CHAPTER 5

Keeping Up with the Times with the Observer

One of the knottiest design challenges is the problem of building a system that is highly integrated—that is, a system where every part is aware of the state of the whole. Think about a spreadsheet, where editing the contents of one cell not only changes the number in the grid but also changes the column totals, alters the height of one of the bars in a bar chart, and enables the Save button. Even more simply, how about a personnel system that needs to let the payroll department know when someone's salary changes?

Building this kind of system is hard enough, but throw in the requirement that the system be maintainable and now you are talking truly difficult. How can you tie the disparate parts of a large software system together without increasing the coupling between classes to the point where the whole thing becomes a tangled mess? How do you construct a spreadsheet that properly displays changes in the data without hard-coding a link between the spreadsheet editing code and the bar chart renderer? How can you make the `Employee` object spread the news about salary changes without tangling it up with the payroll system?

Staying Informed

One way to solve this problem is to focus on the fact that the spreadsheet cell and the `Employee` object are both acting as a source of news. Fred gets a raise and his `Employee` record shouts out to the world (or at least to anyone who seems interested), "Hello! I've got something going on here!" Any object that is interested in the state of

Fred's finances need simply register with his `Employee` object ahead of time. Once registered, that object would receive timely updates about the ups and downs of Fred's paycheck.

How would all of this work in code? Here is a basic version of an `Employee` object with no code to tell anyone anything—it just goes about its business of keeping track of an employee:

```
class Employee
 attr_reader :name
 attr_accessor  :title, :salary

 def initialize( name, title, salary )
   @name = name
   @title = title
   @salary = salary
 end
end
```

Because we have made the salary field accessible with `attr_accessor`, our employees can get raises:[1]

```
fred = Employee.new("Fred Flintstone", "Crane Operator", 30000.0)

# Give Fred a raise

fred.salary=35000.0
```

Let's now add some fairly naive code to keep the payroll department informed of pay changes:

```
class Payroll
  def update( changed_employee )
    puts("Cut a new check for #{changed_employee.name}!")
    puts("His salary is now #{changed_employee.salary}!")
  end
end
```

1. Of course, the employees can also suffer pay cuts, but we will silently pass over such unpleasant matters.

```
class Employee
  attr_reader :name, :title
  attr_reader :salary

  def initialize( name, title, salary, payroll)
   @name = name
   @title = title
   @salary = salary
   @payroll = payroll
  end

  def salary=(new_salary)
    @salary = new_salary
    @payroll.update(self)
  end
end
```

We can now change Fred's wages:

```
payroll = Payroll.new
fred = Employee.new('Fred', 'Crane Operator', 30000, payroll)
fred.salary = 35000
```

And the payroll department will know all about it:

```
Cut a new check for Fred!
His salary is now 35000!
```

Note that since we need to inform the payroll department about changes in salary, we cannot use `attr_accessor` for the salary field. Instead, we need to write the `salary=` method out by hand.

A Better Way to Stay Informed

The trouble with this code is that it is hard-wired to inform the payroll department about salary changes. What do we do if we need to keep other objects—perhaps some accounting-related classes—informed about Fred's financial state? As the code stands right now, we must go back in and modify the `Employee` class. Needing to change the `Employee` class in this situation is very unfortunate because nothing in the `Employee`

class is really changing. It is the other classes—the payroll and accounting classes—that are actually driving the changes to `Employee`. Our `Employee` class seems to be showing very little change resistance here.

Perhaps we should step back and try to solve this notification problem in a more general way. How can we separate out the thing that is changing—who gets the news about salary changes—from the real guts of the `Employee` object? What we seem to need is a list of objects that are interested in hearing about the latest news from the `Employee` object. We can set up an array for just that purpose in the `initialize` method:

```
def initialize( name, title, salary )
  @name = name
  @title = title
  @salary = salary
  @observers = []
end
```

We also need some code to inform all of the observers that something has changed:

```
def salary=(new_salary)
  @salary = new_salary
  notify_observers
end

def notify_observers
  @observers.each do |observer|
    observer.update(self)
  end
end
```

The key moving part of `notify_observers` is `observer.update(self)`. This bit of code calls the `update` method on each observer,[2] telling it that something—in this case, the salary—has changed on the `Employee` object.

2. Recall that `array.each` is Ruby terminology for a loop that runs through all of the elements in an array. We will have more to say about this `array.each` in Chapter 7.

