Software Product Line Engineering

Components and Frameworks

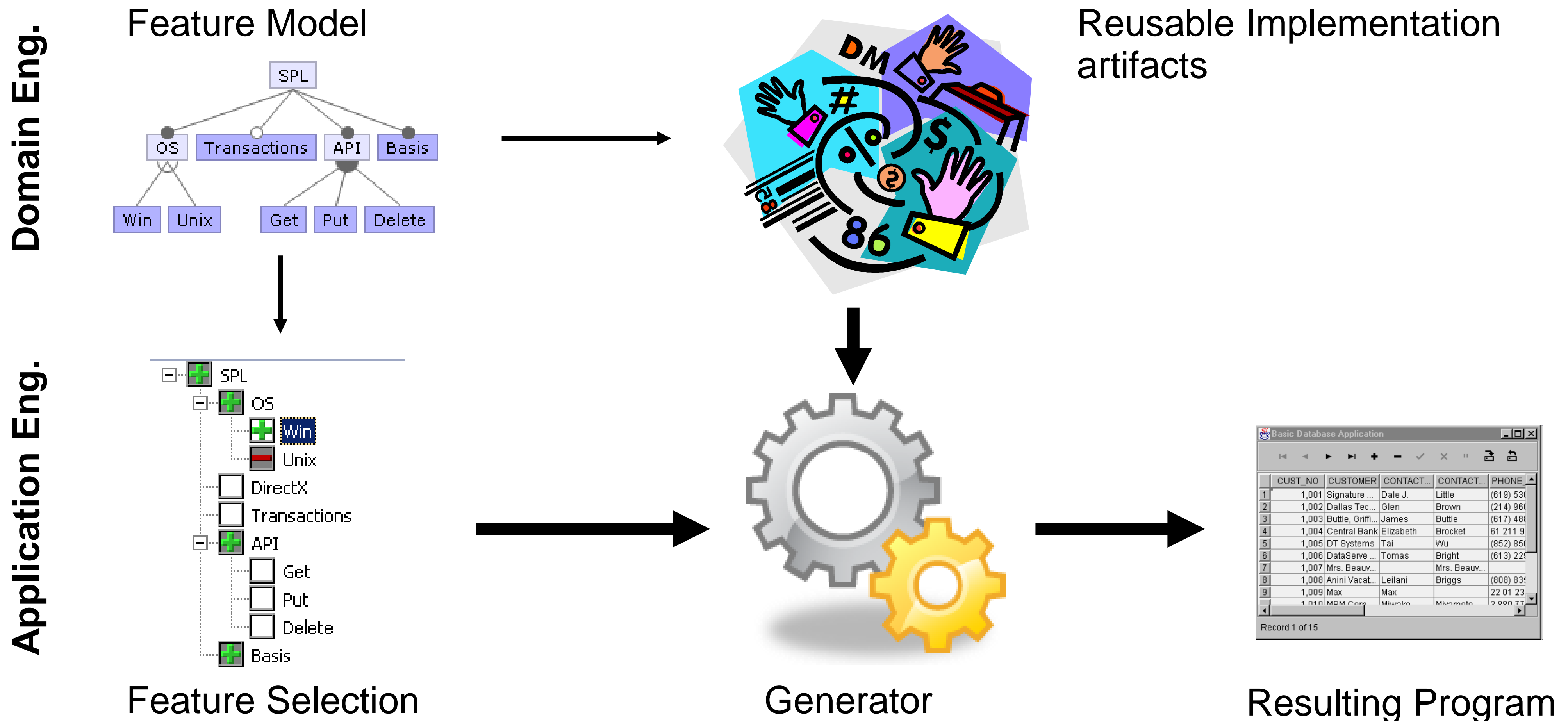
Christian Kästner (Carnegie Mellon University)
Sven Apel (Universität Passau)
Norbert Siegmund (Bauhaus-Universität Weimar)
Gunter Saake (Universität Magdeburg)

Bauhaus-Universität Weimar

Recap: Configuration Management and Preprocessors

- Variability at compile time
  - Versioning systems
    - Suitable when having only few variants, but established tools
    - Flexible combinations of features not possible
  - Build systems
    - Simple mechanism with high flexibility
    - Development of variants separated (no individual features)
    - Changes at file level (limited reusability)
  - Preprocessors
    - Simple pattern: “tune and prune”
    - Standard tools, high flexibility, fine granularity, feature development
    - Error prone, hard to read, scattering/tangling, limited traceability,…
- How to implement features in a modular way?

How to implement variability in a **modular** way?



Components

Component

- Cohesive, modular implementation unit with an interface (black box); provides a “service”
- Composed with other components – possibly from other vendors – to build software systems
- Can be distributed and “executed” individually
- Context (e.g., JavaEE, CORBA, COM+/DCOM, OSGi) and dependences (imports, exports) explicitly specified
- Small enough to be created and maintained in one piece, but also large enough to provide a useful functionality

Components vs. Objects/Classes

- Similar concepts: Encapsulation, interfaces, information hiding
  - Objects structure a problem
  - Components provide reusable functionality
- Objects are typically smaller than components: "Components scale object-oriented applications"
- Objects typically have many dependencies to other objects; components have only few dependencies
- Interfaces of objects are usually implementation specific; components provide a more abstract description

