Principles of Software Construction: Objects, Design, and Concurrency

Design for Large-Scale Reuse: Libraries and Frameworks (Part 2)

Christian Kästner Bogdan Vasilescu

School of Computer Science
institute for SOFTWARE RESEARCH

```java
public class Board {

    private static Sqaure[][] board;
    private static LetterTile[][] tileBoard;
    private static SpecialTile[][] specialTileBoard;
    private final int scale =15;
    private final int four =4;
    private final int three =3;
    private final int eleven =11;
    private final int fourteen =14;
    private final int six =6;
    private final int eight =8;
```

Quiz

- What is the key difference between a framework and a library?
- What is the key difference between a whitebox framework and a blackbox framework?
- Name the design pattern(s) most prominent in whitebox frameworks.
- Name the design pattern(s) most prominent in blackbox frameworks.

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

Intro to Java

Git, CI

UML

GUIs

Static Analysis

Performance

GUIs

More Git

Design

Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Design Patterns, Unit Testing	Understanding the Problem	Concurrency Primitives, Synchronization
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies	Designing Abstractions for Concurrency
	Testing Subsystems	Distributed Systems in a Nutshell
	Design for Reuse at Scale: Frameworks and APIs	

General distinction: Library vs. framework

```
public MyWidget extends JContainer {

ublic 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

Library

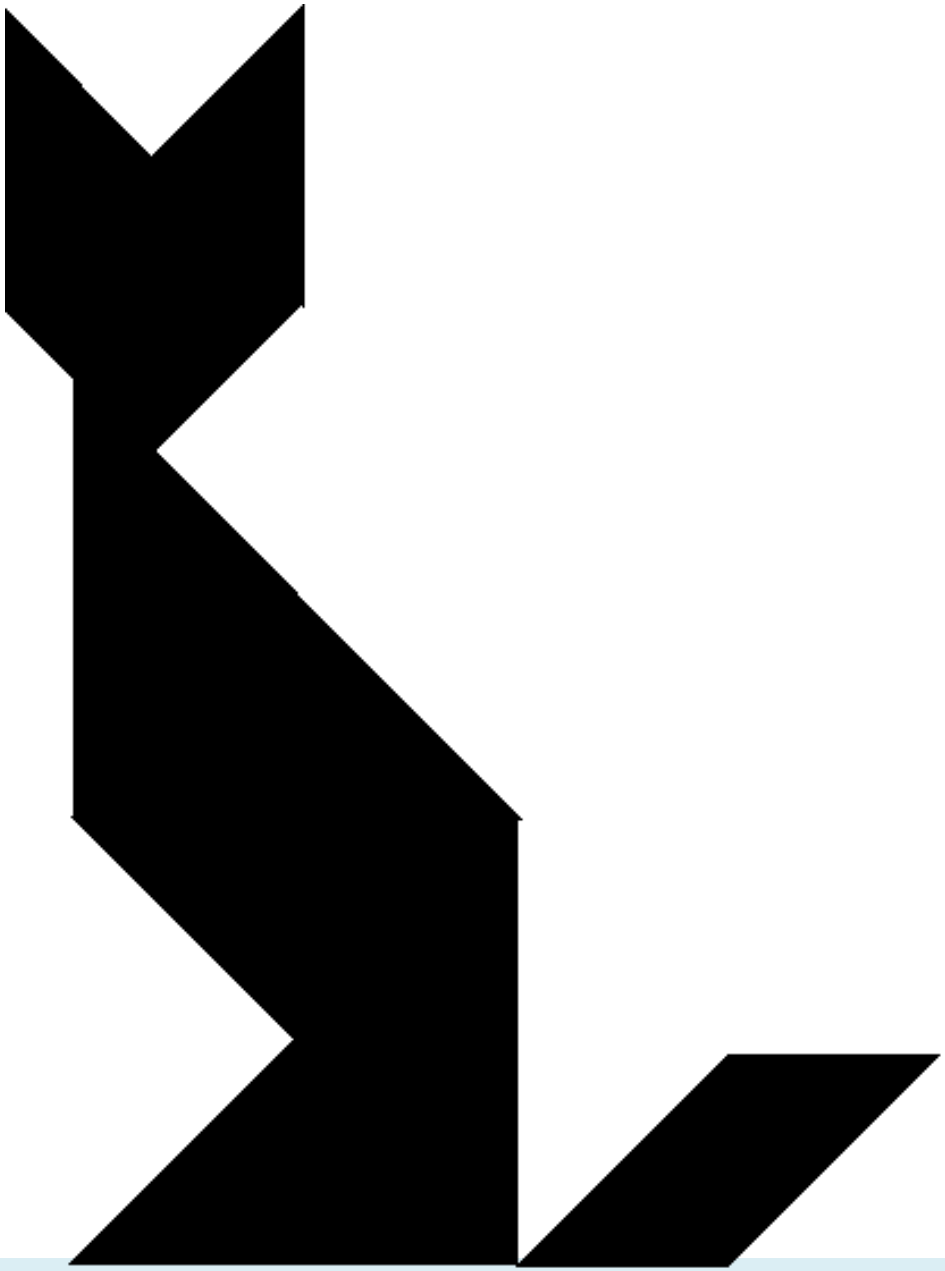
user interacts

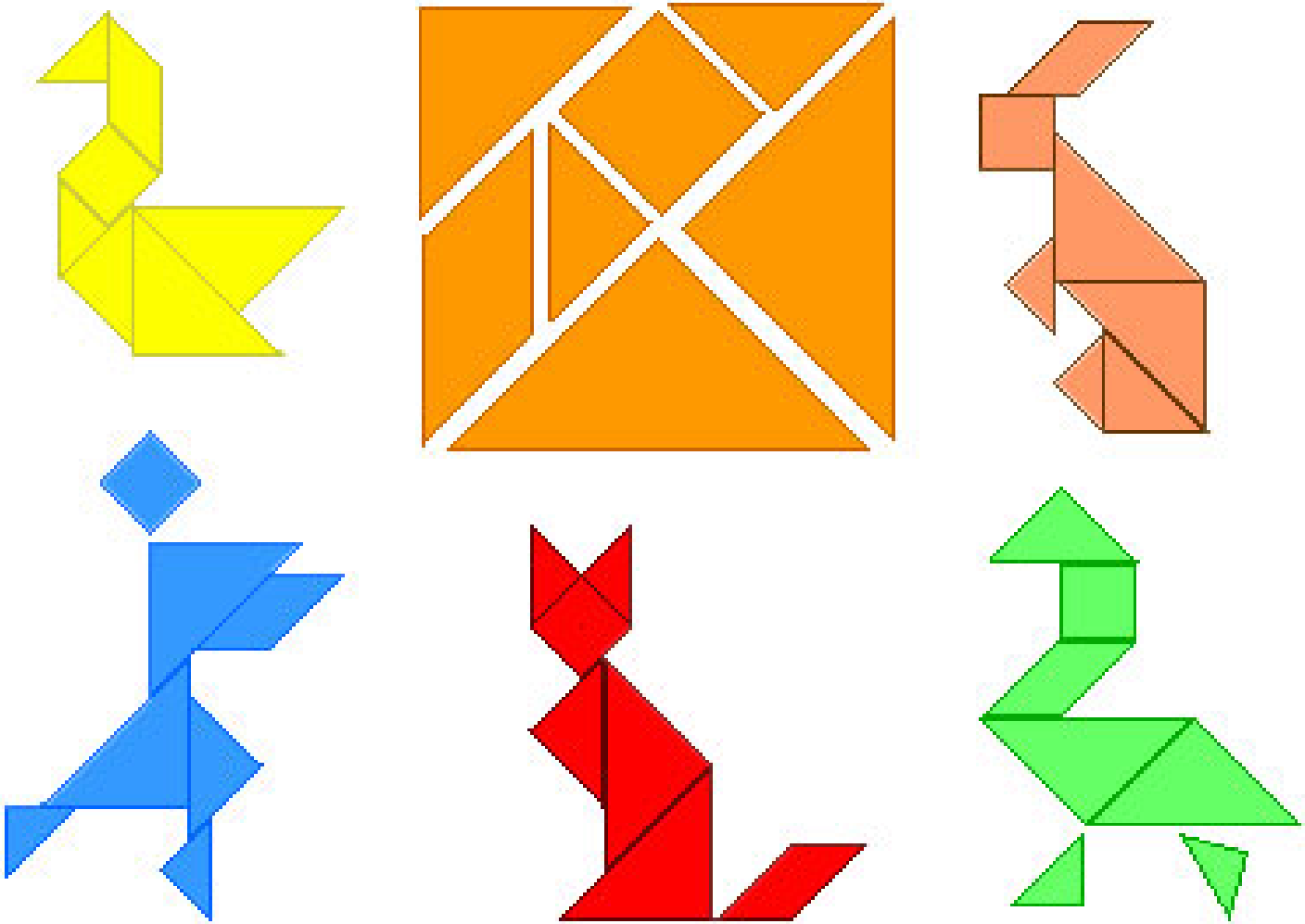
Framework

