

Principles of Software Construction: Objects, Design, and Concurrency

Libraries and Frameworks

(Design for large-scale reuse)

Claire Le Goues

Vincent Hellendoorn



Administrivia

Midterm is next Thursday, we will release sample questions this week.

HW5:

- Goal with coupling Game/rules
- How to pick God cards vs. default?
- You can change things post checkpoint.

Learning goals for today

- 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

Where Does That Get Us?

	<i>Small scale:</i> One/few objects	<i>Mid scale:</i> Many objects	<i>Large scale:</i> Subsystems
<i>Design for</i>	Subtype	Domain Analysis ✓	GUI vs Core ✓
understanding	Polymorphism ✓	Inheritance & Del. ✓	Frameworks and Libraries, APIs
change/ext.	Information Hiding, Contracts ✓	Responsibility Assignment, Design Patterns, Antipattern ✓	Module systems, microservices ✓
reuse	Immutability ✓	Promises/ Reactive P. ✓	(Testing for) Robustness
robustness	Types ✓	Integration Testing ✓	CI ✓, DevOps, Teams
...	Unit Testing ✓		

Earlier in this course: **Class-level** reuse

Language mechanisms supporting reuse

- Inheritance
- Subtype polymorphism (dynamic dispatch)
- Parametric polymorphism (generics)*

Design principles supporting reuse

- Small interfaces
- Information hiding
- Low coupling
- High cohesion

Design patterns supporting reuse

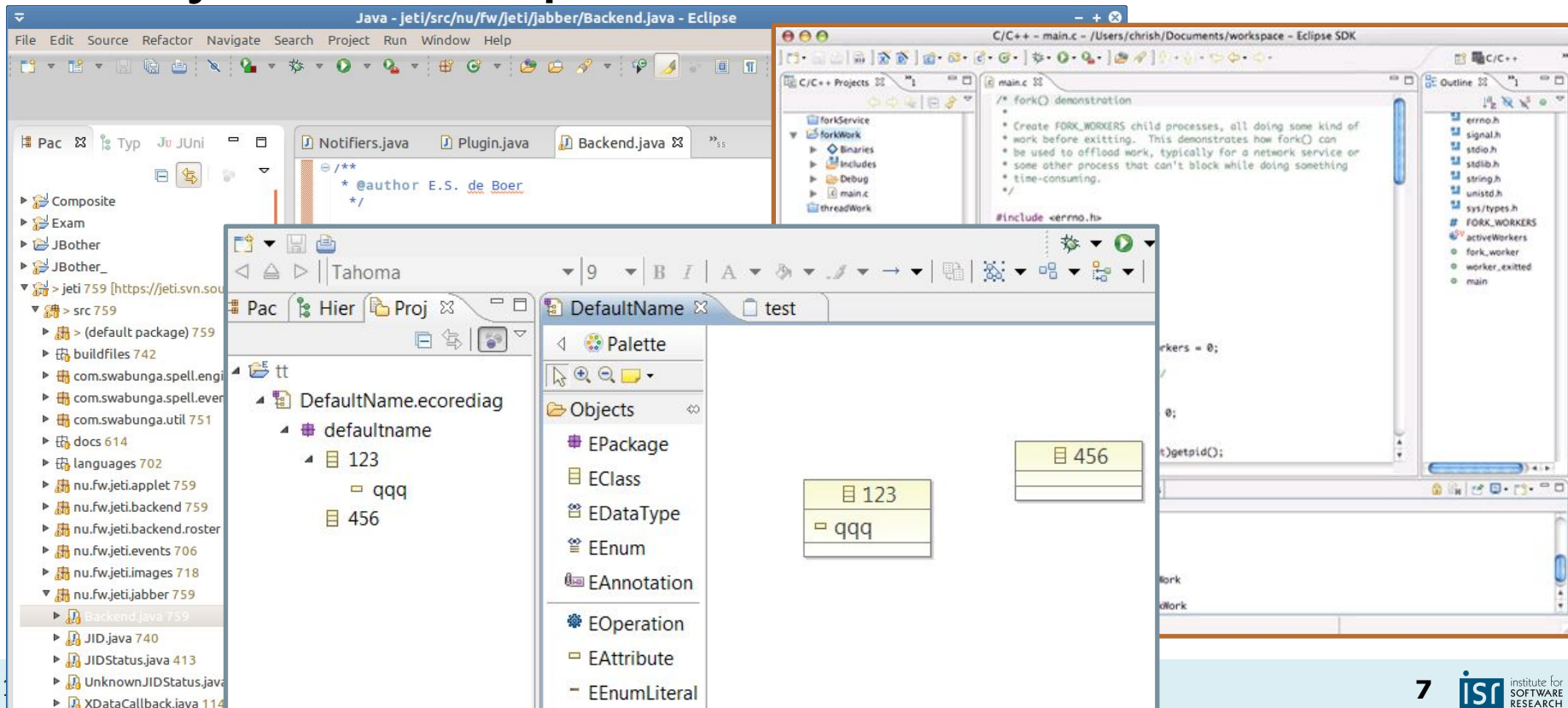
- Template method, decorator, strategy, composite, adapter, ...

