Principles of Software Construction: Objects, Design, and Concurrency

Design for large-scale reuse: Libraries and frameworks

Josh Bloch Charlie Garrod **Darya Melicher**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 4b due tonight (!)
- Required reading due next Tuesday: Effective Java, Items 51, 60, 62, and 64

Key concepts from Tuesday

Java I/O Recommendations

- Everyday use – `Buffered{Reader, Writer}`
- Casual use - `Scanner`
  - Easy but not general and swallows exceptions
- Stream integration – `Files.lines`
  - Support for parallelism in Java 9
- Async – `java.nio.AsynchronousFileChannel`
- Many niche APIs, e.g. memory mapped files, line numbering
  - Search them out as needed
- Consider Okio if third party API allowed
  - Very powerful, very fast, high-quality API

Conclusion

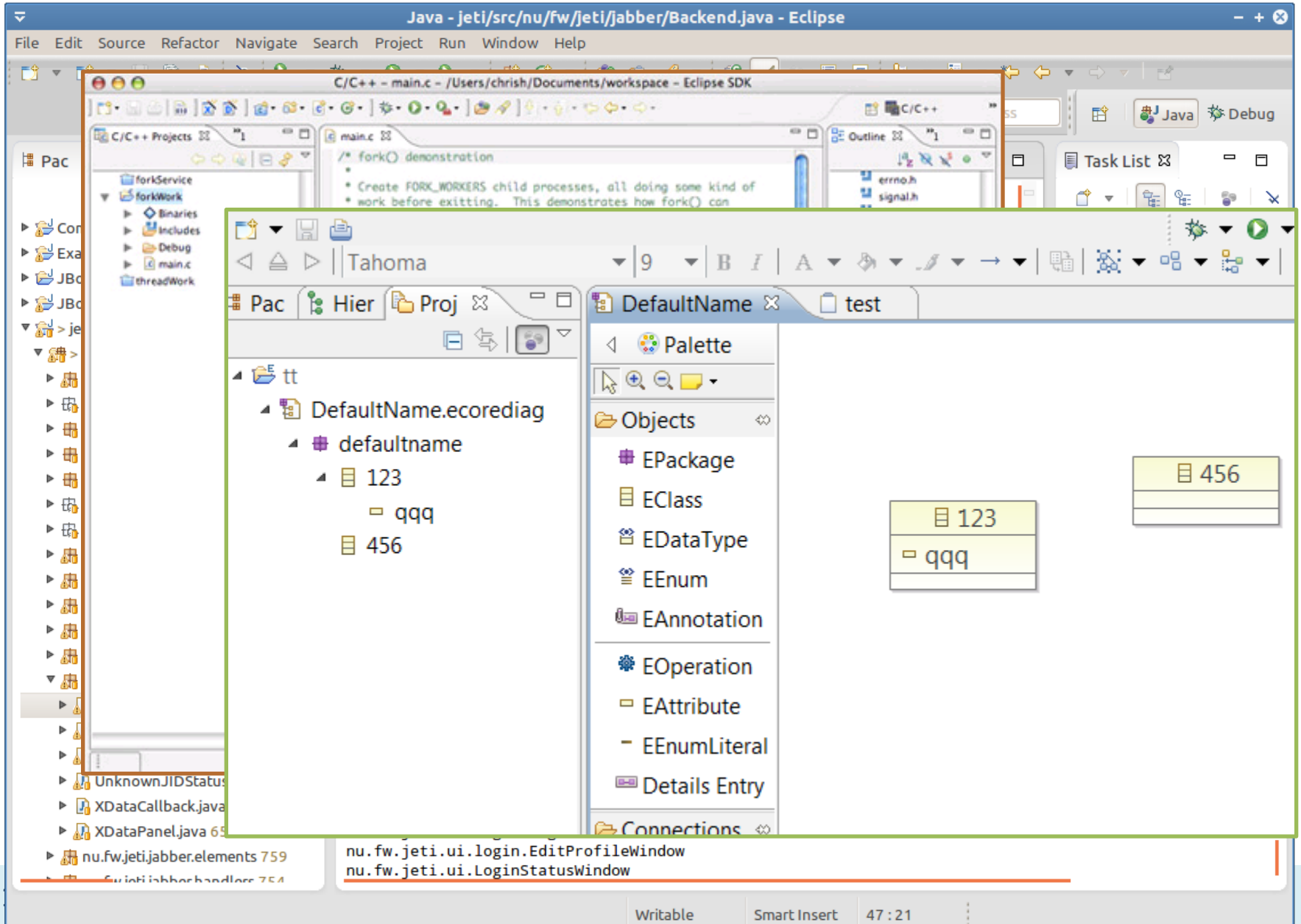
- Generics provide API flexibility with type safety
- Java I/O is a bit of a mess
  - There are many ways to do things
  - Use readers most of the time
- Reflection is tricky
  - but `Class.forName` and `newInstance` go a long way
  - A more powerful option that hides the reflection: `ServiceLoader`

