

# Lecture 15: Design Patterns for Events and Wrappers

October 9, 2002

In this, the final lecture on design patterns, we consider two other groups of design patterns. One group supports *multiway communication*. One client calling one receiver is a straightforward problem that we already know how to solve using method calls. What happens when a client needs to communicate with multiple receivers, which may not even be known until runtime? What about communications involving multiple clients and multiple receivers? These problems are addressed by the observer, blackboard, and mediator patterns.

The other group of patterns considered here are *wrapper* patterns. A wrapper intercedes between a client and a service, in order to adapt incompatible interfaces, provide extra functionality, or control access. The wrapper patterns are adapter, decorator, and proxy.

## 15.1 Observer

Suppose you are developing a personal finance application that allows a user to track the value of a stock portfolio. The application has a class `StockTicker` that uses the Web to obtain the latest quotes for the stocks in the user's portfolio. At some point in time, `StockTicker` might have these prices recorded:

Ticker Symbol	Price
DELL	25
IBM	57
YHOO	9

One way to use `StockTicker` would display the prices of the stocks as a line graph, scrolling across the corner of the user's screen. Another class, call it `Graph`, is responsible for displaying this graph. It provides a method for `StockTicker` to call to send it updates:

```
void update (Symbol sym, double price) {  
    ... // add a point to the line graph for sym  
}
```

In order to keep the graph up to date, `StockTicker` must call this method of `Graph` whenever the price of a stock changes:

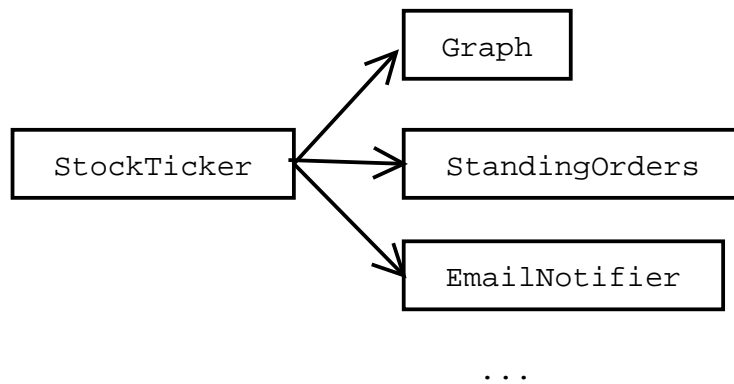
```
graph.update (DELL, 26);
```

Now we decide to add a new feature that depends on stock price data: `StandingOrder`, which represents an order to buy or sell a stock whenever its price reaches a certain level. Now we must change `StockTicker` so that it sends updates to the standing orders as well:

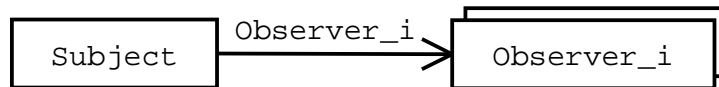
```
graph.update (DELL, 26);
standingOrders.update (DELL, 26);
```

With this approach, every new class that needs to receive stock price information will require modifying `StockTicker` and recompiling it. Good programming practice is supposed to relieve us from such hard-coded modifications. `StockTicker` should be reusable without recompiling it for every new purpose.

The problem can be easily visualized with a module dependence diagram, which shows that `StockTicker` has a direct dependence on each of its consumers:



In more abstract terms, we have a set of modules or objects that are interested in being notified about changes in another object. The interested modules are called *observers*, and the module they're observing, the *subject*:



The observer pattern solves this problem. Rather than hard-coding dependences on each observer `Observer_i`, the subject interacts with its observers through an `Observer` interface that each observer must implement. The resulting MDD looks like this:

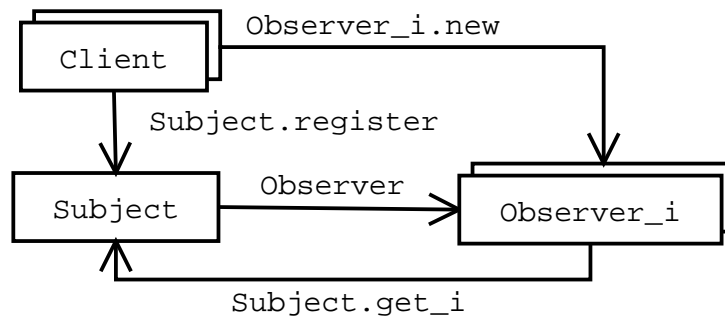


The `Observer` interface has one or more methods that signify changes in the subject. Here's a generic observer interface:

```
interface Observer {
    void update (Event event);
}
```

