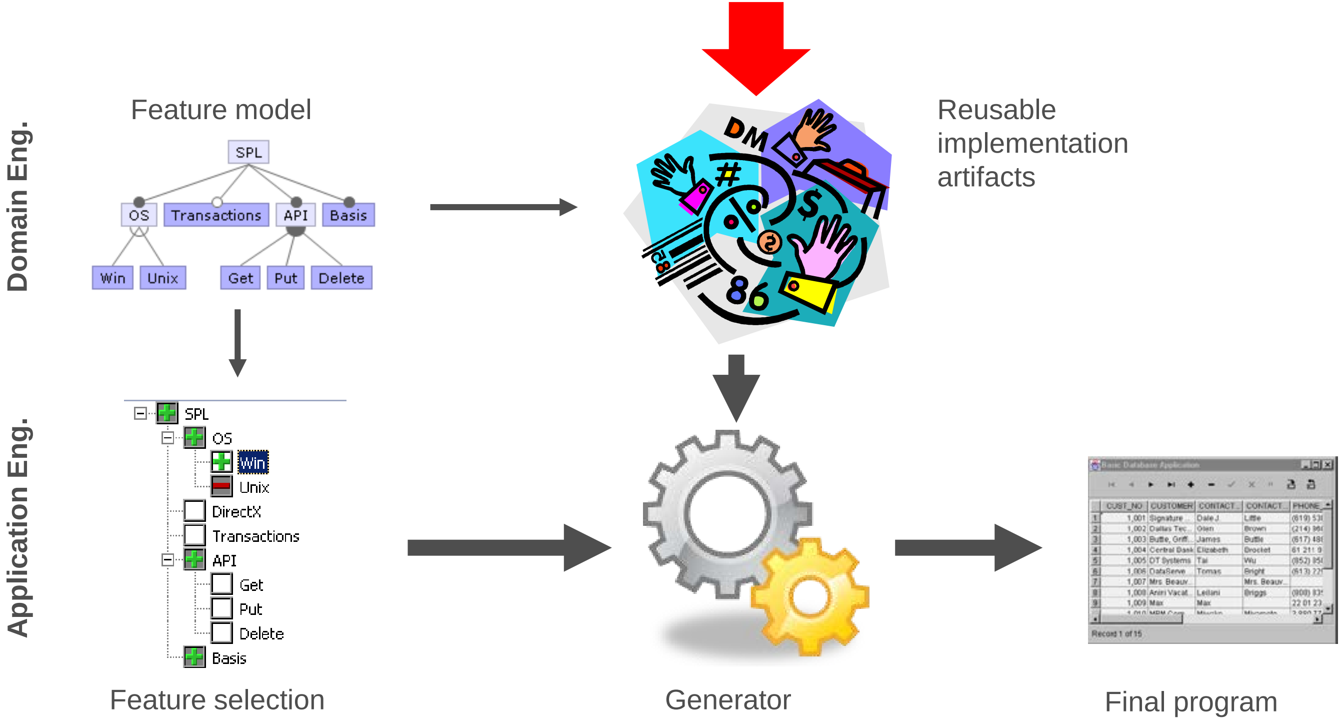
# Lecture 8: Feature Orientation and Aspect Orientation

Daniel Strüber
TDA 594/DIT 593 - November 22, 2022

# How to implement variability?



# Today's Goals

- Solve problems:
  - Feature Traceability
  - Crosscutting concerns
  - Preplanning
  - Inflexible extension mechanisms (inheritance)
- Modular feature implementation
- New types of implementation techniques

# Agenda

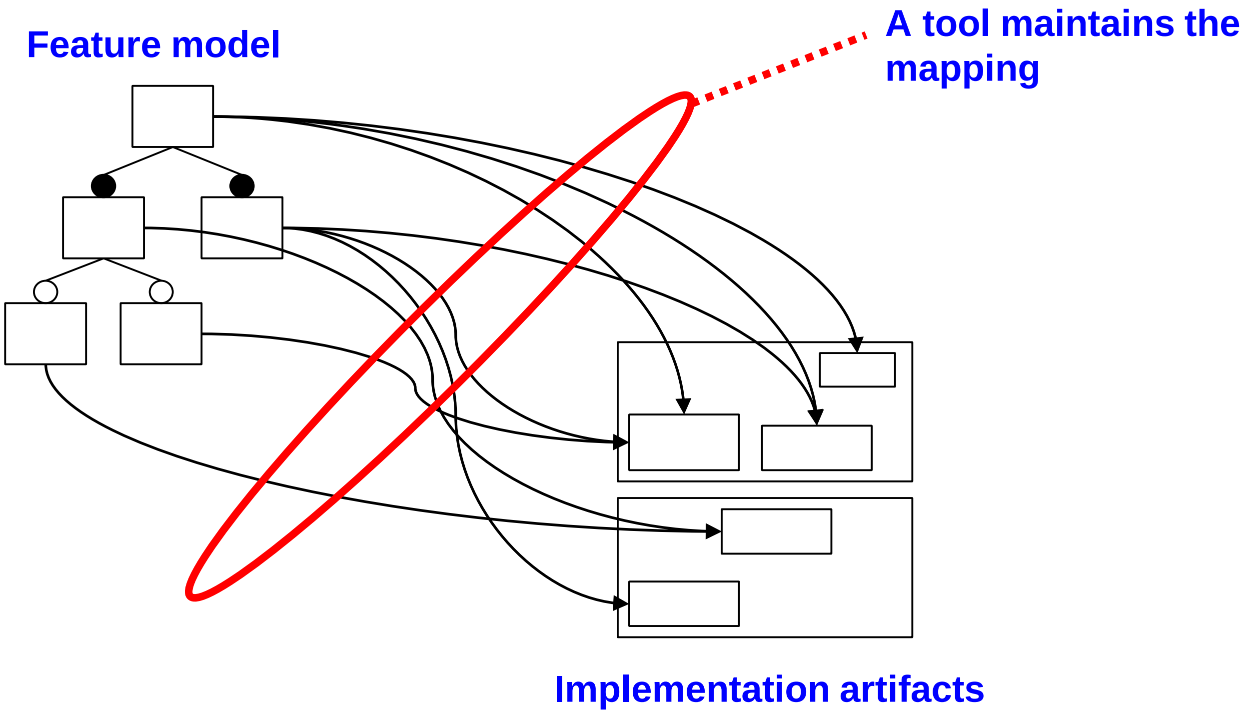
- Feature-oriented programming
  - Key idea
  - Implementation with FeatureHouse
- Aspect-oriented programming
  - Key idea
  - Implementation with AspectJ
- Features vs. Aspects

# Feature-Oriented Programming

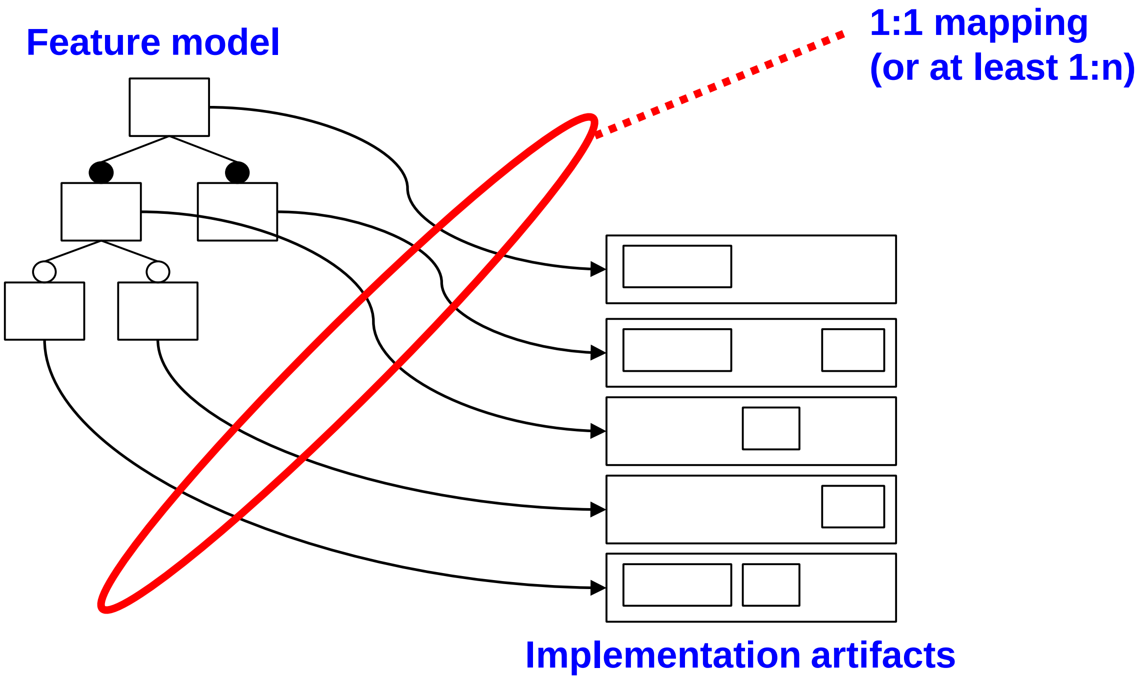
# Goal: feature cohesion

- we want to have all implementation artifacts for a feature a single location in the code
  - features explicit in code
- A question of programming language and programming environment
  - physical vs. virtual cohesion
- Automatically gives us traceability as well

# Feature traceability with tool support



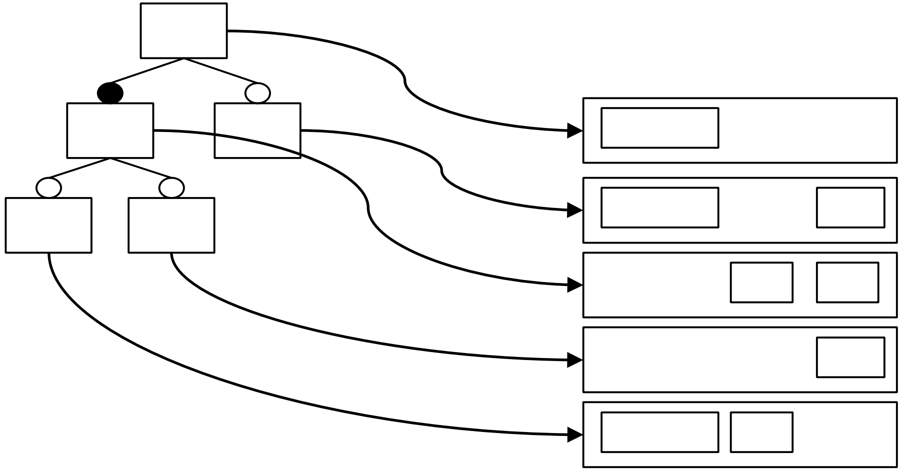
# Feature traceability with language support



