Theme 2
Program Design

MVC and MVP

References

- Next to the books used for this course, this part is based on the following references:
  - *Interactive Application Architecture Patterns*, http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/
  - *Android Architecture: Part 10, The Activity Revisited* http://www.therealjoshua.com/
  - *Catalog of Patterns of Enterprise Application Architecture (Martin Fowler)* http://martinfowler.com/eaaCatalog/
  - *GUI Architectures (Martin Fowler)* http://martinfowler.com/eaaDev/uiArchs.html

MVC

Some history

- remember that this was one of the first attempts to do serious UI work on any kind of scale
- Smalltalk 80 MVC
- Different people reading about MVC in different places take different ideas from it and describe these as 'MVC'

Two central ideas

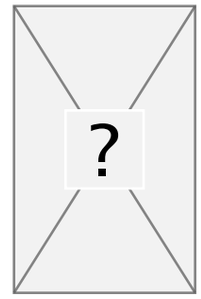
- **Separated Presentation:**
  - to make a clear division between domain objects that model our perception of the real world,
  - and presentation objects that are the GUI elements we see on the screen.
  - Domain objects should be
    - completely self contained and
    - work without reference to the presentation,
    - able to support multiple presentations, possibly simultaneously.

Two central ideas

- **Observer Synchronization**
  - have views (and controllers) observe the model to allow multiple widgets to update without need to communicate directly
  - all the views and controllers observe the model. When the model changes, the views react.
  - the controller is very ignorant of what other widgets need to change when the user manipulates a particular widget.

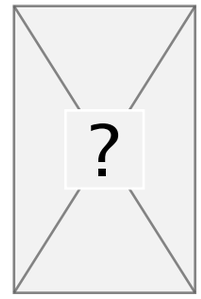
rephrasing....

Problem Statement



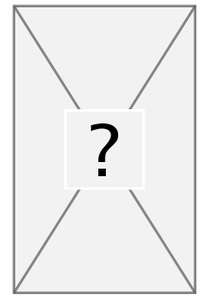
- **Challenge:**
  - writing programs with a graphical user interface
  - multiple open windows showing the same data
    - keeping them consistent
  - manipulating data in many different ways by direct manipulation (e.g. move, resize, delete, create, ...)
    - process input events from multiple windows

Solution: MVC



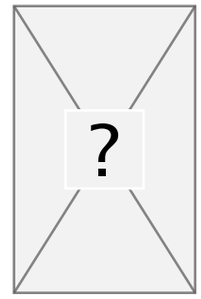
- The **Model-View-Controller** is an architectural pattern because:
  - it defines a solution to the problem of structuring the 'large-scale' / architectural challenge of building graphical user interface applications.
- But the 'engine behind the scene' is a careful combination of **strategy** and **observer**...