```
public MyWidget extends JContainer {

ublic 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

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 starting a framework/component
- 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

- 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 …
- 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
  - 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

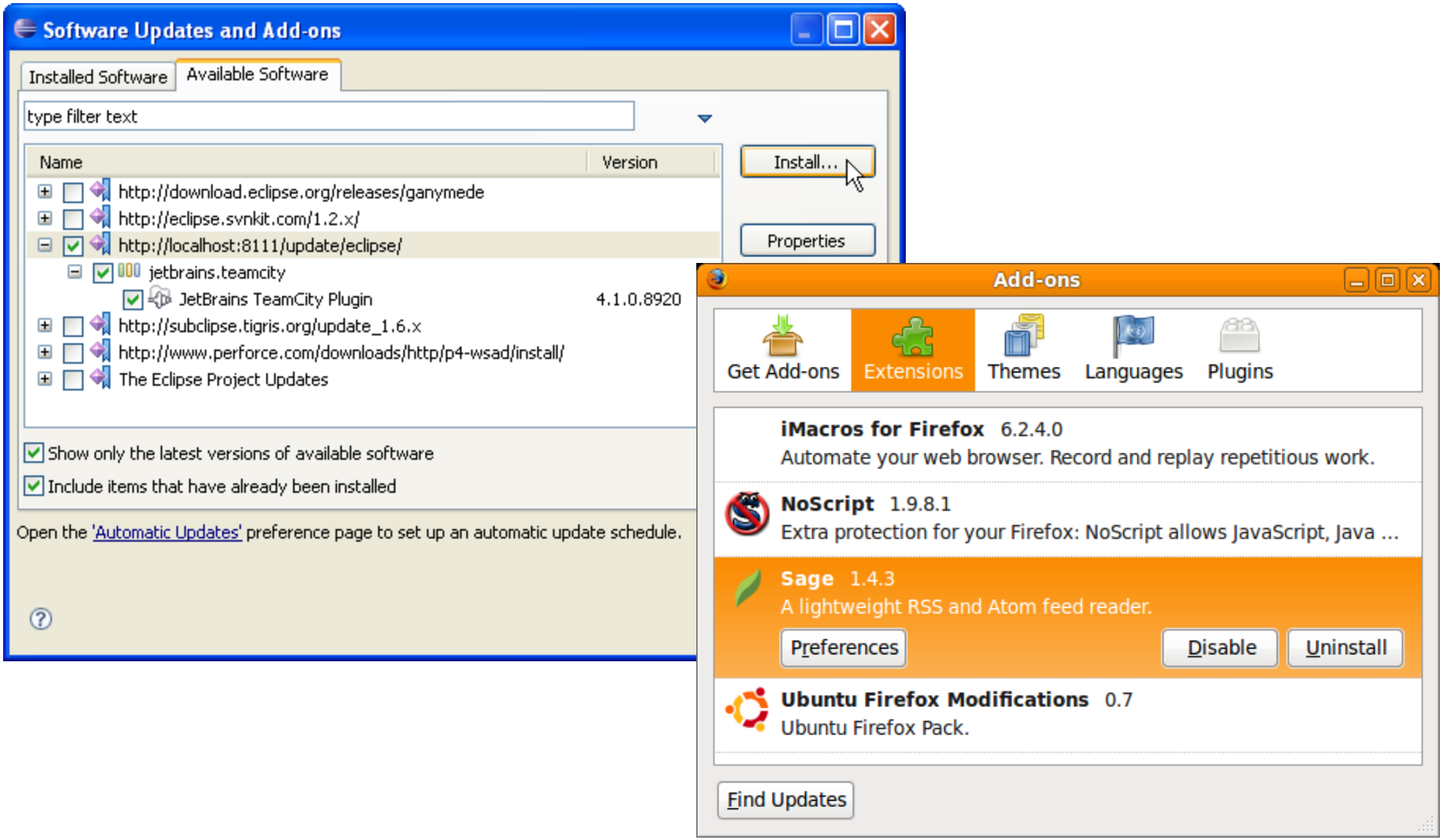
An example plugin loader using Java Reflection

```
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

