Principles of Software Construction: Objects, Design, and Concurrency

Libraries and Frameworks
(Design for large-scale reuse)

Christian Kästner  Vincent Hellendoorn
Michael Hilton
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
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)
- 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: Reuse at scale

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

    public void paintComponent(Graphics g) {
        // draw a red box on the component
        g.setColor(Color.red);
        g.drawRect(0, 0, d.getWidth(), d.getHeight());
    }
}
```

Frameworks:
- Swing
- IntelliJ
- Firefox
- Express
- NanoHttpd
- Spring
- Express
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.getContentPane(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?
```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();
    }
}
```
Using the example framework

```java
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.getContentPane(contentPane);
this.pack();
```
Using the example framework again

```java
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"; }
    protected void buttonClicked() { ... }
}
```
General distinction: Library vs. 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?

- IntelliJ / VSCode
- Java Collections / Node Streams
Framework or library?

- IntelliJ / VSCode
- Java Collections / Node Streams
- Command line parser
- Express/NanoHttpd
- Handlebars (the template library used in HW4)

- On a piece of paper:
  1. Describe the software (<= one sentence)
  2. Describe one way the software is like a library.
  3. Describe one way the software is like a framework.
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.
WHITE-BOX VS BLACK-BOX*
FRAMEWORKS

* old 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?

```java
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()));
    }
}

class Calculator {
    ...
}

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() { ... }
}
```
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 String getInput() {
        return textField.getText();
    }
}

An example blackbox framework

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

public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    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 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";
    }
}
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("");
        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();
    }
}

An aside: Plugins could be reusable too…

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) {
        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";
    }
}

public interface InputProvider {
    String getInput();
}
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 (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("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);
        button.addActionListener(/* … 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(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.getText() + " is " + application.calculate());
    }
    public String getApplicationTitle() {
        return "My Great Calculator";
    }
}
```

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

Consider adding an extra method. Many changes require changes to all plugins.
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

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

Framework has main function, 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/.js files in a plugins/ directory 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);
        }
    }
}
```
An example plugin loader in Node.js

```javascript
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

```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<? extends Plugin> pluginClass = Class.forName(pluginName);
            application.addPlugin(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
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.texteditor.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.
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