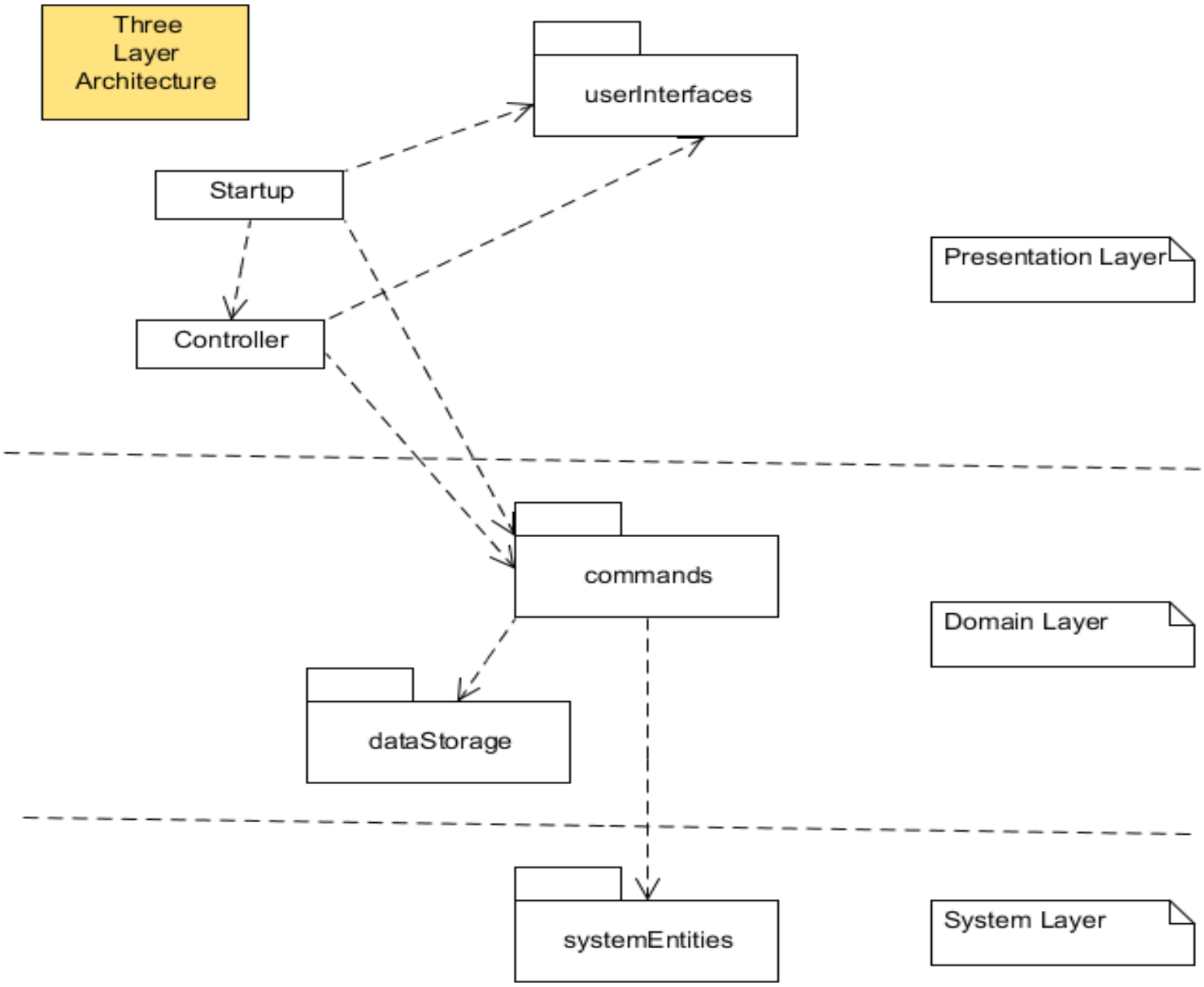
LECTURE 17

MODEL-VIEW-CONTROLLER

Recall the layered architecture used in the bank application / hospital system



Model-View-Controller

- Model-view-controller is a very popular software design pattern
- Created in the 1970's
- Common for GUIs and web apps

Recall: Controller tasks

Non-GUI controller:

- Obtain the command request from the interface
- Obtain the parameter values from the interface
- Invoke the command
- Obtain the results from the command
- Display the command results to the user

