Software Product Lines
Part 7: Aspect Orientation, AOP vs. FOP

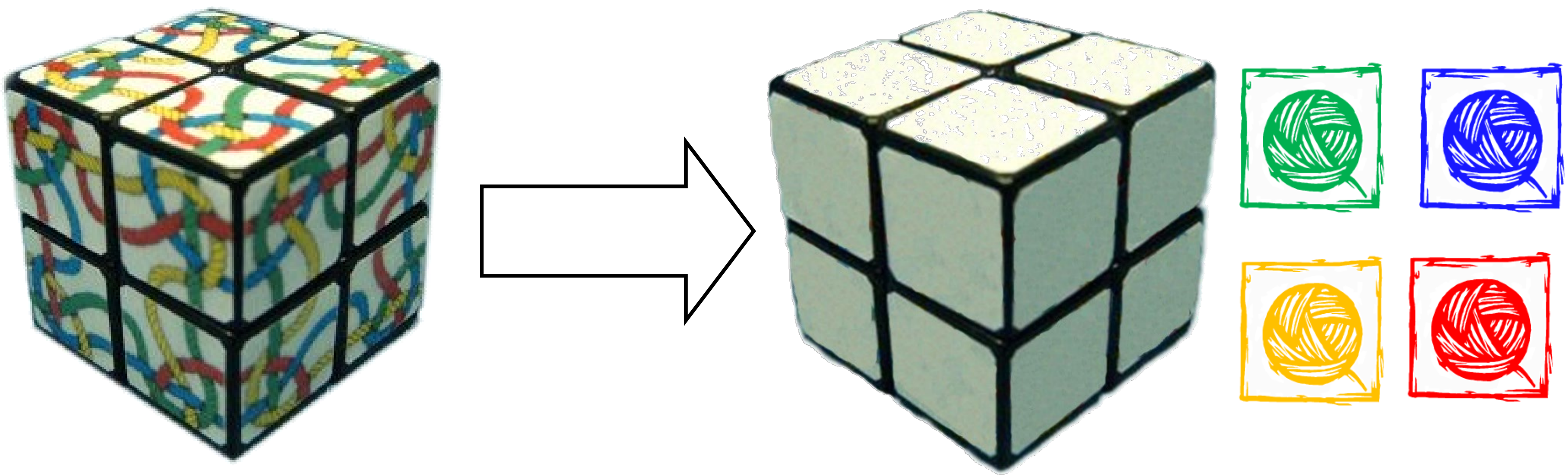
Daniel Strüber, Radboud University

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

Agenda

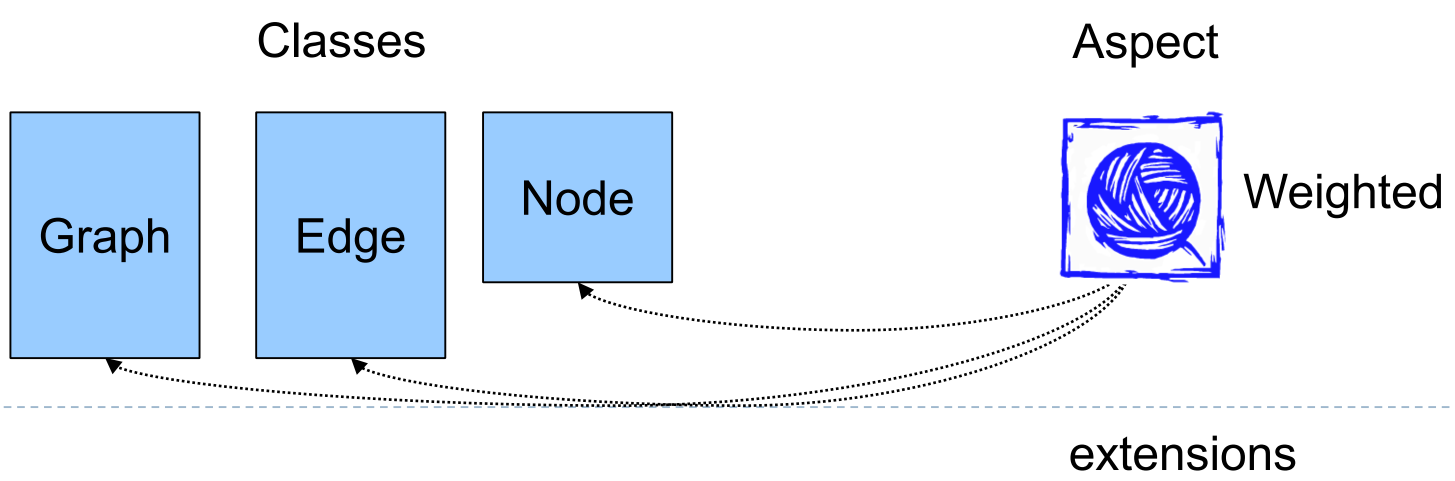
- Aspect-oriented programming (AOP): ideas and concepts
- AspectJ
  - Basics
  - *Join point* model
  - Development environment AJDT
- AOP vs. FOP

Modularizing cross-cutting concerns with aspects



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 (vocabulary to access and manipulate objects)
  - as some sort of feature module



AspectJ

Basics

join point model

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

field access (set)

constructor call

field access (get)

method call

method call

method execution

Pointcut *execution*

- Captures the execution of a method

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

```
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;
  }
}
```

execution

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

Explicit vs. anonymous pointcuts

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

```
aspect A2 {
  after() : execution(int MathUtil.twice(int)) {
    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);
  }
}
```

thisJoinPoint

- To get more information about current join point: use „thisJoinPoint“ in advice

```
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
```

Patterns

- allow „incomplete“ specification of target join point for quantification

```
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(..));
}
```

* as placeholder for one value

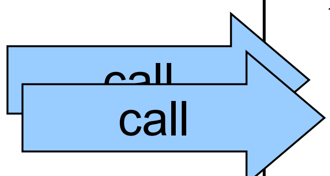
.. as placeholder for multiple values

\+ for subclasses

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

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 {
  public int twice(int i) {
    return i * 2;
  }
}
```

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");
  }
}
```

```
void main(String[] args) {
 new MathUtil();

i);
i);
  System.out.println(i);
}
}
class MathUtil {
  int counter;
  public int twice(int i) {
    counter = counter + 1;
    return i * 2;
  }
}
```

