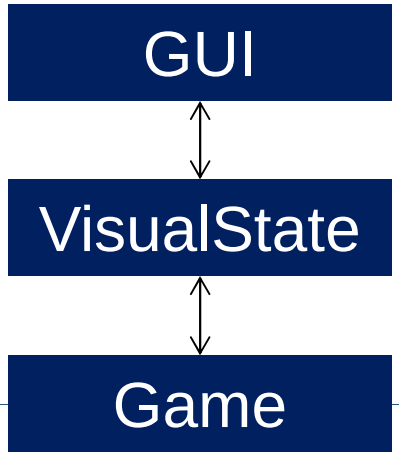
Design Problem of the Day

- Consider the Scrabble game
  - The game has simple actions
    - You play a word and it is scored
  - The GUI is more complex
    - You choose tiles one by one and place them
- How to resolve?

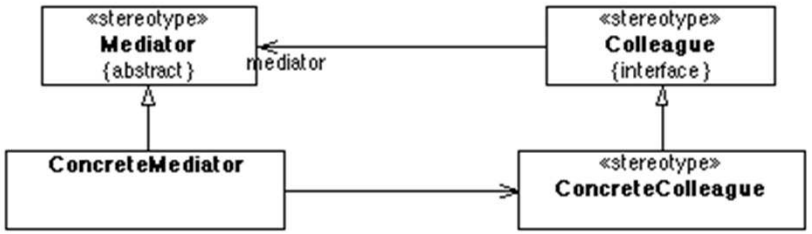
Design Problem of the Day

- Consider the Scrabble game
  - The game has simple actions
    - You play a word and it is scored
  - The GUI is more complex
    - You choose tiles one by one and place them
- How to resolve?
  - Have the game keep track of temporary tile placement
  - Have the GUI keep track of temporary tile placement
  - Use a ***mediator***



The Mediator Design Pattern

- Applicability
  - A set of objects that communicate in well-defined but complex ways
  - Reusing an object is difficult because it communicates with others
  - A behavior distributed between several classes should be customizable without a lot of subclassing



- Consequences
  - Avoids excessive subclassing to customize behavior
  - **Decouples colleagues, enhancing reuse**
  - Simplifies object protocols: many-to-many to one-to-many
  - **Abstracts how objects cooperate into the mediator**
  - Centralizes control
    - Danger of mediator monolith

Design Advice of the Day

Why is **`instanceof`** bad?

- Often OK if you are testing against just one thing
  - Especially if that thing is an interface
    - i.e. an abstract concept, not an implementation
- Problems come when testing against several things:
  - You might **forget** one

```
if (x instanceof Rabbit) …
else if (x instanceof Fox) …
// oops, forgot the Grass!
```

Design Advice of the Day

Why is **`instanceof`** bad?

- Often OK if you are testing against just one thing
  - Especially if that thing is an interface
    - i.e. an abstract concept, not an implementation
- Problems come when testing against several things:
  - You might **forget** one
  - A sequence of instanceofs is **inefficient**
    - Better to do one dispatch than several instanceofs

Design Advice of the Day

Why is **`instanceof`** bad?

- Often OK if you are testing against just one thing
  - Especially if that thing is an interface
    - i.e. an abstract concept, not an implementation
- Problems come when testing against several things:
  - You might **forget** one
  - A sequence of instanceofs is **inefficient**
  - instanceof-based code is not **extensible**
    - If a new class comes, you have to change code everywhere

Example of instanceof problems

```
class Animal { … }
class Rabbit extends Animal
{ … }
class Fox extends Animal
{ … }
```

```
void move(Animal a) {
  if (a instanceof Rabbit)
      // move like a Rabbit
  if (a instanceof Fox)
      // move like a Fox
}
void eat(Animal a) {
  if (a instanceof Rabbit)
      // eat like a Rabbit
  if (a instanceof Fox)
      // eat like a Fox
}
```

Example of instanceof problems

```
abstract class Animal { … }
class Rabbit extends Animal
{ … }
class Fox extends Animal
{ … }
class Wolf extends Animal
{ … }
```

Multiple files must change
Easy to forget some changes

```
void move(Animal a) {
  if (a instanceof Rabbit)
      // move like a Rabbit
  if (a instanceof Fox)
      // move like a Fox
  if (a instanceof Wolf)
      // move like a Wolf
}
void eat(Animal a) {
  if (a instanceof Rabbit)
      // eat like a Rabbit
  if (a instanceof Fox)
      // eat like a Fox
  if (a instanceof Wolf)
      // eat like a Wolf
}
```

Alternative #1 to instanceof

```
interface Animal {
  void move();
  void eat();
}
class Rabbit extends Animal {
  void move() { /* move like a Rabbit */}
  void eat() { /* eat like a Rabbit */}
}
class Fox extends Animal {
  void move() { /* move like a Fox */}
  void eat() { /* eat like a Fox */}
}
```

```
class Wolf extends Animal {
  void move() { /* move like a Wolf */}
  void eat() { /* eat like a Wolf */}
}
```

- Now extension only affects one file.
- Java's type system ensures we can't forget operations

Alternative #2: the Visitor pattern

```
interface Animal {
  void visit(AnimalVisitor v);
}
interface AnimalVisitor {
  void visitRabbit(Rabbit r);
  void visitFox(Fox f);
}
class Rabbit extends Animal {
  void visit(AnimalVisitor v) {
    v.visitRabbit(this);
  }
}
class Fox extends Animal {
  void visit(AnimalVisitor v) {
    v.visitFox(this);
  }
}
```

```
class MoveVisitor implements Visitor {
  void visitRabbit(Rabbit r) {
    /* move like a Rabbit */ }
  void visitFox(Fox f) {
    /* move like a Fox */ }
}
```

Alternative #2: the Visitor pattern

```
interface Animal {
  void visit(AnimalVisitor v);
}
interface AnimalVisitor {
  void visitRabbit(Rabbit r);
  void visitFox(Fox f);
}
class Rabbit extends Animal {
  void visit(AnimalVisitor v) {
    v.visitRabbit(this);
  }
}
class Fox extends Animal {
  void visit(AnimalVisitor v) {
    v.visitFox(this);
  }
}
```

```
class MoveVisitor implements Visitor {
  void visitRabbit(Rabbit r) {
    /* move like a Rabbit */ }
  void visitFox(Fox f) {
    /* move like a Fox */ }
}
class EatVisitor implements Visitor {
  void visitRabbit(Rabbit r) {
    /* eat like a Rabbit */ }
  void visitFox(Fox f);
    /* eat like a Fox */ }
}
```