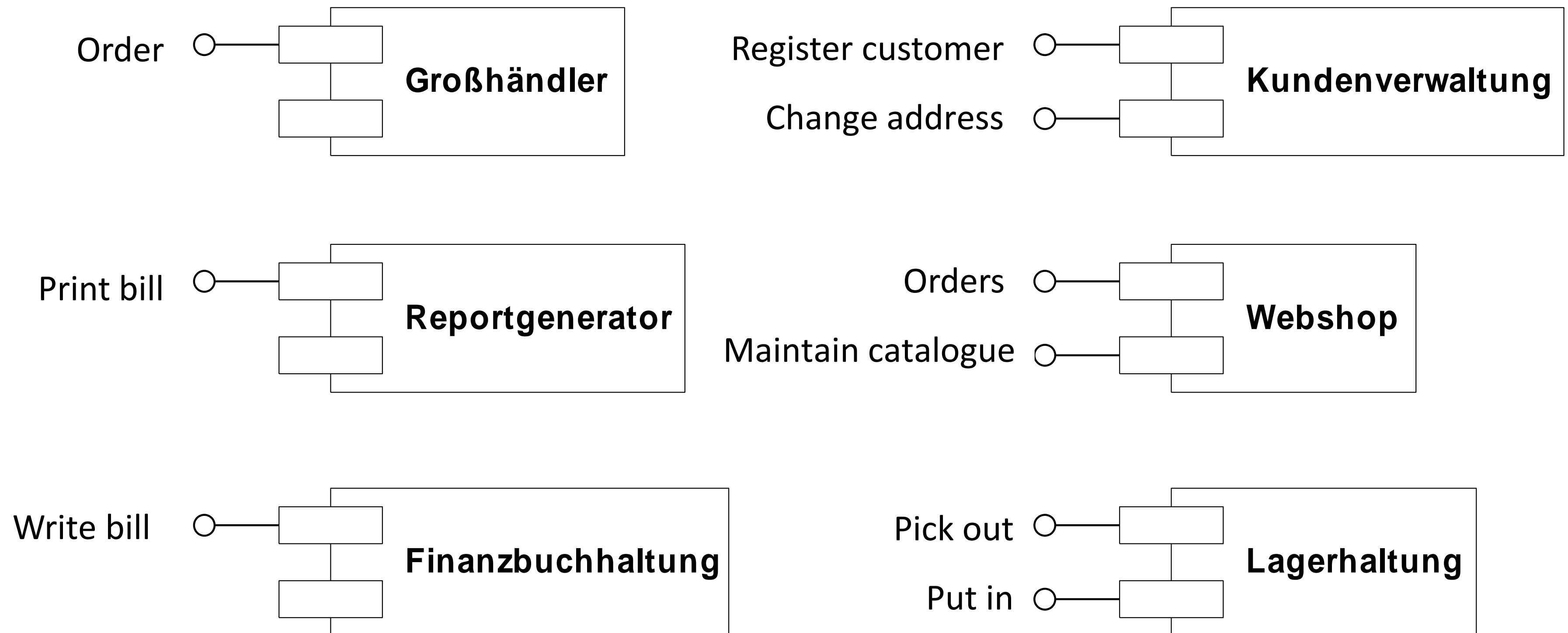
Vision: Marketplace of Components

- Components can be bought and integrated into own programs
- Best of Breed: Developer can chose for each subsystem what is the best/cheapest provider of a component
- Provider can focus on their main expertise and provide this as a component

Hasan Albari

Components of a Web Shop

- (UML notation)



Scenario: Register customer → Shop → Write bill → Print bill

Product Lines from Components

- Features are implemented in different components
  - Components for transaction management, logging & recovery, buffer management, optimization, etc.
  - Components could include additional runtime variability
- Feature selections maps to component selection
- Developers need to connect the components (write glue code)

