

Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

7a. Limitations of Object Orientation

- Recap on Modularity
- Preplanning Problem
- Crosscutting Concerns
- Tyranny of the Dominant Decomposition
- Example: Arithmetic Expressions
- Summary

7b. Feature-Oriented Programming

- Motivation
- Feature Modules
- Feature Composition
- Feature Modules in Java
- Principle of Uniformity
- Discussion
- Summary

7c. Aspect-Oriented Programming

- Recap and Motivation
- Aspects and Aspect Weaving
- Static vs. Dynamic Extensions
- Quantification
- Executing Additional Code
- Aspects for Product Lines
- Discussion
- Summary
- FAQ

7. Languages for Features – Handout

Software Product Lines | Timo Kehrer, Thomas Thüm, Elias Kuiter | June 2, 2023

7. Languages for Features

7a. Limitations of Object Orientation

Recap on Modularity

Preplanning Problem

Crosscutting Concerns

Tyranny of the Dominant Decomposition

Example: Arithmetic Expressions

Summary

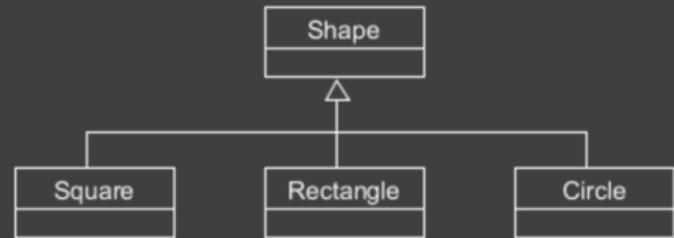
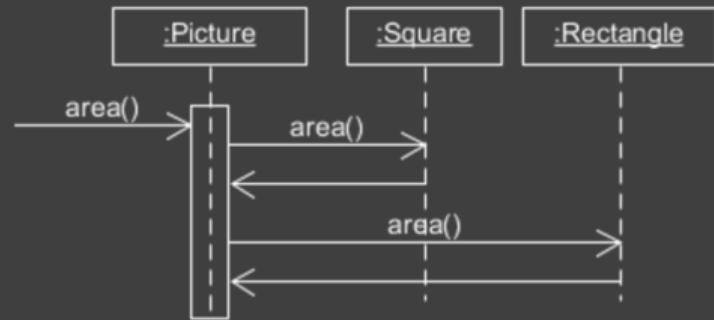
7b. Feature-Oriented Programming

7c. Aspect-Oriented Programming

Recap: Object Orientation

Key Concepts

- **Encapsulation:**
abstraction and information hiding
- **Composition:**
nested objects
- **Message Passing:**
delegating responsibility
- **Distribution of Responsibility:**
separation of concerns
- **Inheritance:**
conceptual hierarchy, polymorphism, reuse



Recap: Design Patterns [Gang of Four]

Design Patterns

- Document common solutions to concrete yet frequently occurring design problems
- Suggest a concrete implementation for a specific object-oriented programming problem

Design Patterns for Variability

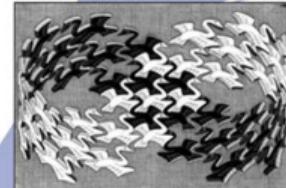
Many Gang of Four (GoF) design patterns for designing software around stable abstractions and interchangeable (i.e., variable) parts, e.g.

- Template Method
- Abstract Factory
- Decorator

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Recap: Modularity - Components, Services, and Frameworks

Component-/Service-Based SPLs



Specification of “composition” (glue code, orchestration)



Framework-Based SPLs



Neither glue code nor service composition required.

Example: Extending Basic Graphs by Plug-Ins?

```
public class Graph {  
    private List<GraphPlugin> plugins = new ArrayList<GraphPlugin>();  
    // ...  
    public void registerPlugin(GraphPlugin p){  
        plugins.add(p);  
    }  
    public void addNode(int id, Color c){  
        Node n = new Node(id);  
        notifyAdd(n, c);  
        nodes.add(n);  
    }  
    public void print() {  
        for (Node n : nodes) {  
            notifyPrint(n);  
            // ...  
        }  
        // ...  
    }  
    private void notifyAdd(Node n, Color c) {  
        for (GraphPlugin p : plugins) {  
            p.aboutToAdd(n, c);  
        }  
    }  
    private void notifyPrint(Node n) {  
        for (GraphPlugin p : plugins) {  
            p.aboutToPrint(n);  
        }  
        // ...  
    }  
}
```

```
public interface GraphPlugin {  
    public void aboutToAdd(Node n, Color c);  
    public void aboutToAdd(Edge e, Weight w);  
    public void aboutToPrint(Node n);  
    public void aboutToPrint(Edge e);  
}
```

```
public class ColorPlugin implements GraphPlugin {  
    private Map<Node, Color> map = new HashMap<Node, Color>();  
  
    public void aboutToAdd(Node n, Color c) {  
        map.put(n, c);  
    }  
  
    public void aboutToAdd(Edge e, Weight w) {  
        // do nothing  
    }  
  
    public void aboutToPrint(Node n) {  
        Color c = map.get(n);  
        Color.setDisplayColor(c);  
    }  
  
    public void aboutToPrint(Edge e) {  
        // do nothing  
    }  
}
```

Challenges and Problems

In our example, we can observe that:

- There are lots of empty methods in the ColorPlugin
- The Framework consults all registered plug-ins before printing a node or edge

General Challenge: Cross-cutting Concerns

Implementing cross-cutting concerns as plug-ins

- typically leads to huge interfaces, large parts of which are irrelevant for a dedicated plug-in
- causes lots of communication overhead between plug-ins and framework

If we were not familiar with our graph library, would we anticipate that:

- Colors and weights should be part of the Plugin interface?
- Every plug-in needs to be notified that the framework is about to print a node or edge?

Generally known as Preplanning Problem

- Hard to identify and foresee the relevant hot spots and nature of extensions
- Developing a framework needs lots of expertise and excellent domain knowledge

Preplanning Problem

Components, Services, and Frameworks

Extensions are not possible ad-hoc, but must be foreseen and planned in advance:

- Frameworks: Hot spots / extension points
- Components/services: Provided and required interfaces

⇒ **No modular extension without a suitable extension point / interface!**

And classical OO language concepts?

Subtyping and polymorphism allow for ad-hoc extensions to some extent, but:

- Often, client code and/or basic implementation need to be adapted, too (non-modular)
- Design patterns require preplanning of potential future extensions
- Limited flexibility of inheritance hierarchies (no “mix and match” supporting arbitrary feature combinations)

⇒ **No variable extension without loosing modularity!**

Crosscutting Concerns

Concern

[Apel et al. 2013, pp. 55]

A concern is an area of interest or focus in a system, and features are the concerns of primary interest in product-line engineering.

Crosscutting (Concern)

[Apel et al. 2013, pp. 55]

Crosscutting is a structural relationship between the representations of two concerns. It is an alternative to hierarchical and block structure.

Tyranny of the Dominant Decomposition

Tyranny of the Dominant Decomposition

- Many concerns can be modularized, but not always at the same time.
- Developers choose a decomposition, but some other concerns cut across.
- Simultaneous modularization along different dimensions is not possible.

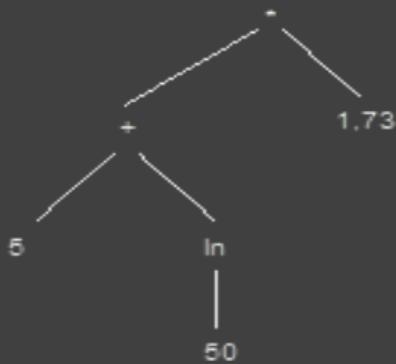
Crosscutting concerns are inherently difficult to separate using traditional mechanisms.