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

call

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");
  }
}
```

```
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;
  }
}
```

call

call

call

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

call

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);
  }
}
```

```
aspect A1 {
  after(int value) : execution(int MathUtil.twice(int)) && args(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 (including patterns) and 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
- For **call**, **set** und **get**: **this** captures the object that calls the method / accesses the field;
  target captures the object whose method is called/field is accessed

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){
    if (j == 0) return 1;
    return i * power(i, j - 1);
  }
}
```

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(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(..)))
```

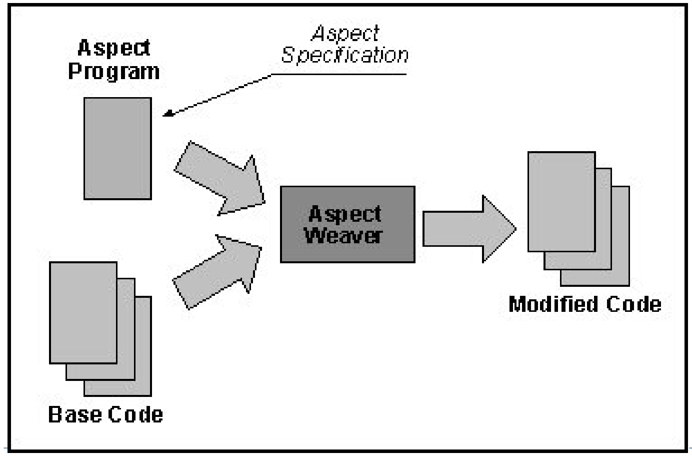
```
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
```

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

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

Aspect weaving

- **Weaving**: the process of applying aspects to a target object
- 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);
  }
}
```

Basic Graph

```
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 { ... }
}
```

Color

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);
  }
}
```

Basic Graph

```
aspect ColorAspect {
  static class Colored { Color color; }
  declare parents: (Node || Edge) extends Colored;
  before(Colored c) : execution(void print()) && this(c) {
    Color.setDisplayColor(c.color);
  }
  static class Color { ... }
}
```

Color

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 resource

```
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();
        }
    }
}
```

Zoom quiz

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

Advanced concepts

Methods in aspects

Order during aspect weaving

Abstract aspects for reuse

Aspect methods

- Like normal classes: aspects can contain methods and fields can mix with inter-type declarations, pointcuts and advices
- Advice executed in the context of the aspect, not of the extended class („third person perspective“)

