Principles of Software Construction: Objects, Design, and Concurrency (Part 5: Large-Scale Reuse)

Design for Large-Scale Reuse: Libraries and Frameworks

Christian Kästner Bogdan Vasilescu

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 4b due today
  - Update of design documents recommended, not required

Learning goals

- Describe example well-known example frameworks
- Know key terminology related to frameworks
- Know common design patterns in different types of frameworks
- Discuss differences in design trade-offs for libraries vs. frameworks
- Analyze a problem domain to define commonalities and extension points (cold spots and hot spots)
  - Analyze trade-offs in the use vs. reuse dilemma
- Know common framework implementation choices

This and next lecture

- Design for reuse: Libraries and frameworks
  - Motivation: reuse with variation
  - Examples, terminology
  - Whitebox and blackbox frameworks
  - Design considerations
  - Implementation details
    - Responsibility for running the framework
    - Loading plugins

Intro to Java

Git, CI

UML

GUIs

Static Analysis

Performance

GUIs

More Git

Design

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

Reuse and variation

15-214 Homework 2

Homework #2: From Calculators to Calculus

Due Thursday, February 2nd at 11:59 p.m.

In this assignment you will first writ
of a calculator and then finish a sm
at JUnit testing. This assignment l
implement common calculator opera
will write a family of classes represe
wrote in Part 1. Part 3 combines ide
where you will write a class represe
expressions can represent algebraic f
derivative of an arbitrary function, a
find zeros of an arbitrary function usi
task in which you will implement an

Your goals for this assignment are t

- Understand and apply the conc
  contracts, including an approp
- Interpret, design, and impleme
- Write unit tests and automate
- Gain experience writing tests l

15-214 Homework 3

Homework #3: From Cryptarithms to Algorithms

Due Thursday, February 9th at 11:59 p.m.

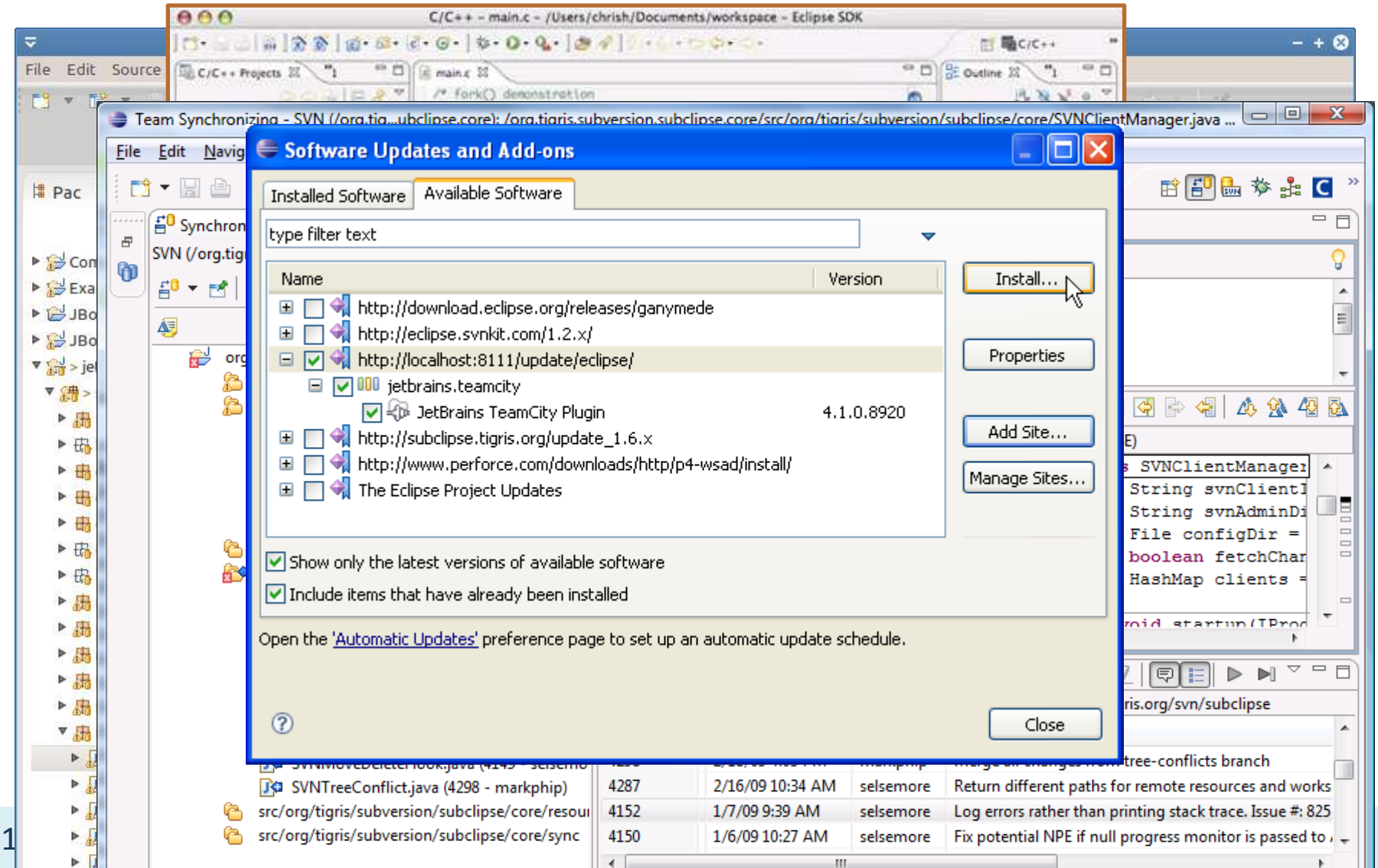
In this assignment, you will build a powerful puzzle solver using the expression evaluator library you wrote for Homework 2. The puzzle of interest is cryptarithm. A cryptarithm (or alphametic) is a puzzle where you are given an equation with letters instead of digits. For example, one famous cryptarithm published by Henry Dudeney in 1924 is:

```
  SEND
+ MORE
------
 MONEY
```