Today: Libraries and frameworks for reuse

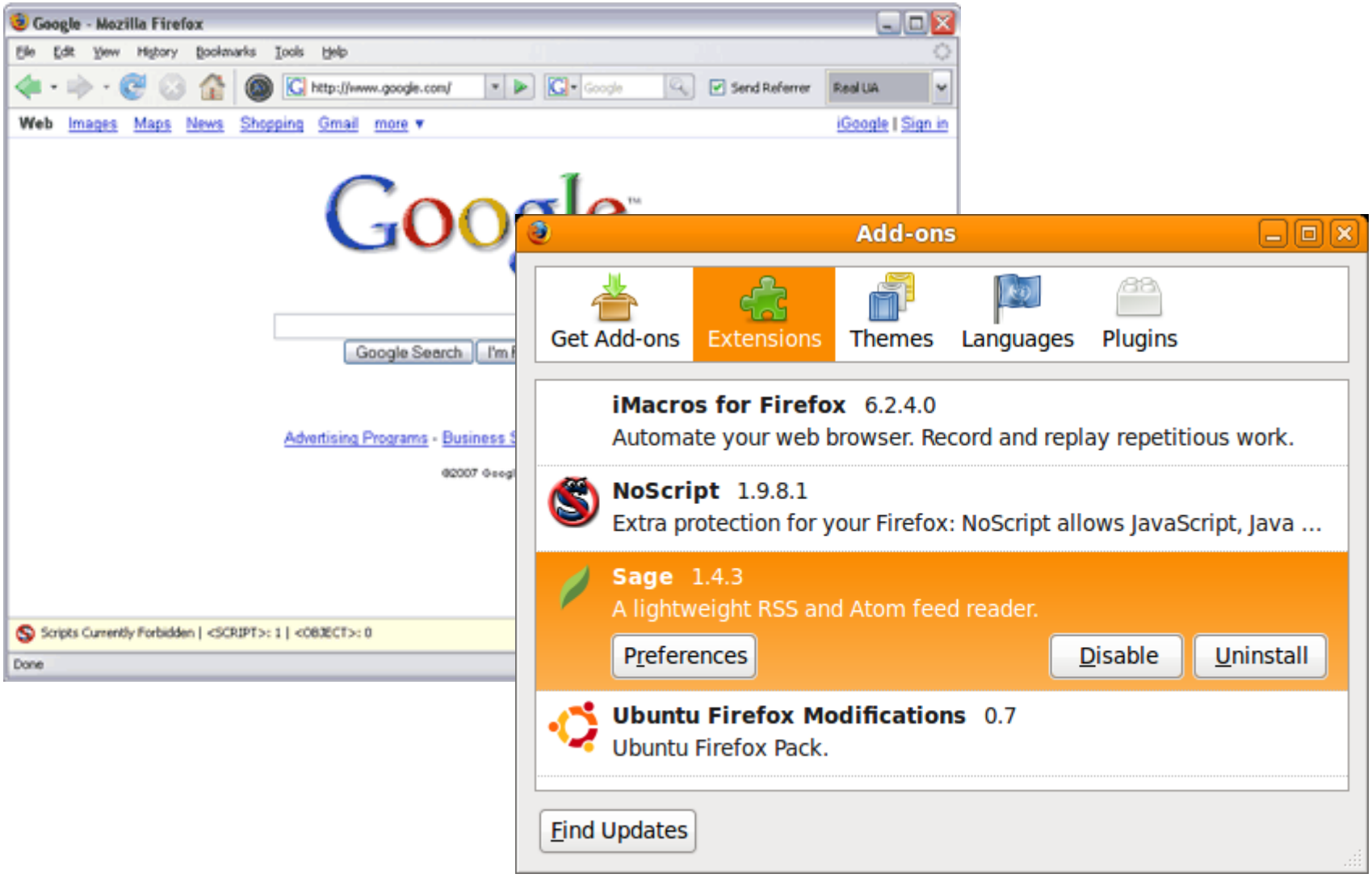
Earlier in this course: Class-level reuse

