# Software Product Lines
# Part 5: Components and frameworks

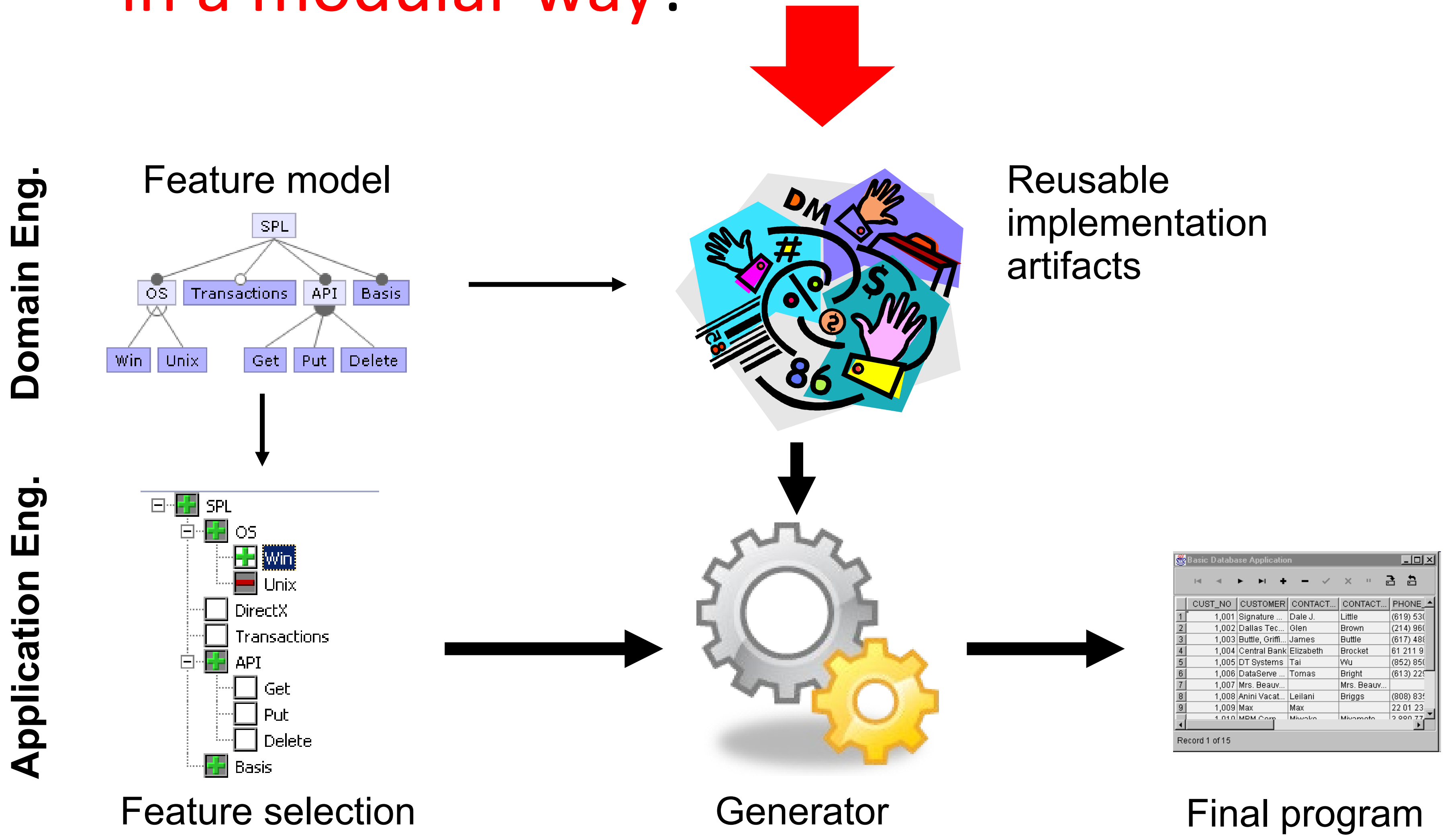
**Daniel Strüber,** Radboud University

with courtesy of: **Sven Apel, Christian Kästner, Gunter Saake**

# Last weeks: Configuration management and preprocessors

- Compile-time variability
  - Version control systems
    - Only useful for a handful of variants, but established
    - Cannot flexibly combine features
  - Build systems
    - Simple mechanism, highly flexible
    - Only file-level changes (limited reuse capabilities)
  - Preprocessors
    - Simple concept: „mark and remove“
    - Standard tools, very flexible, maximally fine-grained, feature-oriented
    - Error-prone, hard to read, scattering/tangling…

# How to implement variability in a modular way?



- ☑ SPL
  - ☑ OS
    - ☑ Win
    - ☒ Unix
  - □ DirectX
  - □ Transactions
  - ☑ API
    - □ Get
    - □ Put
    - □ Delete
  - ☑ Basis

Feature selection

Basic Database Application

| | CUST_NO | CUSTOMER | CONTACT... | CONTACT... | PHONE_ |
|---|---|---|---|---|---|
| 1 | 1,001 | Signature ... | Dale J. | Little | (619) 530 |
| 2 | 1,002 | Dallas Tec... | Glen | Brown | (214) 960 |
| 3 | 1,003 | Buttle, Griffi... | James | Buttle | (617) 488 |
| 4 | 1,004 | Central Bank | Elizabeth | Brocket | 61 211 9 |
| 5 | 1,005 | DT Systems | Tai | Wu | (852) 850 |
| 6 | 1,006 | DataServe ... | Tomas | Bright | (613) 229 |
| 7 | 1,007 | Mrs. Beauv... | | Mrs. Beauv... | |
| 8 | 1,008 | Anini Vacat... | Leilani | Briggs | (808) 835 |
| 9 | 1,009 | Max | Max | | 22 01 23 |
| | 1,010 | MPM Corp... | Miwako | Miyamoto | 3 880 77 |

Record 1 of 15

# Agenda

- Components
- Frameworks

- Crosscutting concerns
- Preplanning problem

# Components

# Components

- Self-contained modular unit of implementation with interface (black box); offers a „service“
- Often „assembled“ with other components – even from different vendors – to software system (composition)
- Ideally: can run and be marketed on its own
- Explicit definition of context (e.g., JavaEE, COM+/DCOM, OSGi) and dependencies (imports, exports)
- Size concerns
  - Small enough to be developed and maintained as one unit
  - Large enough to offer meaningful functionality

# Components vs. objects/classes

- Similar concepts: encapsulation, interfaces, information hiding
  - Objects structure a problem
  - Components offer reusable functionality increments
- Objects are smaller than components
  - „Components scale object-orientation“
- Objects often have dependencies to many other objects; components have fewer dependencies
- Interfaces of objects are often close to implementation; components abstract more

