Principles of Software Construction: Objects, Design and Concurrency

Frameworks

15-214

toad

Spring 2013

Christian Kästner Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

With material from Ciera Jaspan, Jonathan Aldrich, Bill Scherlis, Travis Breaux, and Erich Gamma

Administrivia

- Homework 4b (design + core) due tonight
- Homework 4c (GUI) due next Tuesday

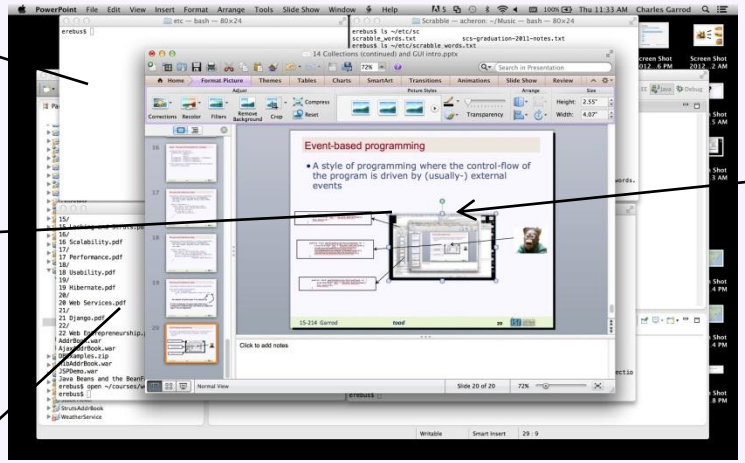
Event-based programming

- A style of programming where the control-flow of the program is driven by (usually-) external events

```
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(42)
}
```

```
public void performAction(ActionEvent e) {
    bigBloatedPowerPointFunction(e);
    withANameSoLongIMadeItTwoMethods(e);
    yesIKnowJavaDoesntWorkLikeThat(e);
}
```

```
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(40)
}
```



Recap – from before the break

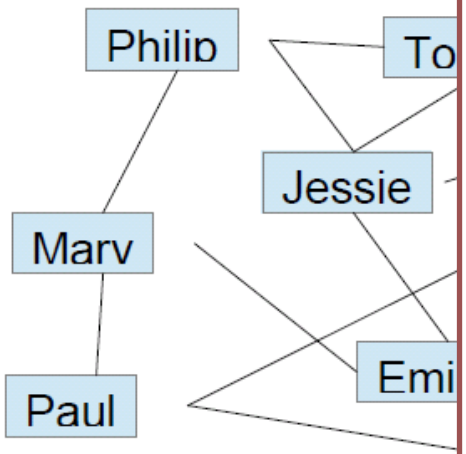
Reuse and Variations

15-214 Homework 0

Homework 0: A Friendship Graph

The goal of this homework is to get familiar with the infrastructure used in this course and to attempt some first steps in Java programming. To simplify the programming task, we have selected a simple task from a familiar domain.

Task: We want to represent the friendship
we want to calculate the shortest connectio
Test your implementation with the followin



We expect a client implementation roughly

```
FriendGraph g = new FriendGraph();
Person anne = new Person("Anne");
```

15-214 Homework 1

Homework #1: Graph Algorithms at Facecook

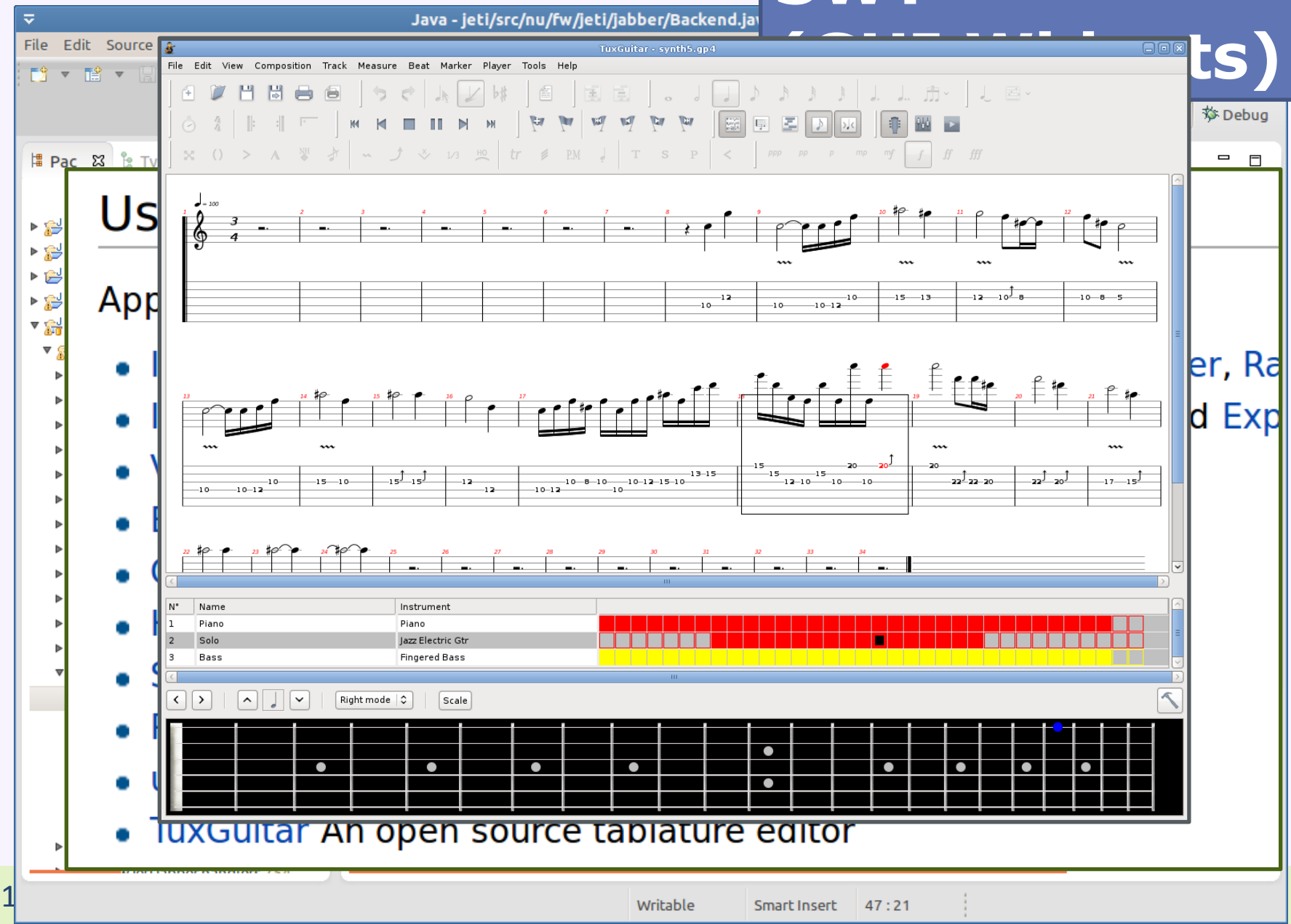
Due Thursday, January 31st at 11:59 p.m.

After graduating from CMU you have been hired by Facecook, an up-and-coming social network company, to develop analytical algorithms for their website. The Facecook design committee has developed a `Graph` interface that will be used to store social network information for Facecook users. You have decided to experiment with two different `Graph` implementations to find which representation is best for different purposes. Be careful! To ensure that your implementation works with existing Facecook code, you must not alter the `Graph` interface in any way!

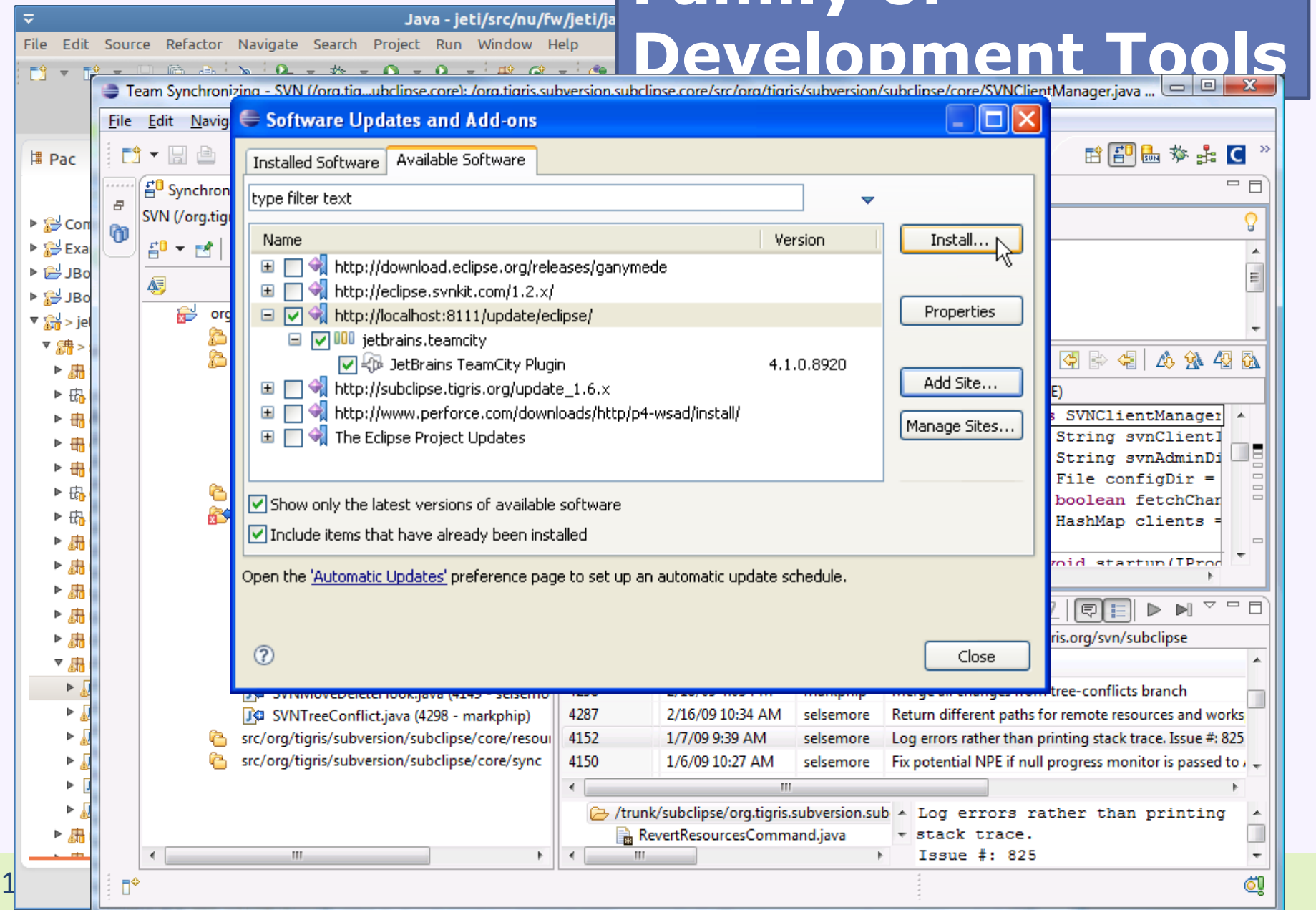
The goals for this assignment are:

