CSE 1720

Lecture 21

Model-View-Controller

Model View Controller

– a software architecture

– separates the aspects of *program logic* from the aspects of *presentation* and *input handling*

– was first devised in 1979 as part of Smalltalk

- Smalltalk was an early object-oriented language developed at Xerox PARC

The Model

- manages the behavior and data of the application domain
- responds to requests for information about its state (usually from the view)
- responds to instructions to change state (usually from the controller)
- notifies observers (usually views) when the information changes so that they can react.

The View

- renders the model into a form suitable for interaction (as a user interface)
- different views are possible for any given model (e.g., for different purposes)
- multiple views can exist for a single model.

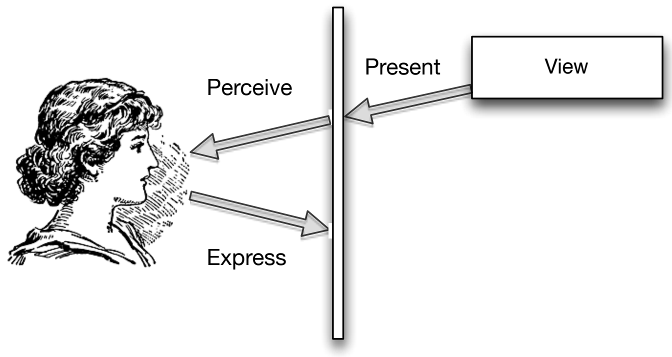
The Controller

- receives user input
- initiates a response by modifying the the model

In our apps…

- each of the model, view, controller are encapsulated by a single class
- this may not be true in other, more complex applications

L15App1



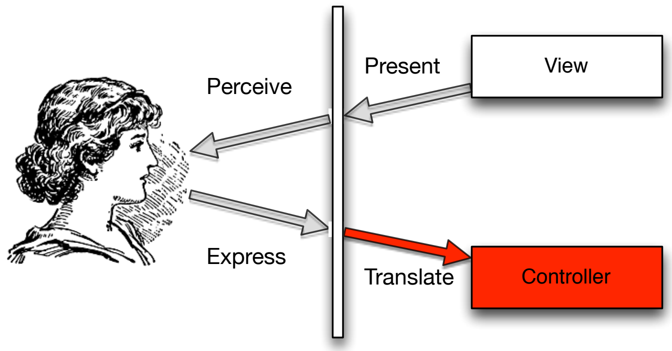
The previous slide represents the architecture of `L15App1`

- The class `L15Frame` encapsulates the view of the system (what the user sees)
- The user has limited interactivity with the view
  - can resize/move/iconify
  - these functions are provided via the window manager (and not the app itself)

Recall: `ObserverMouseMotion`

- the user interacts with the view and this object listens for events that have been dispatched
  - these events are dispatched whenever the user performs mouse motion actions
- this object causes some information to be printed to the console, but **does not actually change the view**
  - as a listener, it is pretty lame (it doesn't do much)
- the other Observer* class definitions implement various other listeners; they fall into the same category
- BUT THEY DO DEMONSTRATE A CONCEPT!

L15App2



The previous slide represents the architecture of `L15App2`

– The class `L15Frame` encapsulates the view of the system (what the user sees)

– The user does interact with a component that fulfills the role of the controller, the `ObserverMouseMotion`

We want to build an app with true GUI-style interactivity…

– Step 1: first, we need to build up the data model that will support the interaction

– This is the motivation behind the classes `PolkaDot` and `PolkaDotDataModel`

- `PolkaDotDataModel` is responsible for keeping track of what should be drawn on the view

- `PolkaDot` encapsulates information about each polka dot that is to be drawn

– This model supports a set of polka dots; if you want your GUI to show other things, you will need a different model!

View delegates to the model…

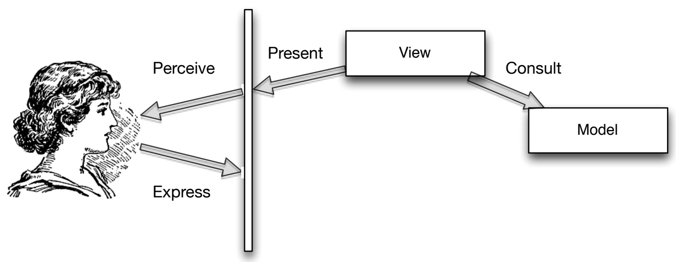
– Step 2: the view must delegate to the model; it knows it needs to paint ***something*** on its components, but ***what*** to paint needs to be derived from the model

– The class `CanvasPanel` implements a class that makes use of such delegation

– the result of this approach is `L20App1`

13

L20App1



14

But L20App1 app is not interactive!

15

We need a controller …

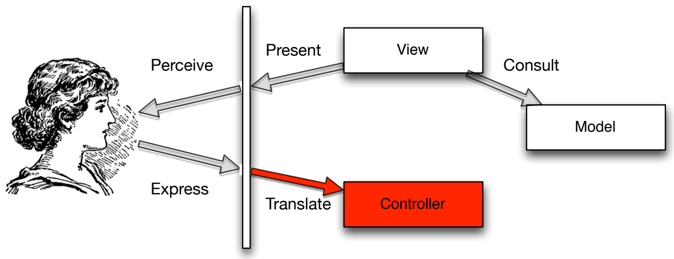
– consider the class `DotControllerTrivial`

– it is a MouseListener
  - it can detect mouse events
  - it does not, however, translate mouse events into any sort of impact on the GUI
  - it just prints info to the console
  - it is just a *trivial* controller

– possibilities for mouse actions,
  - a new polka dot appears
  - the nearest polka dot shrinks/increases
  - the nearest polka dot changes color
  - the nearest polka dot toggles between filled/unfilled

16

L20App2



Let’s use a non-trivial controller…

- consider the class `DotController`
- it is a MouseListener
  - it can detect mouse events
  - for every mouse click action, it generates a new random polka dot and adds it to the model
- Now look at the model
  - any time the state of the model changes, notice that the model **notifies** any and all of its listener

But our view is out of sync with the model…

- the class that implements our view is `L20FrameBasicVersion`
- it takes the model as a parameter to the constructor, but doesn’t coordinate with the model in any meaningful way…
  - the model notifies any listeners that it has changed, but the view is not listening for this

Getting the view in sync with the model…

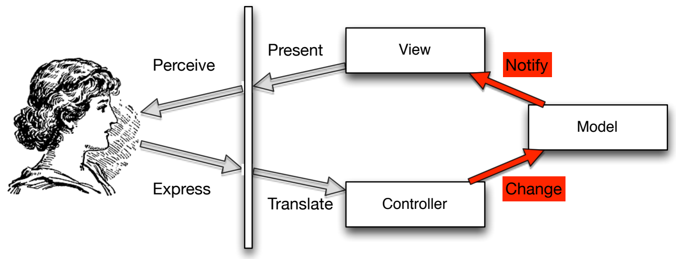
- it is very similar to `L20FrameBasicVersion`, except for the addition of the following:

```
myDataModel.addListener(this);
```

  also, the method `changed()` is implemented

- this means that the view is listening for **change events** that the model may dispatch
  - anytime the model invokes `notifyModelHasChanged()`, the method iterates over all model listeners and invokes their `changed()` method

L20App3



- Now the flow of control is complete
  - The user interacts with the user interface
  - The controller handles the input event
  - The controller notifies the model of the user action
  - The view listens to the model and, upon changes to the model, regenerates itself
  - The user interface waits for further user interactions. This restarts the control flow cycle.
- This is the template for all MVC applications