- Good when you add operations frequently

Alternative #2: the Visitor pattern

```
interface Animal {
  void visit(AnimalVisitor v);
}
interface AnimalVisitor {
  void visitRabbit(Rabbit r);
  void visitFox(Fox f);
  void visitWolf(Wolf w);
}
class Rabbit extends Animal { …}
class Fox extends Animal { …}
class Wolf extends Animal {
  void visit(AnimalVisitor v) {
    v.visitWolf(this);
  }
}
```

```
class MoveVisitor implements Visitor {
  void visitRabbit(Rabbit r) { … }
  void visitFox(Fox f) { … }
  void visitWolf(Wolf w) {
    /* move like a Wolf */ }
}
class EatVisitor implements Visitor {
  void visitRabbit(Rabbit r) { … }
  void visitFox(Fox f) { … }
  void visitWolf(Wolf w) {
    /* eat like a Wolf */ }
}
```

- Bad when you add new classes frequently

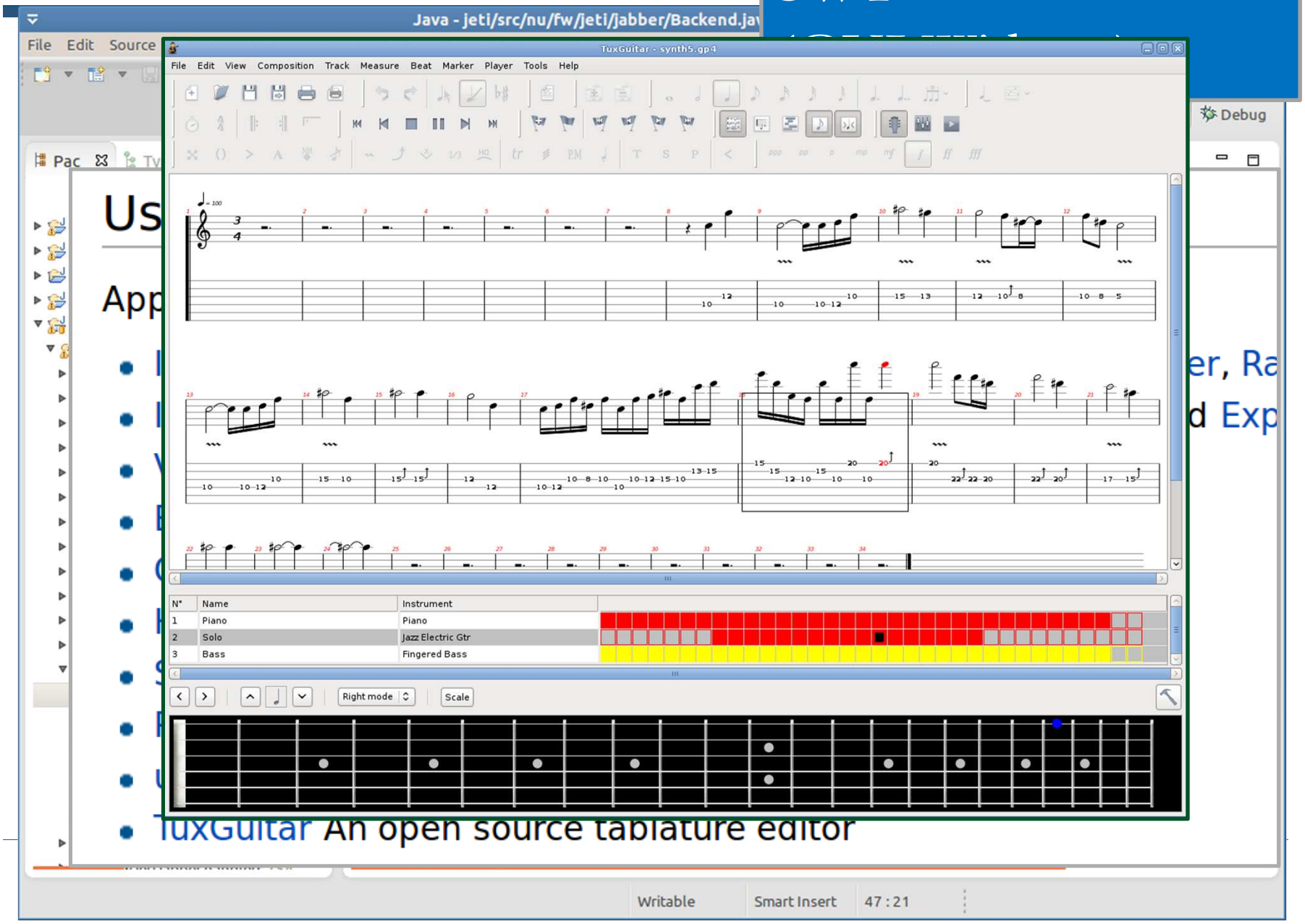
Frameworks

15-214: Principles of Software Construction: Objects, Design, and Concurrency

Some material from Ciera Jaspan,
Bill Scherlis, Travis Breaux, and Erich Gamma

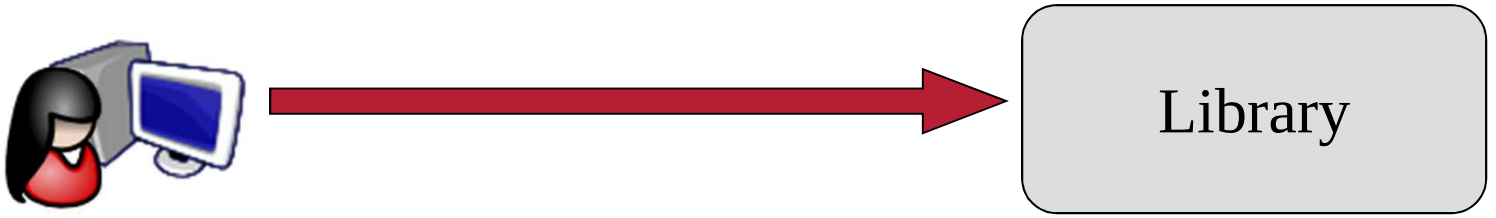
Reuse and Variations

SWT



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



Math

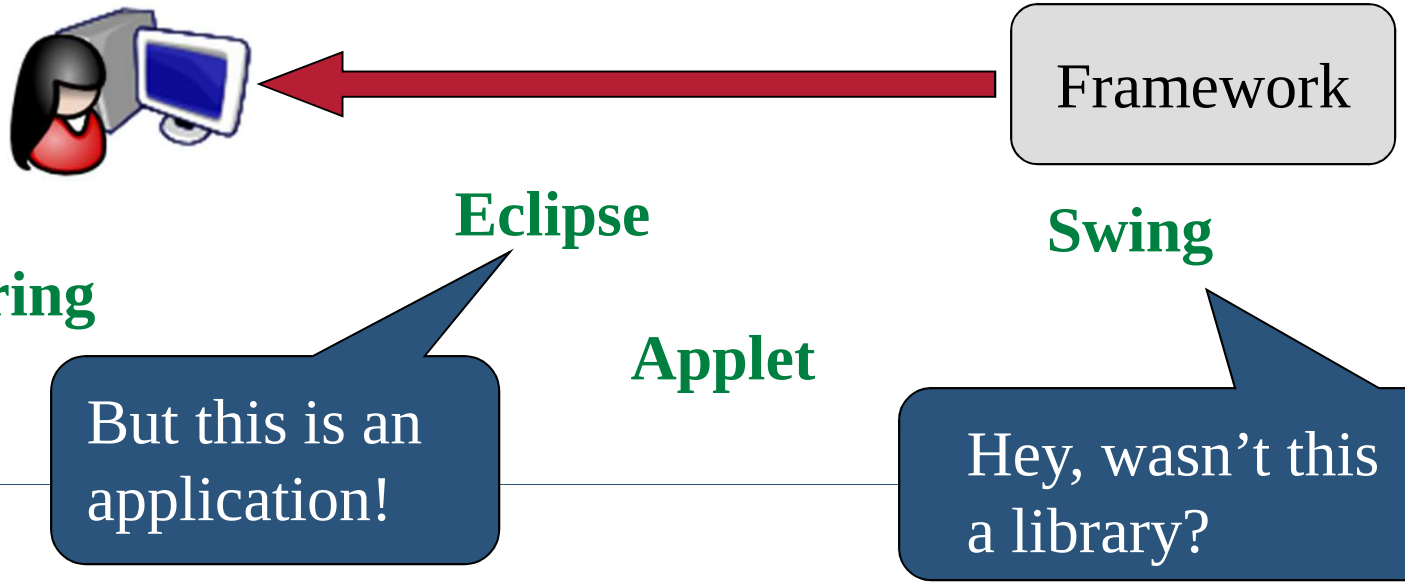
Collections

I/O

Swing

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."