- 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, …

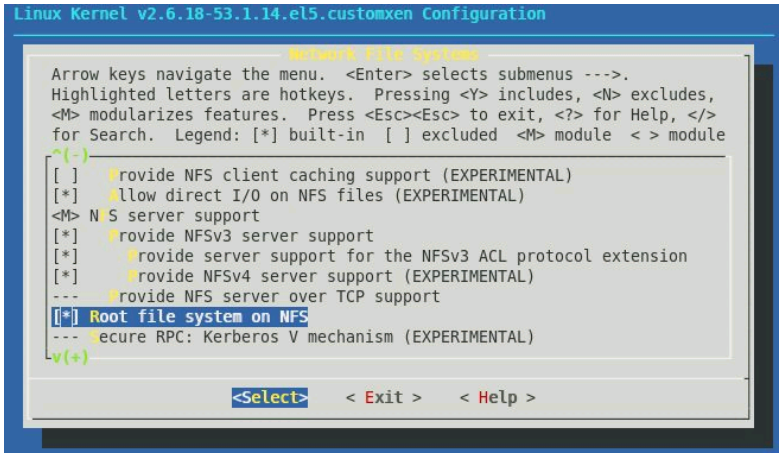
Reuse and variation: Family of development tools

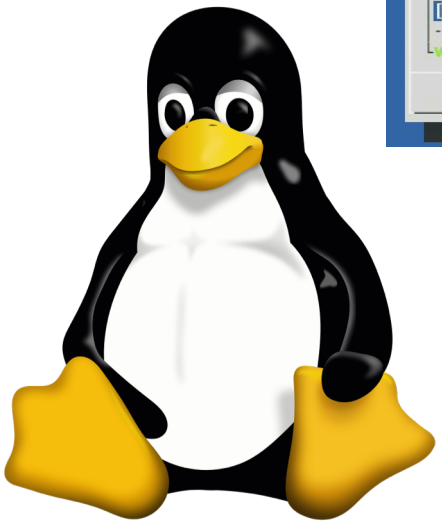


Reuse and variation: Web browser extensions



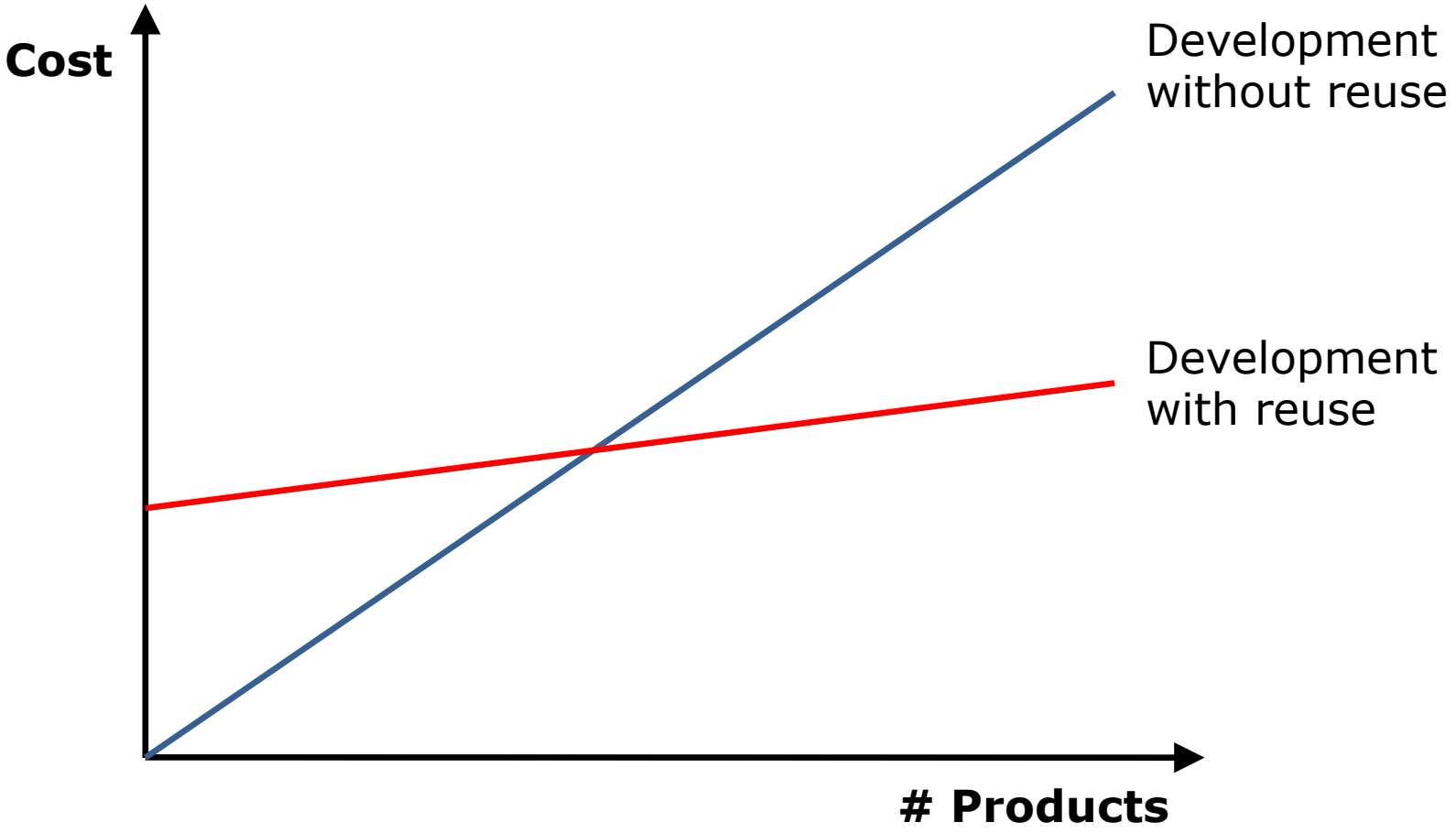
Reuse and variation: Flavors of Linux





Reuse and variation: Product lines

The promise


Cost
Development without reuse
Development with reuse
Products

Today: Libraries and frameworks for reuse

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

Today: Libraries and frameworks for reuse

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

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

- E.g.: Math, Collections, Graphs, I/O, Swing

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

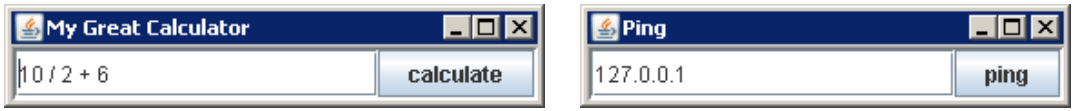
- E.g.: Eclipse, Firefox, Spring, Swing

