Implementation issues
The MVC architecture and implementation

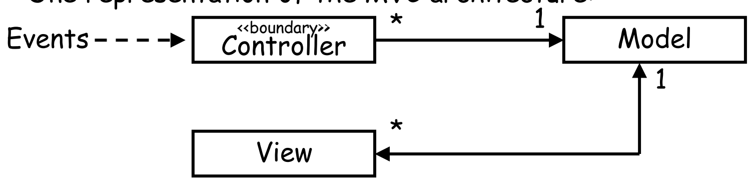
- When organizing a design, identifying roles for components and establishing communications patterns between them:
  - Rather than simply having a "sea" of classes, with *ad hoc* communication patterns
  - Better to adopt a well understood overall organizational pattern as framework around which to build the class diagram
- One very useful framework is MVC – Model View Controller
  - Useful for *highly interactive* applications
  - Based on the Publisher-Subscriber design pattern:
    Model: Publisher View: Subscriber

MVC in detail

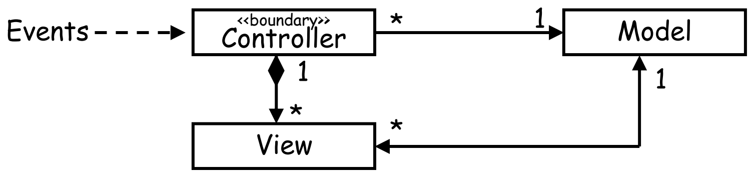
- We consider the following:
  - We have a single *model* that represents (the information content of) our system of interest
  - We have one (or more) *controllers* that are *boundary classes*, that receive *inputs* and send messages to the model
  - Each controller has one (or more) associated *views*, which are *notified* by the model whenever its content changes

MVC: Class diagram

- One representation of the MVC architecture:



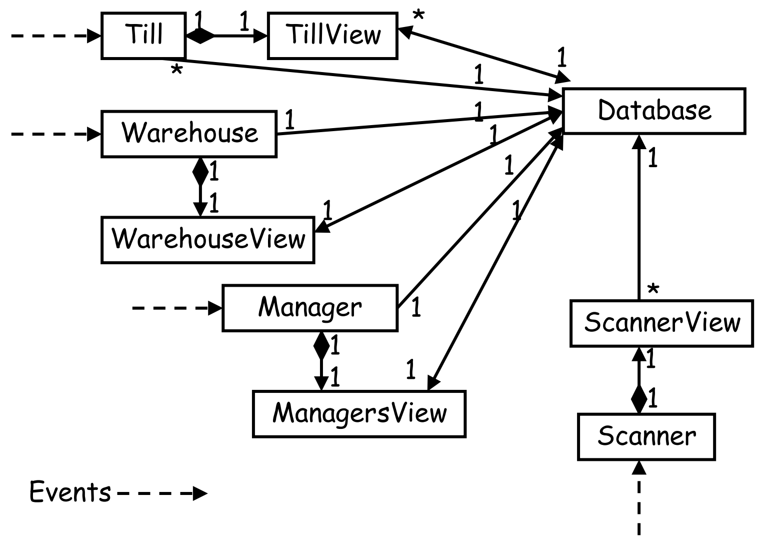
- In practice:
  - A controller may be used to control a view
  - It may be convenient to aggregate the views into controllers (and possibly not even have two classes)



MVC example

- A large modern supermarket:
  - Model: stock database, price database, customer database
  - Controllers: Service tills, warehouse office, manager's office, in-store barcode scanners
  - Each controller has its associated views
  - Candidate classes: Database (ignore the subdivision), Till, TillView, Warehouse, WarehouseView, Manager, ManagersView, Scanner, ScannerView

- Adopting MVC, the classes could be related like this:



Java library classes: Observer and Observable

- `Observer` is the Subscriber role: it is an *interface*:
  - Its only operation is:

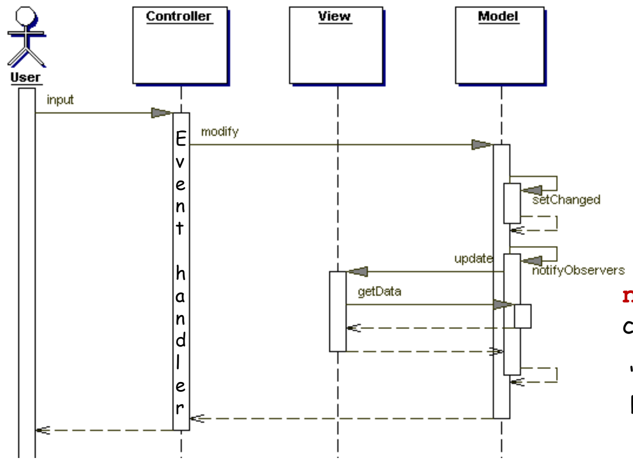
    `public void update(Observable o, Object arg)`
- Each View `implements Observer`:
  - The Model sends a message to `update` to signal it has changed
  - The View's `update` implements an appropriate reaction
- `Observable` is the Publisher role: it is a *class*:
  - Principal operations are:

```
public void addObserver(…)
protected void setChanged()
public void notifyObservers()
public void notifyObervers(Object arg)
```

- Models `extend Observable` (or could include it by aggregation)
  - They are watched/listened to/observed by views

Designing the MVC architecture

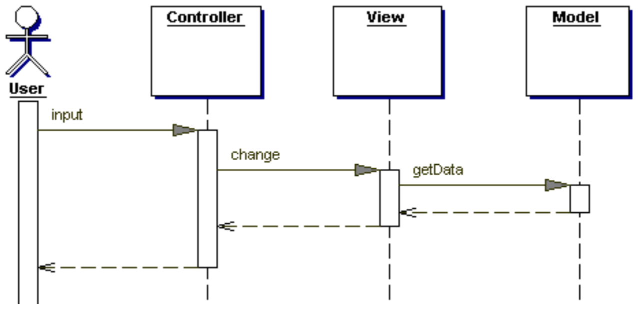
- First review its operation through sequence diagrams:
- Reacting to an input event that causes a change in the model:



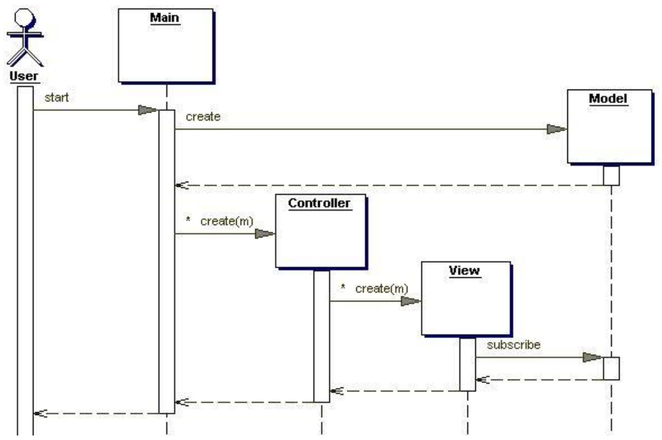
notifyObservers checks change status

'update' may be repeated

- Reacting to an input that only causes a change in the view:



- Initial set up: one model, more than one controller, each with more than one view:



- Following set up, all further activity arises from input/events received by controllers

Coding the MVC architecture

- Implementation decisions:
  - Treat each controller (and its views) as appearing in one window (JFrame) – all on a single screen
    (Later could generalize to controllers inhabiting remote hardware)
  - Each view within a controller will be a JPanel
  - So:

    `Controller extends JFrame`
    Constructor parameter: a `Model`

    `View extends JPanel`
    Constructor parameters: a `Controller` and a `Model`
    - the `Model` is essential, and the `Controller` may be useful

- The main program:

```
public class Main {

  public static void main(String[] args) {
    Model m = new Model();
    Controller1 c1 = new Controller1(m);
    ...
    ... (repeated for other controllers)
    ...
  }
}
```

- A typical controller:

```
public class Controller extends JFrame {
  private Model m;
  private View1 v1; // Maybe several
  public Controller (Model m) {
    this.m = m;
    ... set up input GUI ...
    v1 = new View1(this, m);
    window.add(v1);
    ... (repeat for other views) ...
    setSize(..., ...);
    setLocation(..., ...);
    setVisible(true);
  }
  ... event handler and other methods ...
}
```

- A typical view:

```
public class View extends JPanel
                 implements Observer {
  private Model m;
  private Controller c;
  public View (Controller c, Model m) {
    this.c = c;
    this.m = m;
    ... set up display GUI ...
    m.addObserver(this);
  }
  public void update(Observable o,
                     Object arg) {
    m.getData();
    ... update display ...
  }
}
```

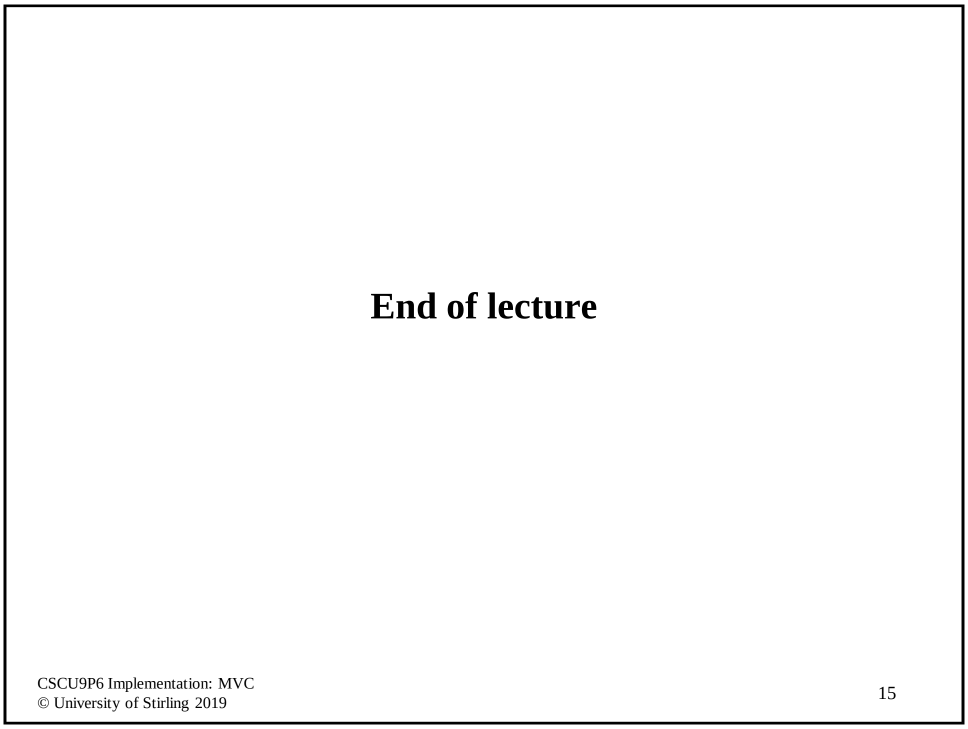
- A typical model:

```
public class Model extends Observable {

  ... attributes ...

  public void modify(...) {
    ...
    setChanged();
    notifyObservers();
  }

  public ... getData(...) {
    ...
    return ...;
  }
}
```



End of lecture

CSCU9P6 Implementation: MVC
© University of Stirling 2019

15