* Effective Java items 26, 29, 30, and 31

Where Does That Get Us?

	<i>Small scale:</i> One/few objects	<i>Mid scale:</i> Many objects	<i>Large scale:</i> Subsystems
<i>Design for</i>	Subtype	Domain Analysis ✓	GUI vs Core ✓
understanding	Polymorphism ✓	Inheritance & Del. ✓	Frameworks and Libraries , APIs
change/ext.	Information Hiding, Contracts ✓	Responsibility Assignment,	Module systems, microservices ✓
reuse	Immutability ✓	Design Patterns, Antipattern ✓	(Testing for) Robustness
robustness	Types ✓	Promises/ Reactive P. ✓	CI ✓ , DevOps, Teams
...	Unit Testing ✓	Integration Testing ✓	

Reuse and variation: Family of development tools



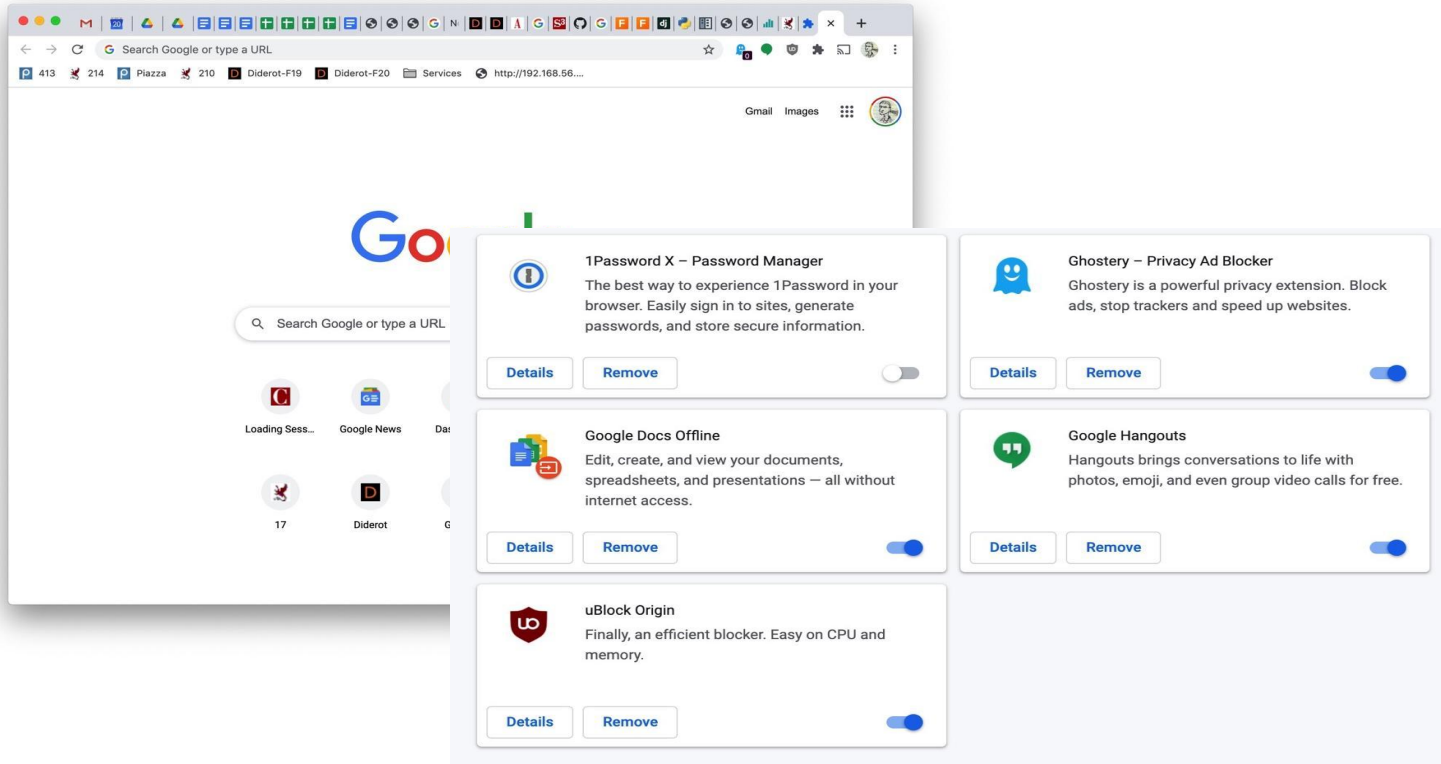
Reuse and variation: Eclipse Rich Client Platform

The screenshot displays the ForeFlight application window with the following components:

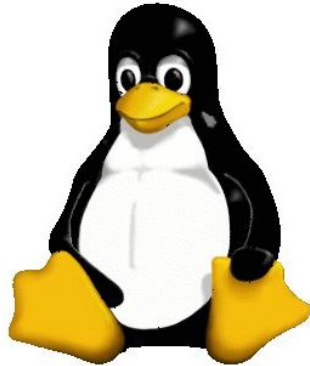
- Left Panel:** A tree view of airports categorized by state (TX, UT, VA, VI, VT, WA, WI). The WI section is expanded, listing airports like KAIG - Antigo, WI and KATW - Appleton, WI.
- Weather Details Panel:**
 - Airport:** DANE COUNTY REGIONAL-TRUAX FIELD
 - Observations/Forecasts:** Thurs Feb 16 9:53 AM EST
 - Alerts:** Winds are close to set limit of 16 kts; Visibility is below set limit of 3 SM; Minimum cloud layer height worse than set limit of 1000 feet.
 - Weather Conditions:** Visual representation of clouds with snowflakes, a ceiling indicator showing 8000 and 1200 feet, a red box labeled "LIFR" (Ceiling below 500 and/or Visibility below 1), and a thermometer showing 20°F.
 - Weather Report:**
 - Airport:** DANE COUNTY REGIONAL-TRUAX FIELD
 - ID:** KMSN
 - Status:** Wx Report download successful
 - Report Date:** Feb 16, 2006 9:53:00 AM (22 minutes ago)
 - Report Period:** Observed at Thurs Feb 16 9:53 AM EST
 - Wind Speed:** 15.0 kts
 - Wind Direction (mag):** 20°
 - Temperature:** 24.8°F (-4°C)
 - Dewpoint:** 21.2°F (-6°C)
 - Pressure:** 29.88 in. Hg
 - Visibility:** 0.25 sm
 - Report Type:**
 - Sky Conditions:** Broken clouds at 100 feet, Overcast at 1200 feet
 - Weather Conditions:** Heavy Snow, Moderate Blowing Snow

- Runways Panel:**
- KMSN Runways**
- Magnetic deviation:** 2E
- Elevation:** 887 ft
- Diagram:** A runway diagram showing Runway 03 and Runway 12.
- Wind (mag):** 15 kts from 20°
- X-wind:** 2 kts from the left for 03
- Predicted Active:** 03
- Width:** 150 feet
- Length:** 7200 feet
- Surface:** Good CONC
- Airport Links Panel:** A list of links including "KMSN on Google Maps", "KMSN AirNav.com Page", "KMSN Approaches", "KMSN PIREPS", "KMSN METAR and/or TAF", and "KMSN NOTAMS (PilotWeb)".
- Nearby Airports Panel:** A list of nearby airports including KDLL - Baraboo, WI - 29.72 NM, KEFT - Monroe, WI - 33.40 NM, and KJVL - Janesville, WI - 33.78 NM.

Reuse and variation: Web browser extensions



Reuse and variation: Flavors of Linux



```
Linux Kernel v2.6.18-53.1.14.el5.customxen Configuration

Network File System
Arrow keys navigate the menu. <Enter> selects submenu --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module <> module
v(+)-
[ ] Provide NFS client caching support (EXPERIMENTAL)
[*] Allow direct I/O on NFS files (EXPERIMENTAL)
<M> NFS server support
[*] Provide NFSv3 server support
[*] Provide server support for the NFSv3 ACL protocol extension
[*] Provide NFSv4 server support (EXPERIMENTAL)
--- Provide NFS server over TCP support
[*] Root file system on NFS
--- Secure RPC: Kerberos V mechanism (EXPERIMENTAL)
v(+)-
<Select> < Exit > < Help >
```



Reuse and variation: Product lines



Today: Reuse **at scale**

- Examples, terminology
- Whitebox and blackbox frameworks
- Design considerations
- Implementation details
 - Responsibility for running the framework
 - Loading plugins

Today: Reuse **at scale**

- **Examples, terminology**
- Whitebox and blackbox frameworks
- Design considerations
- Implementation details
 - Responsibility for running the framework
 - Loading plugins

Terminology: Library



- *Library*: A set of classes and methods that provide reusable functionality
- Client calls library; library executes and returns data
- Client controls
 - Program structure
 - Control flow

```
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 his component  
        Dimension d = getSize();  
        g.setColor(Color.red);  
        g.drawRect(0, 0, d.getWidth(), d.getHeight());  
    }  
}
```

your code



- E.g.: Math, Collections, I/O, command line parsing

Terminology: Frameworks



- **Framework**: Reusable skeleton code that can be customized into an application
- Framework calls back into client code
 - The Hollywood principle: “Don’t call us. We’ll call you.”
- Framework controls
 - Program structure
 - Control flow

```
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 his component  
        Dimension d = getSize();  
        g.setColor(Color.red);  
        g.drawRect(0, 0, d.getWidth(), d.getHeight());  
    }  
}
```

your code



Framework

- E.g.: VSCode, Firefox, IntelliJ, NanoHttpd, Express, Android, React?

