Principles of Software Construction: Objects, Design, and Concurrency

Frameworks

Charlie Garrod    Jonathan Aldrich

With material from Ciera Jaspan, Jonathan Aldrich, Bill Scherlis, Travis Breaux, Erich Gamma, and Christian Kaestner

© 2012-14  C Kästner, C Garrod, J Aldrich, and W Scherlis
Key concepts from Tuesday
Design patterns we have seen so far

- Iterator
- Composite
- Template Method
- Adapter
- Strategy
- Marker Interface
- Decorator
- Model-View-Controller
- Factory Method
Design patterns we have seen so far

- Iterator
- Composite
- Template Method
- Adapter
- Strategy
- Marker Interface
- Decorator
- Model-View-Controller
- Factory Method
A note on design

• The previous exercise is backward
  • "Here's a design pattern. Now use it..."

• The real exercise: "How do I design this program to accomplish my goals?..."
  • "Aha! I've seen this problem before..."
The design patterns you should know…

- Abstract Factory
- Observer
- Composite
- Template Method
- Iterator
- Marker Interface
- Strategy
- Façade
- Decorator
- Model-View-Controller
- Adapter
- Command
- Factory Method
The design patterns you should know…

- Abstract Factory
- Observer
- Composite
- Template Method
- Strategy
- Iterator
- Façade
- Decorator
- Marker Interface
- Model-View-Controller
- Factory Method
- Command
Administrivia

- Homework 4b due next Tuesday
- Homework 4c due one week after that
- Midterm exam Thursday, October 30th
Today, continued next week

• Design for reuse: Libraries and frameworks
  ▪ Motivation: reuse with variation
  ▪ Examples, terminology
  ▪ Whitebox and blackbox frameworks
  ▪ Design considerations
  ▪ Implementation details
    ▪ Responsibility for running the framework
    ▪ Loading plugins
Learning goals

• Describe example well-known example frameworks
• Know key terminology related to frameworks
• Know common design patterns in different types of frameworks
• Discuss differences in design trade-offs for libraries vs. frameworks
• Analyze a problem domain to define commonalities and extension points (cold spots and hot spots)
  ▪ Analyze trade-offs in the use vs. reuse dilemma
• Know common framework implementation choices
Reuse and variation

Homework #0: A Friendship Graph
Due Tuesday, January 21st at 11:59 p.m.

The goals of this assignment are to familiarize you with our course infrastructure and let you practice Java object-oriented programming. To complete this assignment you implement a simple graph class that could represent a network of friendships.

Your learning goals for this assignment are to:

- Use Git and GitHub to revision and share work
- Get familiar with Java development environments
- Write a first Java program
- Practice Java style and coding conventions

Instructions

To begin your work, import the project into Eclipse from the home repository.

Implement and test a Friendship class that represents friendships can compute the distance between two people in the graph. Your network as an undirected graph where each person is connected to every other person.

For example, suppose you have the following social network:

![Social Network Diagram]

Rachel - Ross - Ben

Homework #1: Graph Algorithms for Social Networks
Due Tuesday, January 28th at 11:59 p.m.

In this assignment, you will implement a Graph interface using two different graph representations. You will then develop several algorithms that use the Graph interface that might be used in a social network.

Your goals for this assignment are to:

- Understand and apply the concepts of polymorphism and encapsulation
- Understand interfaces
- Familiarize yourself with Java and Eclipse
- Learn to compile, run, and debug Java programs
- Practice good Java coding practices and style

Instructions

Graph Implementations

First, write two classes that implement the edu.cmu.cs.ca214.hw1.staff.Graph interface, which represents an undirected graph.