- Understand and apply the concepts of polymorphism and encapsulation
- Learn to use Javadoc
- Better understand graphs and their multiple representations
- Familiarize yourself with Java and Eclipse

Reuse and Variations

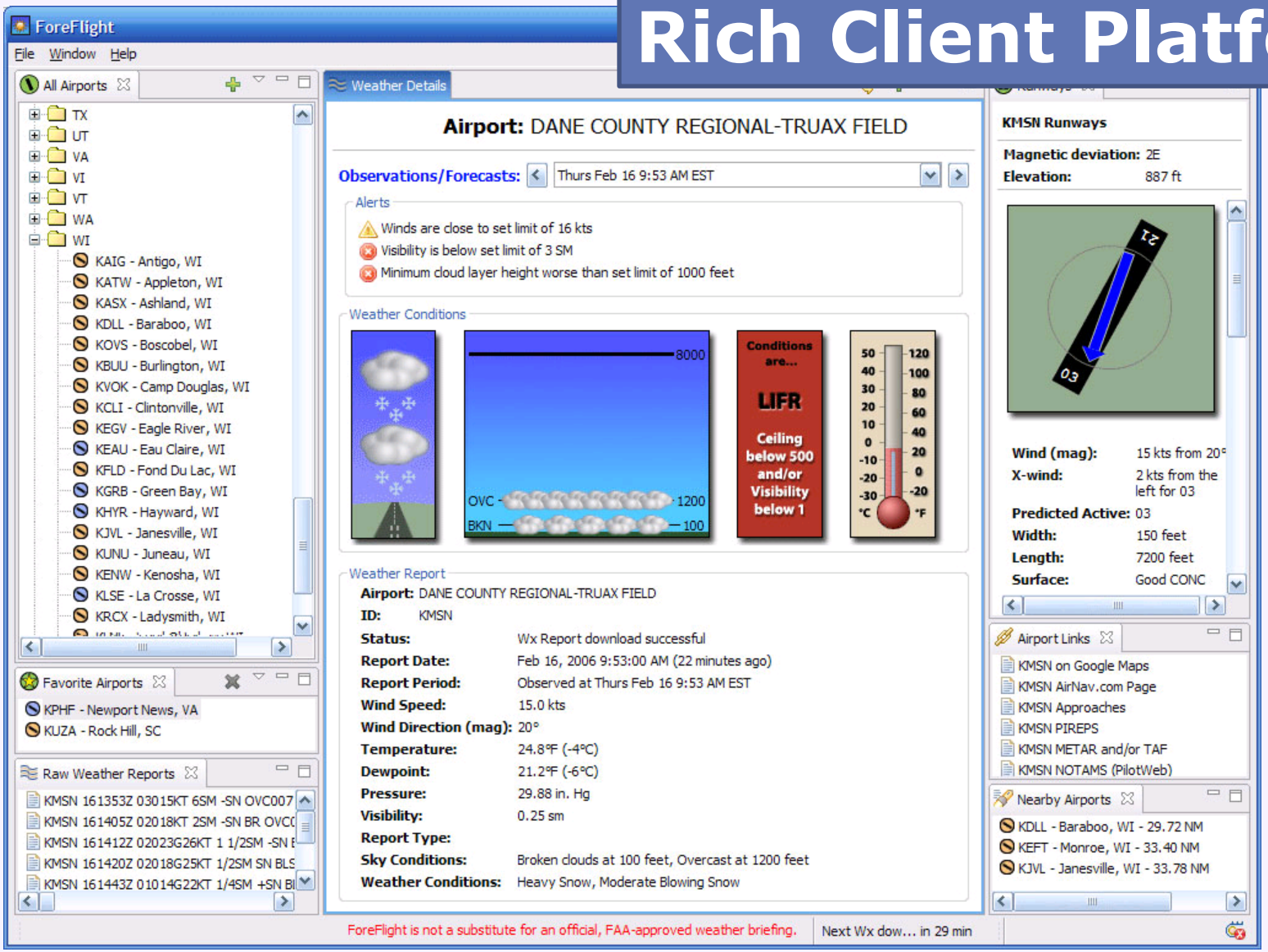


Reuse and Variations

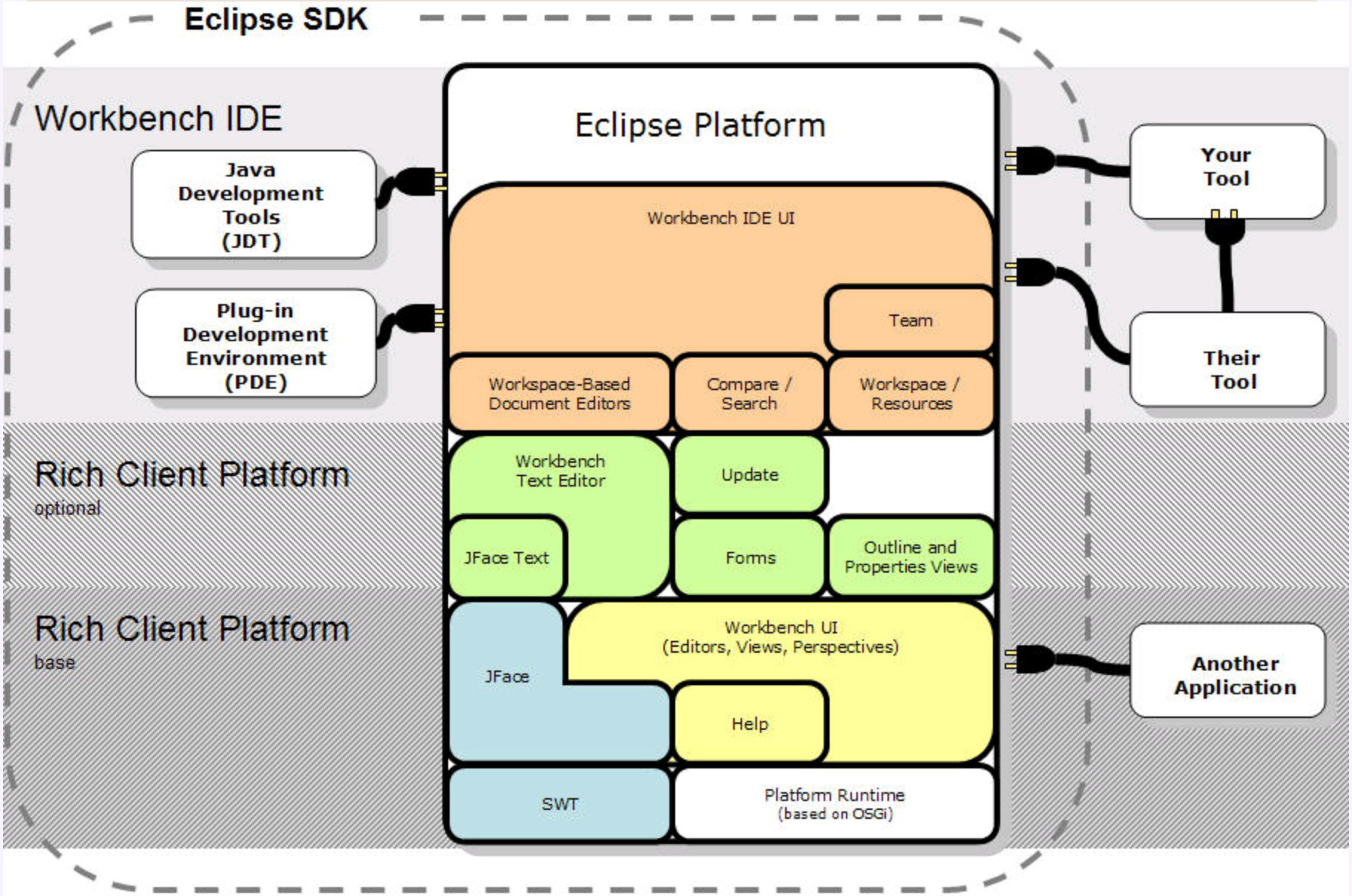


Reuse and Variations

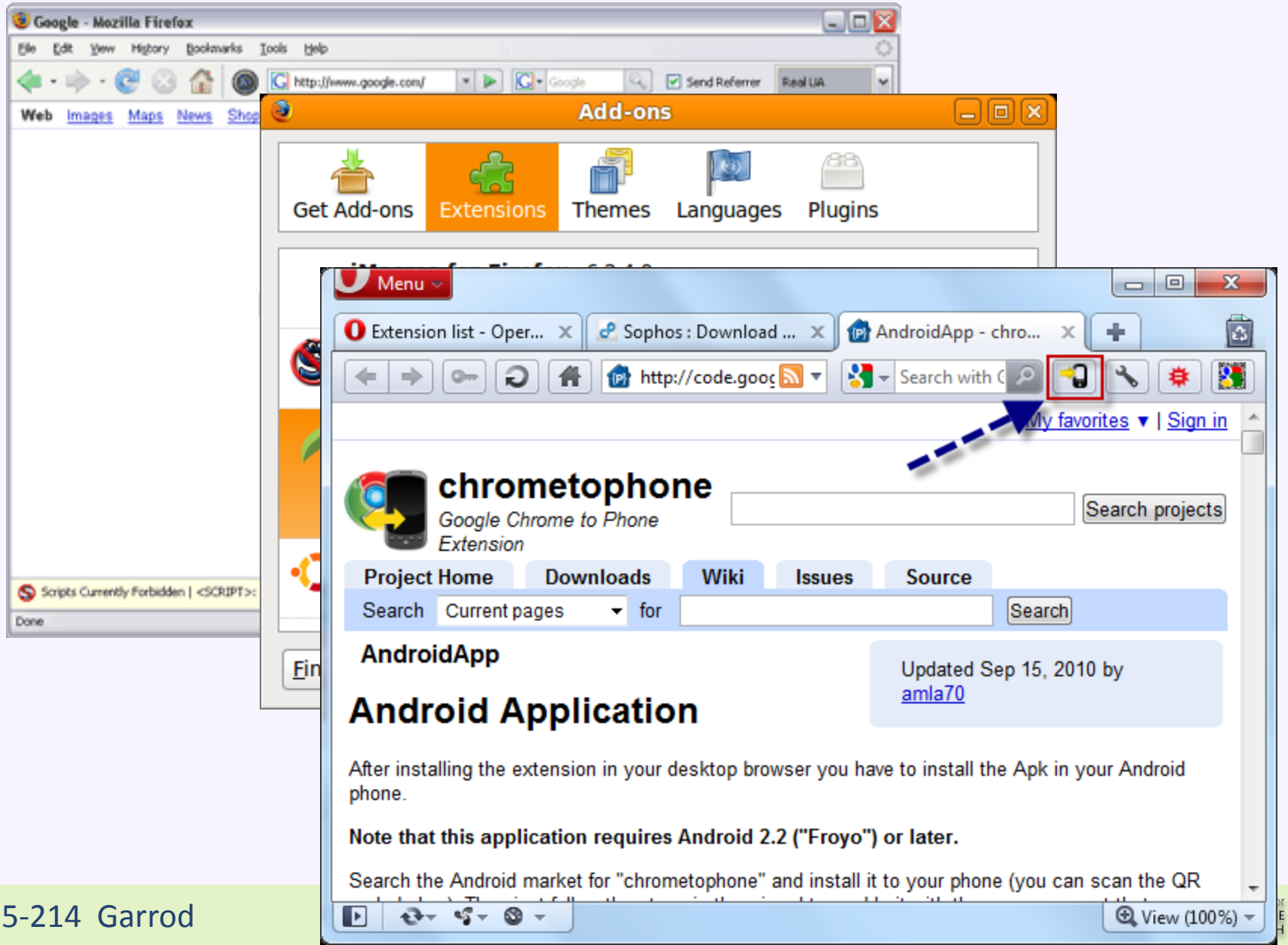
Eclipse Rich Client Platform

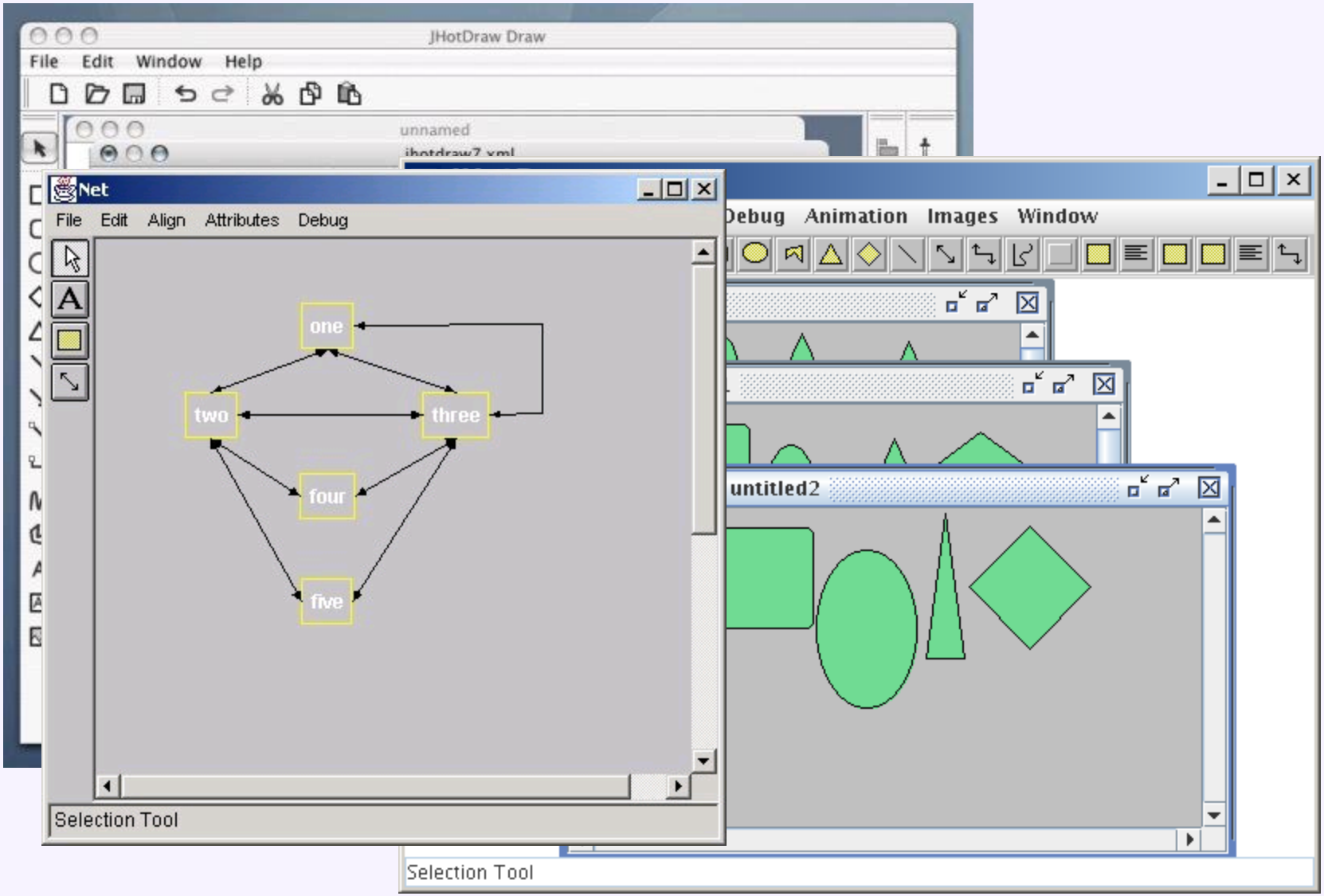


institute for SOFTWARE RESEARCH

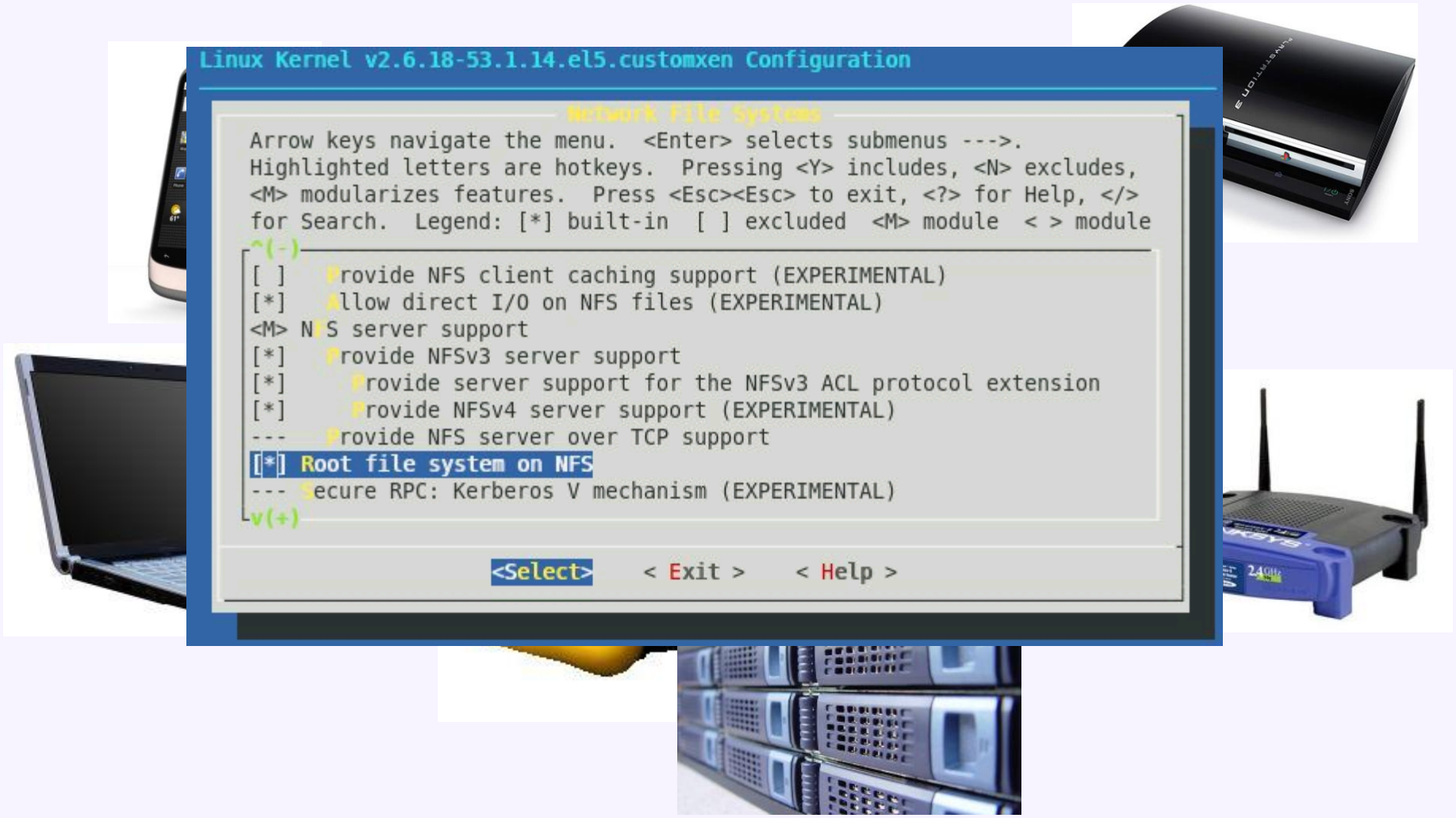

Eclipse SDK
Workbench IDE
Java Development Tools (JDT)
Plug-in Development Environment (PDE)
Eclipse Platform
Workbench IDE UI
Team
Workspace-Based Document Editors
Compare / Search
Workspace / Resources
Workbench Text Editor
Update
JFace Text
Forms
Outline and Properties Views
Rich Client Platform
optional
Rich Client Platform
base
JFace
Workbench UI (Editors, Views, Perspectives)
Help
SWT
Platform Runtime (based on OSGi)
Your Tool
Their Tool
Another Application

Reuse and Variations




JHotDraw Draw
File Edit Window Help
unnamed
Net
File Edit Align Attributes Debug
one
two
three
four
five
Selection Tool
Debug Animation Images Window
untitled2
Selection Tool

Reuse and Variations



Reuse and Variation

```
double getSaleTotal() {
  double result = 0;
  for (LineItem i : items)
        result = result + addTax(i.value*i.count);
  return result;
}
double addTax(int value) {
  return value + TAX*value;
}
```