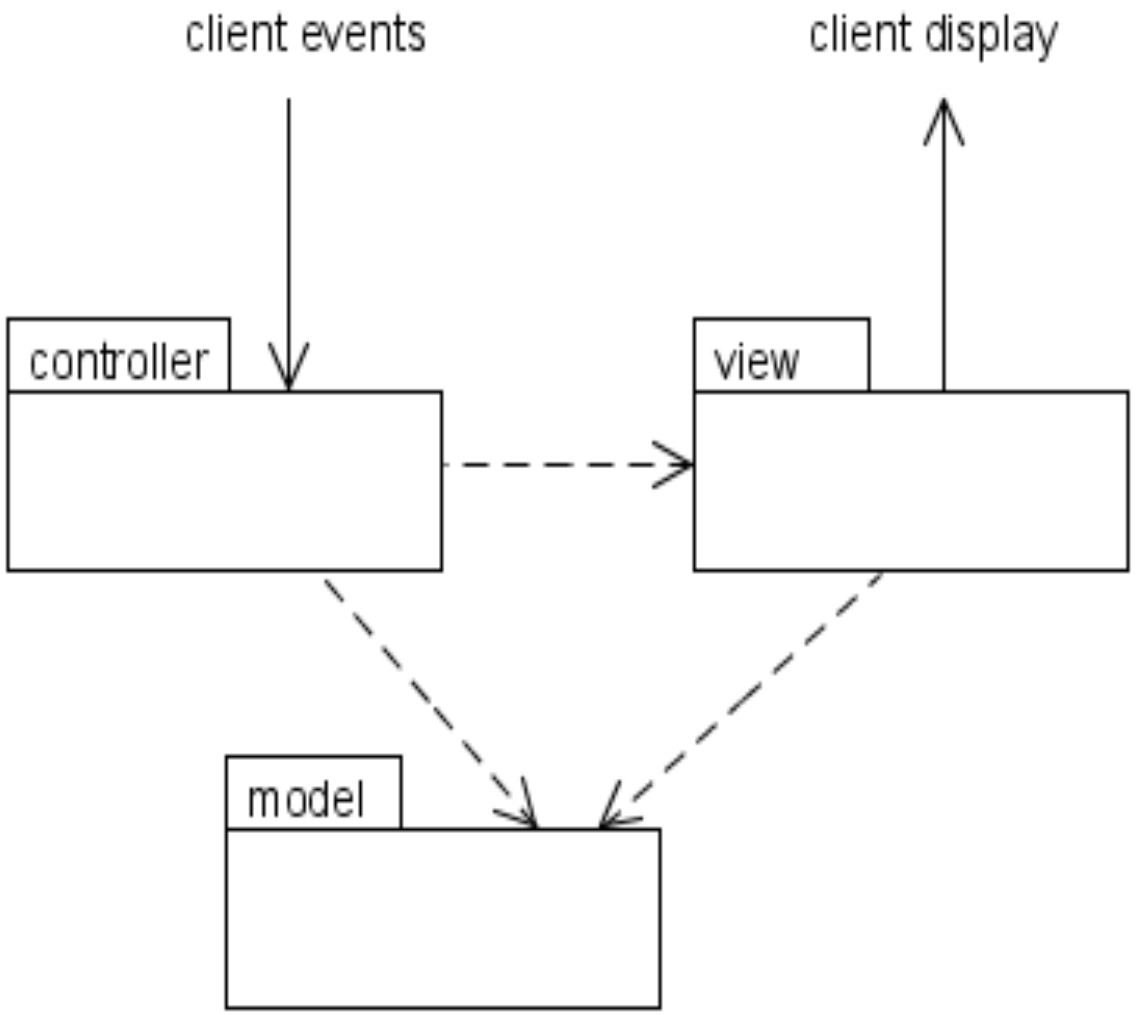
GUI-based controller:

- Build a windows and listener to receive the event
- Accessed from the window by the listener
- Invoked from the listener
- Done by the listener
- Update or new window

GUI version: where is the controller?

