20 Architectural Pattern: Model-View-Controller

Let's look at another architectural pattern. Recall, from our lecture on the Publisher-Subscriber pattern, that architectural patterns pertain to the structuring of an application as a whole. Such patterns are useful to figure out where to start when implementing an application.

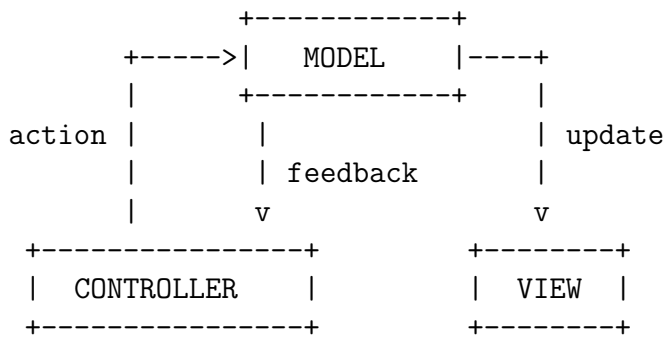
The *Model-View-Controller* (MVC for short) pattern goes back to the 70s, in the Smalltalk community. The idea of the MVC pattern is to separate the following aspects of an application:

- the *model*: capturing the state of an application and its internal logic dictating how its state changes when actions are performed;
- the *controller*: capturing how the model is informed of actions that are performed (generally, this is driven by input from the user);
- the *view*: capturing how the application displays its results to the user.

Why might we want to do that? This decoupling is useful for modern applications which have sometimes specific requirements:

- they may need to work on multiple platforms or operating systems — in which case the model is usually the same, but the details of how to get the view working for a particular operating system may be different, or how to get input from the user; even in the case of a single operating system, we may want an application to have a view that suitable for a mobile phone, and a view suitable for a desktop.
- they may require different models based on the size of data they operate on, or where they store their data (for instance, there may be a need to backup the application state for server application, requiring a different model than one for which the state is completely in memory.)
- they may need to work on the web, there the view (generally HTML code rendered in the browser) can change pretty drastically based on other requirements.

The interaction between the three components is slightly different depending on the application and who you talk to, but a first approximation can be captured by the following diagram:



Roughly, the controller is in charge of directing the model by invoking actions, and the model informs the view when the state has changed so that the view can update its display (thus, the view is really an observer of the model). Often, the model sends feedback to the controller, because changes in the state of the application may change the possibilities open to the controller in terms of what actions can be performed.

20.1 An Example: A Maze Game

The best way to illustrate an architectural pattern is to give a sample application implemented using the pattern. The example I have is a simple maze game. The code for it is available on the website, and I'll just give the highlights here.

The game: you're in a maze, and you have the find the exit. Movement in the maze involves moving forward, moving back, turning left, and turning right. For simplicity, we'll consider a model of the maze where the maze is on a square grid, movement back and forth is done one square at a time, and the directions one can be facing are one of north, east, south, or west.

Let's first describe the interfaces capturing the functionality of the model, the controller, and the view for the Maze game.

```
trait Model {
  def performLeft ():Boolean
  def performRight ():Boolean
  def performForward ():Boolean
  def performBack ():Boolean
}
```

Basically, a model can respond to request to perform actions "turn left", "turn right", "move forward", and "move backwards" applied to the player's position and direction. When we create a model, we will supply it with a view and a controller, so that it can interact with them.

```
trait Controller {
    def attachToModel (m:Model):Unit
    def run ():Unit
}
```

A controller, on the other hand, once created, can either be attached to a model (when the model is created, it will generally attach itself to the controller it is passed as an argument — see below — we need to do this kind of convolution because there is a dependency going both ways between a model and a controller, which both need to know about each other), or it can be "run", meaning that it starts controlling the model.

```
trait View {
    def attachToModel (m:Model):Unit
    def update (vis:Visible):Unit
    def win ():Unit
}
```