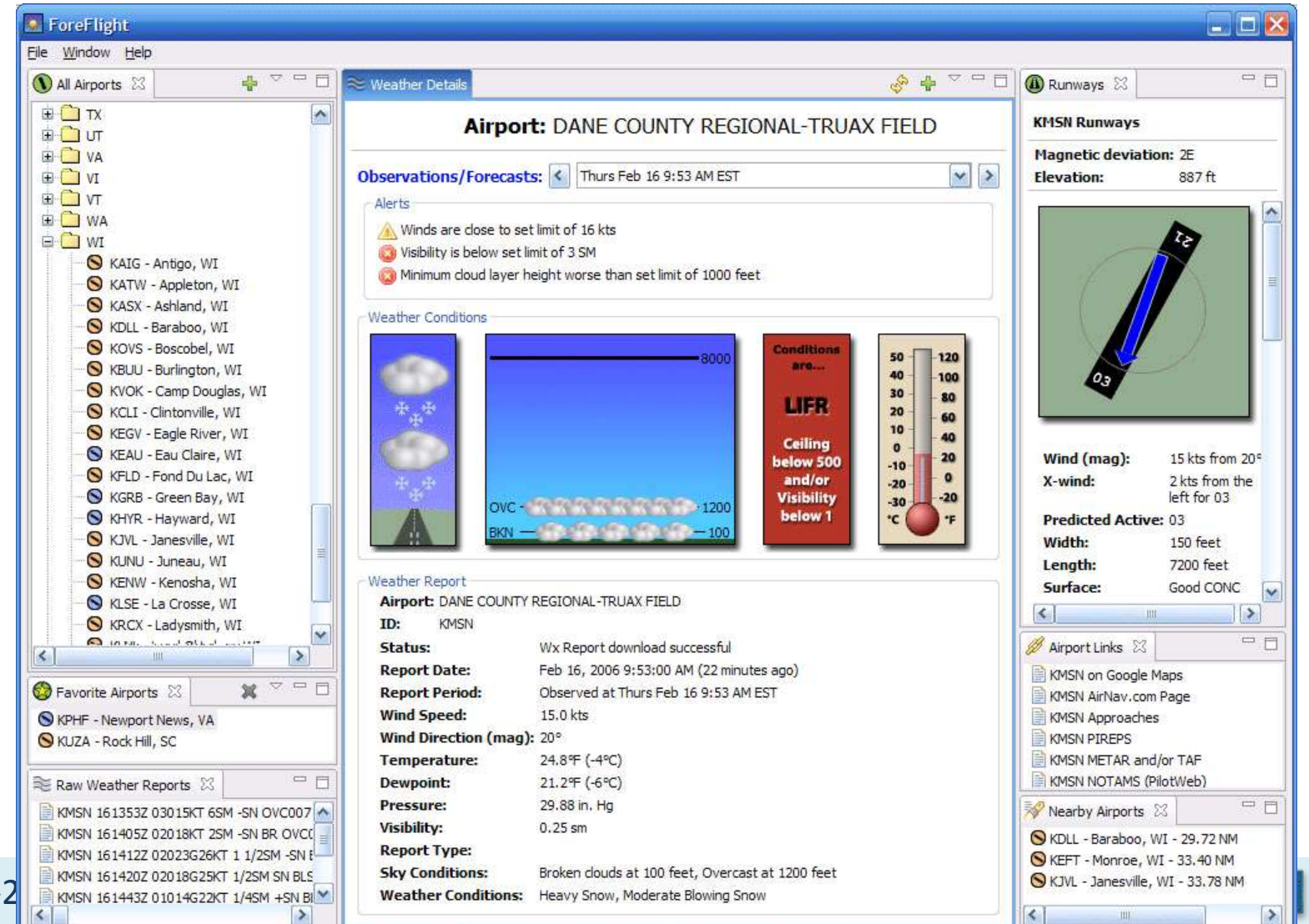
To solve a cryptarithm you must figure out which digit each letter represents. Cryptarithms typically follow standard rules: The first letter of each word (in the above example, S and M) cannot represent zero, and each letter represents a different digit. Good cryptarithms have exactly one solution.