- **Adjacency List:** Inside the package edu.cmu.cs.ca214.hw1.graph, implement the AdjacencyListGraph class. Your implementation must internally represent the graph as an adjacency list. Your class should provide a constructor with a single int argument, maxVertices. Your implementation must then support a graph containing as many as maxVertices vertices. Your implementation may behave arbitrarily.
Reuse and variation: The Standard Widget Toolkit
Reuse and variation: Family of development tools
Reuse and variation: Eclipse Rich Client Platform
Reuse and variation: Web browser extensions
Reuse and variation: Flavors of Linux
Reuse and variation: Flavors of Linux
Reuse and variation: Printer product lines
The promise:

Costs

Development with reuse

# Products
Approaches to reuse and variation

• "Clone and own"
• Subroutines
• Libraries
• Frameworks
• APIs
• Platforms
• Configuration
• Software product lines
Terminology: Libraries

- **Library**: A set of classes and methods that provide reusable functionality
- **Client calls library to do some task**
- **Client controls**
  - System structure
  - Control flow
- **The library executes a function and returns data**
Terminology: Frameworks

- **Framework**: Reusable skeleton code that can be customized into an application

- **Framework controls**
  - Program structure
  - Control flow

- **Framework calls back into client code**
  - The Hollywood principle: “Don’t call us. We’ll call you.”

```java
public MyWidget extends JContainer {
    public MyWidget(int param) { // setup internals, without rendering
        // render component on first view and resizing
        protected void paintComponent(Graphics g) {
            // draw a red box on this component
            Dimension d = getSize();
            g.setColor(Color.red);
            g.drawRect(0, 0, d.getWidth(), d.getHeight());
        }
    }
}
```
public class Calc extends JFrame {
    private JTextField textfield;
    public static void main(String[] args) { new Calc().setVisible(true); }
    public Calc() { init(); }
    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText("calculate");
        contentPane.add(button, BorderLayout.EAST);
        textfield = new JTextField(
"");
        textfield.setText("10 / 2 + 6");
        textfield.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        button.addActionListener("calculate some stuff */);
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle("My Great Calculator");
        // impl. for closing the window
    }
}
A simple example framework

- Consider a family of programs consisting of buttons and text fields only:

- What source code might be shared?
A simple example framework

- Consider a family of programs consisting of buttons and text fields only:
  - What source code might be shared?
    - Main method
    - Initialization of GUI
    - Layout
    - Closing the window
    - ...

![Example GUIs](My Great Calculator, Ping, File Uploader)
A calculator example (without a framework)

```java
public class Calc extends JFrame {
    private JTextField textfield;
    public static void main(String[] args) { new Calc().setVisible(true); }
    public Calc() { init(); }
    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText("calculate");
        contentPane.add(button, BorderLayout.EAST);
        textfield = new JTextField("10 / 2 + 6");
        textfield.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        button.addActionListener(/*calculate some stuff*/);
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle("My Great Calculator");
        // impl. for closing the window
    }
}
```
public abstract class Application extends JFrame {
    protected abstract String getApplicationTitle();
    protected abstract String getButtonText();
    protected String getInitialText() {return "";}
    protected void buttonClicked() { }

    private JTextField textfield;
    public Application() { init(); }
    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText(getButtonText());
        button.addActionListener(/* ... buttonClicked(); ... */);
        contentPane.add(button, BorderLayout.EAST);
        textfield = new JTextField("");
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle(getApplicationTitle());
    // impl. for closing the window
    }
    protected String getInput() { return textfield.getText();}
}
Using the simple example framework

```java
public abstract class Application extends JFrame {
    protected abstract String getApplicationTitle();
    protected abstract String getButtonText();
    protected String getInititalText() {return "";}
    protected void buttonClicked() { }
    private JTextField textfield;
    public Application() { init(); }
    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText(getButtonText());
        contentPane.add(button, BorderLayout.EAST);
        textfield = new JTextField("uellement");
        textfield.setText(getInititalText());
        textfield.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        button.addActionListener(/* … buttonClicked(); … */);
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle(getApplicationTitle());
    }
    protected String getInput() { return textfield.getText(); }
}

public class Calculator extends Application {
    protected String getButtonText() { return "calculate"; }
    protected String getInititalText() { return "(10 – 3) * 6"; }
    protected void buttonClicked() {
        JOptionPane.showMessageDialog(this, "The result of " + getInput() + " is " + calculate(getInput()));
    }
    protected String getApplicationTitle() { return "My Great Calculator"; }
    public static void main(String[] args) {
        new Calculator().setVisible(true);
    }
}
```