First challenge

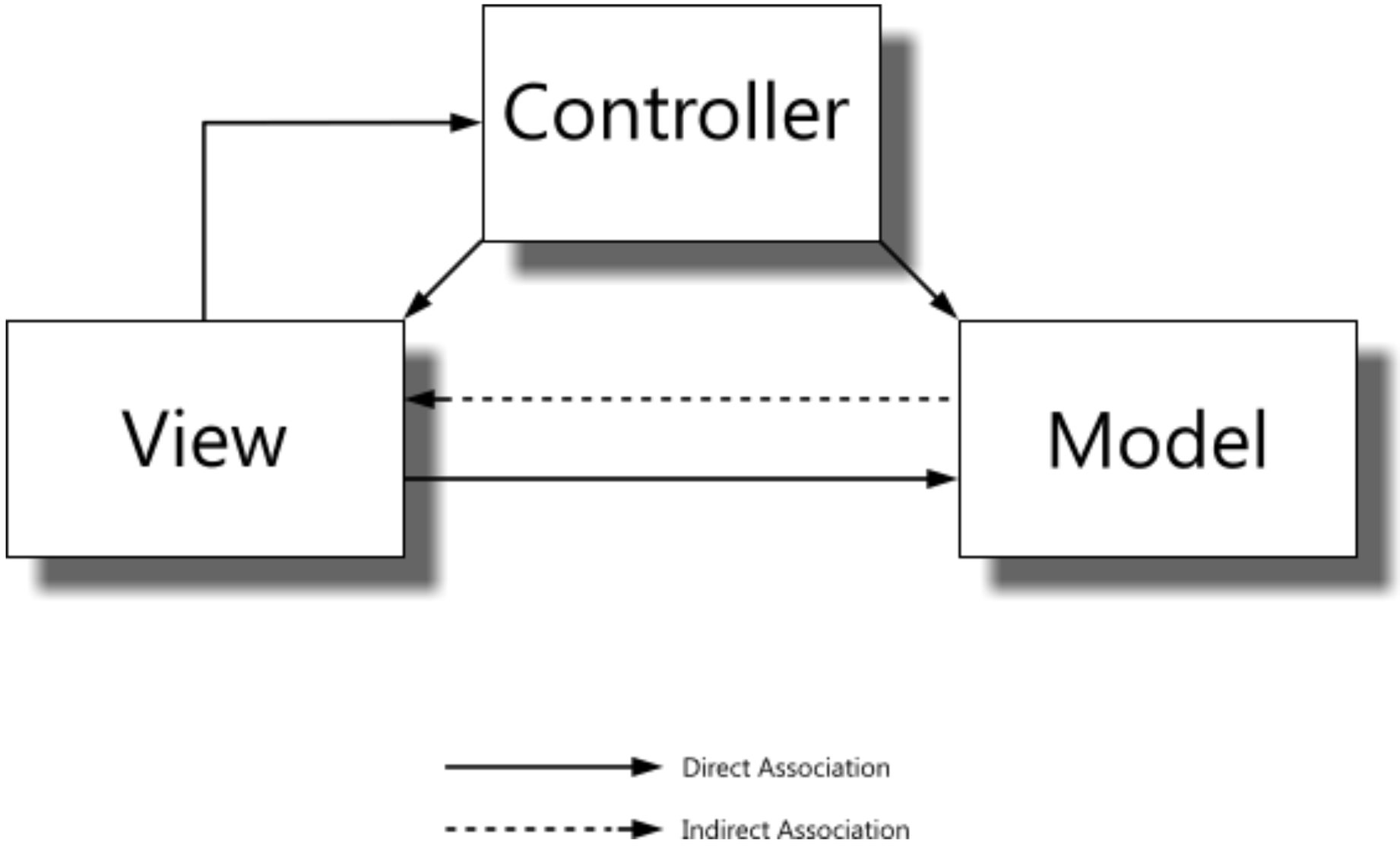


- need to keep the windows consistent:
  - solved by the **Observer** pattern
  - remember: underlying information must be stored in a **Subject** object that can notify its **Observers**.
  - In MVC : **Model** (containing state and notifying upon state changes) and **View** (rendering the graphics) respectively.

Second Challenge



- receiving and interpreting events from the user
  - solved by the **Strategy** pattern
  - change the way a view responds to user input without changing its visual representation.
  - response mechanism is encapsulated in a **Controller** object.
  - a **View** uses an instance of a **Controller** subclass to implement a particular response strategy.



http://aspiringcraftsman.com/wp-content/uploads/2010/02/MVC1.png

http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/

The Model represents the state, structure, and behaviour of the data being viewed and manipulated by the user. The Model contains no direct link to the View or Controller, and may be modified by the View, Controller, or other objects with the system. When notification to the View and Controller are necessary, the Model uses the Observer Pattern to send a message notifying observing objects that its data has changed.

The View and Controller components work together to allow the user to view and interact with the Model. Each View is associated with a single Controller, and each Controller is associated with a single View. Both the View and Controller components maintain a direct link to the Model.

The View's responsibility can be seen as primarily dealing with output while the Controller's responsibility can be seen as primarily dealing with input. It is the shared responsibility of both the View and the Controller to interact with the Model. The Controller interacts with the Model as the result of responding to user input, while the View interacts with the Model as the result of updates to itself. Both may access and modify data within the Model as needed.

As data is entered by the user, the Controller intercepts the user's input and responds appropriately. Some user actions will result in interaction with the Model, such as changing data or invoking methods, while other user actions may result in visual changes to the View, such as the collapsing of menus, the highlighting of scrollbars, etc.