You can use logic to solve a cryptarithm by hand, but on a computer, brute force works fine due to the small size of the search space. Because each letter represents a different digit there can be at most ten distinct letters, so there are only 10! (or 3,628,800) possible

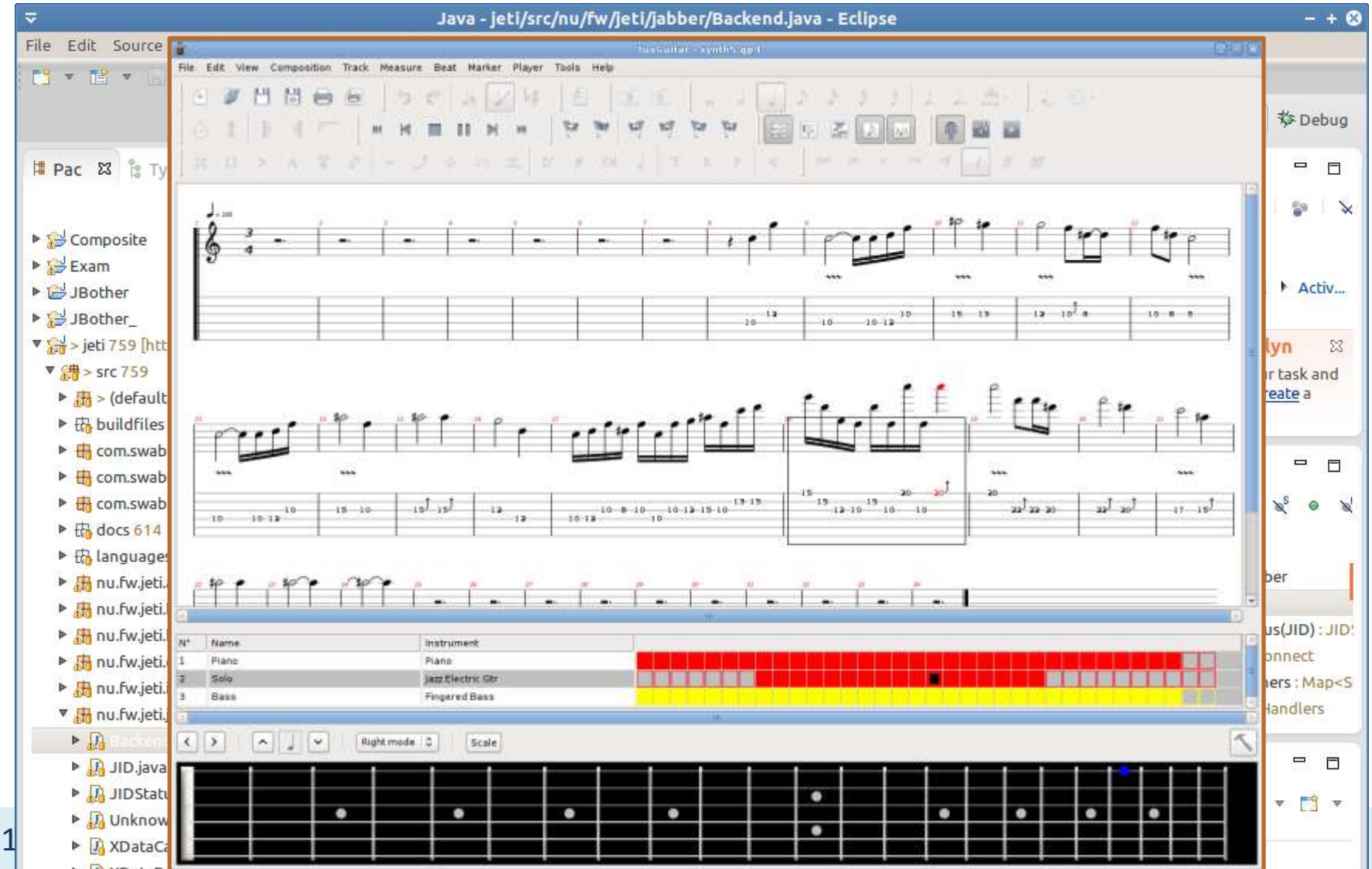
Reuse and variation: Family of development tools