A calculator example (without a framework)

```
public class Calc extends JFrame {
    private JTextField textField;
    public Calc() {
        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(/* calculation code */);
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle("My Great Calculator");
        ...
    }
}
```



A simple example framework

- Consider a family of programs consisting of a button and text field only:



- What source code might be shared?

A calculator example (without a framework)

```
public class Calc extends JFrame {
    private JTextField textField;
    public Calc() {
        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(/* calculation code */);
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle("My Great Calculator");
        ...
    }
}
```



A simple example framework

```
public abstract class Application extends JFrame {
    protected String getApplicationTitle() { return ""; }
    protected String getButtonText() { return ""; }
    protected String getInitialText() { return ""; }
    protected void buttonClicked() { }
    private JTextField textField;
    public Application() {
        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(getInitialText());
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        button.addActionListener((e) -> { buttonClicked(); });
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle(getApplicationTitle());
        ...
    }
}
```

Using the example framework

```
public abstract class Application extends JFrame {  
    protected String getApplicationTitle() { return ""; }  
    protected String getButtonText() { return ""; }  
    protected String getInitialText() { return ""; }  
}
```

```
public class Calculator extends Application {  
    protected String getApplicationTitle() { return "My Great Calculator"; }  
    protected String getButtonText() { return "calculate"; }  
    protected String getInitialText() { return "(10 - 3) * 6"; }  
    protected void buttonClicked() {  
        JOptionPane.showMessageDialog(this, "The result of " + getInput() +  
            " is " + calculate(getInput()));  
    }  
    private String calculate(String text) { ... }  
}
```

```
textField.setPreferredSize(new Dimension(200, 20));  
contentPane.add(textField, BorderLayout.WEST);  
button.addActionListener((e) -> { buttonClicked(); });  
this.setContentPane(contentPane);  
this.pack();
```

Using the example framework again

```
public abstract class Application extends JFrame {  
    protected String getApplicationTitle() { return ""; }  
    protected String getButtonText() { return ""; }  
    protected String getInitialText() { return ""; }
```

```
public class Calculator extends Application {  
    protected String getApplicationTitle() { return "My Great Calculator"; }  
    protected String getButtonText() { return "calculate"; }  
    protected String getInitialText() { return "(10 - 3) * 6"; }  
    protected void buttonClicked() {  
        JOptionPane.showMessageDialog(this, "The result of " + getInput() +  
            " is " + calculate(getInput()));  
    }  
    private String calculate(String text) { ... }  
}
```

```
public class Ping extends Application {  
    protected String getApplicationTitle() { return "Ping"; }  
    protected String getButtonText() { return "ping"; }  
    protected String getInitialText() { return "127.0.0.1"; }
```

General distinction: Library vs. framework



user
interacts

```
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 his  
        componentDimension d = getSize();  
        g.setColor(Color.red);  
        g.drawRect(0, 0, d.getWidth(),  
            d.getHeight());  
    }  
}
```

your code



user
interacts

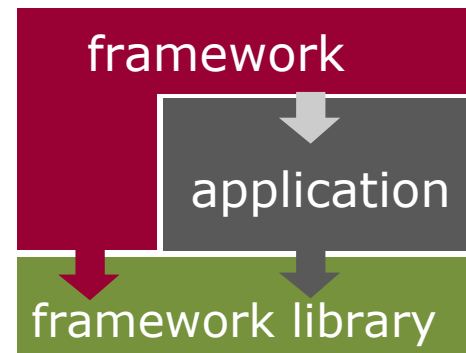
```
public MyWidget extends JContainer {  
    public MyWidget(int param) { /* setup  
        internals, without rendering  
    }  
  
    / render component on first view and  
    resizing  
    protected void  
    paintComponent(Graphics g) {  
        // draw a red box on his  
        componentDimension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

your code



Libraries and frameworks in practice

- Defines key abstractions and their interfaces
- Defines object interactions and invariants
- Defines flow of control
- Provides architectural guidance
- Provides defaults



credit: Erich Gamma

slido



**Join at [slido.com](https://www.slido.com)
#4118918**

① Start presenting to display the joining instructions on this slide.

Framework or library?

- IntelliJ / VSCode
- Java Collections / Node Streams

Framework or library?

- IntelliJ / VSCode
- Java Collections / Node Streams
- **Express/NanoHttpd**
- **Handlebars (used in HW4)**

slido



Express/NanoHttpd: Framework or Library? (include rationale, andrewID)

① Start presenting to display the poll results on this slide.

slido



Handlebars: Framework, or Library? (include rationale, andrewID)

① Start presenting to display the poll results on this slide.

Is Santorini a Framework?



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:** 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 that gets called in a sequence according to the protocol and the state of the plugin

Today: Libraries and frameworks for reuse

- Terminology and examples
- **Whitebox and blackbox frameworks**
- Designing a framework
- Implementation details

WHITE-BOX VS BLACK-BOX* FRAMEWORKS

* outdated terms, not aware of common replacements; maybe Inheritance-Based vs Delegation-Based Frameworks

Whitebox (inheritance-based) frameworks

- Extension via subclassing and overriding methods
- Common design pattern(s):
 - Template method
- Subclass has main method but gives control to framework

Blackbox (delegation-based) frameworks

- Extension via implementing a plugin interface
- Common design pattern(s):
 - Strategy
 - Command
 - Observer
- Plugin-loading mechanism loads plugins and gives control to the framework

Is this a whitebox or blackbox framework?