A view, just like a controller, can be attached to a model (and for the same reasons), and it can updated (to display the current position of the player and presumably what he or she sees), or it can be told that the game is won, in order to congratulate the player. To ensure that the update does not depend too much on the details of the implementation of the model, we use an interface `Visible` to abstract away the information that the model sends to the view about what the player sees. For simplicity, I defined `Visible` to be a set of predicates that lets the view ask about what openings are visible in the current square (to the left, to the right, to the front), which openings are visible one square in front of the player, if visible (to the front left, to the front right, to the front front), and one square beyond, if visible (to the front front left, to the front front right, to the front front front). The view can also query for whether the square in front or the one beyond is a winning square.

```
trait Visible {
  def direction ():Direction

  def left ():Boolean
  def right ():Boolean
  def front ():Boolean
  def frontleft ():Boolean
  def frontright ():Boolean
  def frontfront ():Boolean
  def frontfrontleft ():Boolean
  def frontfrontright ():Boolean
  def frontfrontfront ():Boolean

  def frontwin ():Boolean
```

```
  def frontfrontwin ():Boolean
}
```

And that's it. That defines the interactions between the model, the controller, and the view, in the context of the Maze game. Everything else is filling in the details.

We have interface, now we need implementations for those interfaces. First, let's define the basic model for the maze, which represents the maze as a set of connected cells, where each cell can be connected to up to four other cells, one in each of the four cardinal directions.

```
class BasicModel (view:View, controller:Controller) extends Model  {

  /* The current position of the player in the maze */
  private var cell:Cell = initializeMaze()

  /* The current direction faced by the player in the maze */
  private var direction:Direction = Direction.NORTH

  // constructor
  view.attachToModel(this)
  controller.attachToModel(this)
  // initial display of information
  updateView()



  private class BasicVisible (d:Direction,
                              l:Boolean,r:Boolean,f:Boolean,
                              fl:Boolean,fr:Boolean,ff:Boolean,
                              ffl:Boolean,ffr:Boolean,fff:Boolean,
                              fw:Boolean,ffw:Boolean) extends Visible {

    def direction ():Direction = d
    def left ():Boolean = l
    def right ():Boolean = r
    def front ():Boolean = f
    def frontleft ():Boolean = fl
    def frontright ():Boolean = fr
    def frontfront ():Boolean = ff
    def frontfrontleft ():Boolean = ffl
    def frontfrontright ():Boolean = ffr
    def frontfrontfront ():Boolean = fff
    def frontwin ():Boolean = fw
```

```
  def frontfrontwin ():Boolean = ffw
}

def isOpening (oc:Option[Cell]):Boolean = !(oc.isNone())

def updateView ():Unit = {
  val l = isOpening(cell.exit(direction.left()))
  val r = isOpening(cell.exit(direction.right()))
  val front = cell.exit(direction)
  if (!isOpening(front)) {
    view.update(new BasicVisible(direction,l,r,false,false,false,false,
                                 false,false,false,false,false))
    return
  }
  val fl = isOpening(front.valOf().exit(direction.left()))
  val fr = isOpening(front.valOf().exit(direction.right()))
  val ffront = front.valOf().exit(direction)
  val fwin = front.valOf().isWin()
  if (!isOpening(ffront)) {
    view.update(new BasicVisible(direction,l,r,true,fl,fr,false,
                                 false,false,false,fwin,false))
    return
  }
  val ffl = isOpening(ffront.valOf().exit(direction.left()))
  val ffr = isOpening(ffront.valOf().exit(direction.right()))
  val fffront = ffront.valOf().exit(direction)
  val ffwin = ffront.valOf().isWin()
  view.update(new BasicVisible(direction,l,r,true,fl,fr,true,
                               ffl,ffr,isOpening(fffront),
                               fwin,ffwin))
  return
}


private def initializeMaze ():Cell = {

  val cells:Array[Cell] = new Array[Cell](8)

  // create cells
  for (i <- 0 to 7)
    cells(i) = new Cell
  val wincell = new WinCell
```

```
  // connect cells
  cells(0).connect(Direction.SOUTH,cells(1))
  cells(1).connect(Direction.SOUTH,cells(2))
  cells(2).connect(Direction.EAST,cells(3))
  cells(3).connect(Direction.EAST,cells(4))
  cells(4).connect(Direction.NORTH,cells(5))
  cells(5).connect(Direction.NORTH,cells(6))
  cells(6).connect(Direction.WEST,cells(7))
  cells(7).connect(Direction.WEST,cells(0))

  cells(4).connect(Direction.SOUTH,wincell)

  // choose starting cell
  cells(0)
}


def performLeft ():Boolean =  {
  direction=direction.left()
  updateView()
  true
}

def performRight ():Boolean = {
  direction=direction.right()
  updateView()
  true
}

def performForward ():Boolean = {
  val res = cell.exit(direction)
  if (res.isNone())
    false
  else {
    cell=res.valOf()
    updateView()
    if (cell.isWin())
      view.win()
    true
  }
}
```

```
  def performBack ():Boolean = {
    val res = cell.exit(direction.opposite())
    if (res.isNone())
      false
    else {
      cell=res.valOf()
      updateView()
      if (cell.isWin())
        view.win()
      true
    }
  }
}
```

