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.

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”

- Embeds 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.

View 1

 Graphical user interface

View 2

 Command line interface

View 3

 Photo interface
Model-View-Controller (MVC)

**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 usable
- E.g., the GUI of our MVC elements, a page of a web app, 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).
Aside: GUI Toolkits

$ A \textbf{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 \textit{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 \textit{not} useful to decouple the controller and the view
  $ If the controller handles a "new photo button event", it's coupled automatically & not it assumes there's 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

The Photo Organiser

$ We get the required \textbf{decoupling} 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?
  $ \textbf{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.

```java
public interface ModelViewInterface {
    public void register(ViewInterface vi);
    public void deregister(ViewInterface vi);
    public Set<Photo> getAlbumPhotos(String album);
    public Set<String> getAlbums();
}
```

### 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

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

### 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

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

### ModelControllerInterface
How Swing Works

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

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. JTree has an event 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

- 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

\[
\text{public class PhotoTile extends JPanel} \{
\]
\[
\text{@Override}
\]
\[
\text{public void paintComponent(Graphics g) \{}
\]
\[
\text{super.paintComponent(g);}
\]
\[
\text{if (mPhoto != null) mPhoto.drawImage(g, 0, 0, this.getWidth(), this.getHeight());}
\]
\[
\text{g.setColor(Color.black);}
\]
\[
\text{gdrewed(0, 0, getWidth()+1, getHeight()+1);}
\]
\[
\text{\}}
\]

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
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 List 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

    public void 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
        // So see PhotoOrganiserView.java

        JButton mAddButton = new JButton("Add a Photo");
        mAddButton.addActionListener(mController);

        mAddButton.addActionListener(mController);

        mAddButton.addActionListener(mController);

        mAddButton.addActionListener(mController);

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

        The Model

        // get the selected item
        if (jTable.getSelectedRow() > -1) {
            String action = jTable.getValueAt(jTable.getSelectedRow(), 0).toString();
            if (action.equals("CREATE")) {
                mModel.add(new TableItem(mPhotoModel));
            } else if (action.equals("DELETE")) {
                mModel.remove(new TableItem(mPhotoModel));
            } else if (action.equals("EDIT")) {
                mModel.setNewValue(new TableItem(mPhotoModel));
            }
        }

        // get list of ऑ र ज
        if (jList.getSelectedIndex() > -1) {
            String action = jTable.getValueAt(jList.getSelectedIndex(), 0).toString();
            if (action.equals("CREATE")) {
                mModel.add(new TableItem(mPhotoModel));
            } else if (action.equals("DELETE")) {
                mModel.remove(new TableItem(mPhotoModel));
            } else if (action.equals("EDIT")) {
                mModel.setNewValue(new TableItem(mPhotoModel));
            }
        }
    }
}
The Model Data Structure 1

$ The Observer part needs us to keep track of every registered View
$ Java offers us Lists and Sets

* Sequence of elements
  * Order important
  * 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 Maps that link keys to values

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

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 v){
        mViews.add(v);
    }

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

$ 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 v : mViews)
            v.update();
    }

$ Why is this method private?
Done!

$ We do something similar for the JList (ListSelectionListener interface) and the PhotoFile (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!