In practice, the method often has a more specific name describing the kind of change that occurred, such as `stockPriceChanged`. The arguments passed to the update method are usually encapsulated as a single object, an *event* describing the change. The event may include a lot of data about the change (a *push* structure), or it may have very little, leaving it up to an observer to query the subject and obtain the changes (a *pull* design). There is a tradeoff between how much data you push and how much you leave the observers to pull. If you push nothing, you leave the observers with the problem of determining what actually changed on the object, which may be costly and redundant between observers. If you push too much, however, you may be sending more data than your observers actually care about. Pushing too much can also cause subtle consistency problems, which are described in more detail below.

The MDD above has left out one important piece of the puzzle: how the subject finds out which observers it should send its updates to. We assume there is some *client* that takes care of creating (or otherwise obtaining a reference to) each observer, and passing them to the subject through a registration interface. We'll also add an edge that indicates that observers may need to pull data from the subject:



The `register` method adds an observer to the subject's list of active observers. A corresponding `unregister` method removes an observer from the list.

Here's what a typical subject's code might look like:

```

class Subject {
    private List observers = new LinkedList ();
    ...

    // effects: adds o to observers of this object
    public void register (Observer o) {
        observers.add (o);
    }

    // effects: removes o from observers of this object
    public void unregister (Observer o) {
        observers.remove (o);
    }
    ...
}
  
```

It's helpful to encapsulate the event broadcast code in a private method. Whenever an interesting change occurs in the subject (such as an updated stock price), the subject calls this method to broadcast the change to all its observers:

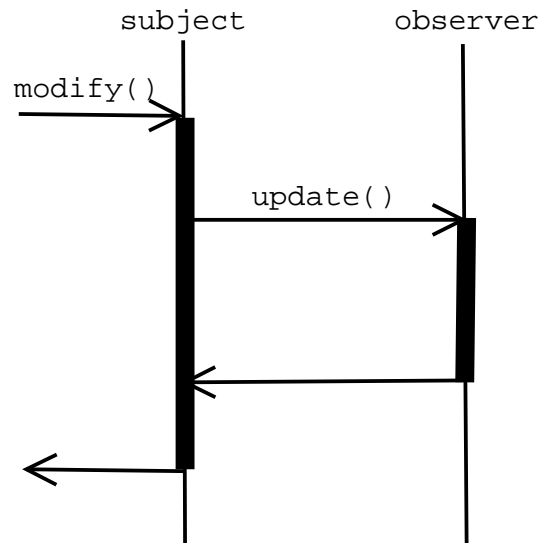
```

private void fireUpdates (Event e) {
    Iterator g = observers.iterator ();
    while (g.hasNext ()) {
        Observer o = (Observer) g.next ();
        o.update (e);
    }
}

```

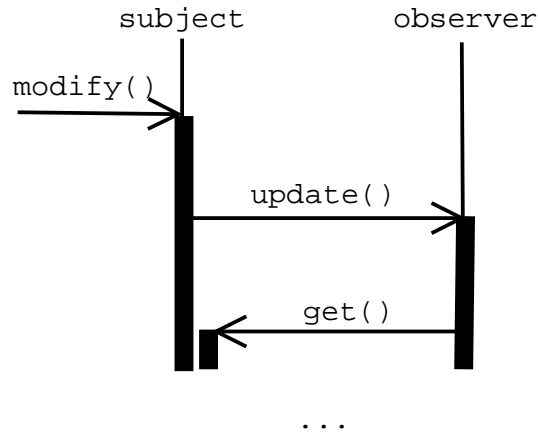
Here's where some of the subtleties of the observer pattern start to appear. When a subject calls `update` on one of its observers, the subject is making a *callback*, giving up control to arbitrary code in another module while the subject is still running. The observer is free to call other methods on the subject, and frequently does. As a result, the subject must be designed to be *reentrant*, meaning one method can be called while another method is running somewhere higher on the stack.

Let's look at some of the problems that can crop up when a subject isn't reentrant. It's helpful to visualize these problems using an *interaction diagram*, which depicts several interacting objects and the method calls they make on each other over time. In an interaction diagram, time flows downward. Each vertical line represents an object's *lifeline*. Method calls are represented by an arrow from the calling object to the receiver, labelled with the method name. When a method returns, an arrow is drawn back. Whenever an object is active (i.e., when one of its method calls is executing), its lifeline is drawn thickly. Here is a simple interaction between one subject and one observer:



The subject is originally entered by some method, here generically referred to as `modify`, that triggers a change. The subject then calls `update` on its observer. Multiple observers would be depicted as additional vertical lines in the diagram, with `update` calls to each in succession.