The only job left is to write the methods that add and delete observers from the `Employee` object:

```
def add_observer(observer)
  @observers << observer
end

def delete_observer(observer)
  @observers.delete(observer)
end
```

Now any object that is interested in hearing about changes in Fred's salary can simply register as an observer on Fred's `Employee` object:

```
fred = Employee.new('Fred', 'Crane Operator', 30000.0)

payroll = Payroll.new
fred.add_observer( payroll )

fred.salary=35000.0
```

By building this general mechanism, we have removed the implicit coupling between the `Employee` class and the `Payroll` object. `Employee` no longer cares which or how many other objects are interested in knowing about salary changes; it just forwards the news to any object that said that it was interested. In addition, instances of the `Employee` class will be happy with no observers, one, or several:

```
class TaxMan
  def update( changed_employee )
    puts("Send #{changed_employee.name} a new tax bill!")
  end
end

tax_man = TaxMan.new
fred.add_observer(tax_man)
```

Suppose we change Fred's salary again:

```
fred.salary=90000.0
```

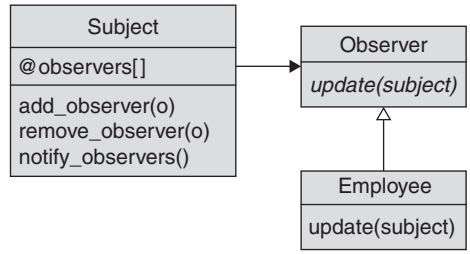

Figure 5-1 The Observer pattern

Now both the payroll department and the tax man will hear about it:

```
Cut a new check for Fred!
His salary is now 80000.0!
Send Fred a new tax bill!
```

The GoF called this idea of building a clean interface between the source of the news that some object has changed and the consumers of that news the Observer pattern (Figure 5-1). The GoF called the class with the news the **subject** class. In our example, the subject is the `Employee` class. The **observers** are the objects that are interested in getting the news. In our employee example, we have two observers: `Payroll` and `TaxMan`. When an object is interested in being informed of the state of the subject, it registers as an observer on that subject.

It has always seemed to me that the Observer pattern is somewhat misnamed. While the observer object gets top billing—in fact, the only billing—it is actually the subject that does most of the work. It is the subject that is responsible for keeping track of the observers. It is also the subject that needs to inform the observers that a change has come down the pike. Put another way, it is much harder to publish and distribute a newspaper than to read one.

Factoring Out the Observable Support

Implementing the Observer pattern in Ruby is usually no more complex than our `Employee` example suggested: just an array to hold the observers and a couple of methods to manage the array, plus a method to notify everyone when something changes. But surely we can do better than to repeat this code every time we need to make an object observable. We could use inheritance. By factoring out the code that manages the observers, we end up with a functional little base class:

```
class Subject
  def initialize
    @observers=[]
  end

  def add_observer(observer)
    @observers << observer
  end

  def delete_observer(observer)
    @observers.delete(observer)
  end

  def notify_observers
    @observers.each do |observer|
      observer.update(self)
    end
  end
end
```

Now we can make `Employee` a subclass of `Subject`:

```
class Employee < Subject
  attr_reader :name, :address
  attr_reader :salary

  def initialize( name, title, salary)
   super()
   @name = name
   @title = title
   @salary = salary
  end

  def salary=(new_salary)
    @salary = new_salary
    notify_observers
  end
end
```

This is not a completely unreasonable solution; in fact, Java has gone exactly this route with its `java.util.Observable` class. But, as we saw in Chapter 1, using inheritance can cause grief. The problem with using `Subject` as a base class is that it

shuts out the possibility of having anything else as a base class. Ruby allows each class to have exactly one superclass: Use up your single superclass of `Employee` on `Subject` and you are done. If our domain model demands that we make `Employee` a subclass of `OrganizationalUnit` or `DatabaseObject`, we are out of luck; we cannot also make it a subclass of `Subject`.

The problem is that sometimes we want to share code between otherwise unrelated classes. Our `Employee` class wants to be a `Subject`, but perhaps so does that spreadsheet cell. So how can we share the `Subject` implementation without using up our one allotted superclass?

The solution to this dilemma is to use a module. Recall that a module is a package of methods and constants that we can share among classes, but that does not soak up the single allotted superclass. If we recast our `Subject` class as a module, it does not really look all that different:

```
module Subject
  def initialize
    @observers=[]
  end

  def add_observer(observer)
    @observers << observer
  end

  def delete_observer(observer)
    @observers.delete(observer)
  end

  def notify_observers
    @observers.each do |observer|
      observer.update(self)
    end
  end
end
```

Using the new `Subject` module is simplicity itself. We include the module and call `notify_observers` when something changes:

```
class Employee
  include Subject

  attr_reader :name, :address
  attr_reader :salary
```

```
  def initialize( name, title, salary)
   super()
   @name = name
   @title = title
   @salary = salary
  end

  def salary=(new_salary)
    @salary = new_salary
    notify_observers
  end
end
```

