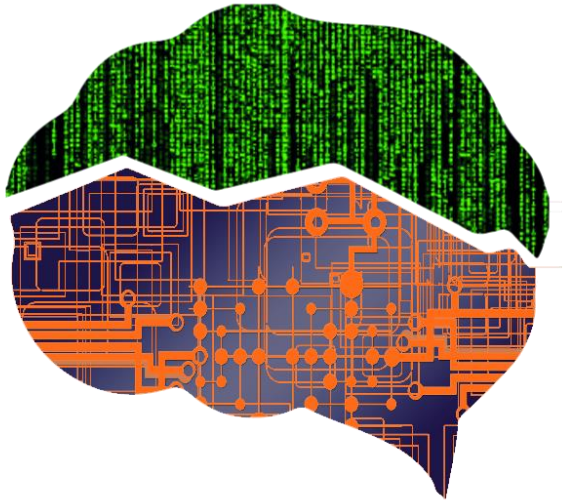
Software Product Line Engineering

Aspect-Oriented Development

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

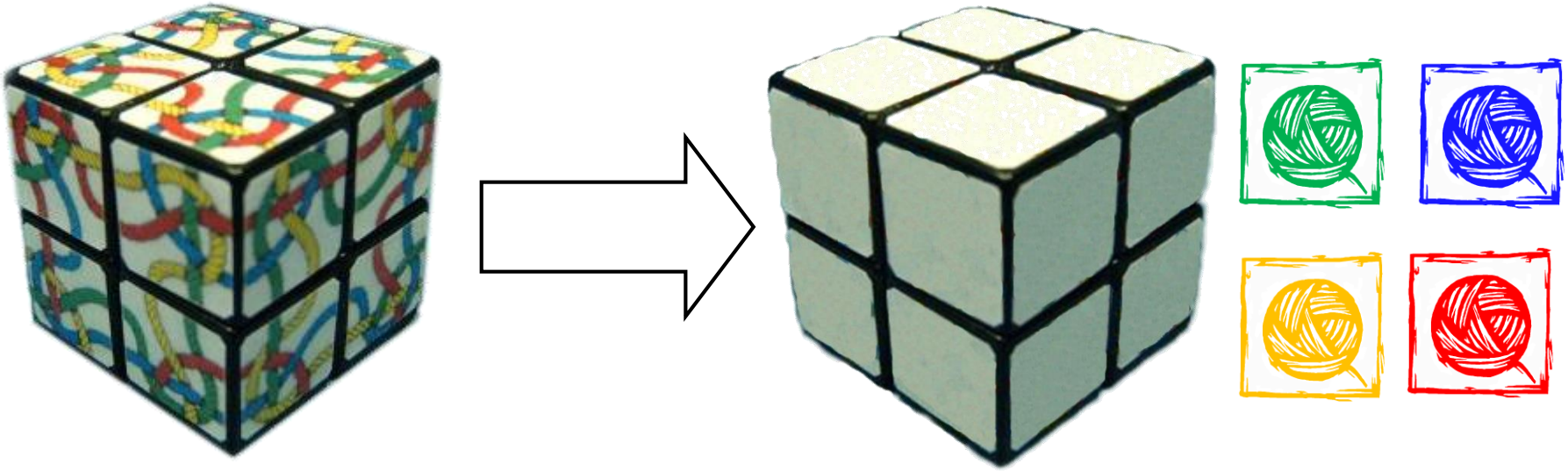


Bauhaus-Universität Weimar

Agenda

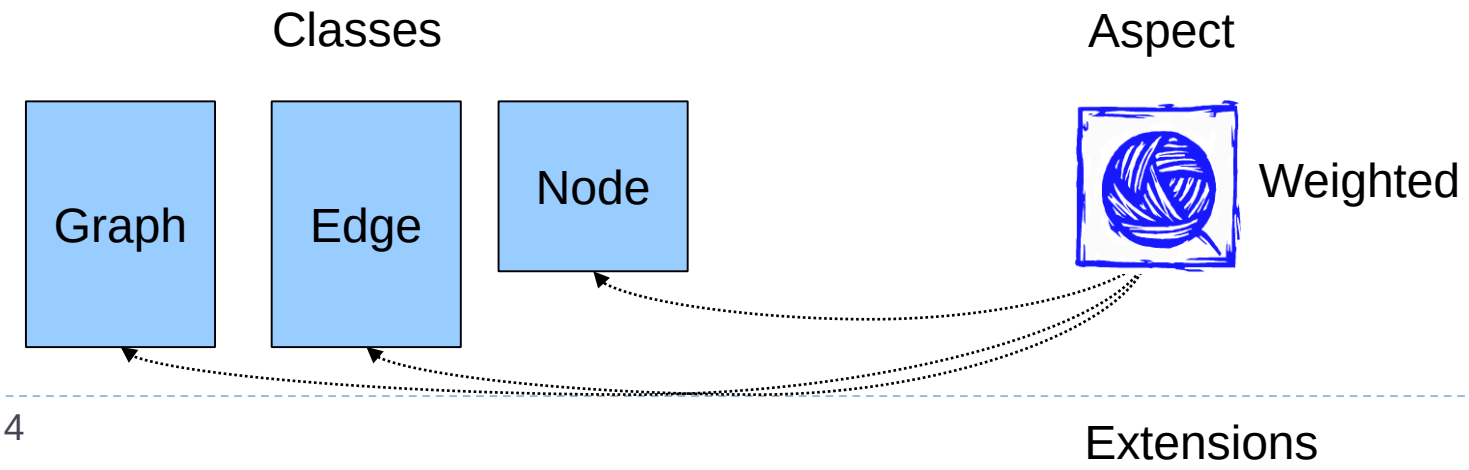
- Aspect-oriented programming: Ideas and concepts
- AspectJ
  - Basics
  - Join-Point-Modell
  - Development environment AJDT
- Discussion