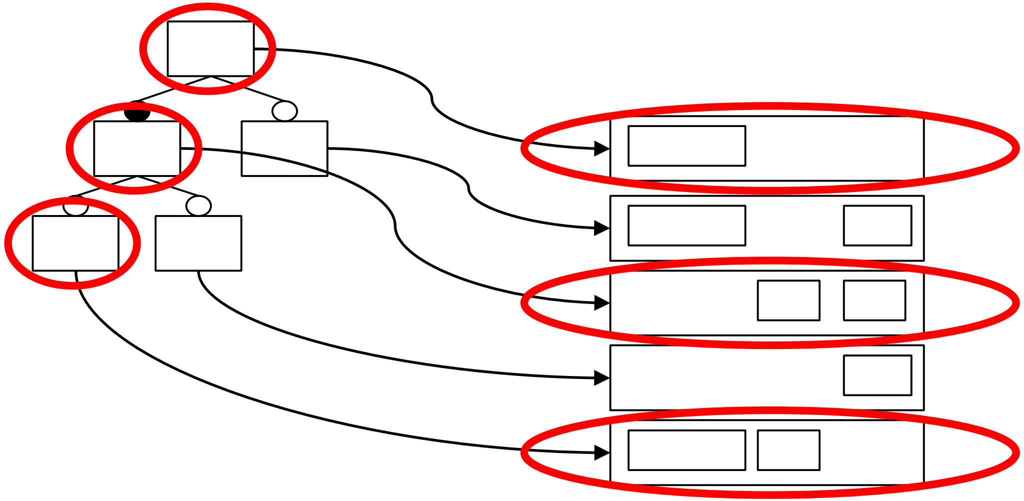
# Feature-oriented programming

- Language-based approach for feature traceability
- Implement each feature in a feature module
  - Perfect feature traceability
  - Separation and modularization of features
- Feature-based program generation
  - Programs generated via **feature composition**
- As a research idea, introduced 20 years ago
  - Prehofer, ECOOP‘97 and Batory, ICSE'03