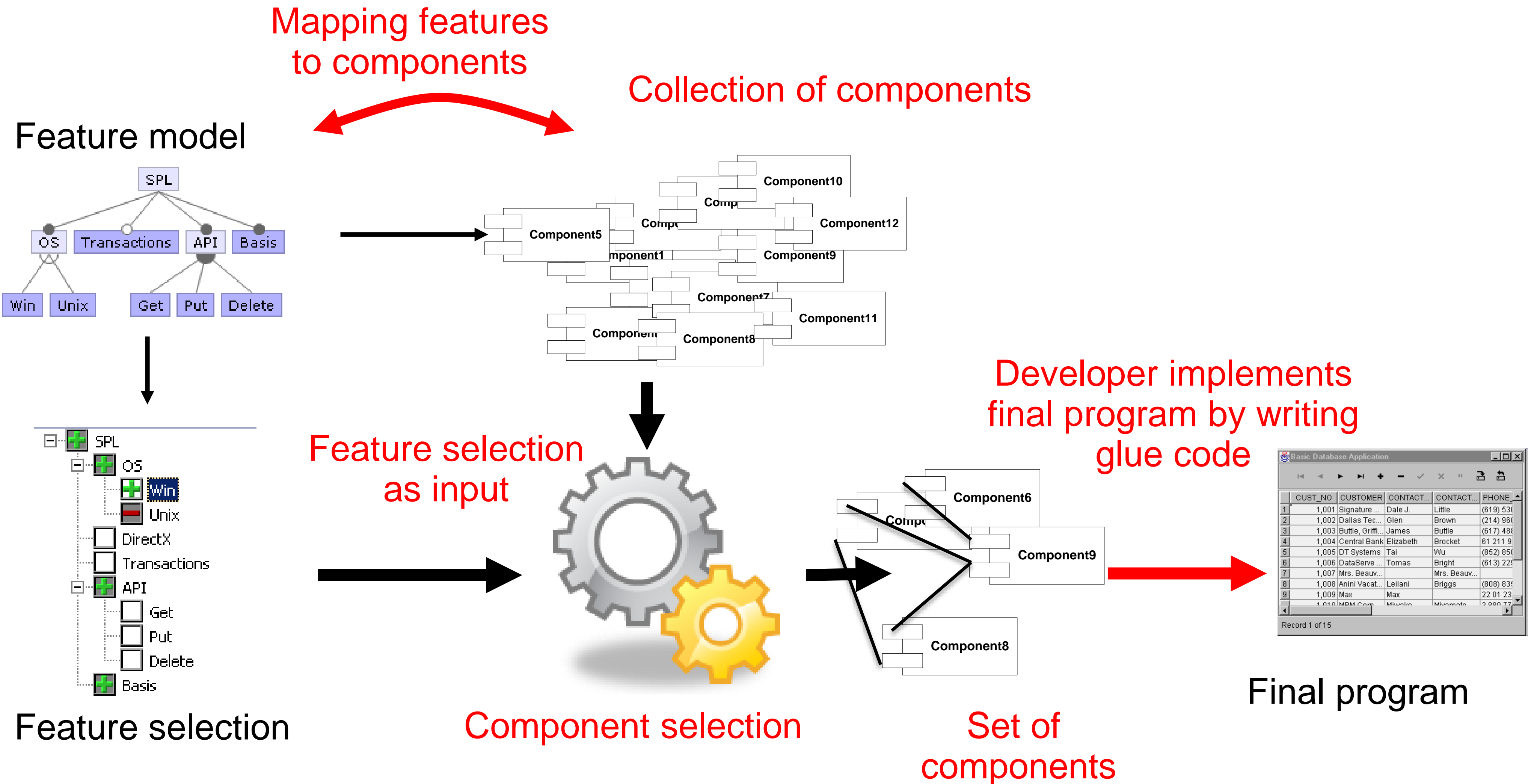
Example: Component „Color“ in Java

```
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 the module

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

Product Lines from Components



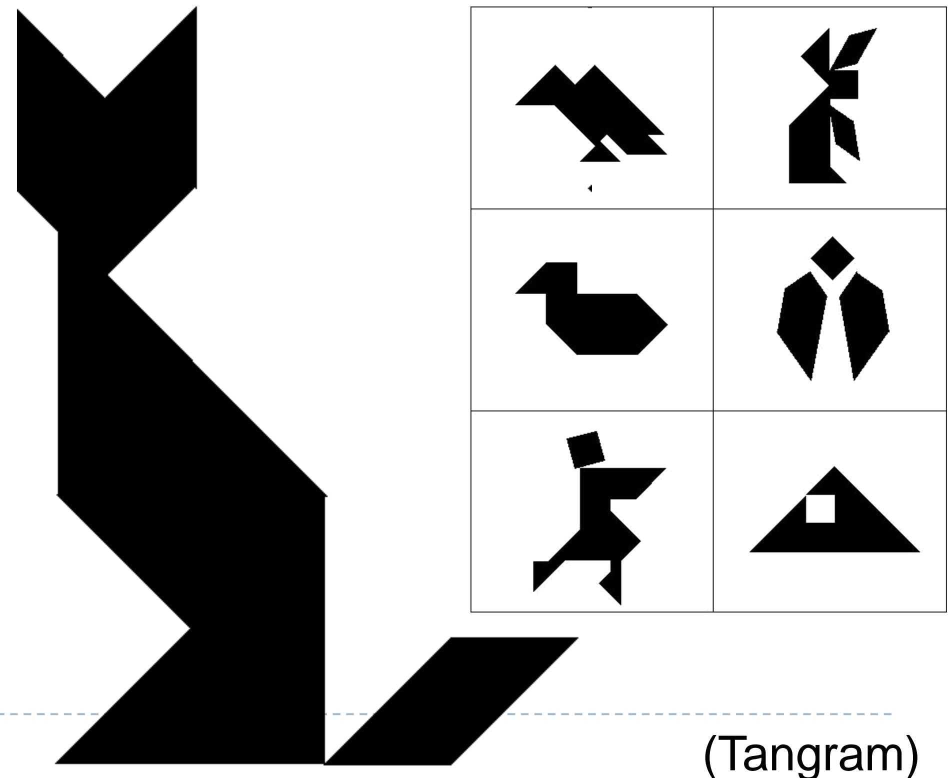
Class vs. Component vs. Package vs. Module

- Class: Blue print for generating objects; important properties are inheritance, encapsulation, polymorphism
- Package: Namespace that lets you create equally named classes in different context; coarse-grained encapsulation
- Component: Can map to a Jar file with a well-defined interface; stand-alone compliable and executable
- Module: Weakly defined term; could be a component, a package, a library, etc.

- Term component can be also different things: EJB, GUI AWT base class, JComponent in Swing, etc.

How to Define a Component?

- Marketplace for arbitrary components does not work
- Too small components → high connection effort
- Too large components → not reusable
- Product line techniques provide necessary domain analysis technique
  - Which functionality in which level of granularity will be reused
  - Systematic reuse



(Tangram)

Discussion: Product Lines from Components

- Typical technique in industry (e.g., home electronic with Koala components from Phillips)
- Systematic (planned) reuse of components
- Reuse in the large (compared to classes)
- Easy share and distribution of work

- No complete automation and high development effort in application engineering (writing manual glue code is always needed)
- No free feature selection

Discussion: Modularity

- Components hide implementation details
- Ideally small interfaces
- Feature cohesion

but …

- Coarse granularity
  - Page-replacement strategies, search algorithms, locking in B-tree, or VARCHAR as components?
  - Colors or weighted edges in the graph as components?
- Functionality possibly hard to encapsulate
  - Transaction management as component?

Services and Service-oriented Architectures (SOA)

- Similar to components: encapsulate partial functionality (services)
- Often in distributed (Web-based) scenarios
  - Bus communication, Web Services, SOAP, REST…
- Abstracts from the concrete programming language by using XML as representation and exchange format
- Product lines via connecting of related services using orchestration (workflow languages, such as BPEL)
- Many tools available (partially “management suitable”)

Frameworks

Frameworks

- Set of abstract and concrete classes
- Abstract structure that can be tailored/extended for a specific use case
  - See template method pattern und strategy pattern
- Reusable solution for a family of problems in a domain
- Specific regions in the code for extending functionality: **hot spots** (also known as variation points, extension points)
- Reversed control flow: the framework decides about the execution order
  - Hollywood principle: „Don’t call us, we’ll call you.“

Plugins

- Extensions to frameworks
- Enable the addition of specific functions
- Usually individually compilable; third-party
- Example: e-mail clients, visualization tools, media player, Web browser

Web Portal

- Web app – frameworks, such as Struts, which provide basic concepts and preimplementations
- Developer focus on application logic instead of navigation between Web pages

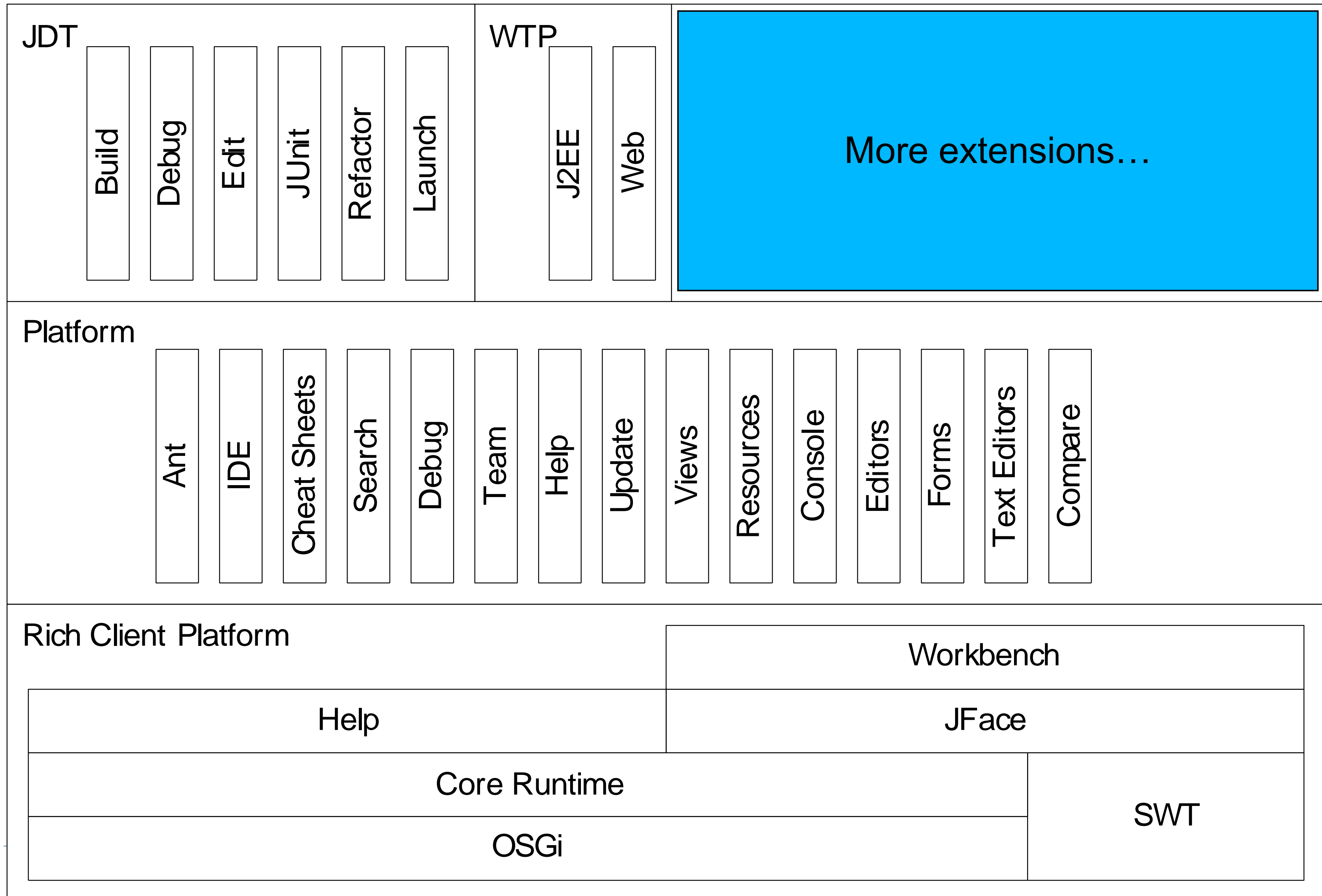
```
<?php
class WebPage {
        function getCSSFiles();
        function getModuleTitle();
        function hasAccess(User u);
        function printPage();
}
?>
```