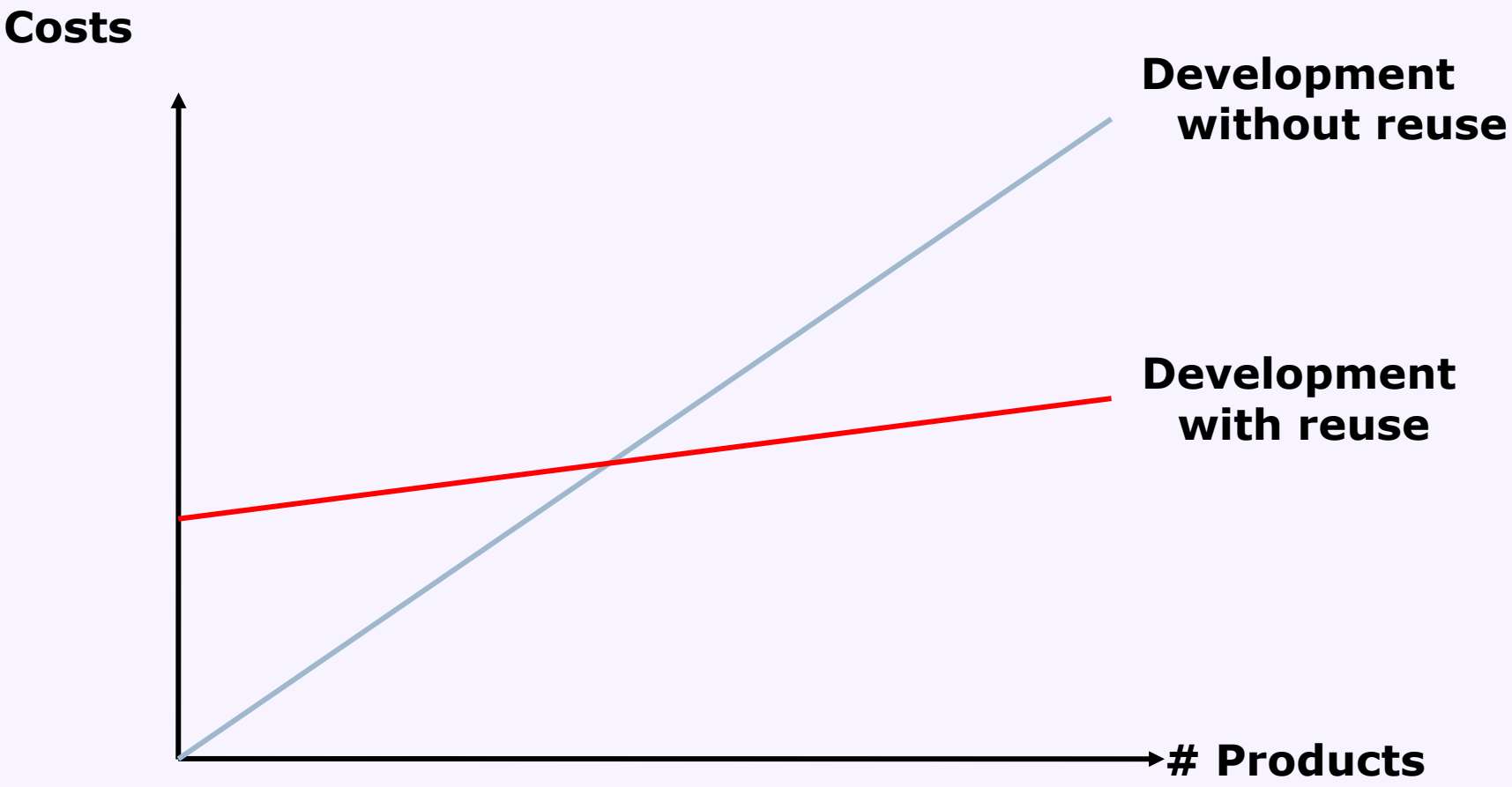
Reuse and Variation

Reuse and Variation

- Clone and Own
- Subroutines
- Libraries
- Frameworks
- APIs
- Platforms
- Configuration

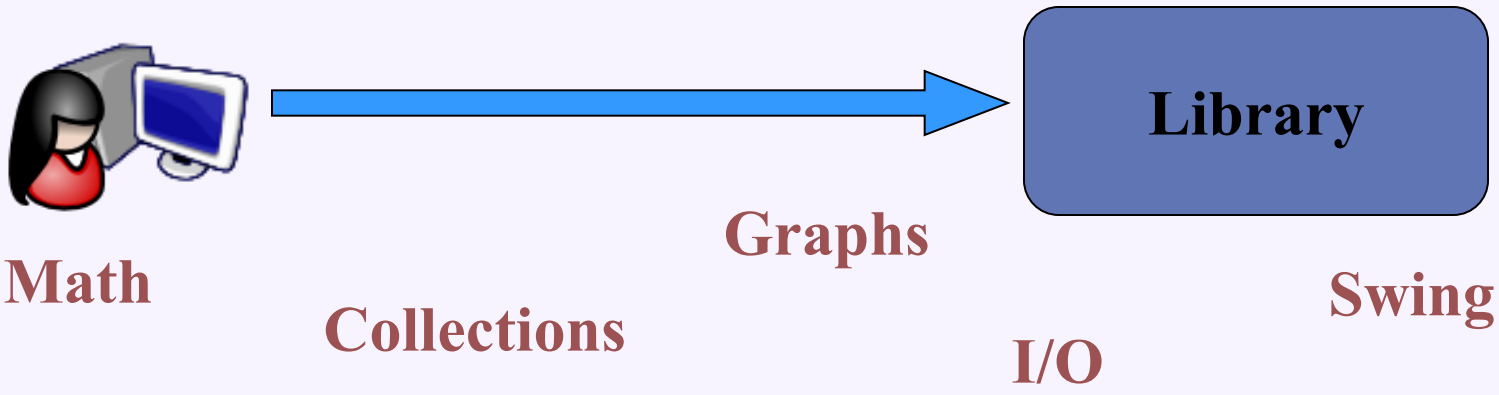
- From reuse-in-the-small to systematic planned reuse and product planning

The Promise


Costs
Development without reuse
Development with reuse
Products

Terminology: Libraries

- Library: A set of classes and methods that provide reusable functionality
- Client calls library to do some task
- Client controls
  - System structure
  - Control flow
- The library executes a function and returns data



Influence of Design

- Good design helps to write good libraries
- Loosely coupled classes easiest to detach and reuse as library
- Libraries with high cohesion easiest to reuse (and most useful)
- Few and small interfaces, modular protection, modular continuity, …
- Façade pattern, observer pattern, …
- -> Modular composability

Terminology: Frameworks

- Framework: Reusable skeleton code that can be customized into an application
- Framework controls
  - Program structure
  - Control flow

- Framework calls back into client code
  - The Hollywood principle: "Don't call us; we'll call you."

Framework

Spring

Eclipse

Firefox

Swing

Applet

But this is an application!

Hey, wasn't this a library?

More terms

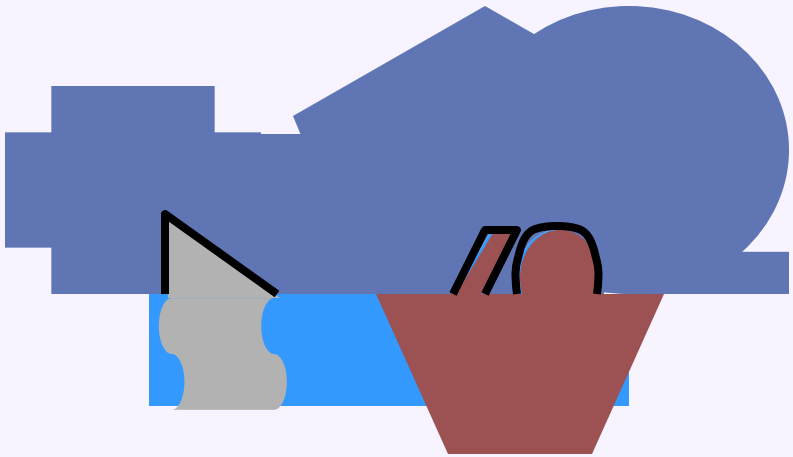
- API: Application Programming Interface, the interface of a library or framework
- Client: The code that uses an API
- Plugin: Client code that customizes a framework
- Extension point, hot spot: A place where a framework supports extension with a plugin

More terms

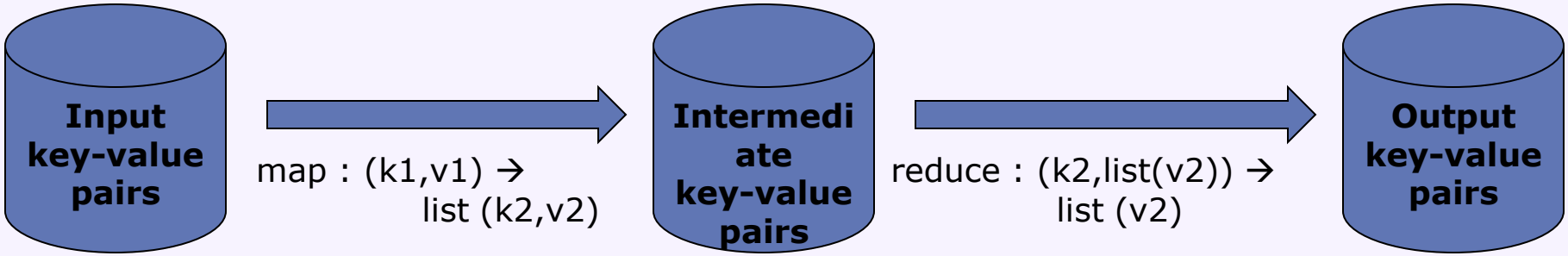
- Protocol: The expected sequence of interactions between the API and the client
- Callback: A plugin method that the framework will call to access customized functionality
  - (see also Strategy design pattern)
- Lifecycle method: A callback method of an object that gets called in a sequence according to the protocol and the state of the plugin

Using an API

- Like a partial design pattern
- Framework provides one part
- Client provides the other part

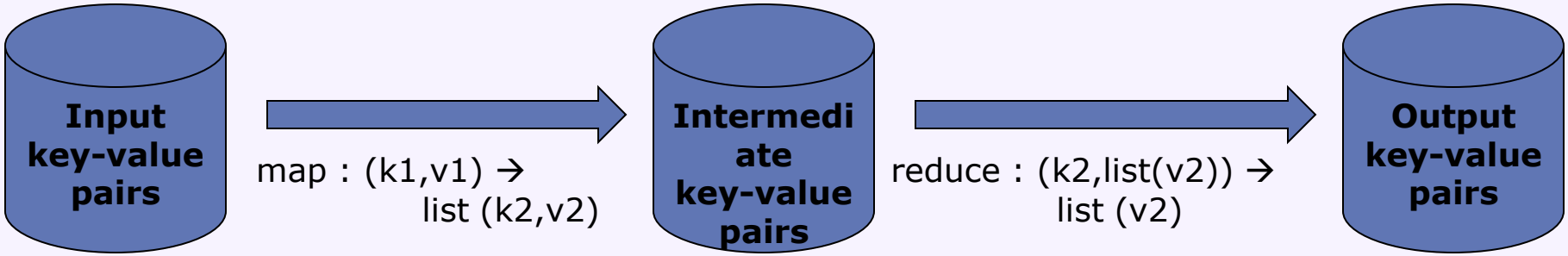


Google's Map-Reduce



- Programming model for processing large data sets
- Example: word count
  - map(URL, contents):
    for each word w in contents
    emit (w, 1)
  - reduce(word, listOfCounts):
    for each count c in listOfCounts
    result += c
    emit result

Google's Map-Reduce



- Questions
  - Is this a framework? How do you know?
  - What are the benefits?
  - Could those benefits be achieved if it were not?