```
aspect Logging {
    PrintStream loggingTarget = System.out;
    private void log(String logStr) {
        loggingTarget.println(logStr);
    }
    pointcut anySetMethodCall(Object o) :
                      call(public *.set*(..)) && target(o);
    after(Object o) : anySetMethodCall(o) {
        log('A public method was called on '+ o.getClass().getName());
    }
}
```

Aspect precendence

- If not defined explicitly: order in which aspects are weaved undefined
  - Practically: determined by definition order
- Order can be relevant if multiple aspects extend the same join point
  - Example:
    - Aspect 1 implements synchronization with an *around* advice
    - Aspect 2 implements logging with an *after* advice for the same join point
    - Depending on weaving order, logging code synchronized or not

Aspect precendence II

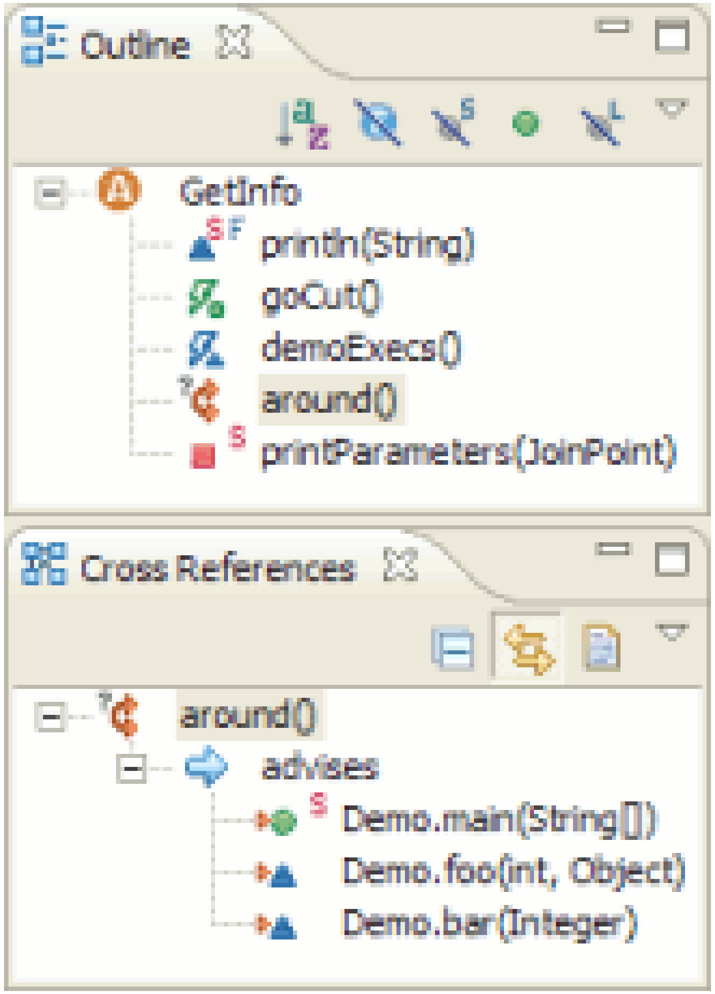
- Explicit declaration possible: *declare precedence*

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

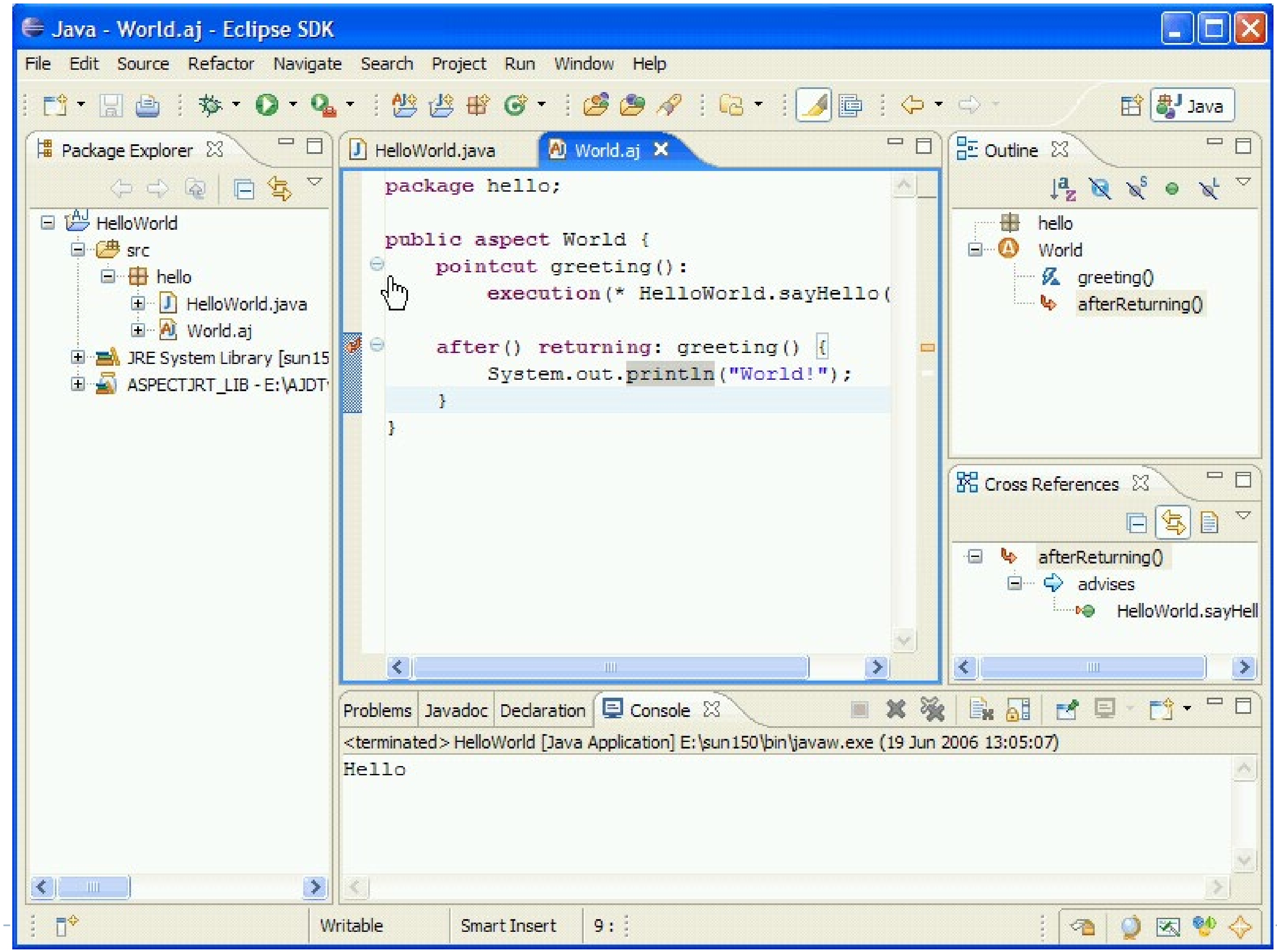
- Aspect with highest priority weaved first
  - On *before* first, *on after* last, on *around* as most outer

Development environment AJDT

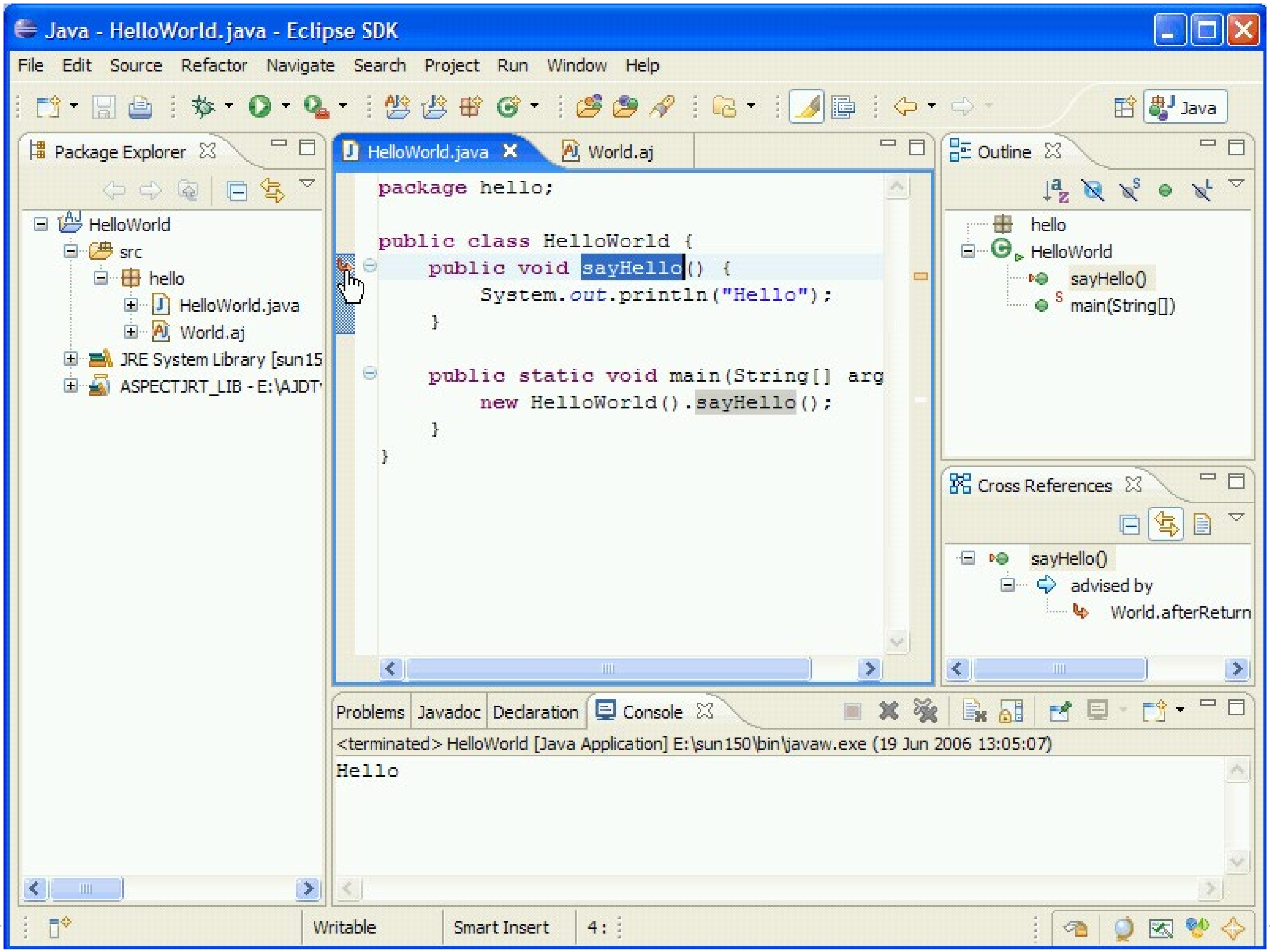
- Eclipse plugin for aspect-oriented programming
  - Integrates aspects into Eclipse; like JDT integrates Java
  - Compiler und debugger integration
  - Syntax highlighting, outline
  - Links between aspect and extended locations (shows where locations are extended)



AJDT in action



AJDT in action



Discussion

Principle of obliviousness I

- Principle of obliviousness (*onbewustheid*) says that the base program does not need to know about aspects