Using the simple example framework (again)

```java
public abstract class Application extends JFrame {
    protected abstract String getApplicationTitle();
    protected abstract String getButtonText();
    protected String getInititalText() {return "";}
    protected void buttonClicked() { }
    private JTextField textfield;

    public Application() { init(); }
    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText(getButtonText());
        contentPane.add(button, BorderLayout.EAST);
        textfield = new JTextField("");
        textfield.setText(getInititalText());
        textfield.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        button.addActionListener(/* … buttonClicked(); … */);
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle(getApplicationTitle());
    }
    protected String getInput() { return textfield.getText(); }

    public class Calculator extends Application {
        protected String getButtonText() { return "calculate"; }
        protected String getInititalText() { return "(10 – 3) * 6"; }
        protected void buttonClicked() {
            JOptionPane.showMessageDialog(this, "The result of "+getInput()+" is "+calculate(getInput()));
        }
        protected String getApplicationTitle() { return "My Great Calculator"; }
        public static void main(String[] args) {
            new Calculator().setVisible(true);
        }
    }

    public class Ping extends Application {
        protected String getButtonText() { return "ping"; }
        protected String getInititalText() { return "127.0.0.1"; }
        protected void buttonClicked() { /* … */ }
        protected String getApplicationTitle() { return "Ping Anything"; }
        public static void main(String[] args) {
            new Ping().setVisible(true);
        }
    }
}
```
General distinction: Library vs. framework

public MyWidget extends JComponent {
    public MyWidget(int param) { // setup internals, without rendering }
    /\ render component on first view and resizing
    protected void paintComponent(Graphics g) {
// draw a red box on his component
    g.setColor(Color.red);
    g.drawRect(0, 0, d.getWidth(), d.getHeight());
    }
}

public MyWidget extends JComponent {
    public MyWidget(int param) { // setup internals, without rendering }
    /\ render component on first view and resizing
    protected void paintComponent(Graphics g) {
// draw a red box on his component
    g.setColor(Color.red);
    g.drawRect(0, 0, d.getWidth(), d.getHeight());
    }
}
More terms

- **API**: Application Programming Interface, the interface of a library or framework
- **Client**: The code that uses an API
- **Plugin**: Client code that customizes a framework
- **Extension point, hot spot**: A place where a framework supports extension with a plugin
More terms

• **Protocol**: The expected sequence of interactions between the API and the client

• **Callback**: A plugin method that the framework will call to access customized functionality

• **Lifecycle method**: A callback method of an object that gets called in a sequence according to the protocol and the state of the plugin
Whitebox frameworks

- Extension via subclassing and overriding methods
- Common design pattern(s):
  - Template Method
- Design steps:
  - Identify the common code and the variable code
  - Abstract variable code as method calls
- Subclass has main method but gives control to framework
Blackbox frameworks

