23 Architectural Pattern: Publisher-Subscriber

We now consider a design pattern a bit different than the ones we have seen until now. It is more *architectural*, in the sense that it pertains to how classes are put together to implement an application.

The motivating scenario is as follows. Suppose we have an object in the system that is in charge of generating news of interest for the rest of the application. For instance, perhaps it is in charge of keeping track of user input, and tells the rest of the application whenever the user does something of interest. Or, it is in charge of maintaining a clock, and tells the rest of the application whenever the clock ticks one time step. Is there a general approach for handling this kind of thing?

If we analyze the situation carefully, you’ll notice that we have two sorts of entities around: a *publisher* that is in charge of publishing or generating items of interest to the rest of the application, and the dual *subscribers* that are the parts of the application that are interested in getting these updates.\(^\text{22}\)

Think about the operations that we would like to support on subscribers, first. Well, the main thing we want a subscriber to be able to do is to be notified when a news item is published. Thus, this calls for a subscriber implementing the following trait, parameterized by a type \(D\) of values conveyed during the notification (e.g., the news item itself).

```-scala
trait Subscriber[D] {
  def notify (data:D):Unit = ()
}
```

Note that we provide a default implementation for `notify()`, namely, do nothing.

What about the other end? What do we want a publisher to do? Mainly, we have to be able to *subscribe* (or register) a subscriber, so that that subscriber can be notified when a news item is produced. The other operation, naturally enough, is to *broadcast* (or publish) a piece of data, which should let every subscriber know that the data has been produced. When notifying a subscriber, we will also pass a value (perhaps the news item in question). This leads to the following trait that a publisher should implement, parameterized over a type \(D\) of values to pass when notifying a subscriber.

\(^\text{22}\)The Publisher-Subscriber pattern is sometimes called the Observer pattern, where publishers are called observables, and subscribers are called observers.
trait Publisher[D] {
  def subscribe (s:Subscriber[D]):Unit
  def broadcast (data:D):Unit
}

And that’s it. These two traits together define the Publisher-Subscriber design pattern.

Let’s look at an example. Suppose that the publisher we care about is a loop that simply queries an input string from the user, and notifies all subscribers that a new string has been input, passing that string along as the notification value.

Here is the class for the input loop, implementing the Publisher[String] trait.

object InputLoop {

  def create ():InputLoop = new InputLoopImpl

  private class InputLoopImpl extends InputLoop {

    private var subscribers:List[Subscriber[String]] = List.empty()

    def subscribe (sub:Subscriber[String]):Unit = {
      subscribers = List.cons(sub,subscribers)
    }

    def broadcast (data:String):Unit = {
      def notifySubscriber (sub:Subscriber[String]):Unit =
        sub.notify(data)
      subscribers.foreach(notifySubscriber)
    }

    def loop ():Unit = {
      val s:String = readLine("> ")
      broadcast(s)
      loop()
    }
  }

  abstract class InputLoop extends Publisher[String] {
    def loop ():Unit
  }
}
Note that we are using an implementation of List equipped with a `foreach` method, taking a function to apply to every element of the list. The `loop()` method simply repeatedly queries a string from the user, and notifies all observers of that string. Note that there is no way built into the loop to actually terminate the loop. We’ll see how to deal with that shortly. The subscribers are recorded in a `List[Subscriber[String]]`, which is initially empty. Registering a new subscriber is a simple matter of adding that subscriber to the list. Notifying the subscribers is a simple matter of walking over the list, calling the `notify()` method of each subscriber in the list.

Just to have something concrete, here is how we launch the loop.

```scala
val il: InputLoop = InputLoop.create()
il.loop()
```

Of course, this does nothing useful. It simply repeatedly gets a string from the user, and does absolutely nothing with it:

```
> 10
> 20
> 30
> foo
> bar
```

Let’s define some subscribers, then. The first subscriber is a simple subscriber that echoes the input string back to the user. Since it is a subscriber and we want it to work with the `InputLoop` class, it implements the `Subscriber[String]` interface:

```scala
object Echo {
  def create (text: String): Subscriber[String] = new EchoImpl(text)

  private class EchoImpl (text: String) extends Subscriber[String] {
    override def notify (data: String): Unit =
      println(text + data);
  }
}
```

Another subscriber we can define is one that checks whether the input string is a specific string (in this case, the string `quit`), and does something accordingly (in this case, quit the application).

```scala
object Quit {
```
def create ():Subscriber[String] = new QuitImpl

private class QuitImpl extends Subscriber[String] {
    override def notify (data: String) {
        if (data.startsWith("quit"))
            System.exit(0)
    }
}

Finally, a more general subscriber than can print a response for any particular input.

object Response {
    def create (s1: String, s2: String): Subscriber[String] = new ResponseImpl(s1, s2)

    private class ResponseImpl (ifThis: String, thenThat: String) extends Subscriber[String] {
        override def notify (data: String): Unit = {
            if (data == ifThis)
                println(thenThat)
        }
    }
}