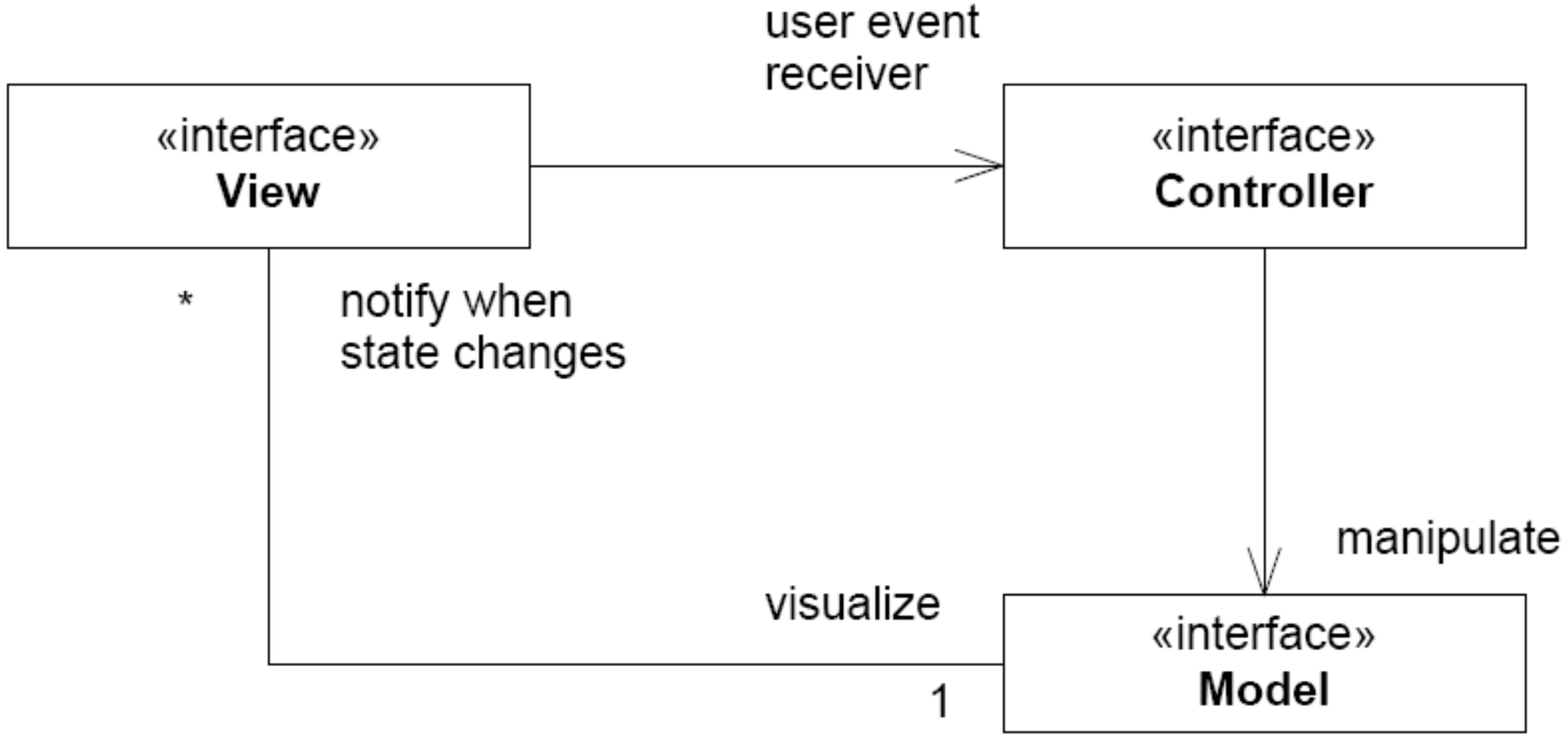
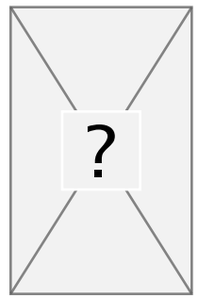


Figure 29.2: MVC role structure.



Model

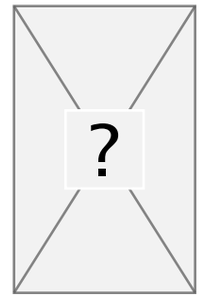
- Store application state.
- Maintain the set of Views associated.
- Notify all views in case of state changes.

