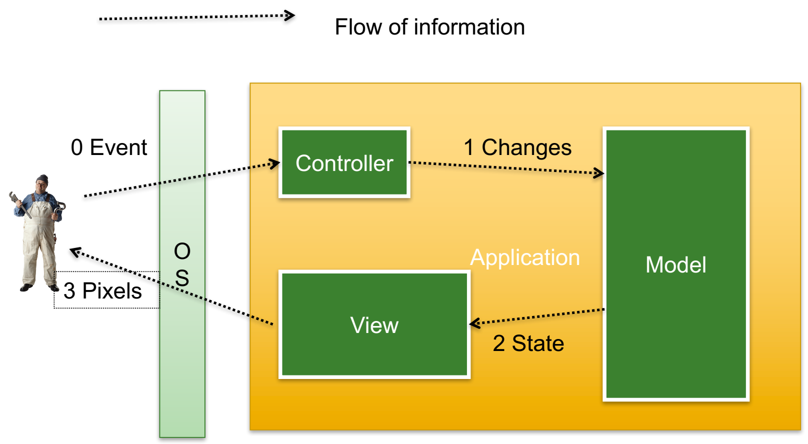
Model-view-controller

An architecture for UI

The flow of information (idealized)



Responsibilities

- Model: Holds state information.
  - It models the user's conception of what it is that they are manipulating or viewing.
  - Should align with the user's mental model.
- View: Presents a visualization of the models state.
  - Views are usually stateless, but might include "view state" such as zoom level, scroll position, selection or highlighting, caret position.

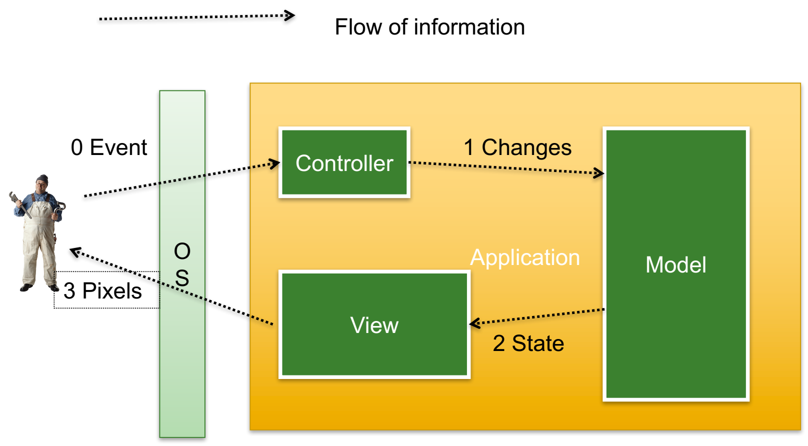
Responsibilities

- Controller: Interprets UI events (mouse events, keyboard events, screen touches, etc.)
  - Turns UI events into changes to the model (and sometimes view state).

MVC Encourages

- Separation of presentation from representation.
- Separation of view from control.
  - Allows these components to be independently extended and, perhaps, reused.

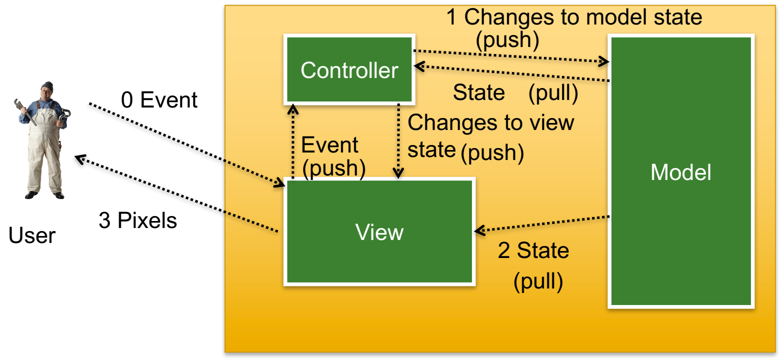
The flow of information (idealized)