  - „program Java as always, add the aspects later“
- Ideally means that…
  - classical OO design good enough, do not have to prepare source code for aspects
  - base program developers need no knowledge about aspects; few skilled specialists sufficient
  - assumption: AOP language is sufficiently expressive

Principle of obliviousness II

- Controversial because...
  - ... programmers might change the base code without noticing the effect to aspects and addressing it („fragile pointcut problem“; no explicit interfaces)
  - ... can lead to bad design if aspects are just „hacked into“ the system
  - ... might require complex constructs like cflow or call && withincode to still express changes in unprepared code

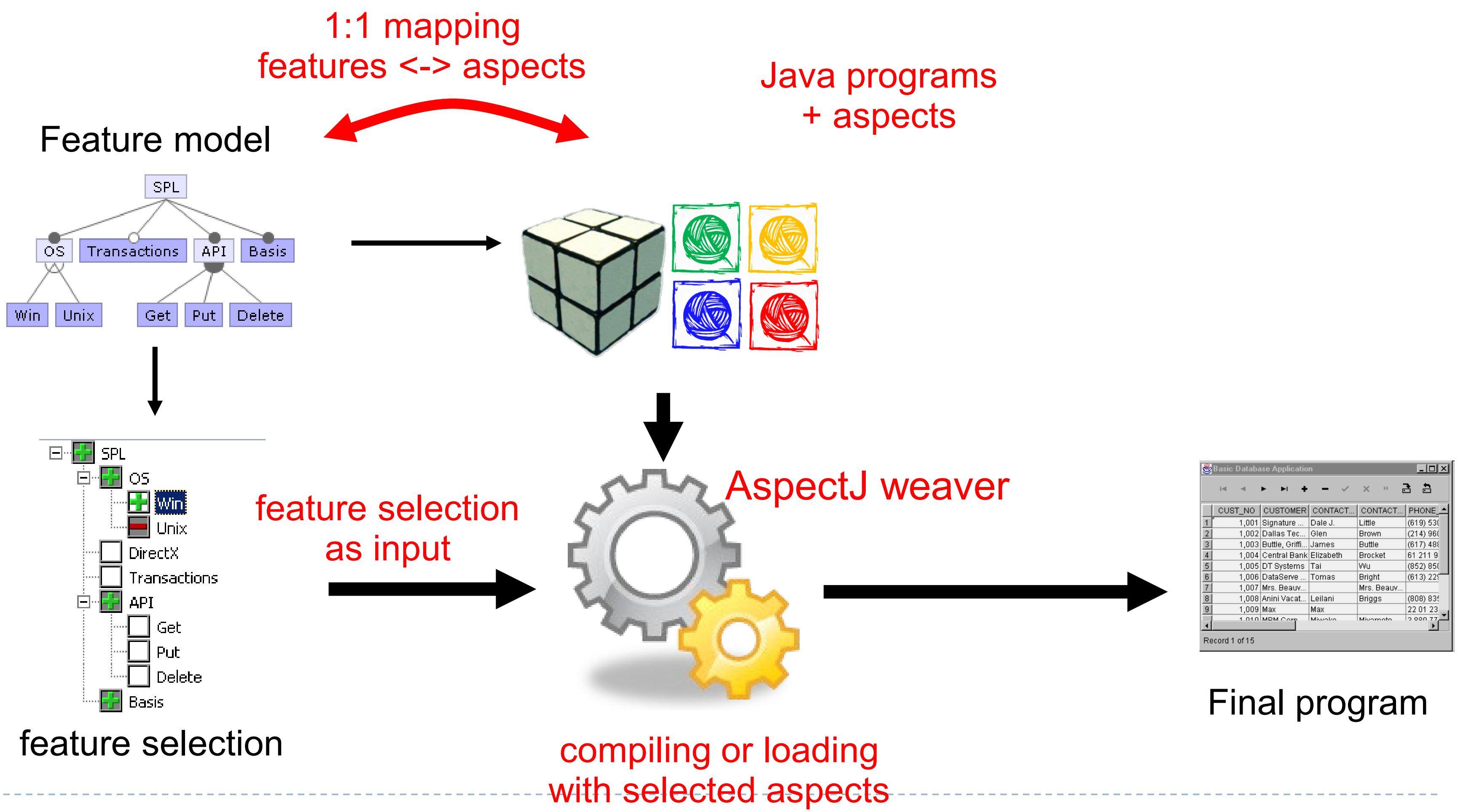
Why aspects in the first place?

- Support cohesive implementation of cross-cutting concerns
- Support to **quantify** over join points (homogenous extensions at many places)
- Support analyses of **dynamic** properties of control flow (cflow) that require elaborate workarounds in OOP

Aspects for features?

- Aspects are similar to collaborations
  - Can statically add code
  - Can extend methods
  - Beyond that, support dynamical homogenous extensions based on control flow
- Aspects can be switched on or off
  - Manually in Eclipse: Right click on aspect and „Exclude from build path“
  - Automated with build system
  - FeatureIDE in combination with AJDT

Product lines with aspects



Problems

Lexical comparisons / Fragile Pointcut Problem /
Complex Syntax

Pointcuts use lexical comparisons

- Pointcuts involve name comparisons, but names can be chosen freely
- Patterns use naming conventions, for example, „get*“, „draw*“ usw.

