From objects to aspects

P. Cointe
OBASCO group

FMCO 2004, Leiden,
Friday 5th November

ECOLE DES MINES DE NANTES

INRIA

Understanding aspects

Quoting M. Wand :

"Aspect Oriented Programming is a promising recent technology for allowing adaptation of program units across modules boundaries"

3

Goal of this talk

Giving a VERY general introduction of the AOP intuitions

based on some AspectJ examples

- going beyond (meta)objects and components to get a better Separation of Concerns
- but of course aspects are not limited to OOP and OOD
- and there are more and more needs for formalization!!

4

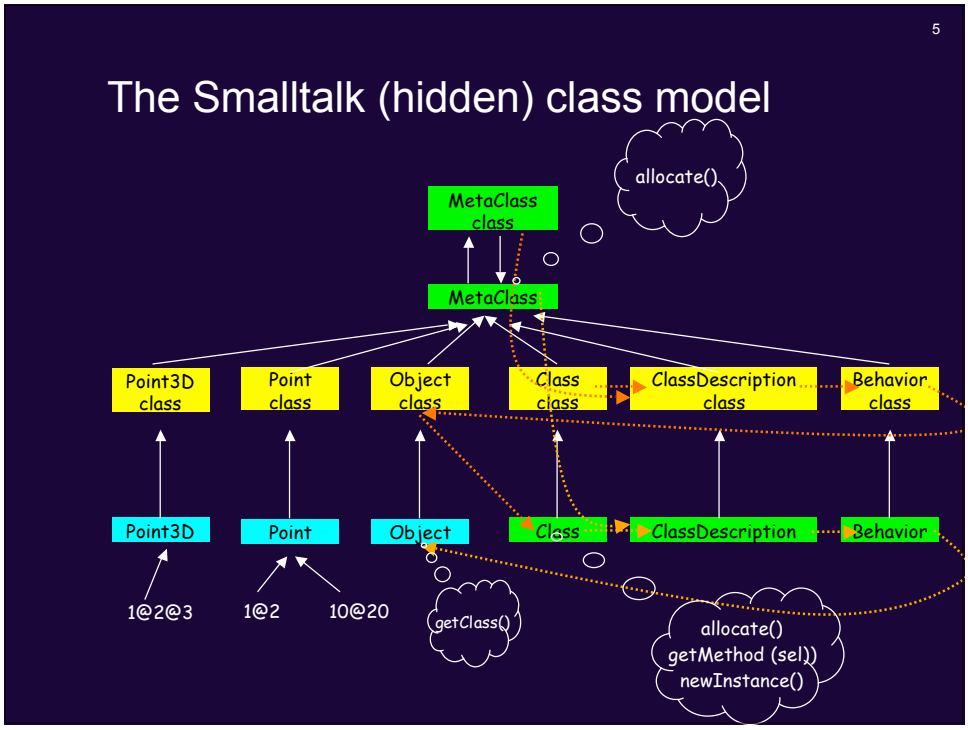
« Objects have failed » Dick Gabriel Opening Remarks

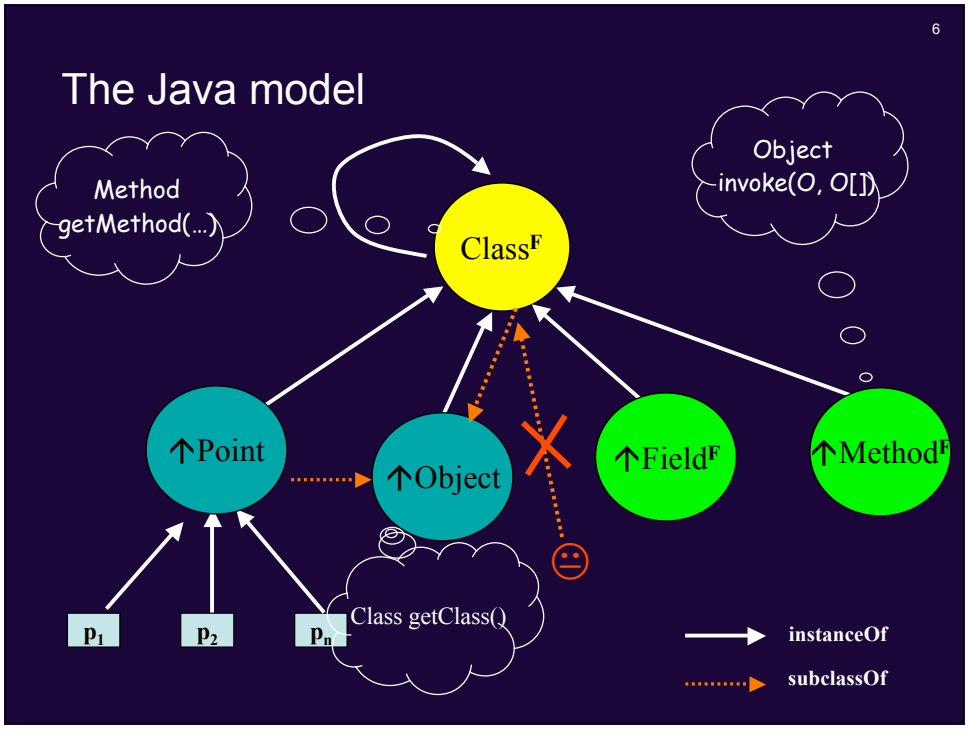
From Lisp to Smalltalk

« Objects, as envisioned by the designers of languages like Smalltalk and Actors were for modeling and building complex, dynamic words. Programming environment for language like Smalltalk were written in those languages and were extensible by developers. Because the philosophy of dynamic change was part of the post-Simula worldview, languages and environments of that era were highly dynamics. »

From Eiffel/C++ to Java

« But with C++ and Java, the dynamic thinking fostered by OOL was nearly fatally assaulted by the theology of static thinking inherited from our mathematical heritage and the assumptions built into our views of computing by C. Babbage whose factory-building worldview was dominated by omniscience and omnipotence ».


5
The Smalltalk (hidden) class model
allocate()
MetaClass class
MetaClass
Point3D class
Point class
Object class
Class class
ClassDescription class
Behavior class
Point3D
Point
Object
Class
ClassDescription
Behavior
1@2@3
1@2
10@20
getClass()
allocate()
getMethod (sel))
newInstance()


6
The Java model
Method getMethod(...)
Object invoke(O, O[])
Class^F
↑Point
↑Object
↑Field^F
↑Method^F
p1
p2
pn
Class getClass()
instanceOf
subclassOf

7

Some lessons from OOP

- classes
  - Inheritance is not THE solution for reusing cross-cutting “modules”
- metaobject protocols (metalevel architectures)
  - structural and behavioral reflection are expensive to use, difficult to understand and to secure
- design-patterns
  - no direct representation (tracability)
- frameworks
  - no real support for unanticipated extensions (adaptability)

8

Some “middleware” lessons

Quoting again M. Wand

- Applications typically need multiple services
  - logging, tracing, profiling, locking, displaying, transporting, authentication, security,
- These services don’t naturally fit in usual module boundaries (“crosscutting”):
  - These services must be called from many places (“scattering”),
  - An individual operation may need to refer to many services (“code tangling”)

9

More about classes

• Play to many roles and there if some confusion around those concerns :

- object generators,
- method dispatchers,
- parts of the inheritance graph.

• There is no intermediate granularity between a method and a class, no reification of a package, no reification of a design pattern. That lead to current experimentation such as:

- traits and mixin modules
- classboxes, open classes and Envy applications

10

☛ More about Reflection

Structural reflection

- object creation
  allocate o initialize
  - constructors
  - prototypes
  - metaclasses
- class modification
  - adding/removing field
  - changing the class hierachy

Behavioral reflection

- message sending
  lookup o apply
  - error inheritance
  - class based inheritance
  - prototype & delegation
  - encapsulators
  - wrappers & proxies
  - metaobjects

11

```
public Object receive (String selector, Object[] args) {
  Method mth = null; Object r = null; Class[] classes = null;  int l = 0;
  if (args != null) {
    l = Array.getLength(args);
    classes = new Class[l];
  }
  for (int i = 0; i < l; i++) { classes[i] = args[i].getClass();}
  try {
    mth = getClass().getMethod(selector, classes); // lookup
    // before poincut
    r = mth.invoke(this, args);                    // apply
    // after pointcut
    } catch (Exception e) {System.out.println(e);}
  return r;
}
```

12

```
Point p1 = new Point(3,4);
Point p2 = new Point(4,7);
Line l1 = new Line(p1,p2);
```

```
p1.getX();
l1.setP1(p1);
```

```
p1.receive("getX", null);
// perfom
p1.receive("get"+"X", null);

l1.receive("setP1", new Object[]{p1})
```

13

Objects have failed ?
Scalability and modularity

Considering the org.appache.tomcat application

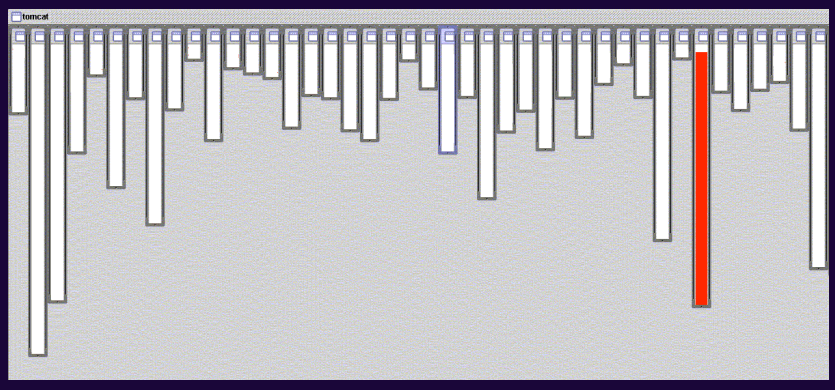
and the three next concerns:

- XML parsing
- URL pattern matching
- Logging

14

good modularity

XML parsing



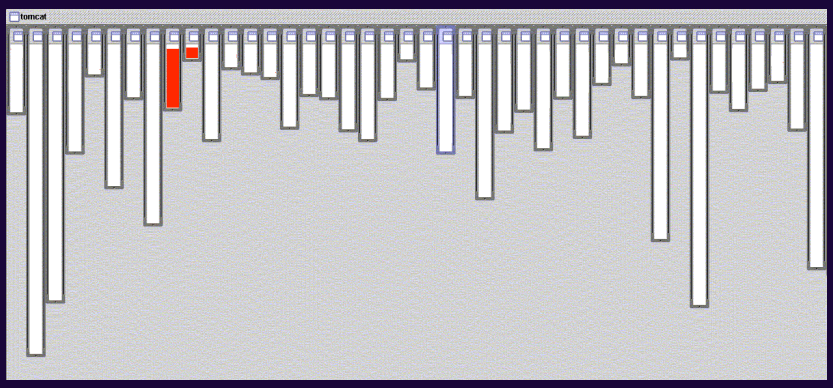
XML parsing in org.apache.tomcat

- red shows relevant lines of code
- nicely fits in one box

15

good modularity

URL pattern matching



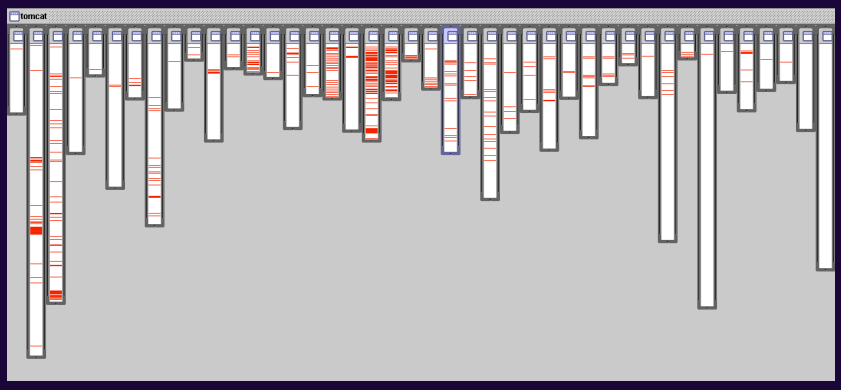
URL pattern matching in org.apache.tomcat

- red shows relevant lines of code
- nicely fits in two boxes (using inheritance)

16

modularity problems

logging is not modularized



logging in org.apache.tomcat

- red shows lines of code that handle logging
- not in just one place : a good example of scattering
- not even in a small number of places

17

Visiting the Leiden' NM of Antiquities

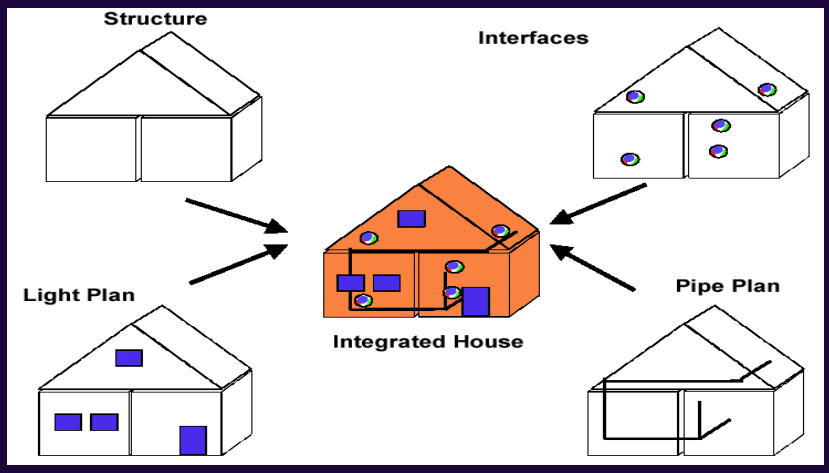
Meet again the Tyranny of the primary decomposition

"Egyptian depictions consist of strange combinations of side views, frontal views and views from above…..