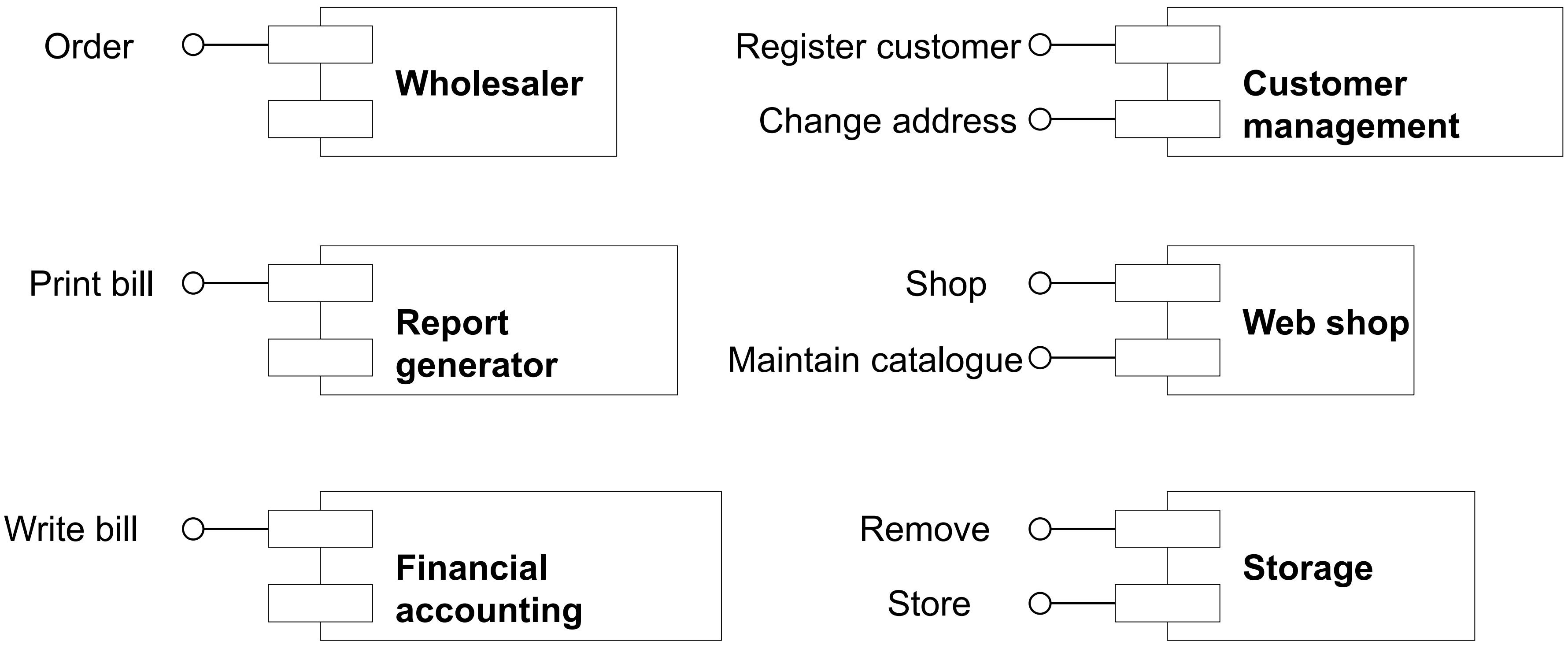
# Vision: marketplaces for components

- Components can be bought and integrated into own systems
- *Best of Breed*: developer can choose the best supplier for each system part
- Suppliers can concentrate on a core competency and offer their solutions as components

# Components of a web shop

- (UML notation: component diagram)

Scenario: Register customer → Go shopping → Create bill → Print bill



Order

Wholesaler

Register customer

Change address

Customer management

Print bill

Report generator

Shop

Maintain catalogue

Web shop

Write bill

Financial accounting

Remove

Store

Storage

# Product lines from components

- Features are implemented as components
  - for example, components for transaction management, log/recovery, buffer management, optimisation
  - Components may include runtime variability
- Components are retrieved based on feature selection (*mapping*)
- Developer has to integrate components (*glue code*)

# Example: Component „Color“ in Java

```
package modules.colorModule;
//public interface
public class ColorModule {
    public Color createColor(r:Int,g:Int,b:Int) { …}
    public void printColor(color: Color) {colorPrint… }

    public void mapColor(elem: Object, col: Color)
                    { colorMapping…}
    public Color getColor(elem: Object)
                    { colorMapping…}

    //just one module instance
    public static ColorModule getInstance()
                    { return module; }
    private static ColorModule module =
                    new ColorModule();
    private ColorModule() { super(); }
}
public interface Color { … }

//hidden implementation
class ColorPrinter { … }
class ColorMapping {…}
```

- Facade pattern
  - Hides implementation details
  - Common interface for many classes
- Singleton pattern
  - Only one instance of module

```
ColorModule.getInstance().createColor(…)
```

# Services

- Special type of component: encapsulate partial functionality (services) in distributed scenarios
  - Bus communication, Web Services, SOAP, REST…
- Abstract from programming languages, use dedicated exchange formats (based on XML, JSON etc.)
- Product lines via connection of services, usually via orchestration (workflow languages such as BPEL)
- Many tools available (often “management-oriented”)
- Aims at high degree of standardisation

# How to tailor components?

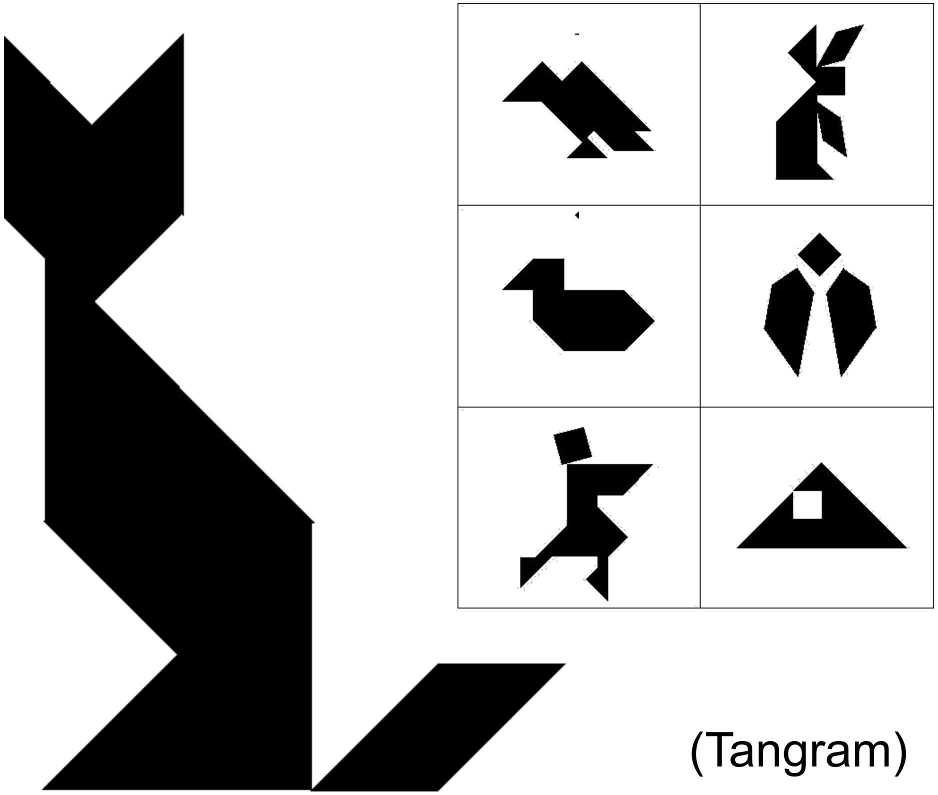
- Marketplace for arbitrary components does not work; trade-off *use* vs. *reuse*
  - Too small → high effort for use (glue code)
  - Too large → hardly reusable

- **Example**: Developer searches the web for a software solution for problem, finds 2 solutions. How to decide?
  - Solution 1: small, 1K LoC, only parts of desired functionality
  - Solution 2: large, 100K LoC, contains entire functionality, which, however, is tangled with additional, not required functionality, possibly incompatible assumptions

# How to tailor components?

- Without knowing the application context, component developers have to „guess“

- Solution approach: software product lines offer the required domain analysis
    - Systematic reuse
    - Which partial functionality used at which granularity?
    - Which parts always used together? → become component



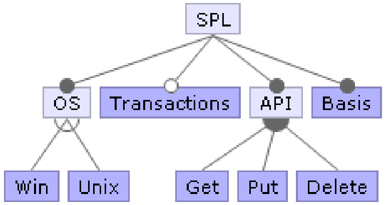
(Tangram)