- Logging: Each time a method is called.
- Caching/Pooling: Each time an object is created.
- Synchronization/Locking: Extension of many methods with lock/unlock calls.

Features in a software product line are often cross-cutting (e.g., color and weight in our graph example).

Example: Arithmetic Expressions

$(5 + \ln(50)) * 1.73$



- Arithmetic expressions are stored in a tree structure.
- Terms (i.e., sub-trees) can be evaluated and printed.

Question

How to separate data structures and operations such that both can be extended independently of each other?

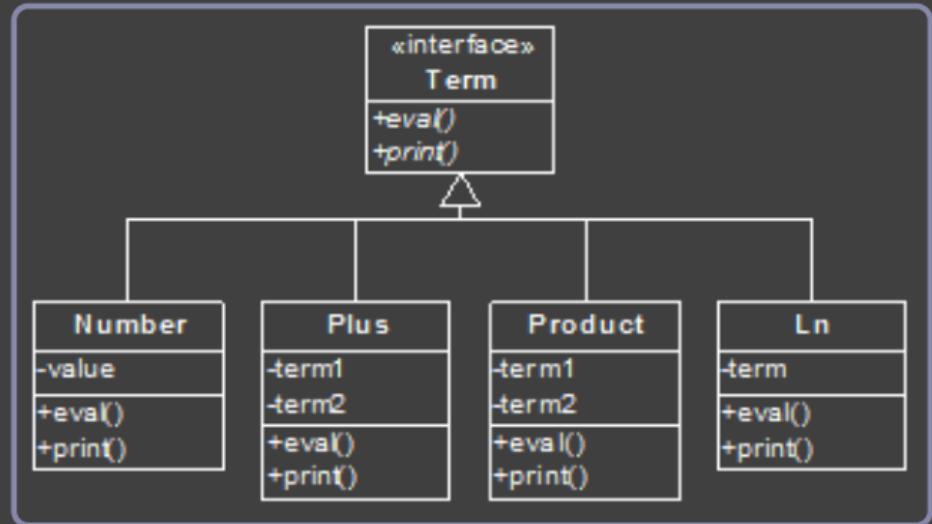
Arithmetic Expressions – Data-Centric Decomposition

Implementation Variant 1: Data-Centric

- Recursive class structure (composite pattern)
- For each operation (eval, print, ...) there is a dedicated method in each class (Number, Plus, ...)

Terms are modular, but...

- New operations, e.g. drawTree or simplify, cannot simply be added
- All existing classes must be adjusted!
- Operations cut across the expressions.



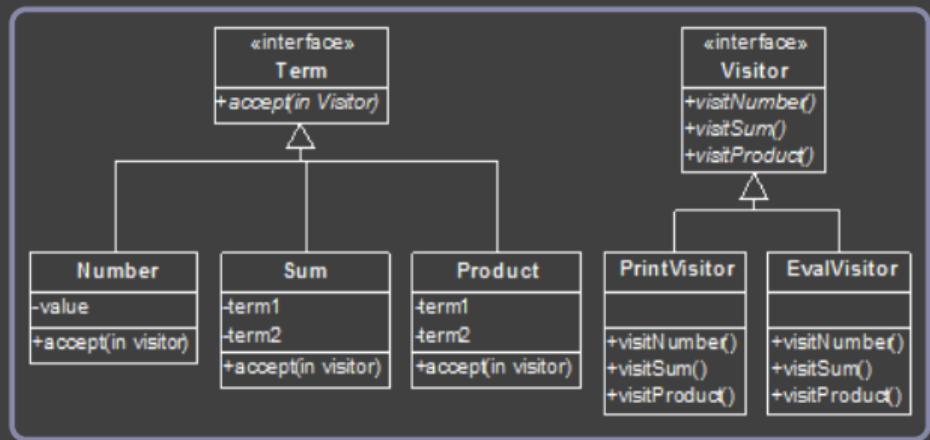
Arithmetic Expressions – Operation-Centric Decomposition

Implementation Variant 2: Operation-Centric

- Just a single accept method per class (visitor pattern).
- Each operation is implemented by a dedicated visitor.

Operations are modular, but...

- New expressions, e.g. Min or Power, cannot simply be added
- For each new class, all visitor classes must be adjusted
- Expressions cut across operations



Lessons Learned from the Simple Example

Hardly possible to modularize expressions and operations at the same time!

Data-Centric Decomposition

- New expressions can be added directly: modular.
- New operations must be added to all classes: not modular.

Operation-Centric Decomposition

- New operations can be added as another visitor: modular.
- For new expressions, all existing visitors must be extended: not modular.

Limitations of Object Orientation – Summary

Lessons Learned

Important problems of previous approaches:

- Inflexible inheritance hierarchies (especially with runtime variability, frameworks, components, services)
- Feature traceability (especially with runtime variability, branches, build systems, preprocessors)
- Preplanning problem (esp. with frameworks, components, services)
- Cross-cutting issues (esp. with frameworks, components, services)

Practice

Looking at our graph implementation serving as running example throughout the course:

- Which concern is the dominant one regarding modular decomposition?
- What are crosscutting concerns?
- Can we restructure the implementation to come up with a different decomposition?

Further Reading

s. previous lectures

7. Languages for Features

7a. Limitations of Object Orientation

7b. Feature-Oriented Programming

Motivation

Feature Modules

Feature Composition

Feature Modules in Java

Principle of Uniformity

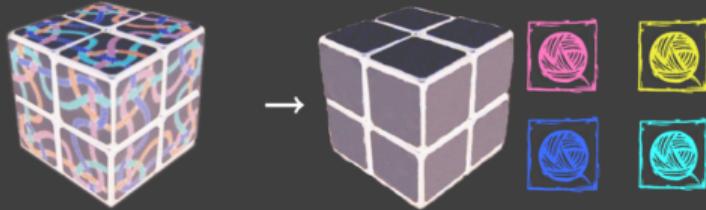
Discussion

Summary

7c. Aspect-Oriented Programming

Motivation

Modularization of Cross-Cutting Concerns



Flexible Extension / Minimal Preplanning



Feature Traceability



find feature  in product



Achieving all this requires novel implementation techniques that overcome the limitations of classical object-oriented paradigms.

Background: Collaboration-Based Design

Inspiration: Collaborations in the Real World

- People collaborate to achieve a common goal.
- A collaboration typically comprises several persons playing different roles.
- Persons may play multiple roles by participating in different collaborations.

Mentor-Student Collaboration

- A person in the role of a mentor has responsibilities to instruct students on certain topics.
- A person in the role of a student has responsibilities to study the offered material.

Collaborations in Java

[Apel et al. 2013, pp. 131]

- A **collaboration** is a set of interacting classes, each class playing a distinct role, to achieve a certain function or capability.
- A **role** defines the responsibilities a class takes in a collaboration.

- Different classes play different roles within a collaboration.
- A class plays different roles in different collaborations.
- A role encapsulates the behavior/functionality of a class relevant to a collaboration.

Example: Collaborations, Classes and Roles

		Classes				
		Graph	Edge	Node	Weight	Color
Collaborations	Base	Graph Edge add(Node,Node) void print()	Edge Node a,b void print()	Node void print()		
	Directed	Graph Edge add(Node,Node)	Edge Node start void print()			
	Weighted	Graph Edge add(Node,Node) Edge add(Node,Node,Weight)	Edge Weight weight void print()		Weight ...	
	Colored			Node Color color void print()		Color ...

Feature Modules and Feature Module Composition

Feature Modules

- Each collaboration mapped to a feature and is called a feature module (or layer).
- Feature modules may refine a base implementation by adding new elements or by modifying and extending existing ones.