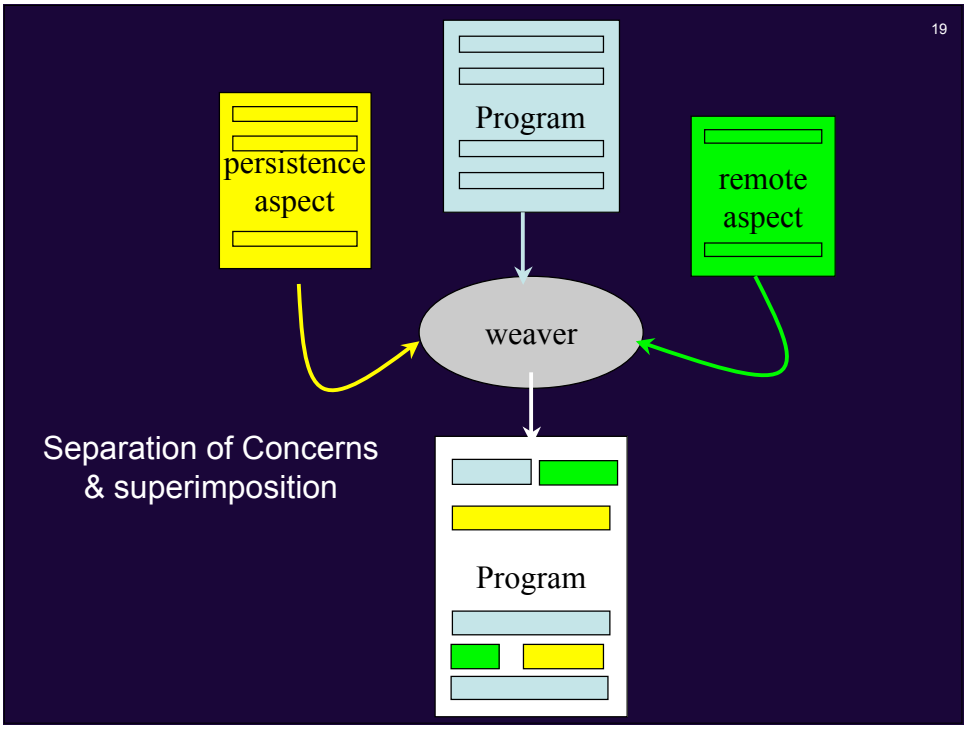
Every part of the representation should be as clear as possible."

18

Intuition (U. Aßmann)



19



Separation of Concerns & superimposition

20

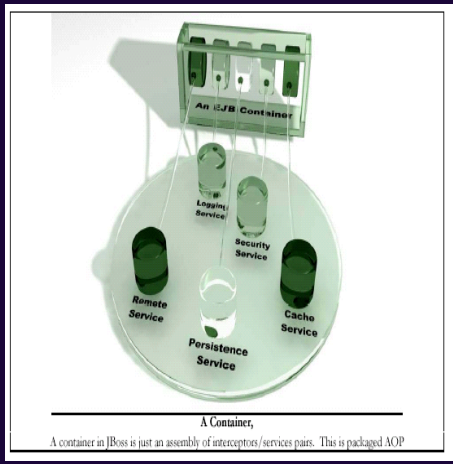
Two alternatives

① Expressing every aspect with an “aspectual domain specific language” (ASL).

② Using the same (general purpose aka Java) language for the aspects and the base program.

21

JBOSS = EJB + AOP

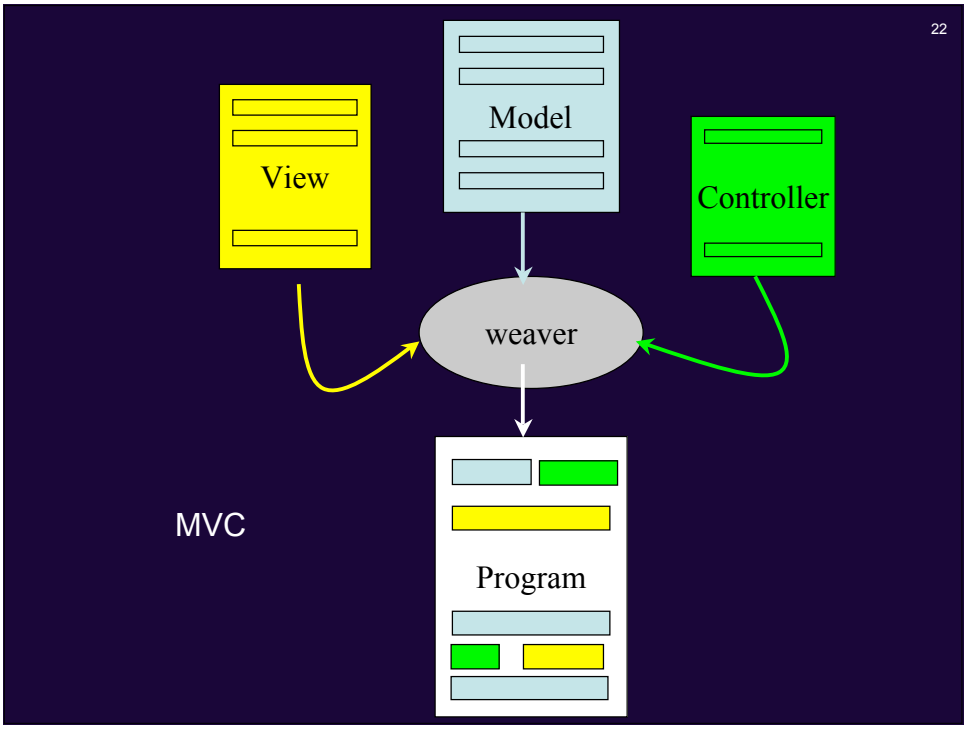


Associating technical services to EJB components and composing those services on demand :

- persistence
- security
- caching
- logging
- remote
- ….

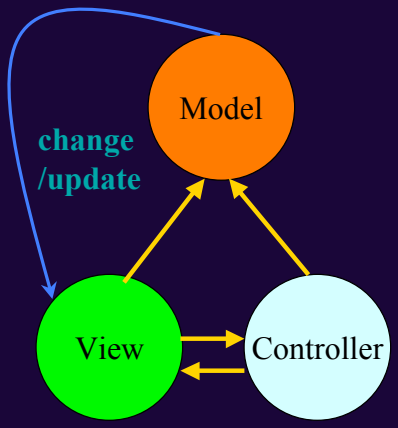
« We give you the hooks to simply specify the interceptor stack you want for a given component. »

22



23

Revisiting the Smalltalk MVC design pattern



A need to introduce a new mechanism (dependencies) to couple the code of the model, its view and its controller.

The dependencies mechanism prefigures event programming. The principle is to connect a source objet with a set of listeners. Those listeners will be notified as soon as the source will emit a new event.

24

About the Model, View et Controller

Le principe : séparer l'implémentation du modèle de l'aspect affichage (le système de vue) et de l'aspect contrôle (le système d 'interactions avec les périphériques)

À chaque hiérarchie de classe décrivant un modèle il faut donc associer une hiérarchie décrivant le système de vues et une autre décrivant la système de contrôleurs.

Nécessite l'ajout d 'un mécanisme supplémentaire pour coupler/tisser le code des trois objets du triptyque

Ce mécanisme dit des dépendances est le précurseur de celui réalisant la programmation par évènements. Il permet de mettre en relation un objet source avec un ensemble d 'objets abonnés en assurant que ces derniers seront notifiés à chaque émission d'un événement par la source

25

Smalltalk Counters Object vs Model?

```
Counter (value)
-----------------------------------------------
value
  ↑value
value: anInteger
  value ← anInteger.

incr: anInteger
  self value: value + anInteger
incr
  self incr: 1
raz
  self value: 0
```

```
Counter (value)
-----------------------------------------------
value
  ↑value
value: anInteger
  value ← anInteger.
  self changed
incr: anInteger
  self value: value + anInteger
incr
  self incr: 1
raz
  self value: 0
```

26

Smalltalk CounterViews

```
View (model controller)
```

```
View (model controller)
  CounterView()
-----------------------------------------------
defaultControllerClass
  ↑CounterController

update: dummy
  super displayView

displayView
  model value printString displayAt:
       insetDisplayBox center
```

27

Smalltalk CounterControllers

```
Controller (model view sensor)
  MouseMenuController (redButton..)
    CounterControler() {
                YMenu YMessage}
-------------------------------------------------
incr
  ↑model incr
raz
  ↑model raz
decr
  ↑model decr
```

```
initialize
  super initialize.
  self
    yellowButtonMenu: YMenu
    yellowButtonMessages: YMessage

isControlActive
  super isControlActive &
  sensor blueButtonPressed not
```

28

A guide tour of AspectJ

Thanks to G. Kiczales

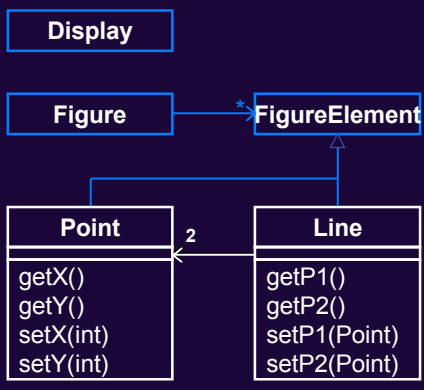
29

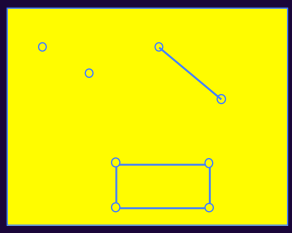
A guide tour of AspectJ

- C has “hello word”
- Lisp/Scheme have the factorial function
- Smalltalk has the Counter class
- Java has the Observer pattern
- AspectJ has the figure editor system

30

A simple Figures editor





When a FigureElement has moved then the Display must be redisplayed.

31

Java expression of a figure' move

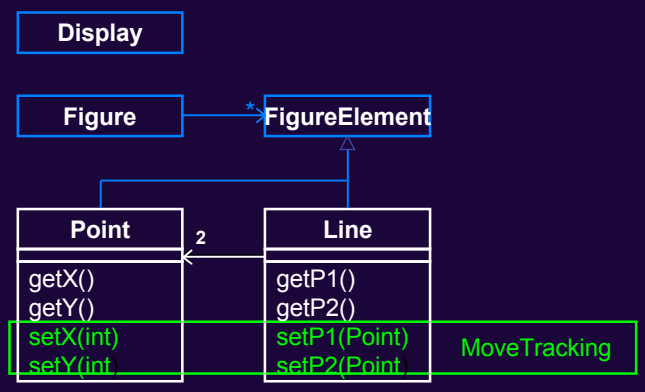
```
class Point implements FigureElement {
   private int _x, _y;

   public Point(int x, int y) {_x=x; _y=y;}

   public int getX() {return _x;}
   public int getY() {return _y;}
   public void setX(int x) {_x=x;}
   public void setX(int y) {_y=y;}
}
```

```
class Line  implements FigureElement {
   private int _p1, _p2;

   public Line(Point p1, Point p2) {
                        _p1=p1;_p2=p2;}

   public Point getP1() {return p1;}
   public Point getP2() {return p2;}
   public void setP1(Point p1) {_p1=p1;}
   public void setP2(int p2) {_p2=p2;}
}
```

```
Interface FigureElement{…}
```

32

The MoveTracking concern cuts across the class scope



33

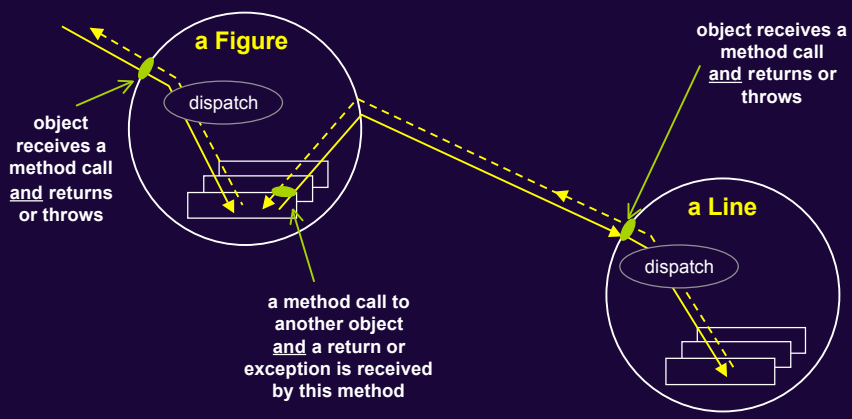
A guide tour of AspectJ

Dynamic AOP : the dynamic Join Point Model

34

join points

key points in dynamic call graph



35

Some definitions 1/2

Join points : principled points in the dynamic execution of the program (points at which an object receives a method call and points at which a field of an object is referenced).

Pointcut : a set of join points that optionally exposes some of the values in the execution context of those join points.

Advice (before/after/around) : a method-like mechanism used to declare that certain code should execute at each of the join points in a pointcut.

Inter-type declarations (Introductions) : declarations of members that cut across multiple classes or declarations of change in the inheritance relationship between classes.

36

Some definitions 2/2

Aspects : are modular units of crosscutting implementation. Aspects are defined by aspect declarations, which have a form similar to that of class declarations. Aspect declarations may include pointcut declarations, advice declarations, inter-type declarations as well as all other kinds of declaration permitted in class declarations.

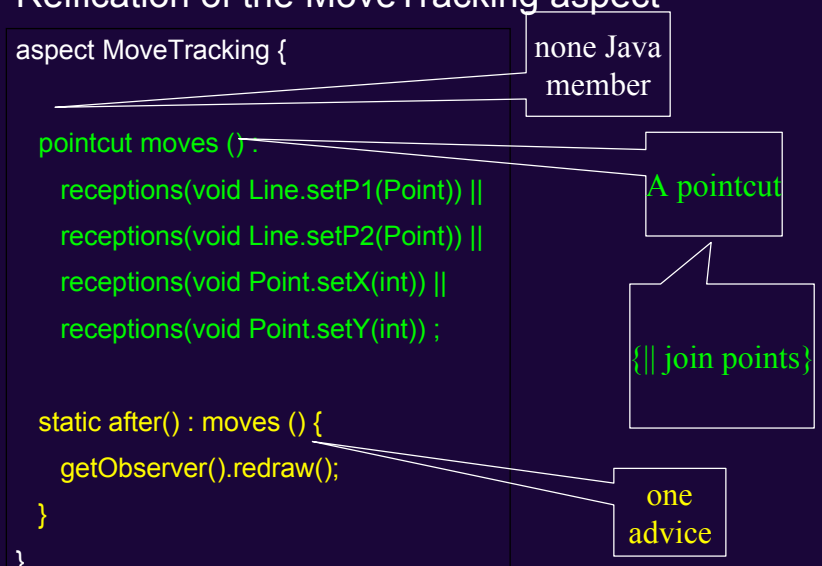
Notice that :

- Aspects are implemented as « singleton classes ».
- Aspects can be specialized.

37

Reification of the MoveTracking aspect

```
aspect MoveTracking {

  pointcut moves () :
    receptions(void Line.setP1(Point)) ||
    receptions(void Line.setP2(Point)) ||
    receptions(void Point.setX(int)) ||
    receptions(void Point.setY(int)) ;

  static after() : moves () {
    getObserver().redraw();
  }
}
```



38

Picking join points :

- method call
- method execution
- constructor call
- initializer execution
- static initializer execution
- constructor execution
- object preinitialization
- object initialization
- field reference
- exception handling execution

39

Pointcut designator

pointcut Name(arguments) : body

body := primitive pointcut || combination pointcuts

kind(Signature|TypePattern|Identifier)

kind := method call | method execution | …
TypePattern := wild card pattern * and (..)
Identifier := Java parameter name | pointcut name

40

Set-based semantics for pointcuts

- && operator = logical and = set intersection
- || operator = logical or = set union
- ! operator = logical negation = set difference

41

Advices

Are executed when a given pointcut is reached. The associated code can be told to run :

- before the actual method starts running
- after the actual method body has run
  - after returning
  - after throwing
  - plain after (after returning or throwing)
- around instead the actual method body
  - proceed

42

```
public class FP {
  static boolean pair(int n) {
    if (n == 0)
      return true;
    else
      return impair(n - 1);
  }
  static boolean impair(int n) {
    if (n == 0)
      return false
    else
      return pair(n - 1);
  }
```

```
  public static void main(String[] args) {
      S.o.pln("pair(4)-->" + pair(4));
      S.o.pln("pair(3)-->" + pair(3));
  }
```

```
pair(4)-->true          pair(3)-->false
```

43

```
public aspect Demo {

  pointcut callpair(): call(static boolean FP.pair(int)) ;
  pointcut callimpair(): call(static boolean FP.impair(int)) ;

  before(int n): callpair() && args(n) {
    System.out.println("Before callpair(" + n + ")");
  }
  after() returning (boolean b) : callpair() {
    System.out.println("After callpair(" + b + ")");
  }

  before(int n): callimpair() && args(n) {/*dito*/}
  after() returning (boolean b) : callimpair() {/*dito*/}
}
```

44

```
public class FP {
  static boolean pair(int n) {
    if (n == 0)
      return true;
    else
      return impair(n - 1);
  }
  static boolean impair(int n) {
    if (n == 0)
      return false
    else
      return pair(n - 1);
  }
```

```
public static void main(String[] args) {
    S.o.pln("pair(4)-->" + pair(4));
    S.o.pln("pair(3)-->" + pair(3));
}
```

```
Before callpair(4)
Before callimpair(3)
Before callpair(2)
Before callimpair(1)
Before callpair(0)
After callpair(true)
After callimpair(true)
After callpair(true)
After callimpair(true)
After callpair(true)
pair(4)-->true
```

```
Before callpair(3)
Before callimpair(2)
Before callpair(1)
Before callimpair(0)
After callimpair(false)
After callpair(false)
After callimpair(false)
After callpair(false)
pair(3)-->false
```

45

FP next

```
public class FP {
  public static int fact(int n){
    if (n == 0)
      return 1;
    else return n * fact(n - 1);
  }
public static int fib(int n){
    if (n <= 1)
      return 1;
    else
      return fib(n-1) + fib (n-2);
  }
  public static void main(String[] args) {
    fact(5);fact(4);fact(6);
    fib(3);

}
```

46

```
abstract aspect TraceProtocol {
  int counter = 0;
  abstract pointcut trace();

  before(int n): trace() && args(n){
    counter++;
    for (int j = 0; j <counter; j++) {System.out.print(" ");}
    System.out.println("-->f(" + n + ")");
  }
  after(int n) returning (int r): trace() && args(n){
    for (int j = 0; j < counter; j++) {System.out.print(" ");}
    System.out.println(r+"<--f(" + n +")");
    counter--;
  }
}
```

47

```
public aspect Trace extends TraceProtocol {

  pointcut trace() :
    call(static int FB.fact(int))  ||
    call(static int FP.fib(int))  ||
    call(void Counter.incr(int));

}
```

48

Execution traces (fact & fib)

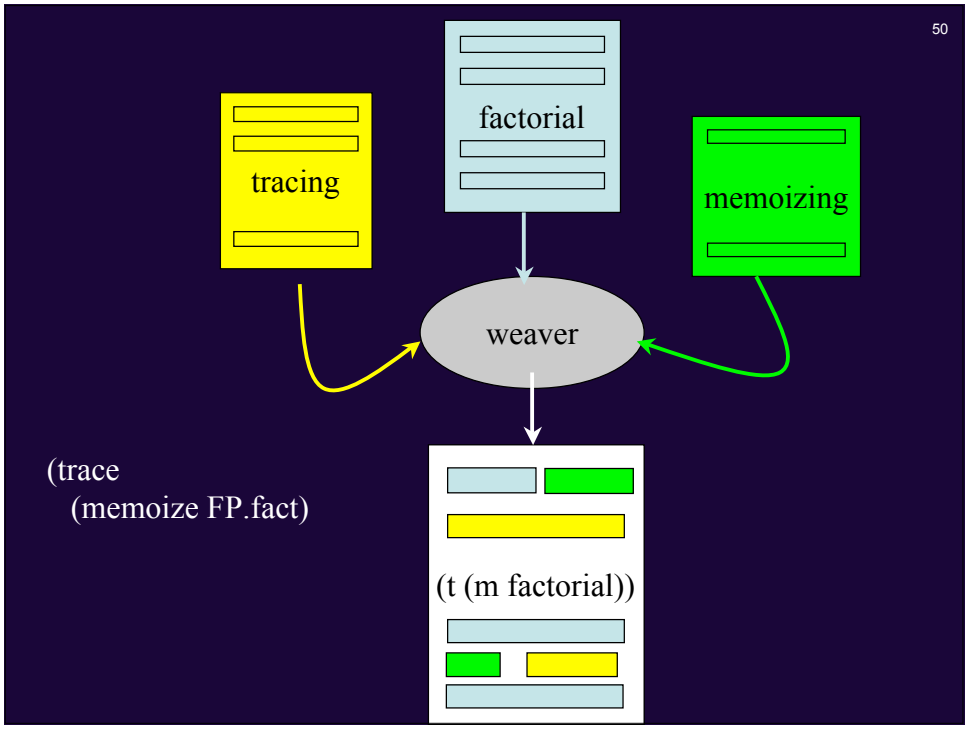
```
-->f(5)
 -->f(4)
  -->f(3)
   -->f(2)
    -->f(1)
     -->f(0)
     1<--f(0)
    1<--f(1)
   2<--f(2)
  6<--f(3)
 24<--f(4)
120<--f(5)
```

```
-->f(4)
 -->f(3)
  -->f(2)
   -->f(1)
   1<--f(1)
   -->f(0)
   1<--f(0)
  2<--f(2)
  -->f(1)
  1<--f(1)
 3<--f(3)
 -->f(2)
  -->f(1)
  1<--f(1)
  -->f(0)
  1<--f(0)
 2<--f(2)
5<--f(4)
fib(4)=5
```

49

```
public aspect Memoization {
  public static HashMap memo = new HashMap();
  pointcut callfact(): call(int FP.fact(int)) ;

  int around(int n): callfact() && args(n){
    Integer N = new Integer(n);
    if (memo.containsKey(N)) {
      return ((Integer)memo.get(N)).intValue();}
    else {
      int r = proceed(n) ;
      memo.put(N, new Integer(r));
      return r;
    }
  }
}
```

50



FP next

```
public class FP {
  public static int fact(int n){
    if (n == 0)
      return 1;
    else return n * fact(n - 1);
  }
public static int fib(int n){
    if (n <= 1)
      return 1;
    else
      return fib(n-1) + fib (n-2);
  }
  public static void main(String[] args) {
    fact(5);fact(4);fact(6);
    fib(3);

}
```

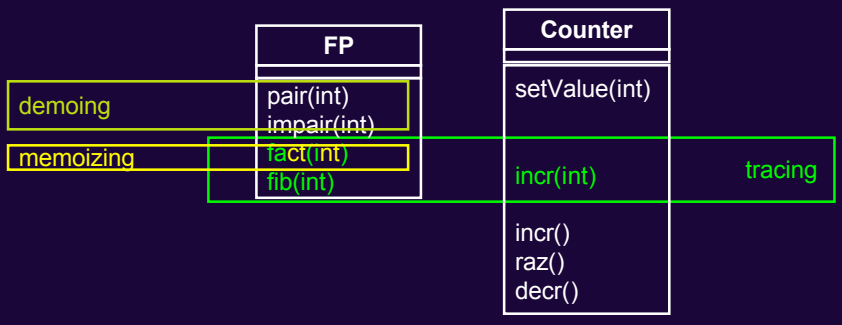

Execution traces (factorial)

```
-->f(5)
 -->f(4)
  -->f(3)
   -->f(2)
    -->f(1)
     -->f(0)
     1<--f(0)
    1<--f(1)
   2<--f(2)
  6<--f(3)
 24<--f(4)
120<--f(5)
```

```
-->f(4)
 24<--f(4)



-->f(6)
 -->f(5)
 120<--f(5)
720<--f(6)
```

53

Where are we?



54

A guide tour of AspectJ

Static AOP : the Inter-type declarations

- modeling traits
- implementing (some) design patterns

55

Dynamic vs static crosscutting

Intercessing with the program behavior

- users and primitive defined pointcuts
- cflow

Changing the program structure

- introducing field
- introducing method
- introducing constructor
- modifying class hierarchy

56

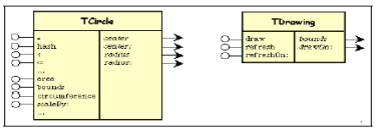


Fig. 3. The traits TDrawing and TCircle with provided methods in the left column and required methods in the right column.

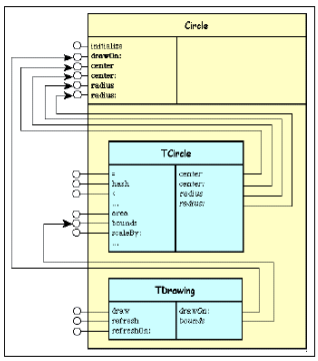
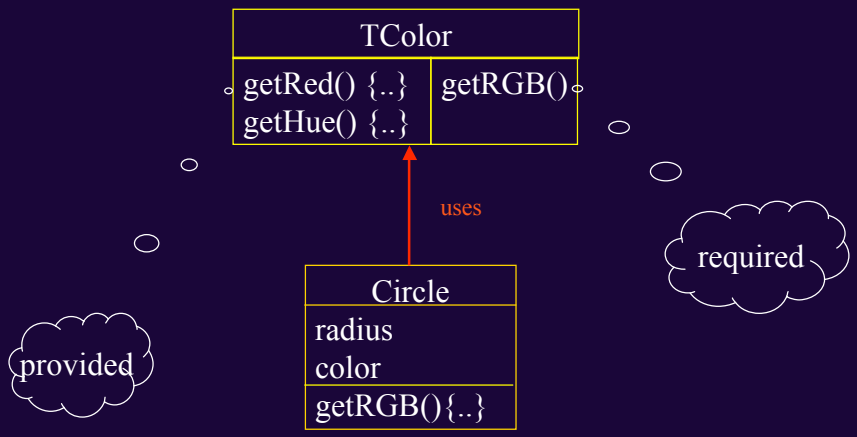
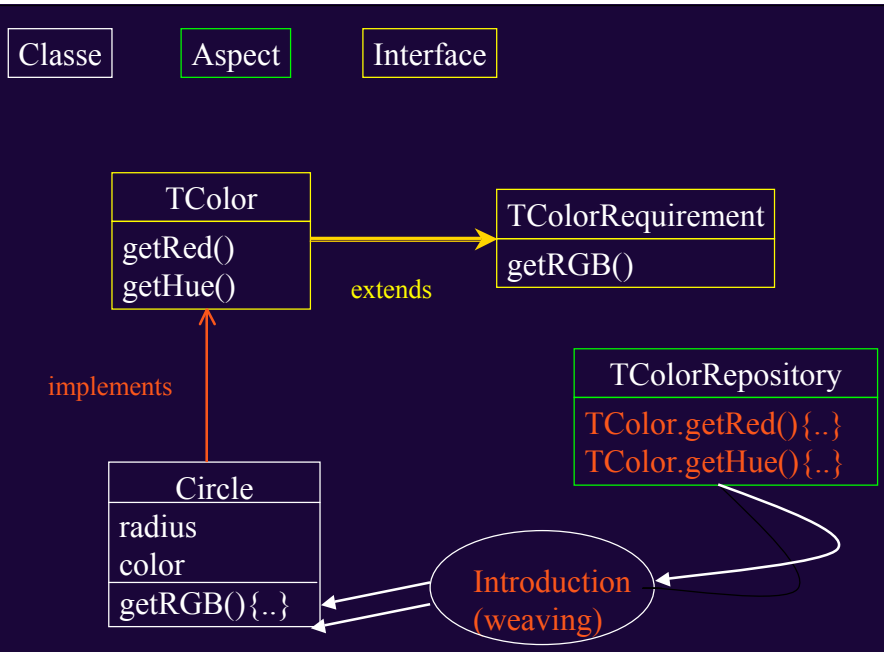


Fig. 4. The class Circle is composed from the traits TCircle and TDrawing. The requirement for TDrawing>>bounds is fulfilled by the trait TCircle. All the other requirements are fulfilled by accessor methods specified by the class.

Squeak traits





59

Trait = Java Interface + AspectJ Introduction

```
interface TColorRequirement {
   public java.awt.Color getRGB();
}

interface TColor extends TColorRequirement {
   public int getRed();
   pubic float getHue();
}
```

60

Using AspectJ 'introduction

```
aspect TColorRepository {
  declare parents: Circle implements TColor;
   public int TColor.getRed() {…};
   public float TColor.getHue() {…};
}
```

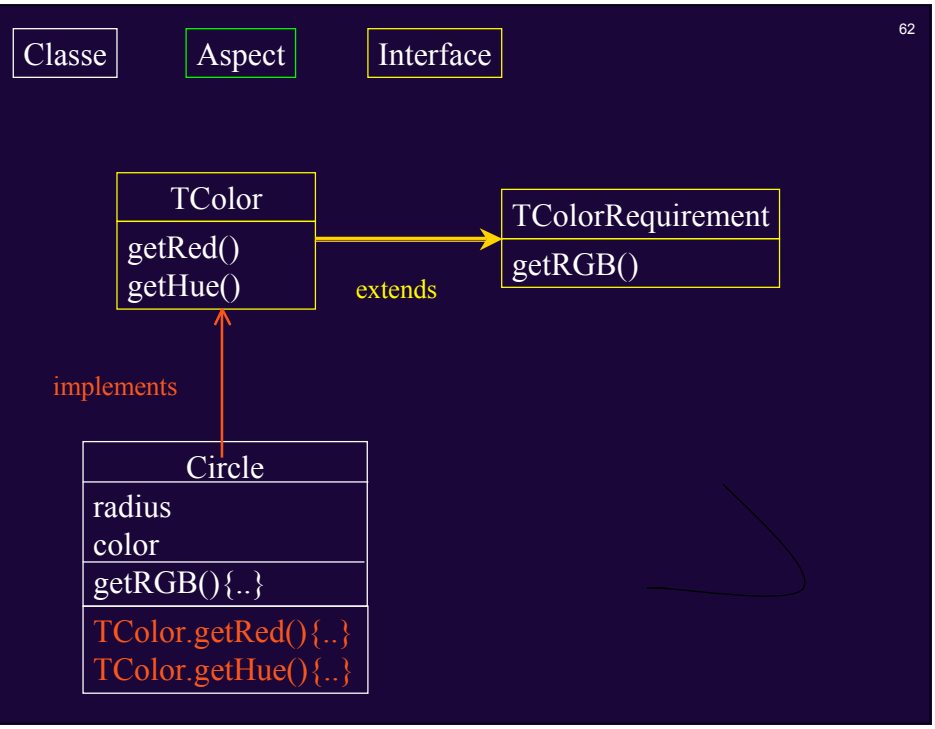
61

Static weaving

```
class Circle {
   private Color color;
   private int radius;

   public Color getRGB() {
      return color;
}
```

```
class Circle implements TColor {
   private Color color;
   private int radius;
   public int getRed() {…};
   public float getHue() {…};
   public Color getRGB() {
      return color;
   }
}
```

62

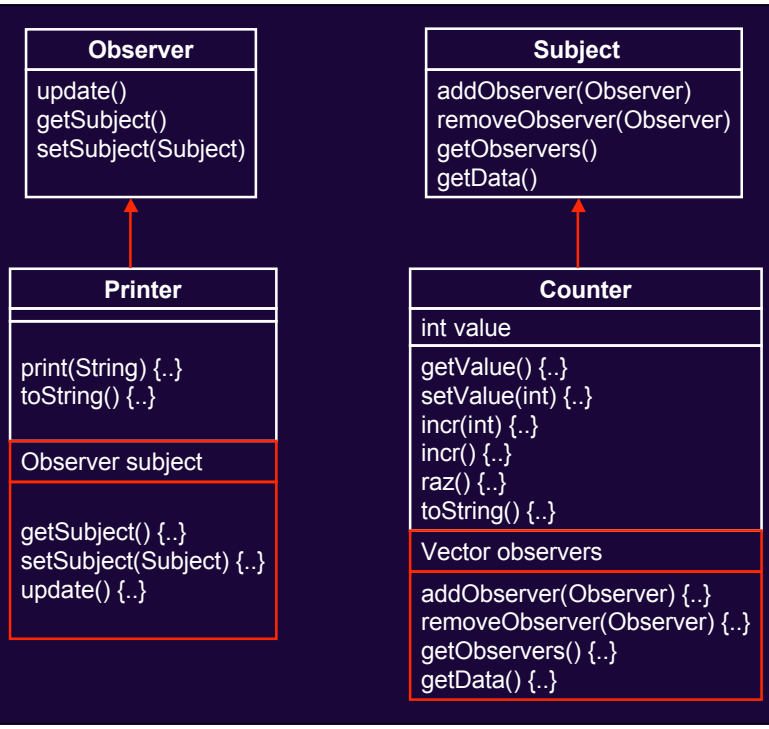


63

Java Counters

```
class Counter extends Object {
  private int value;
  public int getValue() {
    return value;
  }
  public void setValue(int nv) {
    value=nv;
  }
  public String toString() {
    return "@" + value;
  }
  public void incr() {
    this.incr(1);
  }
  public void incr(int delta) {
    this.setValue(value+delta);
  }
  public void raz() {
    this.setValue(0);
  }
}
```

64



65

// defining the abstract aspect monitoring state change (1/3)

```
abstract aspect PatternObserverProtocol {

   abstract pointcut stateChanges(Subject s);

   after(Subject s): stateChanges(s) {
      for (int i = 0; i < s.getObservers().size(); i++) {
         ((Observer)s.getObservers().elementAt(i)).update();
      }
   }
```

66

// extending classes implementing Subject (2/3)

```
 private Vector Subject.observers = new Vector();
   public void  Subject.addObserver(Observer obs) {
      observers.addElement(obs);
      obs.setSubject(this);
   }
   public void Subject.removeObserver(Observer obs) {
      observers.removeElement(obs);
      obs.setSubject(null);
   }
   public Vector Subject.getObservers() { return observers; }
```

67

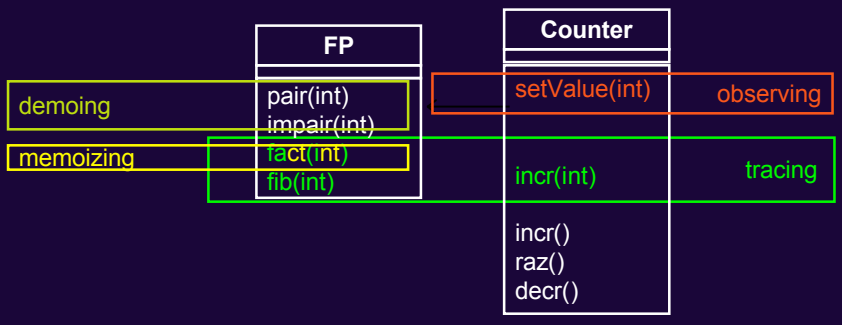
```
// extending classes implementing Observer (3/3)


    private Subject Observer.subject = null;
    public void Observer.setSubject(Subject s) { subject = s; }
    public Subject  Observer.getSubject() { return subject; }

}
```

68

```
aspect CounterObserver extends PatternOberverProtocol {
   declare parents: Counter implements Subject;
   public Object Counter.getData() { return this; }

   declare parents: Printer implements Observer;
   public void Printer.update() {
      this.print("update in " + this.getSubject());
   }

   pointcut stateChanges(Subject s) :
      target(s) &&
      call(public void Counter.setValue(int));
}
```

69

Where are we?



70

```
public static void main(String[] args) {
  Counter c1 = new Counter();
  Printer scribe = new Printer();
  c1.raz();
  c1.addObserver(scribe);
  c1.incr();
  c1.incr(6);
  c1.raz();
}
```

```
-->f(1)
[Printer]update occured in Counter@1
  -->f(6)
[Printer]update occured in Counter@7
[Printer]update occured in Counter@0
```

71

```
aspect FigureObserver extends PatternOberverProtocol {
  declare parents: Point implements Subject;
  declare parents: Line implements Subject;
  declare parents: Screen implements Observer;
  public Object Point.getData() { return this; }

  public void Screen.update() {
    this.display("coordinate change");
  }
  pointcut stateChanges(Subject s) :
    target(s) &&
    call(void Point.setX(int)) || call(void Point.setY(int) ||
    call(void Line.setP1(Point) || void Line.setP2(Point);
}
```

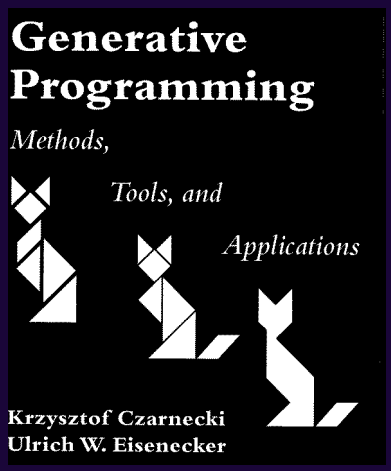
72

A more general context (page 1 of)

"The transition to automated manufacturing in software requires two steps.

First we need to move our focus from engineering single systems to engineering families of systems - this will allow us to come up with the "right" implementation components.

Second, we need to automate the assembly of the implementation components using generators."



73

Hot topics in Generative Programming

- Development for reuse
  - Reification and combination of cross-cutting models
- Modeling system families
  - Feature Modeling
  - Model Driven Architecture
- Mapping between problem space and solution space
  - From Reflection to Aspects
  - From Reflex to AspectJ
- Transformation/generation from domain specific languages to general purpose languages
  - Declarations & Annotations
  - Metaprogramming
- Extension/augmentation of (OO) programming languages by new abstraction mechanisms
  - Formalizing components
  - Understanding aspects
  - Introducing Aspect Specific Languages
- Improving IDE and assistants
  - Programmable program translators
- Introducing Aspect Specific Languages

74

Works in progress inside OBASCO

- Reflex
  - A model for partial behavioral reflection
  - A framework for Java Reflection based on Javassist
  - OOSPLA 03, ECOOP 04
  - A versatile kernel for AOP
  - Implementation of AspectJ and Traits in progress
  - http://www.emn.fr/x-info/reflex
- Arachne
  - Dynamic weaving of binary C code
  - http://www.emn.fr/x-info/arachne

75

Works in progress inside OBASCO

- EAOP (Event Based OOP)
  - dynamic composition of aspects (control/data flow)
  - static analyses and resolution of aspects interactions
  - static analyses for reusable aspects
  - LMO 03, GPCE 02, AOSD 2004
  - http://www.emn.fr/x-info/eaop
- AXL
  - a generalization of AspectJ context passing (cflow)
  - GPCE 04
  - http://www.emn.fr/x-info/axl

76

Interesting discussions at :

- G. Kiczales
  - http://eclipse.org/aspectj
- M. Wand
  - http:// www.ccs.neu.edu/home/wand/ICFP 2003
- R. Gabriel
  - http://www/dreamsongs/com/Essays.html
- J.-F. Perrot's group
  - http://www.lip6.fr/colloque-JFP/

77

Papers : AspectJ's friends

• Composition filter

L. Bergman. The composition filter objet model, PhD thesis, U. Twente, 1994.

• HyperJ

P. Tarr & H. Ossher & W Harrison & S.M Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns, ICSE 1999.

• Demeter

K. Lieberherr and D. Lorenz.Coupling Aspect-Oriented and Adaptive Progamming, AOSD 2004 Book Chapter, Addison Wesley

• DemeterJ = AOP + adaptive Programming

J. Ovlinger & M. Wand. A Language for specifying Recursive Traversal of Object Structures. OOPSLA 1999.

78

Papers : formal works

• Semantics

M. Wand & G. Kiczales & C. Dutchyn. Join Points in AOP. TOPLAS 2003.

• Aspect calculus

R. Jagadeesan, A. Jeffrey & J. Riely. A Calculus of Untyped Aspect-Programs, ECOOP 2003.

• Aspect calculus + type safe advices

D. Walker & S. Zdancewic & J. Ligatti. A theory of aspects. ICFP 2003.

• Grammars manipulation

Lämmel. PEPM'99

79

Important Dates

- Model View Controller [Goldberg 81]
- Metalevel Architectures [Smith 82, Maes 86]
- Meta Object Protocols [Kiczales 88]
- Composition filters [Aksit 92]
- Separation of Concerns [Hürsh & Lopes 95]
- Adaptative O.O.Software/Demeter [Lieberherr 96]
- Open Implementations [Kiczales & Paepacke]
- Functional vs. Behavioral levels [Garbinato & Guerraoui 94]
- Subject-Oriented Composition Rules [Ossher 95]
- Aspect Oriented Programming [Kiczales 96]
- AspectJ [Kiczales 01]

80

Dont be OO centric / Classification?

AOSD > ASL> AOP > OOP > FP > DSL

Defining AOP

Quoting G. Kiczales :

"AOP does for concerns that are naturally crosscutting what OOP does for concerns that are naturally hierarchical, it provides language mechanisms that explicitly capture crosscutting structure. This make it possible to program crosscutting concerns in a modular way, and thereby achieve the usual benefits of modularity: simpler code, that is easier to develop and maintain, and that has greater potential for reuse. We call such well-modularized crosscutting concerns aspects."