```
class Chess {
    void drawKing() {...}
    void drawQueen() {...}
    void drawKnight() {...}
}

aspect UpdateDisplay {
    pointcut drawn : execution(* draw*(..));
    ...
}
```

Fragile pointcut problem / evolution paradox

- Changes of base code:
  can lead to new join points not being captured, or old joint points not being captured anymore
- Chess example: A developer unaware of the aspect adds a method for matches ending in a draw: „void draw()“
- Such changes can go without notice. Impossible to know if the intended pointcuts have been captured.

Complex syntax

- AspectJ is expressive and needs special constructs for that
- Leads to complicated syntax for extensions, even for simple extensions. Example method extensions:

OOP / FOP

```
public void delete(Transaction txn, DbEntry key) {
    super.delete(txn, key);
    Tracer.trace(Level.FINE, "Db.delete", this, txn, key);
}
```

AOP

```
pointcut traceDel(Database db, Transaction txn, DbEntry key) :
    execution(void Database.delete(Transaction, DbEntry))
    && args(txn, key) && within(Database) && this(db);
after(Database db, Transaction txn, DbEntry key): traceDel(db, txn, key) {
    Tracer.trace(Level.FINE, "Db.delete", db, txn, key);
}
```

Aspects vs. Features

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, mixin/jampack composition

Motivation

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

Jak-style feature modules

Basic Graph

```
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);
  }
}
```

Jak-style feature modules

Basic Graph

```
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);
  }
}
```

Weight

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

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

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

AspectJ-style aspects

Basic Graph

```
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);
  }
}
```

AspectJ-style aspects

Basic Graph

```
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();
  }
}
```

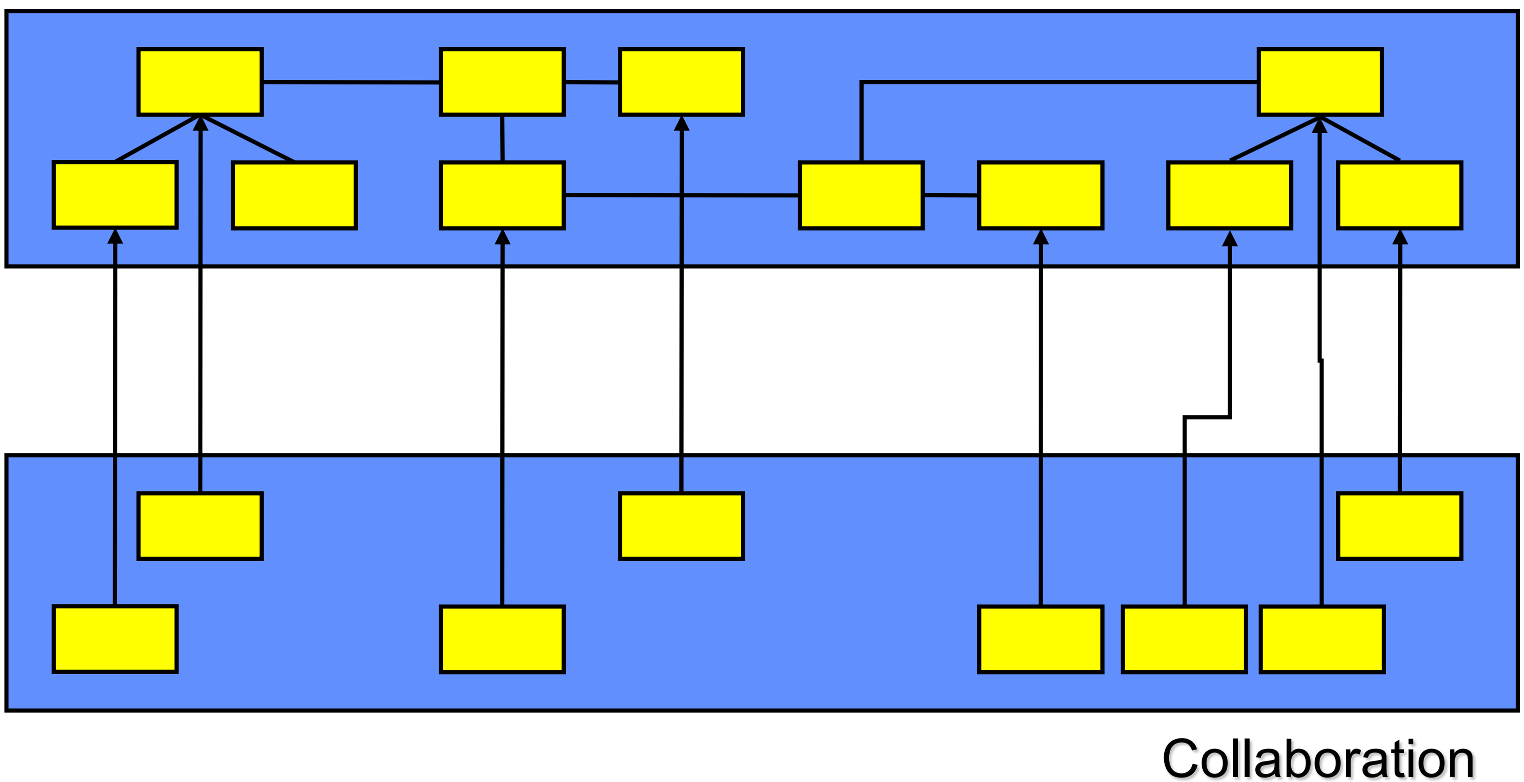
```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) {
    a = _a; b = _b;
  }
  void print() {
    a.print(); b.print();
  }
}
```

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

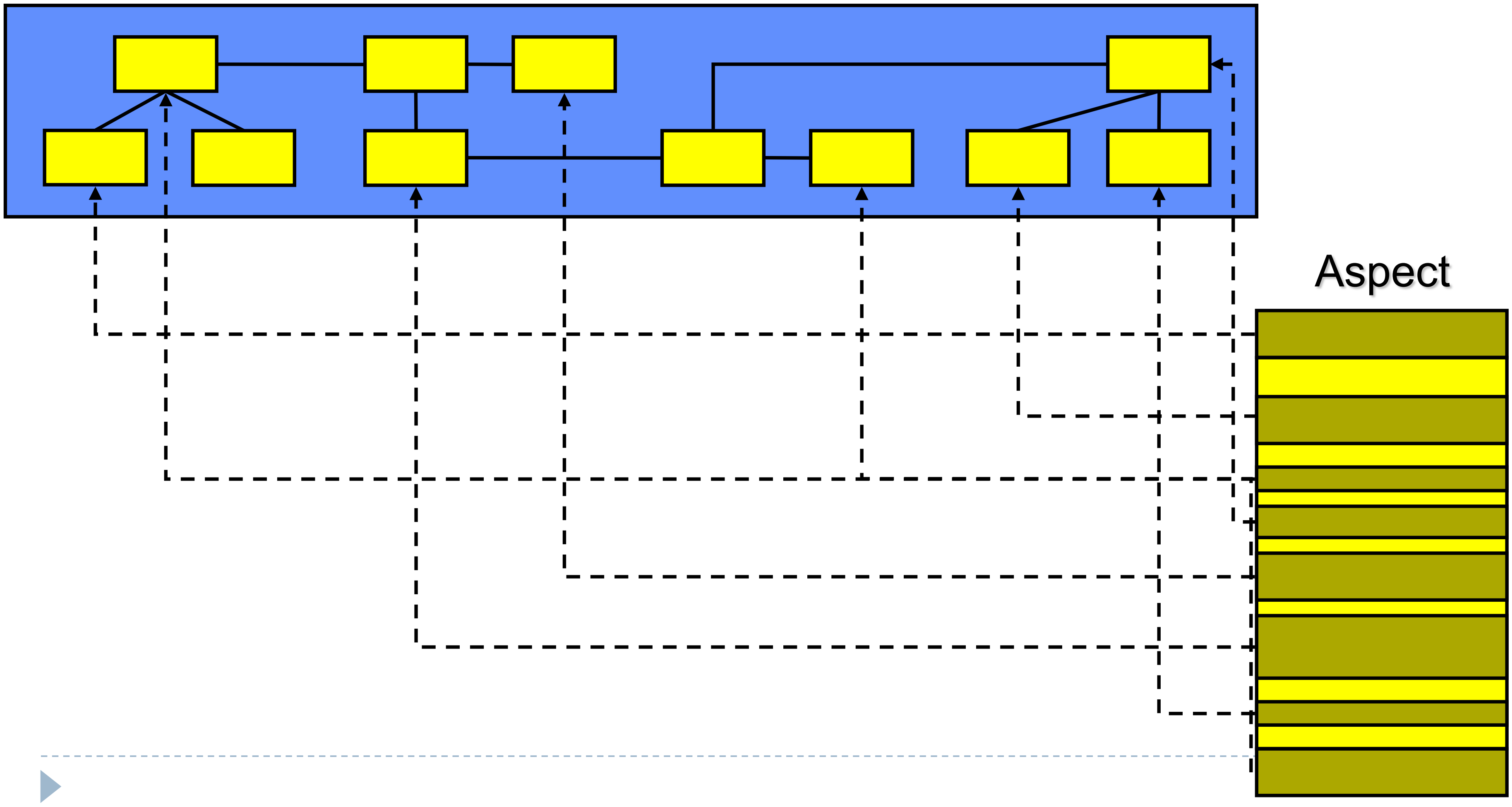
Color