Now, if we subscribe those subscribers before invoking the loop() method of a newly created InputLoop:

    val il: InputLoop = InputLoop.create()
    il.subscribe(Quit.create())
    il.subscribe(Echo.create("Input = "))
    il.subscribe(Response.create("foo", "bar"))
    il.loop()

we get the following sample output:

> 10
Input = 10
> 20
Input = 20
> foo
bar
Input = foo
> quit
Input = quit

It is also easy to add a subscriber that recognizes URLs and reads off the corresponding web page. Here is such a subscriber, using some of the Java networking libraries:

```java
import java.io._
import java.net._

object Url {

    def create ():Subscriber[String] = new UrlImpl

    private class UrlImpl extends Subscriber[String] {

        def printUrlContent (input: String): Unit = {
            try {
                val url = new URL(input)
                val in =
                    new BufferedReader(new InputStreamReader(url.openStream()))
                var inputLine = in.readLine()
                while (inputLine != null) {
                    println(inputLine)
                    inputLine = in.readLine()
                }
            } catch {
                case e: Exception => println(" Error trying to read URL: " +
                    e.getMessage())
            }
        }

        override def notify (data: String): Unit = {
            if (data.startsWith("http://"))
                printUrlContent(data);
        }
    }
}
```

Tossing it into the input loop:

```scala
val il: InputLoop = InputLoop.create()
```
il.subscribe(Quiit.create())
il.subscribe(Echo.create("Input = "))
il.subscribe(Response.create("foo","bar"))
il.subscribe(Url.create())
il.loop()

and trying it out:

> http://www.ccs.neu.edu/index.html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
  <title>
    College of Computer and Information Science | College of Computer and Information Science
  </title>
  <link href="ccis/css/ccis.css" rel="stylesheet" type="text/css" />
  <script src="/ccis/scripts/swfobject.js" type="text/javascript">
    <script type="text/javascript">
      swfobject.embedSWF("flash/ccis_home_slideshow.swf", "flashcontent",
    </script>
  </script>
</head>

<body>

<div id="toplinks">
  <span class="toplinks-inside"><a href="index.html" class="toplinks-link"/>
</div> <!-- end toplinks -->

<div id="header">
  <div id="logo">
    <h1 id="header-image">
      <a href="/ccis/index.html"><span>Northeastern University College of Computer and Information Science</span></a>
    </h1>
  </div> <!-- end logo -->

  <div id="usernav">
    301
and it continues like that for a long time.

Slightly harder is to implement a subscriber that can read a URL and extracts text that actually looks good printed out, by interpreting the HTML. (Try it if you’re bored...)

The next example subscriber is really an subscriber transformer: it puts itself between a publisher and another subscriber (which I’ll call the underlying subscriber) and manages substitutions of the notifications from the publisher to the underlying subscriber. The subscriber intercept notifications such as $ foo = bar$, which get recorded as substitutions to be remembered. Any other subsequent notification from the publisher gets scanned for occurrences of ${foo}$, which are replaced by bar, before the resulting string is passed to the underlying subscriber.

```scala
object Substitution {
  def create (s:Subscriber[String]):Subscriber[String] = new SubstImpl(s)

  private class SubstImpl (s:Subscriber[String])
    extends Subscriber[String] {

    private var map = SubstitutionMap.empty()

    override def notify (data:String):Unit =
      if (data.startsWith($ " ")) {
        val res = data.split(" +",4)
        if (res.size >= 4 && res(2)=="=")
          map = map.add(res(1),res(3))
      } else
```
The above code uses the following implementation of a substitution map, for performing that actual substitutions:

```scala
object SubstitutionMap {

  def empty () : SubstitutionMap = new SubstMap(List.empty())

  private class SubstMap (subs:List[Subst]) extends SubstitutionMap {

    def add (s: String, t: String) : SubstitutionMap =
      new SubstMap(List.cons(new Subst(s, t), subs))

    def subst (s: String) : String =
      subs.foldr((sub: Subst, s: String) => sub.subst(s), s)
  }

  private class Subst (src: String, tgt: String) {

    def subst (s: String) : String = {
      for (i <- 0 to s.size) {
        if (s.startsWith($"\\${" + src + "}\", i))
          return subst(s.substring(0, i) + tgt + s.substring(i + 3 + src.size))
      }
      return s
    }
  }

  abstract class SubstitutionMap {

    def add (s: String, t: String) : SubstitutionMap
    def subst (s: String) : String
  }

  Thus, if we define:

  val il: InputLoop = InputLoop.create()
  val resp = Response.create("knock knock", "who’s there")
  val echo = Echo.create("Input w/ subs = ")

  304
```
then we can try:

> 10
Input = 10
Input w/ subs = 10
> 20
Input = 20
Input w/ subs = 20
> $ foo = bar
Input = $ foo = bar
> $ name = Riccardo
Input = $ name = Riccardo
> This is is ${foo} and ${name} and ${stuff}
Input = This is is ${foo} and ${name} and ${stuff}
Input w/ subs = This is is bar and Riccardo and ${stuff}
> $ k = knock
Input = $ k = knock
> ${k}
Input = ${k}
Input w/ subs = knock
> ${k} ${k}
Input = ${k} ${k}
Input w/ subs = knock knock
who’s there