```
<?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 framework for IDEs
  - Only language-specific extensions (syntax highlighting, compiler) must be implemented
  - The shared part is given by the framework (editors, menues, projects, folder tree, copy & paste & undo operations, CVS, etc.)
  - Framework composed of many small frameworks

Eclipse

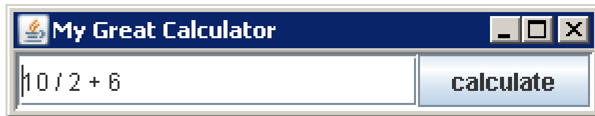


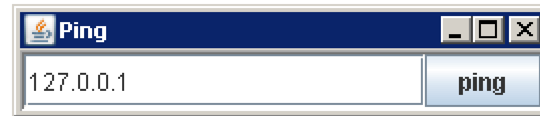
Additional Framework Examples

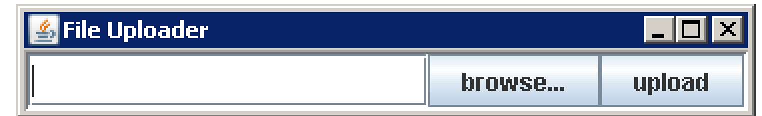
- Frameworks for graphical user interfaces, such as MacApp, Swing, SWT, MFC
- Multimedia frameworks, e.g., DirectX
- Instant messenger frameworks, such as Miranda, Trillian, ...
- Compiler frameworks, such as Polyglot, abc, JastAddJ

Framework Implementation: Mini Example

- Family of dialogs, consisting of Text field and Button







- 90% of the source code is identical
  - Main method
  - Initializing of window, text field, button
  - 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");
	  // code zum schliessen des fensters
	}
}
```


Source code of all variants nearly identical, only read source code is different (hot spots)

White-Box Frameworks

- Extensions via overriding and addition of methods (see template method pattern)
- Internals of the framework needs to be known
  - → hard to learn
- Flexible extensions possible
- Many subclasses required → can be confusing
- State of superclass is accessible (if not private)
- No plugins, not compilable in isolation

Calculator as White-Box Framework

```
public abstract class Application 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 zum schliessen des fensters
        }
        protected String getInput() { return textfield.getText();}
}
```

Calculator as White-Box Framework

Modularity?

```
public abstract class Application 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 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

- Embedding of application-specific behavior via components having a specific interface (**plugin**)
  - See strategy pattern, observer pattern
- Only the interface needs to be known
  - Easier to learn, but harder to design
- Flexibility via provided hot spots (often by using design patterns)
- State of framework classes are only accessible when provided in the interface
- In total, better reusability (?)