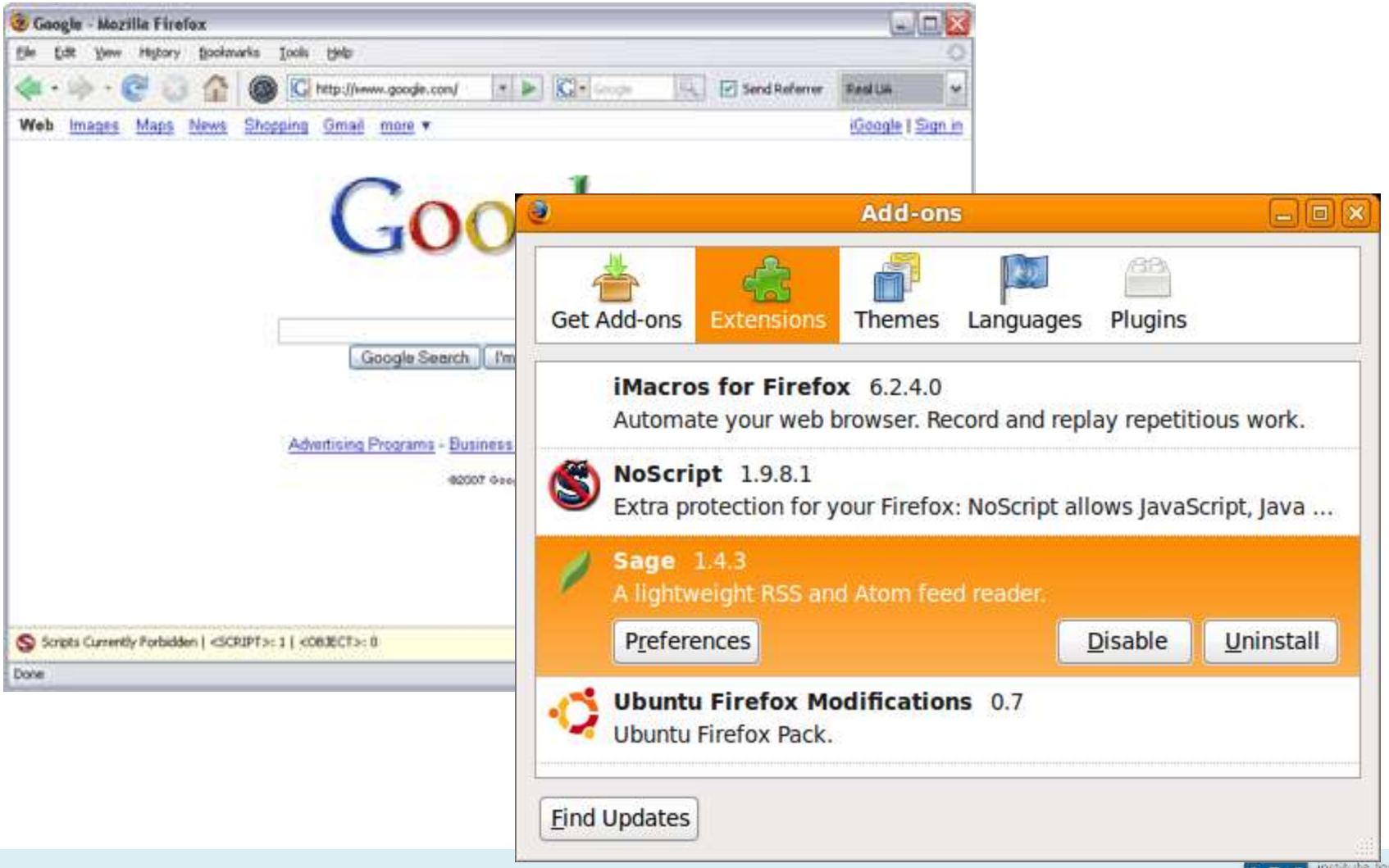
Reuse and variation: Eclipse Rich Client Platform



