Frameworks

- A reusable, semi-complete application
- An object-oriented technique
- Reuse of both code & design
- Hollywood Principle + Hooks
- Enables the creation of Components
- Origin of many design patterns
- Many application domains
  - Our focus: User interfaces

Frameworks: Swing Case Study

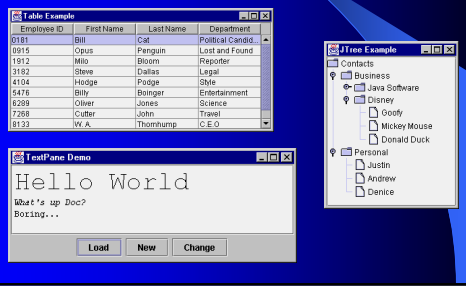
David Talby

The Problem

- Hardware and operating system support primitive I/O operations
- Drawing pixels and lines on the screen
  - Class java.awt.Graphics has *drawLine(), setColor(), fillRect(), setFont(), …* methods
- Receiving user input
  - Reading file and device input streams
  - Platform-dependent

Swing

- Java's User Interface FW since JDK 1.2



Swing Features

- Wide variety of visual components
  - Button, Label, List, Panel, Table, Tree, …
  - Standard Dialog Boxes (Open, Save, Color, …)
- Pluggable look-and feel
  - Platform independence
  - Dynamically changeable
- MVC architecture
  - Facilitates writing components or look-and-feels
- Action objects
  - Shared commands in toolbars and menus
  - Generic Undo capability

The Problem II

- Visual components are not reused
  - Should be in standard library
  - Look-and-feel should be consistent
  - Easy to create / buy new visual components
- Design of user interface is not reused
  - Separating visual design, data structures, user input handling and applicative code
- Code is not platform-independent
- A lot of code & design is required

A Simple Form

- The application will show this dialog box:

- We'll use JButton and JTextField
- Inside a JPanel container
- Inside a JFrame (a window container)
- Whose main() method will run the show

Swing Features II

- Keystroke Handling
  - Global, form, container and component shortcuts
  - Conflict management
- Nested Containers
  - Windows, Dialogs, Frames, Panels, Tables, …
  - Virtually anything can be in anything, at any depth
- Text Manipulation
  - HTML and RTF editing (multi-font, colors, etc.)
- Accessibility
  - Alternative user interface support: Braille, sound…

The Simple Form Code

- Step 1 is to subclass JPanel:

```
class SimplePanel extends JPanel {
    JTextField textField;
    JButton button;
    public SimplePanel() {
        button = new JButton("Clear Text");
        add(button);
        textField = new JTextField(10);
        add(textField);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                textField.setText("");     }   });
} }
```

Anonymous Classes Syntax

- We need a little extra Java syntax first
- Given an interface, it's possible to construct an object that implements it by providing an implementation during the initialization
- For example, since the *ActionListener* interface only defines the *actionPerformed()* method:

```
ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        textField.setText(""); }   } );
```

- The "new" class is an inner class of its scope
- This is the anonymous inner classes syntax

The Simple Form Code III

- The same class also contains main():

```
public static void main(String args[]) {
    JFrame frame =
        new SimplePanelTest("SimplePanel Example");
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);     }
    });
    frame.setSize(WIDTH, HEIGHT);
    frame.setVisible(true);   }  }
```

- The framework takes over after main()

The Simple Form Code II

- Step 2 is to subclass JFrame:

```
public class SimplePanelTest extends JFrame {
    static final int WIDTH = 300;
    static final int HEIGHT = 100;
    SimplePanelTest(String title) {
        super(title);
        SimplePanel simplePanel = new SimplePanel();
        Container c = getContentPane();
        c.add(simplePanel, BorderLayout.CENTER);
    }
```

Component Hierarchy

- Common ancestor is *javax.swing.JComponent*
  - Listener registration for keyboard & mouse events
  - Painting infrastructure: double buffering, borders
  - Keyboard mapping, custom properties, tool-tips, look and feel, accessibility, serialization, …
- Except for top-level heavyweight containers:
  - *JFrame*, *JDialog*, *JApplet*, *JWindow*
- Any subclass of JComponent is a component
  - Easy to write ComboBox, DatePicker, ImageList
  - Standard dialog boxes are implemented this way

The Framework in Action

- Inversion of Control
  - Event loop is handled by a Swing thread
  - Hardware- and OS-specific input formats are translated to standard interfaces
- Hooks
  - Building the visual controls is white-box style
  - Registering to events is black-box style
- Design Patterns
  - Composite: *JPanel.add(Component c)*
  - Observer: *JButton.addActionListener(al)*

Text Editor Kits

- Editor kits can do two things
  - Read & write documents in a particular format
  - Hold a list of actions supported by that format
- Predefined kits: Default, Styled, HTML, RTF
- Editor kits dynamically define text editors:
  - Each EditorKit registers with the JEditorPane
  - When a file is loaded into the pane, it checks its format against the registered editor kits
  - The matching kit reads, writes and edits the text
- Follows the State design pattern

Editor Actions

- All text editors share some commands
  - Cut, Copy, Paste, Clear, Insert Special Char, …
- These are encapsulated in *Action* objects
- Each text component supports *getActions()*
- Actions are added to menus, toolbars, etc:

```
JMenu menu = new JMenu("Edit");
menu.add(cutAction);
```

- Action objects are shared by default
  - Don't modify them
- Follows the Command design pattern

Documents and Views

- Editor Kits follow the Builder design pattern
- Input: The Document interface
  - Content, Mutation, Notification, Properties
  - Structure: an Hierarchy of *Element* objects
- Output: The View interface
  - Participate in layout, paint its portion of document, translate coordinate systems, respond to events
- These interfaces are only used internally
  - To enable reuse of builder, data Structure, view hierarchy and data parser separately