Calculator

```
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);
}
```

Calculator

Modularity?
Application does not know plugins

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

Further Decoupling Possible

Modularity?
Only plugin and InputProvider interface

```
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 BorderLayout(
                contentPane.setBorder(new BevelBorder(BevelBorder.
                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.
                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 Application(new CalcPlugin()).setVisible(true); }}
```

Load Plugins

- Typical for many frameworks: Plugin Loader …
  - … searches in folders for DLL/Jar/XML files
  - … tests whether a file implements a plugin (interface)
  - … checks dependencies
  - … initializes plugin
- Often, addition GUI for plugin configuration
- Examples:
  - Eclipse (plugin folder + Jar)
  - Miranda (plugin folder + DLL)
- Alternative: Define plugins in configuration files or generate them via a starter program

Sample Plugin Loader (uses 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 Plugins

- See observer pattern
- Load and register multiple plugins
- In case of events, inform all plugins

```
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: Specify different plugin interfaces

Frameworks for Product Lines

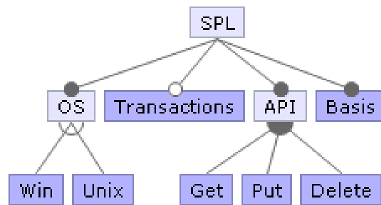
Domain Eng.

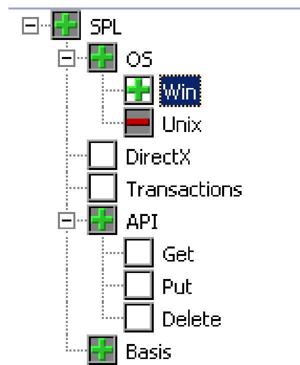
Application Eng.

Mapping from features to plugins

Framework + Plugins

Feature model





Feature selection as input

Feature selection

Plugin selection (and generated starting configuration if needed)

Application = Framework with required plugins

Discussion: Frameworks

- Full automation is possible
- Modularity
- Successfully used in practice
- High effort to build the framework and possible runtime overhead using the framework
- Preplanning required to design the framework + interfaces; requires experience
- Hard to maintain and evolve
- **Coarse granularity** or having huge interfaces
  - Plugin for transaction management, VARCHAR or colored nodes and weighted edges?

Crosscutting Concerns

Crosscutting Concerns

- Claim: Not all concerns (features) can be modularized with objects (or components and plugins)

- Concerns are semantically coherent units
- But, their implementation is sometimes distributed, mixed, and replicated in the whole code base

Crosscutting Concerns -- Example

```
class DatabaseApplication
  //... Data fields
  //... Logging Stream
  //... Cache Status
  public void authorizeOrder(
        Data data, User currentUser, ...){
    // check authentication
    // Lock object for synchronization
    // Check status of buffer
    // Log start of operation
    // execute actual operation
    // Log end of operation
    // Release lock of object
  }
  public void startShipping(
        OtherData data, User currentUser, ...){
    // check authentication
    // Lock object for synchronization
    // Check status of buffer
    // Log start of operation
    // execute actual operation
    // Log end of operation
    // Release lock of object
  }
}
```

- Code of different concerns is intermingled
- Replicated code
- Here, operations are modular, but locking, logging, buffer management, and authentication are not

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();
  (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() {
    fo
      ((
    }
  }
}
```

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() { ... } }
```

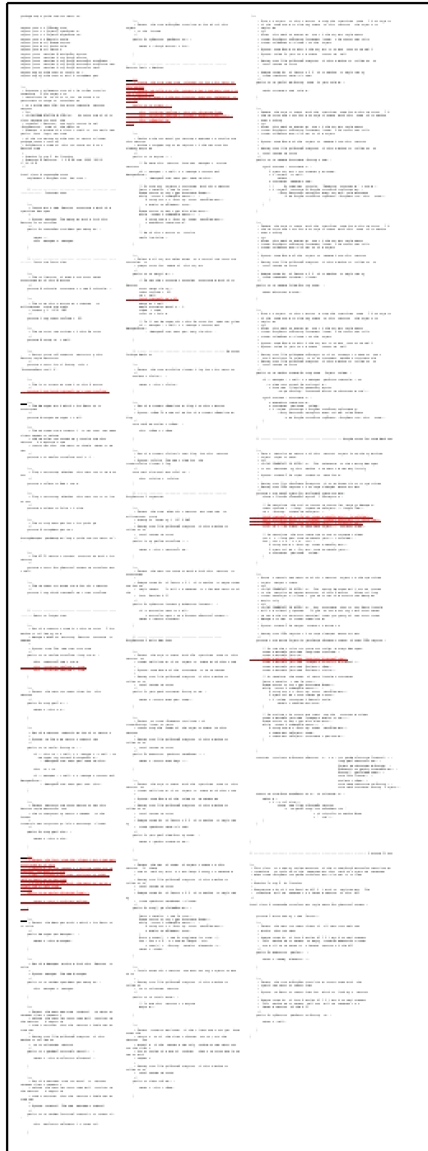
Scattering und Tangling

- Code scattering
  - Code belonging to a concern is not modularized, but scattered through the whole code base
  - Often, code is copied (even though it might be just a single method call)
  - Or, heavily distributed implementation of complementary parts of a concerns
- Code tangling
  - Code belonging to different concerns are mixed within the same module (or method)

A Question of Size

ApplicationSession

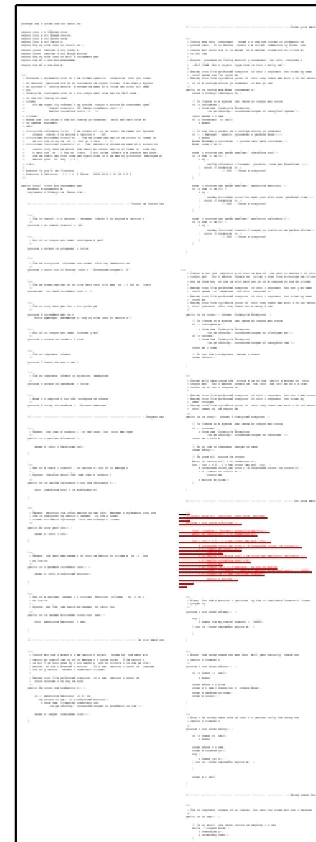
StandardSession



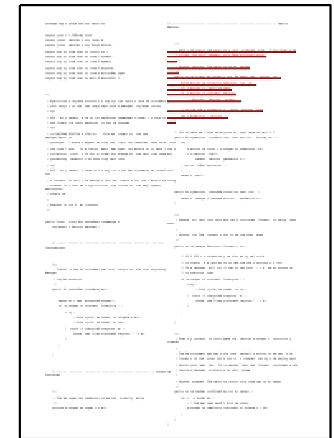
Example: Time management of sessions in the Apache Tomcat Server
As part of the session management

SessionInterceptor

StandardManager

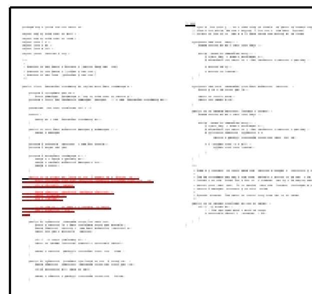


StandardSessionManager



ServerSession

ServerSessionManager



Recap: Problems of Crosscutting Concerns

- Concerns vanish in the implementation
  - What parts belongs to a single concern?
  - Maintenance requires to search through the whole code base
- Difficult division of work
  - Different concerns might have different experts, but all have to work at the same code fragment
- Low productivity and difficult evolution
  - Adding new functionality requires to understand and maintain multiple concerns, which are not part of the actual functionality

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();
  }
}
```