Private fields hold the view and controller the model is connected to, as well as a `Cell` holding the current position of the player in the maze, as well as the direction the player is facing. Directions are implemented as a form of enumerations. If you know Java enumeration types, then you should know that Scala doesn't have them. It has a few different ways of doing enumerations, though, and this below is just one of them, using *case classes* — see the Scala documentation for more information.

```
sealed abstract class Direction {
  def left ():Direction
  def right ():Direction
  def opposite ():Direction
  def ordinal ():Int
}

object Direction {

  case object NORTH extends Direction {
    def left ():Direction = WEST
    def right ():Direction = EAST
    def opposite ():Direction = SOUTH
    def ordinal ():Int = 0
  }

  case object SOUTH extends Direction {
    def left ():Direction = EAST
    def right ():Direction = WEST
    def opposite ():Direction = NORTH
    def ordinal ():Int = 2
```

```
  }

  case object EAST extends Direction {
    def left ():Direction = NORTH
    def right ():Direction = SOUTH
    def opposite ():Direction = WEST
    def ordinal ():Int = 1
  }

  case object WEST extends Direction {
    def left ():Direction = SOUTH
    def right ():Direction = NORTH
    def opposite ():Direction = EAST
    def ordinal ():Int = 3
  }
}
```

Intuitively, there are four objects subtyping `Description`, `NORTH`, `SOUTH`, `EAST`, and `WEST`, and there are four methods defined for those objects: `left()`, `right()`, `opposite()`, that gives us the direction 90 degrees counter-clockwise, 90 degrees clockwise, and 180 degrees from the direction these methods are applied to, and `ordinal()` converting the direction to a number.

Back to `BasicModel`: the constructor of the class attaches the created instance to the view and to the controller, after having initialized the maze — basically creating and connecting the cells making up the maze, which also sets the starting cell and the starting direction — and then calls helper method `updateView()` that will give the first rendering to the player. That method takes the current cell and the cells in the direction the player is facing and uses that information to create an instance of `BasicVisible`, which is a class implementing the `Visible` interface, and that can be passed to the view for displaying purposes.

Operations `performLeft()`, `performRight()`, `performForward()`, and `performBack()` update the position or direction of the player, updating the view after every action.

I will not say much about cells, except to give their signature (the code is on the website):

```
    def connect (d:Direction, c:Cell):Unit
    def exit (d:Direction d):Option[Cell]
    def isWin ():Boolean
```

Once created, a cell is not connected to any other cell. Calling `connect(d,c)` mutates the cell so that its exit in the `d` direction leads to cell `c` — this also has the effect of mutating cell `c` so that its exit in the `d.opposite()` direction leads to the original cell. To get the exit of a cell in a particular direction `d`, call `exit(d)` — the result is either `none()` if there is no exit in that direction (i.e., it is a wall), or `some(c)` where `c` is the cell that the direction leads to.

Method `isWin()` checks whether the cell is a winning cell. (Winning cells are implemented as a subclass of cells.)

So this essentially takes care of the model. What we're missing is a controller and a view. First, the controller. I'm going to describe a simple controller here, driven by commands from the player that he or she types at a simple command line.

```
class TextController extends Controller {

  private var maze:Option[Model] = Option.none()

  def attachToModel (m:Model):Unit =
    maze = Option.some(m)

  private def process (response:String):Boolean =
    if (response=="forward" || response=="f")
      maze.valOf().performForward()
    else if (response=="back" || response=="b")
      maze.valOf().performBack()
    else if (response=="left" || response=="l")
      maze.valOf().performLeft()
    else if (response=="right" || response=="r")
      maze.valOf().performRight()
    else if (response=="quit" || response=="q") {
      System.exit(0)
      true
    }
    else
      false

  def run ():Unit = {
    if (maze.isNone())
      throw new Error("TextController not initialized with model")

    while (true) {
      print("> ")
      val response = readLine()
      if (!process(response))
        println("Cannot perform action")
    }
  }
}
```

Nothing really special here, it is similar to other command loops we have played with in the last few lectures. The thing is that when a response is recognized as a command, the appropriate method from the model is called, from that model that is attached to the controller.

