John M. Vlissides and Mark A. Linton, Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237-268, July 1990.