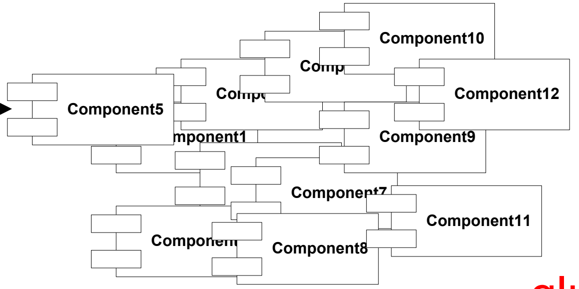
# Product lines from components

Mapping
features <-> components

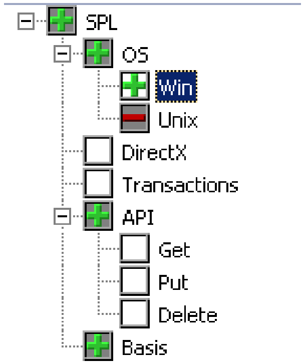
Component repository

Feature model

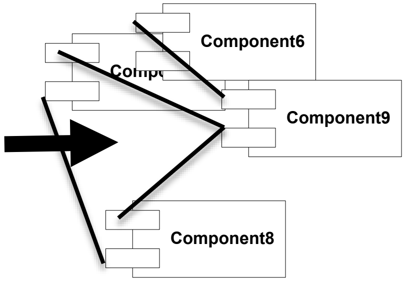


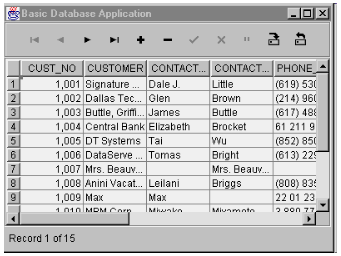


Developer writes
glue code to connect components



Feature selection
as input





| | CUST_NO | CUSTOMER | CONTACT... | CONTACT... | PHONE_ |
|---|---|---|---|---|---|
| 1 | 1,001 | Signature ... | Dale J. | Little | (619) 530 |
| 2 | 1,002 | Dallas Tec... | Glen | Brown | (214) 960 |
| 3 | 1,003 | Buttle, Griffi... | James | Buttle | (617) 488 |
| 4 | 1,004 | Central Bank | Elizabeth | Brocket | 61 211 9 |
| 5 | 1,005 | DT Systems | Tai | Wu | (852) 850 |
| 6 | 1,006 | DataServe ... | Tomas | Bright | (613) 229 |
| 7 | 1,007 | Mrs. Beauv... | | Mrs. Beauv... | |
| 8 | 1,008 | Anini Vacat... | Leilani | Briggs | (808) 835 |
| 9 | 1,009 | Max | Max | | 22 01 23 |
| | 1,010 | MRM Corp | Miwako | Miyamoto | 3 880 77 |

Record 1 of 15

Feature selection

Component selection

Component
set

Final program

# Assessment:
# Product lines from components

- Widely used in industry (for example Philips home electronics with Koala components)
- Systematic (planned) reuse of components
- Reuse in the large
- Easy division of labor

- Not fully automated, high development effort in application engineering (glue code)
- No free feature selection

# Discussion: Modularity

- Components hide internal implementation details
- Ideally small interfaces
- Cohesive features

but …

- Coarse granularity
  - Colors, weighting in graph as components?
  - Paging strategies, search algorithms, B-tree locking, VARCHAR as components?
- Functionality might be hard to encapsulate
  - Transaction management component?

# Frameworks

# Frameworks

- Incomplete set of abstract and concrete classes
- Abstract structure that can be instantiated and adapted to specific context
  - cf. *template method* and *strategy* patterns
- Reusable solution for a problem family in a domain
- Dedicated points for extensions:
  **hot spots** (a.k.a. variation points, extension points)
- Inversion of control, framework decides control flow and execution order
  - Hollywood principle: „Don’t call us, we’ll call you.“

# Plugins

- Extension of a framework
- Add special functions on demand
- Usually compiled separately; third-party
- Popular in end-user software
  - Email programs, graphic editors, media player, web browser

# Web Portal

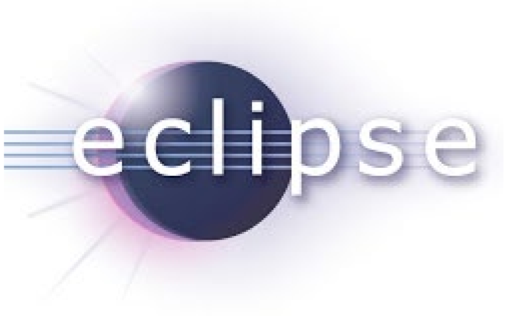
- Web application frameworks like Struts implement and offer core functionality
  - Developers can concentrate on application logic and navigation between pages

```
<?php
class WebPage {
            function getCSSFiles();
            function getModuleTitle();
            function hasAccess(User u);
            function printPage();
}
?>
```

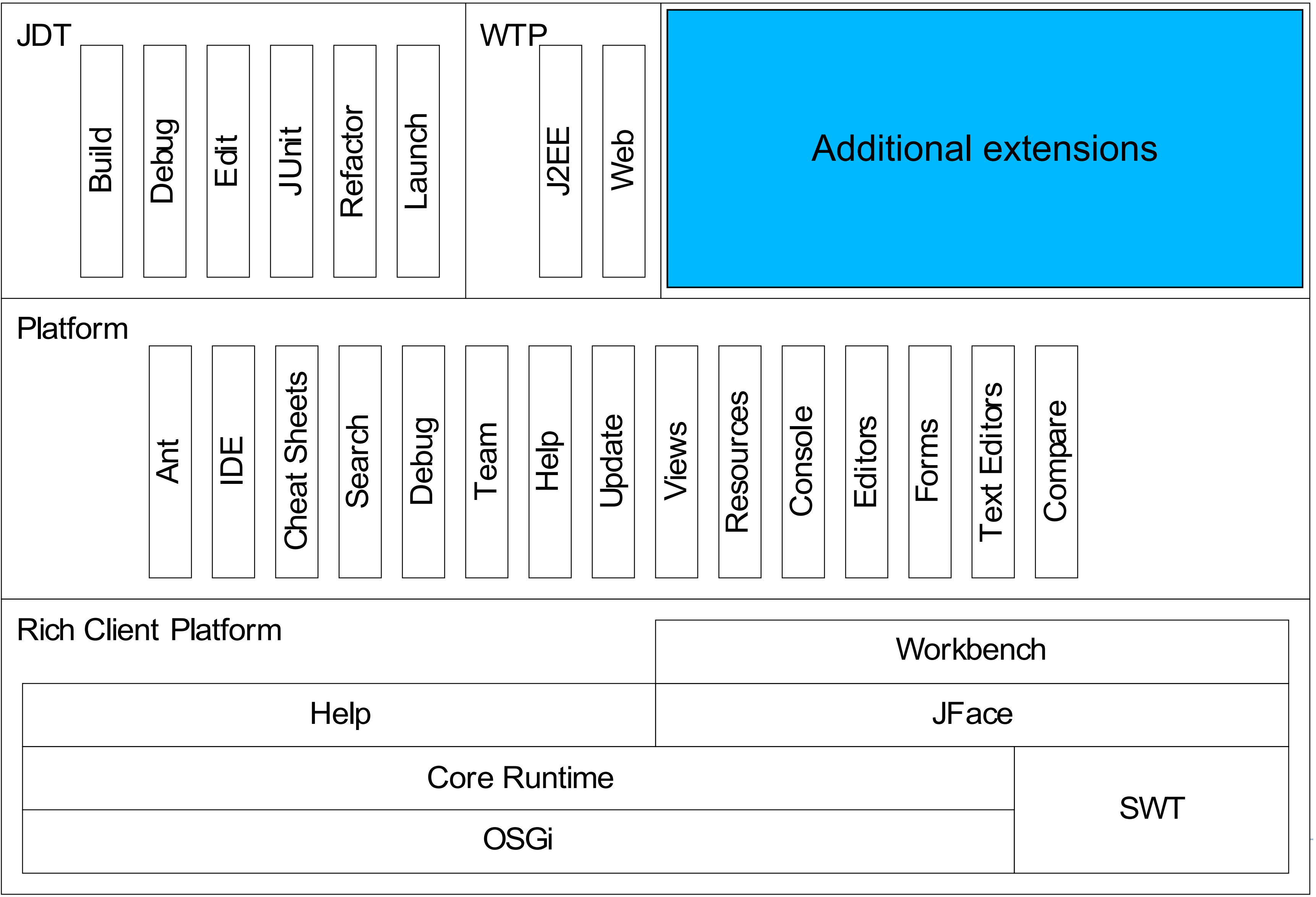
```
<?php
class ConfigPage extends WebPage {
            function getCSSFiles() {…}
            function getModuleTitle() {
               return “Configuration”;
            }
            function hasAccess(User u) {
               return user.isAdmin();
            }
            function printPage() {
               print “<form><div>…”;
            }
}
?>
```