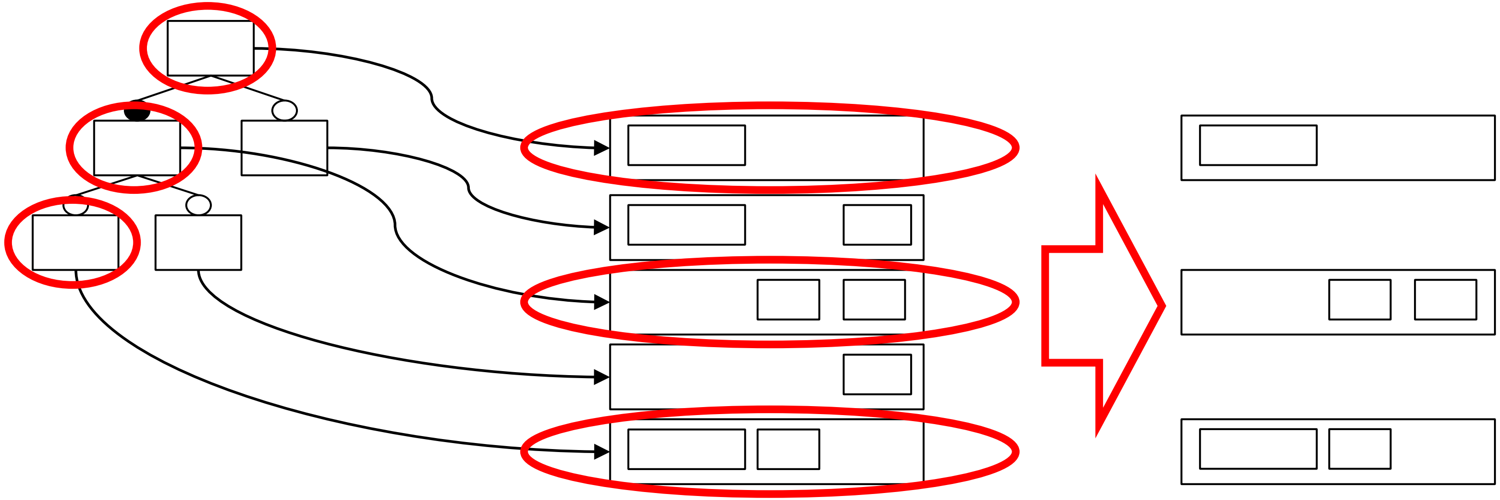
# Feature composition



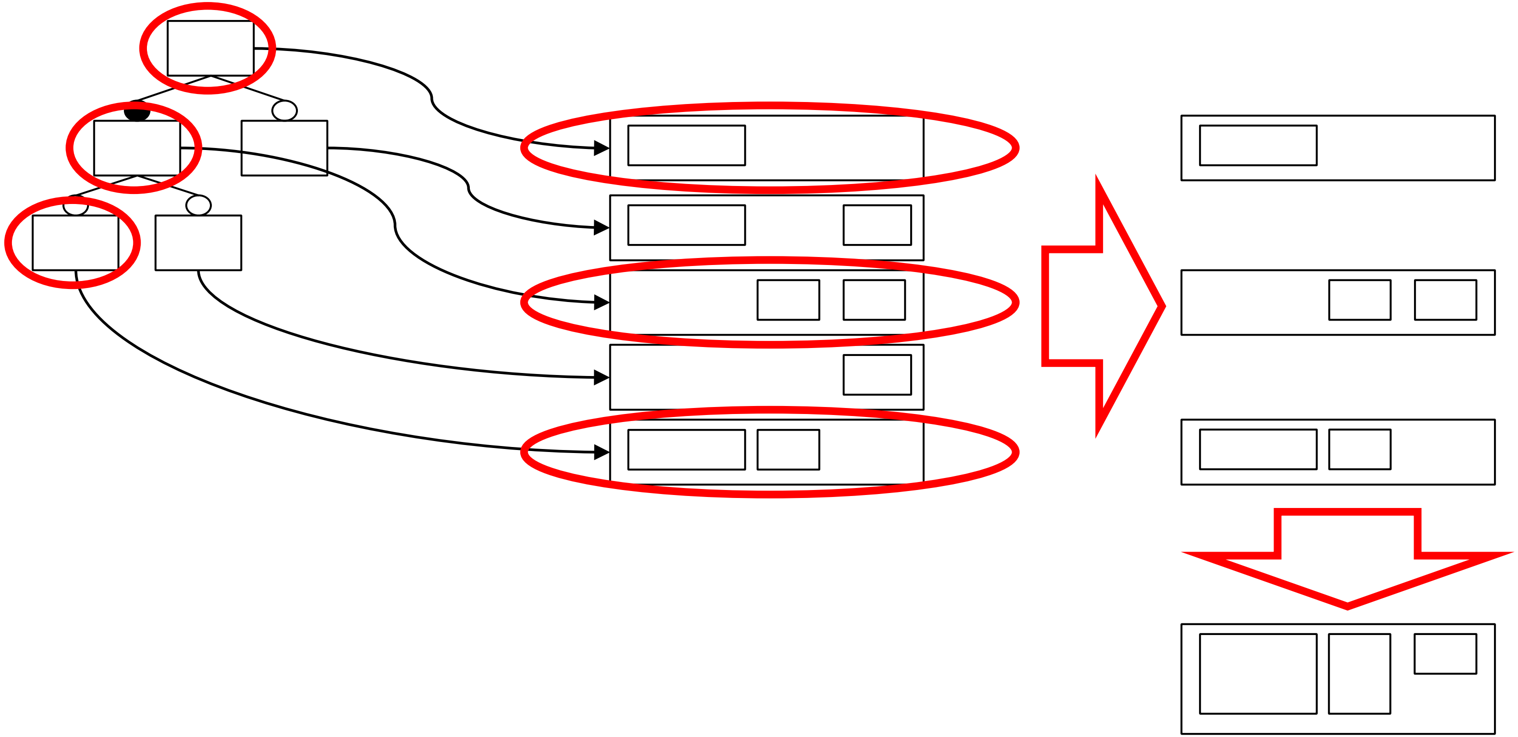
# Feature composition



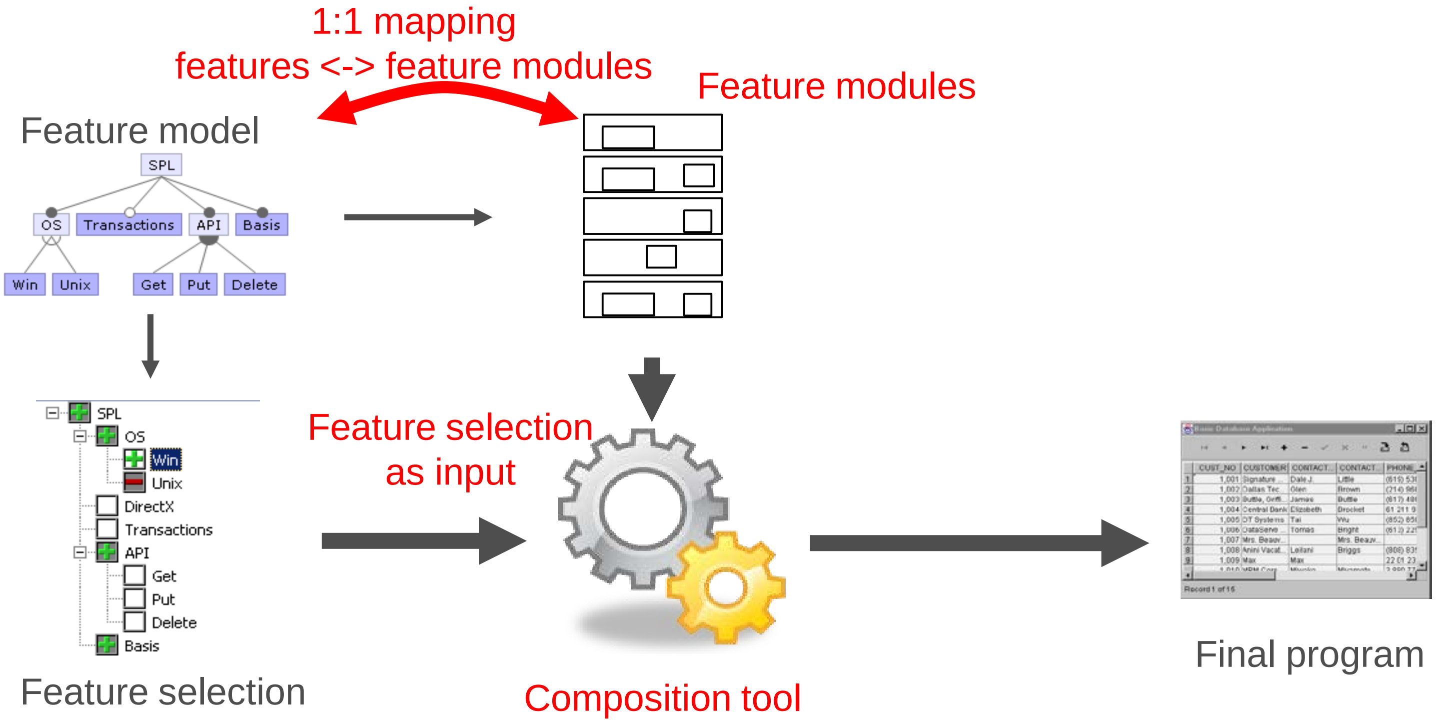
# Feature composition



# Feature composition



# Product lines with feature modules

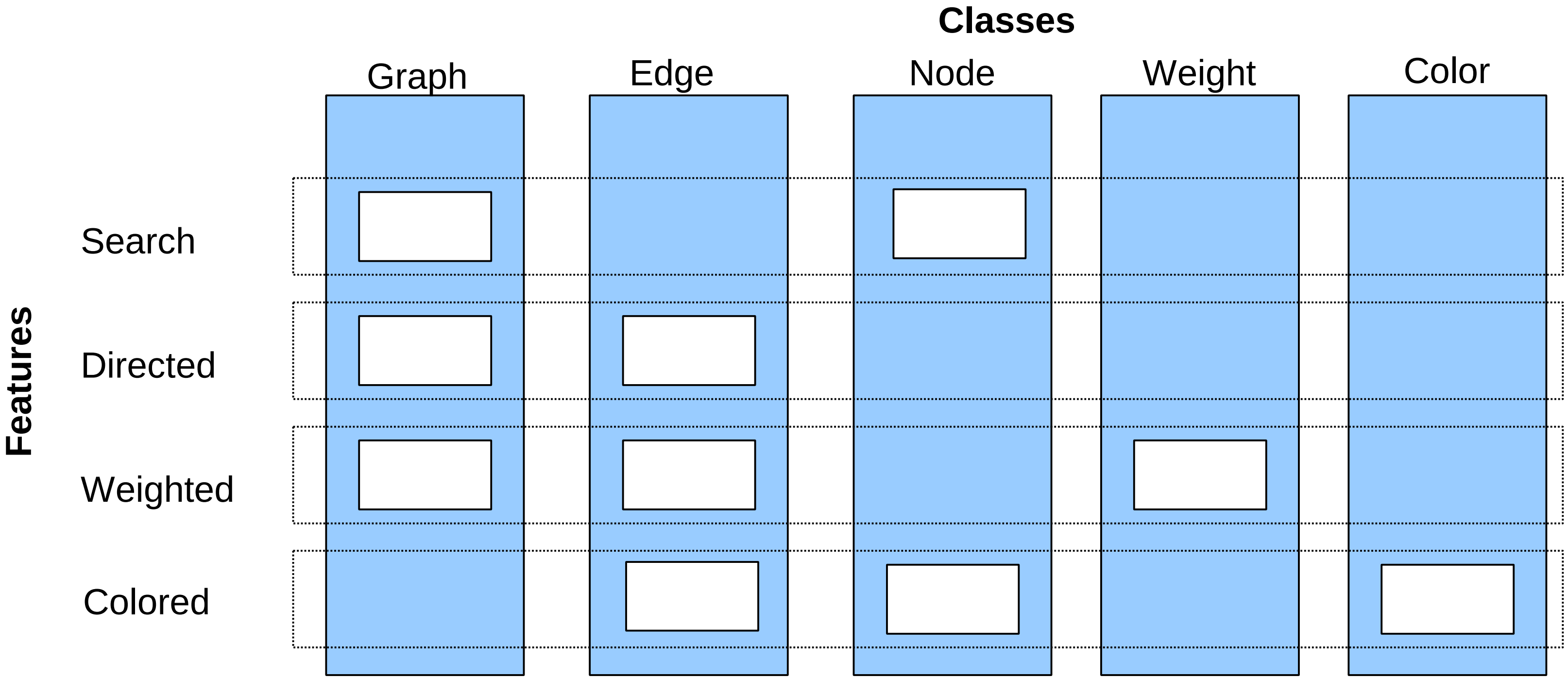


# Implementing feature modules

- Starting point: code base structured into classes
- Features often implemented by several classes
- Classes often implement more than one feature

- Idea: keep class structure, but split classes along features
  - Implemented in tools **FeatureHouse** and **AHEAD**

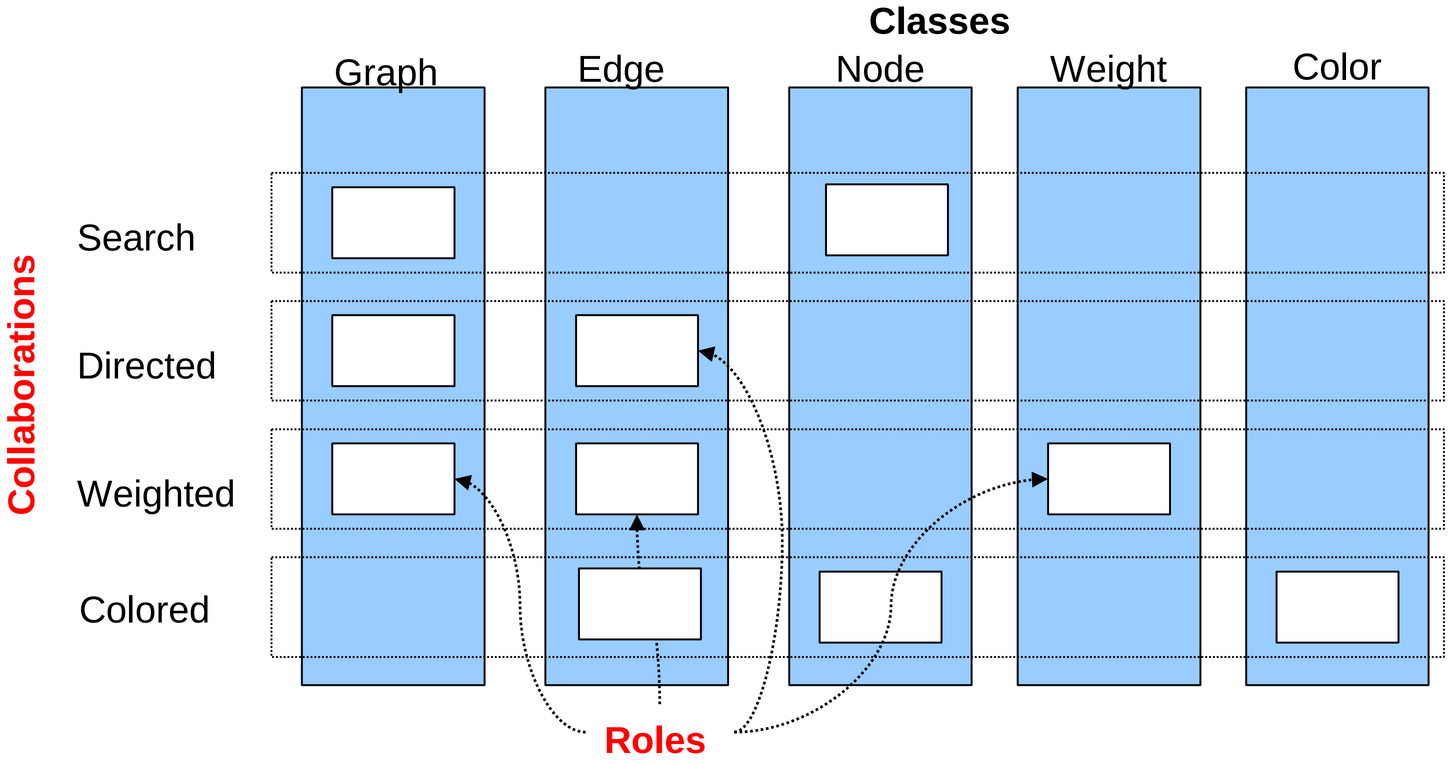
# Splitting of classes



# Collaborations and roles

- Collaboration: a set of classes that interact to implement a feature
- Different classes play different roles in collaborations
- One class plays different roles in different collaborations
- A role encapsulates the functionality (methods, fields) of a class that is relevant for the collaboration

# Collaborations and roles



# Collaborations in graph example

```
class Graph {
  List nodes = new List();
  List edges = new List();
  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() {
    for(int i = 0; i < edges.size(); i++)
      ((Edge)edges.get(i)).print();
  }
}
```

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

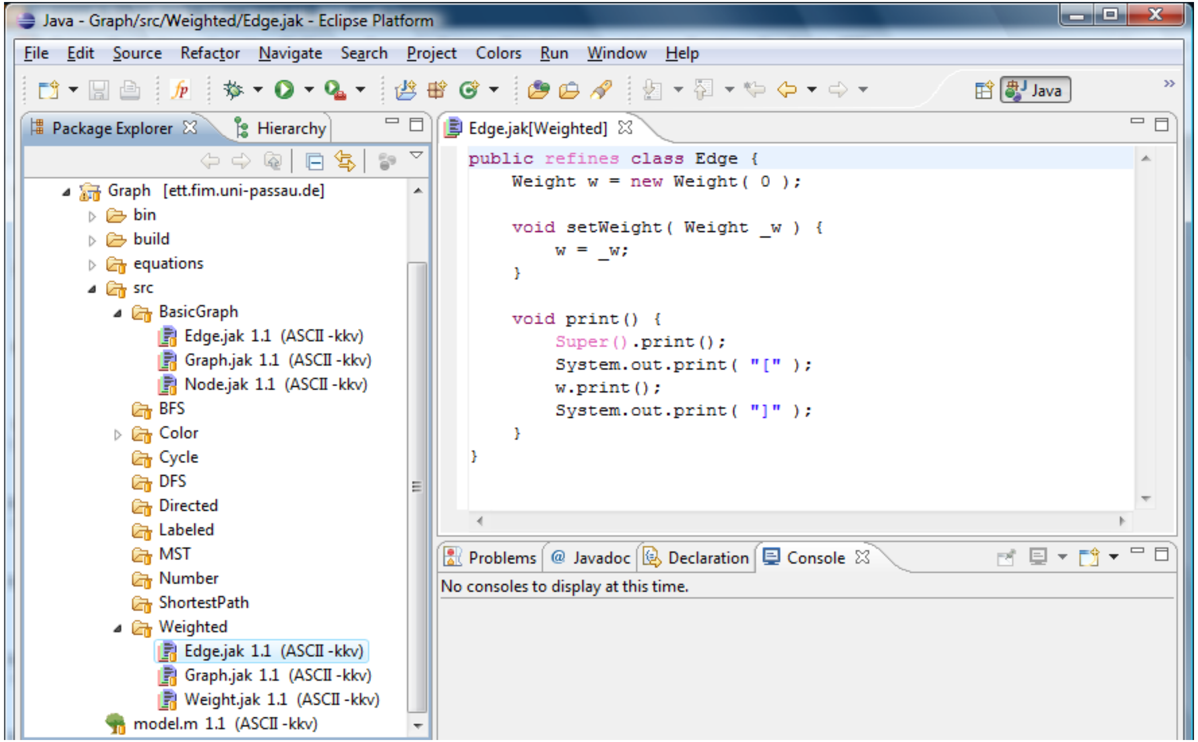
```
class Node {
  int id = 0;
  void print() {
    System.out.print(id);
  }
}
```

```
refines class Graph {
  Edge add(Node n, Node m) {
    Edge e = Super.add(n, m);
    e.weight = new Weight();
  }
  Edge add(Node n, Node m, Weight w)
    Edge e = new Edge(n, m);
    nodes.add(n); nodes.add(m);
    edges.add(e);
    e.weight = w; return e;
  } }
```

```
refines class Edge {
  Weight weight = new Weight();
  void print() {
    Super.print(); weight.print();
  }
}
```