By including the `Subject` module, our new `Employee` class gains all of the `Subject` methods: It is now fully equipped to play the subject in the Observer pattern. Note that we needed to call `super()` in the `initialize` method of `Employee`, which has the effect of calling `initialize` in the `Subject` module.[3]

Building our own `Subject` module was great fun and a good exercise in creating a mixin module. But would you really want to use the `Subject` that we just cooked up? Probably not. The Ruby standard library comes with a fine, prebuilt `Observable` module that provides all of the support you need to make your object, well, observable. Using it is not all that different from using the `Subject` module that we built:

```
require 'observer'

class Employee
  include Observable

  attr_reader :name, :address
  attr_reader :salary

  def initialize( name, title, salary)
   @name = name
   @title = title
   @salary = salary
  end
```

3. Calling `super()` is one of the few places in Ruby where you need to supply the parentheses for an empty argument list. Calling `super` the way we do here, with the parentheses, calls the method in the superclass with no arguments. If you omit the parentheses, you will be calling `super` with the original set of arguments to the current method—in this case, `name`, `title`, `salary`, and `payroll_manager`.

```
  def salary=(new_salary)
    @salary = new_salary
    changed
    notify_observers(self)
  end
end
```

The standard `Observable` module does feature one twist that we omitted from our hand-built version. To cut down on redundant notifications to the observers, the standard `Observable` module requires that you call the `changed` method before you call `notify_observers`. The `changed` method sets a Boolean flag that says the object really has changed; the `notify_observers` method will not actually make any notifications if the changed flag is not set to `true`. Each call to `notify_observers` sets the changed flag back to `false`.

Code Blocks as Observers

A common Ruby variation on the Observer pattern is our old friend the code block. The code becomes a lot simpler if we can just pass in a code block as our listener. Because the Ruby library `Observable` does not support code blocks, perhaps we can find a use for a slightly modified version of our `Subject` module after all:

```
module Subject
  def initialize
    @observers=[]
  end

  def add_observer(&observer)
    @observers << observer
  end

  def delete_observer(observer)
    @observers.delete(observer)
  end

  def notify_observers
    @observers.each do |observer|
      observer.call(self)
    end
  end
end
```

```
class Employee
  include Subject

  attr_accessor :name, :title, :salary

  def initialize( name, title, salary )
    super()
    @name = name
    @title = title
    @salary = salary
  end

  def salary=(new_salary)
    @salary = new_salary
    notify_observers
  end
end
```

The advantage of using code blocks as observers is that they simplify the code; we no longer need a separate class for the observers. To add an observer, we just call `add_observer` and pass in a code block:

```
fred = Employee.new('Fred', 'Crane Operator', 30000)

fred.add_observer do |changed_employee|
    puts("Cut a new check for #{changed_employee.name}!")
    puts("His salary is now #{changed_employee.salary}!")
end
```

This example passes a two-line code block as an observer into `Employee` object. By the time those two lines reach the `Employee` object, they are all wrapped up in a convenient `Proc` object and are set to act as a ready-made observer. When Fred's salary changes, the `Employee` object calls the `call` method on the `Proc`, and the two `puts` get fired.

Variations on the Observer Pattern

The key decisions that you need to make when implementing the Observer pattern all center on the interface between the subject and the observer. At the simple end of the spectrum, you might do what we did in the example above: Just have a single method

in the observer whose only argument is the subject. The GoF term for this strategy is the *pull* method, because the observers have to pull whatever details about the change that they need out of the subject. The other possibility—logically enough termed the *push* method—has the subject send the observers a lot of details about the change:

```
observer.update(self, :salary_changed, old_salary, new_salary)
```

We can even define different update methods for different events. For example, we could have one method for a salary update

```
observer.update_salary(self, old_salary, new_salary)
```

and a different method for title changes

```
observer.update_title(self, old_title, new_title)
```

The advantage in providing more details is that the observers do not have to work quite as hard to keep track of what is going on. The disadvantage of the push model is that if all of the observers are not interested in all of the details, then the work of passing the data around goes for naught.