- Extension via implementing a plugin interface
- Common design pattern(s):
  - Strategy
  - Observer
- Design steps:
  - Identify the common code and the variable code
  - Abstract variable code as methods of an interface
  - Decide whether there might be one or multiple plugins
- Plugin-loading mechanism loads plugins and gives control to the framework
Is this a whitebox or blackbox framework?

```java
public abstract class Application extends JFrame {
    protected abstract String getApplicationTitle();
    protected abstract String getButtonText();
    protected String getInititalText() { return ""; }
    protected void buttonClicked() { }

    private JTextField textfield;

    public Application() { init(); }

    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout);
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText(getButtonText());
        contentPane.add(button, BorderLayout.EAST);
        textfield = new JTextField("");
        textfield.setText(getInititalText());
        textfield.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        button.addActionListener(/* … buttonClicked(); … */);
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle(getApplicationTitle());
    }  

    protected String getInput() { return textfield.getText(); }

    public class Calculator extends Application {
        protected String getButtonText() { return "calculate"; }
        protected String getInititalText() { return "(10 – 3) * 6"; }
        protected void buttonClicked() {
            JOptionPane.showMessageDialog(this, "The result of "+getInput()+" is "+calculate(getInput()));
        }
        protected String getApplicationTitle() { return "My Great Calculator"; }
        public static void main(String[] args) {
            new Calculator().setVisible(true);
        }
    }

    public class Ping extends Application {
        protected String getButtonText() { return "ping"; }
        protected String getInititalText() { return "127.0.0.1"; }
        protected void buttonClicked() { /* … */ }
        protected String getApplicationTitle() { return "Ping"; }
        public static void main(String[] args) {
            new Ping().setVisible(true);
        }
    }
```
public class Application extends JFrame {
    private JTextField textField;
    private Plugin plugin;
    public Application(Plugin p) { this.plugin=p; p.setApplication(this); init(); }
    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        if (plugin != null)
            button.setText(plugin.getButtonText());
        else
            button.setText("ok");
        contentPane.add(button, BorderLayout.EAST);
        textField = new JTextField("");
        if (plugin != null)
            textField.setText(plugin.getInitialText());
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        if (plugin != null)
            button.addActionListener(/* … plugin.buttonClicked();… */);
        this.setContentPane(contentPane);
    }
    public String getInput() { return textField.getText();
}
public class Application extends JFrame {
    private JTextField textfield;
    private Plugin plugin;
    public Application(Plugin p) { this.plugin=p; p.setApplication(this); }
    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        if (plugin != null)
            button.setText(plugin.getButtonText());
        else
            button.setText("ok");
        contentPane.add(button, BorderLayout.EAST);
        textfield = new JTextField("");
        if (plugin != null)
            textfield.setText(plugin.getInititalText());
        textfield.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        if (plugin != null)
            button.addActionListener(/* … plugin.buttonClicked();… */);
        this.setContentPane(contentPane);
    }
    public String getInput() { return textfield.getText(); }
}

public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked();
    void setApplication(Application app);
}

public class CalcPlugin implements Plugin {
    private Application application;
    public void setApplication(Application app) { this.application = app; }
    public String getButtonText() { return "calculate"; }
    public String getInititalText() { return "10 / 2 + 6"; }
    public void buttonClicked() {
        JOptionPane.showMessageDialog(null, "The result of " + application.getInput() + " is " + calculate(application.getText()));
    }
    public String getApplicationTitle() { return "My Great Calculator"; }
}

public class Calculator {
    public static void main(String[] args) {
        new Application(new CalcPlugin()).setVisible(true);
    }
}
public class Application extends JFrame implements InputProvider {
    private JTextField textfield;
    private Plugin plugin;
    public Application(Plugin p) { this.plugin=p; p.setApplication(this); init(); }
    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout);
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        if (plugin != null)
            button.setText(plugin.getButtonText());
        else
            button.setText("ok");
        contentPane.add(button, BorderLayout.EAST);
        textfield = new JTextField("");
        if (plugin != null)
            textfield.setText(plugin.getInititalText());
        textfield.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        if (plugin != null)
            button.addActionListener(/* … plugin.buttonClicked();… */);
        this.setContentPane(contentPane);
    }
    public String getInput() { return textfield.getText(); }
    ...
}

class CalcStarter {
    public static void main(String[] args) {
        new Application(new CalcPlugin()).setVisible(true);
    }
}

public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked();
    void setApplication(InputProvider app);
}

public interface InputProvider {
    String getInput();
}

public class CalcPlugin implements Plugin {
    private InputProvider application;
    public void setApplication(InputProvider app) { this.application = app; }
    public String getButtonText() { return "calculate"; }
    public String getInititalText() { return "10 / 2 + 6"; }
    public void buttonClicked() {
        JOptionPane.showMessageDialog(null, "The result of " + application.getInput() + " is " + calculate(application.getInput()));
    }
    public String getApplicationTitle() { return "My Great Calculator"; }
}