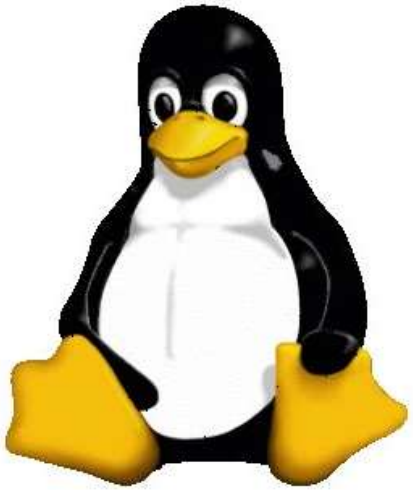
Reuse and variation:
The Standard Widget Toolkit



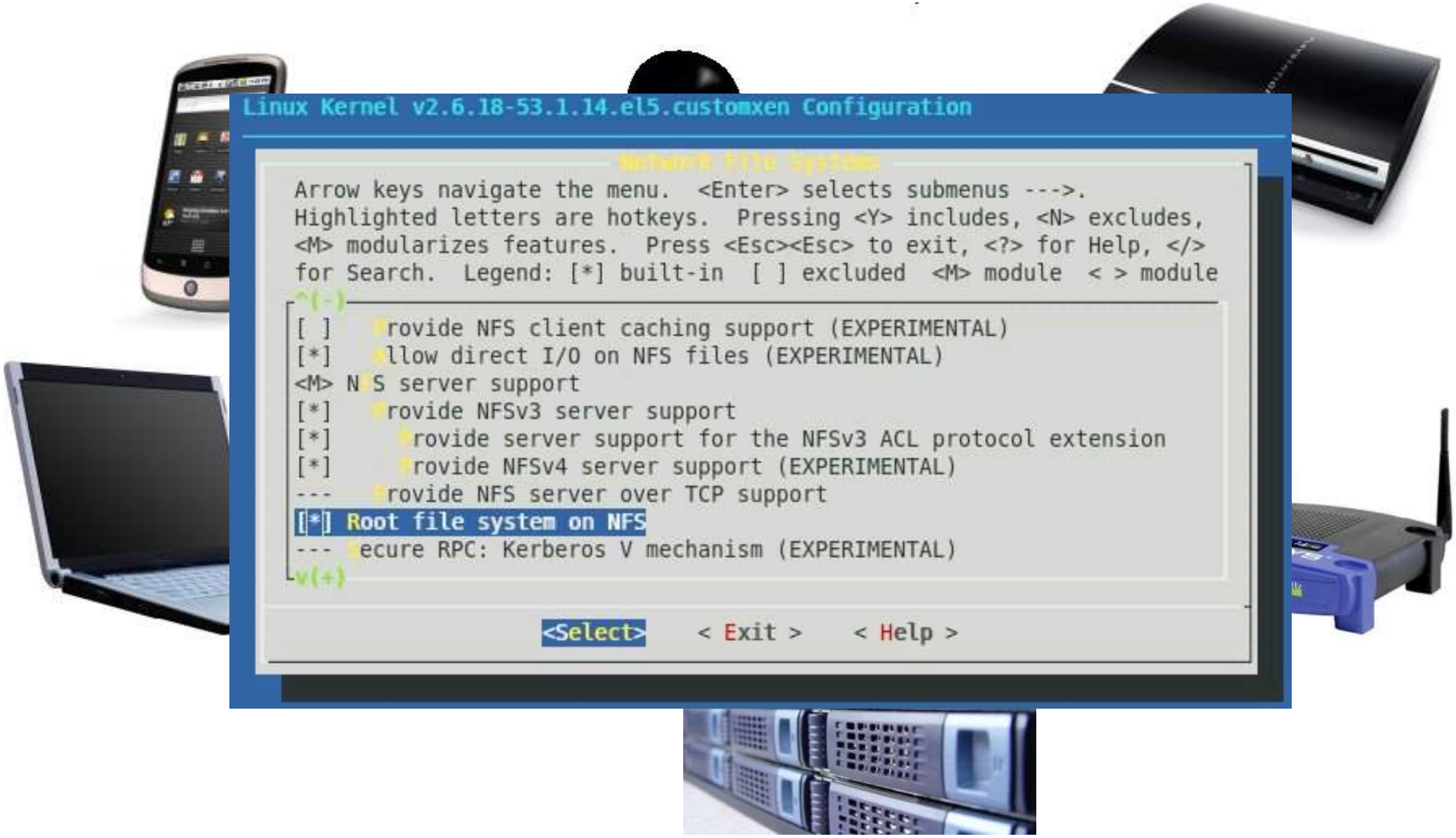
Reuse and variation: Web browser extensions