- In a GUI application:
  - many of the controller tasks are done by listeners
  - often have a different listener for each command
  - or the switch statement is in the listener
- Therefore, so far, all our Controller code is intermixed with the Interface code (within the Presentation layer).
- But we can separate the Controller from the Interface by an OO-design scheme called the Model-View-Controller architecture

Basic Model-View-Controller



Model

View

Controller

Simple Example

- See CalcMVC
- The model is the data (entities) part.
- For the Calculator, it's just the current answer
- The view is the GUI for the Calculator (inherited from JFrame)
- The controller is separated from both the model and view

Simple Example

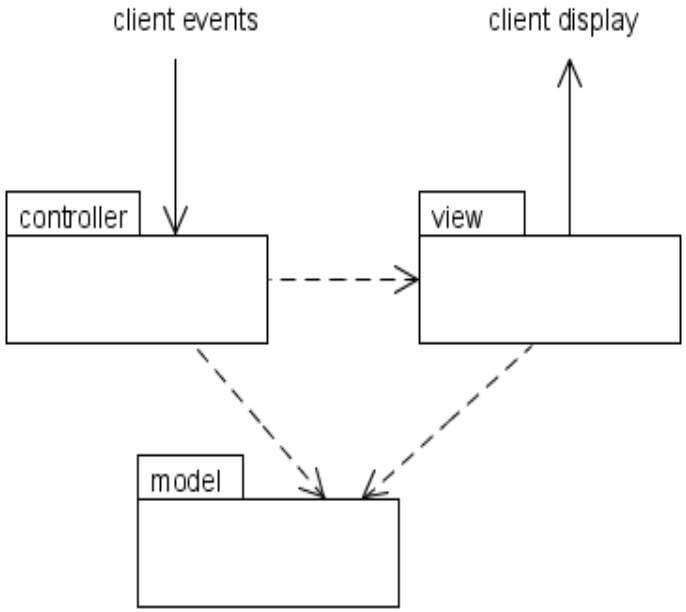
- When the user interacts with the UI, the controller takes charge
- It updates the model, and can update the display

Advanced Implementations

- This is one simple implementation of model-view-controller
- There are several other types of communication possible within the paradigm
- In another type of communication, if the model changes, it “informs” the view directly to change

MVC: Updating the View

- Need to inform the view when a change is made in the model.
- But, do not want the model invoking arbitrary methods of the view.
- Ideally, the model shouldn't even know about the view.
- Should a class for Books and Book Collections call on GUI methods to change the interface?

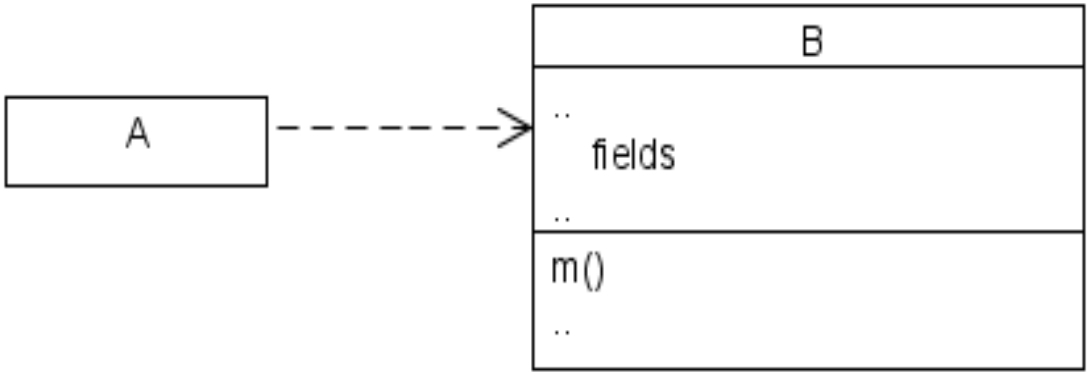


How to minimize coupling?

- The controller (in the control package) should not know everything about the model or the view either.
- The view should not know everything about the model.
- The model needs to notify the view (and sometimes the controller) that the model has changed, but it should not know who it is notifying.

Minimize coupling

- Usually, if an object **a** of type `A` can invoke a method in another object **b** of type `B`, **a** can invoke any visible method of **b**. This can make it difficult to modify class `B` without needing to modify class `A`.
- Also, type `A` needs to know of the existence of type `B`.



- If `A` only needs method **`m()`** of `B`, how can this strong coupling be reduced?

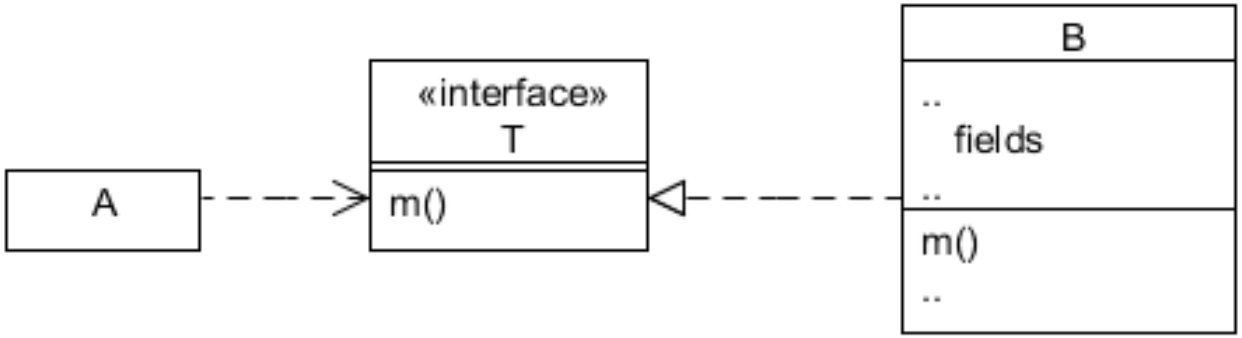
How is this done with Listeners?

```
class ChoiceListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      setLabelFont();
   }
}
```

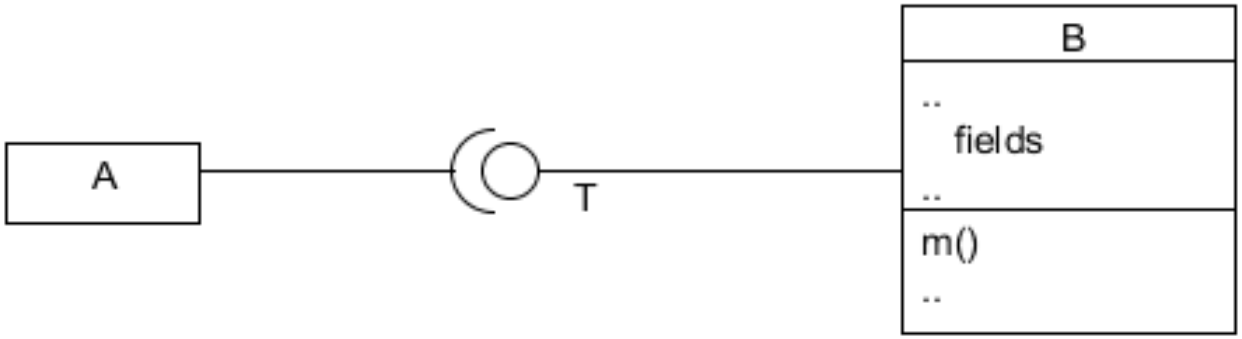
Minimize coupling: use interfaces

- If `A` only needs method `m()` of `B`, how can this strong coupling be reduced?

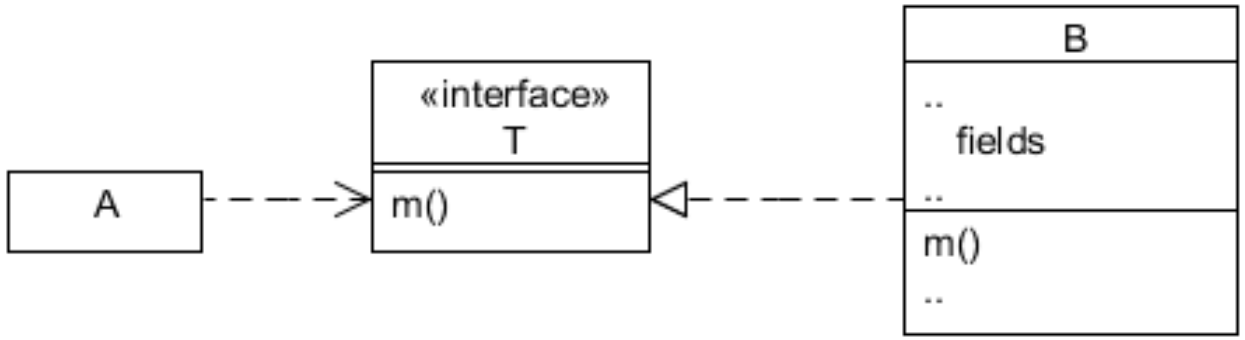
  Define an interface `T` with the features of `B` needed by `A`.



Alternate notation:



Minimize coupling: use interfaces



- An object **a** of type A can be passed an object **o** that has a type that is an implementation of T.
- Object **a** can interact with **o** without knowing **o**'s precise type, or any of its methods (or fields) other than the methods of T.

- A only needs to know of the existence of interface T, and its methods.
- B only needs to provide an implementation of T. Their coupling is minimal.

Example: JButton

- The class JButton has the method

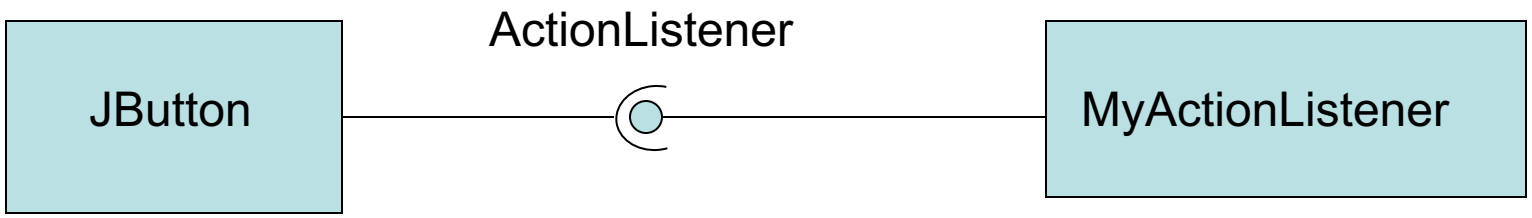
  public void addActionListener(ActionListener l)

  where ActionListener is an interface.

- The class JButton never knows anything about the object (even its class name, if it has one) that is passed in as a listener.
- Class JButton is part of the Java API so it certainly does not know anything about a listener that you write, except that it implements the ActionListener interface.

Example: JButton

- The relationship is that some class, JButton in this example, interacts with an object of an interface type, while some other class provides the implementation of the interface.
- This interaction is common enough that there is special notation for it in a UML diagram.



This is called the ball-and-socket notation. The ball end is called a lollipop.

Model Notifies the View

- In Java, there is an interface called `Observer`
- There is only one method in it:
- `void update(Observable o, Object arg)`
- Any class (such as a view) which implements the `Observer` interface, can be "notified" whenever a model changes

View

- Put inside the `update()` method, any changes to the interface to be made based on model changing
- How does a model tell the views that its data has changed?

Observable

- Make the model inherit from the `Observable` class in java
- This allows the model to have an Observer added to it
  - `addObserver(Oberserver o)` is a method inherited from Observable class
  - model.addObserver(view) can be placed somewhere (usually right after model and view are created) to let the view be notified when the model wants

In the model

- In the model, after making a change to the data, run
  - `setChanged()` to indicate data has been changed
  - Then `notifyObservers()` to notify all observers that the model has changed
  - This causes their update() method to run

```
public void setValue(int x){
    value = x;
    setChanged();
    notifyObservers();
```

The view

- Once the view has been notified that a change has been made, it can either:
  - request for the data from the model (pull) via accessor methods,
  - or can be sent some data directly (push).
- If `pull' is used, the view needs access to the model (stores the model as an instance variable)

Push

- To get “pushed” certain data when the model changes, instead of using
  - `notifyObservers()`, use
  - `notifyObservers(Object arg)`
- This sends the Object arg to all Observers
- If “push” is used, the view does not necessarily need direct access to the model

Observers

- There can be more than one observer for a model
- They all get notified when the model changes and issues a notification
- Notice that the model knows nothing about the observers
- This promotes reuse and reduces coupling

Controller

- The controller is still doing a lot of the work
- If the user is interacting, it is the controller that is responding
- If a button is clicked, the controller intercepts and tells the model to change
- This causes observers, such as the view, to be notified
- Then view either receives data from model, or asks for the data from the model

Objects

- Notice how the main method created a new Model object, a new View object, and a new controller object, and linked them
- Since they are just Objects, and not classes, we can create multiple, and they are linked independently

Custom Implementations

- To send and receive notifications, you do not need model to extend `Observable` and have the view implement `Observer`
- It's possible to make your own interface like Observer with a custom `update` method
- Make your own class like Observable that keeps a list of Observers and allows adding Observers
- A custom update method (or methods!) can push anything