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

# Directory hierarchy: features + roles



# Example: class refinements

Successive extension of base implementation by means of refinements

Edge.java

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

Edge.java

```
class Edge {
    private int weight;
    ...
}
```

Edge.java

```
class Edge {
    private Color color;
    ...
}
```

# Method refinement (FeatureHouse)

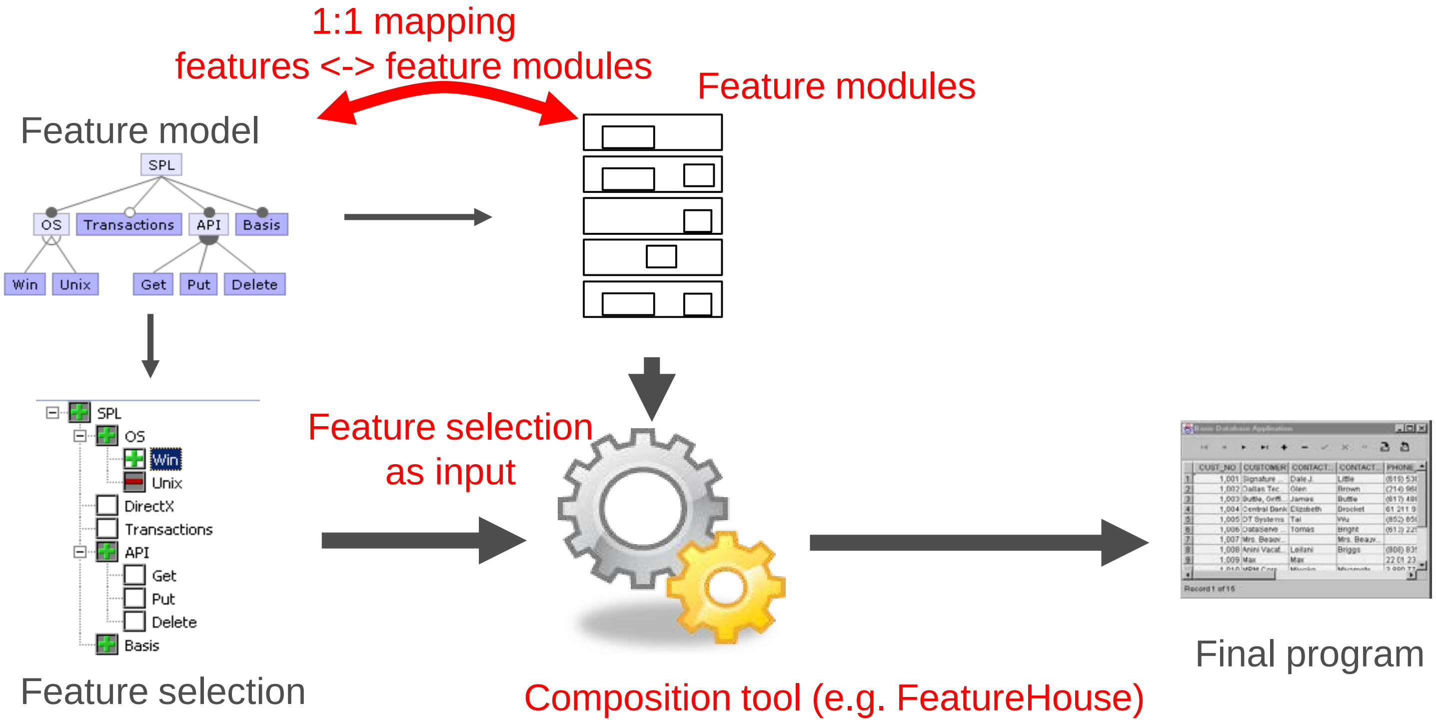
- Each extension can refine and introduce methods
- Methods can be overriden
- Can call methods from next refinement level with **`original()`**
- Similar to inheritance

```
class Edge {
    void print() {
        System.out.print(
            " Edge between " + node1 +
          " and " + node2);
    }
}
```

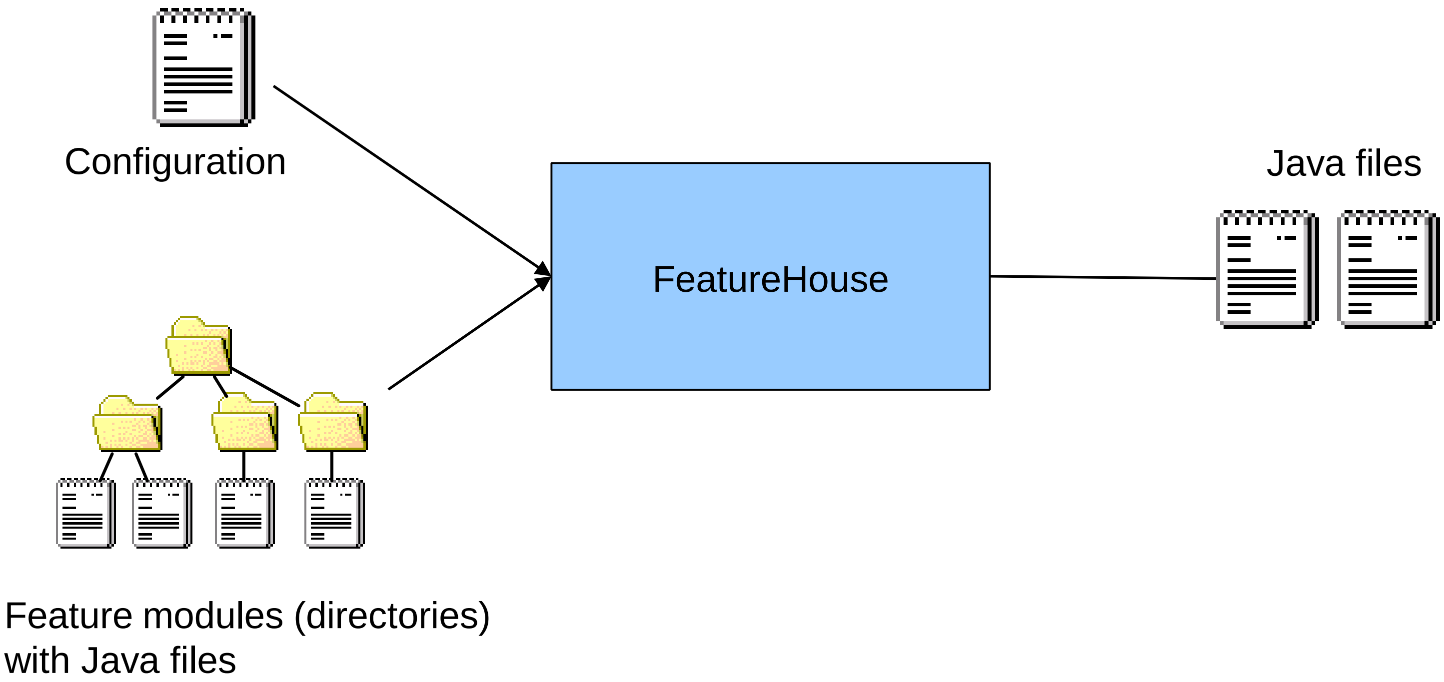
```
class Edge {
    private Node start;
    void print() {
        original();
        System.out.print(
            " directed from " + start);
    }
}
```