Some Benefits of Map-Reduce

- Automatically parallelizes and distributes computation
- Scales to 1000s of machines, terabytes of data
- Automatically handles failure via re-execution
- Simple programming model
  - Successful: hundreds of plugins
  - Functional model facilitates correctness

Constraints

- Computation must fit the model
  - Not everything can be phrased in terms of map and reduce
- Map and Reduce must be largely functional
  - Side effects allowed but must be atomic and idempotent
- What benefits does the client get in exchange for accepting these restrictions?

Hadoop: Map-Reduce in Java

- See http://hadoop.apache.org/

Implementing Frameworks

- Family of programs consisting of buttons and text fields only

- Share 90% of the source code
  - Main method
  - Initialization of GUI
  - Layout
  - Closing the window
  - …

Calculator

```
public class Calc extends JFrame {
        private JTextField textfield;
        public static void main(String[] args) { new Calc().setVisible(true); }
        public Calc() { init(); }
        protected void init() {
                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(/* code zum berechnen */);
                this.setContentPane(contentPane);
                this.pack();
                this.setLocation(100, 100);
                this.setTitle("My Great Calculator");
                // impl. for closing the window
        }
}
```

White-Box Frameworks

- Extension through subclassing and method overriding
- see Template Method design pattern

- Design steps:
  - Identify the common and the variable code
  - Abstract variable code as method calls

- Subclass has main method but gives control to framework

Example Whitebox Framework

```
public abstract class Application extends JFrame {
        protected abstract String getApplicationTitle();
        protected abstract String getButtonText();
        protected String getInititalText() {return "";}
        protected void buttonClicked() { }
        private JTextField textfield;
        pub
        pro
    }
    protecte
}
```

```
public class Calculator extends Application {
        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())); }
        protected String getApplicationTitle() { return "My Great Calculator"; }
        public static void main(String[] args) {
                new Calculator().setVisible(true);
        }
}
```

```
public class Ping extends Application {
        protected String getButtonText() { return "ping"; }
        protected String getInititalText() { return "127.0.0.1"; }
        protected void buttonClicked() {  /* … */  }
        protected String getApplicationTitle() { return "Ping"; }
        public static void main(String[] args) {
                new Ping().setVisible(true);
        }
}
```

Black-Box Frameworks

- Extension by Implementing Plug-in Interface
- see Strategy Pattern, Observer Pattern

- Design steps:
  - Identify the common and the variable code
  - Abstract variable code as methods of an interface
  - Decide whether there may be one or multiple plug-ins

- Plug-in loading mechanisms loads plug-ins and gives control to the framework

Example Black-Box 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 Bor
                contentPane.setBorder(new BevelBorder(B
                JButton button = new JButton();
                if (plugin != null)
                        button.setText(plugin.getButtonT
                else
                        button.setText("ok");
                contentPane.add(button, BorderLayout.EAST);
                textfiel
                if (plug

                textfiel
                content
                if (plug

                this.set
                …
        }
        public String getIn
}
```

```
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.getInput() + " is "
                        + calculate(application.getText())); }
        public String getApplicationTitle() { return "My Great Calculator"; }
}
```

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

Reuseable Plugins?

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

```
public class Application extends JFrame implements InputProvider {
        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 B
                contentPane.setBorder(new BevelBorder
                JButton button = new JButton();
                if (plugin != null)
                        button.setText(plugin.getButt
                else
                        button.setText("ok");
                contentPane.add(button, BorderLayout.LAST);
                textfield = new JTextField("");
                if (plug

                textfiel
                content
                if (plug

                this.set
                …
        }
        public String getIn
}
```

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

```
public class CalcPlugin implements Plugin {
        private InputProvider application;
        public void setApplication(InputProvider 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.getInput() + " is "
                        + calculate(application.getInput())); }
        public String getApplicationTitle() { return "My Great
```

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

Running a Framework

- Some frameworks are runnable by themselves
  - E.g. Eclipse
- Other frameworks must be extended to be run
  - MapReduce, Swing, Servlets, JUnit
- The Golden Rule of Framework Design:

  Extending the framework should NOT require modifying the framework source code!

Loading Plugins (options)

- 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 there are automatically loaded and processed
- Often Graphical User Interface for Plugin Management

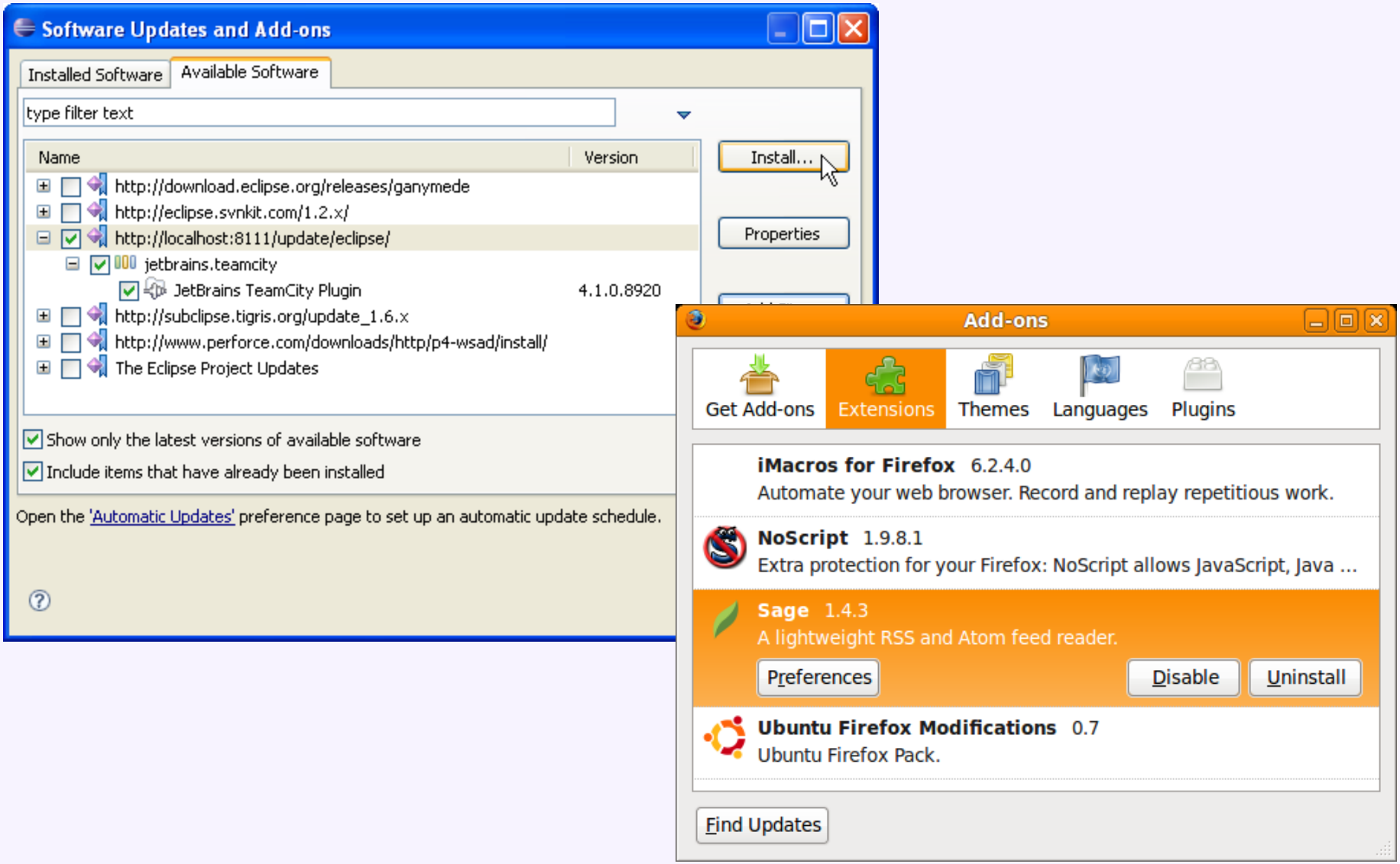
Example Plugin Loader (using Java Reflection)

```
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);
            }
      }
}
```

Example 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);
}
```

Plugin Management



Supporting Multiple Plug-ins

- see Observer Pattern
- Load and initialize multiple plugins
- Plugins can register for events
- Multiple plug-ins 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;
      }
}
```

Whitebox vs Blackbox Framework

- Whitebox uses subclassing
  - Allows to extend every nonprivate method
  - Need to understand implementation of superclass
  - Only one extension at a time
  - Compiled together
  - Often "developer frameworks"
- Blackbox uses composition
  - Allows to extend only functionality exposed in interface
  - Only need to understand the interface
  - Multiple plugins
  - "Modularity"
  - Separate deployment possible (.jar)
  - Often "end-user frameworks", platforms

Example: An Eclipse Plugin

- A popular Java IDE
- More generally, a framework for tools that facilitate "building, deploying and managing software across the lifecycle."
- Plug-in 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
          editor.BasicTextEditorActionContribut
          or"
            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

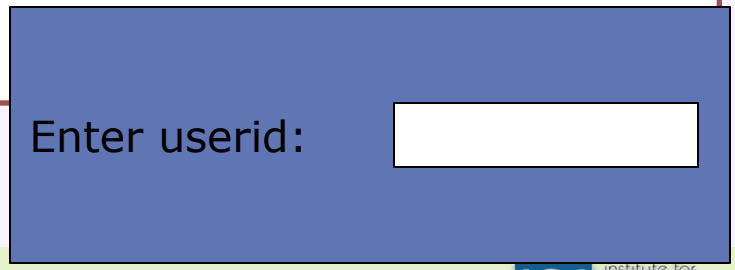
Java Swing: It's a Library!

- Create a GUI using pre-defined containers
  - JFrame, JPanel, JDialog, JMenuBar
- Use a layout manager to organize components in the container
- Add pre-defined components to the layout
  - Components: JLabel, JTextField, JButton

This is no different that the File I/O library!

Swing: Containers and Components

```
// create the container
JPanel panel = new JPanel();


// create the label, add to the container
JLabel label = new JLabel();
label.setText("Enter your userid:");
panel.add(label);


// create a text field, add to the container
JTextField textfield = new JTextField(16);
panel.add(textfield)
```

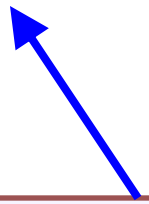


Swing: Layout Managers

```
panel.setLayout(new GridBagLayout());

GridBagConstraints c = new GridBagConstraints();