The first problem occurs when an observer pulls some data from the subject, using an observer method (in the creator/mutator/observer sense) on the subject to find out about its changed state. We'll generically refer to this method as `get`:

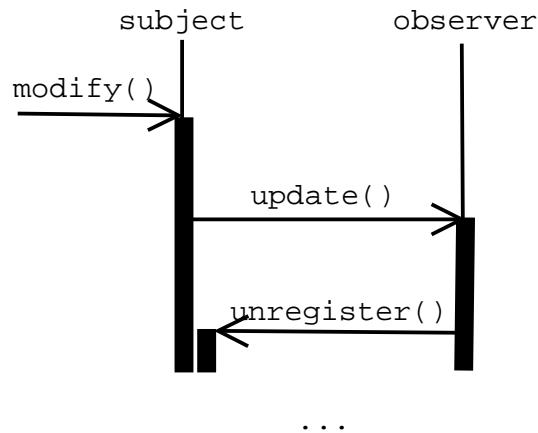


The pitfall is this: the subject is in the midst of changing (because of the `modify` method call). What if it didn't finish updating its rep before it notified the observer about the change? Then its rep may be inconsistent, and `get` may return the wrong results. This is a particularly sticky problem if `modify` makes several changes that need to trigger notifications to different observers. It can be tempting to call `fireUpdates` as you go along, instead of waiting until the rep is consistent, but this is the wrong thing to do. The subject must treat a callback to an observer in the same way as it treats returning to a caller. Just as you make sure to establish the rep invariant before returning from a method, you need to establish your rep invariant before making a callback. Often the best way to do this is to postpone calling `fireUpdates` until the end of the method that triggered the change.

To summarize:

*Establish the rep invariant before calling observers.*

Another treacherous call an observer might make on its subject is registration or unregistration. Unregistration is more common, because an observer may decide that it no longer has any interest in the subject. For example, a standing order to sell a stock is only interested in updates to the stock price until the price reaches the threshold and the order executes. After that, the standing order should simply unregister itself and disappear. Here's what this looks like in an interaction diagram:



The problem here is that `fireUpdates` has an active iterator over the list of observers, so `unregister` shouldn't be allowed to change the list out from under it. The solution to this problem is defensive copying. One way to do it is to have `fireUpdates` copy the list each time, and always iterate over a copy of the list:

```
Iterator g = new LinkedList (observers).iterator ();
```

This approach is probably overkill, since in most applications, unregistering during an event is rare relative to the total number of events. A better approach is copy-on-demand. Use a boolean variable to indicate when an iterator is active, and have `register` and `unregister` check this variable to determine if they should copy the observer list before modifying it:

```
class Subject {
    private List observers = new LinkedList ();
    private boolean iteratingOverObservers = false;
    ...

    // effects: removes o from observers of this object
    public void unregister (Observer o) {
        if (iteratingOverObservers) {
            observers = new LinkedList (observers);
            iteratingOverObservers = false;
        }
        observers.remove (o);
    }

    private void fireUpdates (Event e) {
        Iterator g = observers.iterator ();
        iteratingOverObservers = true;
        try {
            while (g.hasNext ()) {
                Observer o = (Observer) g.next ();
                o.update (e);
            }
        } finally { iteratingOverObservers = false; }
    }
}
```

Note that a `try-finally` construct is used here to reset the flag even if a broken observer throws an exception. Unexpected exceptions are another reason to establish your rep invariant before making callbacks!

In summary:

*Make sure registration and unregistration are reentrant.*

Things get even trickier when the observer calls a mutator method on the subject. What if the mutator triggers another round of updates:



This ensures that all observers see the events in the same order, but it doesn't solve the inconsistency problem: when o2 sees event A, the subject's state has already been changed (by `subject.modify`) to reflect the queued event B.

Another solution is to skip sending event A once it's out of date:

```
o1.update (A)
  subject.modify ()
    o1.update (B)
    o2.update (B)
now A is out of date, so don't send it to o2
```

The drawback here is that o2 is now missing some events that o1 is seeing, which may not work for some applications.