- Authentication is modular
- Thereby, other operations and concerns are not modular

Next Trial – Method Calls

- Move concerns to separate modules (e.g., locking or authentication module)
  - Scattering und tangling reduced to only method calls
  - Clearer, but still explicit method calls in code
- ➔ Many extensions points in framework required, leading to large interfaces

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

Next Trial – Middleware

- Middleware takes care of concerns; developers implement only actually required functionality (inversion of control)
  - Example: Enterprise Java Beans provide distributed objects, persistency, transaction management, authentication, authorization, and synchronization
  - Complex architecture
  - Not all concerns are addressed by middleware; especially those related to the business logic

Tyranny of the Dominant Decomposition

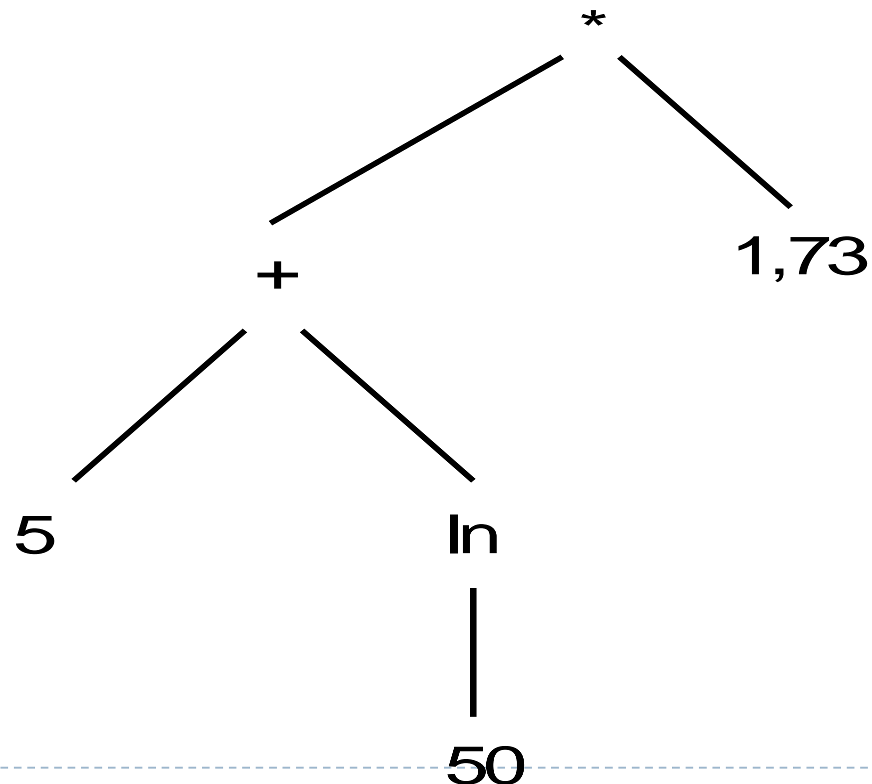
- Many concerns can be modularized, but not at the same time
  - Modularization only in one direction possible
  - For the graph example, only colors can be modularized…
  - …then, the data structures (Node, Edge) are distributed
- Developers chose the decomposition (e.g., operations, authentication, data structures), but other concerns “cut across”
- Simultaneous modularization along different dimensions not possible

Example: Expression Problem

- Question: How far can we abstract data structures and methods so that we can implement both independently…
  - Without to change existing code (or even without the need to recompile existing code)
  - In arbitrary order and
  - Without non-trivial code replication ?