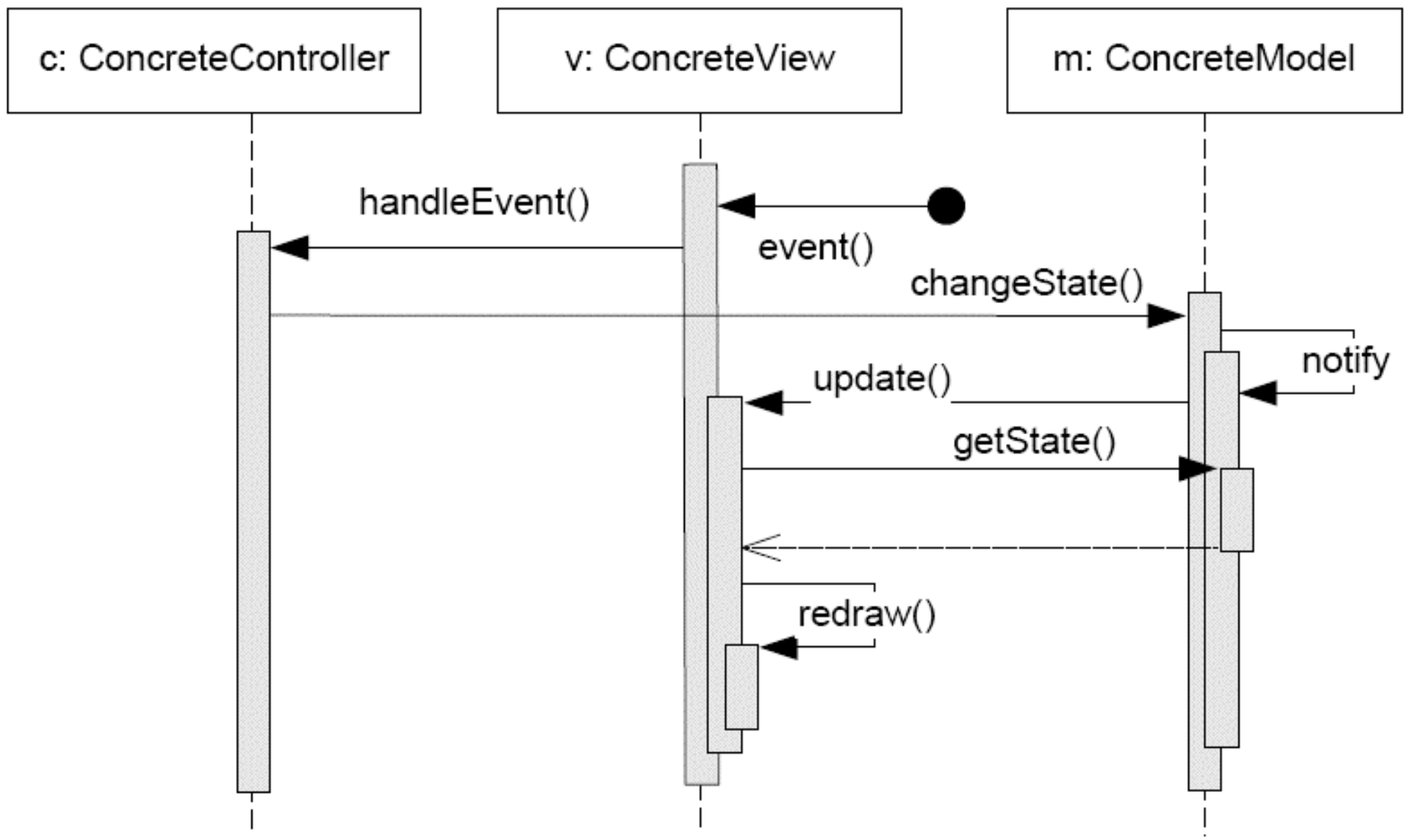
View

- Visualize model state graphically.
- ~~Accept user input evens, delegate them to the associated Controller.~~
- Potentially manage a set of controllers and allow the user to set which controller is active.

Controller

- Interpret user input events and translate them into state changes in the Model.


?


c: ConcreteController
v: ConcreteView
m: ConcreteModel
handleEvent()
event()
changeState()
notify
update()
getState()
redraw()

MVC misconception

- *"The purpose of the Controller is to separate the View from the Model."*
  WRONG!!!
  - The MVC pattern does decouple the application's domain layer from presentation concerns, this is achieved through the Observer Pattern, not through the Controller.
  - The Controller was conceived as a mediator between the end user and the application, not between the View and the Model.

Some complications

- Suppose an application with a text field "variance" and need to set the colour of the text depending on the value of "variance"
- who is responsible for colour?
  - domain object? NO
  - view? YES BUT mapping values to colours and altering the variance field is not part of a standard text field.
- Presentation state:
  - basic MVC assumption: all the state of a view can be derived from the state of the model

Solutions

- **Presentation Model**
  - make a new kind of model object, one that's oriented around the screen,
    - but is still independent of the widgets.
    - would be the *model for the screen*.
    - it's a model that is really designed for and thus part of the presentation layer.
- example: VisualWorks Application Model

MVP

Model-View-Presenter

- first appeared in IBM (90's), paper of Potel
- popularised and described by the developers of Dolphin Smalltalk
- Ideas:
  - treat the view as a structure of widgets
  - view does not contain any behaviour that describes how widgets react to user interaction
  - presenter decides how to react to an event, updates the model, the view is updated through the Observer Synchronization

Variations

- the degree to which the presenter controls the widgets in the view.
  - all view logic is left in the view and the presenter doesn't get involved in deciding how to render the model (approach implied by Potel).
  - Supervising Controller: where the view handles a good deal of the view logic that can be describe declaratively and the presenter then comes in to handle more complex cases.
  - Passive View: the presenter does all the manipulation of the widgets.