```
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

```
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.Basic
TextEditorActionContributor"
        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

FRAMEWORK COSTS

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

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(BevelBorde
                JButton button = new JButton();
                if (plugin != null)
                        button.setText(plugin.getButtonText());
                else
                        button.setText("ok");
                contentPane.add(button, BorderLayout.EAST);
                textfield
                if (plugin

                textfield.
                contentP
                if (plugin
```

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

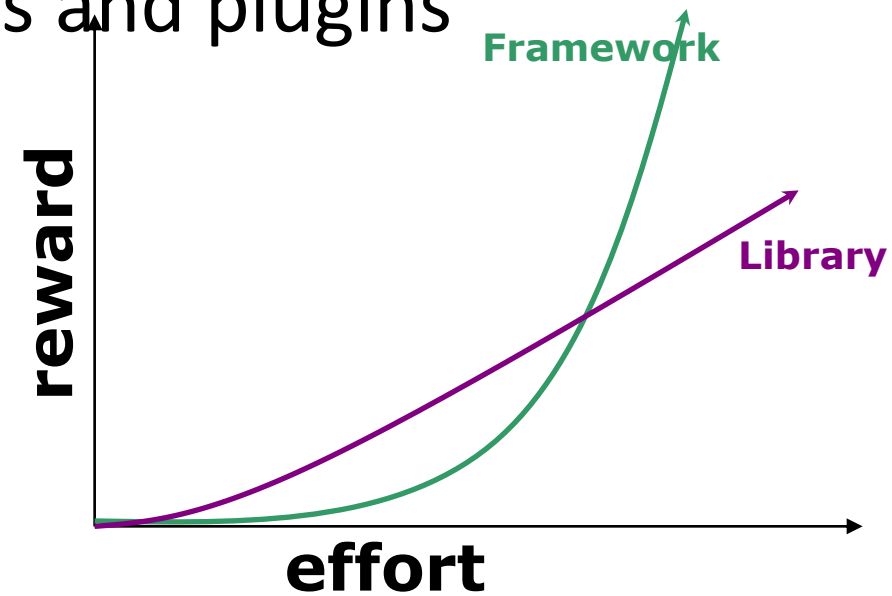
```
public class CalcPlugin implements Plugin {
        private Application application;
        public void setApplication(Application app) { this.application = app; }
        public String getButtonText() { return "calculate"; }
        public String getInititalText() { return "10 / 2 + 6"; }
    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

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