# Eclipse

- Eclipse as a framework for IDEs
  - Framework offers common functionality (editors, menus, projects, directory tree, copy & paste, undo, VCS integration, etc.)
  - Only language-specific extensions (syntax highlighting, compiler, type checking) have to be implemented
  - Framework from many smaller frameworks



# Eclipse



JDT

Build

Debug

Edit

JUnit

Refactor

Launch

WTP

J2EE

Web

Additional extensions

Platform

Ant

IDE

Cheat Sheets

Search

Debug

Team

Help

Update

Views

Resources

Console

Editors

Forms

Text Editors

Compare

Rich Client Platform

Workbench

Help

JFace

Core Runtime

OSGi

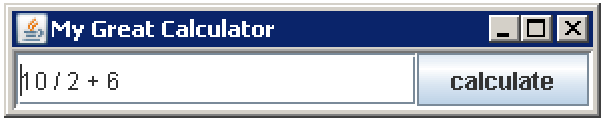
SWT

# Additional framework examples

- Frameworks for graphical user interfaces
  - MacApp, Swing, SWT, MFC
- Multimedia-Frameworks
  - DirectX
- Instant Messenger-Frameworks
  - Miranda, Trillian, ...
- Compiler-Frameworks
  - Polyglot, abc, JastAddJ

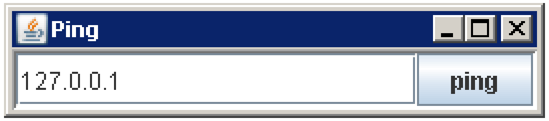
# Framework implementation: minimal example

- Family of dialogs with buttons and text fields



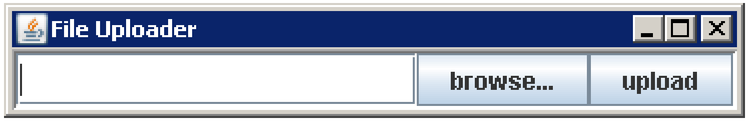
My Great Calculator

10 / 2 + 6



Ping

127.0.0.1



File Uploader

- 90% of source code identical
  - main method
  - initialize window, text field, button(s)
  - layout
  - close 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 for calcuting*/);
          this.setContentPane(contentPane);
          this.pack();
          this.setLocation(100, 100);
          this.setTitle("My Great Calculator");
          // code for closing window
        }
}
```

Source code for all variants identifcal except for red parts (hot spots)

# White-Box frameworks

- Extend by overwriting and adding methods
  - cf. *template method* pattern
- Implementation developer knows framework internals
  - → might be difficult to learn
- (Relatively) flexible extensions
- Might need many subclasses → hard to overview?
- Can directly access state of superclasses
- No plug-ins, not compiled separately

# Calculator as a white-box framework

```
public abstract class GuiApplication extends JFrame {
        protected abstract String getApplicationTitle();            // abstract methods
        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());
                // code for closing window
        }
        protected String getInput() { return textfield.getText();}
}
```

# Calculator as a white-box framework

```
public abstract class GuiApplication extends JFrame {
            protected abstract String getApplicationTitle();                // abstract methods
            protected abstract String getButtonText();
            protected String getInititalText() {return "";}
            protected void buttonClicked() { }
            private JTextField textfield;
            public Application() { init(); }
            protected v
                  this.setContentPane(contentPane);
                  this.pack();
            }
               protected
}
```

```
public class Calculator extends GuiApplication {
            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 GuiApplication {
            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);
            }
}
```

Modularity?

# Black-Box frameworks

- Embed application-specific behavior via components with a special interface (**plug-ins**)
  - cf. *strategy* and *observer* patterns
- Implementation developer only needs to know interface
  - easier to learn, harder to design
- Flexibility is determined by the offered hot spots, often implemented with design patterns
- State only known if available via interface
- Loose coupling (esp. compared to white-box frameworks)

# Calculator

```
public class GuiApplication extends JFrame {
        private JTextField textfield;
        private Plugin plugin;
        public GuiApplication(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(GuiApplication app);
}
```

# Calculator

```
public class GuiApplication extends JFrame {
        private JTextField textfield;
        private Plugin plugin;
        public GuiApplication(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)
                      utton.setText(plugin.getButtonText());
                el
                      on.setText("ok");
                      l(button, BorderLayout.EAST);
                      v JTextField("");
                      != null)
                      textfield.setText(plugin.getInititalText());
                textfield.
                contentP
                if (plugin

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

Modularity?

Application does not know plug-ins

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

```
public class CalcPlugin implements Plugin {
        private GuiApplication application;
        public void setApplication(GuiApplication 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 GuiApplication(new CalcPlugin()).setVisible(true); }}
```

# Further decoupling possible

```
public class GuiApplication extends JFrame implements InputProvider {
        private JTextField textfield;
        private Plugin plugin;
        public GuiApplication(Plugin p) { this.plugin=p; p.setApplication(this); init(); }
        protected void init() {
                JPanel contentPane = new JPanel(new BorderLayout(
                contentPane.setBorder(new BevelBorder(BevelBorder.
                JButton button = new JButton();
                if (plugin != null)
                        b         tText(plugin.getButtonText());
                else
                                  ext("ok");
                                  , BorderLayout.EAST);
                                  tField("");
                              ull)
                        textfield.setText(plugin.getInititalText());
                extfield.
                contentP
                if (plugin

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

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

```
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 Calculator"; }
}
```

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

**Modularity?**

Only plug-in and InputProvider interface

# Loading of plug-ins

- Typical in many frameworks: plugin loader …
  - … searches directory for dll/jar/xml files
  - … tests if the file implements a plug-in
  - … checks dependencies
  - … initializes plug-in on loading
- Often additional GUI for plug-in installation and configuration
- Examples:
  - Eclipse (plugin directory + Jar)
  - Firefox (plugin directory + DLL)
- Alternative: determine plug-ins in config file or create a launcher program

# Example plugin loader (using Java reflection)