An aside: Plugins could be reusable, too…
Whitebox vs. blackbox framework summary

- **Whitebox frameworks use subclassing**
  - Allows to extend every nonprivate method
  - Need to understand implementation of superclass
  - Only one extension at a time
  - Compiled together
  - Often so-called developer frameworks

- **Blackbox frameworks use composition**
  - Allows to extend only functionality exposed in interface
  - Only need to understand the interface
  - Multiple plugins
  - Often provides more modularity
  - Separate deployment possible (.jar, .dll, ...)
  - Often so-called end-user frameworks, platforms
Another aside:

- In what way is Homework 4 (Scrabble with Stuff) a framework?
Framework design considerations

• Once designed there is little opportunity for change

• Key decision: Separating common parts from variable parts
  ▪ Identify hot spots vs. cold spots

• Possible problems:
  ▪ Too few extension points: Limited to a narrow class of users
  ▪ Too many extension points: Hard to learn, slow
  ▪ Too generic: Little reuse value

• The golden rule of framework design:
  ▪ Writing a plugin/extension should NOT require modifying the framework source code
public class Application extends JFrame {
    private JTextField textfield;
    private Plugin plugin;
    public Application(Plugin p) { this.plugin=p; p.setApplication(this); init(); }
    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        if (plugin != null)
            button.setText(plugin.getButtonText());
        else
            button.setText("ok");
        contentPane.add(button, BorderLayout.EAST);
        textfield = new JTextField("");
        if (plugin != null)
            textfield.setText(plugin.getInititalText());
        textfield.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        if (plugin != null)
            button.addActionListener(/* … plugin.buttonClicked();… */);
        this.setContentPane(contentPane);
    }
    public String getInput() { return textfield.getText(); }
}

class CalcStarter {
    public static void main(String[] args) {
        new Application(new CalcPlugin()).setVisible(true);
    }
}

The cost of changing a framework

public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked();
    void setApplication(Application app);
}

public class CalcPlugin implements Plugin {
    private Application application;
    public void setApplication(Application app) { this.application = app; }
    public String getButtonText() { return "calculate"; }
    public String getInititalText() { return "10 / 2 + 6"; }
    public void buttonClicked() {
        JOptionPane.showMessageDialog(null, "The result of " + application.getInput() + " is " + calculate(application.getText()));
    }
    public String getApplicationTitle() { return "My Great Calculator"; }
}

Consider adding an extra method.
Many changes require changes to all plugins.
OO frameworks (credit: Erich Gamma)

- A customizable set of cooperating classes that defines a reusable solution for a given problem
  - Defines key abstractions and their interfaces
  - Defines object interactions & invariants
  - Provides flow of control
  - Provides defaults

- Reuse
  - Reuse of design and code
  - Reuse of a macro architecture

- Framework provides architectural guidance
Framework layering (credit: Erich Gamma)

**Framework Package**
- reuse of design
- stable
  - no implementation
- specifies public interfaces

**Defaults/Standard Package**
- reuse of design and code
- less stable
  - more implementation
- specifies protected interfaces

**Kit Packages**
- reuse of implementation
- least stable
  - implementation only

Diagram:
- **Drawing**
- **Figure**
- **StandardDrawing**
- **AbstractFigure**
- **LineConnection**
The use vs. reuse dilemma

• Large rich components are very useful, but rarely fit a specific need

• Small or extremely generic components often fit a specific need, but provide little benefit

“maximizing reuse minimizes use”

C. Szyperski
(one modularization: tangrams)
Domain engineering

• Understand users/customers in your domain
  ▪ What might they need? What extensions are likely?