```
public abstract class Application extends JFrame {  
    protected String getApplicationTitle() { return ""; }  
    protected String getButtonText() { return ""; }  
    protected String getInitialText() { return ""; }  
}
```

```
public class Calculator extends Application {  
    protected String getApplicationTitle() { return "My Great Calculator"; }  
    protected String getButtonText() { return "calculate"; }  
    protected String getInitialText() { return "(10 - 3) * 6"; }  
    protected void buttonClicked() {  
        JOptionPane.showMessageDialog(this, "The result of " + getInput() +  
            " is " + calculate(getInput()));  
    }  
}
```

```
public class Ping extends Application {  
    protected String getApplicationTitle() { return "Ping"; }  
    protected String getButtonText() { return "ping"; }  
    protected String getInitialText() { return "127.0.0.1"; }  
    protected void buttonClicked() { ... }  
}
```

An example blackbox framework

```
public class Application extends JFrame {
    private JTextField textField;
    private Plugin plugin;
    public Application() { }
    protected void init(Plugin p) {
        p.setApplication(this);
        this.plugin = p;
        JPanel contentPane = new JPanel()
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText(plugin != null ? plugin.getButtonText() : "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((e) -> { plugin.buttonClicked(); } );
        this.setContentPane(contentPane);
    }
}
```

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

An example blackbox framework

```
public class Application extends JFrame {
    private JTextField textField;
    private Plugin plugin;
    public Application() { }
    protected void init(Plugin p) {
        p.setApplication(this);
        this.plugin = p;
    }
}
```

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

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

An aside: Plugins could be reusable too...

```
public class Application extends JFrame implements InputProvider {
```

```
    private JTextField textField;  
    private Plugin plugin;  
    public Application() { }  
    protected void init(Plugin p) {  
        p.setApplication(this);  
        this.plugin = p;
```

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

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

```
    public String getApplicationTitle() { return "My Great Calculator"; }
```