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? What code will differ?

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 ""; }
  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) { ... }
}
```

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

Using the example framework again

```
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() { ... }
}
```

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.: Eclipse, Firefox, Spring, Swing

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

your code

Library

user interacts

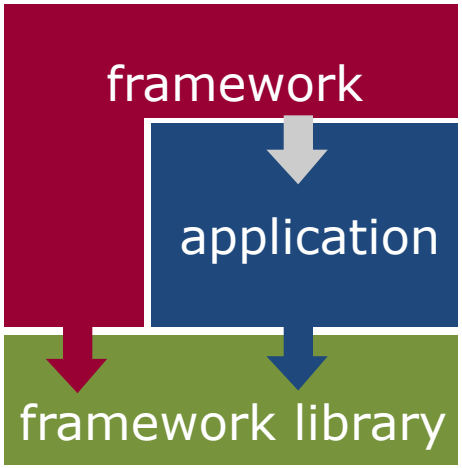
Framework

```
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

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

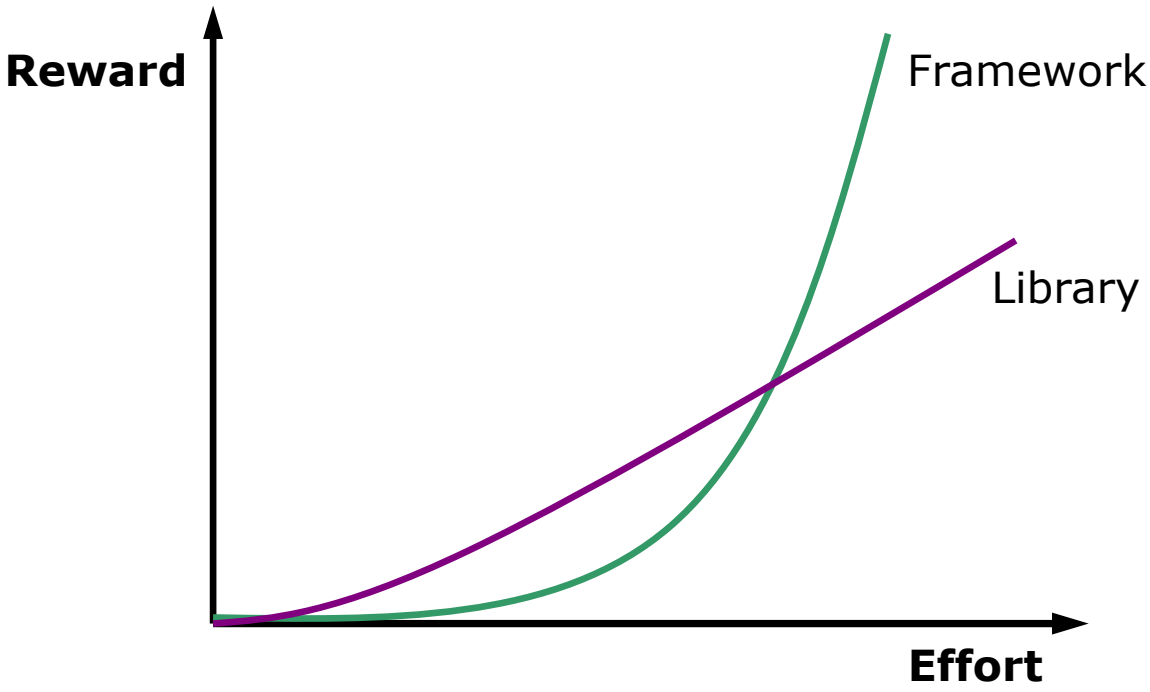
A simple example framework

```
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());
      ...
  }
```