Feature Module Composition

Selected feature modules may be superimposed by lining-up classes according to the roles they play.



Feature Modules and Feature Module Composition

Open Questions

- How to bundle classes to feature modules and specify their refinements?
- How to handle refinements during composition of feature modules?



Jak: A Java Extension for Feature-Oriented Programming

[Batory et al. 2004]

Layers

- keyword **layer** denotes the feature a class belongs to
- layer = feature module = collaboration

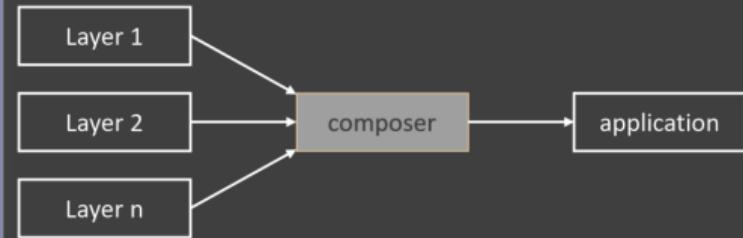
Class Refinement

A class refinement (keyword **refines**) can add new members to a class and extend existing methods.

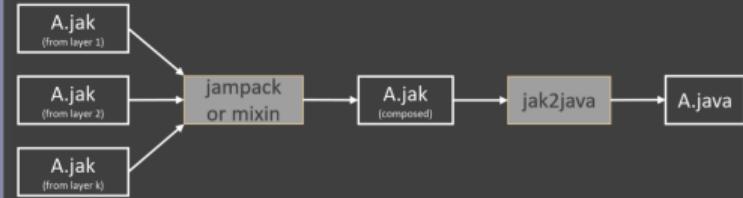
Composer

- AHEAD (Algebraic Hierarchical Equations for Application Design) + jampack/mixin
- Application constructed by composing layers
- Internally, the composer invokes a variety of tools to perform its task

Composer (High-Level View)



Composer (Jak File Composition)



Jak: Layers



- Layer (i.e., feature module) Base consists of the classes Graph, Node, and Edge
- The three classes collaborate to provide the functionality to construct and display graph structures.

Graph.jak

```
class Graph {  
    private List nodes = new ArrayList();  
    private List edges = new ArrayList();  
  
    Edge add(Node n, Node m) {  
        Edge e = new Edge(n, m);  
        nodes.add(n); nodes.add(m); edges.add(e);  
        return e;  
    }  
    void print() { ... }  
}
```

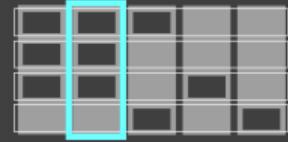
Node.jak

```
class Node {  
    private int id = 0;  
  
    void print() {  
        System.out.print(id);  
    }  
}
```

Edge.jak

```
class Edge {  
    private Node a, b;  
  
    Edge(Node a, Node b) {  
        this.a = a; this.b = b;  
    }  
    void print() {  
        a.print(); b.print();  
    }  
}
```

Jak: Class Refinement – New Members



Mixin-Based Inheritance

Subclasses are abstract in the sense that they can be applied to **different** concrete superclasses.

Refinement Chain

A refinement chain is a linear inheritance chain where the bottom-most class of the chain is the only class that is meant to be instantiated.

New Members

A stepwise refinement can add new members (i.e., fields and methods) to a class.

Edge.jak

```
class Edge {  
    ...  
}
```

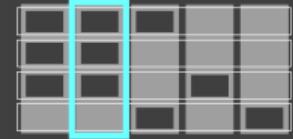
Edge.jak

```
refines class Edge {  
    private Node start;  
    ...  
}
```

Edge.jak

```
refines class Edge {  
    private Weight weight;  
    ...  
}
```

Jak: Class Refinement – Method Extensions



Mixin-Based Inheritance

Subclasses are abstract in the sense that they can be applied to **different** concrete superclasses.

Refinement Chain

A refinement chain is a linear inheritance chain where the bottom-most class of the chain is the only class that is meant to be instantiated.

Method Extension

A method extension is implemented by method overriding and calling the overridden method via the keyword Super.

Edge.jak in Layer Base

```
class Edge {  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
    }  
}
```

Edge.jak in Layer Directed

```
refines class Edge {  
    ...  
    void print() {  
        Super().print();  
        System.out.print(" directed from " + start);  
    }  
}
```

Edge.jak in Layer Colored

```
refines class Edge {  
    ...  
    void print() {  
        Super().print();  
        System.out.print(" weighted with " + weight);  
    }  
}
```

AHEAD: Composition Using Jampack

Jampack

- Jampack superimposes the refinement chain into a single class.
- Super calls are integrated by method inlining (cf. optimization in compiler construction).

Composition Result

```
class Edge {  
    private Node start;  
    private Weight weight;  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
        System.out.print(" directed from " + start);  
        System.out.print(" weighted with " + weight);  
    }  
}
```

AHEAD: Composition Using Mixin

Mixin

- Mixin retains layer relationships as an inheritance chain.
- Produces a single file that contains a (linear) inheritance hierarchy where only the bottom-most class is “public”.
- Super calls are integrated by method inlining (cf. optimization in compiler construction).

Do Not Confuse with Mixins

mixins are a language concept to decompose classes into parts without inheritance and rather similar to Jampack

Composition Result

```
class Edge$$Base {  
    void print() { ... }  
}  
class Edge$$Directed extends Edge$$Base {  
    private Node start;  
    void print() {  
        super.print();  
        System.out.print(" directed from " + start);  
    }  
}  
class Edge extends Edge$$Directed {  
    private Weight weight;  
    void print() {  
        super.print();  
        System.out.print(" weighted with " + weight);  
    }  
}
```

Jampack vs. Mixin

Jampack

- Assignment of generated code to roles disappears after generation
- Local variables can be accessed from within refined methods

Mixin

- Code overhead and method call indirections negatively impact runtime performance
- Feature modularity preserved even after composition

Jampack and Mixin in Practice

Recommended Usage

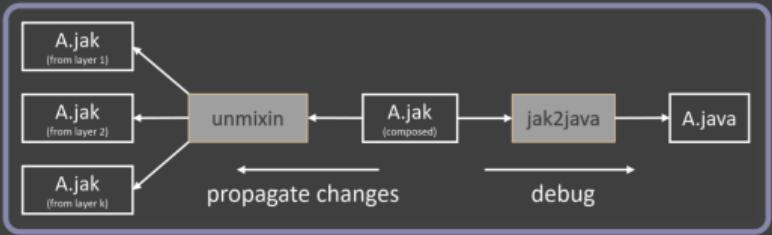
- Use Mixin during development and iterative refinement (debugging)
- Use Jampack when a production version of a class is to be produced (performance)

Unmixin

Automatically propagates changes from the composed .jak file back to its original layer files

Un mixin and Debugging

- Make changes to a mixin-composed .jak file during debugging
- Then automatically back-propagate changes to the layer files



Composition: Order Matters!

(a) Edge.jak

```
class Edge {  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
    }  
}
```

(b) Edge.jak

```
refines class Edge { ...  
    void print() {  
        Super().print();  
        System.out.print(" directed from " + start);  
    }  
}
```

(c) Edge.jak

```
refines class Edge { ...  
    void print() {  
        Super().print();  
        System.out.print(" weighted with " + weight);  
    }  
}
```

Class refinements themselves are (largely) independent of the order in which they are eventually composed.

Composition Order (a), (b), (c)

```
class Edge {  
    private Node start;  
    private Weight weight;  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
        System.out.print(" directed from " + start);  
        System.out.print(" weighted with " + weight);  
    }  
}
```

However, the order in which features are applied is important (e.g., earlier features in the sequence may add elements that are refined by later features).