```
public interface InputProvider {  
    String getInput();  
}
```

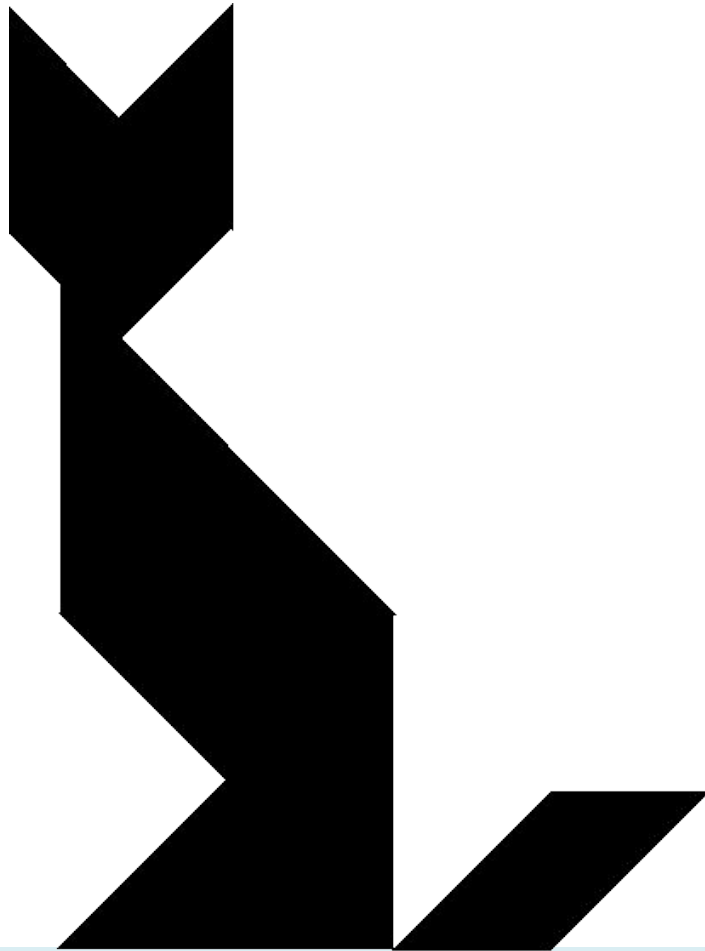
Frameworks summary

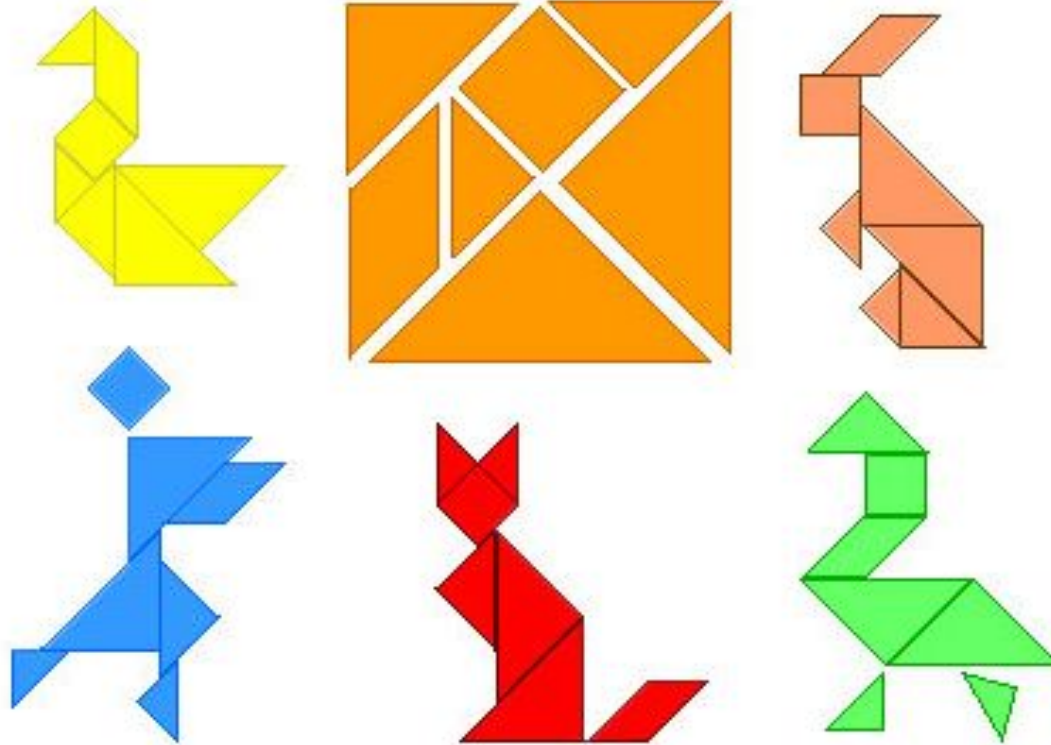
- Whitebox frameworks use subclassing
 - Allows extension of every nonprivate method
 - Need to understand implementation of superclass
 - Only one extension at a time
 - Compiled together
 - Often so-called developer frameworks
- Blackbox frameworks use composition
 - Allows extension of functionality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

Framework design considerations

- Once designed there is little opportunity for change
- Key decision: Separating common parts from variable parts
 - What problems do you want to solve?
- Possible problems:
 - Too few extension points: Limited to a narrow class of users
 - Too many extension points: Hard to learn, slow to extend
 - Too generic: Little reuse value

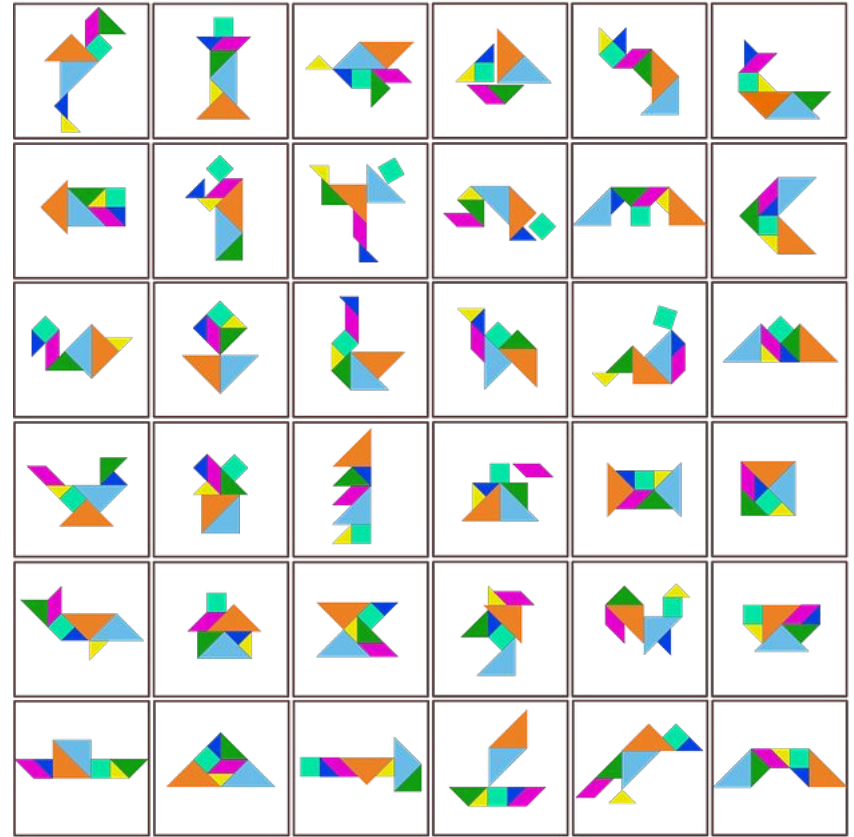
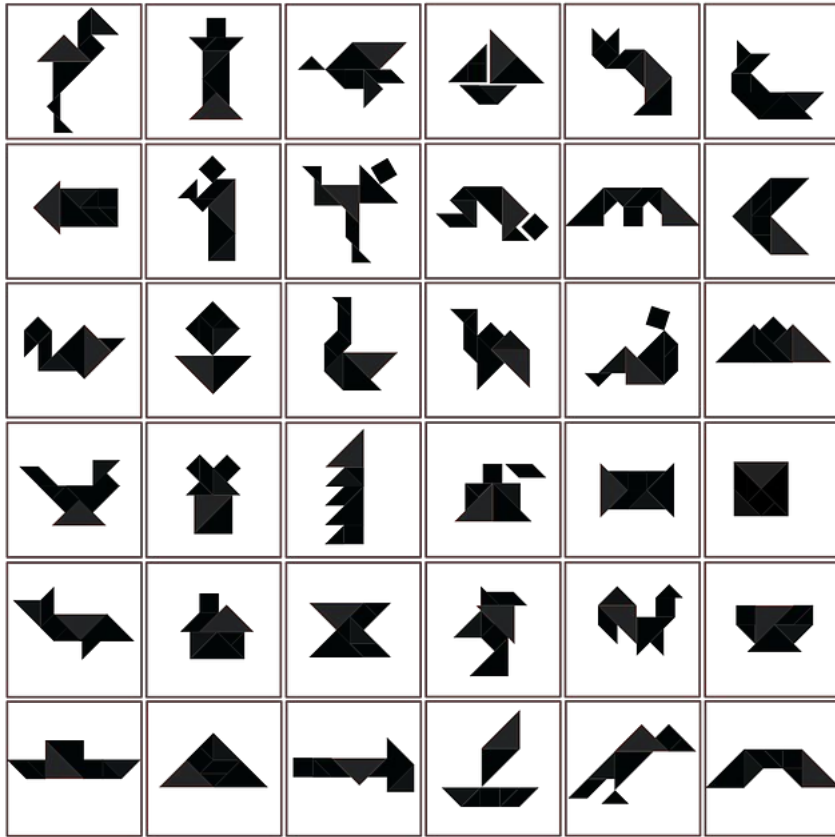
USE VS REUSE: DOMAIN ENGINEERING





(one modularization: tangrams)

Tangrams



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

Domain engineering

- Understand users/customers in your domain: What might they need? What extensions are likely?
- Collect example applications before designing a framework
- Make a conscious decision what to support (*scoping*)
- e.g., the Eclipse policy:
 - Plugin interfaces are internal at first
 - Unsupported, may change
 - Public stable extension points created when there are at least two distinct customers

The cost of changing a framework