Reuse and variation: Flavors of Linux

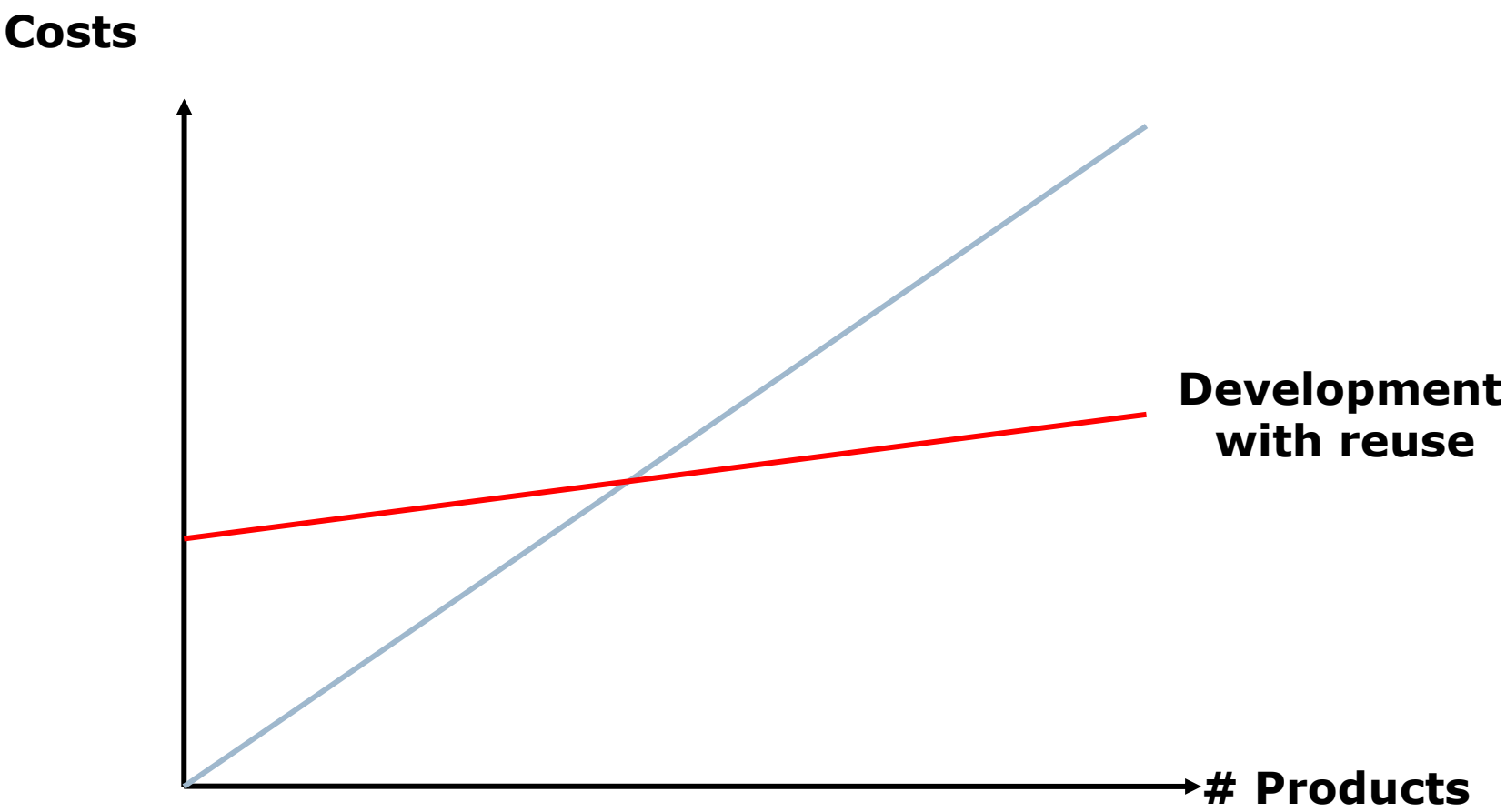


Reuse and variation: Flavors of Linux



Reuse and variation: Printer product lines

The promise:



Earlier in this course: Class-level reuse

- Language mechanisms supporting reuse
  - Inheritance
  - Subtype polymorphism (dynamic dispatch) for delegation
  - Parametric polymorphism (generics)
- Design principles supporting reuse
  - Information Hiding
  - Low coupling
  - High cohesion
  - Low representational gap
- Design patterns supporting reuse
  - Template method, decorator, strategy, composite, adapter, …

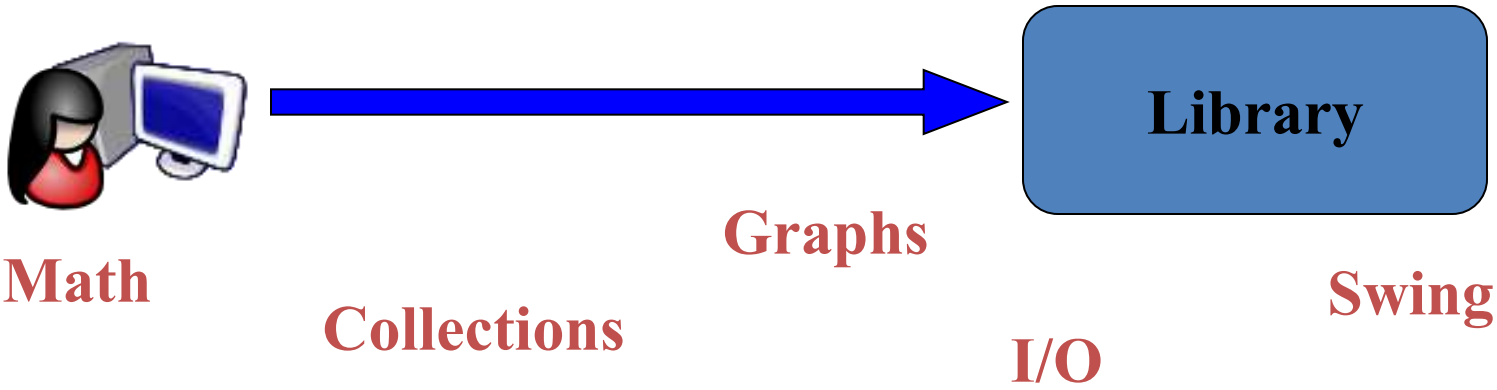
Approaches to reuse and variation

- "Clone and own"
- Subroutines
- Libraries
- Frameworks
- APIs
- Platforms
- Configuration
- Software product lines

LIBRARIES AND FRAMEWORKS

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



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.”

```
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

Eclipse **Firefox** **Swing** **Applet** **Spring**

A calculator example (without a framework)


```
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(/* calculate some stuff */);
                this.setContentPane(contentPane);
                this.pack();
                this.setLocation(100, 100);
                this.setTitle("My Great Calculator");
                // impl. for closing the window
        }
}
```

A simple example framework

- Consider a family of programs consisting of buttons and text fields only:

- What source code might be shared?

A simple example framework

- Consider a family of programs consisting of buttons and text fields only:

- What source code might be shared?
  - Main method
  - Initialization of GUI
  - Layout
  - Closing the window
  - …

A calculator example


```
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(/* calculate some stuff */);
                this.setContentPane(contentPane);
                this.pack();
                this.setLocation(100, 100);
                this.setTitle("My Great Calculator");
                // impl. for closing the window
        }
}
```

A simple example 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;
        public Application() { init(); }
        protected void init() {
                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(getInititalText());
                textfield.setPreferredSize(new Dimension(200, 20));
                contentPane.add(textfield, BorderLayout.WEST);
                button.addActionListener(/* … buttonClicked(); … */);
                this.setContentPane(contentPane);
                this.pack();
                this.setLocation(100, 100);
                this.setTitle(getApplicationTitle());
                // impl. for closing the window
        }
    protected String getInput() { return textfield.getText();}
}
```

Using the simple example 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
        prot
```

```
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);
        }
}
```

```
                this.setContentPane(contentPane);
                this.pack();
                this.setLocation(100, 100);
                this.setTitle(getApplicationTitle());
                // impl. for closing the window
        }
    protected String getInput() { return textfield.getText();}
}
```

Using the simple example framework (again)

```
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
        prot
    }
    protected S
}
```

```
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 Anything"; }
        public static void main(String[] args) {
                new Ping().setVisible(true);
        }
}
```

General distinction: Library vs. framework

```
public MyWidget extends JContainer {

ublic MyWidget(int param) {/ setup
internals, without rendering
}

/ render component on first view and
resizing
protected void
paintComponent(Graphics g) {
// draw a red box on his
componentDimension d = getSize();
g.setColor(Color.red);
g.drawRect(0, 0, d.getWidth(),
d.getHeight());                }
}
```

your code

Library

user interacts

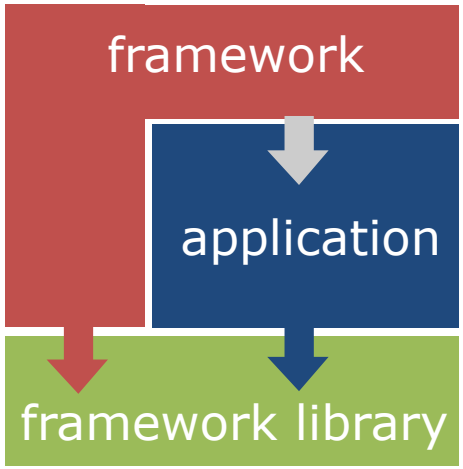
Framework

```
public MyWidget extends JContainer {

ublic MyWidget(int param) {/ setup
internals, without rendering
}

/ render component on first view and
resizing
protected void
paintComponent(Graphics g) {
// draw a red box on his
componentDimension d = getSize();
g.setColor(Color.red);
g.drawRect(0, 0, d.getWidth(),
d.getHeight());                }
}
```

your code

Libraries and frameworks in practice

- Defines key abstractions and their interfaces
- Defines object interactions & invariants
- Defines flow of control
- Provides architectural guidance
- Provides defaults



Framework or Library?

- Java Collections
- Eclipse
- The Java Logging Framework
- Java Encryption Services
- Wordpress
- Ruby on Rails
- Homework 1 Graph Implementation
- Scrabble with Stuff (HW4)

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
- Lifecycle method: A callback method of an object that gets called in a sequence according to the protocol and the state of the plugin

More terms

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

WHITE-BOX VS BLACK-BOX FRAMEWORKS

Whitebox frameworks

- Extension via subclassing and overriding methods
- Common design pattern(s):
  - Template Method
- Design steps:
  - Identify the common code and the variable code
  - Abstract variable code as method calls
- Subclass has main method but gives control to framework

Blackbox frameworks

- Extension via implementing a plugin interface
- Common design pattern(s):
  - Strategy
  - Observer
- Design steps:
  - Identify the common code and the variable code
  - Abstract variable code as methods of an interface
  - Decide whether there might be one or multiple plugins
- Plugin-loading mechanism loads plugins and gives control to the framework

Is this a whitebox or blackbox 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
        prot
```

```
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);
        }
}
```

```
    }
    protected S
}
```

An example blackbox 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.LOWERED));
                JButton button = new JButton();
                if (plugin != null)
                        button.setText(plugin.getButtonText());
                else
                        button.setText("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(/* … 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(Plugin p) { this.plugin=p; p.setApplication(this);
        protected void init() {
                JPanel contentPane = new JPanel(new BorderLayout(
                contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
                JButton button = new JButton();
                if (plugin != null)
                        button.setText(plugin.getButtonText());
                else

                contentP
                textfield
                if (plugin

                textfield.
                contentP
                if (plugin

                this.setC
                …
        }
    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 Calculator {
    public static void main(String[] args) {
        new Application(new CalcPlugin()).setVisible(true);
    }
}
```

An aside: Plugins could be reusable, too…

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

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

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

                textfield.
                contentP
                if (plugin

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

```
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 Calculator"; }
}
```

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

Whitebox vs. blackbox framework summary

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

Framework design considerations

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

The cost of changing a framework

```
public class Application extends JFrame {
        private JTextField textfield;
        private Plugin plugin;
        public Application(Plugin p) { this.plugin=p; p.setApplication(this); init(); }
        protected void init() {
                JPanel contentPane = new JPanel(new BorderLayout(
                contentPane.setBorder(new BevelBorder(BevelBorde
                JButton button = new JButton();
                if (plugin != null)
                        button.setText(plugin.getButtonText());
                else
                        button.setText("ok");
                contentPane.add(button, BorderLayout.EAST);
                textfield
                if (plugin

                textfield.
                contentP
                if (plugin
```

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

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

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

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

Summary

- Reuse and variation essential
  - Avoid reimplementing from scratch
- Object-oriented design principles for library design
- From low-level code reuse to design/behavior reuse with frameworks
- Design for reuse with domain analysis: find common and variable parts
- Use design patterns for framework design and implementation

Further Reading

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