Composition: Order Matters!

(a) Edge.jak

```
class Edge {  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
    }  
}
```

(b) Edge.jak

```
refines class Edge { ...  
    void print() {  
        Super().print();  
        System.out.print(" directed from " + start);  
    }  
}
```

(c) Edge.jak

```
refines class Edge { ...  
    void print() {  
        Super().print();  
        System.out.print(" weighted with " + weight);  
    }  
}
```

Class refinements themselves are (largely) independent of the order in which they are eventually composed.

Composition Order (a), (c), (b)

```
class Edge {  
    private Node start;  
    private Weight weight;  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
        System.out.print(" weighted with " + weight);  
        System.out.print(" directed from " + start);  
    }  
}
```

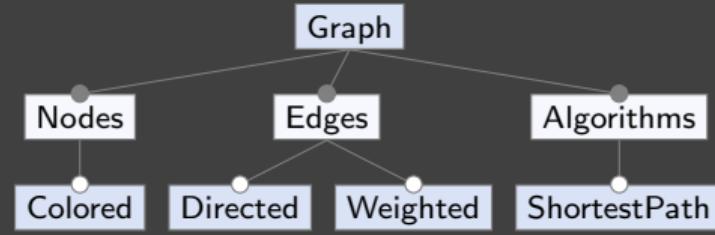
However, the order in which features are applied is important (e.g., earlier features in the sequence may add elements that are refined by later features).

Composition: Order Matters!

The order in which compositions are to be applied is an input parameter of the composition tool.

Composition Order in FeatureIDE

- In FeatureIDE, a total order can be defined based on the feature model
- Default: depth-first traversal of the feature model (only concrete features)