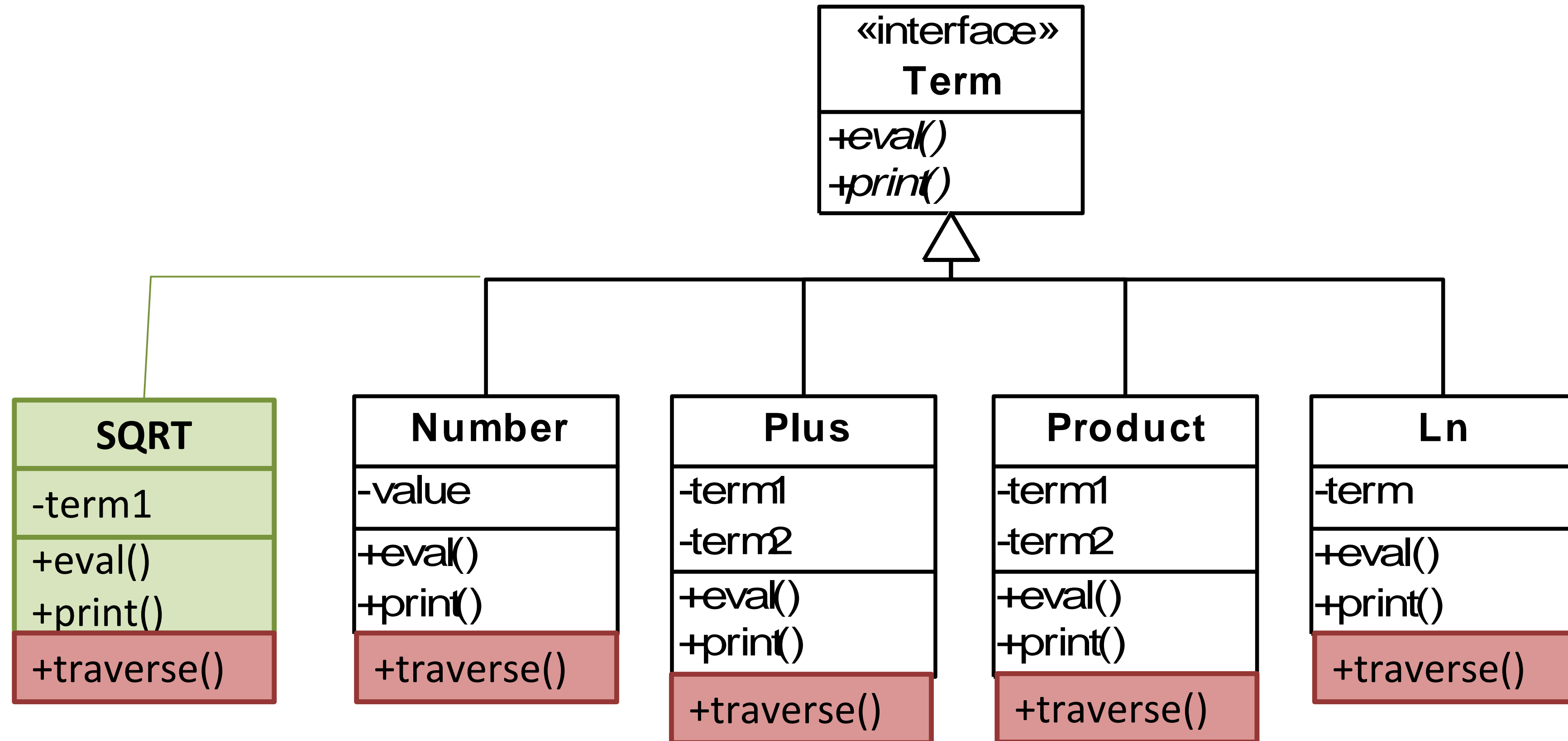
Expressions

- Task: Mathematical expressions are stored in a tree structure and can be evaluated or printed



Implementation 1: Data Centric

- Recursive class structure (composite pattern)
- For each expression, define an individual class

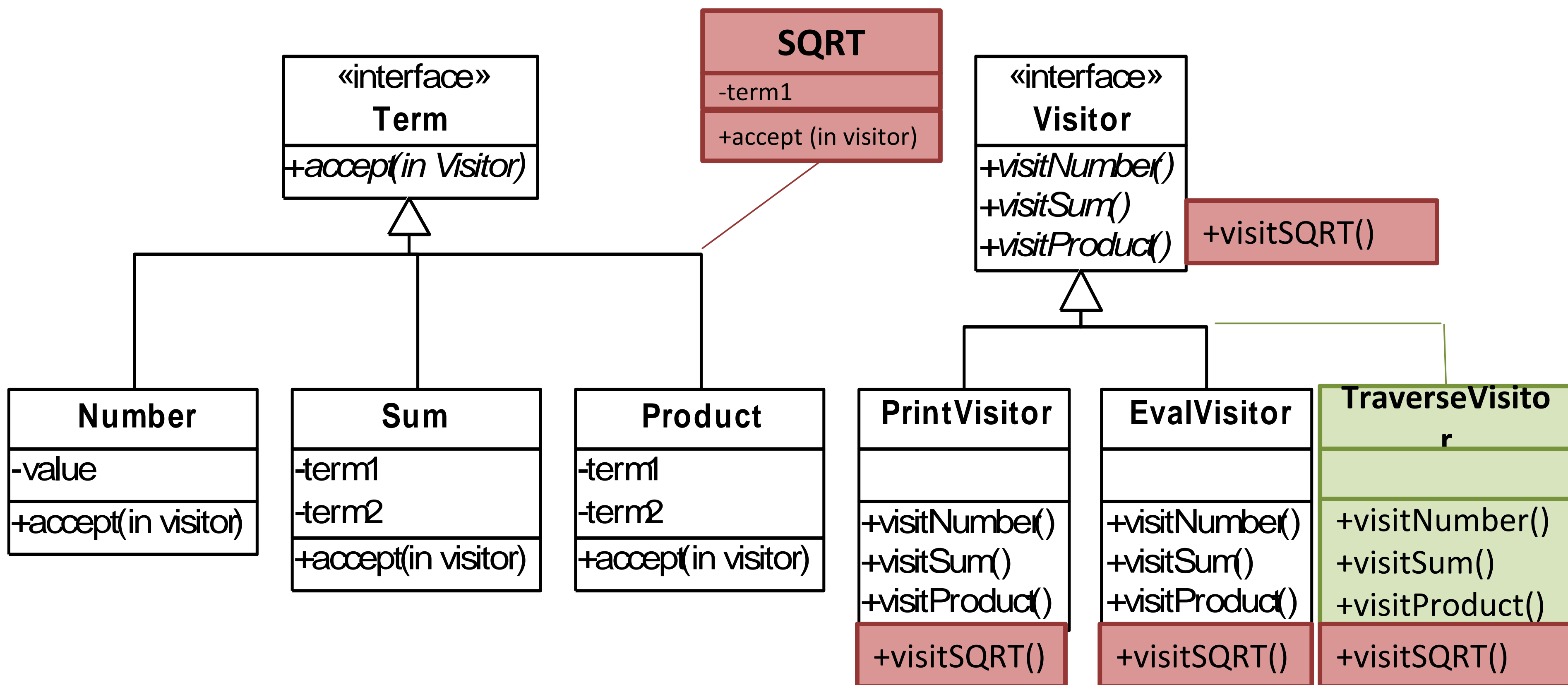


Problem of Implementation 1

- Expressions are modular
- Easy to add new expressions, such as division
- New operations, such as drawTree or simplify, cannot easily be added
- All existing classes have to be adapted!
- Operations crosscut the expressions

Implementation 2: Method Centric

- Only one method (accept) per class
- Operations are implemented via the visitor pattern



(Ln class omitted due to space limitations)

