* Model
  + encapsulates application state
  + responds to state queries
  + exposes applicaiton functionality
  + notifies views of changes
* View
  + renders the models
  + requests updates from models
  + sends user gestures to controller
  + allows controller to select view
* Controller
  + defines application behavior
  + maps user actions to model updates
  + selects view for response
  + one for each functionality

Example: Create a calculator

* CalculatorModel
  + encapsulates a history of operations
  + a current avlue on which the next operation will occur
  + observable(setChanged(), notifyObservers(op);
    - notifies observers when model changes which in this case whenever anew operation occurs
* Operation Class
  + encapsulates two fields: operator type and value
* Controller
  + sits between model and view
  + responds to what happens in the view(when buttons are pushed)
  + instructs view to change when necessary
    - what is in the display, what is on the tape
  + manipulates the model
    - registers operations to affect the value
  + responds to model changes
    - causes registered operations to appear on the tape
  + you must register the controller for all the view's buttons

* CalculatorView
  + create all the GUI elements, such as the tape and the JButtons
  + create a method:

public void addActionListener(ActionListener l){

for(Component c: bbutton\_panel.getComponents()){

JButton b = (JButton) c

b.addActionListener(l);

}

}

* CalculatorController implements ActionListener
  + encapsulates a model and a view
  + in the constructor, call view.addActionListener(this)

* what should happen when a digit is pressed?
  + depends on what has happened before
    - if starting a new number, we should replace the display with this digit
    - if we have already started a umber, we should append the digit to the number already in the display
  + keeping track of this kind of logic and doing the right thing is exactly the job of the controller
    - Thus, the contorller needs to be able to keep track of whether we're starting a new number or not
      * and a way to get and set the display
      * Ex:

if(startOfNumber){

if(button\_char != 0){

view.setDisplay(Character.toString(stuff))

}

else{

view.appendToDisplay(Character.toString(stuff))

}

* setDisplay(String s) and appendToDisplay(String s) are then defined in the view, which manipulate the correct GUI element
* Handling decimal point
  + should only allow one decimal point
  + special case if decimal point is first
    - introduce boolean flag: point\_pressed
      * initialize as false when constructed

if(start\_ofNumber){

view.setDisplay("0.");

start\_of\_number = false;

}else if (!point\_pressed){

view.appendToDisplay(".");

}

point\_pressed = true;

* Handling operations
  + what happens when an operator is pressed?
    - the operator indicates which operation is now in progress
      * but we can't actually do the operation until another number is entered
      * if an operation was already in progress, the operation should now complete using number in the display
    - add a field in controller to keep track of operation in progress
    - logic to apply operator in progress at model at the right time
      * right time = when operand is complete and next operator is known
      * to get operand, we need to add method in view object to retrieve current display
* the controller must observe the model
  + when the model calls notifyObservers(op), it should call the controller's update method
    - this update method:

public void update(Observable o, Object arg){

Operation op = (Operation) arg;

view.appendToTape(op.toString() + "\n");

view.setDisplay(Double.toString(model.getValue());

start\_of\_number = true;

point\_pressed = false;

}

* Four bugs and an inversion
  + delay in seeing result when pressing =
  + right now, the controller and view requires the controller to know about view internals
    - instead of exposing raw underlying UI events, we should translate them into something more abstraction-specific
    - create OperationEvent class
      * whenever a button is pressed, fire the events to listeners