```
aspect ColorAspect {
  static class Colored { Color color; }
  declare parents: (Node || Edge) extends Colored;
  before(Colored c) : execution(void print()) && this(c) {
    Color.setDisplayColor(c.color);
  }
  static class Color { ... }
}
```

AOP vs. FOP



AOP vs. FOP



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

- 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();
  }
}
```

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

Simple dynamic extensions are like method extensions with overriding!

Examples for simple dynamic extensions

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

Jak

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

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

AspectJ

```
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:
  - Consider *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() {...}
  ...
}
```

Jak

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

AspectJ

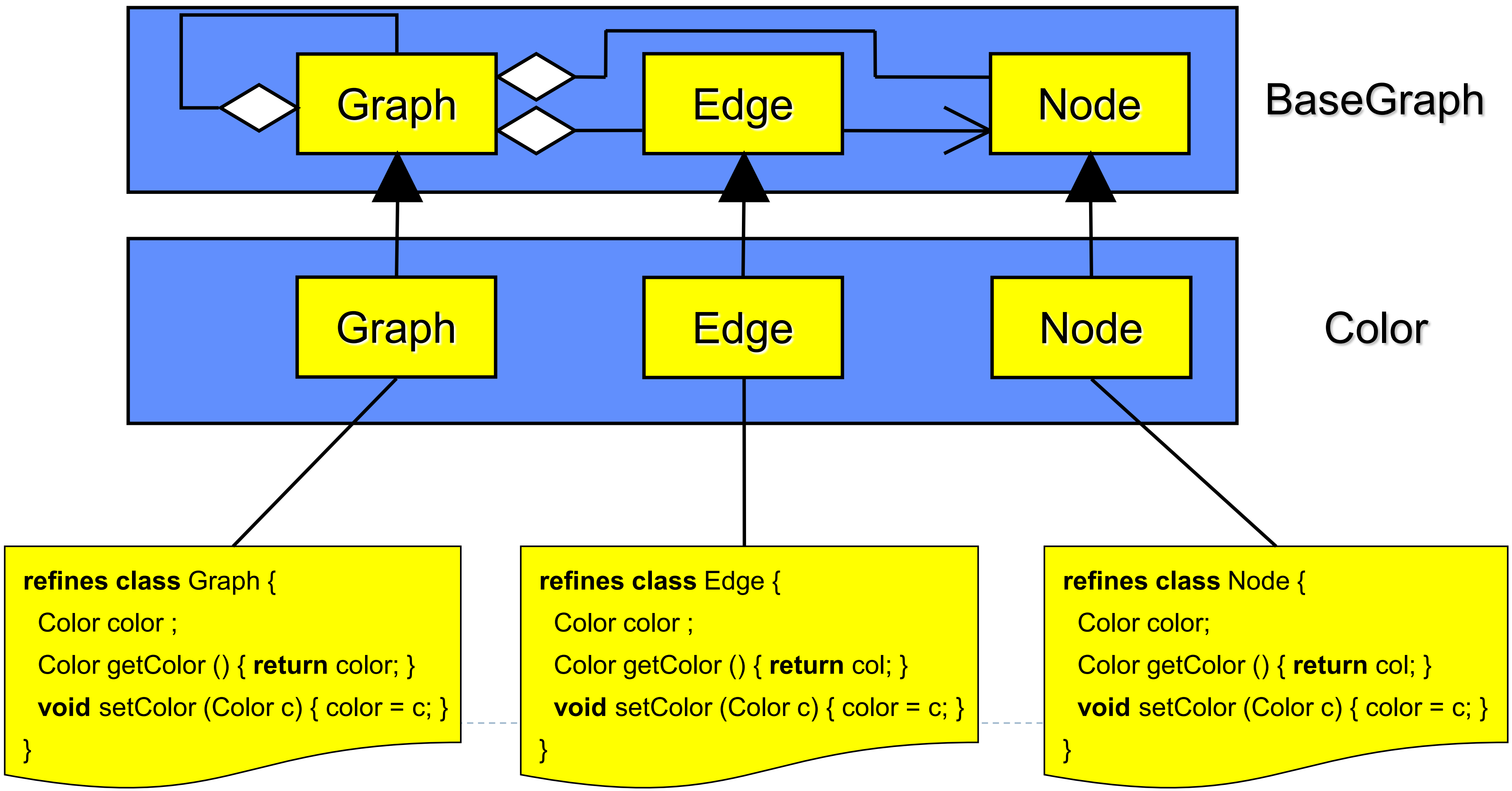
```
aspect PrintHeader {
  before() :
    execution(void print()) &&
    !cflowbelow(execution(void print())) {
      printHeader();
  }
  void printHeader() { /* … */ }
}
```