The above substitution subscriber works, but note the following: if you instantiate multiple substitution subscribers, each servicing a different underlying subscriber, then each of those substitution subscribers intercepts and records the same substitutions. That’s a lot of wasted effort. A better alternative is to have a single substitution subscriber, servicing multiple underlying subscribers. This in fact turns the substitution subscriber into a publisher for other subscribers, instead of just a subscriber wrapping an underlying subscriber. Thus, we see that Publisher-Subscriber patterns can nest, in some sense. Here is the alternate substitution subscriber:

```java
object Substitution2 {
```
def create () : Substitution2 = new SubstImpl

abstract class Substitution2
    extends Publisher[String] with Subscriber[String]

private class SubstImpl extends Substitution2 {

    private var map = SubstitutionMap.empty()
    private var subscribers = List.empty[Subscriber[String]]()

    def subscribe (s:Subscriber[String]):Unit =
        subscribers = List.cons(s,subscribers)

    def publish (data:String):Unit =
        for (sub <- subscribers)
            sub.notify(data)

    override def notify (data:String):Unit =
        if (data.startsWith("\$ ")) {
            val res = data.split(" +",4)
            if (res.size >= 4 && res(2)=="=")
                map = map.add(res(1),res(3))
        } else
            publish(map.subst(data))
}

Note the definition of abstract class Substitution2 whose sole role is to serve as a way to say that the result of create() is a class implementing both Publisher[String] and Subscriber[String]. We can test a variant of the preceding code:

val il:InputLoop = InputLoop.create()
val subst = Substitution2.create()
il.subscribe(subst)
subst.subscribe(Response.create("knock knock","who’s there"))
subst.subscribe(Echo.create("Input w/ subs = "))
il.subscribe(Response.create("foo","bar"))
il.subscribe(Quiet.create())
il.subscribe(Echo.create("Input = "))
il.loop()

and the behavior is the same as with the Substitution1 subscriber — except we only need
to create a single instance of Subscriber2.

The basic publisher/subscriber pattern described here often needs to be extended to be more generally useful. For instance, it is sometimes convenient to pass more arguments to the notification function of the subscriber. To illustrate, suppose that we are interested, in the above input loop example, in defining a subscriber that can create new responses (in the sense of the Response subscriber) on-the-fly. The subscriber I have in mind, which we may call a ResponseCreator subscriber, will recognize an input of the form response $x \ y$ and add a Response subscriber that responds $y$ to $x$ at the input loop. In order to do that, the notify() operation of the ResponseCreator subscriber needs to get not only the data from the publisher, but also the actual publisher, so that it can subscribe a new Response subscriber to that publisher. So we’re looking at the following updated Subscriber trait:

```scala
trait Subscriber[D] {
  def notify (data:D):Unit = ()
  def notify (pub:Publisher[D], data:D):Unit = notify(data)
}
```

We override notify() so that it can also accept a publisher. By default, notify() with a publisher argument will simply invoke notify() without a publisher argument — so that if we only implement notify() without a publisher argument, everything should work as before.

We modify the broadcast() implementation in InputLoop to pass the current publisher as an argument to the notify() call to each subscriber:

```scala
  def broadcast (data:String):Unit = {
    def notifySubscriber(sub:Subscriber[String]):Unit =
      sub.notify(this, data)
    subscribers.foreach(callNotify)
  }
```

And now we can implement ResponseCreator:

```scala
object ResponseCreator {

  def create ():Subscriber[String] = new ResponseCreatorImpl()

  private class ResponseCreatorImpl () extends Subscriber[String] {

    override def notify (pub:Publisher[String], data:String):Unit = {
      if (data.startsWith("response")) {
        val res = data.split(" ")
      }
  }
```
and if we subscribe it in Main:

```
    il.subscribe(ResponseCreator.create())
```

we get as a sample output (in the presence of an Echo subscriber with prefix text You entered: ):

```
> hello
You entered: hello
> response hello yes?
Creating new response from hello to yes?
You entered: response hello yes?
> hello
yes?
You entered: hello
> help
You entered: help
> response help me?
Creating new response from help to me?
You entered: response help me?
> help
me?
You entered: help
```

The Publisher-Subscriber pattern is central to much of GUI programming: the application is a tight loop (often called an event loop) that simply collects inputs from the user such as mouse movement, mouse button clicks, and key presses, and notifies its subscribers of those events. Those subscribers, which are graphical elements such as buttons and windows and the likes, react to those events that concerns them (such as a mouse click over their surface), and affect the application accordingly.

More complicated forms of publisher/subscriber relationships can be layered on top of the basic pattern I described here. For instance, we may be interested in unsubscribing subscribers
(which can have an interesting effect when this unsubscription happens in the context of a notification of another subscriber!), or we may be interesting in defining different categories of news that we can notify subscribers with, so that when a subscriber registers with a publisher it gets to tell the publisher what category of news it wants to be notified about.