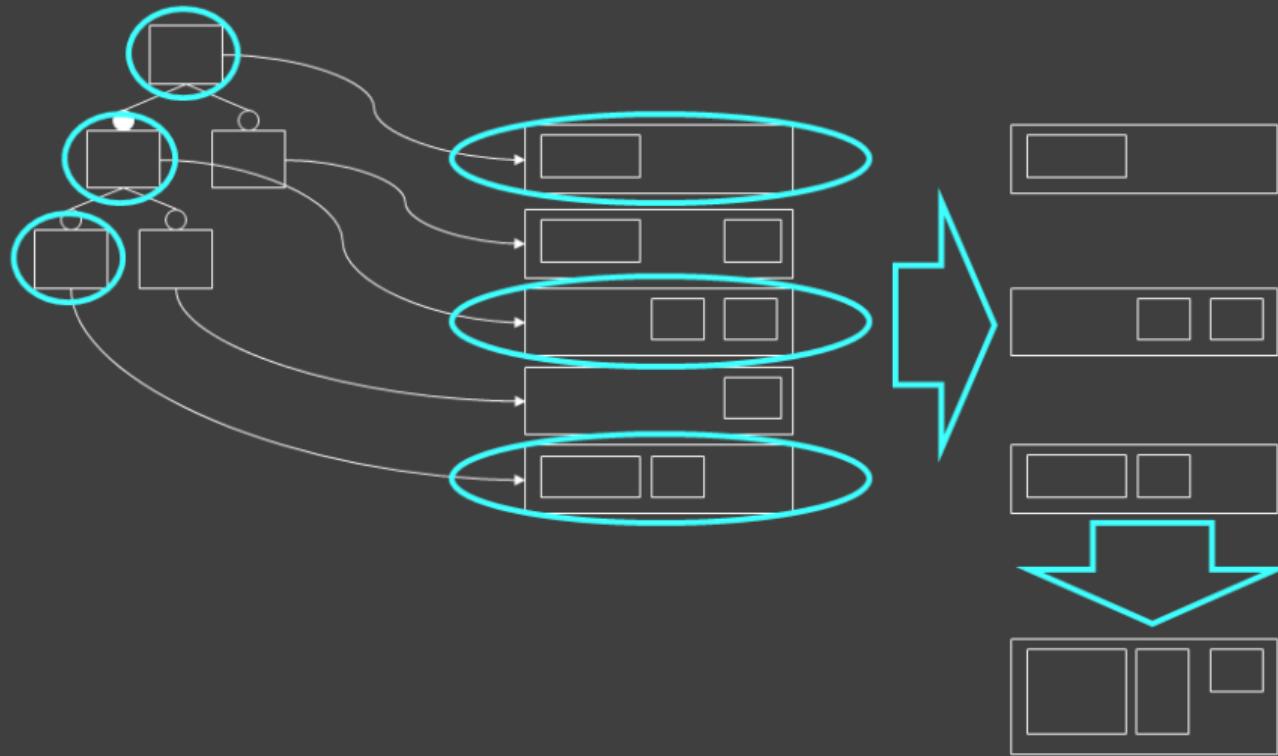


ShortestPath → Weighted

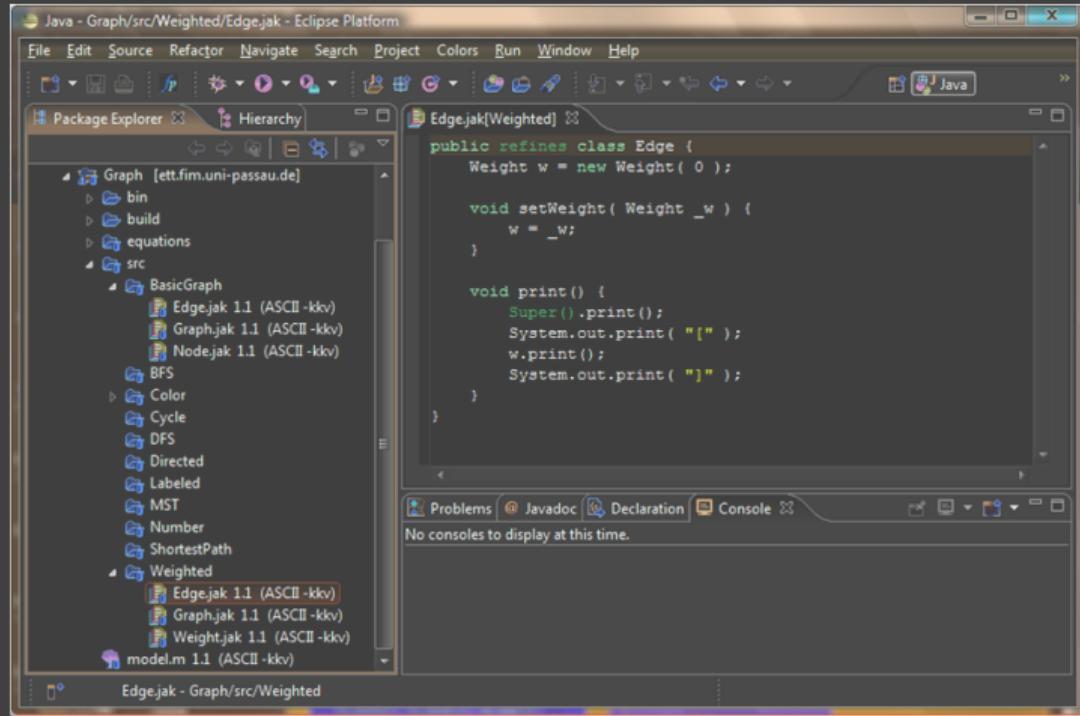
Default Order for Example

Graph, Colored, Directed, Weighted, ShortestPath

Big Picture: Product-Line Implementation and Product Generation



Practical Organization of Feature Modules



Beyond Jak: Uniformity

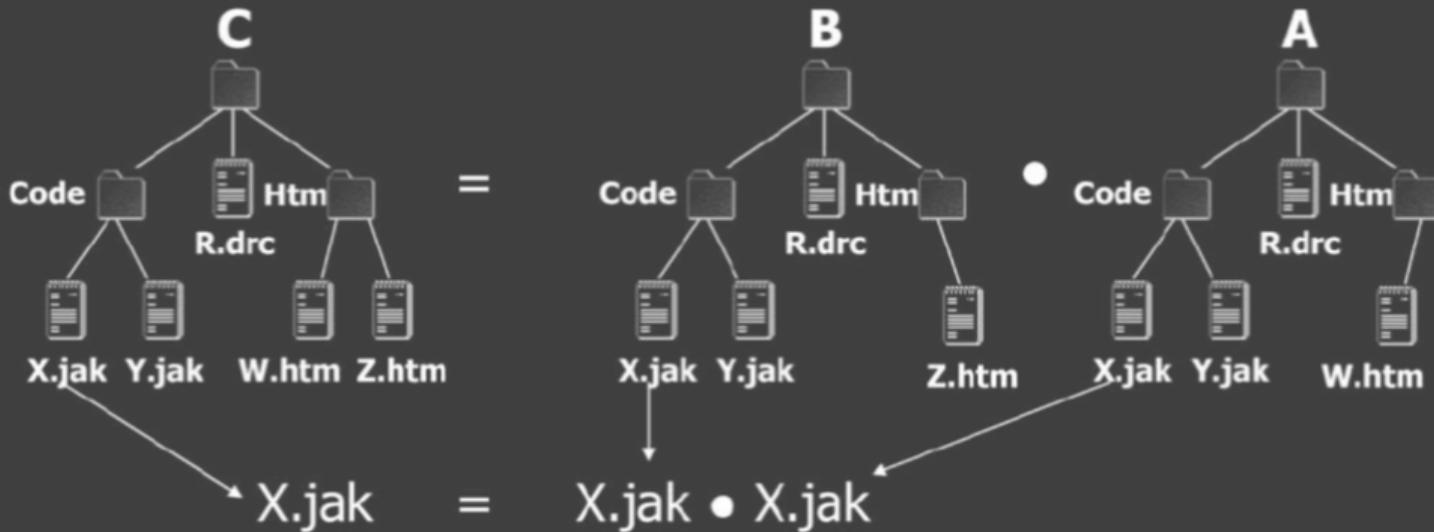
Motivation

- Software does not only consist of Java source code, but also other programming languages, build scripts, documentation, models, etc.
- All software artifacts must be refined
- Integration of different artifacts in collaborations

Idea

- Each feature is represented by a containment hierarchy:
 - Directory structure organizes the feature's artifacts.
 - At the file level, there may be heterogeneous artifacts.
- Composing features means composing containment hierarchies and, to this end, composing corresponding artifacts recursively by name and type
- For each artifact type, a different implementation of the composition operator “ \bullet ” has to be provided in AHEAD (just like the Jak-composition tool)

Beyond Jak: Uniformity



FeatureHouse: A Model and Framework for FOP

Goal

Language-independent model and tool chain to enhance given languages rapidly with support for feature-oriented programming

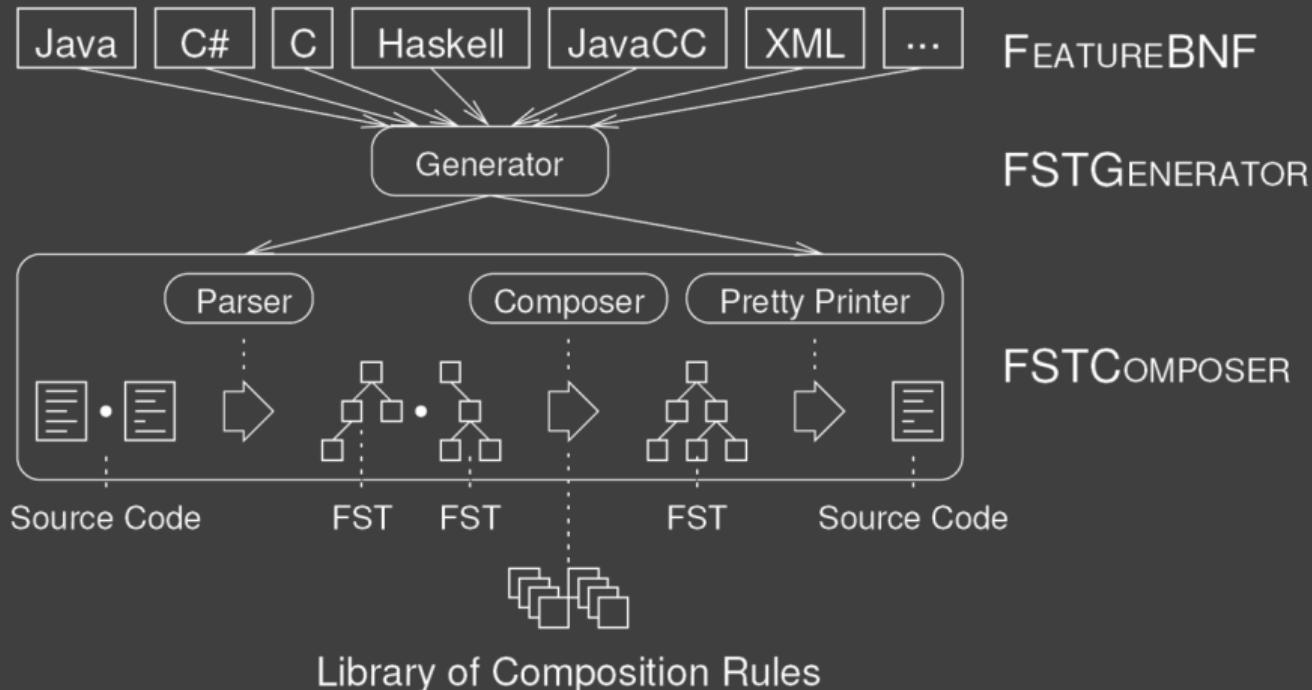
Assumption

- A feature may be represented as tree, known as **Feature Structure Tree (FST)**
- Example; Java: Packages, Classes, Methods and Fields