More terms

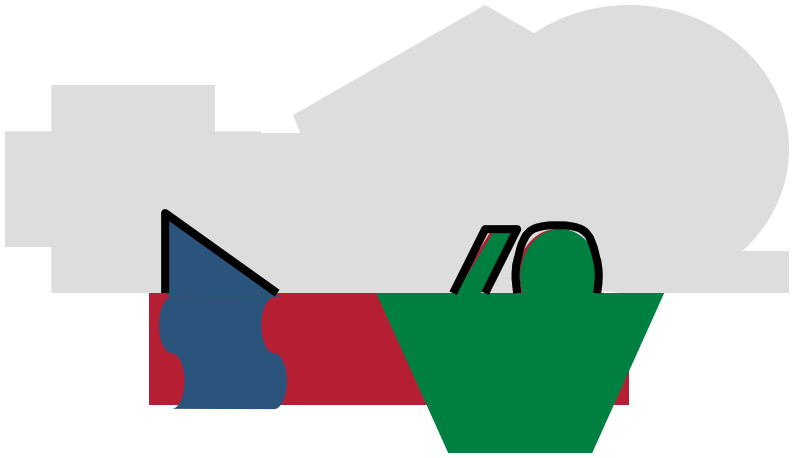
- API: Application Programming Interface, the interface of a library or framework
- Client: The code that uses an API
- Plugin: Client code that customizes a framework
- Extension point: A place where a framework supports extension with a plugin

More terms

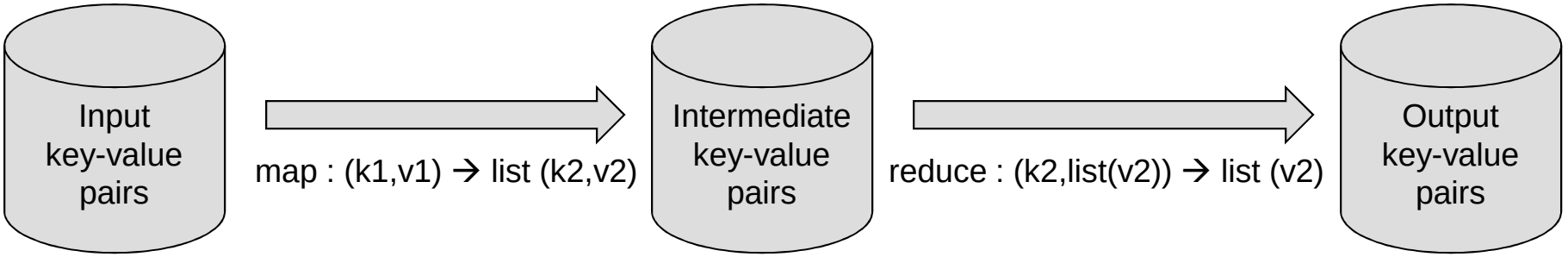
- Protocol: The expected sequence of interactions between the API and the client
- Callback: A plugin method that the framework will call to access customized functionality
- Lifecycle method: A callback method of an object that gets called in a sequence according to the protocol and the state of the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

Using an API

- Like a partial design pattern
- Framework provides one part
- Client provides the other part
- Very common for plugin trees to exist
- Also common for two frameworks to work better together

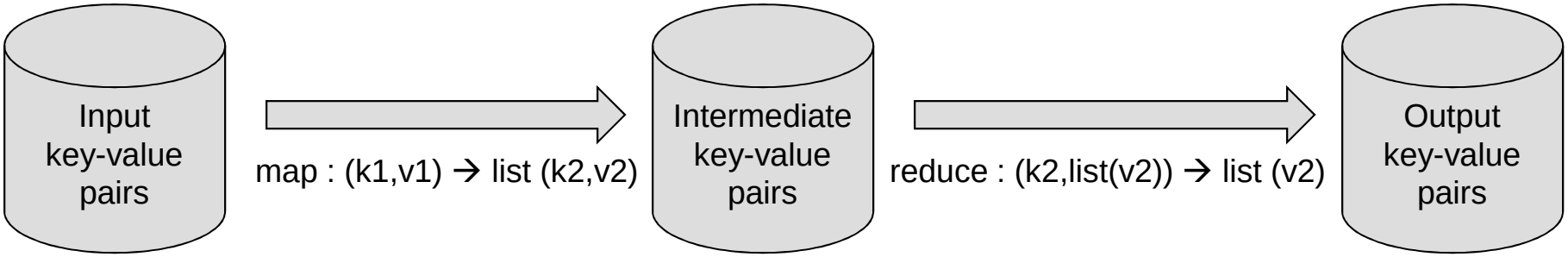


Google's Map-Reduce



- 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/
  - http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/Mapper.html
  - http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/Reducer.html

- The interface is richer than the obvious one!
  - Map and Reduce are separate abstractions
    - May have several maps followed by one reduce, for example
  - configure() supports setup operations, e.g. prefilling a cache
  - close() allows a job to clean up resources
  - OutputCollector supports mapping to >1 pair
  - Reporter supports incremental progress updates
    - Used to decide whether to kill a job, for example

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

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;
        publ
        prot
    }
    protected St
}
```

```
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

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

                textfield.s
                contentP
                if (plugin

                this.setCo
                …
        }
        public String getInput() {
}
```

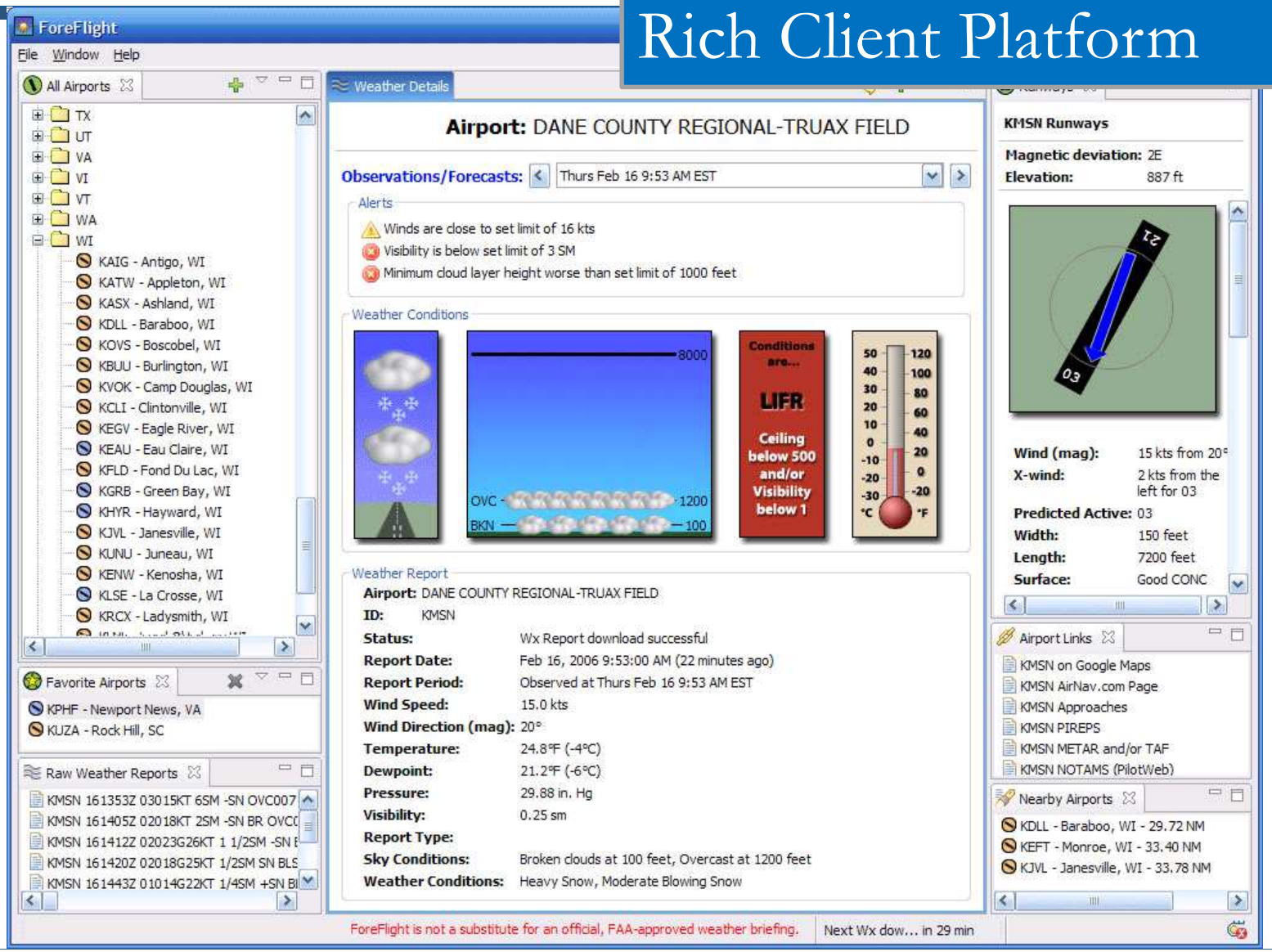
```
public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked() ;
    void setApplication(Application app);
}
```