Finally, you might decide not to push any data at all, so that every event looks like every other:

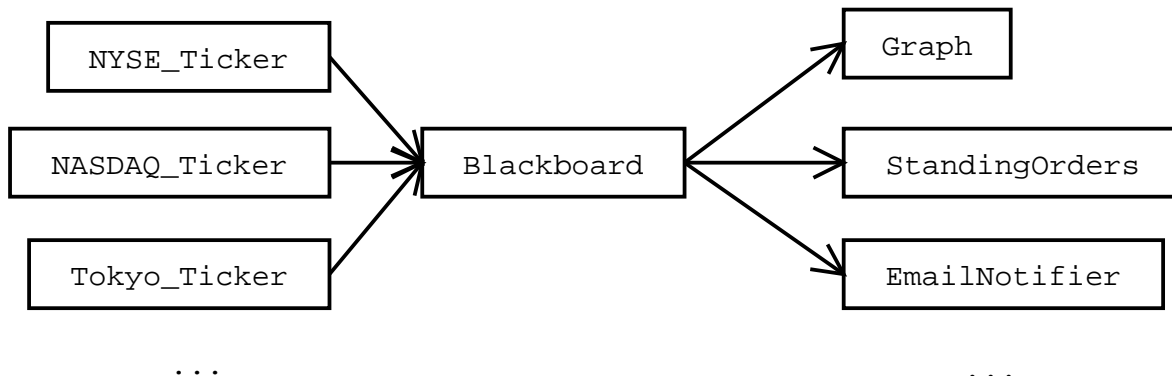
```
o1.update ()
  subject.modify ()
    o1.update ()
    o2.update ()
o2.update ()
```

But now the observers must discover for themselves how the subject changed.

There is no perfect solution to this problem. Different solutions are appropriate for different applications. Sometimes the problem can be specified away, too: the subject can forbid its observers to invoke mutators that would trigger recursive updates.

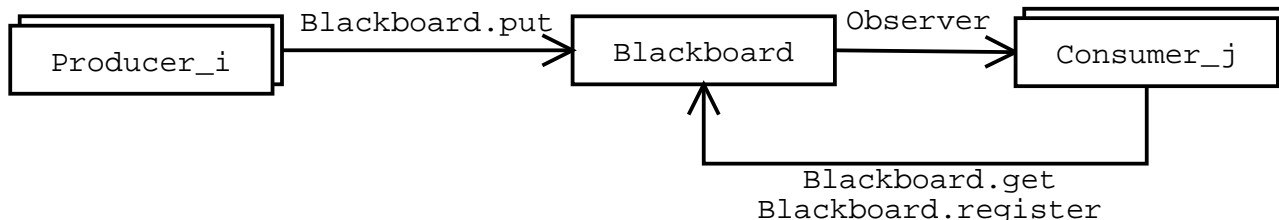
## 15.2 Blackboard

Suppose now there are several subjects that we want to watch. For example, we might have a different `Stock-Ticker` for each stock market: `NYSE_Ticker`, `NASDAQ_Ticker`, `Tokyo_Ticker`... Instead of making  $O(mn)$  connections between  $m$  data sources and  $n$  observers, we make only  $O(m + n)$  connections through an intervening mediating called a *blackboard*:



A blackboard is a repository of messages that is readable and writable by many modules. Whenever an event occurs that might be of interest to another party, the module responsible for or aware of the event adds to the blackboard an announcement of the event. Other modules can read the blackboard. In the typical case, they will ignore most of its contents, which do not concern them, but they may take action on other events. A module that posts an announcement to the blackboard has no idea whether zero, one or many other modules are paying attention to its announcements. The blackboard thus permits *implicit invocation*, where a module calls other modules without knowing who they are.

The abstract MDD for the blackboard pattern looks like this:



The blackboard pattern generalizes the observer pattern to permit multiple data sources, here called *producers*, as well as multiple observers, called *consumers*. It completely decouples producers from consumers, so neither has to know about the others. Some modules may be both producers and consumers, posting some events to the blackboard and listening for others.

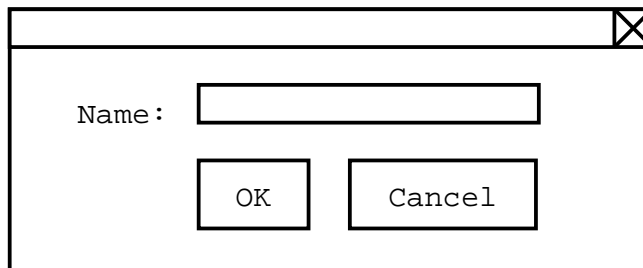
Blackboards generally do not enforce much structure on their announcements, but a well-understood message format is required so that processes can interoperate. Some blackboards provide filtering services so that clients do not see all announcements, just those of a particular type. Other blackboards automatically send announcements to clients which have registered interest.