Comparison: FOP vs. 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

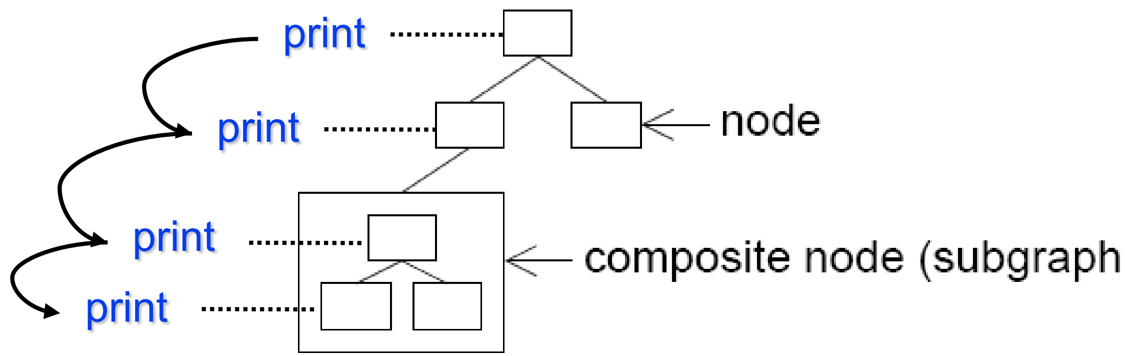
Collaborations instead of aspects

- Homogeneous extensions lead to replication



Collaborations instead of aspects

- Workarounds for dynamic extensions



AspectJ

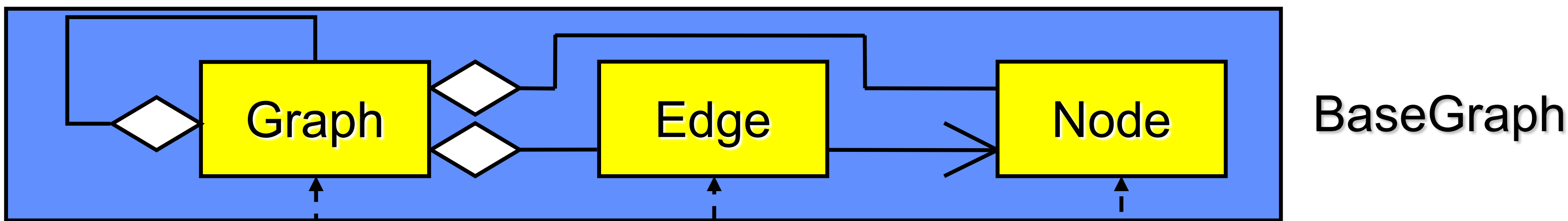
```
aspect PrintHeader {
  before() : execution(void print ()) &&
    !cflowbelow (execution(void print ())) {
      printHeader ();
  }
  void printHeader () { /* … */ }
}
```

