Principles of Software Construction: Objects, Design, and Concurrency

Sub-system reuse

Libraries and frameworks

Josh Bloch     Charlie Garrod
Administrivia

- Homework 4c due Thursday night
- Midterm exam next Thursday (November 3rd)
  - Review session Wednesday, November 2nd, 7-9 pm, HH B103
Key concepts from last Thursday...
Get feedback early, often

<table>
<thead>
<tr>
<th>API</th>
<th>vote</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>yes</td>
<td>But remove binarySearch* and toList</td>
</tr>
<tr>
<td>BasicCollection</td>
<td>no</td>
<td>I don't expect lots of collection classes</td>
</tr>
<tr>
<td>BasicList</td>
<td>no</td>
<td>see List below</td>
</tr>
<tr>
<td>Collection</td>
<td>yes</td>
<td>But cut toArray</td>
</tr>
<tr>
<td>Comparator</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>DoublyLinkedList</td>
<td>no</td>
<td>(without generics this isn't worth it)</td>
</tr>
<tr>
<td>HashSet</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>LinkedList</td>
<td>no</td>
<td>(without generics this isn't worth it)</td>
</tr>
<tr>
<td>List</td>
<td>no</td>
<td>I'd like to say yes, but it's just way bigger than I was expecting</td>
</tr>
<tr>
<td>RemovalEnumeration</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td>yes</td>
<td>BUT IT NEEDS A DIFFERENT NAME</td>
</tr>
<tr>
<td>TreeSet</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

I'm generally not keen on the toArray methods because they add complexity.

Similarly, I don't think that the table Entry subclass or the various views mechanisms carry their weight.
Java Collections design conclusion

- It takes a lot of work to make something that appears obvious
  - Coherent, unified vision
  - Willingness to listen to others
  - Flexibility to accept change
  - Tenacity to resist change
  - Good documentation!

- It’s worth the effort!
  - A solid foundation can last two+ decades
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
Today: Libraries and frameworks for reuse
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:
Product lines
Earlier in this course: Class-level reuse

• Language mechanisms supporting reuse
  – Inheritance
  – Subtype polymorphism (dynamic dispatch) for delegation
  – 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, ...
Today: Libraries and frameworks for reuse

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

- **Library**: A set of classes and methods that provide reusable functionality
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.”

```java
public MyWidget extends JComponent {
    // setup internals, without rendering
    // render component on first view and resizing
    protected void paintComponent(Graphics g) {
        // draw a red box on the component
        Dimension d = getSize();
        g.setColor(Color.red);
        g.drawRect(0, 0, d.getWidth(), d.getHeight());
    }
}
```
A calculator example (without a framework)

```java
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(""整天);
        textField.setText("10 / 2 + 6");
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        button.addActionListener(/* calculation code */);
        this.getContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle("My Great Calculator");
        ...
    }
}
```
A simple example framework

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

• What source code might be shared?
A calculator example (without a framework)

```java
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

```java
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

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

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) { ... }
}

button.addActionListener((e) -> { buttonClicked(); });
this.setContentPane(contentPane);
this.pack();
this.setLocation(100, 100);
this.setTitle(getApplicationTitle());
... 
```
Using the example framework again

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

public class Calculator extends Application {
    protected String getApplicationTitle() { return "My Great Calculator"; }
    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()));
    }
    private String calculate(String text) { ... }
}

public class Ping extends Application {
    protected String getApplicationTitle() { return "Ping"; }
    protected String getButtonText() { return "ping"; }
    protected String getInititalText() { return "127.0.0.1"; }
    protected void buttonClicked() {
    }
}
```
General distinction: Library vs. framework

Your code

User interacts

Library

Your code

Framework
Libraries and frameworks in practice

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

credit: Erich Gamma
Framework or library?

- Java Collections
- Eclipse
- The Java Logging Framework
- Java Encryption Services
- Wordpress
- Ruby on Rails
A Scrabble framework?

• In what way is Homework 4 (Scrabble with Stuff) 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
WHITE-BOX VS BLACK-BOX FRAMEWORKS
Whitebox frameworks

• Extension via subclassing and overriding methods
• Common design pattern(s):
  – Template Method
• Subclass has main method but gives control to framework
Blackbox frameworks

• Extension via implementing a plugin interface

• Common design pattern(s):
  – Strategy
  – Observer

• 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 String getApplicationTitle() { return ""; }
    protected String getButtonText() { return ""; }
    protected String getInitialText() { return ""; }
    protected void buttonClicked() { }
}

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"; }
    protected void buttonClicked() { ... }
}
```
An example blackbox framework

class Application extends JFrame {
    private JTextField textField;
    private Plugin plugin;
    Application() {
    }
    protected void init(Plugin p) {
        p.setApplication(this);
        this.plugin = p;
        JPanel contentPane = new JPanel(new BorderLayout());
        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.getInitialText());
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        if (plugin != null)
            button.addActionListener((e) -> { plugin.buttonClicked(); });
        this.setContentPane(contentPane);
    ...
    }
    public String getInput() { return textField.getText(); }
}
An example blackbox framework

```java
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(new BorderLayout());
        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.getInitialText());
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        if (plugin != null) button.addActionListener((e) -> plugin.buttonClicked());
        this.setContentPane(contentPane);
        ...
    }
    public String getInput() { return textField.getText(); }
}

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...

```java
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;
        JPanel contentPane = new JPanel(new BorderLayout());
        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(""ISA");
        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 String getInput() { return textField.getText(); }
}
```

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

```java
public class CalcPlugin implements Plugin {
    private InputProvider app;
    public void setApplication(InputProvider app) { this.app = 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"; }
}
```

```java
public interface InputProvider {
    String getInput();
}
```
Whitebox vs. blackbox framework 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
  - Too generic: Little reuse value
USE VS REUSE:
DOMAIN ENGINEERING
(one modularization: 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
  – 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

• 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
Evolutionary design: Extract interfaces from classes

- 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

(credit: Erich Gamma)
FRAMEWORK MECHANICS
Running a framework

• Some frameworks are runnable by themselves
  – e.g. Eclipse
• Other frameworks must be extended to be run
  – Swing, JUnit, MapReduce, Servlets
Methods to load plugins

- Client writes `main()`, creates a plugin and passes it to framework
- Framework writes `main()`, client passes name of plugin as a command line argument or environment variable
- Framework looks in a magic location
  - Config files or .jar files are automatically loaded and processed
- GUI for plugin management
Supporting multiple plugins

- Observer design pattern is commonly used
- 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 p : plugins)
            p.setApplication(this);
    }
    public Message processMsg(Message msg) {
        for (Plugin p : plugins)
            msg = p.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
<?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.textor.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

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());
    }
}
Learning a framework

• Documentation
• Tutorials, wizards, and examples
• Other client applications and plugins
• Communities, email lists and forums
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
• Revise, revise, revise...