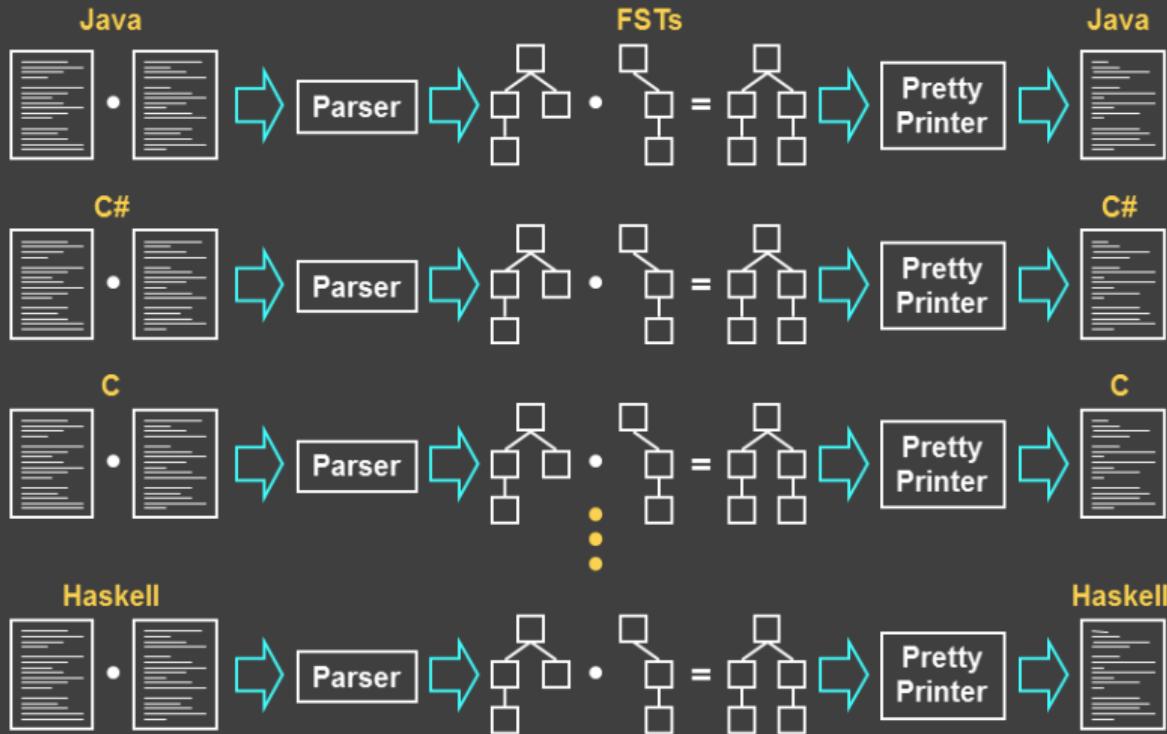
Idea

- Composition = Superimposition of FSTs (i.e., recursively superimpose nodes of FST, starting with the root node)
- Inner nodes: Can be safely superimposed if they are identical (superimposed parents and same name), or added if non-identical
- Leaf nodes: Type-specific resolution of conflicts

FeatureHouse: Overview



FeatureHouse: Composition



Example: Java Support in FeatureHouse

```
class Edge {  
    private Node a, b;  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
    }  
}
```

```
class Edge {  
    private Node start;  
    void print() {  
        original();  
        System.out.print(" directed from " + start);  
    }  
}
```

```
class Edge {  
    private Weight weight;  
    void print() {  
        original();  
        System.out.print(" weighted with " + weight);  
    }  
}
```

Differences Compared to Jak

- No explicit keyword `refines`
- Calling the method from previous refinement using keyword `original`

Discussion

Advantages

- Easy to use language-based mechanism, requires only minimal language extensions.
- Conceptually uniformly applicable to code and noncode artifacts.
- Separation of (possibly crosscutting) feature code into distinct feature modules.
- Little preplanning required due to mixin-based extension mechanism.
- Direct feature traceability from a feature to its implementation in a feature module.

Disadvantages

- Requires adoption of a language extension and composition tools.
- Tools need to be constructed for every language (although with the help of a framework).
- Only academic tools so far, little experience in practice.
- Granularity restricted to method-level (or other named structural entities).

Feature-Oriented Programming – Summary

Lessons Learned

- Idea: Mixin-based inheritance getting rid of the traditional limitations of inflexible inheritance hierarchies.
- Supports encapsulation of (cross-cutting) concerns and feature traceability by design.
- Academic approach not widely adopted in industry.

Further Reading

- Batory et al.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering, 30(6), 2004.
- Apel et al.: Language-Independent and Automated Software Composition: The FeatureHouse Experience. IEEE TSE, 39(1), 2013.
- Apel et al. 2013, Chapter 6.1
- Meinicke et al. 2017, Part 4

Practice

- How is class refinement in FOP different from inheritance in OOP?
- To some extent, FOP can be considered as static counterpart to the Decorator design pattern in OOP. Why?
- In which sense does FOP violate the classical principles of information hiding and encapsulation of OOP? What are the consequences?

7. Languages for Features

7a. Limitations of Object Orientation

7b. Feature-Oriented Programming

7c. Aspect-Oriented Programming

Recap and Motivation

Aspects and Aspect Weaving

Static vs. Dynamic Extensions

Quantification

Executing Additional Code

Aspects for Product Lines

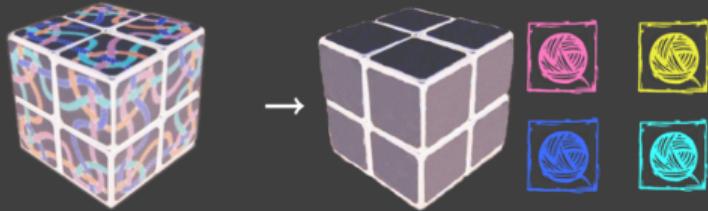
Discussion

Summary

FAQ

Motivation

Modularization of Cross-Cutting Concerns



Flexible Extension / Minimal Preplanning



Feature Traceability



find feature  in product



Achieving all this requires novel implementation techniques that overcome the limitations of classical object-oriented paradigms.

Aspects and Aspect Weaving

Aspect

[Apel et al. 2013, pp. 143–145]

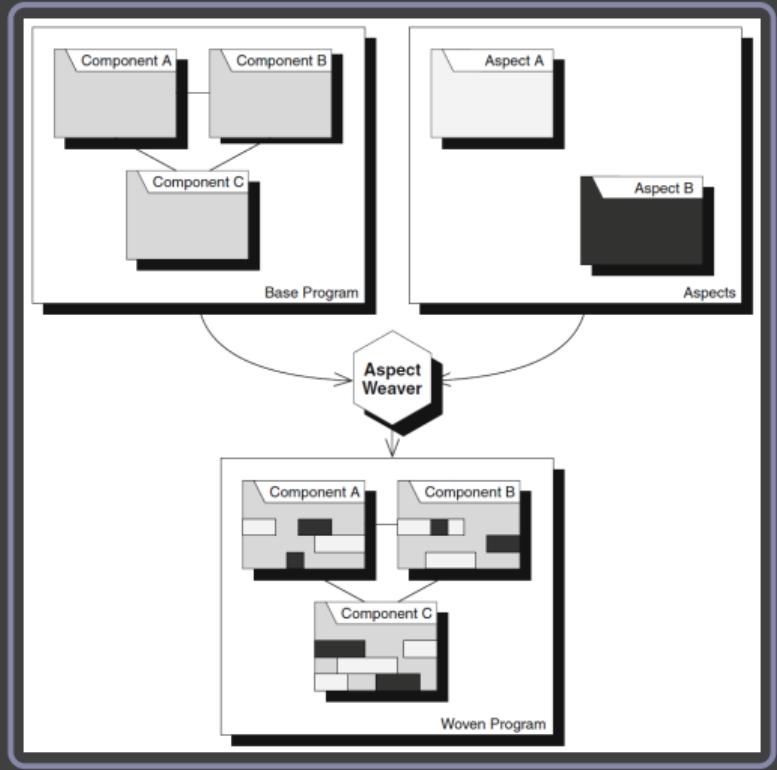
An aspect encapsulates the implementation of a crosscutting concern.

Aspect Weaving

[Apel et al. 2013, pp. 143–145]

An aspect weaver merges the separate aspects of a program and the base program at user-selected program locations.

- Localizing a crosscutting concern within one code unit eliminates code scattering and tangling.
- An aspect can affect multiple other concerns with one piece of code, thereby avoiding code replication.