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

# Product lines with feature modules

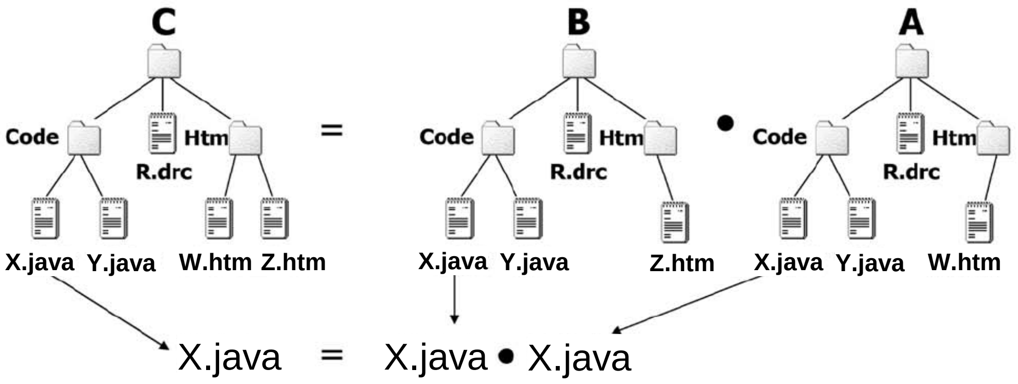


# Composition in FeatureHouse

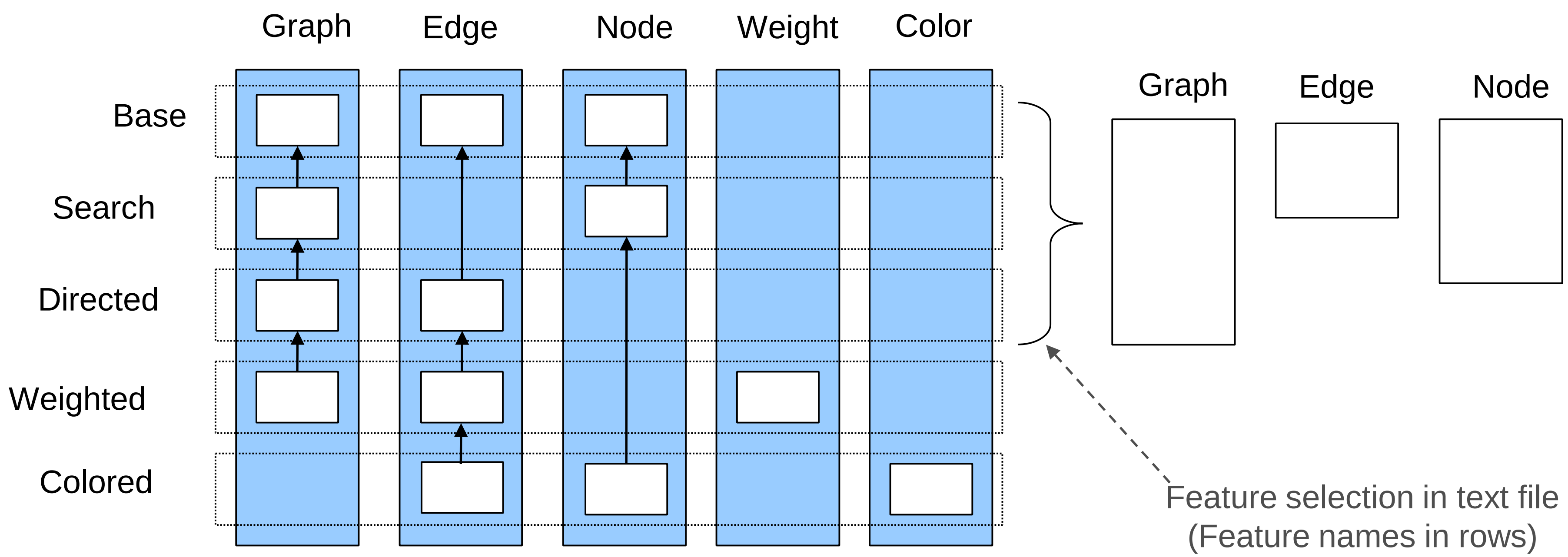


# Composition of directories

- All roles of a collaboration are stored in a package/module, typically in a directory
- Composition of collaborations: composing classes with all contained refinements of same name



# Example composition

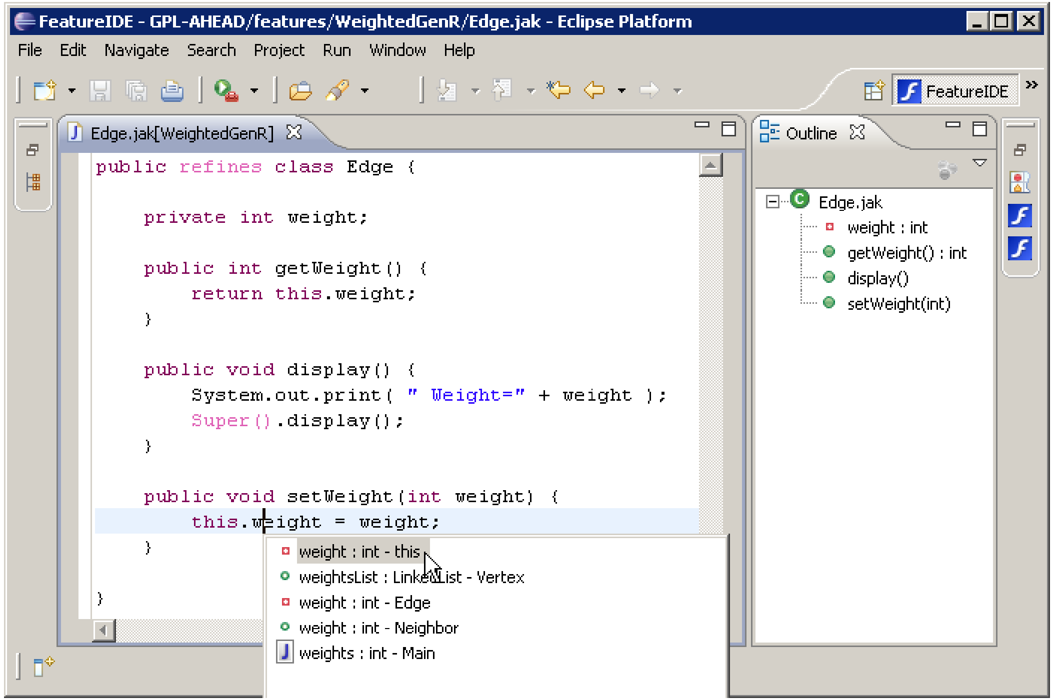


# Tools

- AHEAD Tool Suite + Documentation
  - Command line tools for Jak (Java 1.4 extension) http://www.cs.utexas.edu/users/schwartz/ATS.html
- FeatureHouse
  - Command line tool for Java, C#, C, Haskell, … http://www.fosd.de/fh
- FeatureC++
  - Alternative to AHEAD for C++ http://www.fosd.de/fcpp
- FeatureIDE
  - Eclipse plugin for AHEAD, FeatureHouse und FeatureC++
  - Automated build, syntax highlighting, etc… http://www.fosd.de/featureide

# FeatureIDE – Demo

- Video tutorial




https://www.youtube.com/watch?v=yRF0Kfs1NRA

# Summary FeatureHouse

- One base class + arbitrary refinements (roles)
- Class refinements can…
  - Introduce fields
  - Introduce methods
  - Change (extend) method implementations
- Feature module (collaboration): directory with base classes and/or refinements
- Composition of base class+refinements per feature

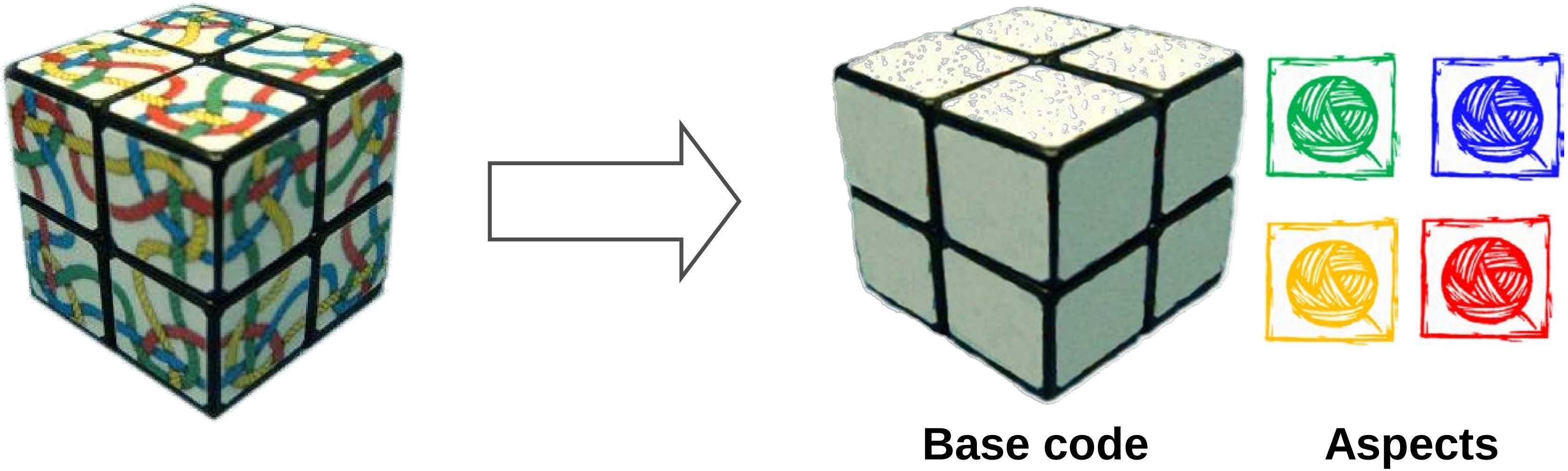
# Quiz

- How many roles can a program with three classes and four features have

  (a) maximally and (b) minimally?

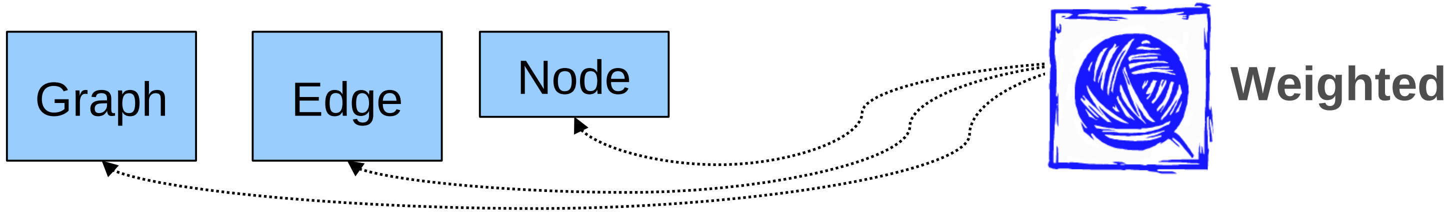
# Aspect-Oriented Programming

# Modularizing cross-cutting concerns



# Idea

- Modularize a cross-cutting concern into an aspect
- Aspect describes effects on rest of software
- How interpreted? Multiple options:
  - as a program transformation
  - as a metaobject protocol
  - some sort of feature module



# AspectJ

- AspectJ is an AOP extension for Java
- Program = base code + extensions (*aspects*)
    - Base code implemented in Java
    - Aspects similiar to Java, but there are a few special constructs
- Provides special components (*weavers*) for „weaving“ aspects into base code

# Whan can an aspect do?

- In AspectJ, an aspect can:
  - add methods and fields to a class
  - extend methods with additional code
  - catch events (e.g., method calls and fiel accesses) and respond by executing additional or alternative code
  - (add classes: only in a restricted form)

# Static extensions

- Static extensions with „inter-type declarations“
  - for example, add method X to class Y

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

# Dynamic extensions

- Based on AspectJ‘s *join point* model
  - **Join point**: an event during the program execution. For example, a method call or field access.
  - **Pointcut:** a predicate to select join points
  - **Advice**: code that is to be executed if a joint point was selected by a pointcut

```
aspect Weighted {
    pointcut printExecution(Edge edge) :
        execution(void Edge.print()) && this(edge);
    after(Edge edge) : printExecution(edge) {
        System.out.print(' weight ' + edge.weight);
    }
}
```

# Quantification

- Pointcuts describe join points *declaratively* and can select multiple join points at the same time
- Examples:
  - Execute advice X whenever the method „setWeight“ in class „Edge“ is called
  - Execute advice Y whenever **any** field in class „Edge“ is accessed
  - Execute advice Z whenever **any** public method in the system is called, **and** the method „initialize“ has been called before that