```
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("Calculate");
        contentPane.add(button);
        textfield = new JTextField(20);
        if (plugin != null)
            textfield.setText(plugin.getInititalText());
        textfield.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null, "The result of " +
                    plugin.getApplicationTitle() + " is " +
                    plugin.getButtonText() + ": " +
                    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() {
        application.buttonClicked();
    }
}

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

this.setCon }
```


The cost of changing a framework

```
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("Calculate");
        contentPane.add(button);
        textfield = new JTextField(20);
        if (plugin != null)
            textfield.setText(plugin.getInititalText());
        textfield.setText("");
    }
}
```

Consider adding an extra method.
Requires changes to *all* plugins!

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

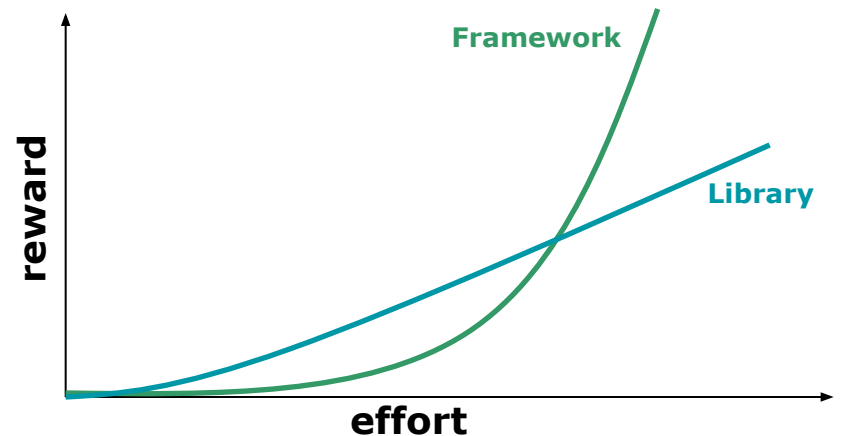
```
public class CalcPlugin implements Plugin {
    private Application application;
    public void setApplication(Application app) { this.application = app; }
    public String getButtonText() { return "calculate"; }
    public String getInititalText() { return "10 / 2 + 6"; }
    public void buttonClicked()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

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

```
return "My Great Calculator"; }
```

Learning a framework

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



Typical framework design and implementation

Define your domain

- Identify potential common parts and variable parts

Design and write sample plugins/applications

Factor out & implement common parts as framework

Provide plugin interface & callback mechanisms for variable parts

- Use well-known design principles and patterns where appropriate...