Creating Kits

- To support a new format & actions
  - Extend *DefaultEditorKit* or *StyledEditorKit*
  - Kit may store date about its current EditorPane
  - Call *JEditorPane.registerEditorKitForContentType*
  - Kits are created by cloning the prototype
  - However, this is done by reflection on class name

Keystroke Mapping

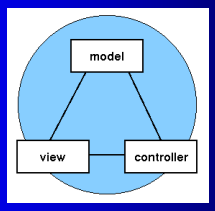
- A *KeyMap* is a *<KeyStroke,Action>* map
- A text component has one or more Keymaps
  - By default, only *JTextComponent.DEFAULT_KEYMAP*
  - Can be modified, but the default Keymap is shared
- Custom Keymaps can be added and removed
  - Adding requires a name and a parent Keymap
  - Then, *addActionForKeyStroke()* can modify it
- Keymap matching is by most-specific-first
- Follows the Chain of Responsibility Pattern

The HTMLEditorKit

- An Abstract Factory for creating views
  - Interface *ViewFactory* has *View create(Element e)*
  - There are HTMLViewFactory and BasicTextUI
- Very customizable using Factory Methods
  - Replace document: override *createDefaultDocument*
  - Replace views: override *getViewFactory*
  - Replace parser: override *getParser*
  - EditorKit responsible only for *read()* and *write()*
- Multiple views per document, for printing

Model / View / Controller

- The Basic User Interface Design Pattern
  - Origin is SmallTalk-80, the first OOFW



Undo / Redo

- Package *javax.swing.undo* offers *UndoableEdit*
  - Has undo(), redo(), canUndo(), canRedo(), die(), isSignificant(), getPresentationName() methods
  - Package offers an *AbstractUndoableEdit* class
  - And a *CompoundEdit* class for composite commands
- Class *UndoManager* manages done commands
  - Extends *CompoundEdit* - has addEdit() method
  - undo(), redo(), setLimit(), trimEdits(), undoTo(), redoTo(), undoOrRedo(), discardAllEdits(), …
- Each text component supports:
  addUndoableEditListener(UndoableEditListener uel)

MVC by Example

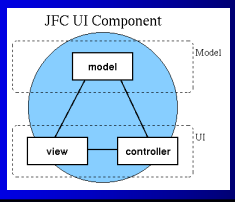
- ButtonModel class
  - Private *color* field, *setColor()* and *getColor()*
  - *setColor()* will also notify observers
- ButtonView class
  - Gets a *ButtonModel* in its constructor, stores is in a private field and registers to it
  - Has a public *paint()* method, called on notification
- MyController class
  - Initialized with both model and view objects
  - Handles user input events (mouse/keyboard/etc.) by changing the model and the view

MVC Participants

- Model
  - Data structure of displayed data
  - Notifies observers on state change
- View
  - Paints the data on the screen
  - Observer on its model
- Controller
  - Handles user input
  - Changes model, which causes views to update

Document / View

- View and Controller are often merged
  - MFC: "Document" and "View"
  - Swing (Text Editors): "Document" and "View"
  - Swing (Components): "Model" and "Delegate"

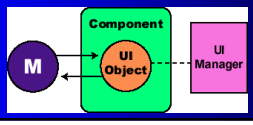


MVC Benefits

- Three elements can be reused separately
- Synchronized user interface made easy
  - Multiple views observe one model
- Models know nothing about presentation
  - Easy to modify or create views
  - Easy to dynamically change views
- More efficient
  - Shared models & controllers save memory
  - Easy to maintain pools of views and models

MVC Inside a Component

- Each component is a façade for two objects
  - Each components defines getModel() and getUI()
  - Usually only one component per model and delegate
- UIManager is a singleton
  - Holds current look & feel properties
- ComponentUI defines drawing interface
  - javax.swing.plaf.* includes ButtonUI, SliderUI, …



Swing and MVC

- There are several levels of using MVC
- "Manually", to synchronize complex views
  - A file explorer, with a tree and current dir
- In forms or complex components
  - Custom form logic to synchronize its field
  - A table or tree and its sub-components
  - A variation of the Mediator design pattern
- A component is a façade to two objects:
  - Model: data structure of property values
  - Delegate: handles painting and user input

The Façade Pattern

- A flexible framework becomes very complex
- It is important to provide simple façades
- JEditorPane class
  - No need to know EditorKit & its subclasses, Document, Element, View, ViewFactory, KeyMap, …
- JButton class
  - No need to know ComponentModel, ComponentUI, UIManager, LookAndFeel, …
- Provide users only with concepts they know
  - ✓ Button, Window, Action, Menu
  - ✗ Document, ViewFactory, EditorKit

Changing Look & Feel

- The default is in a text configuration file

  homepath/lib/swing.properties
- At runtime before creating components:

```
UIManager.setLookAndFeel(
"com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```
- At runtime after initialization:

```
UIManager.setLookAndFeel(lnfName);
SwingUtilities.updateComponentTreeUI(frame);
frame.pack();   // in case sizes have changed
```
- Replaces the ComponentUI object
  - Follows the Strategy pattern for *paint()*

Summary

- Swing is a classic OOD framework
  - Contains a lot of domain knowledge
  - Highly customizable through design patterns
  - Comes with a set of implemented components
  - Also intended for writing new ones
  - Inversion of control + hooks
- It's a medium-sized framework
  - Several hundred classes and interfaces
  - Plus free & commercial 3rd party components

Other Features

- Swing supports several other features that we don't have time to cover:
  - Drag & Drop
  - Printing
  - Internationalization
  - Trees and Tables
  - Menus & Popup menus
  - Layout Management
- Other standard Java graphic libraries:
  - 2D drawing, 3D drawing, Multimedia