One example of a blackboard system in common use at MIT is Zephyr. The Zephyr server acts as the blackboard. Messages posted to a Zephyr instance are stored on the server, and Zephyr clients that have registered an interest in the Zephyr instance are notified in turn.

Blackboards can also be used to organize large-scale distributed processing. The blackboard stores chunks of work — e.g., recordings of radio signals to be scanned for signs of extraterrestrial life. Consumers pick up work from the blackboard, do the processing, and leave the answer back on the blackboard.

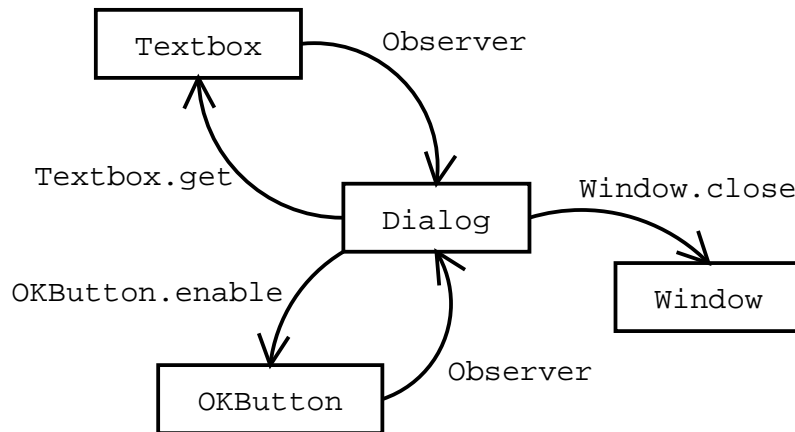
### 15.3 Mediator

What if we want tighter coordination between multiple subjects and multiple observers than the blackboard can conveniently or efficiently provide? Many examples can be found in user interface programming. Consider a simple dialog box with a text entry field and a couple of buttons:

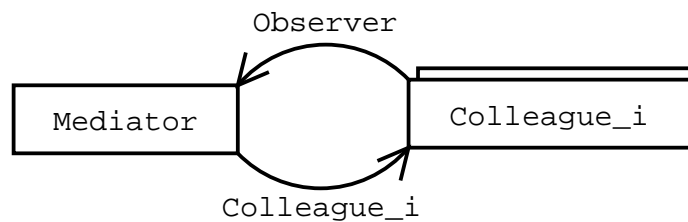


In order to work well, the widgets in the dialog box need to cooperate closely. We want the OK button to be disabled (grayed out) until the user types something in the text field; in other words, an event on the text field (the user's typing) should trigger an action on the OK button (enabling or disabling). Similarly, clicking the OK button should cause the dialog window to close. As a result, each widget has dependencies on the others. The OK button needs to observe events from the text field, and the window needs events from the OK button. But these widgets are off-the-shelf parts. We don't want to make a new kind of OK button for every dialog box in an application.

The solution is another object, a *mediator* that sits in the middle, listening to events from each widget and triggering the appropriate actions on other widgets. The MDD might look like this:



In abstract terms, the mediator pattern consists of a set of *colleagues* (here, the widgets) and a mediator that decouples the colleagues from each other:



The mediator is tightly coupled to the colleagues, so changing, say, the text field into a drop-down list box would require changing the mediator. But the colleagues have *no* dependences on each other, so this change won't affect any of the other widgets. The colleagues have a weak dependence on the mediator, using an observer pattern or something similar, so colleagues can be written without prior knowledge of the mediator and assembled into a library.

On the down side, the mediator itself may become large, complex, and monolithic, particularly when there are many colleagues. When this happens, it is often a good idea to try to subdivide the colleagues further and introduce new mediators. User interfaces often have a natural subdivision based on the layout of the interface. A window with several panes might have a different mediator for each pane.

## 15.4 Wrappers

Our final set of patterns, wrappers, all share the same form. Wrappers modify the behavior of another class. They are usually a thin veneer over the encapsulated class, here called the *service*, which does the real work. The wrapper may modify the interface of the service, extend its behavior, or restrict access to it.