Flow of information

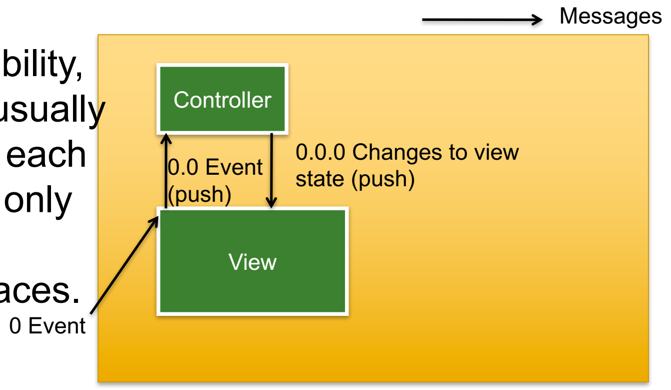
- Often the flow is more complicated because
  1. The underlying GUI system associates events with view objects.
     - E.g. in AWT/Swing. Events are routed through the GUI component the user directs them at.
  2. The controller may need to know the model's state
  3. Some events affect only the view and so should not go through the model.
     - E.g. Scrolling, cursor position, selection.

The flow of information (more realistic)



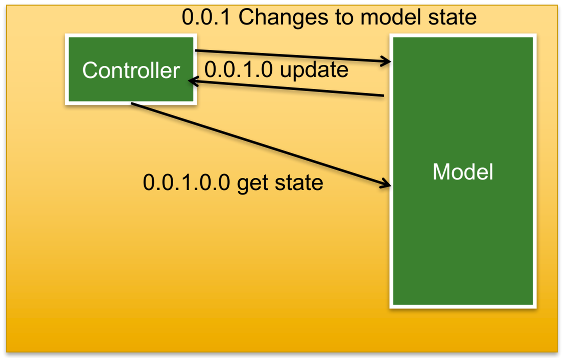
Strategy

- The View uses the Controller as a strategy to help it deal with input events
- For reusability, they usually know each other only via interfaces.



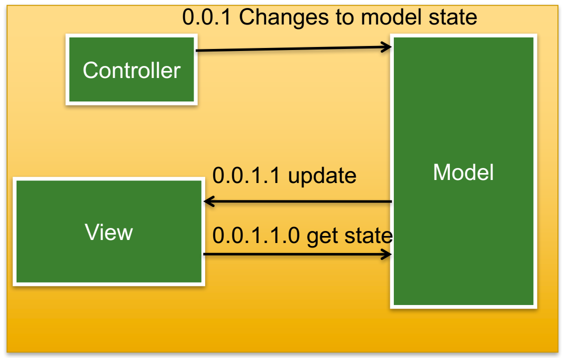
Observer

- The Controller observes the Model so that it is aware of relevant changes to state.



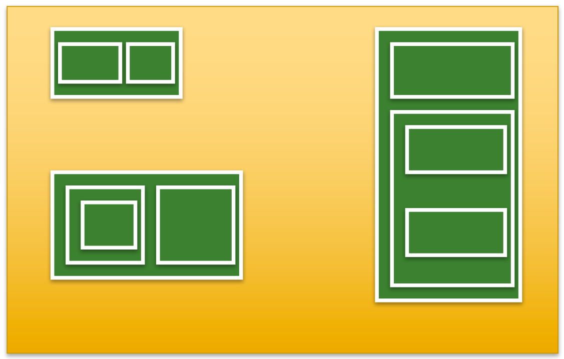
Observer

- The View also observes the Model so that it is aware of relevant changes to state.



Composite and Façade

- The composite pattern is often used in any of the three parts. The model may use Façade.

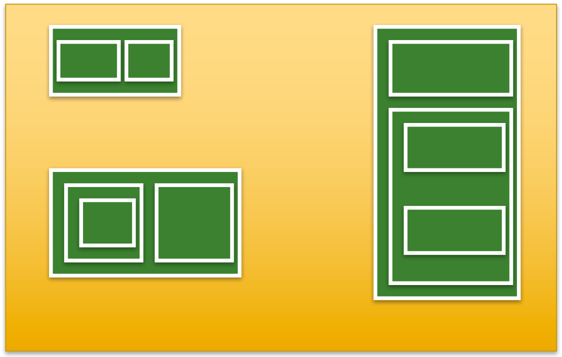


State pattern

- Controllers and models are often state machines and may use the state pattern.

State pattern:

Object-oriented Implementation of state machine. Each state is represented by a different class (all with the same interface---strategy pattern).



Advantages

- Clean separation of presentation (view) from domain modelling (model).
- Clean synchronization. The observer pattern helps keep all views and controllers in sync with the data.

Advantages

- Separation of view from control.
  - The view is typically platform dependent. And events that come to it are typically defined by the platform (e.g. Swing).
  - By separating the controller you can reuse the controller independently of the view.

Case study: The Rat Race game.

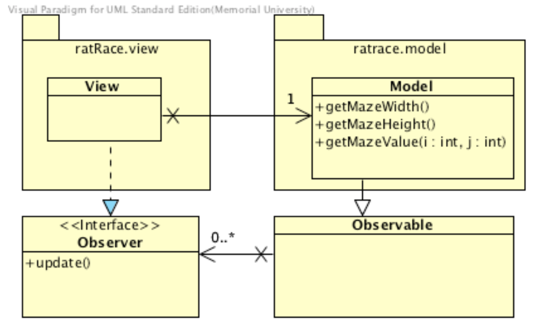
- Model keeps a map of a maze, with a cheese and a rat.
  - The model's interface is in terms of "world coordinates"
- View draws the maze, cheese, and rat.
  - The world—view transformation is a secret of the view.
  - The view must then translate mouse events to world coordinates.

Case study: The Rat Race game.

- Controller
  - Forwards mouse events from the View to the Model.
  - Sends periodic “pulse” events to the model so that it is animated.

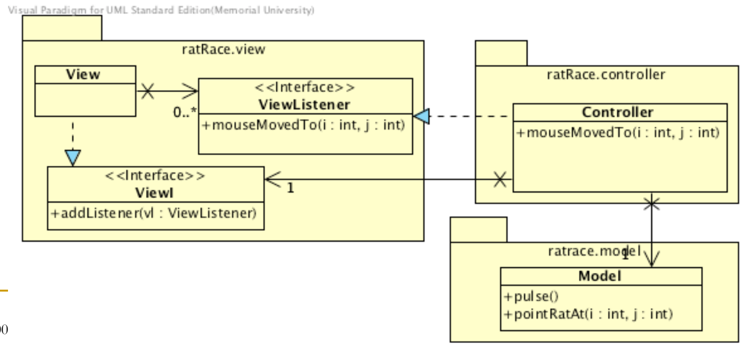
Model and view: Observer

- The model and view relate by the observer pattern



Controller and View: Strategy/Listener

- The controller listens to the view for events and propagates them to the model.
- It also produces pulse events, on its own.


Visual Paradigm for UML Standard Edition(Memorial University)
ratRace.view
View
<<Interface>>
ViewListener
+mouseMovedTo(i : int, j : int)
0..*
<<Interface>>
ViewI
+addListener(vl : ViewListener)
1
ratRace.controller
Controller
+mouseMovedTo(i : int, j : int)
ratrace.model
1
Model
+pulse()
+pointRatAt(i : int, j : int)

What’s missing

- In this case, there was no need to have the controller observe the model
- And there is no need for the controller to send messages to the view (after registering as a listener).

Is the controller needed?

- In simple cases the controller is just forwarding information from the view to the model. So is the controller needed?
  - If the view just sent change messages directly to the model, it would have two responsibilities (display and control), which makes it more complicated.
  - Also the view would be more tightly bound to the model, which makes it less reusable.
  - We might not want one controller per view.
  - Independent testability.

Sources and further reading

- Martin Fowler has an interesting article on styles of UI architecture
  - http://martinfowler.com/eaaDev/uiArchs.html
- Head-First Design Patterns by Freeman, Robson, Bates, and Sierra has a good chapter on MVC

Variations and alternatives

- Trygve Reenskaug and James O. Coplien
  - have an interesting article on what they call DCI (Domain, Context, Interaction). This is not so much on UI design as a challenge to a lot of (bad) OO design.
  - http://www.artima.com/articles/dci_vision.html
- Mike Potel describes the Model-View-Presenter
  - http://www.wildcrest.com/Potel/Portfolio/mvp.pdf
- Martin Fowler on Presentation Models
  - http://martinfowler.com/eaaDev/PresentationModel.html
- MF on Passive View and Supervising Controller
  - http://martinfowler.com/eaaDev/PassiveScreen.html
  - http://martinfowler.com/eaaDev/SupervisingPresenter.html