Frameworks

- "A reusable, semi-complete application that can be specialized to produce a custom application"
- "A set of cooperating abstract and concrete classes that makes up a reusable design for a specific class of software"
- An Object-Oriented Reuse Technique
  - Design Reuse + Code Reuse

Frameworks

David Talby

Designing an OO Framework

1. Domain Knowledge
   - What applications is the framework for?
   - What is common to all of them?
2. Architecture
   - Biggest, most critical technical decisions
   - What is required besides classes?
3. Object-oriented design
   - Design Reuse: Patterns
   - Inversion of Control + Find right hooks

A Tiny Example: Calculators

- interface Calculator
  - getValue(), compute(Operator o), clear(), undo()
  - Uses Command pattern, optionally Singleton
  - Remembers parentheses, can iterate on their tree
- interface Operator
  - Descendants: UnaryOperator, BinaryOperator
  - Concrete classes: Plus, Minus, Power, …
  - Acts as Command class, supports Composites
- interface VisualCalculator
  - Observer on Calculator, can display operators on buttons, can display current computation tree
- All are extendible, "Main" receives interfaces

Architecture

- The set of significant decisions about the structure of software, the division to components and subsystems and their interfaces, and guidelines to composing them
- Common "significant" decisions:
  - Programming language, operating system, hardware
  - Use of major external libraries or applications
  - Physical Distribution, processes and threads
  - Main Concepts: Kinds of modules and interfaces
  - Communication and synchronization between modules
  - Security Model
  - Performance and scalability

Domain Knowledge

- a.k.a. Analysis or Modeling
- Common "significant" decisions:
  - Major concepts of the modeled domain
  - Major operations
  - Use cases: How users do common tasks
- For example, a calculator
  - Concepts: unary operator, binary operator, current value, in-memory value, shift key
  - Operations: Clear, use operator, compute
  - Use case: Computing an average of *n* numbers

JCA II

- Generating a public/private key pair:

```
KeyPairGenerator keygen =
    KeyPairGenerator.getInstance("DSA", "MY_PROVIDER");
keygen.initialize(keySize, new SecureRandom(userSeed));
KeyPair pair = keygen.generateKeyPair();
```

  - Cast to *DSAKeyPairGenerator* is required to initialize it with algorithm-specific parameters (p,q,g)
- Generating a signature:

```
Signature sha = Signature.getInstance("SHA-1");
PrivateKey priv = pair.getPrivate();
sha.initSign(priv);
byte[] sig = sha.sign();
```

  - Provider is optional in *getInstance()*

For Example: JCA

- Java Cryptography Architecture (JCA, JCE)
  - Encryption, Digital Signatures, Key Management
  - Open for new algorithms, new implementations
- Main Concepts
  - Provider: provides implementations for a subset of the Java Security API, identified by name
  - Engine Classes: functionality for a type of crypto behavior, such as *Signature* and *KeyPairGenerator*
  - Factory Methods: static methods in engine classes that return instances of them for a given algorithm
  - Key Store: System identity scope

JCA IV: Summary

- So what does the architecture answer?
  - Domain Knowledge: What behavior (engine classes) should be supported at all?
  - How are different algorithms and different implementations defined and selected?
  - How should non-Java implementations be used?
  - How can an administrator configure a key store and a trusted set of providers and implementations?
  - How can commercial companies sell Java-compatible closed-source implementations of security features
- Not only classes and interfaces
  - Persistent key store, config files, non-Java code
  - Practical, management and economic considerations

JCA III

- Although implementations will usually be non-Java, they must be wrapped in Java classes
- Statically, add lines to java.security text file
  - Security.providerName.n = com.acme.providerPackage
  - n is the preference order of the provider, 1 is highest
- Providers can be managed dynamically too:
  - Class Security has addProvider(), getProvider()
  - Class Provider has getName(), getVersion(), getInfo()
- Providers must write a "Master class"
  - Specifies which implementations are offered by it
  - There are standard names for known algorithms

Hooks

- Hook = Hotspot = Plug-point
  - Points where the FW can be customized
- Design issues requiring domain knowledge
  - How to find the right hooks?
  - Few or many hooks?
  - What should be the default behavior?
- Implementation alternatives
  - Template Method
  - Strategy or Prototype
  - Observer

Inversion of Control

- a.k.a. The Hollywood Principle
- Don't call us, we'll call you
- Benefits
  - Code reuse: Flow of control is coded once
  - Makes it clear when and how hooks are called
  - Produces uniformity in program behavior, which makes it easier to understand
- Drawbacks
  - Debugging is more difficult
  - Integrating two frameworks can be hard

Framework Colors II

- Frameworks tend to evolve to being black-box
- AWT 1.0 had a white-box event model
  - Each visual component had an *handleEvent()* method
  - Each frame inherited and overrode it
  - The method was a long *switch* statement
- AWT 1.1 and Swing are black-box
  - Observer pattern: UI components publish events to registered listeners
- Why is black-box better?
  - Separation of concerns: better abstractions
  - Important for (automatic) code generation

Framework Colors

- White-Box Frameworks
  - Extended by inheritance from framework classes
  - Template Method, Builder, Bridge, Abstract Factory
  - Require intimate knowledge of framework structure
- Black-Box Frameworks
  - Extended by composition with framework classes
  - Strategy, State, Visitor, Prototype, Observer
  - More flexible, slightly less efficient
- Gray-box Frameworks
  - What usually happens in real life…

Application Domains

- System Infrastructure
  - Operating System Wrappers: MFC, MacApp
  - Communication Protocols: RMI
  - Database Access: ADO, JDO
  - Security: JCA, JSA
- User Interfaces
  - SmallTalk-80 is the first widely used OOFW
  - Swing, Delphi, MFC, COM…
  - Integrated with development environments

Designing an OO Framework

1. Domain Knowledge
   - What applications is the framework for?
   - What is common to all of them?
2. Architecture
   - Biggest, most critical technical decisions
   - What is required besides classes?
3. Object-oriented design
   - Design Reuse: Patterns
   - Inversion of Control + Find right hooks

Framework Strengths

- Reuse, Reuse, Reuse!
  - Design + Code
- Extensibility
  - Enables the creation of reusable Components
- Enforced Design Reuse
  - An "Educational" Tool
- Partitioning of Knowledge & Training
  - Technical vs. Applicative Specialization

Application Domains II

- Middleware / Object Request Brokers
  - Object Request Brokers: CORBA, COM+, EJB
  - Web Services: .NET, Sun One
- Enterprise Applications
  - Enterprise = Critical to day-to-day work
  - Usually developed inside organizations
  - Notable Exception: IBM's San-Francisco
  - Telecomm, Manufacturing, Avionics, Finance, Insurance, Healthcare, Warehouses, Billing…

There's Big Money Involved

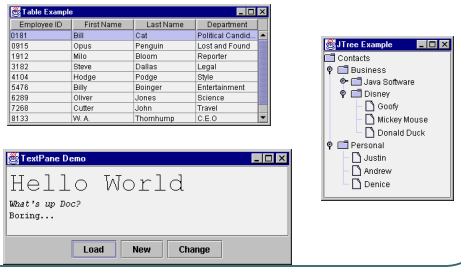
- All "big players" develop and sell FWs
  - So you must use our language (Swing)
  - So you must use our operating system (MFC)
  - So you must use our development tool (Delphi)
  - So you must use our database (Oracle)
- There's a component industry too
  - Companies that write and sell components
- Frameworks are an economic necessity
  - Unwise to develop UI, DB, ORB alone today

Framework Weaknesses

- Development effort
  - Generic frameworks are harder to design and build
  - They are also hard to validate and debug
- Maintenance
  - Does the FW or the app need to change?
  - Interface changes require updating all apps
- Learning Curve
  - Unlike class libraries, you can't learn one class at a time
- Integratibility of multiple frameworks
- Efficiency
- Lack of standards

Swing

- Java's User Interface FW since JDK 1.2



Frameworks: Swing Case Study

David Talby

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

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 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…

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 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("");     }   });
} }
```


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

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);
    }
```

Patterns in Swing

- Command
  - All text editors shared some commands (cut, paste)
  - These are encapsulated in *Action* objects
  - Each text components supports *getActions()*
  - *Action* objects are globally shared, and are used on menu items, toolbars and shortcuts
- Strategy
  - The *LookAndFeel* interface has several heirs
  - The *UIManager* singleton points to the current one
  - It has methods to dynamically change look and feel

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)*

Patterns in Swing III

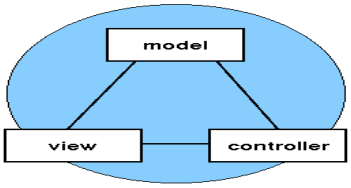
- Builder
  - Editor Kits act as builders: their input is the hirarchical *Document* interface
  - Their output is the *View* interface: has paint method, can layout, translate coordinates, …
- Abstract Factory
  - interface *ViewFactory* creates views
  - Two heirs: HTMLViewFactory, BasicTextUI
  - Another singleton
- Factory Method
  - Editor kits use factory methods to facilitate changing parser, view factory and default document

Patterns in Swing II

- State
  - JEditorPane is a visual editor that supports reading and writing files in multiple formats
  - Each format is represented by an *EditorKit* that registers with the JEditorPane
  - Upon reading a file, its format is used to select an EditorKit for it
  - That kit is used to read, write, list actions, …
- Prototype
  - Editor Kits are created by cloning the prototype
  - However, this is done by reflection on class name

Model / View / Controller

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



Patterns in Swing IV

- Chain of Responsibility
  - A *KeyMap* is a *<KeyStroke,Action>* map
  - A text component has one or more Keymaps
  - Custom Keymaps can be added and removed
  - Keymap matching is by most-specific-first
- Command
  - Package *javax.swing.undo* offers *UndoableEdit*
  - Also *AbstractUndoableEdit* and *CompoundEdit* classes
  - Class *UndoManager* manages done commands
  - Extends *CompoundEdit* - supports addEdit(), undo(), redo(), setLimit(), trimEdits(), undoTo(), redoTo(), …

MVC Benefits

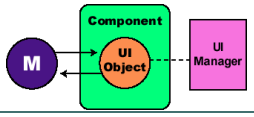
- Three elements can be reused separately
- Synchronized user interface made easy
  - Multiple views observe one model
- Models know nothing about presentation
  - Easy to modify or create views
  - Easy to dynamically change views
- More efficient
  - Shared models & controllers save memory
  - Easy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

MVC 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

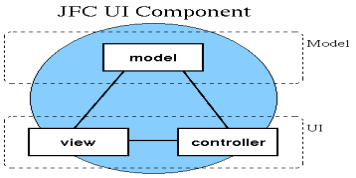
MVC Inside a Component

- 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, …



Document / View

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



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
  - X Document, ViewFactory, EditorKit

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
- Event-handling at the application level

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