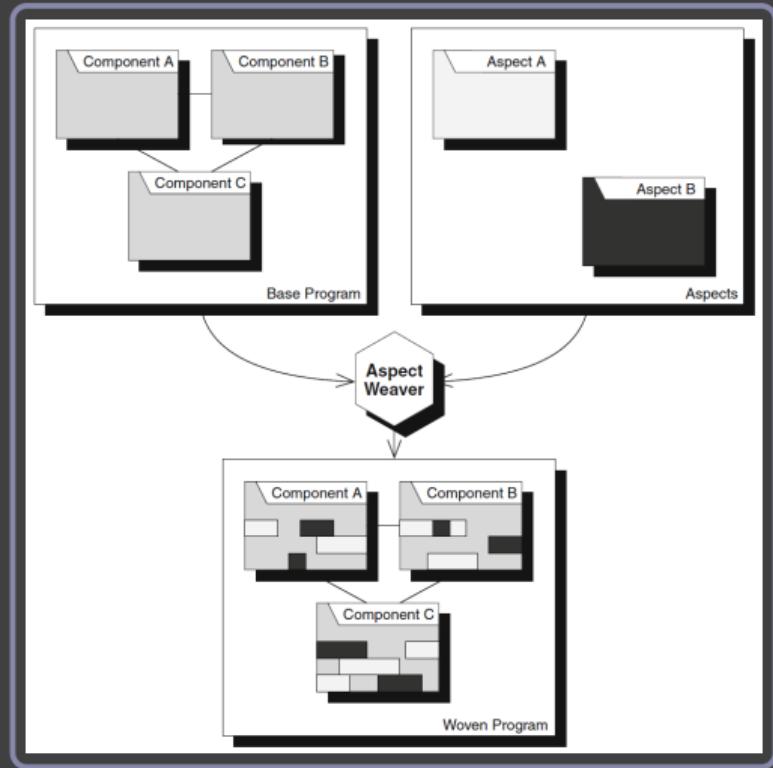


Aspects and Aspect Weaving in Java: AspectJ

AspectJ is an aspect-oriented language extension of Java.

- Base program is written in Java (i.e., components = classes)
- Aspects are written in Java but typically include a multitude of new language constructs introduced by AspectJ
- Aspect weaver (aka. AspectJ Compiler) follows a compile-time binding approach (though certain decisions are made at runtime, s. later).

AspectJ is the most popular and widely used aspect-oriented language, all examples in this lecture will be given in AspectJ.



Static Extensions through Inter-Type Declarations

Inter-Type Declaration

[Apel et al. 2013, pp. 143–145]

An inter-type declaration injects a method, field, or interface from inside an aspect into an existing class or interface.

Typical Usage

Add field X / method Y to class Z

```
aspect Weighted {  
    private int Edge.weight = 0;  
    public void Edge.setWeight(int w) {  
        weight = w;  
    }  
}
```

Dynamic Extensions through Join Points

Joint Point

[Apel et al. 2013, pp. 143–145]

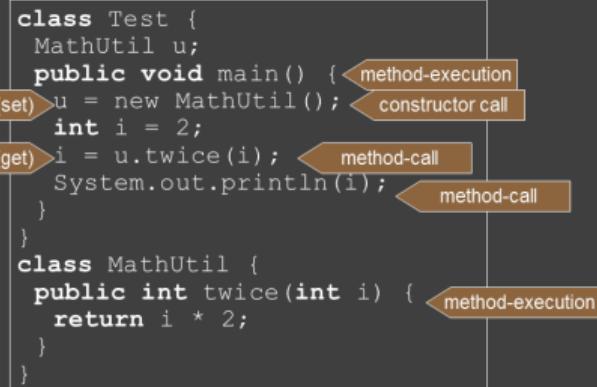
A join point is an event in the execution of a program at which aspects can be woven into the program.

Advice

Code which is being executed when a join point matches.

Join-Points in AspectJ:

- Calling/executing a method/constructor
- Access to a field (read or write)
- Catching an Exception
- Execution of a Advice
- ...



Open Question

How to specify the join points an aspect (i.e., an advice) affects?

Quantification through Pointcuts

Pointcut

[Apel et al. 2013, pp. 143–145]

A pointcut is a declarative specification of the join points that an aspect affects. It is a predicate that determines whether a given join point matches.

Quantification

[Apel et al. 2013, pp. 143–145]

Quantification is the process of selecting multiple join points based on a declarative specification (that is, based on a pointcut).

- Execute Advice X whenever the method `setWeight` of class `Edge` is called
- Execute Advice Y whenever any field in class `Edge` is accessed
- Execute Advice Z whenever a public method is called anywhere in the system and the method `initialize` has been called beforehand

Anonymous Pointcut

```
aspect A1 {  
    after() : execution(int MathUtil.twice(int)) {  
        System.out.println("MathUtil.twice executed");  
    }  
}
```

Explicit Pointcut

```
aspect A2 {  
    pointcut executeTwice() :  
        execution(int MathUtil.twice(int));  
    after() : executeTwice() {  
        System.out.println("MathUtil.twice executed");  
    }  
}
```

Quantification over other Join Points

Call of a method

```
aspect A1 {  
    after() : call(int MathUtil.twice(int)) {  
        System.out.println("MathUtil.twice called");  
    }  
}
```

Base Program

```
class Test {  
    public static void main(String[] args) {  
        MathUtil u = new MathUtil();  
        int i = 2;  
        i = u.twice(i);  
        i = u.twice(i);  
        System.out.println(i);  
    }  
}  
  
class MathUtil {  
    public int twice(int i) {  
        return i * 2;  
    }  
}
```

Quantification over other Join Points

Call of a method

```
aspect A1 {  
    after() : call(int MathUtil.twice(int)) {  
        System.out.println(" MathUtil.twice called");  
    }  
}
```

Constructor call

```
aspect A1 {  
    after() : call(MathUtil.new()) {  
        System.out.println(" MathUtil created");  
    }  
}
```

Base Program

```
class Test {  
    public static void main(String[] args) {  
        MathUtil u = new MathUtil();  
        int i = 2;  
        i = u.twice(i);  
        i = u.twice(i);  
        System.out.println(i);  
    }  
}  
  
class MathUtil {  
    public int twice(int i) {  
        return i * 2;  
    }  
}
```

Quantification over other Join Points

Call of a method

```
aspect A1 {  
    after() : call(int MathUtil.twice(int)) {  
        System.out.println("MathUtil.twice called");  
    }  
}
```

Constructor call

```
aspect A1 {  
    after() : call(MathUtil.new()) {  
        System.out.println("MathUtil created");  
    }  
}
```

And many more

- get/set: field access (read/write)
- etc.

Base Program

```
class Test {  
    public static void main(String[] args) {  
        MathUtil u = new MathUtil();  
        int i = 2;  
        i = u.twice(i);  
        i = u.twice(i);  
        System.out.println(i);  
    }  
}  
  
class MathUtil {  
    public int twice(int i) {  
        return i * 2;  
    }  
}
```

Further Quantification Options

Pointcuts with “Wildcards”

```
aspect A1 {  
    pointcut P1(): execution(int MathUtil.twice(int));  
    pointcut P2(): execution(* MathUtil.twice(int));  
    pointcut P3(): execution(int MathUtil.twice(*));  
    pointcut P4(): execution(int *.twice(int));  
    pointcut P5(): execution(int MathUtil.twice(..));  
    pointcut P6(): execution(int *Util.tw*(int));  
    pointcut P7(): execution(int *.twice(int));  
    pointcut P8(): execution(int MathUtil+.twice(int));  
}
```

Logical Connections of Pointcuts

```
aspect A1 {  
    pointcut P2(): call(* MathUtil.*(..)) && !call(* MathUtil.twice(*));  
    pointcut P3(): execution(* MathUtil.twice(..)) && args(int);  
}
```

Wildcard Symbols

- * Exactly one value
- .. Arbitrary many values
- + Class or any subclass

Logical Connectors

Pointcuts can be connected by usual logical operators
&&, ||, and !