A Carcassonne framework?

Learning library and framework

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



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
- *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

Today: Libraries and frameworks for reuse

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

Whitebox vs. blackbox framework

Whitebox frameworks	Blackbox frameworks
Subclassing and overriding methods	Composition; implementing a plugin interface
Subclass has `main()` but gives control to framework	Plugin-loading mechanism loads plugins and gives control to 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 ""; }
  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() { ... }
}
```

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 JPar
      contentPane.setBorder(new Bev
      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);
}
```

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 JPar
      contentPane.setBorder(new Bev
      JButton button = new JButton();
```

```
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"; }
}
```

```
}
```

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;
      JPanel contentPane = new
      contentPane.setBorder(new
      JButton button = new JButton();
```

```
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() { retu
  public String getInititalText() { re
  public void buttonClicked() {
      JOptionPane.showMessageDialog(nu
              + application.getInput() + " is "
              + calculate(application.getInput()));
  }
  public String getApplicationTitle() { return "My Great Calculator"; }
}
```

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

```
}
```

Whitebox vs. blackbox framework

Whitebox frameworks	Blackbox frameworks
Subclassing and overriding methods	Composition; implementing a plugin interface
Subclass has the main method but gives control to framework	Plugin-loading mechanism loads plugins and gives control to framework
Need to understand implementation of superclass	Only need to understand interface
Only one extension at a time	Multiple plugins simultaneously
Compiled together	More modularity; separate deployment possible (.jar, .dll, …)
Common pattern: Template method	Common patterns: Strategy, observer
“Developer frameworks”	“End-user frameworks”

Today: Libraries and frameworks for reuse

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

Today: Libraries and frameworks for reuse

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

Domain engineering: Reusing domain knowledge in new software products

- Understand users/customers in your domain
  - What might they need? What extensions are likely?
- Collect example applications
- Make a conscious decision what to support
  - Called *scoping*
- E.g., Eclipse policy:
  - At first, interfaces are internal
    - Unsupported, may change
  - When there are at least two distinct customers, public stable extension points created

Framework design and implementation process

- Define your domain (domain engineering)
- Identify potential common parts and variable parts
  - “Hot spots” vs. “cold spots”
- Design and write sample plugins or applications
- Factor out and implement common parts as framework
- Provide plugin interface and callback mechanisms for variable parts
  - Use well-known design principles and patterns where appropriate
- Get lots of feedback and iterate!

Framework design considerations

- Once designed there is little opportunity for change

Consider adding a method to the `Plugin` interface

```
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 JPar
      contentPane.setBorder(new Bev
      JButton button = new JButton();
```

```
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"; }
}
```

Requires changes to *all* plugins!