# AspectJ – join point model

- Join points can describe:
  - a method call
  - a method execution
  - a constructor call
  - a constructor execution
  - a field access (read or write)
  - catching an exception
  - initialization of a class or an object
  - execution of an advice

# Join point example

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

- method execution (`public void main() {`)
- field access (set) / constructor call (`u = new MathUtil();`)
- field access (get) (`int i = 2;`)
- method call (`i = u.twice(i);`)
- method call (`System.out.println(i);`)
- method execution (`public int twice(int i) {`)

# Pointcut *execution*

- Captures the execution of a method

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

execution →

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

Syntax:
**execution**(ReturnType ClassName.Methodname(ParameterTypes))

# Explicit vs. anonymous pointcuts

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

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

# Advice

- Additional code
  - **before**,
  - **after,** or
  - instead of (**around**) the join point.
- *around* advice:
  - can continue the original code with the keyword „proceed“

# Advice

```
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);
   }
}
```

# Patterns

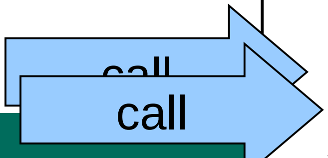
- allow “incomplete" specification of target join point for quantification
- *placeholders*
  one value: *
  multiple values: ..
- subclasses: +

```
aspect Execution {
   pointcut P1() : execution(int MathUtil.twice(int));

   pointcut P2() : execution(* MathUtil.twice(int));

   pointcut P3() : execution(int MathUtil.twice(*));

   pointcut P4() : execution(int MathUtil.twice(..));

   pointcut P5() : execution(int MathUtil.*(int, ..));

   pointcut P6() : execution(int *Util.tw*(int));

   pointcut P7() : execution(int *.twice(int));

   pointcut P8() : execution(int MathUtil+.twice(int));

   pointcut P9() : execution(public int
      package.MathUtil.twice(int)
      throws ValueNotSupportedException);

   pointcut Ptypical() : execution(* MathUtil.twice(..));
}
```

# Pointcut *call*

- Captures the call of a method
- Similar to execution, but on the side of the caller

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




```
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);
```

# Constructors

- „new“ keyword

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

call

```
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 {
```

# Pointcuts *set & get*

- Captures field accesses (of instance variables)

```
aspect A1 {
  after() : get(int MathUtil.counter) {
    System.out.println("MathUtil.value read");
  }
}
```

```
aspect A1 {
  after() : set(int MathUtil.counter) {
    System.out.println("MathUtil.value set");
  }
}
```

```
set(int MathUtil.counter)
set(int MathUtil.*)
set(* *.counter)
```

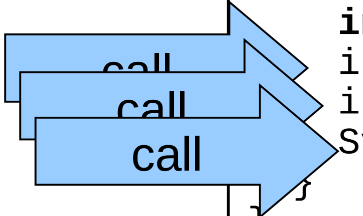
```
void
 new MathUtil();

i);
i);
  System.out.println(i);
}
}
```

# Pointcut *args*

- Matches just the parameters of a method
- Similar to execution(* *.*(X, Y)) or call(* *.*(X, Y))

```
aspect A1 {
  after() : args(int) {
    System.out.println("A method with only one parameter " +
                "of type int called or executed");
  }
}
```

```
args(int)
args(*)
args(Object, *, String)
args(.., Buffer)
```




```
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 {
```

# Combined pointcuts

- Pointcuts can be combined
  - &&, || and !

```
aspect A1 {
  pointcut P1(): execution(* Test.main(..)) || call(* MathUtil.twice(*));
  pointcut P2(): call(* MathUtil.*(..)) && !call(* MathUtil.twice(*));
  pointcut P3(): execution(* MathUtil.twice(..)) && args(int);
}
```

# Parametrized pointcuts

- Pointcuts can have parameters, can be used in advice
- Provides advice with information about context
- For that, use pointcut *args* with a variable (instead of type)

```
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);
  }
}
```

# Advice that uses parameters

- Example for advice that uses parameters:

```
aspect DoubleWeight {
    pointcut setWeight(int weight) :
        execution(void Edge.setWeight(int)) && args(weight);

    void around(int weight) : setWeight(weight) {
        System.out.print('doubling weight from ' + weight);
        try {
          proceed(2 * weight);
        } finally {
          System.out.print('doubled weight from ' + weight);
        }
    }
}
```

# Pointcuts *this* and *target*

- **this** and **target** capture the involved classes
- can be used with types (incl. patterns) & parameters

```
aspect A1 {
  pointcut P1(): execution(int *.twice(int)) && this(MathUtil);
  pointcut P2(MathUtil m) : execution(int MathUtil.twice(int)) && this(m);
  pointcut P3(Main source, MathUtil target): call(* MathUtil.twice(*)) &&
                                  this(source) && target(target);
}
```

For **call**, **set** und **get**: **this** captures object that calls the method / accesses field; **target** captures the object whose method is called/field is accessed

# Pointcuts *this* and *target*

- **this** and **target** capture the involved classes
- can be used with types (incl. patterns) & parameters

```
aspect A1 {
  pointcut P1(): execution(int *.twice(int)) && this(MathUtil);
  pointcut P2(MathUtil m) : execution(int MathUtil.twice(int)) && this(m);
  pointcut P3(Main source, MathUtil target): call(* MathUtil.twice(*)) &&
                                    this(source) && target(target);
}
```

- For **execution**: **this** and **target** capture the object on which the method is called

# Pointcuts *within* and *withincode*

- Restrict join points based on location
- Example: only calls of the method *twice* that come from *Test* or *Test.main*, respectively

```
aspect A1 {
  pointcut P1(): call(int MathUtil.twice(int)) && within(Test);
  pointcut P2(): call(int MathUtil.twice(int)) && withincode(* Test.main(..));
}
```

# Pointcuts *cflow* and *cflowbelow*

- Captures all join points that appear in the control flow of another join point
  - **cflow**: all join points *including* said join point,
  - **cflowbelow**: all join points *excluding* said join point.

```
aspect A1 {
  pointcut P1(): cflow(execution(int MathUtil.twice(int)));
  pointcut P2(): cflowbelow(execution(int MathUtil.twice(int)));
}
```

# Control flow

```
class Test {
  public static void main() {
    MathUtil u = new MathUtil();
    int i = 2;
    i = u.twice(i);
    i = u.twice(i);
    i = u.power(i, 3);
    System.out.println(i);
  }
}

class MathUtil {
  public int twice(int i) {
    return i * 2;
  }
  public int power(int i, int j){
```

Stack:

- Test.main
- MathUtil.twice
- MathUtil.power
- MathUtil.power
- MathUtil.power
- MathUtil.power

# Examples for cflow

```
before() :
execution(* *.*(..))
```

```
execution(void Test.main(String[]))
execution(int MathUtil.twice(int))
execution(int MathUtil.twice(int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
```

```
execution(* *.*(..)) &&
 cflow(execution(* *.power(..)))
```

```
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
```

```
execution(* *.*(..)) &&
 cflowbelow(execution(* *.power(..)))
```

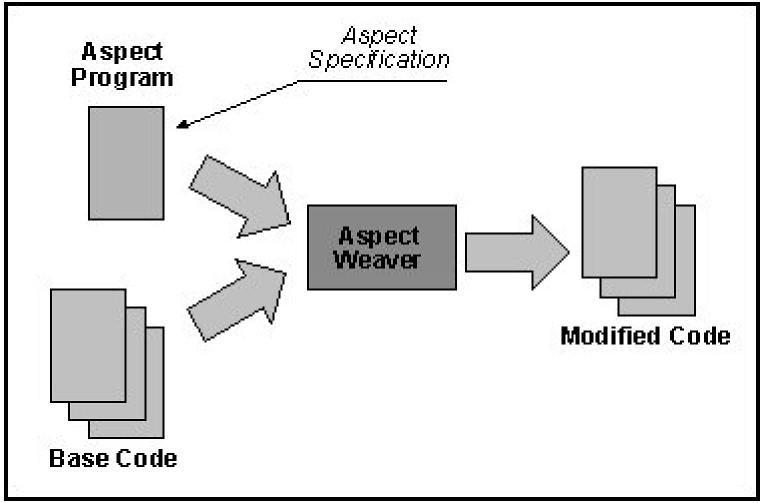
```
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
```

```
execution(* *.power(..)) &&
 !cflowbelow(execution(* *.power(..)))
```

```
execution(int MathUtil.power(int, int))
```

# Aspect weaving

- **Weaving**: the process of applying aspects to objects
- Weaving can take place at several points in time:
  - **Compile time**: weaving is the responsibility of the compiler
  - **Load-time**: weaving is the responsibility of the classloader
  - **Runtime**: application is executed in a special AOP container that is responsible for the weaving



# Aspects in graph example

```
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); return e;
  }
  void print() {
    for(int i = 0; i < ev.size(); i++)
      ((Edge)ev.get(i)).print();
  }
}
```

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

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

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

**Basic Graph**

# Typical aspects

- Logging, Tracing, Profiling
  - Adding the same code to many methods

```
aspect Profiler {
    /** record time to execute my public methods */
    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...
}
```

# Typical aspects II

- Caching, Pooling
  - Cache or resource pool implemented at central location, capture program locations that would create a new