Get lots of feedback, and iterate

FRAMEWORK MECHANICS

Running a framework

- Some frameworks are runnable by themselves
 - e.g. Eclipse, VSCode, IntelliJ
- Other frameworks must be extended to be run
 - MapReduce, Swing, JUnit, NanoHttpd, Express

Methods to load plugins

1. Client writes main function, creates a plugin object, and passes it to framework
(see blackbox example above)
2. Framework has main function, client passes name of plugin as a command line argument or environment variable
(see next slide)
3. Framework looks in a magic location
Config files or .jar/.js files in a plugins/ directory are automatically loaded and processed
4. GUI for plugin management
E.g., web browser extensions

An example plugin loader using Java Reflection

```
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);
        }
    }
}
```

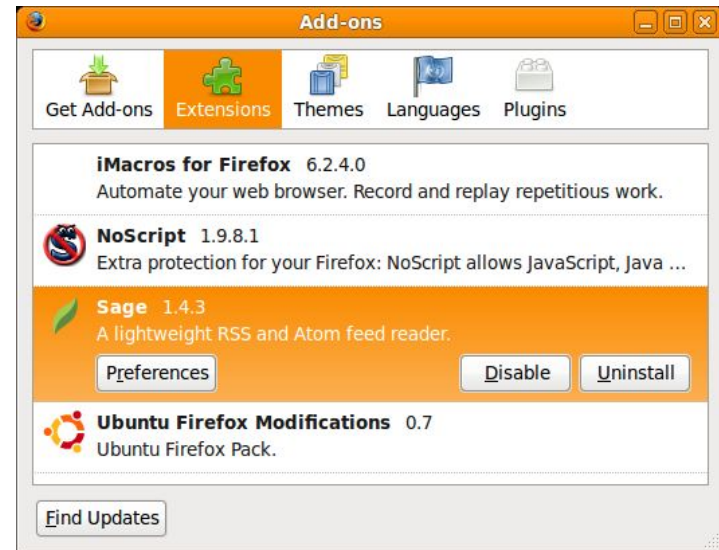
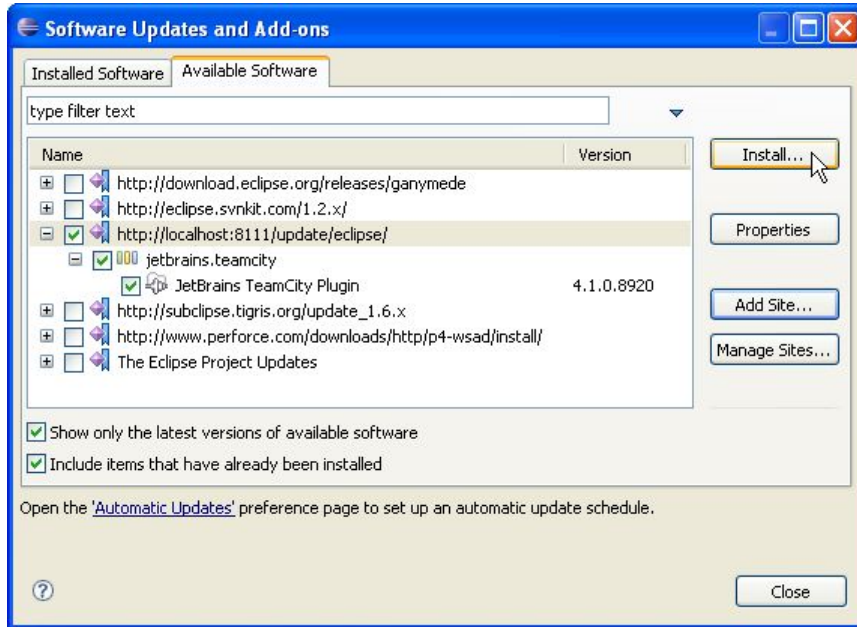
An example plugin loader in Node.js

```
const args = process.argv
if (args.length < 3)
  console.log("Plugin name not specified");
else {
  const plugin = require("plugins/"+args[2]+".js")()
  startApplication(plugin)
}
```


Another plugin loader using Java Reflection

```
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(line);
            application.addPlugin((Plugin) pluginClass.newInstance());
        } catch (Exception e) {
            System.out.println("Cannot load plugin " + line
                + ", 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

```
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
```

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 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.texteditor.BasicText
      EditorActionContributor"
      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

```
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

```
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

Summary

- Reuse and variation essential
 - Libraries and frameworks
- Whitebox frameworks vs. blackbox frameworks
- Design for reuse with domain analysis
 - Find common and variable parts
 - Write client applications to find common parts