```
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); }}
```

Eclipse as a Fwk

Eclipse
Rich Client Platform



The Golden Rule of Framework Design

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

- Discussion: how can we extend without modification?
  - Client writes main(), creates a plugin, and passes it to framework
    - See examples above
  - Framework writes main(), client passes name of plugin
    - E.g. using a command line argument or environment variable
  - Framework looks in a magic location
    - Config files or JAR files there are automatically loaded and processed

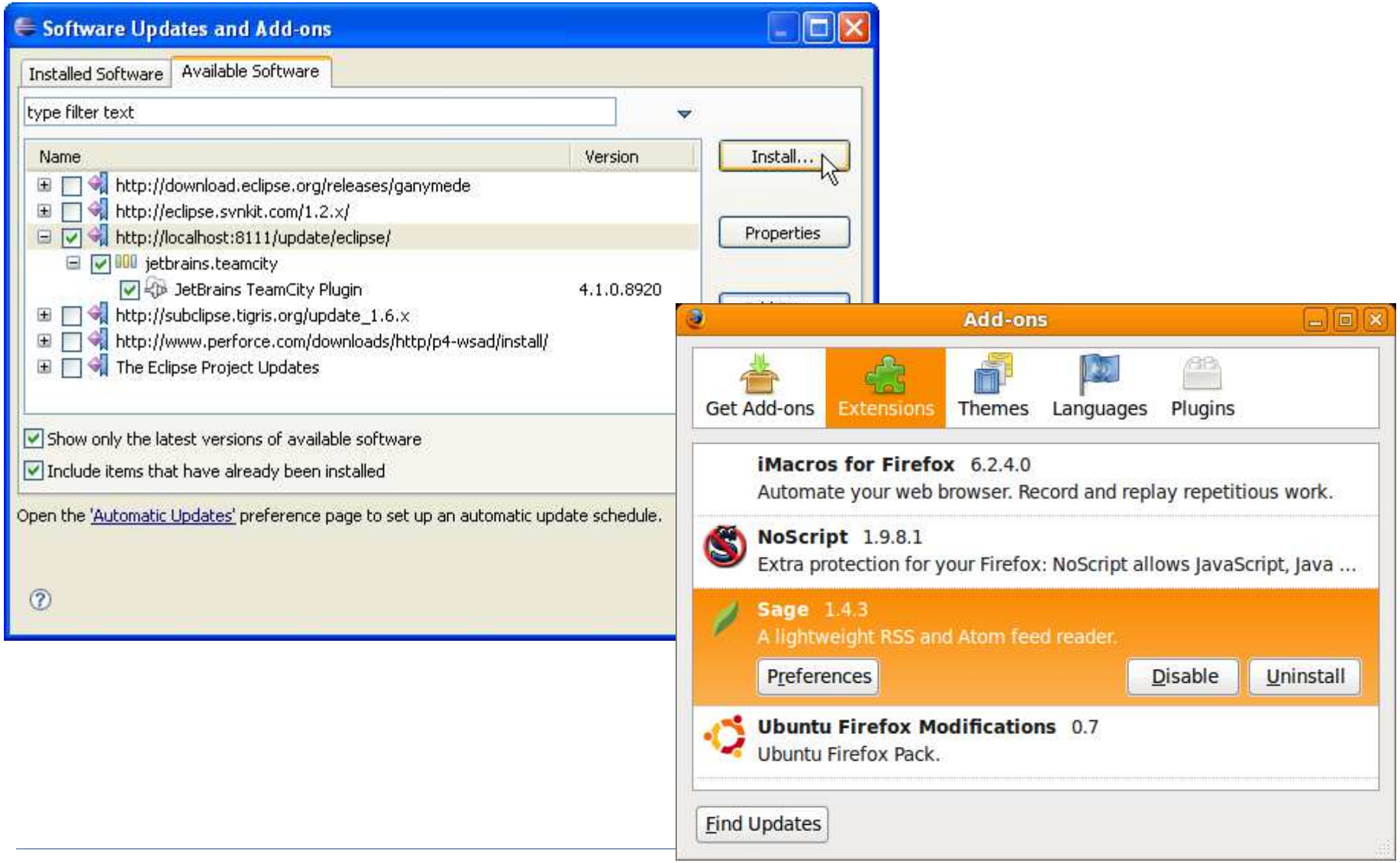
Loading Plugins using Java Reflection

```
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);
            Plugin plugin = (Plugin) pluginClass.newInstance();
            new Application(plugin).setVisible(true);
        } catch (Exception e) {
            System.out.println("Cannot load plugin " + pluginName
                + ", reason: " + e);
        }
    }
}
```

Plugin Loading using Java Reflection

```
public static void main(String[] args) {
    File config = new File(".config");
    BufferedReader reader = new BufferedReader(new FileReader(config));
    Application = new Application();
    String pluginName = null;
    while ((pluginName = 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 Frameworks

- Whitebox uses subclassing
  - Allows clients to extend every non-private/-final method
  - Need to understand when overridable methods are called
    - May need to read the implementation, if not properly documented

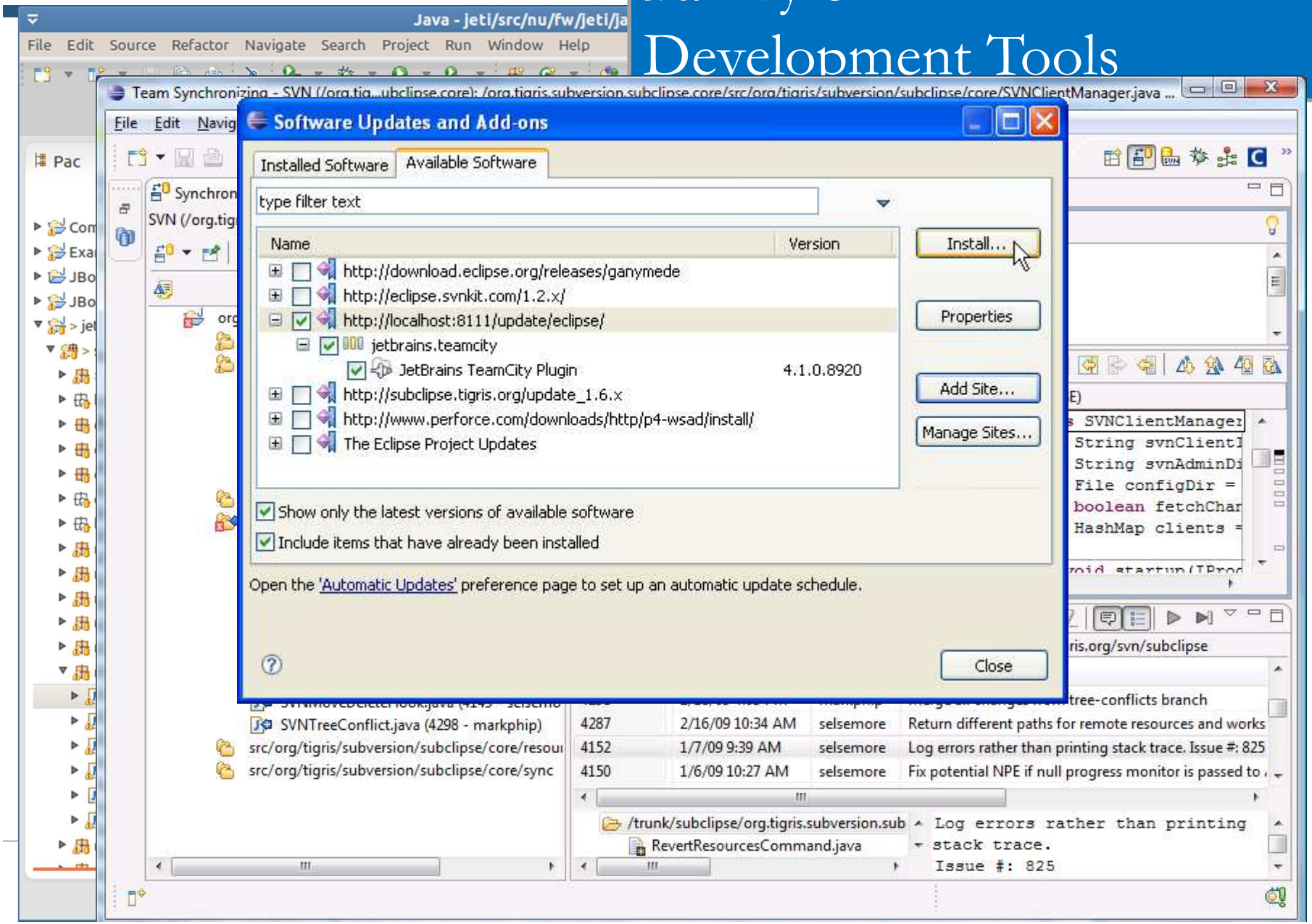
- Blackbox uses composition
  - Allows clients to extend only functionality exposed in interface
  - Only need to understand the interface

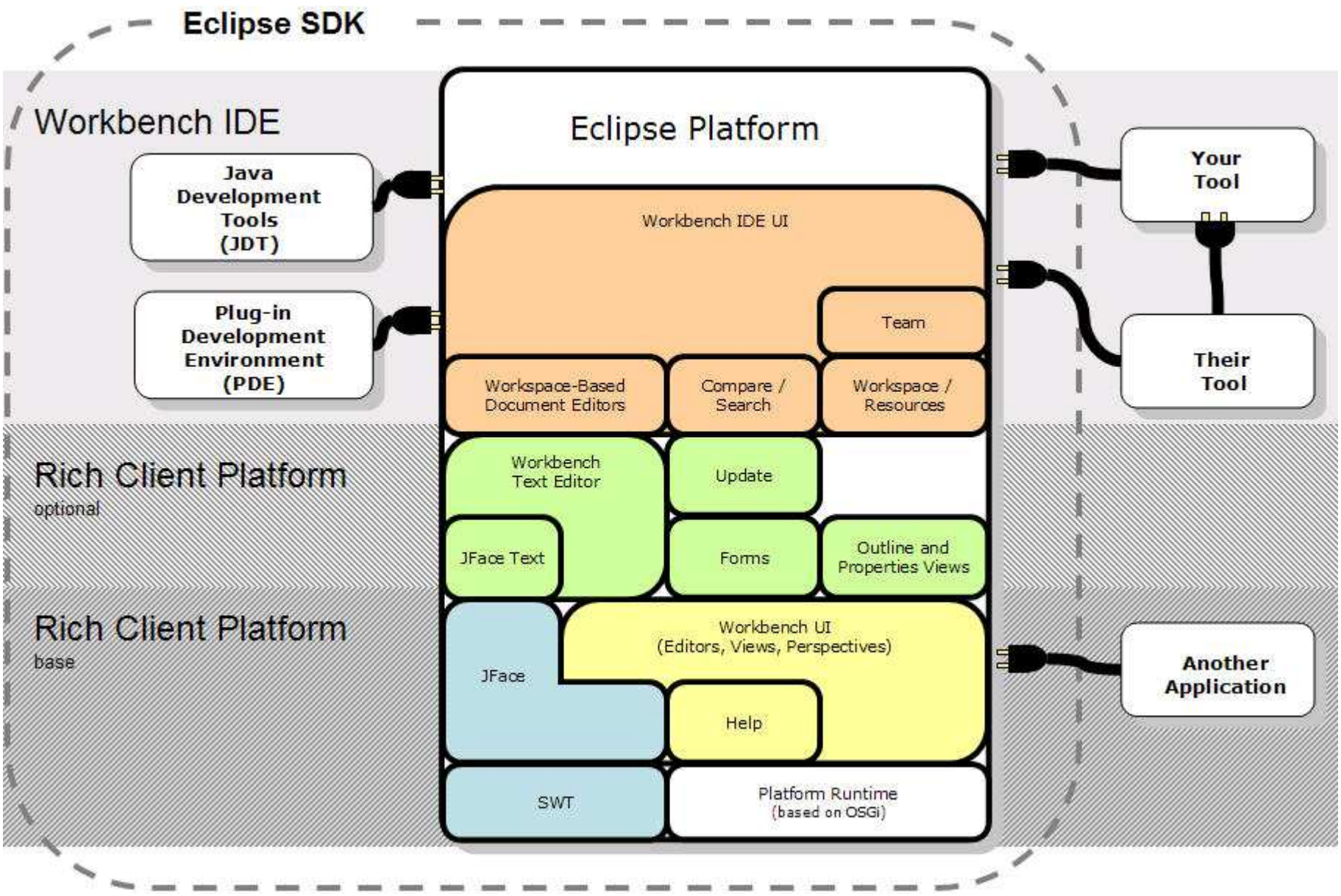
Running a Framework

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

Eclipse as a Fwk

Family of Development Tools




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

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.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

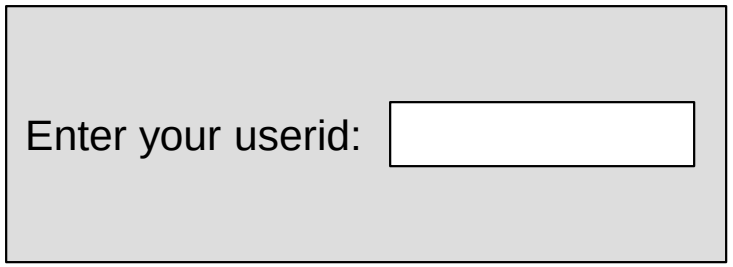
Java Swing: It's a Library!

- 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)
```

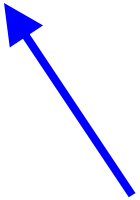


Swing: Layout Managers

```
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);
```

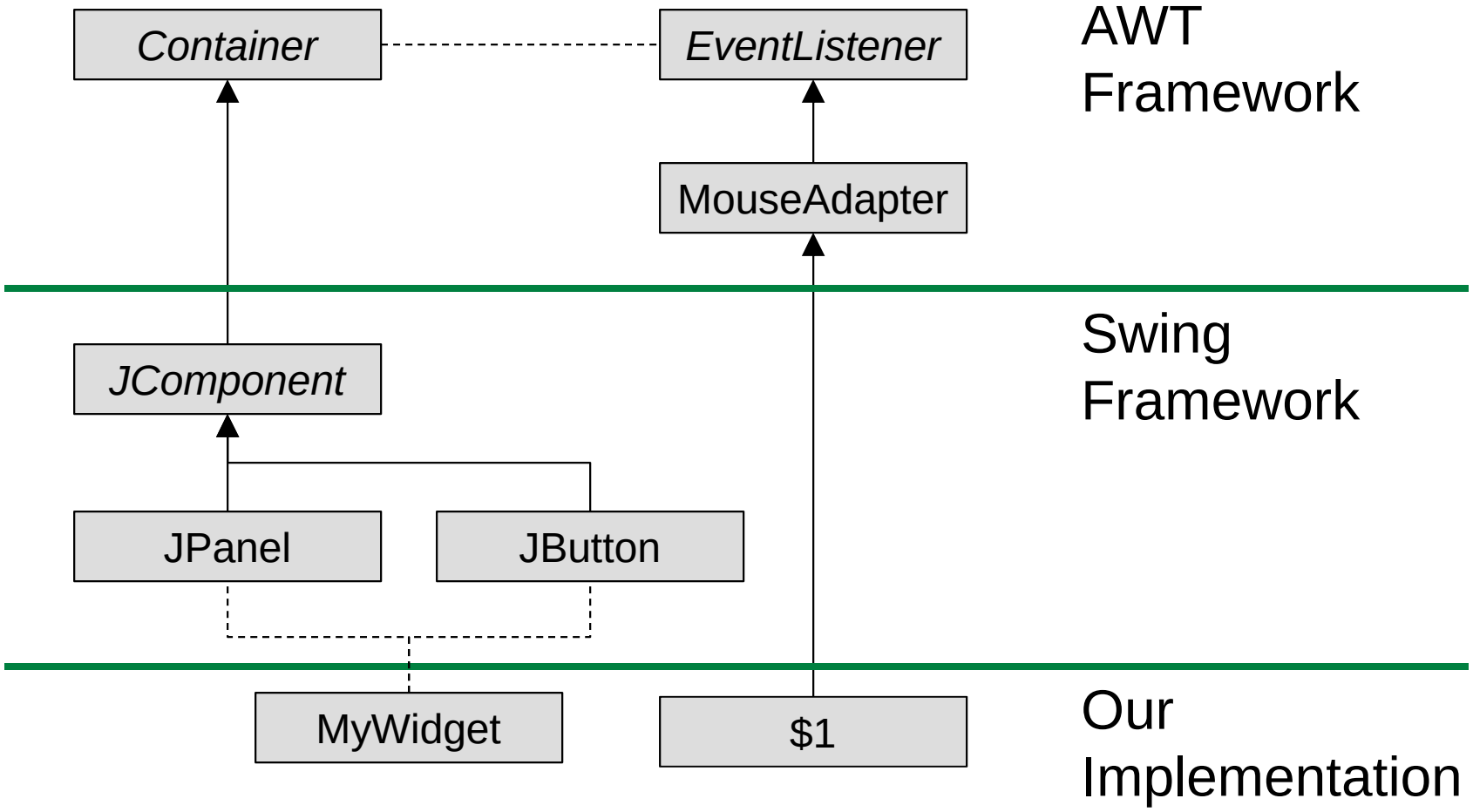
Swing: Events

```
// 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());
        }
}
```

One More Framework: Java Servlets

```
package servletDemo;
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet("/hello")
public class HelloWorld extends HttpServlet {

   @Override
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                     throws ServletException, IOException {

      PrintWriter out = response.getWriter();
      out.println("Hello World");
   }
}
```

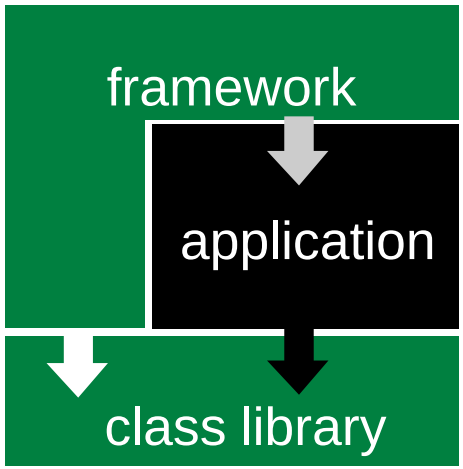
Source: http://courses.coreservlets.com/Course-Materials/csajsp2.html

Other Aspects of Java Servlets

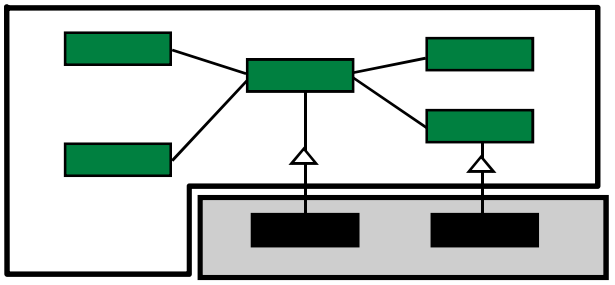
- Lifecycle methods
  - void init(ServletConfig config)
    - Sets up a Servlet object
  - void destroy()
    - Asks a Servlet object to clean itself up
- Service methods
  - doGet, doPut, doPost, doDelete, etc.
- web.xml
  - Defines mapping from pages to Servlet classes
  - Alternative to @WebServlet annotation

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

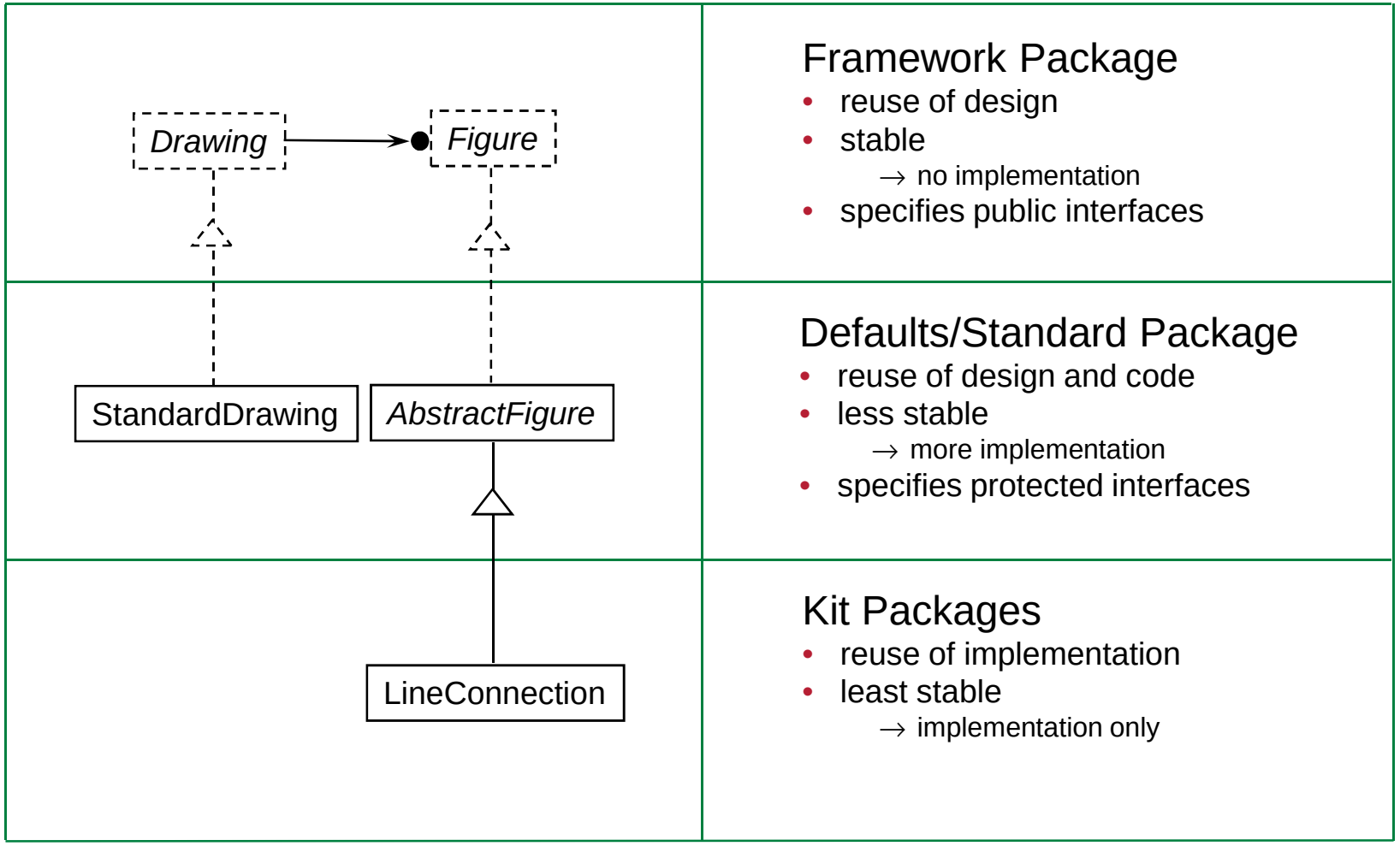


reusing a framework

Framework Challenges (credit: Erich Gamma)

- frameworks are hard to maintain
- framework enables reuse of both design and implementation
  - easy for clients to add implementation dependencies
  - "what is the framework - what is just default implementation"
- therefore:
  - separation of design from implementation
    "we believe that **interface design and functional factoring constitute the key intellectual content of software** and that they are far **more difficult to create** or re-create than code"
    -Peter Deutsch
  - late commitment to implementation
    - but, frameworks still have to work out of the box!

Framework Layering (credit: Erich Gamma)



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

Designing a Framework

- Difficult task – requires experience to do well
- 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

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"

Domain Engineering 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

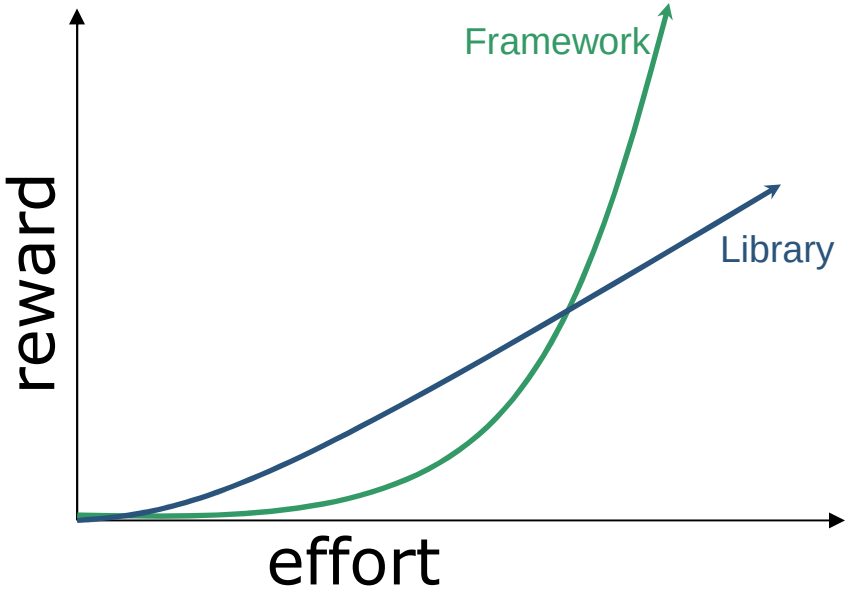
- Questions
  - What are the dimensions of variability/extensibility?
  - What interfaces would you need?
  - What are the core methods for each interface?
  - How do you set up the framework?

Framework Design

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

Getting up a framework's learning curve

- Tips on using frameworks
  - Tutorials, Wizards, and Examples
    - SourceForge, Google Code Search
  - Communities – email lists and forums
    - Eclipse.org
  - Group knowledge dispersal
    - Wiki of resources, Problem/solution log



- Common client trick: Follow the leader
  - Appropriate code from examples – find an "imputed pattern"
    - Search source code
    - Infer compatible intent
    - Identify scope (not too much, not too little)
  - Copy it
    - Tear out the app-specific logic, keep the bureaucracy
    - Insert your own logic into the reused bureaucracy
  - But there's a problem
    - Classic copy-and-paste problem – looks just like my own code
    - Design intent is lost – "my intention is to use the framework this way"

- Framework designer's conundrum: complexity vs. capability

Framework Design Advice

- It's hard to understand framework code
  - So ensure it can be understood from the documentation
  - Preconditions, postconditions, exceptions, order of calls, …
- Keep the interface narrow
  - Don't make methods package-public unless used by clients
- Help your users
  - Provide example plugins
  - Check parameters defensively for validity

Summary

- Frameworks capture reusable infrastructure
  - Hollywood principle: Don't call us, we'll call you
  - Different, and often deeper, reuse than libraries provide
    - Can often internalize qualities like reliability (cf. MapReduce)
  - Also challenging: framework impacts applications that use it
- Framework setup
  - Multiple paths: client in charge, loading plugins by reflection, …
- Designing a framework
  - Key issue is identifying important variability points
- Learning a framework
  - Leverage examples, experience of others