Modularizing Crosscutting Features with Aspects



Idea

- Modularizing crosscutting features in a single aspect
- This aspect describes changes that are made by this feature to the remaining software
- Interpretation as program transformation, meta-object protocol, or feature module



AspectJ

Basics

Join-Point-Modell

AspectJ

- AspectJ is an AOP extension to Java
- Aspects are implemented similar to classes, but with additional language constructs
- The base code is still Java
- Aspects will be “weaved” into the base Java code using a special Aspect-Compiler

What is an Aspect?

- An aspect in AspectJ can
  - Manipulate class hierarchies
  - Add methods and fields to a class
  - Extend methods with additional code
  - Catch events, such as method calls, field accesses
  - Execute additional or alternative code based on the events

Static Extension

- Static extension with „inter type declaration“
  - For instance, add method X to class Y

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

Dynamic Extension

- Dynamic extensions based on the joint-point model
  - An event at program execution, such as a method call or field access, is called a **join point**
  - A **pointcut** is a predicate to select join points (JPs)
  - An **advice** is code that is executed if a JP has been 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);
    }
}
```

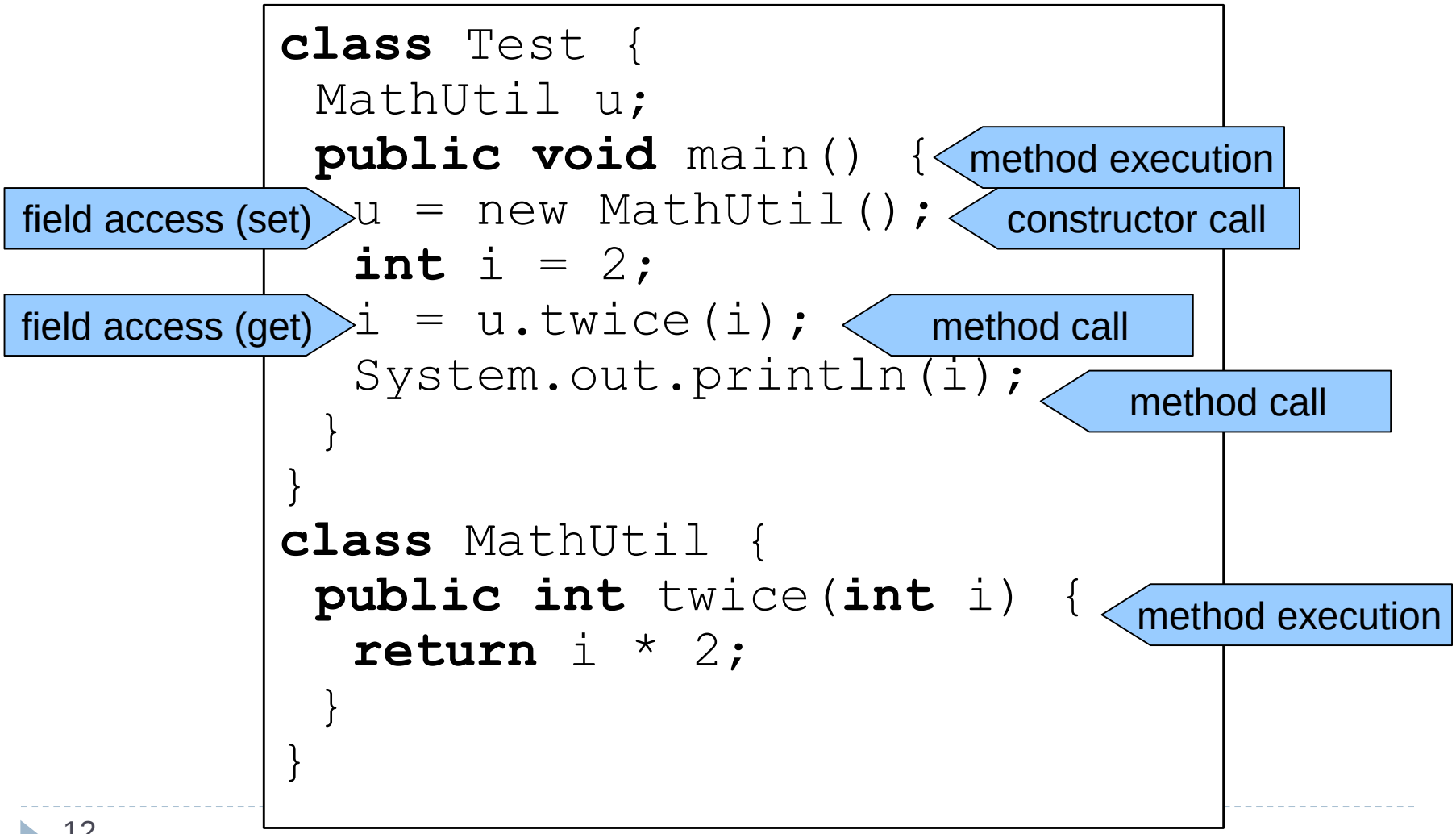
Quantification

- An important property of pointcuts is the ability to declaratively describe multiple joint points at the same time
- Examples:
  - Execute advice X always if the method „setWeight“ in class „Edge“ is called
  - Execute advice Y always if there is an access to any field in class „Edge“
  - Execute advice Z always if there is a public method called and we have executed method „initialize“ previously

AspectJ – Join Point Model

- Join points exist when
  - Calling a method
  - Executing a method
  - Calling a constructor
  - Executing a constructor
  - Accessing a field (read or write)
  - Catching an exception
  - Initializing an object
  - Executing an advice

Join Point Example

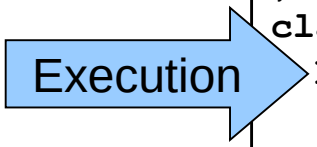


Pointcut Execution

- When execution 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;
  }
}
```



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

Explicit and Anonymous Pointcuts

Anonymous

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

Explicit

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

Advice

- Additional code before, after, or around a join point

- Around advice:
  - Replacing existing code (alternative implementation)
  - Executing existing code with 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

- Keyword „thisJoinPoint“ is used to get more information about the current joint 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());
  }
}
```

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

Patter

- Allows for wildcards when quantifying target of an advice

```
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 Ptypisch() : execution(* MathUtil.twice(..));
}
```

***** Wild card for a single value

.. Wild card for a multiple values

+ for subclasses

Pointcut call

- Matches method calls
- Similar to execution, but at caller side

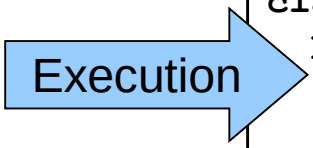
```
aspect A1 {
  after() : call(int MathUtil.twice(int)) {
    System.out.println("MathUtil.twice called");
  }
}
```


```
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“ as special method

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




```
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

- Matches access to a field (instance variable)

```
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)
```

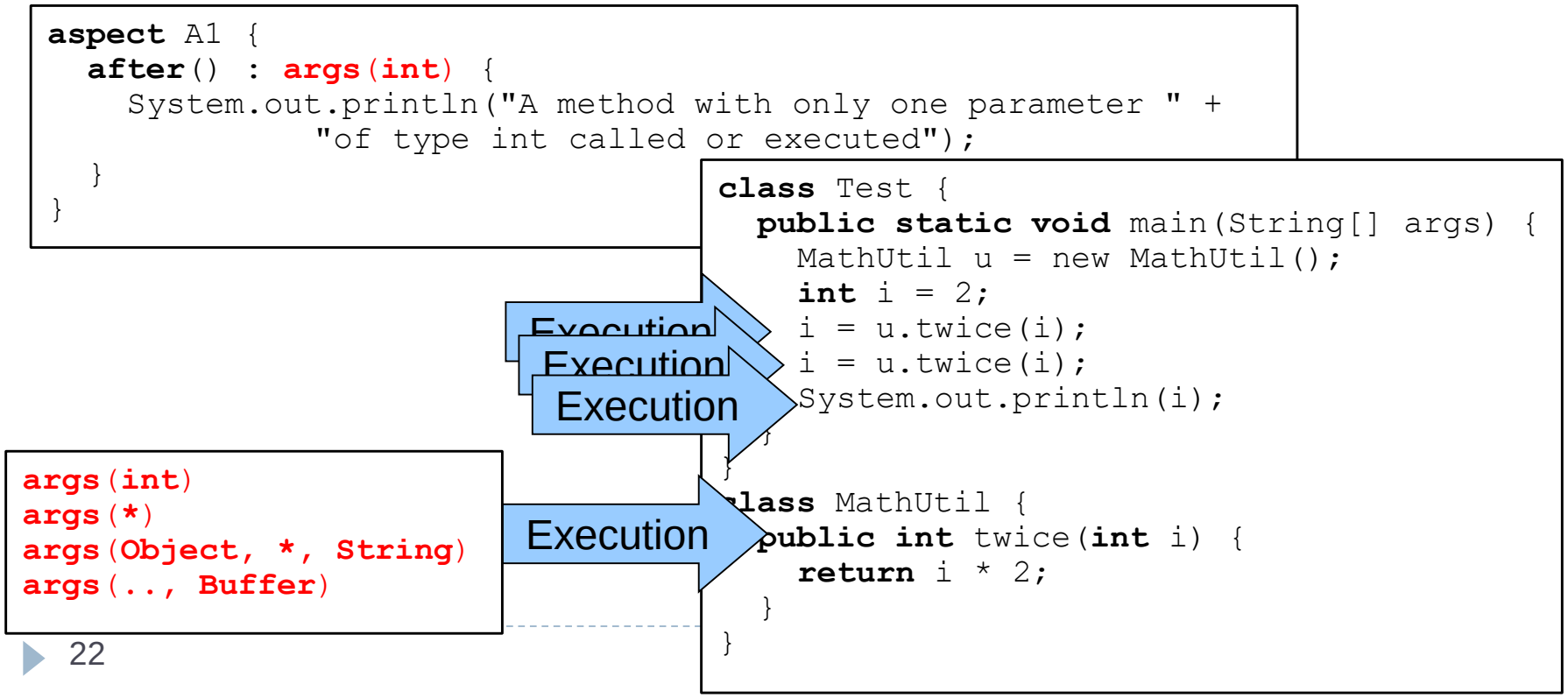
Execution

```
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;
  }
}
```

Pointcut args

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



Combination of Pointcuts

- Pointcuts can be combined with &&, ||, 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);
}
```

Parameterized Pointcuts

- Pointcuts can have parameters to be used within the advice
- Advice can obtain contextual information
- Pointcut args is used with a variable instead of declaring a 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 uses Parameter

- Exemplary advice using a parameter and changing it:

```
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 und target

- this and target store the involved classes
- Can be used with types (incl. wildcards) and with 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);
}
```

- At execution: this und target obtain the object that on which the method is called
- At call, set, and get: this obtains the object that calls the method or accesses the field; target obtains the object on which the method is called or on which the field is accessed

Pointcuts Within und Withincode

- Limit scope of join points to certain areas
- Example: only JPs to method twice that are originating from Test or Test.main

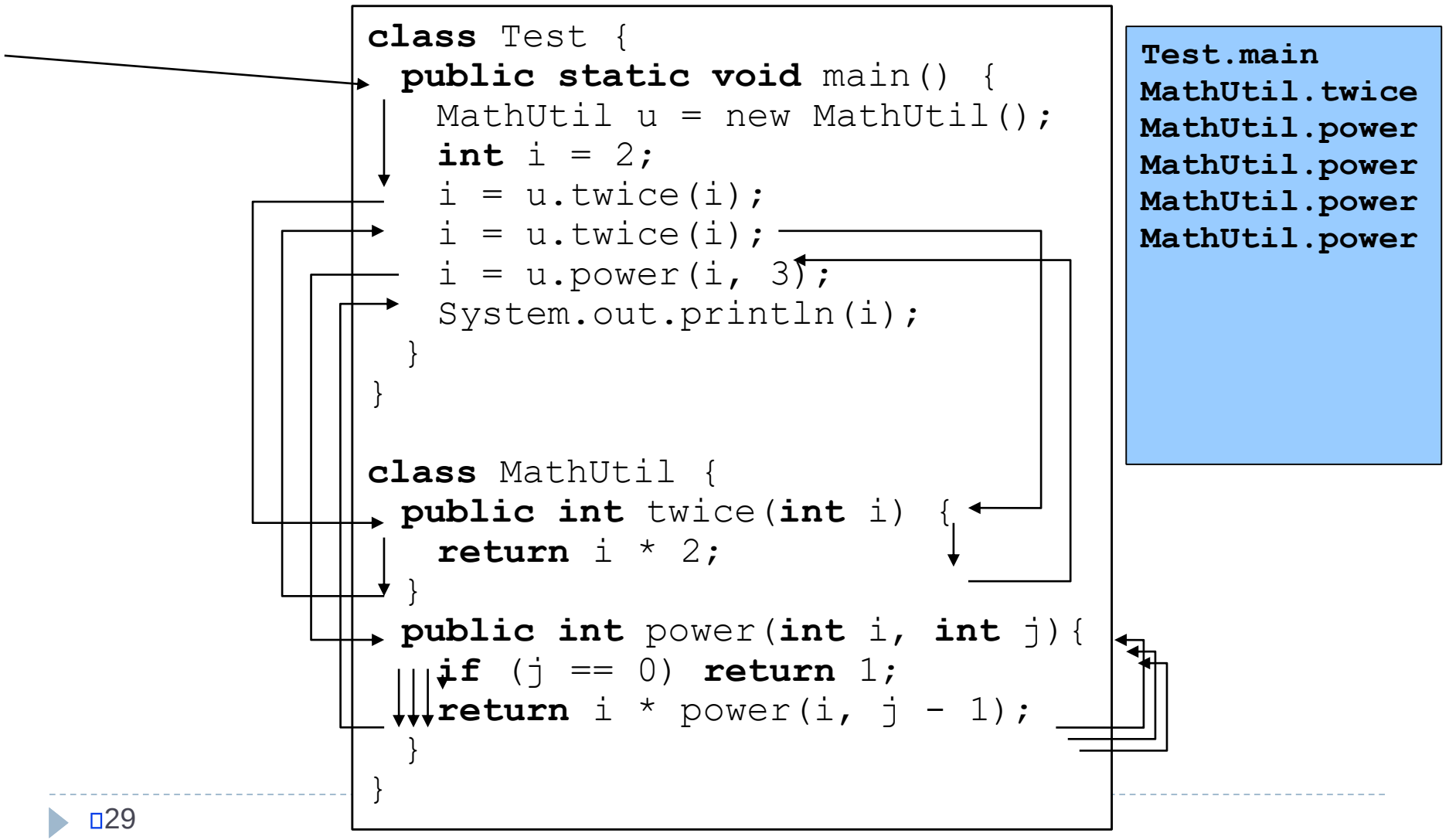
```
aspect A1 {
  pointcut P1(): call(int MathUtil.twice(int)) && within(Test);
  pointcut P2(): call(int MathUtil.twice(int)) && withincode(* Test.main(..));
}
```

Pointcuts cflow und cflowbelow

- Match all join points that occur in the control flow of another join point

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

Control Flow



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))
```

Weaving Aspects

- How are aspects executed?
  - Meta object protocols and interpreted languages: evaluated at runtime
  - AspectJ/AspectC++/…: “weave” the aspect by the compiler
- Weaving:
  - Inter type declaration are added in affected classes
  - Advice is transformed to a method
  - Pointcuts: Method calls from join points to advice
  - Dynamic extensions: add additional source code at every potential join point that can dynamically check whether a condition is true and if so call the corresponding advice

Aspects in Graph Library

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();
  }
}
```

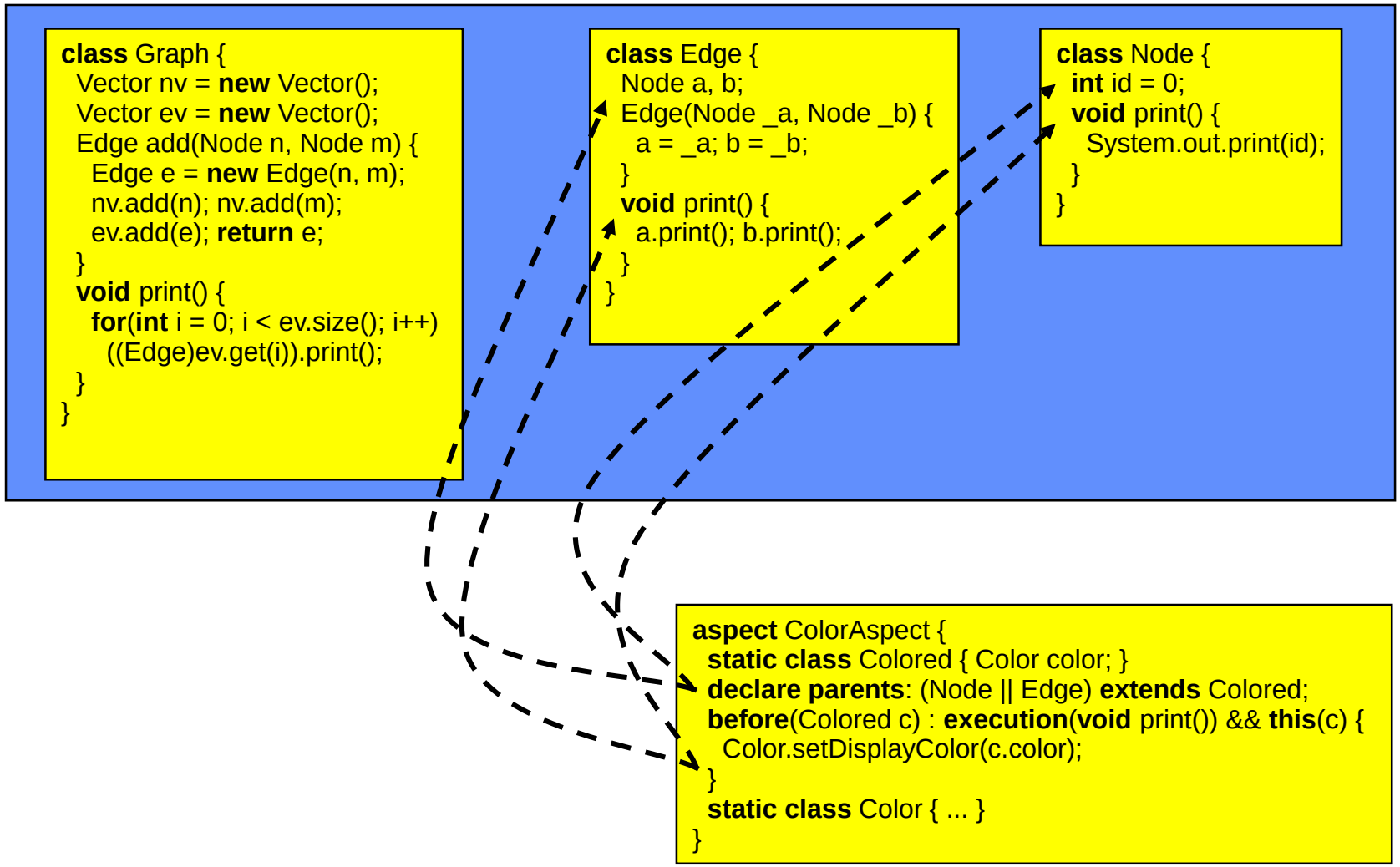
```
class Node {
  int id = 0;
  void print() {
    System.out.print(id);
  }
}
```

Color

```
aspect ColorAspect {
  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 { ... }
}
```

Aspects in Graph Library: Alternative

Basic Graph



Color

Typical Aspects

- Logging, Tracing, Profiling
  - Add identical 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 are centrally implemented and can be used at different positions

```
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
  - Collect different events
  - React on nested events only once (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();
    }
}
```

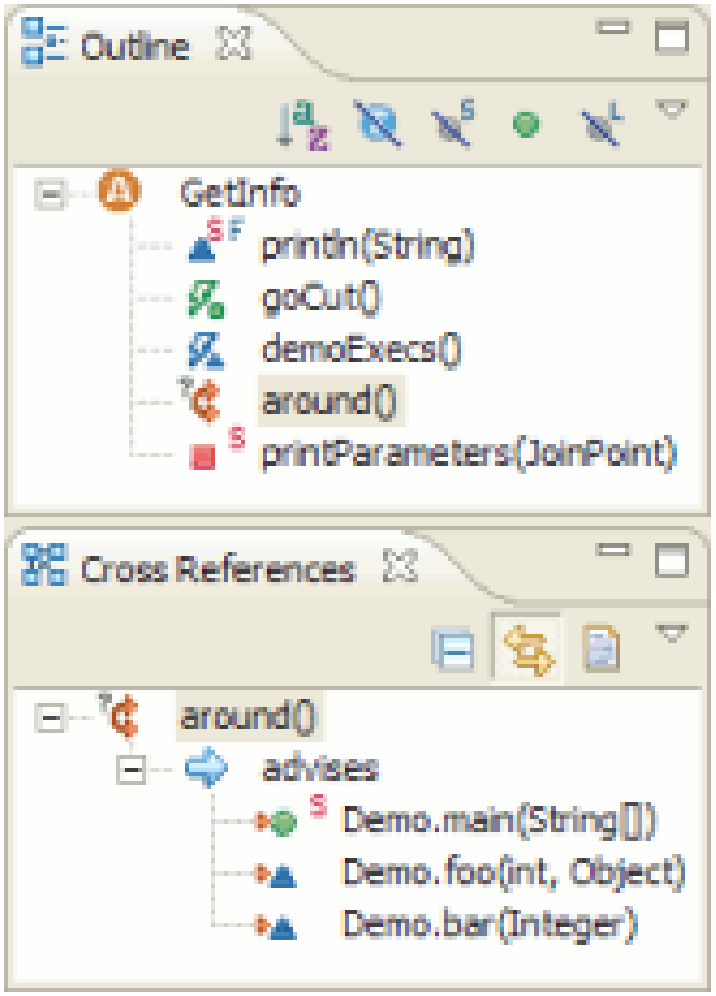
Typical Aspects IV

- Policy enforcement
  - Policy is externally implemented
  - Example: Autosave each 5th action

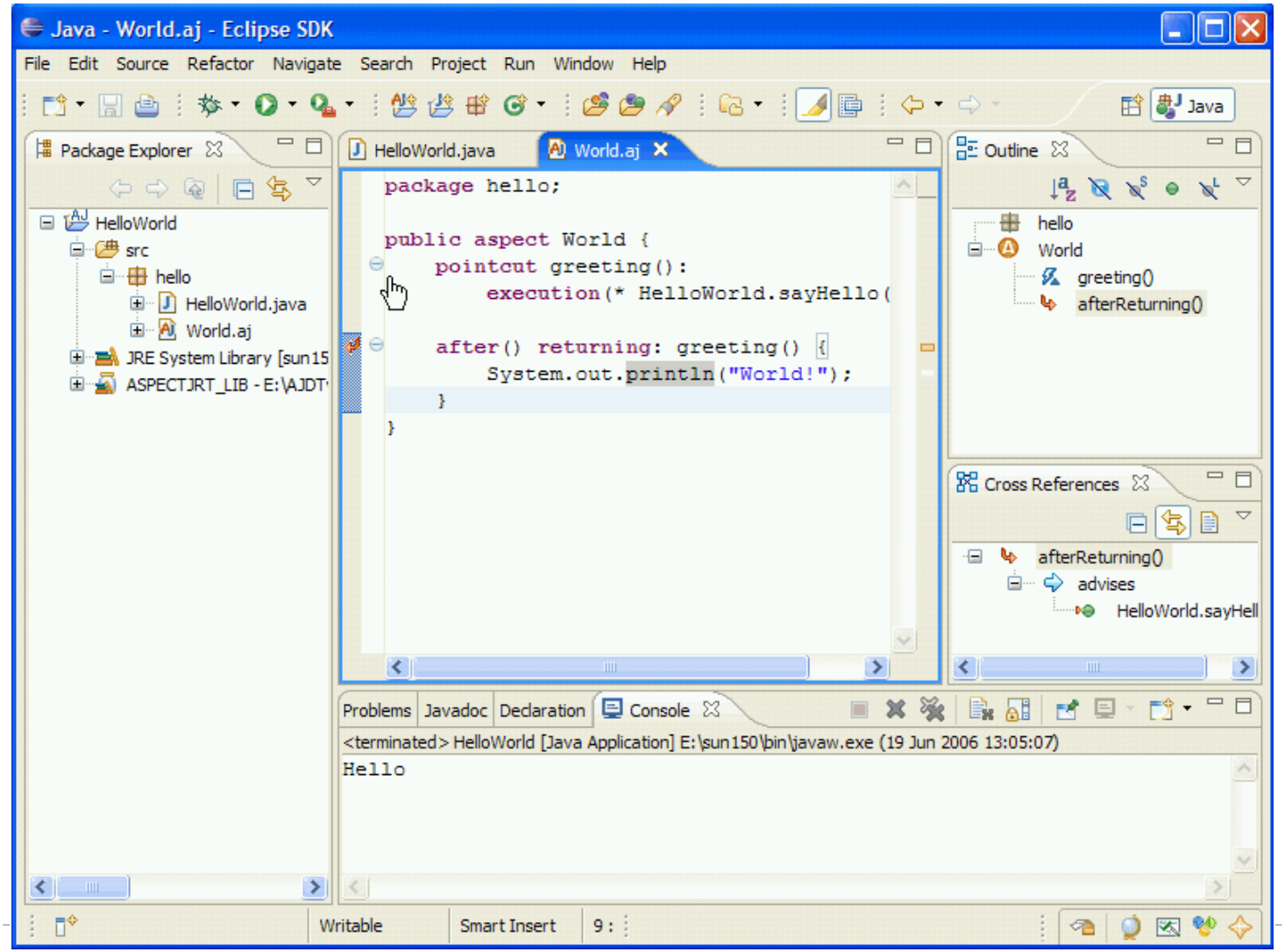
```
aspect Autosave {
    int count = 0;
    after(): call(* Command+.execute(..)) {
        count++;
    }
    after(): call(* Application.save())
         || call(* Application.autosave()) {
        count = 0;
    }
    before(): call (* Command+.execute(..)) {
        if (count > 4) Application.autosave();
    }
}
```

Development Evironment AJDT

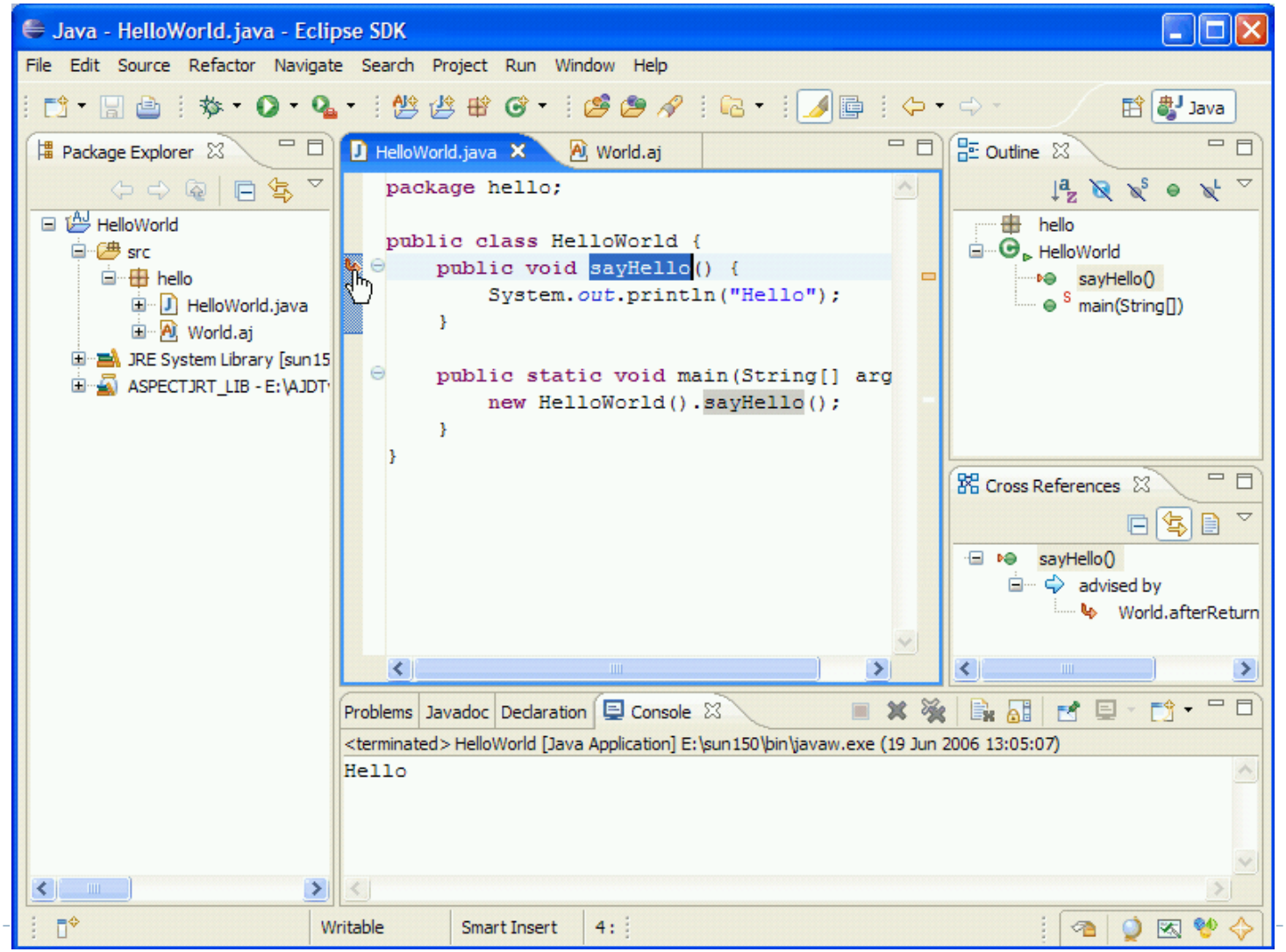
- Eclipse plugin for AspectJ development
  - Integrates aspects in Eclipse, like Java in JDT
  - Compiler and debugger integration
  - Syntax highlighting, outline
  - Links between aspects and extended places (shows where the source code is extended by aspects)