Jak

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

Aspects instead of collaborations

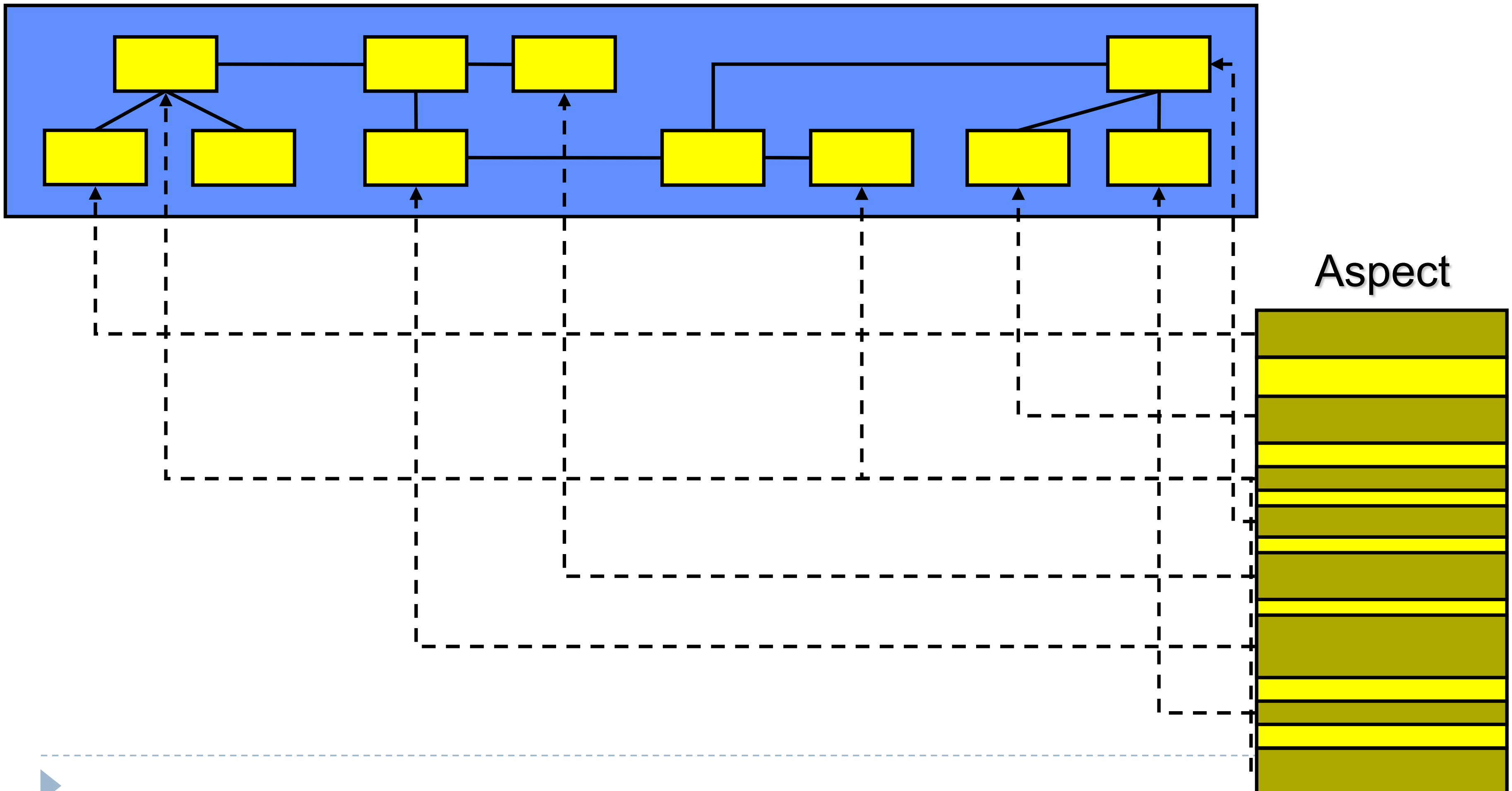
- One aspect per collaborations



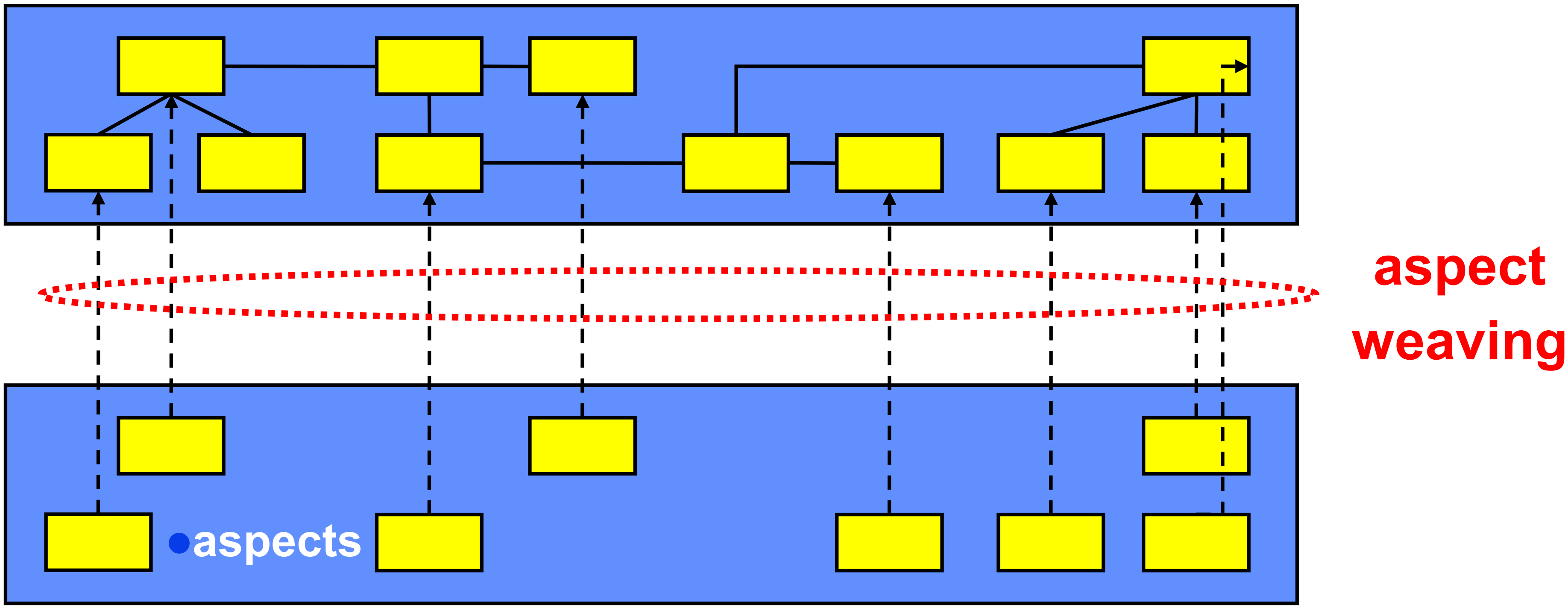

```
aspect AddWeight {
  Edge Graph.add(Node n, Node m, Weight w) {
    Edge res = add(n, m); res.weight = w; return res;
  }
  Weight Edge.weight;
  after(Edge e) : this(e) &&
    execution(void Edge.print ()) {  /* … */  }
}
```

Object-oriented structure not explicit

A question of scalability



Alternative: one aspect per role



Aspect weaving replaces refinement –
more complicated syntax without additional benefit

Summary

- Aspect-oriented programming describes changes declaratively and quantifies over program
- AspectJ is the AOP extension for Java;
  AJDT as development environment
  - Concepts: Join-Point, Inter-Typ-Deklaration, Pointcut, Advice…
- AOP vs. FOP
  - Use of aspect or collaborations depends
    on the problem at hand
  - Aspects and collaborations are fit for different purposes

Literature I

- eclipse.org/aspectj, eclipse.org/ajdt
- R. Laddad. AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications, 2003.
  [Practical book on AspectJ with many examples]
- R.E. Filman and D.P. Friedman, Aspect-Oriented Programming is Quantification and Obliviousness, In Workshop on Advanced Separation of Concerns, OOPSLA 2000
  [Attempt of a definition of AOP]

Literature II

- G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Proc. Europ. Conf. on Object-Oriented Programming (ECOOP), 1997
  [Original paper describing problem and AspectJ as solution]
- G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In Proc. Europ. Conf. on Object-Oriented Programming (ECOOP), 2001
  [Presents the AspectJ language]