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());
```

Problem of Implementation 2

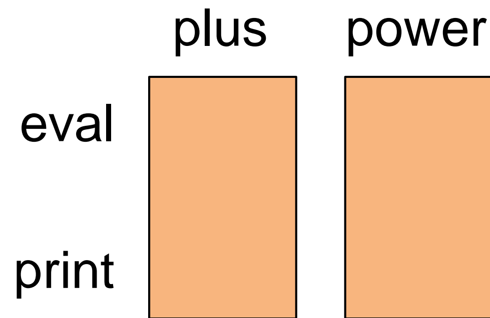
- Operations are modular
- Easy to add new operations
- New expressions, such as Min or Power cannot be easily added
- For each new class, we need to adapt all visitor classes
- Expressions crosscut operations

Expression Problem

- Hardly possible to modularize expressions and operations at the same time (complex solution with Java 1.5 generics exist)
- Data centric approach
  - New expression can be added directly: modular
  - New operations have to be added in every class: not modular
- Method centric approach
  - New operations can be added as additional visitor: modular
  - New classes (expressions) required to extended every existing visitor: not modular

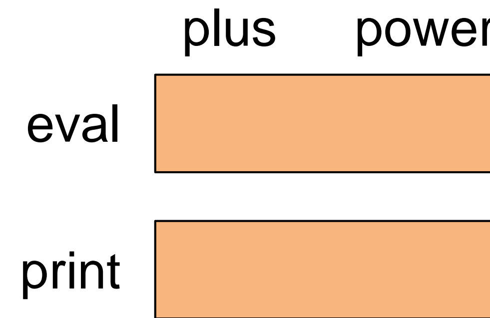
Expression Problem – Visual Comparison

Data centric

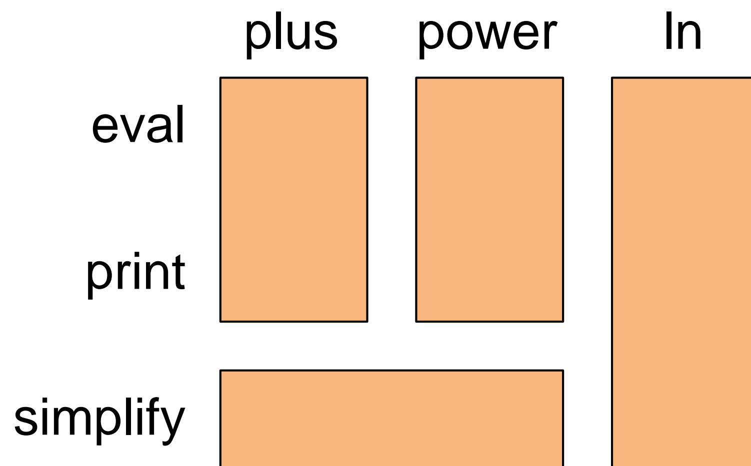


(a)

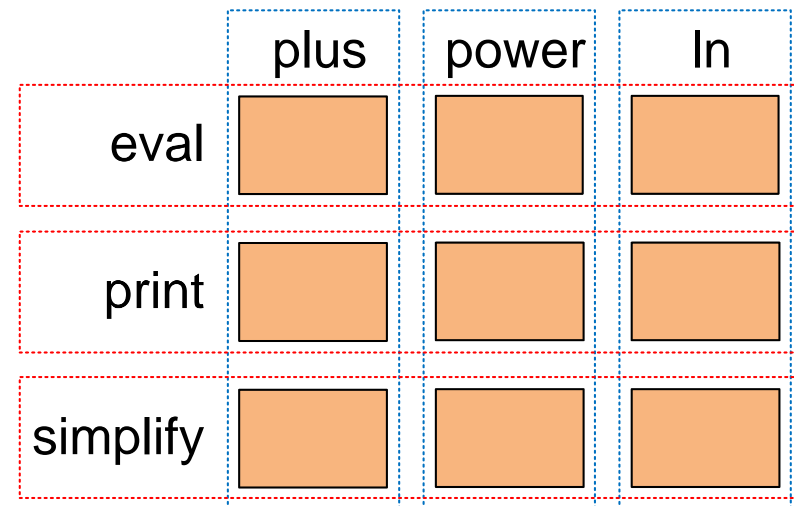
Method centric (visitor)



(b)



(c)



(d)

Outlook: New language paradigms

Outlook: Feature interactions

Typical Examples of Crosscutting Concerns

- Logging: Message for each method
- Caching/Pooling: Code at every object creation
- Synchronization/Locking: Extension of many methods with lock/unlock calls
- Features in product lines!

Dilemma

- Not always possible to modularize all concerns (tyranny …)
- Basic fraction of scattered and tangled code is common
- Some concerns are always „orthogonal“ to others: crosscutting conerns
- Features in product lines are often affected

Preplanning Problem

Preplanning Problem

- Extensions are ad-hoc not possible, but require careful preplanning
- Extension points need to be foreseen
  - Extension points in frameworks
  - Interfaces/Parameter in components
- Without matching extension point there is no modular extension possible

Preplanning Problem - Example

- Stack methods need to be synchronized
- Modular extension via 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();
  }
}
```

Unplanned extension

```
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 Discussion

- Problem: Stack instantiation needs to be adapted in basic code
  - No extension possible without changing the base code (not modular)
- Alternative
  - Design pattern: Factory instead of direct instantiation (enables modular extension)
  - Framework needs matching extension point
- Extension opportunities must be anticipated (preplanning) or retroactively added to the base code (not modular)

Summary

- Modularizing features with components and frameworks
- Usually no complete automation, runtime overhead, and coarse granularity
- Limitations of modularization in case of crosscutting concerns and fine granularity
- Modularity requires preplanning
- Not suitable for all product lines (e.g., graph library, embedded database system)

Outlook

- New programming paradigms
  - Analysis of object-oriented programming and its limitations
  - Feature-oriented programming
  - Aspect-oriented programming

Literature

- C. Szyperski: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998
  [Standard literature of component-oriented development]
- R. Johnson and B. Foote, Desiging reusable classes, Journal of Object-Oriented Programming, 1(2):22-35, 1988
  [OOP reusability, especially frameworks]
- L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, 2003
  [Architecture-driven product lines, typically via frameworks]