```
public class Starter {

    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);
                }
        }
    }
}
```

# Multiple plug-ins

- cf. *observer* pattern
- Load and register multiple plug-ins
- On event, inform all plug-ins

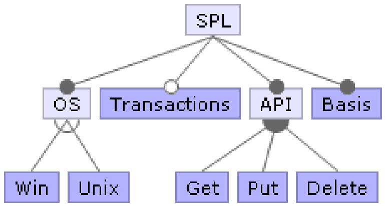
```
public class Application {
        private List<Plugin> plugins;
        public Application(List<Plugin> plugins) {
                this.plugins = plugins;
                for (Plugin plugin : plugins)
                    plugin.setApplication(this);
        }
        public Message processMsg (Message msg) {
                for (Plugin plugin : plugins)
                    msg = plugin.process(msg);
                ...
                return msg;
        }
}
```

- For different tasks: more specific plug-in interfaces

# Frameworks for product lines

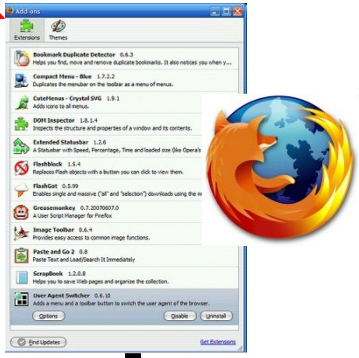
Domain Eng.

Feature model



Mapping
features <-> plug-ins

Framework + plug-ins



**Bookmark Duplicate Detector** 0.6.3
Helps you find, move and remove duplicate bookmarks. It also notices you when y...

**Compact Menu - Blue** 1.7.2.2
Duplicates the menubar on the toolbar as a menu of menus.

**CuteMenus - Crystal SVG** 1.9.1
Adds icons to all menus.

**DOM Inspector** 1.8.1.4
Inspects the structure and properties of a window and its contents.

**Extended Statusbar** 1.2.6
A Statusbar with Speed, Percentage, Time and loaded size (like Opera's

**Flashblock** 1.5.4
Replaces Flash objects with a button you can click to view them.

**FlashGot** 0.5.99
Enables single and massive ("all" and "selection") downloads using the m

**Greasemonkey** 0.7.20070607.0
A User Script Manager for Firefox

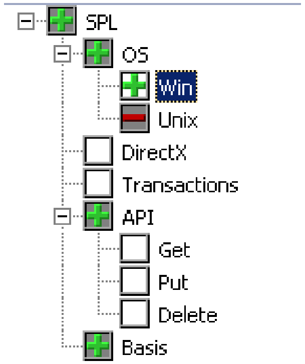
**Image Toolbar** 0.6.4
Provides easy access to common image functions.

**Paste and Go 2** 0.8
Paste Text and Load/Search It Immediately

**ScrapBook** 1.2.0.8
Helps you to save Web pages and organize the collection.

**User Agent Switcher** 0.6.10
Adds a menu and a toolbar button to switch the user agent of the browser.

Application Eng.

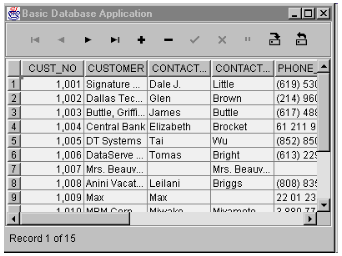


- ☑ SPL
  - ☑ OS
    - ☑ Win
    - ☒ Unix
  - □ DirectX
  - □ Transactions
  - ☑ API
    - □ Get
    - □ Put
    - □ Delete
  - ☑ Basis

Feature selection

Feature selection
as input

plug-in selection
(and possibly, generation
of launch configuration)



| | CUST_NO | CUSTOMER | CONTACT... | CONTACT... | PHONE_ |
|---|---|---|---|---|---|
| 1 | 1,001 | Signature ... | Dale J. | Little | (619) 530 |
| 2 | 1,002 | Dallas Tec... | Glen | Brown | (214) 960 |
| 3 | 1,003 | Buttle, Griffi... | James | Buttle | (617) 488 |
| 4 | 1,004 | Central Bank | Elizabeth | Brocket | 61 211 9 |
| 5 | 1,005 | DT Systems | Tai | Wu | (852) 850 |
| 6 | 1,006 | DataServe ... | Tomas | Bright | (613) 229 |
| 7 | 1,007 | Mrs. Beauv... | | Mrs. Beauv... | |
| 8 | 1,008 | Anini Vacat... | Leilani | Briggs | (808) 835 |
| 9 | 1,009 | Max | Max | | 22 01 23 |
| | 1,010 | MRM Corp | Miyako | Miyamoto | 3 880 77 |

Record 1 of 15

application =
framework with
desired plug-ins

# Assessment
# Frameworks for product lines

- Benefits
  - Fully automation possible
  - Modularity
  - Tested in practice
- Drawbacks
  - Development effort
  - Runtime overhead for framework architecture
  - Needs preplanning and requires suitable experience
  - Evolution and mainentance complicated
- Coarse granularity and large interfaces

# Crosscutting concerns

# Crosscutting concerns

- Claim: Not all concerns of a program can be modularized using objects, components, plugins
  - Applies to features as well, which are one type of concern
- Concerns are semantic units
- But their implementation can be scattered throughout the source code

# Crosscutting concerns - example

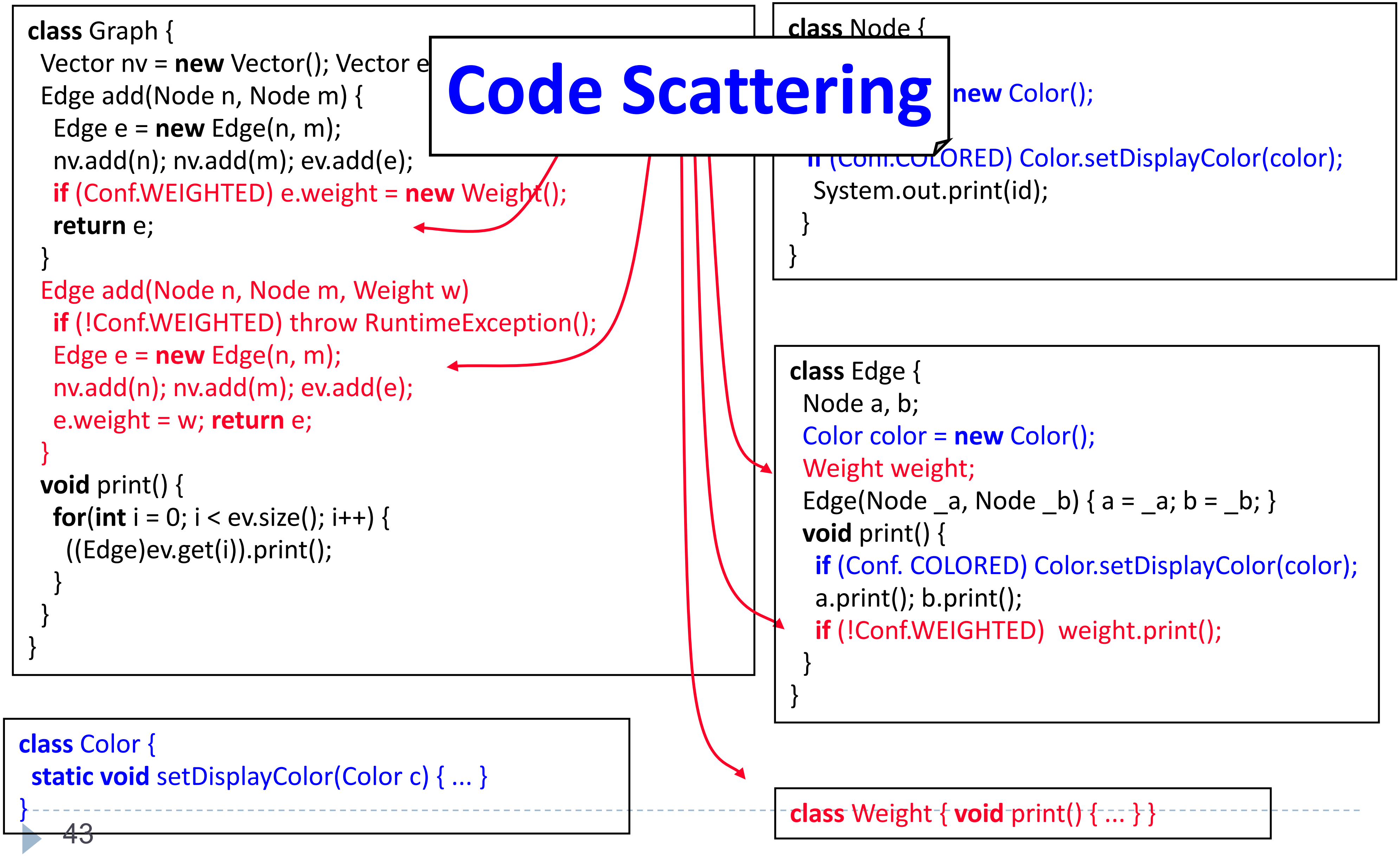
```
class BusinessClass
  //... data fields
  //... logging stream
  //... cache status
  public void importantOperation(
        Data data, User currentUser, ...){
    // check authorization
    // lock objects for synchronization
    // check if buffer up-to-date
    // log start of actual operation
    // execute actual operation
    // log end of actual operation
    // unlock objects
  }

  public void alsoImportantOperation(
        OtherData data, User currentUser, ...){
    // check authorization
    // lock objects for synchronization
    // check if buffer up-to-date
    // log start of actual operation
    // execute actual operation
    // log end of actual operation
    // unlock objects
  }
}
```

- Code for different concerns scattered
- Code replicated
- Operations in this example are modular, but locking, logging, buffer and authentication not

# Scattering and tangling

- Code scattering
  - Code that belongs to a concern is not modularized, but spread throughout the entire program
  - Frequently copied code (e.g., redundant calls of a method)
  - Or spread implementation of parts of the concern
- Code tangling
  - Code that belongs to several concerns is jumbled within one class or method

# Scattered Code



Code Scattering

```
class Graph {
  Vector nv = new Vector(); Vector e
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    if (Conf.WEIGHTED) e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w)
    if (!Conf.WEIGHTED) throw RuntimeException();
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    e.weight = w; return e;
  }
  void print() {
    for(int i = 0; i < ev.size(); i++) {
      ((Edge)ev.get(i)).print();
    }
  }
}
```

```
class Node {
                 new Color();

    if (Conf.COLORED) Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
class Edge {
  Node a, b;
  Color color = new Color();
  Weight weight;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    if (Conf. COLORED) Color.setDisplayColor(color);
    a.print(); b.print();
    if (!Conf.WEIGHTED)  weight.print();
  }
}
```

```
class Color {
  static void setDisplayColor(Color c) { ... }
}
```

```
class Weight { void print() { ... } }
```

# Tangled Code

```
class Graph {
  Vector nv = new Vector(); Vector ev = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    if (Conf.WEIGHTED) e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w)
    if (!Conf.WEIGHTED) throw RuntimeException();
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    e.weight = w; return e;
  }
  void print() {
    for
     ((
    }
  }
}
```

Code Tangling

```
class Node {
  int id = 0;
  Color color = new Color();
  void print() {
   if (Conf.COLORED) Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
class Edge {
  Node a, b;
  Color color = new Color();
  Weight weight;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    if (Conf. COLORED) Color.setDisplayColor(color);
    a.print(); b.print();
    if (!Conf.WEIGHTED)  weight.print();
  }
}
```

```
class Color {
  static void setDisplayColor(Color c) { ... }
}
```

```
class Weight { void print() { ... } }
```

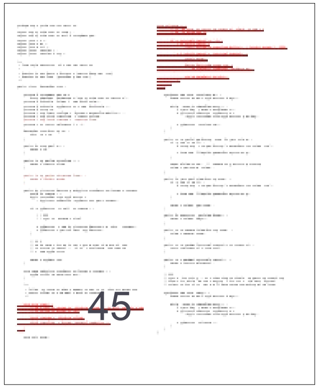
# A question of size

- Example: Session expiration in the Apache Tomcat Server

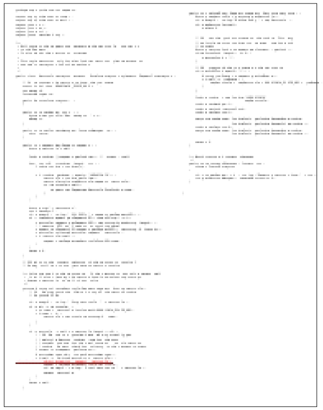
ApplicationSession

StandardSession

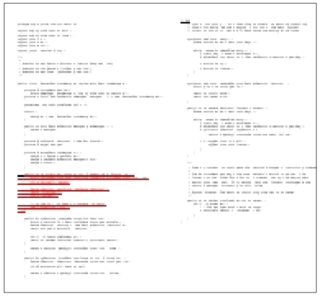
ServerSession



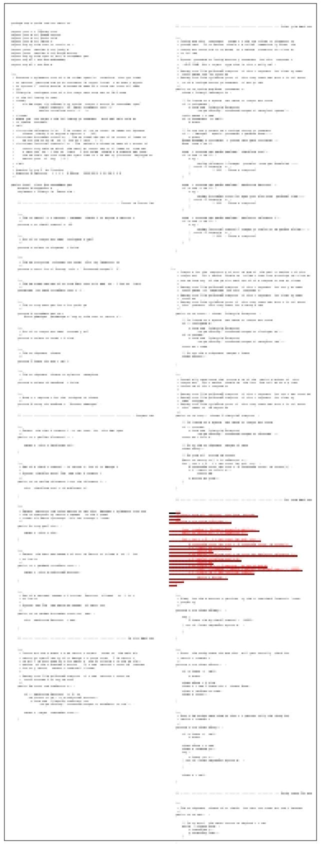
SessionInterceptor



ServerSessionManager



StandardManager



StandardSessionManager

# Problems of crosscutting concerns

- Concerns are buried in implementation
  - What belongs to the concern?
  - During maintenance, have to scan the entire source code
- Complicated collaborative development
  - Different concerns can have different experts; all may have to work in parallel on the same code parts
- Reduced productivity, difficult evolution
  - When adding new code, the developer has to worry about aspects that are not directly relevant for the problem at hand (readability, understandability)

# Alternative implementation (command pattern)

```
class SecureSystem extends System
  private User currentUser;
  public void login() { /* ... */ }

  public void executeOperation(Operation o) {
    if (o instanceof AuthorizeOrder)
      if (!currentUser.isAdmin())
        denyAccess();
      else
        o.execute();

    if (o instanceof StartShipping)
      if (!o.hasWriteAccess())
        denyAccess();
      else
        o.execute();
  }
}
```

- Authentification now modularized
- In turn, other concerns (specifically, the operations) no longer modular

# Another attempt – method calls

- Extract authentification, logging, locking, buffer into seperate methods
  - Scattering and tangling reduced to method calls
  - Clearer, but still explicit calls in code
- ➔ Many extension points in framework required; big component interfaces

```
class BusinessClass
  public void importantOperation(
        Data data, User currentUser, ...){
    checkAuth(currentUser);
    startSynchronization();
    checkCache();
    logStart();
    // execute actual operation
    logEnd();
    endSynchronization();
  }
}
```

# Another attempt - middleware

- Middleware takes care of crosscutting concerns; developer only writes actual operations (inversion of control)
  - Examples: Enterprise Java Beans provide support for distributed objects, persistence, transactions, authentication and authorization, and synchronization
  - Complex architecture
  - Middleware cannot capture all possible concerns, in particular, those concerning business logic

# Tyranny of the dominant decomposition

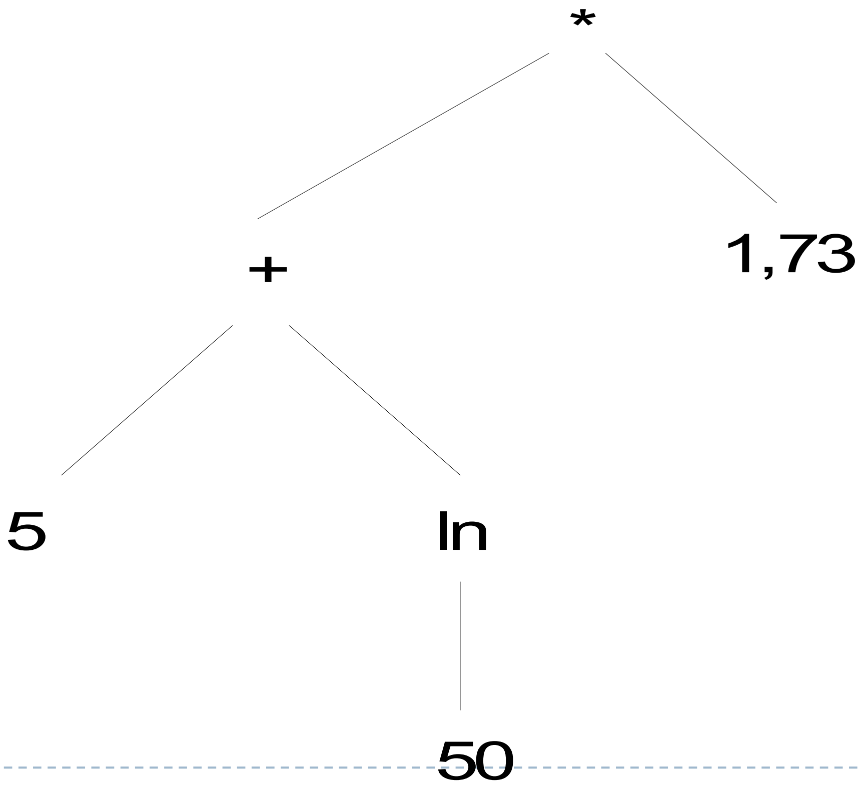
- Many concerns can be modularized, but usually, not all of them at the same time
  - Need to choose a “main dimension” of the modularization
  - Graph example: I can have colors and weights as modules…
  - …but then the data structures (node, edge) are scattered
- Developers decide on a particular modularization (e.g. operations, authentication, data structures), but some other concerns are crosscuttting
- Modularizing along several dimensions at the same time not possible

# The expression problem

- A standard example for the *tyranny of dominant decomposition* problem
- Underlying question: To what extent is it possible to extract methods and data structures so that both can be extended independently…
  - without changing existing code
    - or even: without recompiling existing code
  - several times, in arbitrary order
  - without (non-trivial) code replication

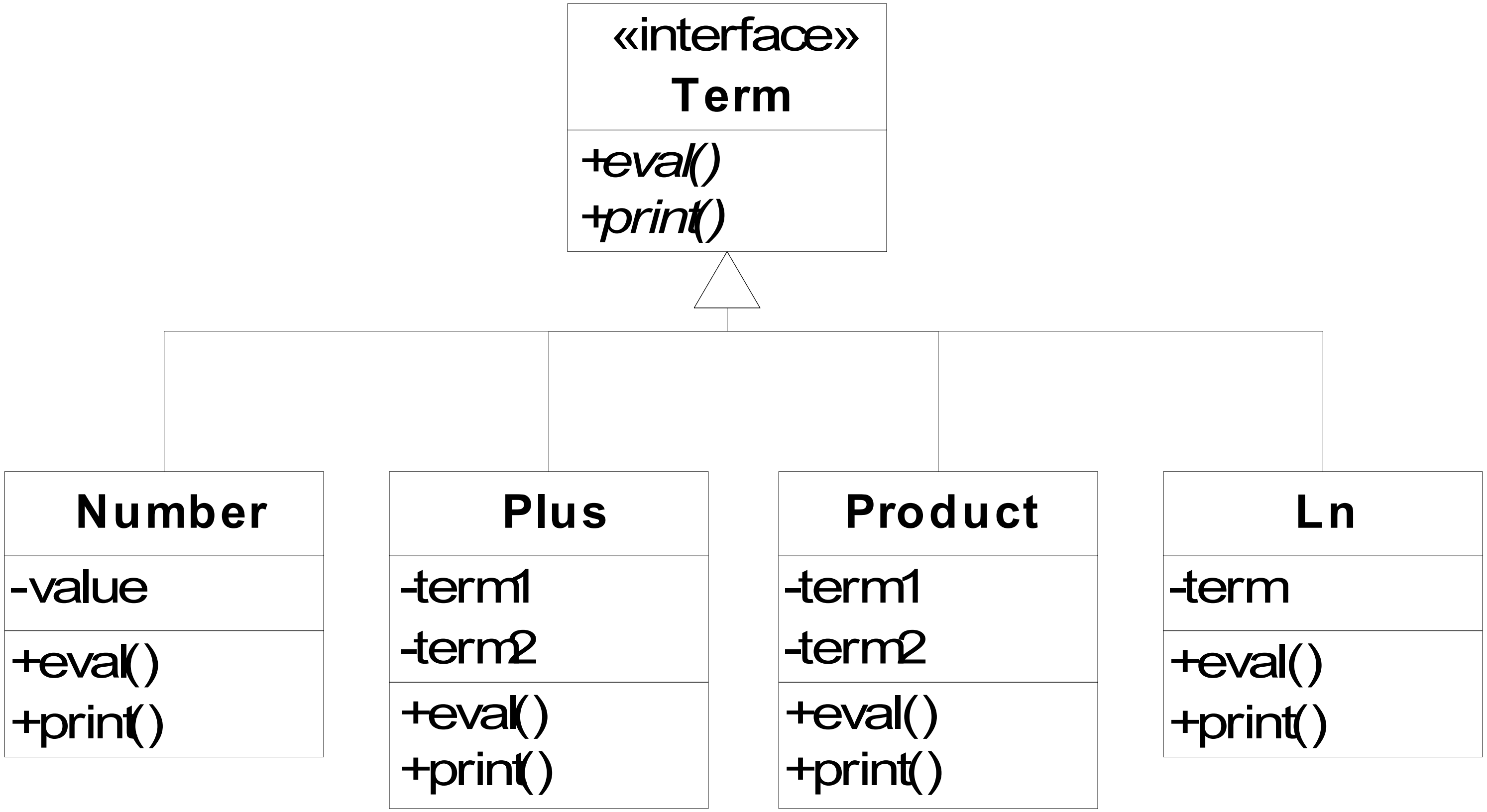
# Expressions

- Task: Save arithmetic expression in trees to evaluate and print them



# Implementation 1: data centric

- Recursive class structure (*composite* pattern)
- Each operation defines one method in each class



«interface»
**Term**
*+eval()*
*+print()*

**Number**
-value
+eval()
+print()

**Plus**
-term1
-term2
+eval()
+print()

**Product**
-term1
-term2
+eval()
+print()

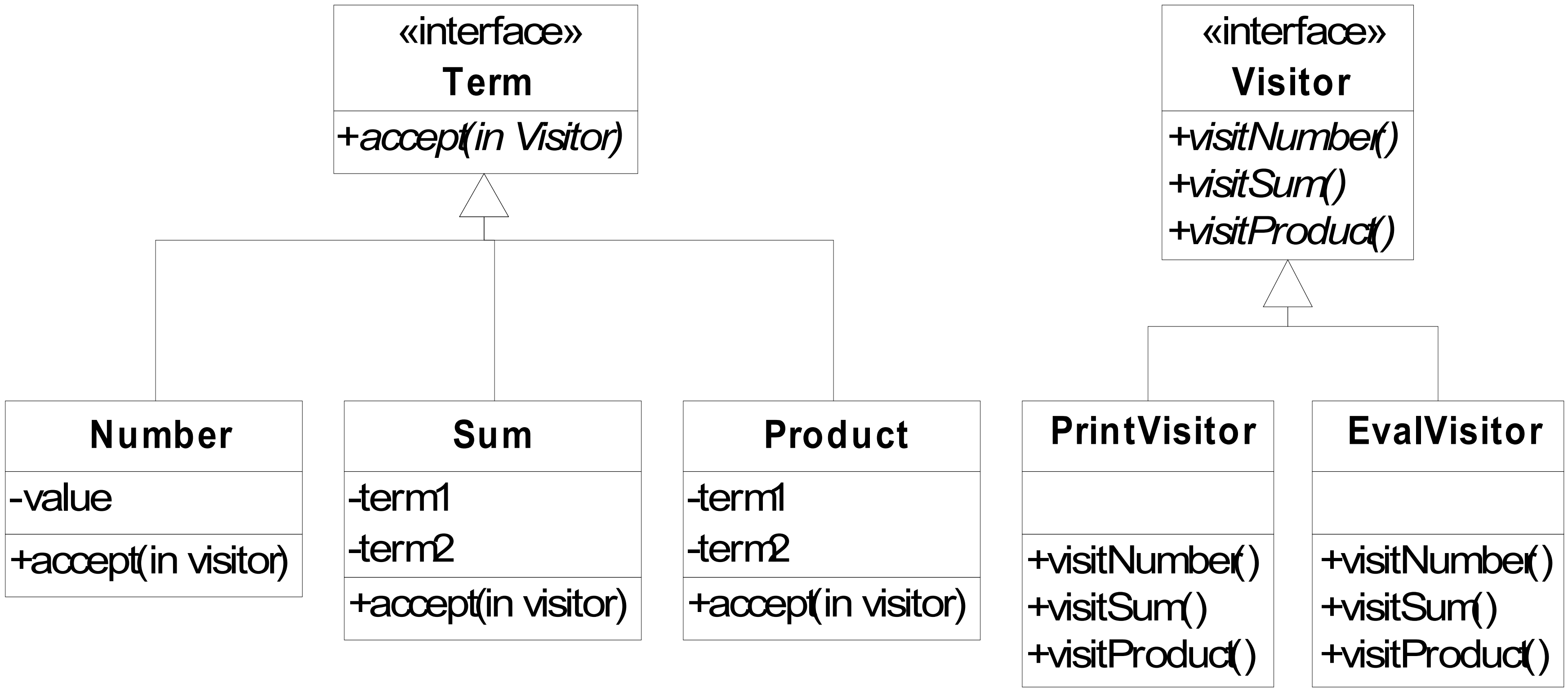
**Ln**
-term
+eval()
+print()

# Implementation 1: problems

- Expressions are modular
- New operations (e.g. *drawTree* or *simplify*) cannot be added without significant effort
- All existing classes have to be changed!
- Operations are crosscutting to expressions

# Implementation 2: method centric

- Only one method *accept* per class
- Methods are implemented with the *visitor* pattern



*(Ln Klasse aus Platzgründen ausgelassen)*

# Code example: method centric

```
interface Term {
  void accept(Visitor v);
}
class Number {
  float value;
  void accept(Visitor v) {
    v.visitNumber(this);
  }
}
class Sum {
  Term term1, term2;
  void accept(Visitor v) {
    v.visitSum(this);
  }
}
class Product {
  Term term1, term2;
  void accept(Visitor v) {
    v.visitProduct(this);
  }
}
```

```
interface Visitor {
  void visitNumber(Number n);
  void visitSum(Sum s);
  void visitProduct(Product p);
}
class PrintVisitor {
  void visitNumber(Number n) {
    System.out.print(n.value);
  }
  void visitSum(Sum s)  {
    System.out.print('(');
    s.term1.accept(this);
    System.out.print('+');
    s.term2.accept(this);
    System.out.print(')');
  }
  void visitProduct(Product p) {
    s.term1.accept(this);
    System.out.print('*');
    s.term2.accept(this);
  }
}

// Main:
// term.accept(new PrintVisitor());
```

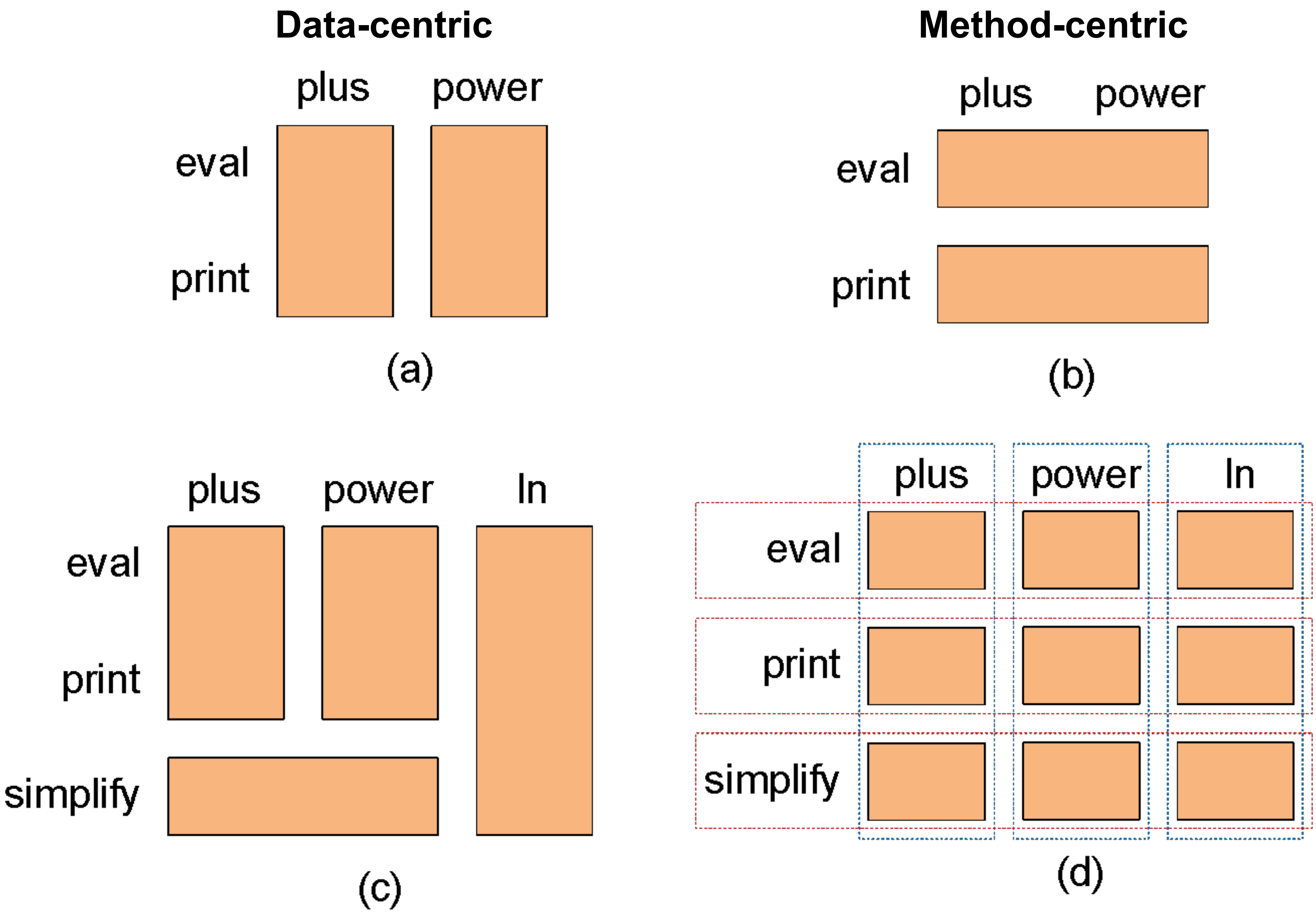
# Implementation 2: problems

- Operations are modular
- New expression types (e.g. *min* or *power*) cannot be added without significant effort
- All existing visitor classes have to be changed!
- Expressions are crosscutting to operations

# Expression problem

- Modularizing along expressions and operations at the same time almost impossible (complicated solutions with Java 1.5 generics exist)
- Data-centric approach
  - New expressions can be added directly: modular
  - New operations have to be added to all classes: not modular
- Method-centric approach
  - New operations can added as additional visitors: not modular
  - New expressions lead to changes of all visitor classes: not modular

# Expression Problem – graphically



**Outlook: new language approaches**

**Outlook: feature interactions**

# Typical examples for crosscutting concerns

- Logging: record each method call
- Caching/Pooling: code for each creation of an object
- Synchronization/Locking: extend many methods with lock/unlock calls
- Features in software product lines!

# Dilemma

- Not always possible to modularize all concerns
- Some degree of scattered and tangled code is commonly accepted
- Some concerns are always „orthogonal“ to others: crosscutting concerns
- Often affects features of product lines

# Preplanning problem

# Preplanning problem

- Cannot add extensions ad hoc, but have to plan them in advance
- Need to explicitly design facilities for extension
  - Extension points in frameworks
  - Interfaces/parameters in components

- Without a suitable extension point, modular extension not possible

# Preplanning problem: example

- Want to synchronize Stack methods
- Modular extension with subclass or delegation

Base code

```
class Stack { /* ... */ }
class Main {
  public static void main(
                      String[] args) {
    Stack stack = new Stack();
    stack.push('foo');
    stack.push('bar');
    stack.pop();
  }
}
```

Later extension (unplanned)

```
class LockedStack extends Stack {
  private void lock() { /* ... /* }
  private void unlock() { /* ... /* }
  public void push(Object o) {
    lock();
    super.push(o);
    unlock();
  }
  public Object pop() {
    lock();
    Object result = super.pop();
    unlock();
    return result;
  }
}
```

# Preplanning problem: example II

- Problem: have to change instantiation of stack in base code
  - cannot be done without changing base code (non-modular)
- Alternative
  - Design Pattern: factory instead of direct instantiation (would allow modular extension)
  - Framework with suitable extension point
- Extension points have to be anticipated (preplanning) or have to be added to the base code after-the-fact (non-modular)

# Summary

- Feature modularization with components and frameworks
- No full automation, runtime overhead, coarse granularity
- Limitations related to crosscutting concerns and fine granularity
- Modularity requires planning
- Not suitable for all product lines (e.g., graph product line, embedded databases)

# Outlook

- Advanced programming concepts
  - Understanding the limits of object-oriented programming
  - Feature-orientation
  - Aspect-orientation

# Literature

- C. Szyperski: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998
  [Reference book on component-oriented programming]
- R. Johnson and B. Foote, Desiging reusable classes, Journal of Object-Oriented Programming, 1(2):22-35, 1988
  [OOP reuse, especially frameworks]
- L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, 2003
  [architecture-driven product lins, usually frameworks]