Principles of Software Construction: Objects, Design, and Concurrency

Part 2: Designing (sub-) systems

Design for large-scale reuse: Libraries and frameworks (part 2)

Charlie Garrod   Bogdan Vasilescu

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 4b due tonight(!)
- Midsemester grades summary in your GitHub repo
- Next required reading due Tuesday after spring break(!)
  - Effective Java, Items 51, 60, 62, and 64

https://commons.wikimedia.org/wiki/File:1_carcassonne_aerial_2016.jpg

Key concepts from Tuesday

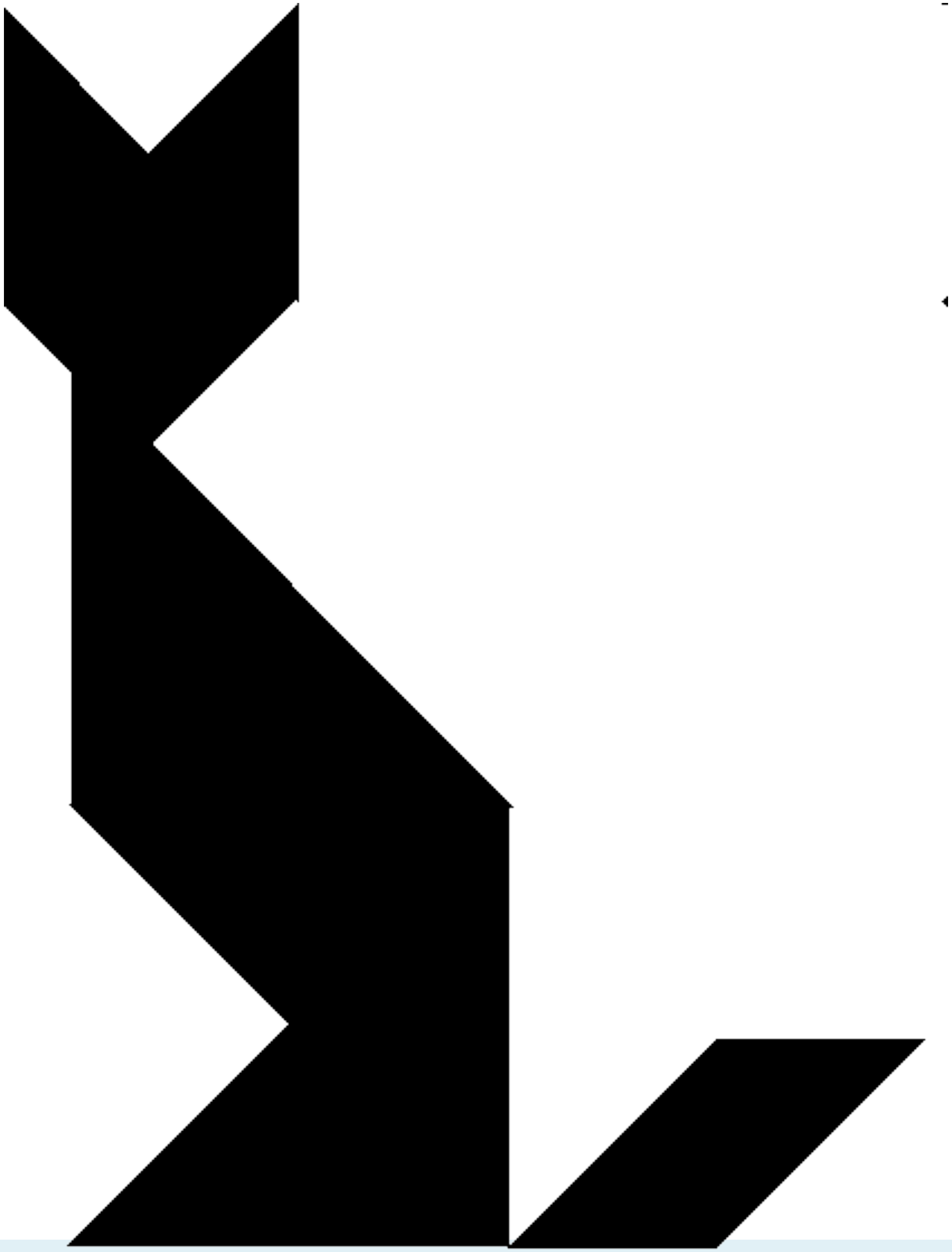
- Libraries vs. frameworks
- Whitebox vs. blackbox frameworks

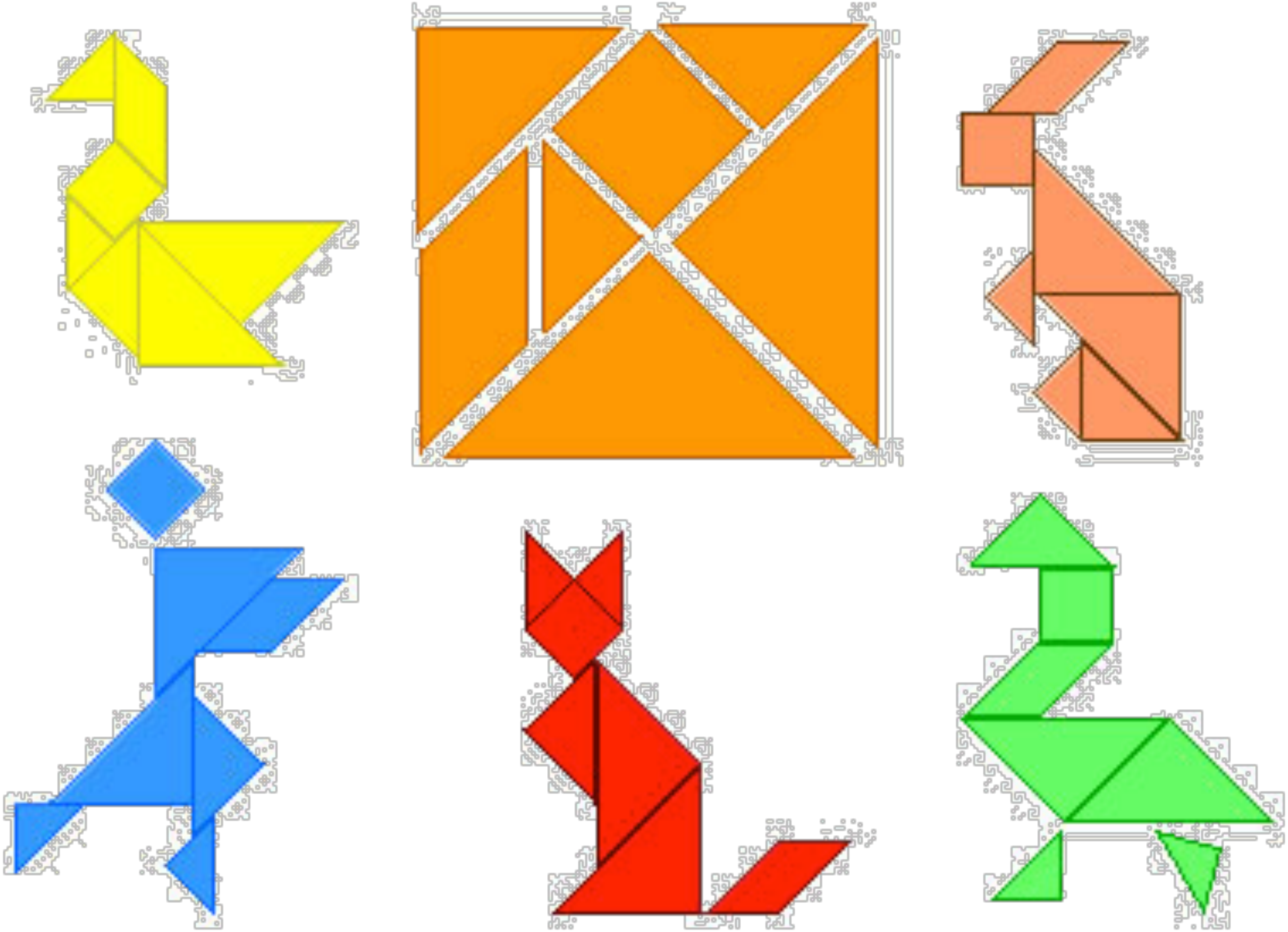
Today:

- Libraries and frameworks for reuse, continued
  - Domain engineering
  - Practical considerations

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





(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

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

```
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;
    }
}
```

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

Aside: Java reflection

- *Reflection* enables programmatic access to language elements
  - e.g., `java.lang.Class`,
    `java.lang.reflect.Method`,
    `java.lang.reflect.Field`
- Can use reflection to dynamically load plugins, e.g.:

```
Plugin p = (Plugin) Class.forName(args[1]).newInstance();
```

Aside: The `java.util.ServiceLoader`

- Uses reflection to load classes from a standard configuration (`META-INF/services/`…)
- E.g.,

```
import java.util.ServiceLoader;
…
for (Plugin p : ServiceLoader.load(Plugin.class)) {
    …
}
```

Example: An Eclipse plugin

- 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.text
or.BasicTextEditorActionContributor"
    class="myeditor.editors.XMLEditor
    id="myeditor.editors.XMLEditor">
   </editor>
</extension>

</plugin>
```

Example: An Eclipse plugin

- At last, the actual plugin
- XMLEditor.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();
                  setSourceViewerConfiguratio
                           new
XMLConfiguration(colorManager));
                  setDocumentProvider(
                           new XMLDocumentProvi
         }

         public void dispose() {
                  colorManager.dispose();
                  super.dispose();
         }
}
```

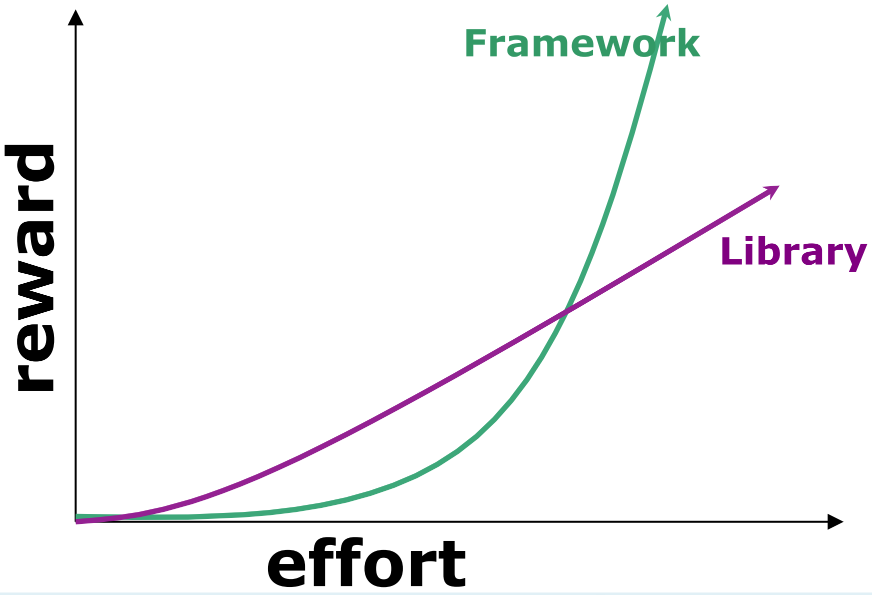
Example: A JUnit Plugin

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

      @Before
      public void setUp() {
          emptyList = new ArrayList<Strin
      }

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

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

In JUnit the plugin mechanism is Java annotations

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…