• Collect example applications before starting a framework/component

• Make a conscious decision what to support
  ▪ So-called scoping

• e.g., the Eclipse policy:
  ▪ Interfaces are internal at first
    ▪ Unsupported, may change
  ▪ Public stable extension points created when there are at least two distinct customers
Typical framework design and implementation

• Identify common parts and variable parts
• Implement common parts
  ▪ Also design and write sample plugins/applications
• Provide plugin interface/extension/callback mechanisms for variable parts
  ▪ Use well-known design principles and patterns: Strategy, Decorator, Observer, Command, Template Method, Factories ...
Evolutionary design: Extract interfaces from classes
(credit: Erich Gamma)

• Extracting interfaces is a new step in evolutionary design:
  ▪ Abstract classes are discovered from concrete classes
  ▪ Interfaces are distilled from abstract classes

• Start once the architecture is stable
  ▪ Remove non-public methods from class
  ▪ Move default implementations into an abstract class which implements the interface
Implementation details: Running a framework

- Some frameworks are runnable by themselves
  - e.g. Eclipse

- Other frameworks must be extended to be run
  - MapReduce, Swing, Servlets, JUnit
Methods to load plugins

- Client writes `main()`, creates a plugin, and passes it to framework
  - (see blackbox example above)

- Framework writes `main()`, client passes name of plugin as a command line argument or environment variable
  - (see next slide)

- Framework looks in a magic location
  - Config files or .jar files are automatically loaded and processed

- GUI for plugin management
An example plugin loader using Java Reflection

```java
public static void main(String[] args) {
    if (args.length != 1)
        System.out.println("Plugin name not specified");
    else {
        String pluginName = args[0];
        try {
            Class<?> pluginClass = Class.forName(pluginName);
            new Application((Plugin) pluginClass.newInstance()).setVisible(true);
        } catch (Exception e) {
            System.out.println("Cannot load plugin " + pluginName + ", reason: " + e);
        }
    }
}
```
Another plugin loader using Java Reflection

```java
public static void main(String[] args) {
    File config = new File("./config");
    BufferedReader reader = new BufferedReader(new FileReader(config));
    Application = new Application();
    Line line = null;
    while ((line = reader.readLine()) != null) {
        try {
            Class<?> pluginClass = Class.forName(pluginName);
            application.addPlugin((Plugin) pluginClass.newInstance());
        } catch (Exception e) {
            System.out.println("Cannot load plugin "+ pluginName
                + ", reason: " + e);
        }
    }
    reader.close();
    application.setVisible(true);
}
```
GUI-based plugin management
Supporting multiple plugins

- Observer design pattern is commonly used
- Load and initialize multiple plugins
- Plugins can register for events
- Multiple plugins can react to same events
- Different interfaces for different events possible

```java
public class Application {
    private List<Plugin> plugins;
    public Application(List<Plugin> plugins) {
        this.plugins = plugins;
        for (Plugin plugin : plugins) {
            plugin.setApplication(this);
        }
    }
    public Message processMsg(Message msg) {
        for (Plugin plugin : plugins) {
            msg = plugin.process(msg);
            ...
        }
        return msg;
    }
}
```
Example: An Eclipse plugin

- A popular Java IDE
- More generally, a framework for tools that facilitate “building, deploying and managing software across the lifecycle.”
- Plugin framework based on OSGI standard
- Starting point: Manifest file
  - Plugin name
  - Activator class
  - Meta-data