```
        new Application(new CalcPlugin()).setVisible(true); }}
```

Learning a framework

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



DESIGN CHALLENGES

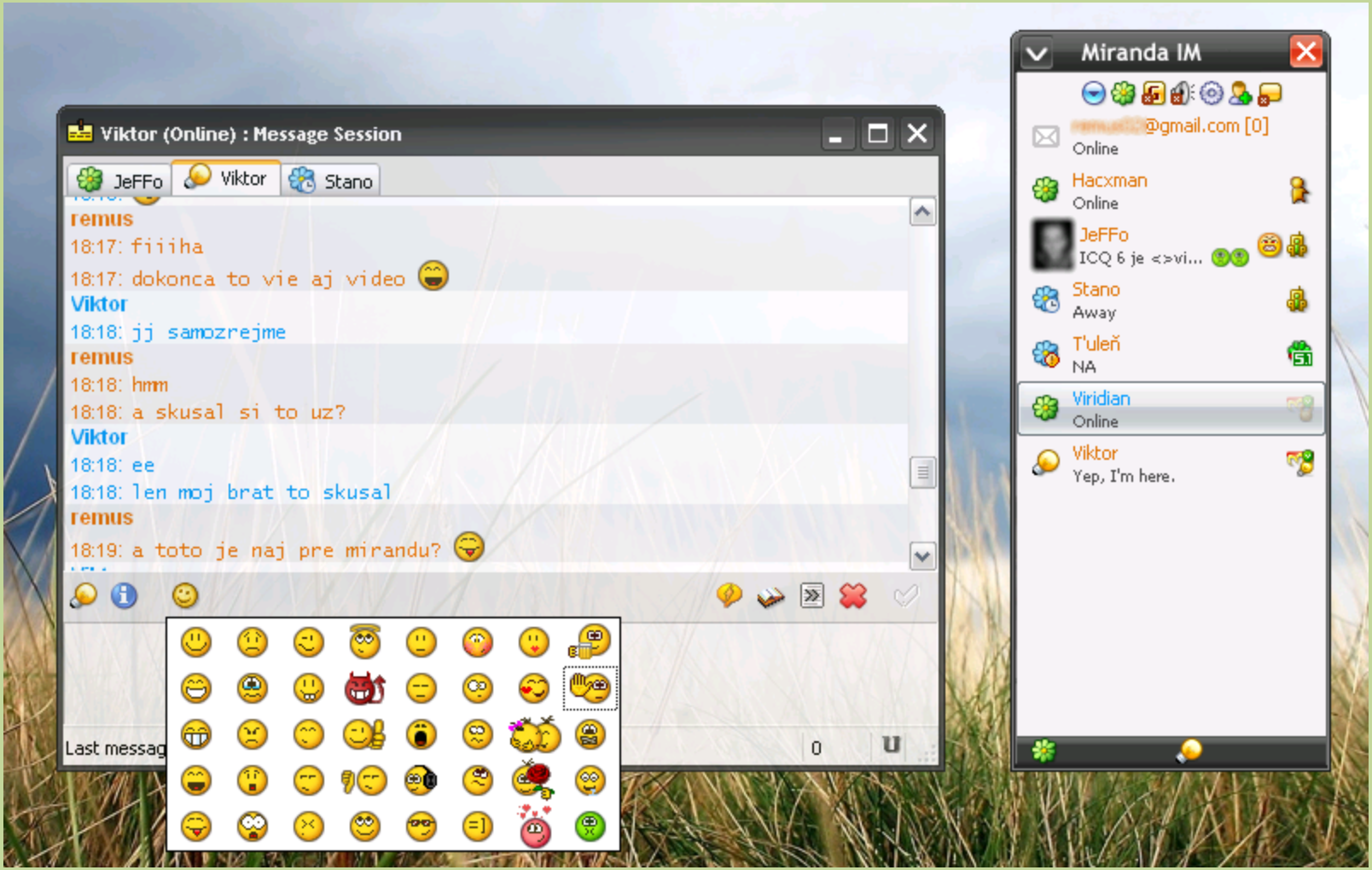
Framework design exercises

- 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?

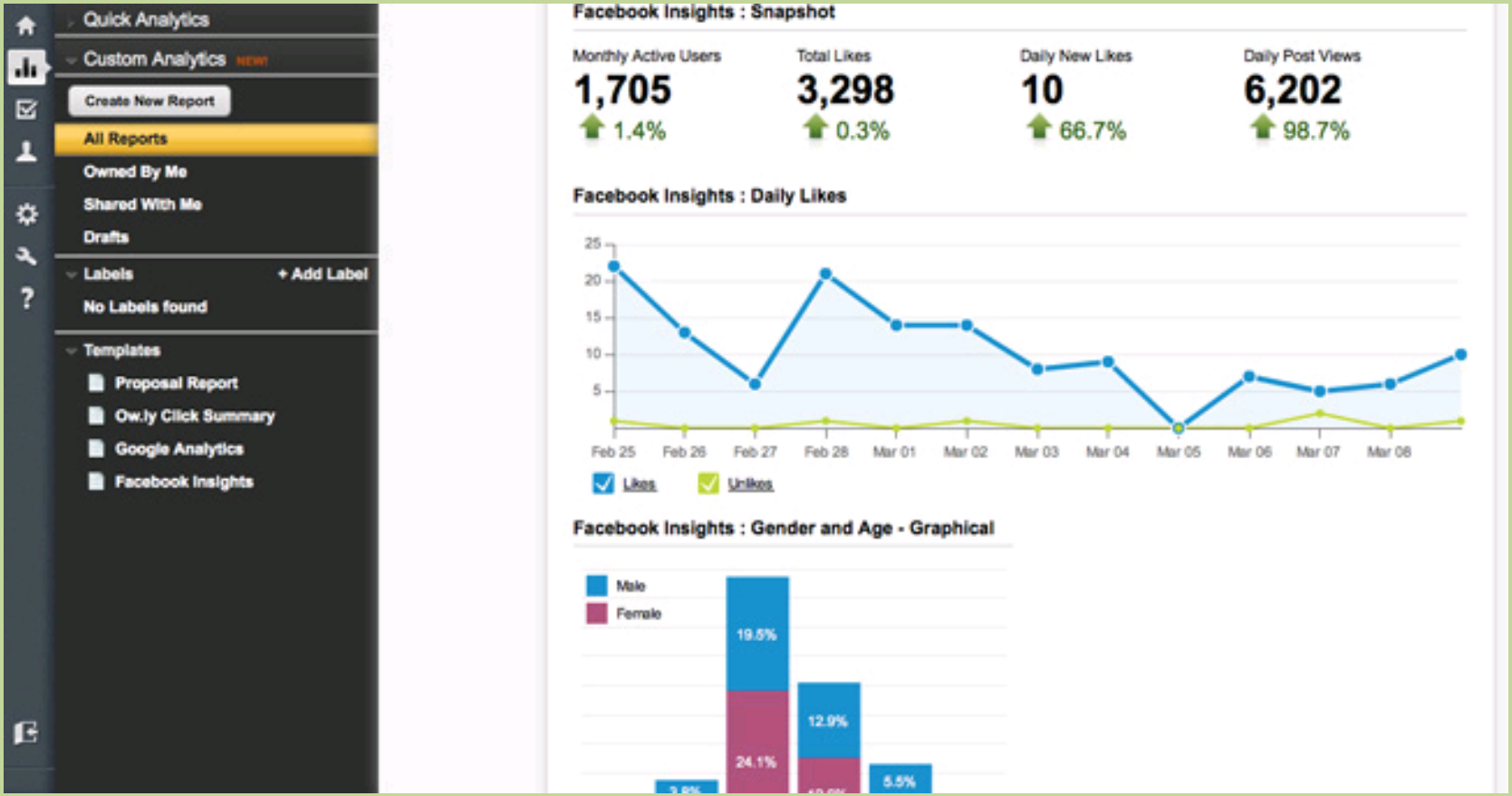
Framework design exercises

Framework design exercises



Framework design exercises

Framework design exercises



REFLECTION

What is reflection?

- Operating programmatically on objects that represent linguistic entities (e.g., classes, methods)
- Allows program to work with classes that were not know (or didn't exist!) at compile time
- Quite complex – involves many APIs
- But there's a simple form
  - Involves `Class.forName` and `newInstance`
  - `ServiceLoader` for plugins

Benchmark interface