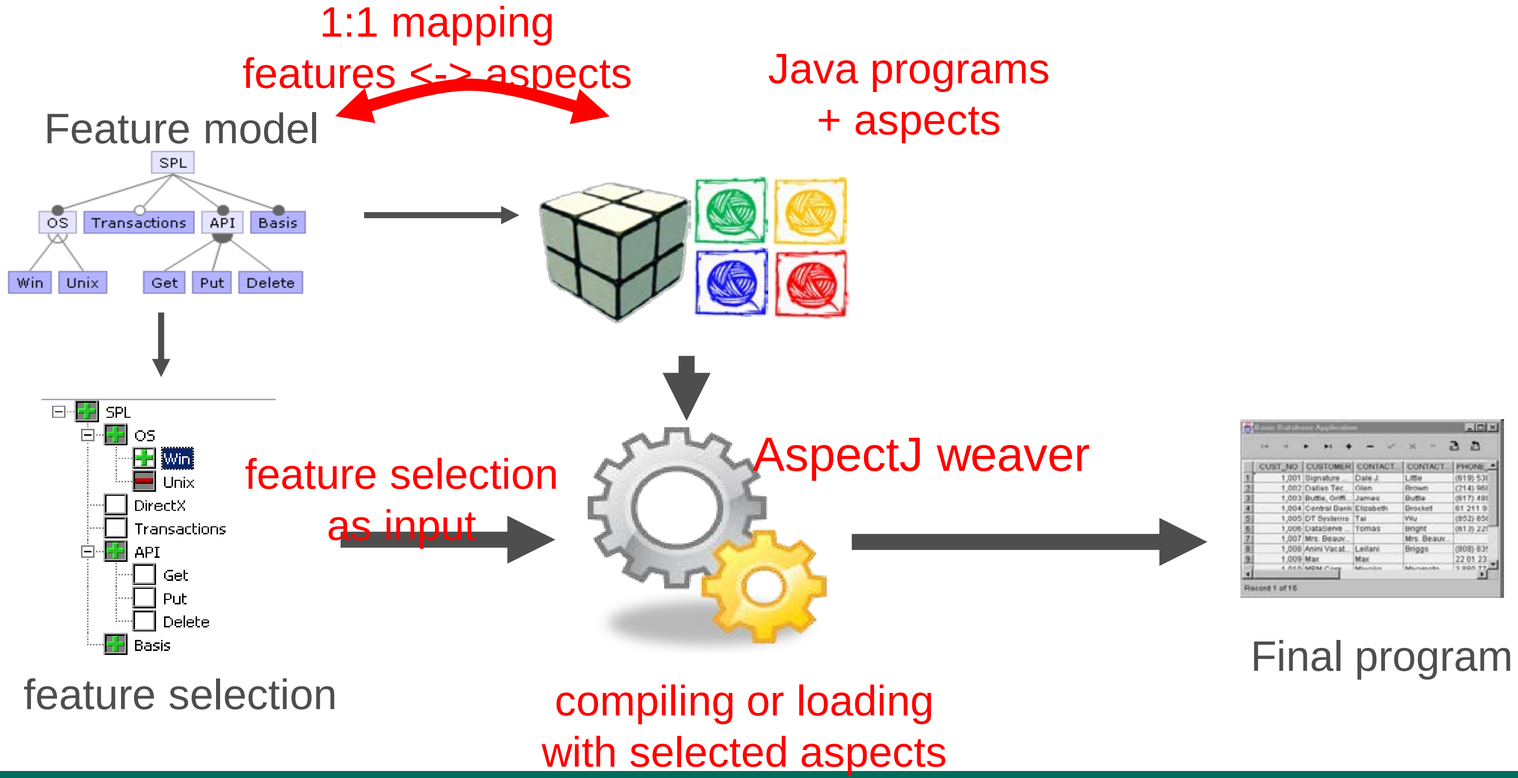
```
aspect ConnectionPooling {
    ...
    Connection around() : call(Connection.new()) {
        if (enablePooling)
            if (!connectionPool.isEmpty())
                return connectionPool.remove(0);
        return proceed();
    }
    void around(Connection conn) :
          call(void Connection.close()) && target(conn) {
        if (enablePooling) {
            connectionPool.put(conn);
        } else {
            proceed();
        }
    }}
```

# Typical aspects III

- Observer: responding to different types of events
  - only need to specify reaction once for complex events with sub-events (cflowbelow)

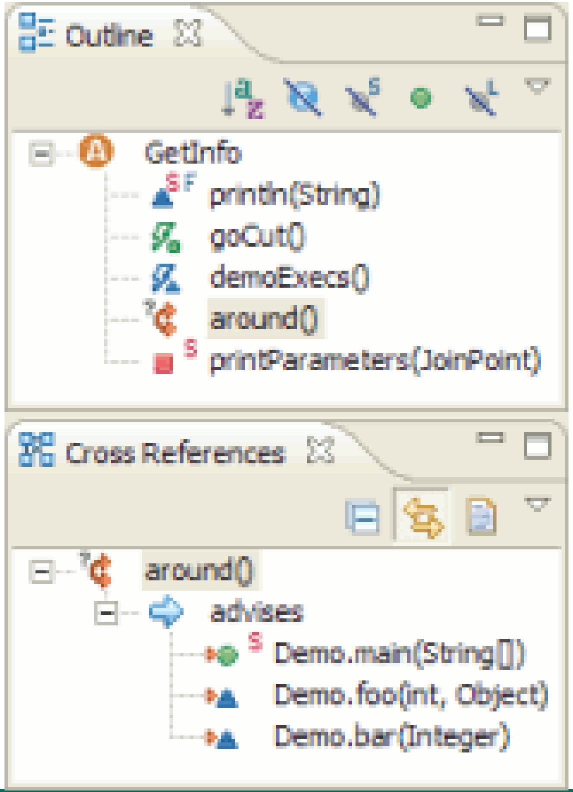
```
abstract class Shape {
    abstract void moveBy(int x, int y);
}
class Point extends Shape { ... }
class Line extends Shape {
    Point start, end;
    void moveBy(int x, int y) { start.moveBy(x,y); end.moveBy(x,y); }
}

aspect DisplayUpdate {
    pointcut shapeChanged() : execution(void Shape+.moveBy(..));

    after() : shapeChanged() && !cflowbelow(shapeChanged()) {
        Display.update();
    }}
```

# Product lines with aspects

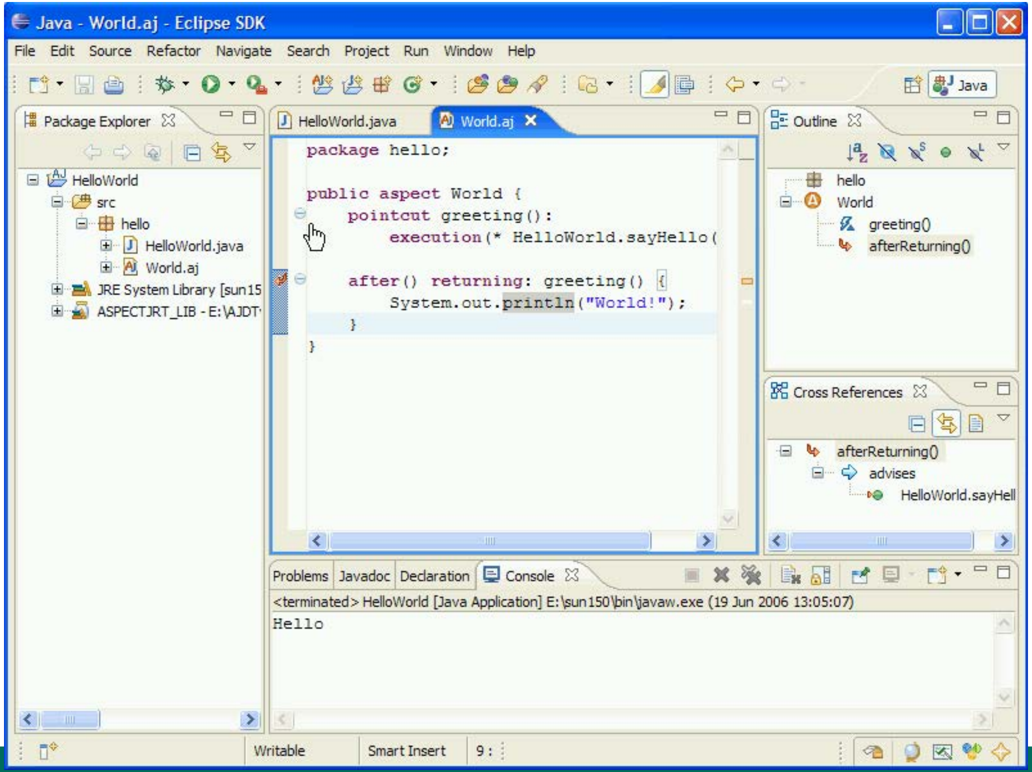


# Development environment AJDT

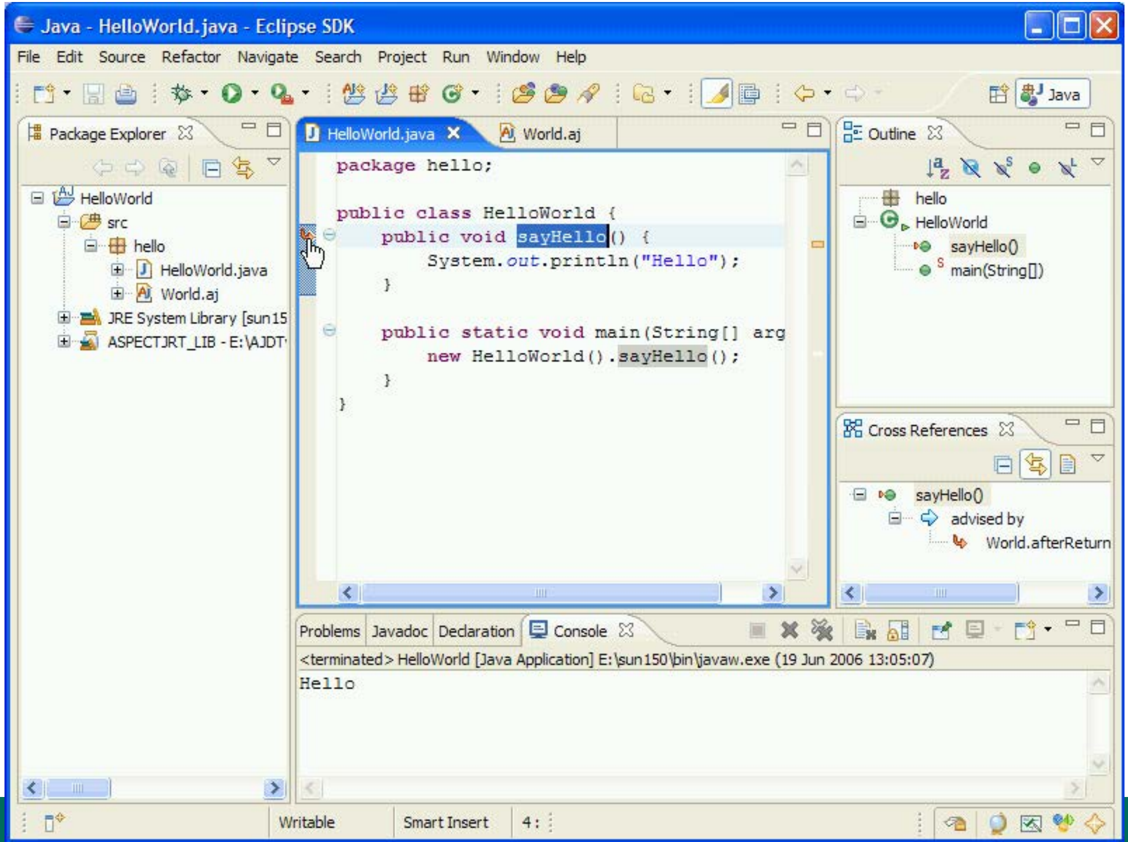
- Eclipse plugin for aspect-oriented programming
  - Integrates aspects into Eclipse; like JDT integrates Java
  - Compiler und debugger integration
  - Syntax highlighting, outline
  - Links between aspect and extended locations



# AJDT in action



# AJDT in action



# Quiz

- Which type of weaving leads to worse runtime performance?
  - Compile-time weaving
  - Run-time weaving