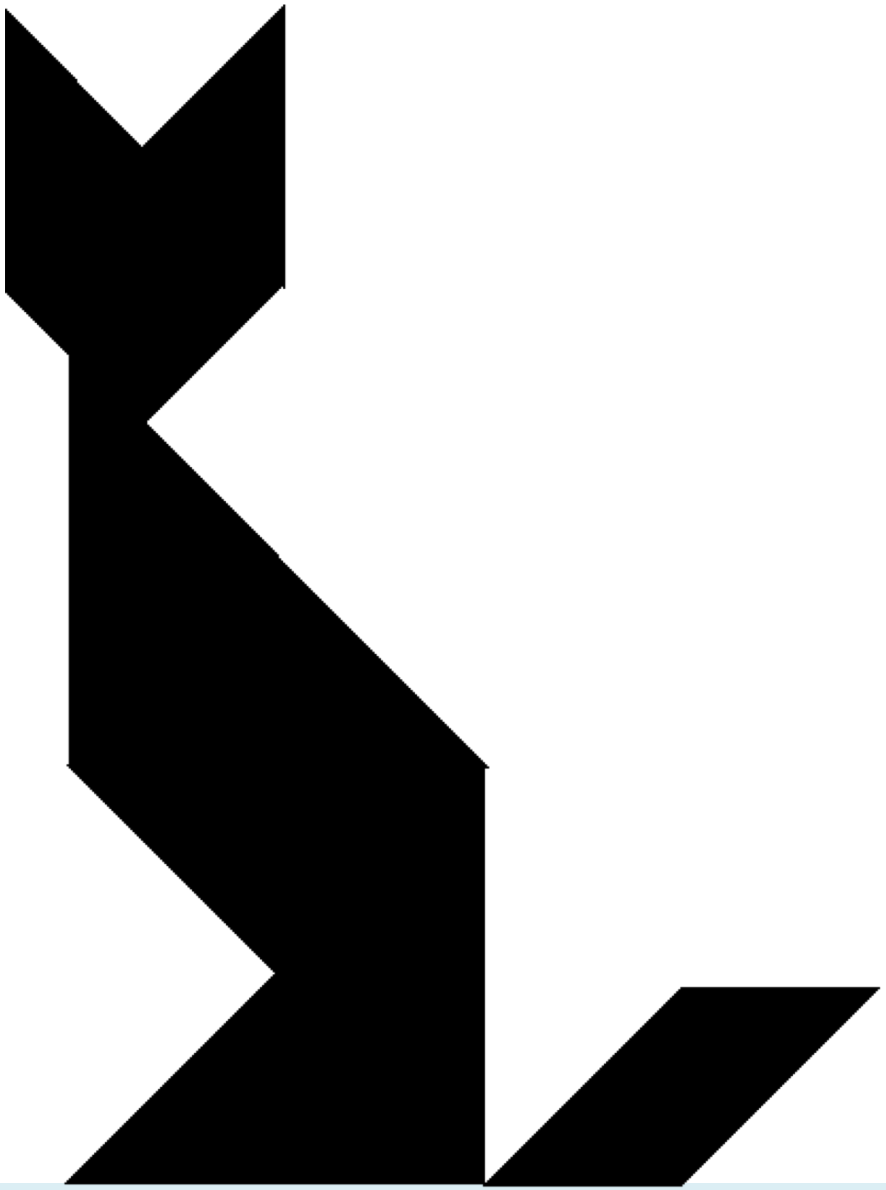
```
}
```

Framework design considerations

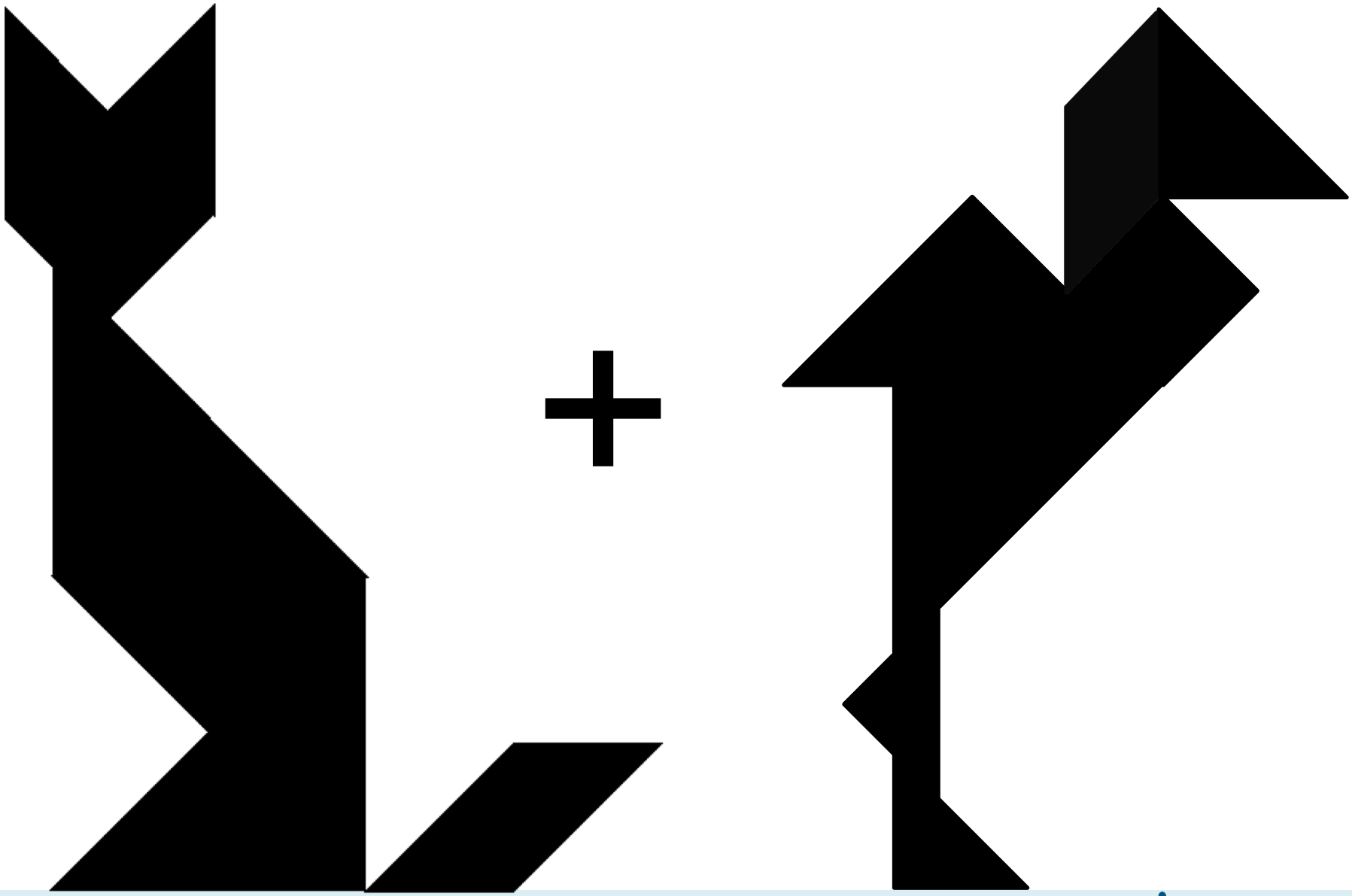
- Once designed there is little opportunity for change
- Writing a plugin or extension should NOT require modifying the framework source code
- Key decision: Separating common parts from variable parts
- Possible problems:
  - Too few extension points: Limited to a narrow class of users
  - Too many extension points: Hard to learn, slow
  - Too generic: Little reuse value

The use vs. reuse dilemma

The use vs. reuse dilemma

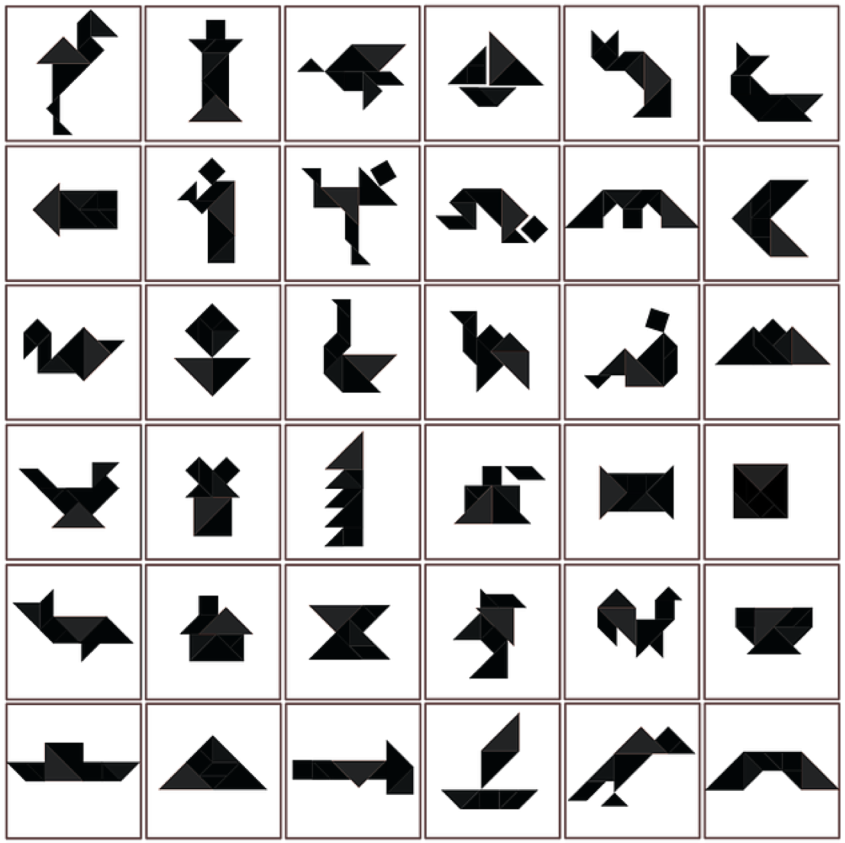


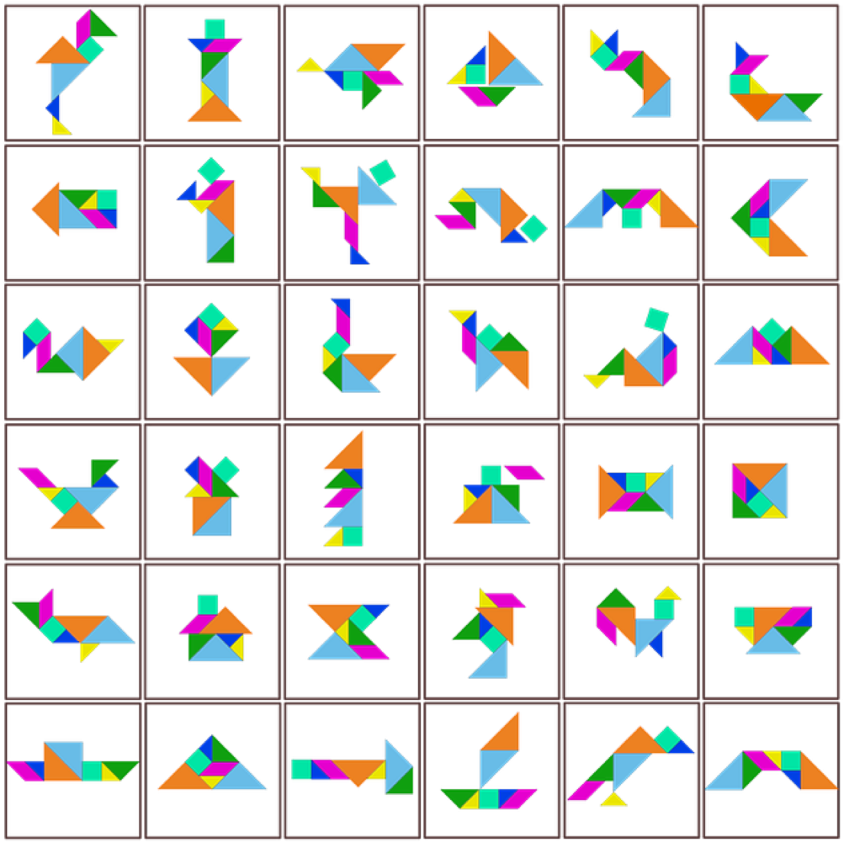
The use vs. reuse dilemma



The use vs. reuse dilemma

- 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."

Clemens Szyperski

Today: Libraries and frameworks for reuse

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

Today: Libraries and frameworks for reuse

- Terminology and examples
- Whitebox and blackbox frameworks
- Designing a framework
- **Implementation details (more in recitation)**

Running a framework

- Some frameworks are runnable by themselves
  - E.g., Eclipse
- Other frameworks must be extended to be run
  - E.g.: Swing, JUnit, MapReduce, Servlets

Methods to load plugins

1. Client writes `main()`, creates a plugin, and passes it to framework
2. Framework writes `main()`, client passes name of plugin as a command line argument or environment variable
   - E.g., use reflection to dynamically load plugins:

   ```
   Plugin p = (Plugin) Class.forName(args[1]).newInstance();
   ```
3. Framework looks in a magic location
   - Config files or .jar files are automatically loaded and processed
   - E.g., use `java.util.ServiceLoader` to load classes from a standard configuration (`META-INF/services/…`)
4. GUI for plugin management
   - E.g., web browser extensions

Summary: Libraries and frameworks for reuse

- Terminology and examples
  - Library, framework, callback, lifecycle method, …
- Whitebox frameworks vs. blackbox frameworks
- Designing a framework
  - Domain engineering
  - Key decision: Separating common parts from variable parts
  - Writing a plugin should NOT require modifying the framework
  - Use vs. reuse
- Implementation details (more in recitation)