Using and Abusing the Observer Pattern

Most of the problems that come up in using the Observer pattern revolve around the frequency and timing of the updates. Sometimes the sheer volume of updates can be a problem. For example, an observer might register with a subject, unaware that the subject is going to spew out thousands of updates each second. The subject class can help with all of this by avoiding broadcasting redundant updates. Just because someone updates an object, it does not mean that anything really changed. Remember the `salary=` method on the `Employee` object? We probably should not notify the observers if nothing has actually changed:

```
def salary=(new_salary)
  old_salary = @salary
  @salary = new_salary
  if old_salary != new_salary
    changed
    notify_observers(self)
  end
end
```

Another potential problem lies in the consistency of the subject as it informs its observers of changes. Imagine we enhance our employee example a bit so that it informs its observers of changes in an employee's title as well as his or her salary:

```
def title=(new_title)
  old_title = @title
  @title = new_title
  if old_title != new_title
    changed = true
    notify_observers(self)
  end
end
```

Now imagine that Fred gets a big promotion and a big raise to go along with it. We might code that as follows:

```
fred = Employee.new("Fred", "Crane Operator", 30000)

fred.salary = 1000000
# Warning! Inconsistent state here!
fred.title = 'Vice President of Sales'
```

The trouble with this approach is that because he receives his raise before his new title takes effect, Fred will briefly be the highest-paid crane operator in the world. This would not matter, except that all of our observers are listening and experiencing that inconsistent state. You can deal with this problem by not informing the observers until a consistent set of changes is complete:

```
# Don't inform the observers just yet

fred.salary = 1000000
fred.title = 'Vice President of Sales'

# Now inform the observers!

fred.changes_complete
```

One final thing to look out for is badly behaved observers. Although we have used the analogy of the subject delivering news to its observer, we are really talking about

one object calling a method on another object. What happens if you update an observer with the news that Fred has gotten a raise, and that observer responds by raising an exception? Do you simply log the exception and soldier on, or do you do something more drastic? There is no standard answer: It really depends on your specific application and the amount of confidence you have in your observers.

Observers in the Wild

The Observer pattern is not hard to find in the Ruby code base. It is, for example, used by `ActiveRecord`. `ActiveRecord` clients that want to stay informed of the goings on as database records are created, read, written, and deleted can define observers that look like this:[4]

```
class EmployeeObserver < ActiveRecord::Observer
  def after_create(employee)
    # New employee record created
  end

  def after_update(employee)
    # Employee record updated
  end

  def after_destroy(employee)
    # Employee record deleted
  end
end
```

In a nice example of the Convention Over Configuration pattern (see Chapter 18), `ActiveRecord` does not require you to register your observer: It just figures out that `EmployeeObserver` is there to observe `Employees`, based on the class name.

You can find an example of the code block-based observer in REXML, the XML parsing package that ships as part of the standard Ruby library. The REXML SAX2Parser class is a streaming XML parser: It will read an XML file and you are welcome to add observers that will be informed when specific XML constructs get read.

4. If the `ActiveRecord::Observer` syntax looks a little odd to you, it is because we haven't talked about it. This syntax exists because the `Observer` class is defined inside of a module and you need the `::` to dig inside the module to get to the class.

While SAX2Parser supports the more formal, separate object observer style, you can also pass in code blocks that act as your observer:

```
require 'rexml/parsers/sax2parser'
require 'rexml/sax2listener'

#
# Create an XML parser for our data
#
xml = File.read('data.xml')
parser = REXML::Parsers::SAX2Parser.new( xml )

#
# Add some observers to listen for start and end elements...
#
parser.listen( :start_element ) do |uri, local, qname, attrs|
  puts("start element: #{local}")
end

parser.listen( :end_element ) do |uri, local, qname|
  puts("end element #{local}")
end

#
# Parse the XML
#
parser.parse
```

Feed the code above an XML file, and you can sit back and observe as the elements go flying by.

Wrapping Up

The Observer pattern allows you to build components that know about the activities of other components without having to tightly couple everything together in an unmanageable mess of code-flavored spaghetti. By creating a clean interface between the source of the news (the observable object) and the consumer of that news (the observers), the Observer pattern moves the news without tangling things up.

Most of the work in implementing the Observer pattern occurs in the subject or observable class. In Ruby, we can factor that mechanism out into either a superclass

or (more likely) a module. The interface between observer and observable can be a complex as you like, but if you are building a simple observer, code blocks work well.

As pointed out in Chapter 4, the Observer pattern and the Strategy pattern look a bit alike: Both feature an object (called the observable in the Observer pattern and the context in the Strategy pattern) that makes calls out to some other object (either the observer or the strategy). The difference is mostly one of intent. In the case of the observer, we are informing the other object of the events occurring back at the observable. In the case of the strategy, we are getting the strategy object to do some computing.

The Observer pattern also serves more or less the same function as hook methods in the Template Method pattern; both are there to keep some object informed of current events. The difference is, of course, that the Template Method pattern will only talk to its relatives: If you are not a subclass, you will not get any news from a Template Method pattern hook.