// create and position the button
JButton button = new JButton("Click Me!");
c.fill = GridBagConstraints.HORIZONTAL;
c.gridx = 0;  // first column
c.gridy = 1;  // second row
c.gridwidth = 2;     // span two columns
c.weightx = 1.0;     // use all horizontal space
c.anchor = GridBagConstraints.WEST;
c.insets = new Insets(0,5,0,5); // add side padding
pane.add(button, c);
```

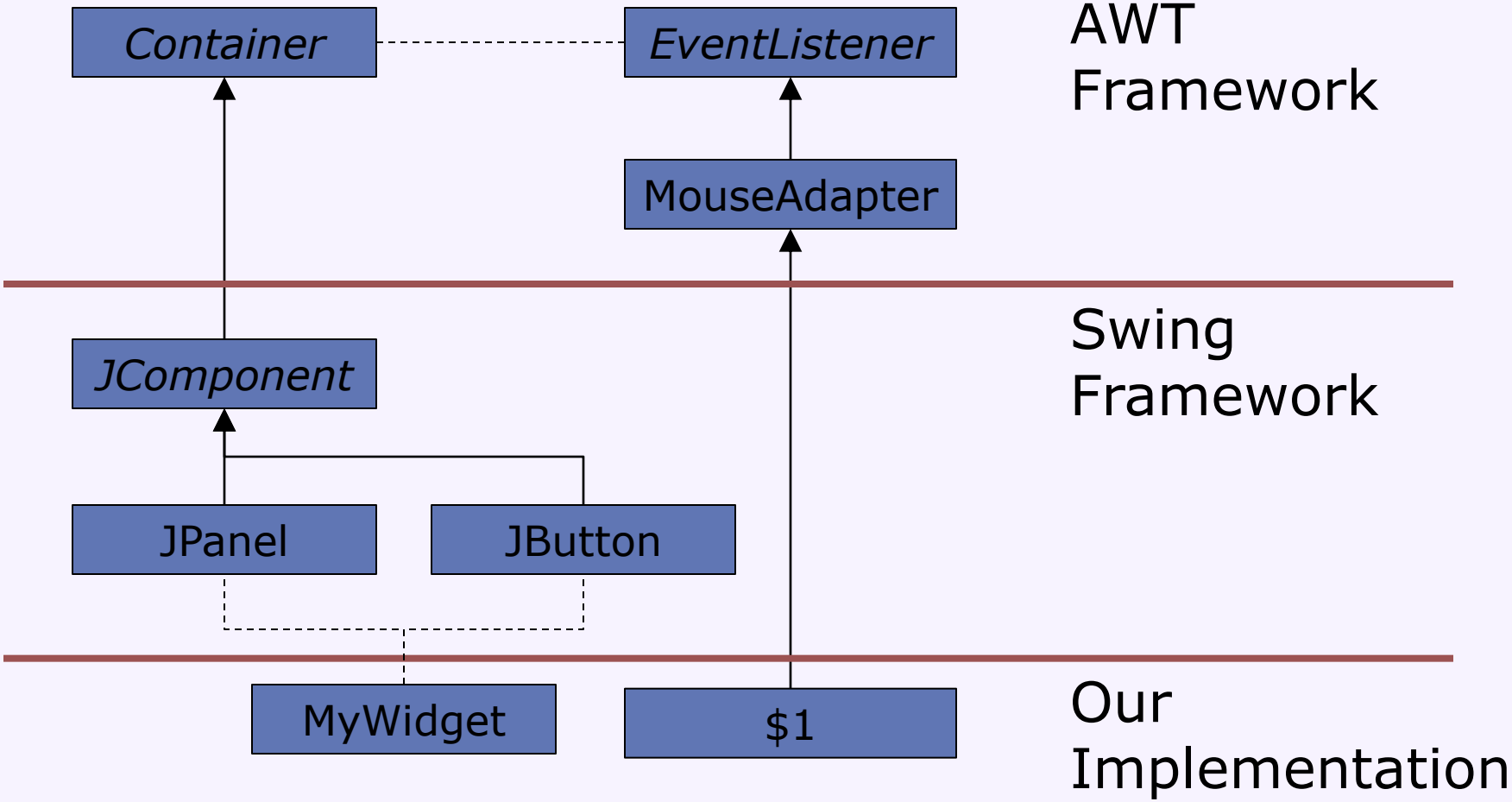
Swing: Events

```
// create an anonymous MouseAdapter, which extends
// the MouseListener class
button.add(new MouseAdapter () {
   public void mouseClicked(MouseEvent e) {
      System.err.println("You clicked me! " +
         "Do it again!")
   }
});
```



But this extending a class to add custom behaviors, right?

Where is the boundary?



Swing: Custom Components (Reuse)

```
public MyWidget extends JPanel {


    public MyWidget(int param) {
        setLayout(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints();
        …
        add(label, c);
        add(textfield, c);
        add(button, c);
    }
    public void setParameter(int param) {
        // update the widget, as needed
    }
}
```

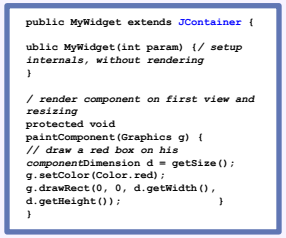
Swing: Custom Components

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

General Distinction: Library vs. Framework

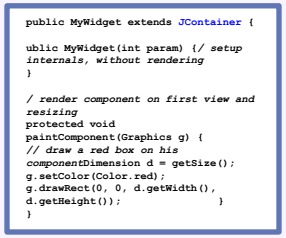
```
public MyWidget extends JContainer {

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



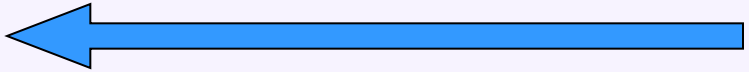
Library

user interacts



```
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



Framework

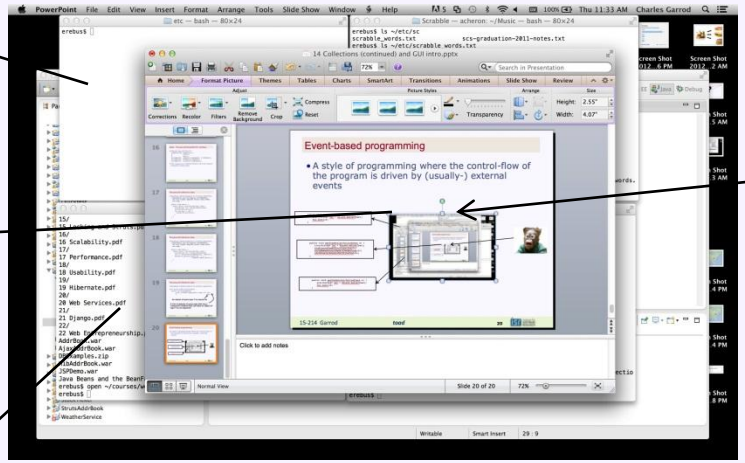
Event-based programming

- A style of programming where the control-flow of the program is driven by (usually-) external events