AJDT in Action



AJDT in Action



Discussion

Obliviousness Principle I

- Obliviousness means that the basic program does not need to know the aspects: “Implement Java as you did before and we will add aspects later.”
- Means that…
  - Classic OO-design is sufficient; code does not need to be prepared for aspects (“write your database as usual and we will add transaction support later”)
  - Basis developer do not need knowledge about aspects (few specialists are sufficient)
  - The AOP-language needs to be powerful and expressiveness

Obliviousness Principle II

- Obliviousness is next to quantification a central concept of AOP
- Controversy discussed, because...
  - ... Developers can change the basic code without even knowing the aspects that make use of it (“fragile pointcut problem”; no explicit interfaces)
  - ... it results in a partially bad design if the aspects are not taken into account when design the base program (they get later hacked into)
  - ... we typically need later complex language features, such as cflow or call && withincode to extended the non-prepared base code

Why Aspects Anyway?

- Enables cohesive implementation of crosscutting concerns
- Enables declarative to quantify many join points for applying homogeneous extensions at many points in the program
- Enables analyses of dynamic properties, such as control flow (cflow), which would need huge work arounds when using classic OOP features

Aspects for Features?

- Aspects are similar to collaborations
  - Can statically add code
  - Can extend methods
  - Can homogeneously extend code or based on the dynamic control flow
- Aspects can be turned on and off when compiling a program with a selection of them
  - Manuel in Eclipse: Right click on aspect and “Exclude from Buildpath”
  - Automates with build system
  - FeatureIDE in combination with AJDT

Aspects for Features?

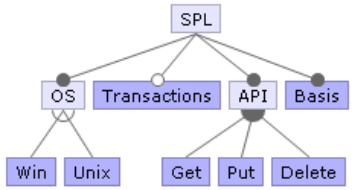
- A aspect per feature?
  - Aspects can become huge and hard to maintain
  - Aspects can introduce new classes only as static inner classes
- So: Comprising multiple aspects and classes in feature modules
- ➔ Mix of collaborations and aspects

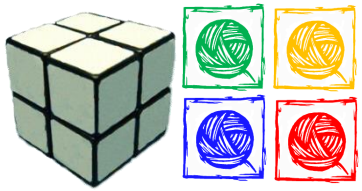
Product Lines With Aspects

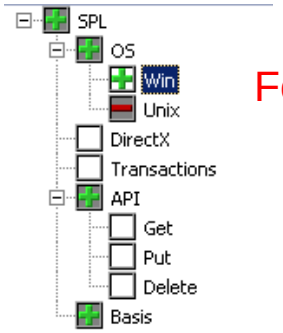
1:1 Mapping from features to aspects

Java program + aspects

Feature model



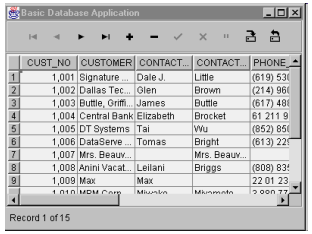




Feature selection

Feature selection as input

AspectJ-Compiler



Final program

Compiles with selected aspects

Problems

Lexical comparisons / fragile pointcut problem / complex syntax

Pointcuts use Lexical Comparison

- Pointcuts select join points based on name comparisons, although method names should be freely to chose
- Patterns use naming conventions, such as ”get*”, “draw*”, etc.

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

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

- What happens if we add a method “draw” to signal tie?

Fragile Pointcut Problem / Evolution Paradox

- If the base code changes, existing pointcuts could match to new join points or existing join points will not match anymore
- Chess example: A developer adds a method to declare a tie “void draw()”
- Such changes in the program behavior could happen without recognition; we cannot determine whether we found the “correct” pointcuts

draw = paint

draw = tie

Complex Syntax

- AspectJ is powerful and provide many expression possibilities with many language constructs
- The language gets complex such that even simple method extensions require substantial effort:

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);
}
```

Summary

- Aspect-oriented programming describes changes declaratively and quantified over an existing program
- AspectJ as AOP extension to Java; AJDT as development environment
  - Concepts: join points, inter type declarations, pointcuts, advice, …

Literature I

- eclipse.org/aspectj, eclipse.org/ajdt
- R. Laddad. AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications, 2003.
  [Practical book about 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
  [Trials for defining AOP]

Literature II

- Original publications:
  - 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
    [Fist paper on the problems of crosscutting concerns and proposing AOP as a 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
    [First introduction of AspectJ as a language]

Quiz

- What are pros and cons of the obliviousness principle?
- Are there extensions that cannot be expressed by AspectJ?
- Do aspects improve modularity (cohesion and coupling)?
- Mark all join points:

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