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyEditor Plug-in
Bundle-SymbolicName: MyEditor;
singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: myeditor.Activator
Require-Bundle: org.eclipse.ui,
                 org.eclipse.core.runtime,
                 org.eclipse.jface.text,
                 org.eclipse.ui.editors
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```
public class Application extends JFrame implements InputProvider {
    private JTextField textfield;
    private Plugin plugin;
    public Application(Plugin p) { this.plugin = p; p.setApplication(this); init(); }
    protected void init() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        if (plugin != null)
            button.setText(plugin.getButtonText());
        else
            button.setText("ok");
        contentPane.add(button, BorderLayout.EAST);
        textfield = new JTextField("");
        if (plugin != null)
            textfield.setText(plugin.getInititalText());
        textfield.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textfield, BorderLayout.WEST);
        if (plugin != null)
            button.addActionListener(/* … plugin.buttonClicked();… */);
        this.setContentPane(contentPane);
    ...
    }
    public String getInput() { return textfield.getText(); }
}

Recall the aside: Plugins could be reusable, too…

public class CalcPlugin implements Plugin {
    private InputProvider application;
    public void setApplication(InputProvider app) { this.application = app; }
    public String getApplicationTitle() { return "My Great Calculator"; }
    public String getButtonText() { return "calculate"; }
    public String getInititalText() { return "10 / 2 + 6"; }
    public void buttonClicked() {
        JOptionPane.showMessageDialog(null, "The result of "
            + application.getInput() + " is "
            + calculate(application.getInput()));
    }
}

class CalcStarter { public static void main(String[] args) {
    new Application(new CalcPlugin()).setVisible(true); }}
Example: An Eclipse plugin

- A popular Java IDE
- More generally, a framework for tools that facilitate “building, deploying and managing software across the lifecycle.”
- Plugin framework based on OSGI standard
- Starting point: Manifest file
  - Plugin name
  - Activator class
  - Meta-data

```manifest
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyEditor Plug-in
Bundle-SymbolicName: MyEditor;
  singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: myeditor.Activator
Require-Bundle: org.eclipse.ui,
  org.eclipse.core.runtime,
  org.eclipse.jface.text,
  org.eclipse.ui.editors
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```
Example: An Eclipse plugin

- **plugin.xml**
  - Main configuration file
  - XML format
  - Lists extension points

- **Editor extension**
  - extension point: `org.eclipse.ui.editors`
  - file extension
  - icon used in corner of editor
  - class name
  - unique id
    - refer to this editor
    - other plugins can extend with new menu items, etc.!

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension point="org.eclipse.ui.editors">
    <editor name="Sample XML Editor"
      extensions="xml"
      icon="icons/sample.gif"
      contributorClass="org.eclipse.ui.text
editor.BasicTextEditorActionContributor"
      class="myeditor.editors.XMLEditor"
      id="myeditor.editors.XMLEditor">
    </editor>
  </extension>
</plugin>
```
Example: An Eclipse plugin

• At last, code!

• XMLEditor.java
  ▪ Inherits TextEditor behavior
    • open, close, save, display, select, cut/copy/paste, search/replace, ...
    • REALLY NICE not to have to implement this
  ▪ But could have used ITextEditor interface if we wanted to
  ▪ Extends with syntax highlighting
    • XMLDocumentProvider partitions into tags and comments
    • XMLConfiguration shows how to color partitions

```java
package myeditor.editors;

import org.eclipse.ui.editors.text.TextEditor;

public class XMLEditor extends TextEditor {
    private ColorManager colorManager;

    public XMLEditor() {
        super();
        colorManager = new ColorManager();
        setSourceViewerConfiguration(new XMLConfiguration(colorManager));
        setDocumentProvider(new XMLDocumentProvider());
    }

    public void dispose() {
        colorManager.dispose();
        super.dispose();
    }
}
```
Example: A JUnit Plugin

```java
public class SampleTest {
    private List<String> emptyList;

    @Before
    public void setUp() {
        emptyList = new ArrayList<String>();
    }

    @After
    public void tearDown() {
        emptyList = null;
    }

    @Test
    public void testEmptyList() {
        assertEquals("Empty list should have 0 elements",
                    0, emptyList.size());
    }
}
```

Here the important plugin mechanism is Java annotations
Java Swing: It’s a library?

- Create a GUI using pre-defined containers
  - JFrame, JPanel, JDialog, JMenuBar
- Use a layout manager to organize components in the container
- Add pre-defined components to the layout
  - Components: JLabel, JTextField, JButton

This is no different than the File I/O library.
// create the container
JPanel panel = new JPanel();

// create the label, add to the container
JLabel label = new JLabel();
label.setText("Enter your userid:");
panel.add(label);

// create a text field, add to the container
JTextField textfield = new JTextField(16);
panel.add(textfield)
Swing: Layout managers

```java
panel.setLayout(new GridBagLayout());

GridBagConstraints c = new GridBagConstraints();

// create and position the button
JButton button = new JButton(“Click Me!”);

// default grid constraint
// add side padding
pane.add(button, c);
```
// create an anonymous MouseAdapter, which extends
// the MouseListener class

button.add(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        System.err.println("You clicked me! " +
                "Do it again!");
    }
});

But this extending a class to add custom behaviors, right?
Where is the boundary?

Container

EventListener

MouseAdapter

JComponent

JPanel

JButton

MyWidget

$1

AWT Framework

Swing Framework

Our Implementation
public MyWidget extends JPanel {

    public MyWidget(int param) {
        setLayout(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints();
        ...
        add(label, c);
        add(textfield, c);
        add(button, c);
    }

    public void setParameter(int param) {
        // update the widget, as needed
    }
}
public MyWidget extends JContainer {

  public MyWidget(int param) {
    // setup internals, without rendering
  }

  // render component on first view and resizing
  protected void paintComponent(Graphics g) {
    // draw a red box on this component
    Dimension d = getSize();
    g.setColor(Color.red);
    g.drawRect(0, 0, d.getWidth(), d.getHeight());
  }
}
Recall event-based programming

- A style of programming where the control-flow of the program is driven by (usually-) external events

```java
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(42)
}
```

```java
public void performAction(ActionEvent e) {
    bigBloatedPowerPointFunction(e);
    withANameSoLongIMadeItTwoMethods(e);
    yesIKnowJavaDoesntWorkLikeThat(e);
}
```

```java
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(40)
}
```
Platform/software ecosystem

- Hardware/software environment (frameworks, libraries) for building applications

- Ecosystem: Interaction of multiple parties on a platform, third-party contributions, co-dependencies, ...
  - Typically describes more business-related and social aspects
Learning a framework

- Documentation
- Tutorials, wizards, and examples
- Communities, email lists and forums
- Other client applications and plugins
Framework design exercises

Thanks for downloading the safest, fastest and most customizable version of Firefox yet. To start browsing, just close this tab as shown above.

Learn More
Wondering what to do now? Our Getting Started page has a list of recommended sites, plus more Firefox info.

Questions?
Our Support page has plenty of answers, plus a live chat feature to guide you through any tricky situations.

Customize?
Now that you’ve got Firefox, find out all the ways to personalize it to fit exactly how you use the Web.
Framework design exercises
Framework design exercises
Framework design exercises
Framework design exercises

• Think about a framework for:
  ▪ Video playing software
  ▪ Viewing, printing, editing a portable document format
  ▪ Compression and archiving software
  ▪ Instant messaging software
  ▪ Music editing software

• Questions
  ▪ What are the dimensions of variability/extensibility?
  ▪ What interfaces would you need?
  ▪ What are the core methods for each interface?
  ▪ How do you set up the framework?
Summary

• Reuse and variation essential
  ▪ Avoid reimplementing from scratch

• Object-oriented design principles for library design

• From low-level code reuse to design/behavior reuse with frameworks

• Design for reuse with domain analysis: find common and variable parts

• Use design patterns for framework design and implementation