# Features vs. Aspects

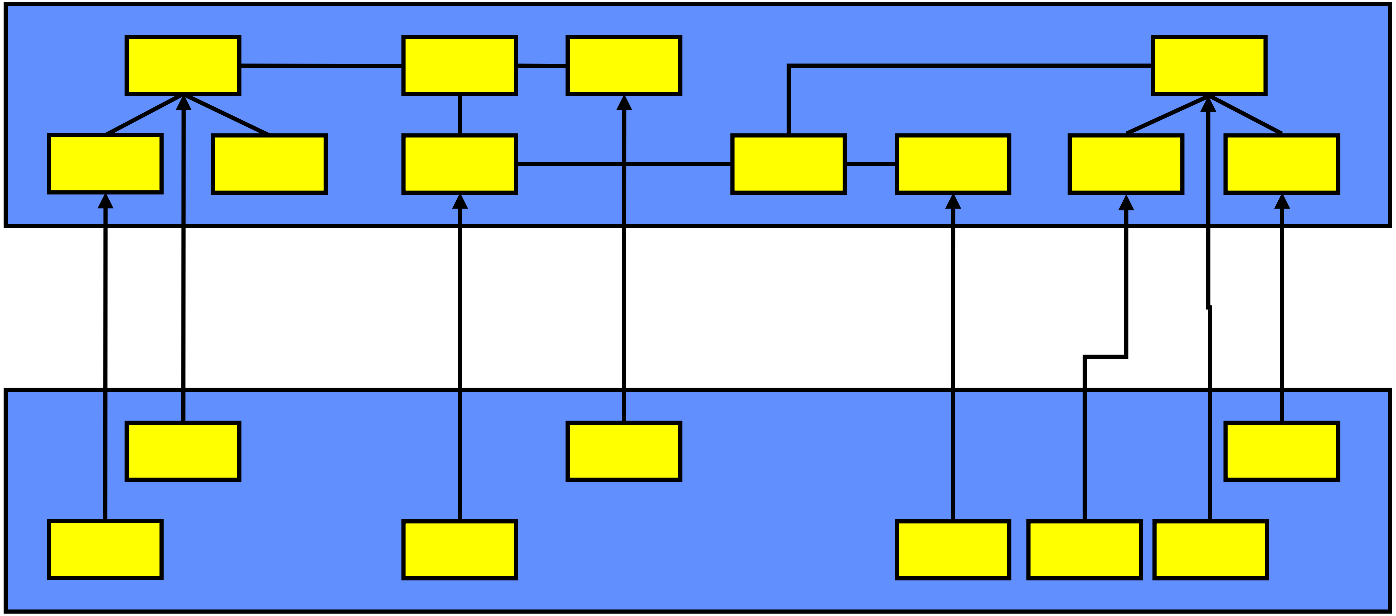
# AOP vs. FOP

- Different philosophies
  - AOP focus on cross-cutting concerns
  - FOP focus on domain abstractions
- Do not implicate specific implementation techniques, but: for object-oriented programming, wide-spread implementation techniques exist
  - AOP → pointcuts & advices, inter-type-declarations
  - FOP → classes, refinements, feature composition

# Motivation

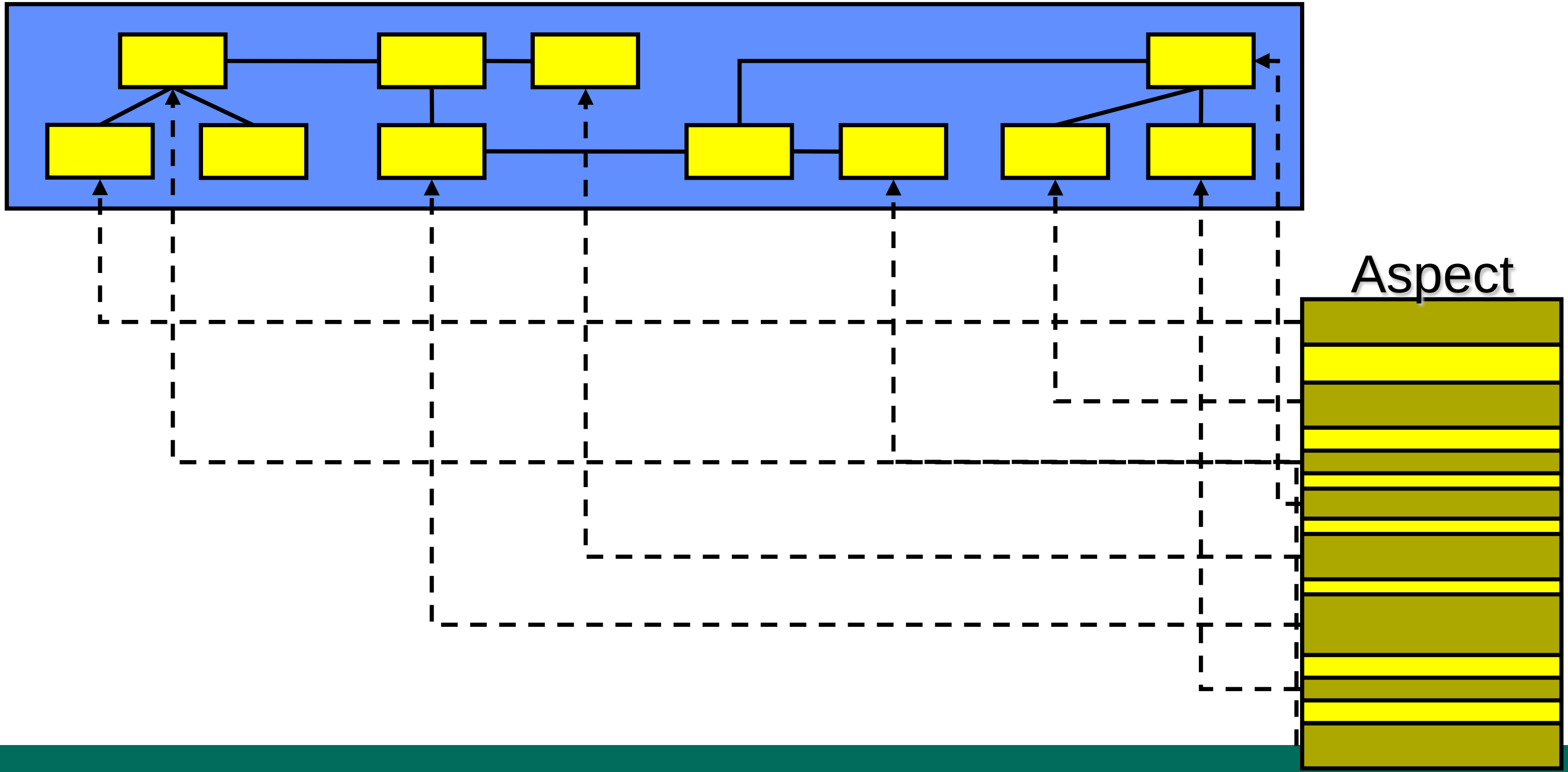
- AspectJ-style AOP and FeatureHouse-style FOP: similar goals
- Studying the use for product line engineering
  - What are differences and commonalities?
  - When to use which?

# AOP vs. FOP



Collaboration

# AOP vs. FOP



# Hetero- vs. homogeneous extensions

Heterogeneous:
different code at
different places

```
class Graph {  …
    Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    e.weight = new Weight(); return e;
  }
  Edge add(Node n, Node m, Weight w)
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    e.weight = w; return e;
  }  …
}
```

```
class Edge {  …
  Weight weight = new Weight();
}
```

Homogeneous:
same code at
different places

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

```
class Edge {
  Node a, b;
  Color color = new Color();
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    Color.setDisplayColor(color);
    a.print(); b.print();
  }}
```

# Dynamic vs. static extensions

- Static:
  change the static structure (new fields and methods)

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

Dynamic:
change the control flow (e.g., extend existing methods)

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

# Simple + advanced dynamic extensions

- **Simple** dynamic extensions
  - Extend method executions
  - Without conditions at run time
  - No access to context of events
    - Only arguments, return type and current object

Simple dynamic extensions are like method extensions with overriding!

- **Advanced** dynamic extensions
  - All kinds of events
  - Conditions at run time (control flow)
  - Access dynamic context
    - For example: are we currently in test execution?

# Examples for simple dynamic extensions

```
class Edge {
  int weight = 0;
  void setWeight(int w) { weight = w; }
  int getWeight() { return weight; }
}
```

**FOP**

```
refines class Edge {
  void setWeight(int w) {
    Super(int).setWeight(2*w);
  }

  int getWeight() {
    return Super().getWeight()/2;
  }
}
```

**AOP**

```
aspect DoubleWeight {
  void around(int w) : args(w) &&
    execution(void
Edge.setWeight(int)) {
      proceed(w*2);
  }
  int around() :
    execution(void Edge.getWeight()) {
      return proceed()/2;
  }
}
```

# Examples for advanced dynamic extensions

- Scenario: *nested graphs*, whose nodes again contain graphs
  - Extension: Logging of print() method, but *only on nodes of the top-level graph*

```
class Node {
  Graph innerGraph;
  void print() {...}
  ...
}
```

```
refines class Node {
  static int count = 0;
  void print() {
    if(count == 0)
      printHeader();
    count++;
    Super().print();
    count--;
  }
  void printHeader() { /* ... */ }
}
```

**FOP**

```
aspect PrintHeader {
  before() :
    execution(void print()) &&
    !cflowbelow(execution(void print())) {
      printHeader();
  }
  void printHeader() { /* … */ }
}
```

**AOP**

| | FOP | AOP |
|---|---|---|
| static | *good support* – fields, method, classes | *limited support* – fields, methods, static inner classes |
| dynamic | *bad support* – only simple extensions (method refinement) | *good support* – advanced extensions, thanks to language support for dealing with execution context |
| hetero-geneous | *good support* – refinements and collaborations | *limited support* – possible, but object-oriented structure gets lost and aspects can get huge |
| homo-geneous | *no support* – one refinement per join point (might lead to code replication) | *good support* – wildcards and logical quantification over pointcuts |

# We Have Learned

- Feature orientation
  - Composition-based, compile-time.
  - Code base seperated into base code + feature modules with feature-specific code (including *refinements*)
  - Composition mechanism produces individual products
  - Provides clear modularity, great for heterogenous extensions and static extensions

# We Have Learned

- Aspect orientation
  - Composition-based, compile-time or run-time
  - Code base seperated into base code + aspects with feature-specific code (pointcuts + advice)
  - Composition mechanism produces individual products
  - Provides clear modularity, great for homogenious extensions and dynamic extensions

# Next Time

- API Design

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY