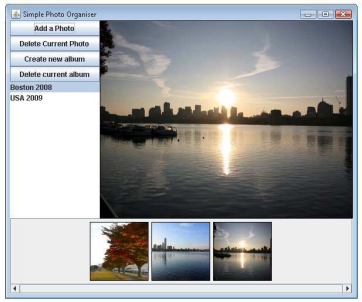
Programming Methods
Dr Robert Harle

IA NST CS and CST
Lent 2008/09
Handout 3

Our Motivating Example

- A simple photo organiser
  - Add, remove photos from collections
  - Thumbnail selection



Where do we start?

- The basic concept embodied by a photo organiser is the organisation of photos i.e. Mapping photos to groups.
- First leap: the graphics stuff (windows, buttons) is independent of this underlying concept
  - It's just a convenient way to provide input/output
  - We might want to change that way, or even have multiple simultaneous ways (we'll come back to that)
- So let's separate out the concept from the interface

The "Model"
Embodies the core data structures and algorithms for a photo organiser

The "View"
The graphical aspects. i.e. The presentation of the model's current data

Where do we start?

- This is sensible because:
  - The code is easier to navigate around
  - We can have software reuse so long as we loosely couple the model and the view(s) [see later]
  - E.g. We can reuse the model with multiple views:

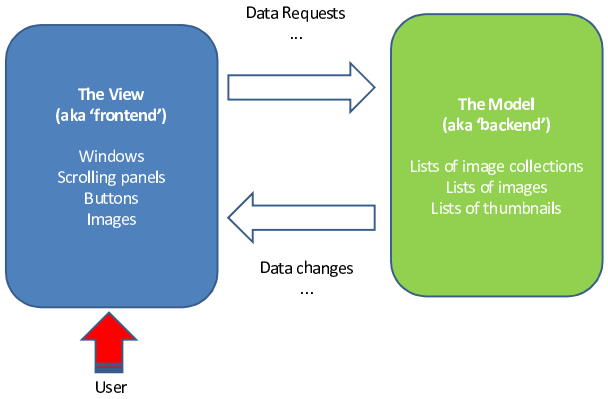
The "Model"
Embodies the core data structures and algorithms for a photo organiser

View 1
Graphical user interface

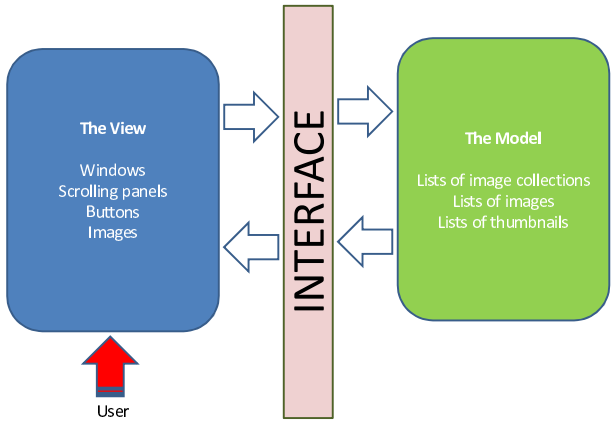
View 2
Console user interface

View 3
Phone interface

General Model



General Model



Model-View-Controller (MVC)

An architectural/design pattern that goes one step further

Model

- A domain-specific representation of the underlying information that the application is presenting (i.e. the backend)
- E.g. The picture data and metadata, a database in a web application, or the data in a spreadsheet

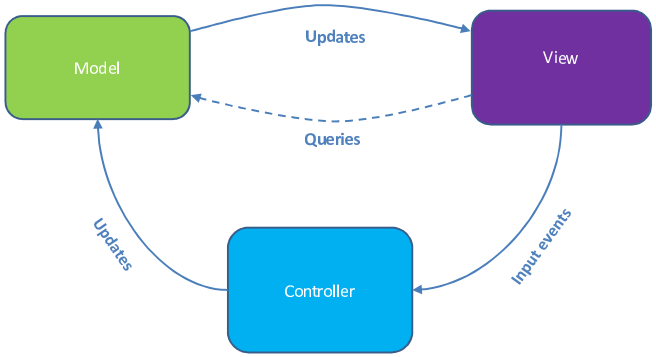
View

- Renders the model into something visible
- E.g. The GUI of our picture elements, a page of a web app, or the cells and graphs of a spreadsheet
- Note that multiple simultaneous views of the model can exist – E.g. Multiple spreadsheet graphs of the same data

Controller

- Handles all input from the user, potentially modifying the model and the view(s)

Model –View-Controller (MVC)

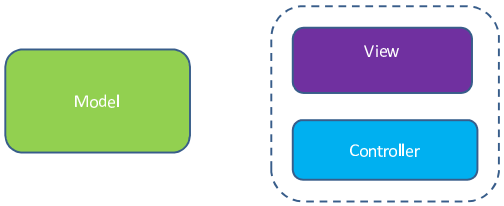


Aside: GUI Toolkits

- A **GUI toolkit** is just a set of tools (classes, algorithms, 'glue') that makes it easy to draw graphical things like buttons and handle mouse clicks etc.
- Most languages *don't* have a toolkit
  - Instead you have a choice and you can download from lots (or make your own – usually a bad idea!)
- Java has **AWT** and **Swing** as part of the language!
  - We will use Swing

Model-View Controller

- The MVC design is used in *almost every* GUI toolkit
  - Including Java
- In most implementations, we find that it is <u>not</u> useful to decouple the controller and the view
  - If the controller handles a "new photo button event", it's coupled automatically since it assumes there IS a "new photo button"!
  - In fact, you will often find that the controller and the view are combined in the same file.



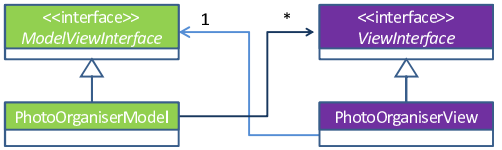
The Photo Organiser

- Back to our example
- We want our model and view to link together

- These are horribly coupled
  - One won't compile without the other
  - But we want to swap out different views L

The Photo Organiser

- We get the required <u>decoupling</u> by specifying interfaces:



- Now the Model can be used with anything that implements the ***ViewInterface*** and vice-versa. It neither knows nor cares which View it's talking to.
- How can we have multiple Views at once?
  - **Observer** pattern!

ModelViewInterface

- This is everything the photo organiser model must support for the view to do its job
  - register(ViewInterface vi) – register a new View (observer pattern)
  - deregister(ViewInterface vi) – deregister a View (observer pattern)
  - Plus queries such as getPhoto() etc.

```
public interface ModelViewInterface {
   public void register(ViewInterface vi);
   public void deregister(ViewInterface vi);

   public Set<Photo> getAlbumPhotos(String album);
   public Set<String> getAlbums();
}
```

Observer

Data Access

ViewInterface

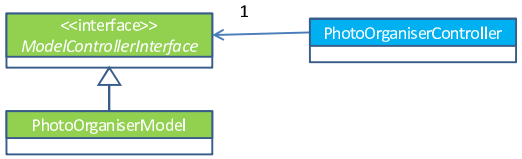
- Everything that a View needs to support for a Model
  - This is just what the observer pattern needs
  - i.e. Some way for the model to tell the view that it should change
  - We add an update() method

```
public interface ViewInterface {
   public void update();
}
```

Observer

The Controller

- We accept the View and Controller need to be coupled
- The controller needs to be able to tell the Model that something has changed
- BUT there is no need to use the **observer** pattern because the controller doesn't care if the model changes (only the View does)
- So we just specify an interface to the model



ModelControllerInterface

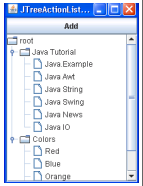
- The Controller just needs to be able to manipulate the Model:

```
public interface ModelControllerInterface {
   public void createAlbum(String name);
   public void deleteAlbum(String a);
   public void addPhoto(String path, String album);
   public void deletePhoto(Photo p, String album);
}
```

The View

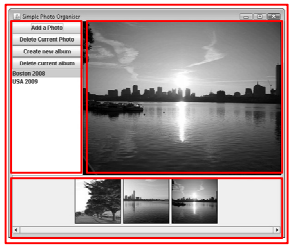
The View

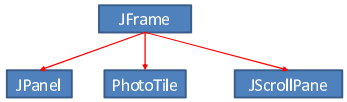
- We use Java Swing for the GUI
  - There are plenty of graphical toolkits out there, but Swing does what we need and is standard
- The toolkit is a set of graphical components (buttons, windows, etc)
  - Each component follows the MVC model!
  - E.g a JTree has an explicit TreeModel where it stores data
  - The Controller and the View get lumped together again
- The components are put together using the **Composite** pattern and communicate via the **Observer** pattern (using **events**)
- Let's look in more detail



How Swing Works

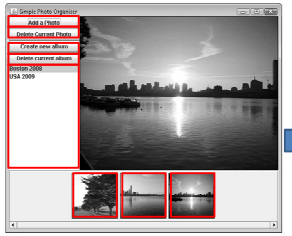
- Every GUI component extends ("is-a") JComponent
- Each component can potentially contain other components

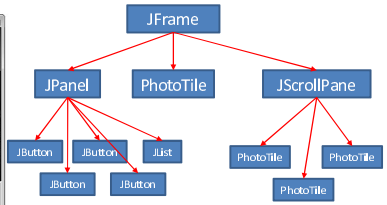




How Swing Works

- Every GUI component extends ("is-a") JComponent
- Each component can potentially contain other components





- End up with a tree of JComponents

How Swing Works

- Any node has a set of child nodes that obey the JComponent interface
  - This is the **composite** model!
- leftpanel.add(mAlbumList, BorderLayout.CENTER);
  - Code like this adds a child JComponent to a parent, and optionally tells it where to display it
  - Once the tree is set up, Java knows how to draw it to the screen

PhotoTile: A Custom Component

- There isn't a handy component that displays images
- So we must make our own: PhotoTile.java
- The closest thing to what we want is a simple JPanel
- **Inheritance** saves us rewriting the JPanel stuff

```
public class PhotoTile extends JPanel {
```

- And we override the paintComponent() method

```
@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (mPhoto!=null) mPhoto.drawImage(g, 0, 0, this.getWidth(), this.getHeight())
    g.setColor(Color.black);
    g.drawRect(0, 0, getWidth()-1, getHeight()-1);
 }
```

The Controller

Events

- Events:
  - The composite pattern is all very nice for display, but what about interaction? How does the GUI *do* stuff?
  - Components generate 'events' to indicate something is happening to them (e.g. Button being pressed)
  - They send these events off to anyone who has registered an interest in receiving them
    - Receivers must implement a predetermined interface so that we know how to talk to them to tell them that an event occurred
    - Ah - this is the **Observer** pattern yet again

The Controller

- So the view registers all the components that have to do something (buttons etc) with a handler – our Controller
  - PhotoOrganiserController.java
- First, we need our controller to implement the interfaces that JButtons and components use to tell us events have happened

```
public class PhotoOrganiserController implements ActionListener, ListSelectionListener, MouseListener {
```

Buttons use this interface to tell us about presses

The JList uses this interface to tell us when a new selection has been made

The PhotoTile uses this interface to tell us when it's been clicked

actionPerformed(ActionEvent e)

- This is the 'callback' function for a button
  - First we register the controller with a button – the observer pattern's register() method is addActionListener() in Swing
  - See PhotoOrganiserView.java

```
mAddPicButton = new JButton("Add a Photo");
...
mAddPicButton.addActionListener(mController);
...
mAddPicButton.setActionCommand("ADDPHOTO");
```

Create

Register

Add a string that makes it easy to identify which button has been pressed

actionPerformed(ActionEvent e)

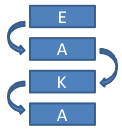
- Now we write the event handler in PhotoOrganiserController.java

```
public void actionPerformed(ActionEvent e) {

  String cmd = e.getActionCommand();

  ModelControllerInterface model = mView.getModel();

  if (cmd.equals("ADDPHOTO")) {
    // Use a standard dialog to select the file
    final JFileChooser fc = new JFileChooser();
    int returnval = fc.showDialog(mView,"Select");
    if (returnval==JFileChooser.APPROVE_OPTION) {
      model.addPhoto(
          fc.getSelectedFile().getAbsolutePath(),
          mView.getCurrentAlbum()
      );
    }
  }

// Deleting a photo
else if (cmd.equals("DELPHOTO")) {
...
```

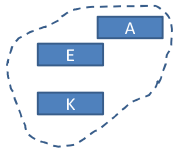
The Model

The Model Data Structure 1

- The **Observer** part needs us to keep track of every registered View
- Java offers us **List**s and **Set**s



- Sequence of elements
- Order important
- Duplicates allowed

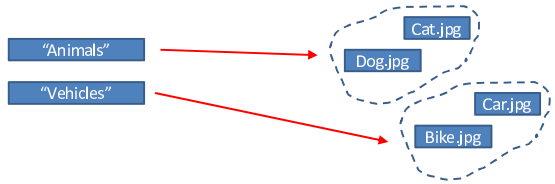


- Like *sets* in Discrete Maths I
- Order usually unimportant
- No duplicates allowed

- We update each View once -> Use a **HashSet**

The Model Data Structure 2

- We need to map albums to their photos
- Java offers us **Map**s that link keys to values



- Like *relations* in Discrete Maths I
- We use a **TreeMap< String, Set<Photo> >**

The Tree bit guarantees the keys will be stored in order

The keys are Strings

The values are Sets of Photo objects

The Model

- If you look in PhotoOrganiserModel you will see:
  - We keep track of the registered views using variable **mViews**
  - We keep track of the Album-Photo mapping in **mPhotos**
  - Then the model implements all the functions required from **ModelViewInterface** and **ModelControllerInterface**
    - These functions are really just manipulating **mViews** and **mPhotos** in sensible ways

Example

- To (de)register Views we simply (remove) add to our HashSet

```
public void
register(ViewInterface vi) {
    mViews.add(vi);
}
```

```
public void
deregister(ViewInterface vi) {
    mViews.remove(vi);
}
```

- Then to tell the Views an update has occurred we cycle over all of them and update() them in turn

```
private void alertViews() {
    for (ViewInterface vi : mViews)
        vi.update();
}
```

- *[Why is this method private?]*

Done!

- We do something similar for the JList (ListSelectionListener interface) and the PhotoTile (MouseListener interface)
  - All the code is in PhotoOrganiserController.java
- Now we just write a start point for the program
  - See PhotoOrganiser.java
- The result is a working (but rather simple) photo organiser!

- Beware! I've deliberately tried to keep the code short and simple
  - I didn't put in any error checking
  - I didn't use any unit testing etc
  - The performance of this program is hardly stellar

Some Fun...

- Just to emphasise the flexibility of our design
  - We can throw together a different view
  - See ThumbnailWindow.java
  - Just register that with the model and away we go!

- We can run multiple views simultaneously
  - They update automatically!!