Advice (Pieces of Advice)

- Additional code before, after or instead of the join point: **before**, **after**, **around**
- Around-Advice allows to continue the original join point using the keyword **proceed**
- Keyword **proceed** corresponds to **original/Super** in FOP and **super** in OOP

```
public class Test2 {  
    void foo() {  
        System.out.println("foo() executed");  
    }  
}  
aspect AdviceTest {  
    before(): execution(void Test2.foo()) {  
        System.out.println("before foo()");  
    }  
    after(): execution(void Test2.foo()) {  
        System.out.println("after foo()");  
    }  
    void around(): execution(void Test2.foo()) {  
        System.out.println("around begin");  
        proceed();  
        System.out.println("around end");  
    }  
    after() returning (): execution(void Test2.foo()) {  
        System.out.println("after returning from foo()");  
    }  
    after() throwing (RuntimeException e): execution(void Test2.foo()) {  
        System.out.println("after foo() throwing "+e);  
    }  
}
```

thisJoinPoint

In an advice, `thisJoinPoint` can be used to get more information about the current join point.

```
aspect A1 {  
    after() : call(int MathUtil.twice(int)) {  
        System.out.println(thisJoinPoint);  
        System.out.println(thisJoinPoint.getSignature());  
        System.out.println(thisJoinPoint.getKind());  
        System.out.println(thisJoinPoint.getSourceLocation());  
    }  
}
```

Output

```
call(int MathUtil.twice(int))  
int MathUtil.twice(int)  
method—call  
Test.java:5
```

Parameterized Pointcuts

```
aspect A1 {  
    pointcut execTwice(int value) :  
        execution(int MathUtil.twice(int)) && args(value);  
    after(int value) : execTwice(value) {  
        System.out.println("MathUtil.twice executed with parameter "  
            + value);  
    }  
}
```

Base Program

```
class Test {  
    public static void main(String[] args) {  
        MathUtil u = new MathUtil();  
        int i = 2;  
        i = u.twice(i);  
        i = u.twice(i);  
        System.out.println(i);  
    }  
}  
  
class MathUtil {  
    public int twice(int i) {  
        return i * 2;  
    }  
}
```

Pointcuts this and target

- **execution:** this and target capture the object on which the method is called
- **call, set, and get:** this captures the object that calls the method or accesses the field; target captures the object on which the method is called or the field is accessed.

```
aspect A1 {  
    pointcut P1(Main s, MathUtil t):  
        call(* MathUtil.twice(*))  
        && this(s)  
        && target(t);  
}
```

Order Matters: Aspect Precedence

Order in which aspect weaver process the aspects may be relevant when multiple aspects extend the same join point.

Example

- 1st aspect implements synchronization with around advice
- 2nd aspect implements logging with after-advice on the same join point
- Depending on the order of weaving, the logging code will be synchronized or not

Explicit definition using declare precedence

```
aspect DoubleWeight {  
    declare precedence : *, Weight, DoubleWeight;  
    [...]  
}
```

Aspect Weaving: Behind the Scenes

Weaving in AspectJ (Conceptually)

- Inter-type declarations are added to respective classes
- Each advice is converted into a method
- Pointcuts: Add method call from the join points to the advice
- Dynamic extensions: Insert source code at all potential join points that checks dynamic conditions and, if conditions hold, calls the advice method.

Weaving in AspectJ (Technically)

AspectJ Compiler; conceptual effect is only visible in Bytecode

Other Options

- Source transformation: Base Program + Aspects → Java (s. FOP/Jak).
- Evaluation at runtime: Meta-Object Protocol, interpreted languages, ...

Typical (Traditional) Aspects

- Logging, Tracing, Profiling
- Adding identical code to a large number of methods

Record time to execute my public methods

```
aspect Profiler {  
    Object around() : execution(public * com.company..*.*(..)) {  
        long start = System.currentTimeMillis();  
        try {  
            return proceed();  
        } finally {  
            long end = System.currentTimeMillis();  
            printDuration(start, end,  
                thisJoinPoint.getSignature());  
        }  
    }  
    // implement recordTime...  
}
```

Aspects for Product Lines

Basic Idea

- Implement one aspect per feature.
- Feature selection determines the aspects which are included in the weaving process.

- Aspects encapsulate changes to be made to existing classes.
- However, aspects do not encapsulate new classes introduced by a feature (only nested classes within an aspect)

A Color Feature for Graphs

```
aspect ColorFeature {  
    Color Node.color = new Color();  
  
    before(Node n): execution(void print()) && this(n) {  
        Color.setDisplayColor(n.color);  
    }  
  
    static class Color {  
        ...  
    }  
}
```

Controversial: Obliviousness and Fragile-Pointcut Problem

Principle of Obliviousness

Base program is (deliberately) supposed to be oblivious wrt. the aspects that “hook into” the system:

- Base program developers implement their concerns as if there were no aspects.
- Aspect programmers extend the base program.

Obliviousness worsens the fragile-pointcut problem

Because the base programmer does not know about aspects, it is more likely that changes may break aspect bindings and can remain unnoticed for a long time.

Fragile-Pointcut Problem

Base program may be modified such that the set of join points changes in an undesired way:

- Join points may be removed accidentally
- Join points may be captured by aspects accidentally

```
class Chess {  
    void drawKing() {...}  
    void drawQueen() {...}  
    void drawKnight() {...}  
    void draw() {...} // new method matches pointcut!  
}  
aspect UpdateDisplay {  
    pointcut drawn : execution(* draw*(..));  
}
```

Discussion

Advantages

- Separation of (possibly crosscutting) feature code into distinct aspects.
- Direct feature traceability from feature to its implementation in an aspect.
- Little or no preplanning effort required.
- Fine-grained variability driven by the join-point model of the aspect-oriented language.

Disadvantages

- Requires adoption of a rather complex extension mechanism (new language and paradigm).
- No unifying theory like no language-independent framework.
- Program evolution and maintenance affected by fragile-pointcut problem.

Aspect-Oriented Programming – Summary

Lessons Learned

- Idea: “In programs P, whenever condition C arises, perform action A”
- AspectJ: Sophisticated joint-point model and powerful language to quantify over join points (through pointcuts).
- Supports encapsulation of (cross-cutting) concerns and feature traceability by design.
- Practical acceptance limited due to fragile-pointcut problem.

Further Reading

- Kiczales et al.: Aspect-oriented programming. Proc. Europ. Conf. Object-Oriented Programming. 1997
- Filman et al.: Aspect-oriented software development. Addison-Wesley. 2005
- Apel et al. 2013, Chapter 6.2

Practice

- Which features particularly benefit from the concept of quantification?
- What similarities and differences do you see between FOP and AOP?

FAQ – 7. Languages for Features

Lecture 7a

- What are problems of previous implementation techniques?
- What is the preplanning problem?
- What are (crosscutting) concerns?
- What is the tyranny of the dominant decomposition?
- What are crosscutting concerns when implementing arithmetic expressions?
- Why cannot all concerns be modularized with object orientation and design patterns?

Lecture 7b

- What is feature-oriented programming and collaboration-based design? What are collaborations, roles, feature modules, class refinements?
- How to compose feature modules?
- What is the difference between Mixin and Jampack? What is better?
- Why does the order matter when composing feature modules? Where does it come from?
- What is the principle of uniformity?
- How to implement product lines with feature modules?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of feature modules?

Lecture 7c

- What is aspect-oriented programming? What are aspects, aspect weaving, join points, pointcuts, pieces of advice, AspectJ?
- What are static/dynamic extensions, quantification, before/after/around advice, inter-type declarations?
- What is aspect precedence (good for)? How is it different from feature modules?
- What are obliviousness and fragile-pointcut problem?
- What are commonalities and differences between feature modules and aspects?
- How to implement product lines with aspects?
- What are (dis)advantages of aspects?