So the controller sends actions to the model, which invokes a view to display results to the player. Here is a simple view, that textually describes what the player sees in his immediate vicinity:

```
class TextView extends View {

  private var maze:Option[Model] = Option.none()

  def attachToModel (m:Model):Unit =
    maze = Option.some(m)

  private def status (b:Boolean):String =
    if (b)
      "open"
    else
      "blocked"

  def update (vis:Visible):Unit = {
    println("Facing: " + vis.direction())
    println("Passage in front is " + status(vis.front()))
    println("Passage on left  is " + status(vis.left()))
    println("Passage on right is " + status(vis.right()))
    if (vis.frontwin())
      println("Winning square is up front")
  }

  def win ():Unit = {
    println("You won")
    System.exit(0)
  }
}
```

It is pretty much self-explanatory.

Now that we have all the components of the application, we can connect them together to create the Maze game with textual input and output:

```
object MainText {

    def main (argv:Array[String]):Unit = {
```

```
      val v = new TextView
      val c = new TextController
      val m = new BasicModel(v,c)

      c.run()
    }
}
```

And that's it. Running it gives:

```
Facing: NORTH
Passage in front is blocked
Passage on left  is blocked
Passage on right is open
> r
Facing: EAST
Passage in front is open
Passage on left  is blocked
Passage on right is open
> f
Facing: EAST
Passage in front is open
Passage on left  is blocked
Passage on right is blocked
> f
Facing: EAST
Passage in front is blocked
Passage on left  is blocked
Passage on right is open
> r
Facing: SOUTH
Passage in front is open
Passage on left  is blocked
Passage on right is open
> f
Facing: SOUTH
Passage in front is open
Passage on left  is blocked
Passage on right is blocked
> f
Facing: SOUTH
```

```
Passage in front is open
Passage on left  is blocked
Passage on right is open
Winning square is up front
> f
Facing: SOUTH
Passage in front is blocked
Passage on left  is blocked
Passage on right is blocked
You won
```

What's more exciting is that we can change the view or the controller, and get different variants of the game. I won't show it here, but in the code I give on the website is a different view, the `GUIView`, which displays what the player sees as a 3D-ish rendering. Creating a version of the Maze game with that view is a simple matter of using:

```
object MainGUI {

  def main (argv:Array[String]):Unit = {

    val v = new GUIView
    val c = new GUIController
    val m = new BasicModel(v,c)

    c.run()
  }
}
```

Very little change, but the game feels quite different. Even more so if we replace the `TextController` by one that can recognize key strokes in the display window, such as the `GUIController` I give on the website. Have a look, although to be honest the GUI code is somewhat clunky, and does not scale well. Nevertheless, it illustrates my point.

Finally, we can play interesting games with views. Here is a special view that multiplexes updates to two sub-views:

```
class DualView (view1:View, view2:View) extends View {

  def attachToModel (m:Model):Unit = {
    view1.attachToModel(m)
    view2.attachToModel(m)
  }

  def update (vis:Visible):Unit = {
```

```
    view1.update(vis)
    view2.update(vis)
  }

  def win ():Unit = {
    view1.win()
    view2.win()
  }
}
```

With this, we can very quickly implement a version of the Maze game that both displays a GUI window, and that gives a textual output of the player status, as follows:

```
object MainDual {

  def main (argv:Array[String]):Unit = {

    val v1 = new GUIView
    val v2 = new TextView
    val v = new DualView(v1,v2)
    val c = new TextController
    val m = new BasicModel(v,c)

    c.run()
  }
}
```

It is also possible to change the view (and the controller for that matter) to play the game across a network, although that requires a bit more infrastructure, and a client on another machine.

The MVC design pattern is quite flexible, and you'll find different variations on it out there in the wild. For instance, our views, even though they are attached to a particular model, do not actually call into the model. That's because we managed to capture all the information required for a view to update itself in the `Visible` argument. For more complex updates, summarizing the information in such a way is not easily done. In that case, it is possible to simply have the model define a set of *query methods* that the view can call to ask the model about information it can use to update its display. The `update()` method then need not take any argument, since the view will query the model directly.