```
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(42)
}
```

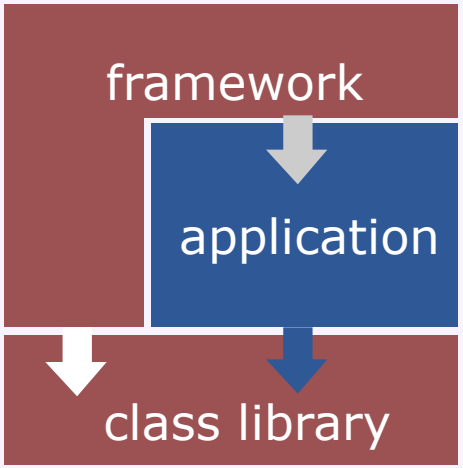
```
public void performAction(ActionEvent e) {
    bigBloatedPowerPointFunction(e);
    withANameSoLongIMadeItTwoMethods(e);
    yesIKnowJavaDoesntWorkLikeThat(e);
}
```

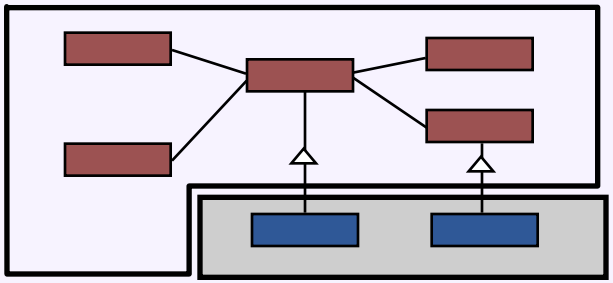
```
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(40)
}
```



OO Frameworks (credit: Erich Gamma)

- A customizable set of cooperating classes that defines a reusable solution for a given problem
  - defines key abstractions and their interfaces
  - object interactions & invariants
  - flow of control
    - override and be called
  - defaults
- Reuse
  - reuse of design and code
  - reuse of a macro architecture
- Framework provides architectural guidance





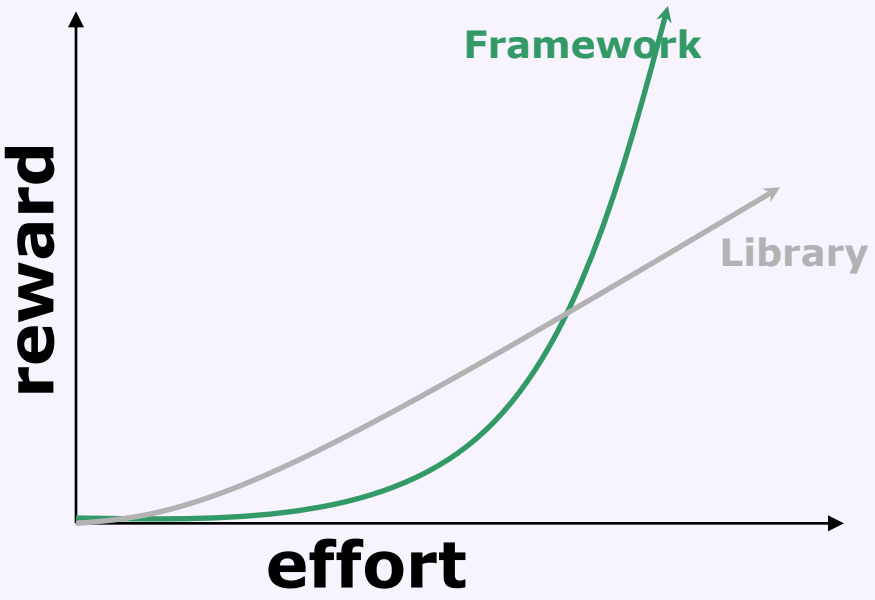
reusing a framework

Platform / Software Ecosystem

- Hardware/software environment (frameworks, libraries) for building applications
- Ecosystem: Interaction of multiple parties on a platform, third-party contributions, co-dependencies, …
  - Typically describes more business-related and social aspects

Learning a Framework

- Documentation
- Tutorials, Wizards, and Examples
- Communities – email lists and forums
- Other Client Applications / Plug-ins



Designing a Framework

- Difficult Task – Requires Experience
- Once Designed, Little Place for Change

- Key Decision:
  Separating Common from Variable Parts