MVC versus MVP

- Commonalities
  - both contain a Model and View
  - both the controller and presenter are involved in updating the model
  - both use observer synchronization to update view when model has changed
- Differences
  - MVC controller: intercepting user input, updating the model is a byproduct
  - MVP presenter: update the model, view intercepts the user's input.

to Android...

MVC pattern in Android?

from: stackoverflow.com

88

Is it possible to implement the Model-View-Controller pattern in Java for Android? Or is it already implemented through Activities? Or is there a better way to implement the MVC pattern for Android?

java android mvc

share | improve this question

53

asked **May 27 '10 at 21:23**

Mohit Deshpande

5,091 ●9 ●61 ●137

99% accept rate

feedback

14 Answers

active oldest **votes**

59

It's already implemented.

- You define your user interface in various XML files by resolution/hardware etc.
- You define your resources in various XML files by locale etc.
- You extend clases like ListActivity, TabActivity and make use of the XML file by inflaters
- You can create as many classes as you wish for your model
- A lot of Utils have been already written for you. DatabaseUtils, Html,

share | improve this answer

answered **May 27 '10 at 22:09**

Pentium10

41.7k ●22 ●140 ●248

73 if you're trying to imply that a static xml file is the view and an activity is a presenter/controller then you're missing the part of MVC/MVP pattern actually decouples the view and presenter. You cannot instantiate an activity without talking to your layout/view. Really what you want to do is use composition and embed the activity/layout into a view class and the have all the application/presentation logic decoupled out into their respective classes. While what you describe is MVC... it's a very bad, strongly coupled MVC. Which is poor to work with. Especially if you wish to unit test. – JDPeckham Apr 14 '11 at 23:20

11 Android is terrible at MVC. Android's API philosophy is template/inheritance over composition; which makes it bad for testing too. That being said, there are ways to get MVC out of android, but it is not intuitive. – Paul Nikonowicz Dec 13 '11 at 16:57

14 I'm writing a Android Architecture series which teaches/demonstrates how to implement MVC (and other patterns) in Android. Check it out: [illegible]android-architecture-part-1-intro – musselwhizzle Dec 20 '11 at 3:02

MV(C)(P) in Android

- Model role: domain model code
- View role: *Activity*, but:
  - Android creates an Activity and manages the lifecycle
  - controller-ish hooks like:
    - *boolean dispatchKeyEvent(KeyEvent event); and*
      boolean onOptionsItemSelected(MenuItem item); and
      void onCreateContextMenu(….);
  - Suppose you want to use it as View
    - data binding: bind to the model on creation and set the activity as an observer of the model
    - sending messages: delegate user input to the controller
    - handles messages from the controller
  - Controller
    - updates the model
    - handles messages from the view
    - sends messages to the view

Problem

- Hooks in the *View (i.e., Activity)* are not related to view behaviour.
- The view is too heavy weight
- !!!Need a clear separation of the MVC parts without overlap!!

```java
public class MainActivity extends Activity {

	private AppModel model;
	private MainView view;
	private Handler handler;
	private boolean isTimerRunning = true;
	private long initTime = 0;

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		initTime = System.currentTimeMillis();
		model = AppModel.getInstance();
		view = (MainView)View.inflate(this, R.layout.main, null);
		view.setViewListener(viewListener);
		setContentView(view);
		handler = new Handler();
	}

	@Override
	protected void onResume() {
		super.onResume();
		timerRun.run();
	}

	@Override
	protected void onPause() {
		super.onPause();
		handler.removeCallbacks(timerRun);
	}

	@Override
	protected void onDestroy() {
		super.onDestroy();
		view.destroy();
	}

	/**
	 * Simple runnable to update our current time in the  model
	 */
	private Runnable timerRun = new Runnable() {
		@Override
		public void run() {
			if (isTimerRunning) {
				long change = System.currentTimeMillis() - initTime;
				initTime = System.currentTimeMillis();
				model.setElapsedTime(model.getElapsedTime() + change); // controller is responsible for updating the model
				handler.postDelayed(timerRun, 100);
			}
		}
	};
```