```
/** Implementations can be timed by RunBenchmark. */
public interface Benchmark {
    /**
     * Initialize the benchmark.  Passed all command line
     * arguments beyond first three.  Used to parameterize a
     * a benchmark This method will be invoked once by
     * RunBenchmark prior to timings.
     */
    void init(String[] args);

    /**
     * Performs the test being timed.
     * @param numReps the number of repetitions comprising test
     */
    void run(int numReps);
}
```

RunBenchmark program (1)

```
public class RunBenchmark {
    public static void main(String[] args) throws Exception {
        if (args.length < 3) {
            System.out.println(
"Usage: java RunBenchmark <# tests> <# reps/test> <class name> [<arg>...]");
            System.exit(1);
        }

        int numTests = Integer.parseInt(args[0]);
        int numReps = Integer.parseInt(args[1]);
        Benchmark b =
               (Benchmark) Class.forName(args[2]).newInstance();
        String[] initArgs = new String[args.length - 3];
        System.arraycopy(args, 3, initArgs, 0, initArgs.length);
```

RunBenchmark program (2)

```
        if (initArgs.length != 0)
          System.out.println("Args: " + Arrays.toString(initArgs));
        b.init(initArgs);

        for (int i = 0; i < numTests; i++) {
            long startTime = System.nanoTime();
            b.run(numReps);
            long endTime = System.nanoTime();
            System.out.printf("Run %d: %d ms.%n", i,
                Math.round((endTime - startTime)/1_000_000.));
        }
    }
}
```

Sample Benchmark

```
public class SortBench implements Benchmark {
    private int[] a;

    public void init(String[] args) {
        int arrayLen = Integer.parseInt(args[0]);
        a = new int[arrayLen];
        Random rnd = new Random(666);
        for (int i = 0; i < arrayLen; i++)
            a[i] = rnd.nextInt(arrayLen);
    }
    public void run(int numReps) {
        for (int i = 0; i < numReps; i++) {
            int[] tmp = a.clone();
            Arrays.sort(tmp);
        }
    }
}
```

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

Further Reading

- For details on the framework example see Apel, Sven, et al. Feature-Oriented Software Product Lines. Berlin: Springer, 2013, Chapter 4.3
- Johnson, Ralph E., and Brian Foote. "Designing reusable classes." Journal of object-oriented programming 1.2 (1988): 22-35.