- Identify hot spots vs cold spots

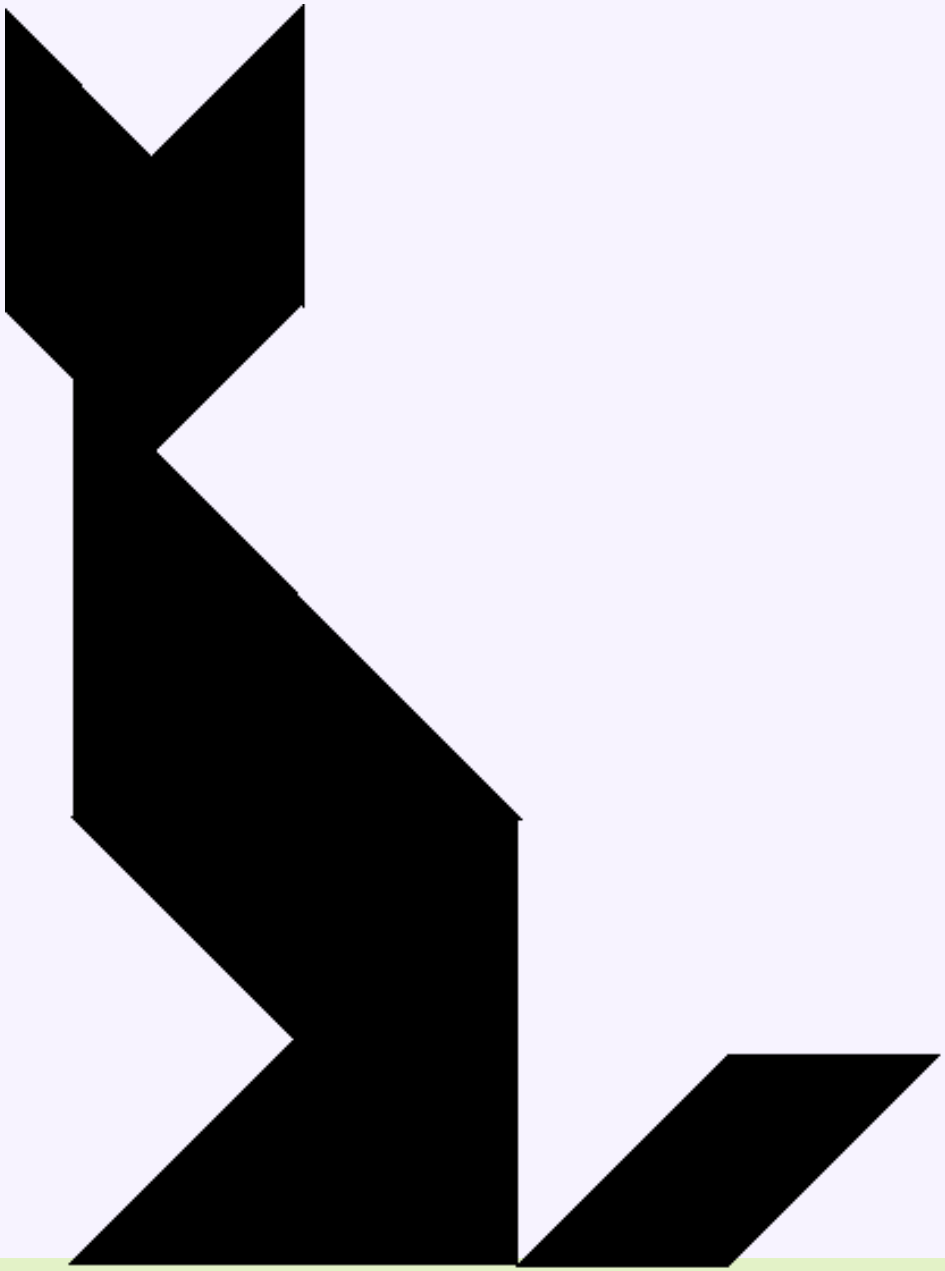
- Too Few Extension Points: Limited to a Narrow Class of Users
- Too Many Extension Points: Hard to Learn, Slow
- Too Generic: Little Reuse Value

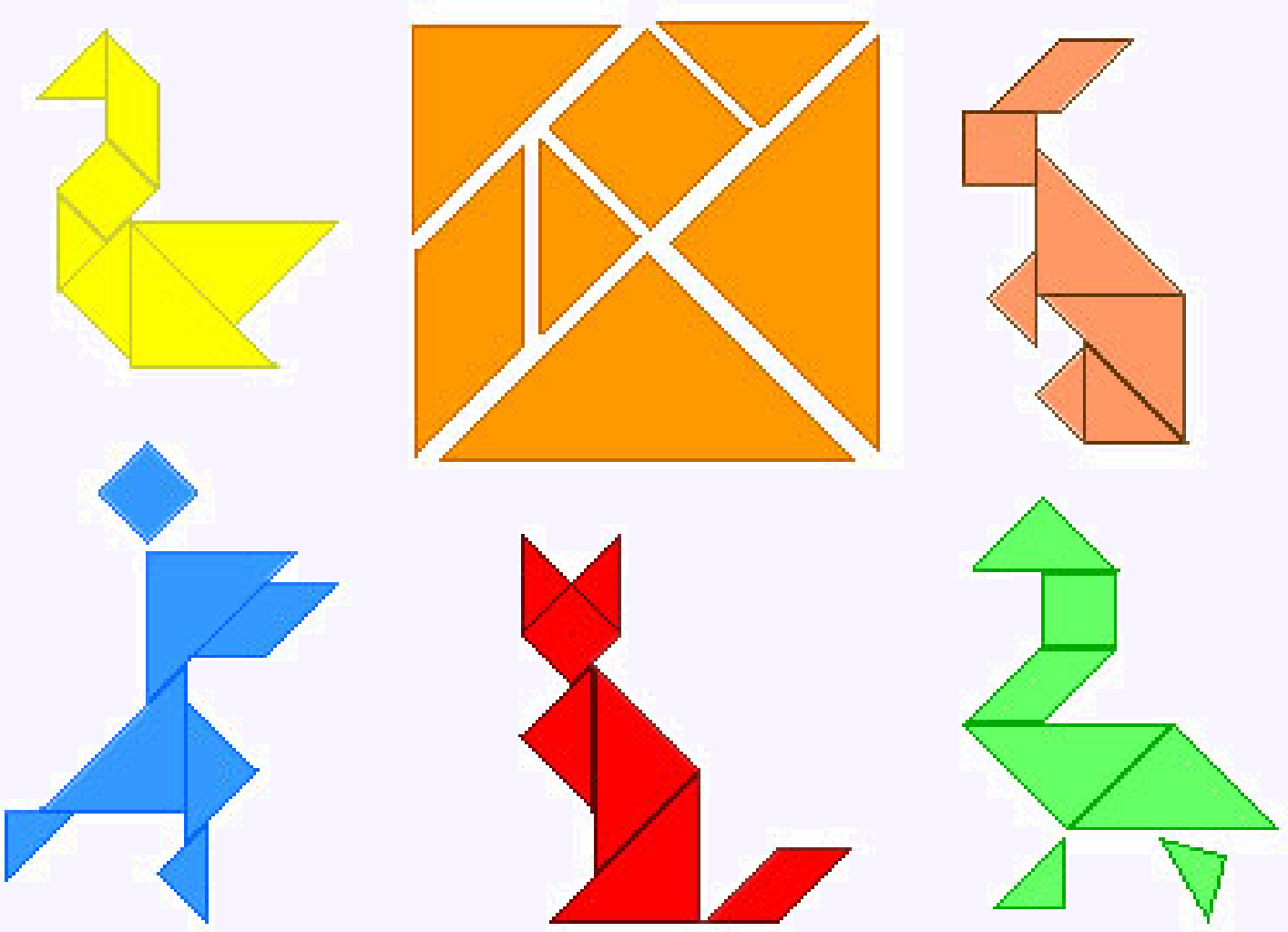
Use vs. Reuse Dilemma

- (for Frameworks, Libraries, Components, …)
- Large rich components are very useful, but rarely fit
- Small or extremely generic components often fit, but provide little benefit

“maximizing reuse minimizes use”

C. Szyperski





(Tangram)

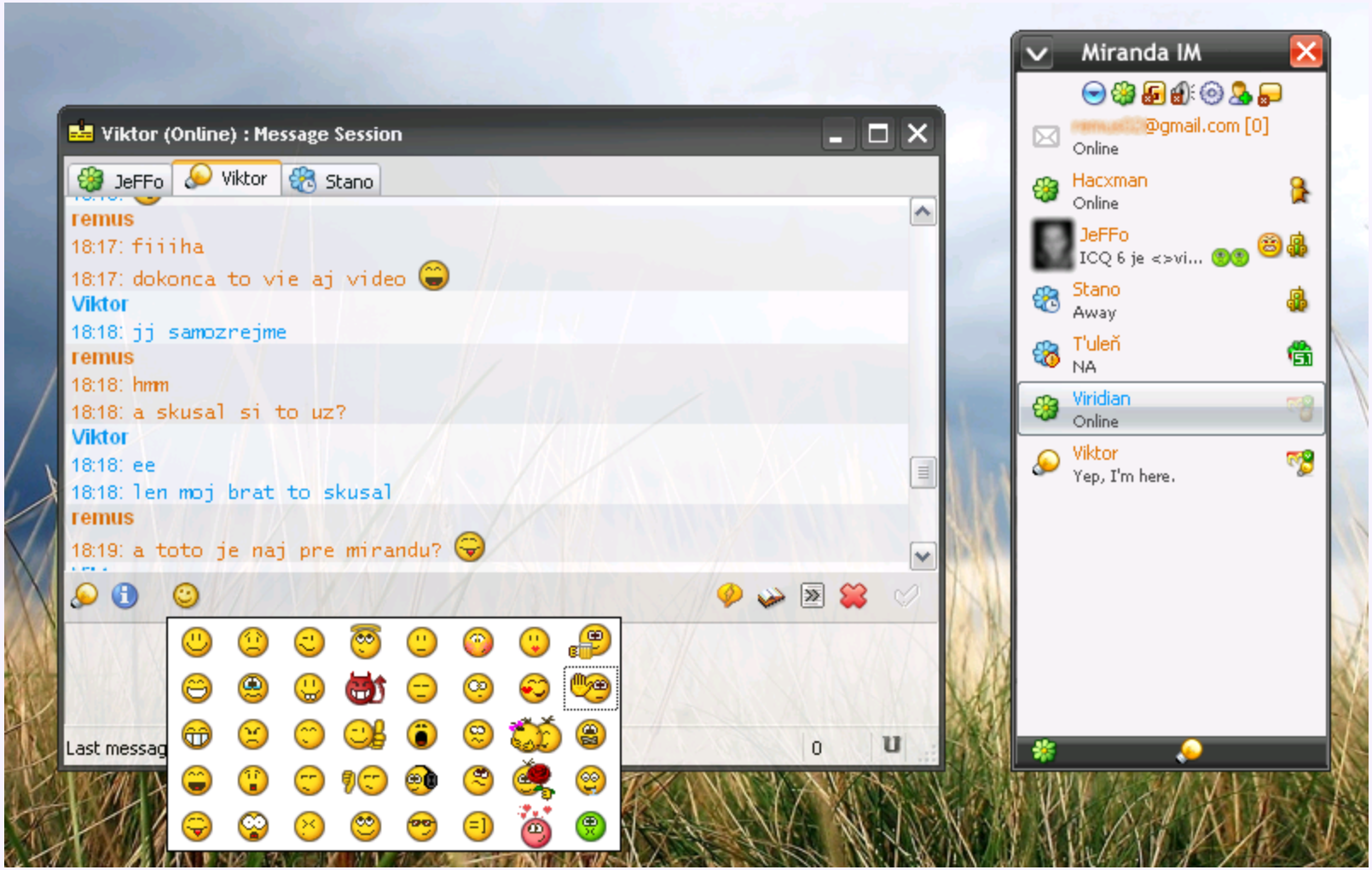
Domain Engineering

- Think of possible 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")

- Eclipse Policy:
  - "Internal" interfaces at first (unsupported, may change)
  - Public stable extension points only with at least two "customers"

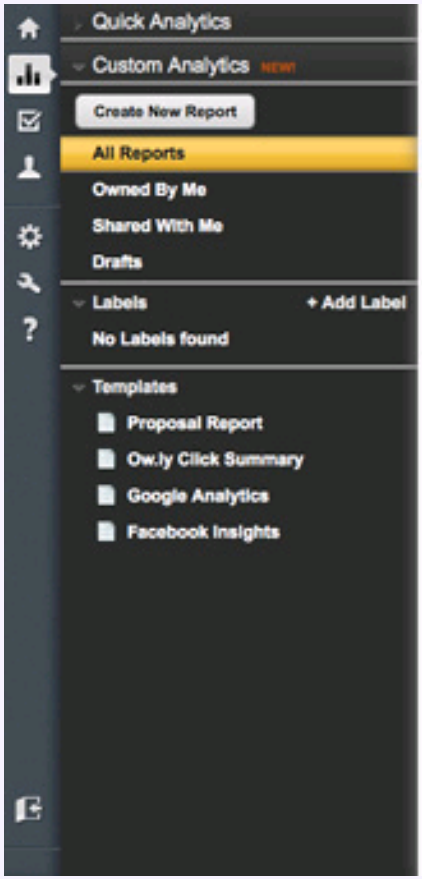
Framework Design Exercises

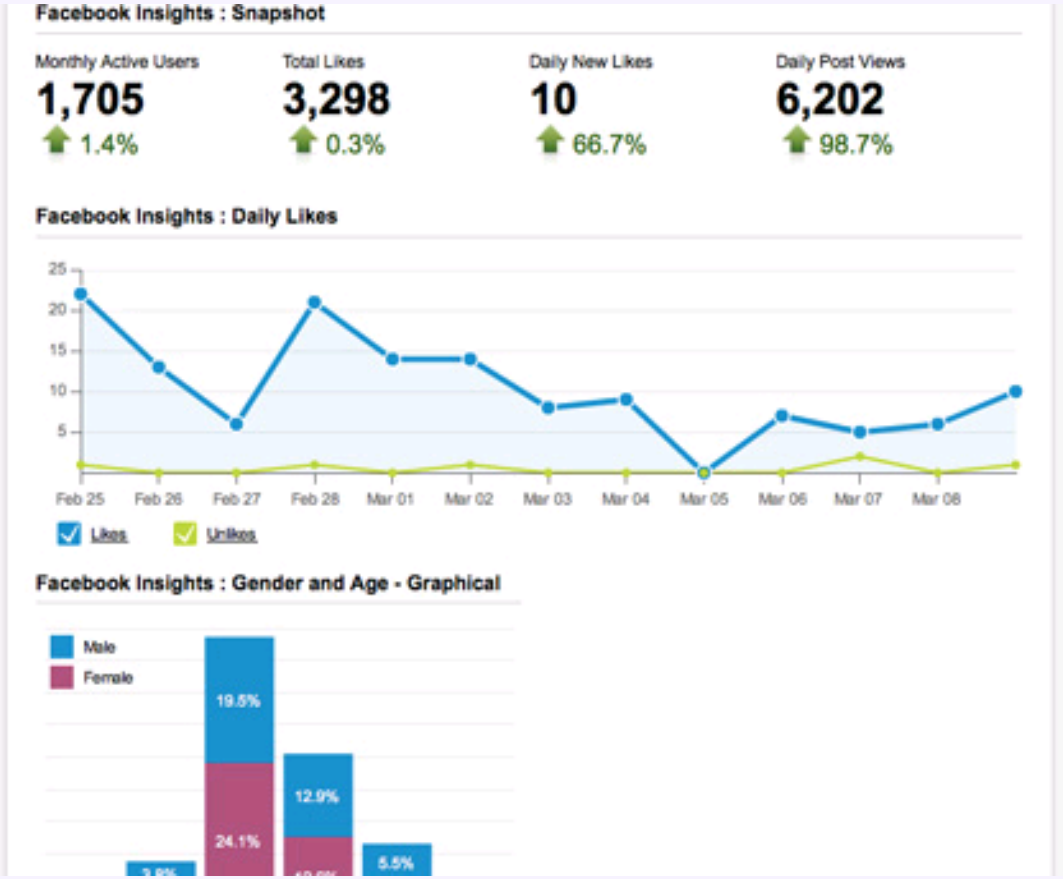
Framework Design Exercises



Framework Design Exercises

Framework Design Exercises





Framework Implementation

- After identifying common and variable parts
- Implement common parts
- Provide plug-in interface/extension/callback mechanisms for variable parts
  - Use design patterns: Strategy, Decorator, Observer, Command, Template Method, Factories …

Evolution: Extract Interface from Class (credit: Erich Gamma)

- JHotDraw defines framework abstractions as interfaces
- 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

Changing Frameworks

```
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 Bor
                contentPane.setBorder(new BevelBorder(B
                JButton button = new JButton();
                if (plugin != null)
                        button.setText(plugin.getButtonT
                else
                        button.setText("ok");
                contentPane.add(button, BorderLayout.EAST);
                textfiel
                if (plug

                textfiel
                content
                if (plug
```

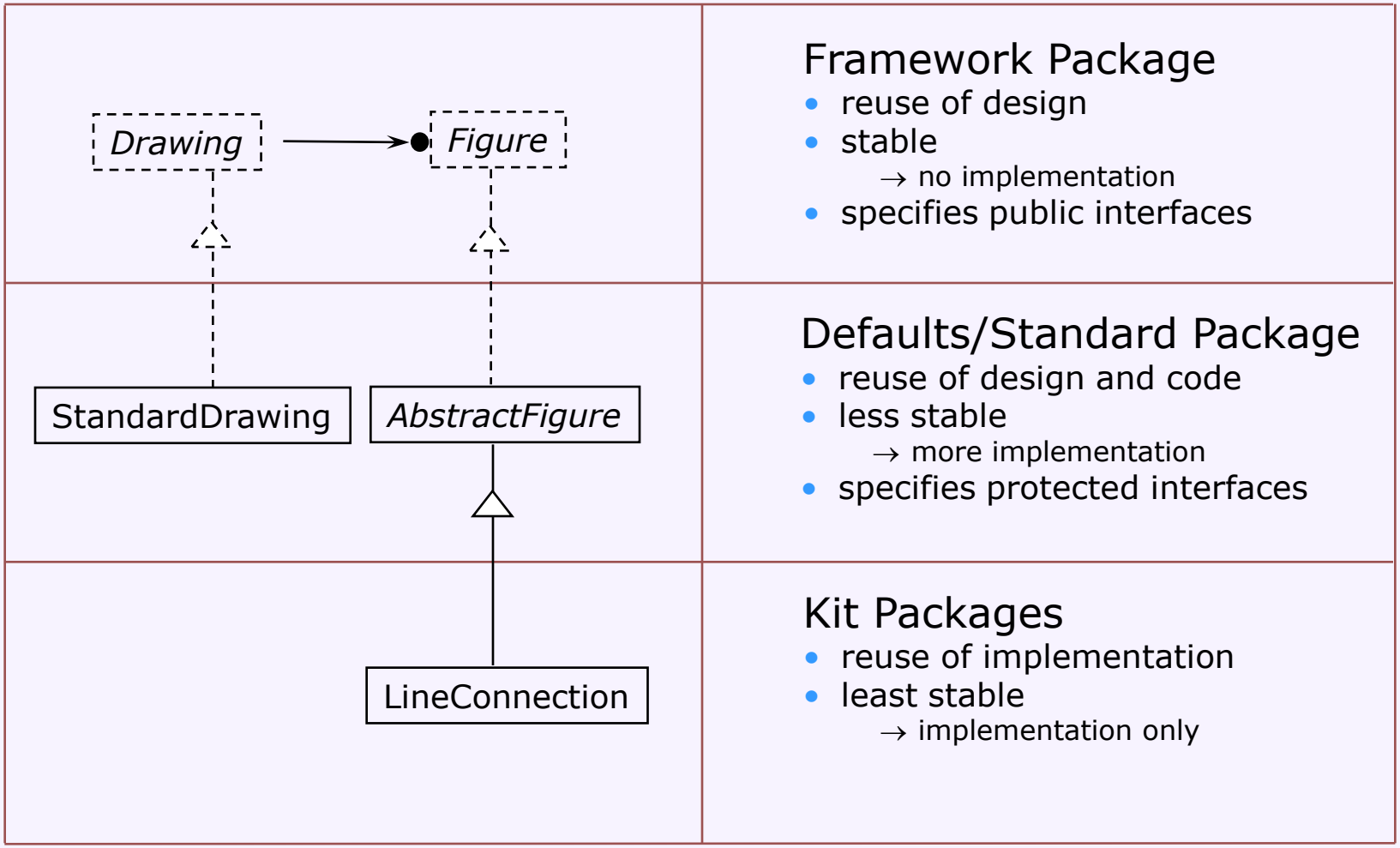
```
public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked() ;
    void setApplication(Application app);
}
```

```
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() {
                                                (null, "The result of "
                                                nput() + " is "
                                                ation.getText())); }
                                                ) { return "My Great Calculator"; }
```

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

Consider adding an extra method
Many changes require changes to *all* plug-ins

Framework Layering (credit: Erich Gamma)



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?

Summary

- Reuse and Variations 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

- Later: Software product lines