controller

view instantiated

```java
/**
 * This is how we receive events from the view.
 * The view takes user actions
 * The controller/activity responds to user actions
 */
private MainView.ViewListener viewListener = new MainView.ViewListener() {
    @Override
    public void onToggleTimer() {
        isTimerRunning = !isTimerRunning;
        view.setPausedState(isTimerRunning); // controller can call method directly on the view
        if (isTimerRunning) timerRun.run();
    }

    @Override
    public void onAddTime(long amountToAdd) {
        model.setElapsedTime(model.getElapsedTime() + amountToAdd);
    }
};
}
```

```java
public class MainView extends LinearLayout {

    /**
     * The interface to send events from the view to the controller
     */
    public static interface ViewListener {
        public void onToggleTimer();
        public void onAddTime(long amountToAdd);
    }

    private static boolean DEBUG = false;
    private static final String TAG = MainView.class.getSimpleName();
    private static final long AMOUNT_TO_ADD = 1234 * 60 * 2;
    private Digit d1, d2, d3, d4, d5;
    private Button toggleBtn, addBtn;
    private AppModel model;

    /**
     * The listener reference for sending events
     */
    private ViewListener viewListener;
    public void setViewListener(ViewListener viewListener) {
        this.viewListener = viewListener;
    }

    /**
     * Constructor for xml layouts
     */
    public MainView(Context context, AttributeSet attrs) {
        super(context, attrs);
        model = AppModel.getInstance();
    }

    /**
     * Exposed method so the controller can set the button state.
     */
    public void setPausedState(boolean isTimerRunning) {
        String txt = (isTimerRunning) ? getContext().getString(R.string.stop) : getContext().getString(R.string.start);
        toggleBtn.setText(txt);
    }

    /**
     * Remove the listener from the model
     */
    public void destroy() {
        model.removeListener(AppModel.ChangeEvent.ELAPSED_TIME_CHANGED, elapsedTimeListener);
    }
```

```java
/**
 * Does the work to update the view when the model changes.
 */
private void bind() {
     int milli = (int)Math.floor((model.getElapsedTime() % 1000));
     int secs = (int)Math.floor((model.getElapsedTime() / 1000) % 60);
     int mins = (int)Math.floor((model.getElapsedTime() / 1000 / 60) % 60);

     if (DEBUG) {
          Log.i(TAG, "elapsed: " + model.getElapsedTime());
          Log.i(TAG, "secs: " + secs);
          Log.i(TAG, "mins: " + mins);
     }

     d1.showTime((int)Math.floor(mins/10));
     d2.showTime(mins % 10);
     d3.showTime((int)Math.floor(secs/10));
     d4.showTime(secs % 10);
     d5.showTime((int)Math.floor(milli/100));
}
```

```
/**
 * Find our references to the objects in the xml layout
 */
@Override
protected void onFinishInflate() {
    super.onFinishInflate();
    toggleBtn = (Button)findViewById(R.id.toggleBtn);
    addBtn = (Button)findViewById(R.id.addTimeBtn);
    d1 = (Digit)findViewById(R.id.digit1);
    d2 = (Digit)findViewById(R.id.digit2);
    d3 = (Digit)findViewById(R.id.digit3);
    d4 = (Digit)findViewById(R.id.digit4);
    d5 = (Digit)findViewById(R.id.digit5);
    DigitObjectPool pool = new DigitObjectPool(getContext(), 10);
    d1.setPool(pool);
    d2.setPool(pool);
    d3.setPool(pool);
    d4.setPool(pool);
    d5.setPool(pool);

    toggleBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            viewListener.onToggleTimer();
        }
    });
    addBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            viewListener.onAddTime(AMOUNT_TO_ADD);
        }
    });
    model.addListener(AppModel.ChangeEvent.ELAPSED_TIME_CHANGED, elapsedTimeListener);
    bind();
}

/**
 * The listener for when the elapsed time property changes on the model
 */
private EventListener elapsedTimeListener = new EventListener() {
    @Override
    public void onEvent(Event event) {
        bind();
    }
};
}
```

REMEMBER

- Application of a pattern needs to be considered for its applicability to a problem
- Use of DP should be
  - the result of having a problem for which an existing pattern is known or found to be applicable
  - not the result of starting with a pattern for which a problem was searched or invented!!!