Three varieties of wrappers are adapters, decorators, and proxies. All three have the same abstract MDD:



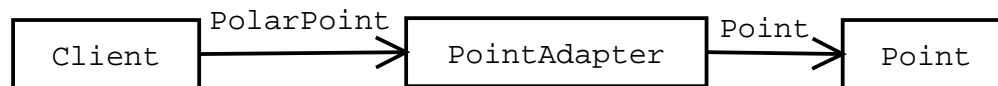
The varieties differ on whether the wrapper's interface ( $I'$ ) is the same as the service's ( $I$ ), and whether the wrapper adds functionality or not:

Pattern	Functionality	Interface
Adapter	same	different ( $I \neq I'$ )
Decorator	different	same ( $I = I'$ )
Proxy	same	same ( $I = I'$ )

The remainder of this section discusses the three varieties of wrapper.

### 15.4.1 Adapter

Adapters change the interface of a class without changing its basic functionality. For instance, they might permit points represented in Cartesian coordinates to be used with an algorithm that expects polar coordinates:



The code for the wrapped class, `Point`, looks like this:

```
class Point {
    int x; int y;
    Point (int x, int y) {
        this.x = x; this.y = y;
    }
    int getX () { return x; }
    int getY () { return y; }
}
```

The interface desired by the client is `PolarPoint`:

```
interface PolarPoint {
    double getRadius ();
    double getTheta ();
}
```

The client can't use a `Point` directly because of the incompatible interface. However, you can write an adapter which permits its use. There are two ways to do this: subclassing and delegation. Here is the subclassing solution:

```
// adapter implemented by subclassing
class PointAdapter extends Point implements PolarPoint {
    PointAdapter (int x, int y) {
        super (x, y);
    }
    double getRadius () {
        int x = getX (); int y = getY ();
        return Math.sqrt (x*x + y*y);
    }
    double getTheta () {
        return Math.atan2 (getY(), getX());
    }
}
```

With subclassing, you get a single object that is simultaneously a `Point` and a `PolarPoint`, with all the methods of both.

Delegation is a technique for “passing the buck,” forwarding a request so that a different object does the requested work:

```
// adapter implemented by delegation
class PointAdapter implements PolarPoint {
    Point pt;
    PointAdapter (Point pt) {
        this.pt = pt;
    }
    double getRadius () {
        int x = pt.getX (); int y = pt.getY ();
        return Math.sqrt (x*x + y*y);
    }
    double getTheta () {
        return Math.atan2 (pt.getY(), pt.getX());
    }
}
```

Delegation is a more common way to write wrappers, because it allows you to defer the choice of the wrapped class until runtime. For instance, if you had a `ColorPoint` object (an instance of a subclass of `Point`), you would be able to wrap it with the delegating adapter, but not with the adapter that subclasses `Point`.

## 15.4.2 Decorator

Whereas an adapter changes interface without adding new functionality, a decorator extends functionality while maintaining the same interface.

Examples of decoration can be found in the Java `OutputStream` classes. An `OutputStream` accepts a stream of bytes passed to it through calls to its `write` method. An output stream might write its byte stream to a file or ship it across a network connection. But some `OutputStream` classes are just decorators, accepting a byte stream and processing it in some way before passing it on to another `OutputStream`. The Java library includes a number of these decorators:

- `BufferedOutputStream` uses a buffer to accumulate a sequence of small writes into a larger write, making disk or network transfer more efficient.
- `GZIPOutputStream` compresses the byte stream into a shorter stream using the `gzip` compression algorithm.
- `CipherOutputStream` encrypts the byte stream before sending it on.

Decorators generally must be implemented with delegation, because a decorator doesn't know until runtime what class it might be wrapped around.

### 15.4.3 Proxy

A proxy is a wrapper that has the same interface and functionality as the class it wraps. This does not sound very useful on the face of it. However, proxies serve an important purpose in controlling access to other objects. This is particularly valuable if the wrapped object must be accessed in a stylized or complicated way.

One variety of proxy is a security proxy. It might operate correctly if the caller has the correct credentials (such as valid Kerberos tickets), but throw an error if an unauthorized user attempts to perform operations.

Another kind of proxy is used to access a service on a remote machine. Instead of requiring the client to perform all the necessary network communications, it is easier to create a local proxy that has the same interface as the desired service, but which packages up the method arguments, ships them over the network, then waits for and returns the result. This simplifies the client by localizing network-specific code in the proxy.

A final example is a proxy for an object that may not yet exist. If creating an object is expensive, then it can be represented by a proxy. The first time a method is invoked on the proxy, the proxy creates the wrapped object, and then delegates the method call and all future method calls to it. If the proxy is never called, the work of creating the object need never be performed.