universitätbonn

ROOTS

Aspect Oriented Software Development

… Lecturer Names, Affiliation , …

The AOSD Course Team

Chapter 1-6

Daniel Speicher

Researcher

Office: A 109
Phone: 73-4315
E-Mail: dsp@cs.uni-bonn.de

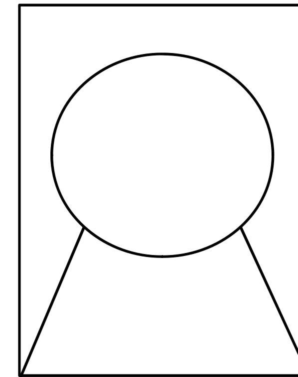
Chapter 7-12

Dr. Günter Kniesel

Lecturer

Office: A 107
Phone: 73-4511
E-Mail: gk@cs.uni-bonn.de

Exercise groups



Andreas Gasper

Student teaching assistant

Office: ---
Phone: ---
E-Mail: gasper@cs.uni-bonn.de

Contact

- To the teaching team
  - aosd-staff@iai.uni-bonn.de
- To the team and your colleagues
  - aosd-course@iai.uni-bonn.de
  - Please register yourself
- You are welcome to
  - e-mail us
  - visit us in our office
  - talk to us after the course

The boring stuff first

Website

Schedule

Exercises

Exams

Website

At http://sewiki.iai.uni-bonn.de/teaching/lectures/aosd/2009/ you find

- Course slides
  - updated each Monday before the lecture
- Exercise sheets
  - updated each Monday before the lecture
- Tips about tools and infrastructure
  - Eclipse, Subversion, AspectJ, …, mailing list registration, …
- Schedule (Google calendar)
  - Lectures, exercises, exams
  - No lecture / exercises on June 1 – 6 (Whitsun) and July 20 (Exam week)
- References
- News

Regular Schedule

- Monday, 11.00-12.30: AOSD lecture (room H2)
  - so that you don‘t have to queue at the mensa
- Monday, 13.00-14.30: ALP lecture (room A6c)
  - so that we could fit both lectures and four exercise groups on the same day
- Practical exercises (in terminal pool A106)
  - Monday
    - ⇨ 14:45: Group 1
    - ⇨ 15:30: Group 2
    - ⇨ 16:15: Break
    - ⇨ 16:30: Group 3
    - ⇨ 17:15: Group 4
  - Wednesday
    - ⇨ 15:00: Group 5
    - ⇨ 15:45: Group 6
- Time budget
  - 1 hour reading / learning + 1 hour practical exercises

Exercise groups

- Small groups
  - 1 tutor , 3-4 students, 45 minutes
- Emphasis on practical skills
  - lots of practical exercises
  - learn to use AspectJ, LogicAJ and other tools
  - electronic result submission via „Subversion“ repository
  - presentation of results and discussion with tutor in front of your computer
- Teamwork is essential
  - you should solve exercises collaboratively
  - you must be able to present any result of the team

From Exercise Groups to Exams

- To be admitted to the exam
  - you must achieve at least 50% of the points for the exercises
  - you may fail to attend at most two of the exercise group meetings

- Explaining ideas is a important as programming
  - Your tutor will ask each group member questions about any exercise
  - You only get points for the solutions that you are able to explain yourself sensibly

- Regularly doing your exercises and actively participating in the group discussions is the best preparation for your oral exams!
  - You must learn to talk fluently about any topic of the course
  - You must practice using the right terms the right way!
  - You must be able to write down short bits of correct code.

Exams

- Oral exams
  - 1 student, 1 lecturer, 40 minutes

- Aims of the exam: Verify that
  - you can discuss confidently any topic of the course
  - you can use the right terms the right way
  - you can asses problems and identify needs / opportunities for using AOSD
  - you can design a sensible AO solution of a problem
  - you can write down short bits of correct code.

- Exam slots
  - Wed. July 22, 09:00-13:30, every 45 minutes
  - Wed. July 22, 14:00-18:30, every 45 minutes
  - Thurs. July 23, 09:00-13:30, every 45 minutes
  - more if need be…

Learning

Tell me and I'll forget.

Show me and I'll remember.

Let me do it and I'll know.

Confucius

Chinese Philosopher

551 – 479 B.C.

Chapter 1. Introduction by Example

The Problem: “Crosscutting Concerns” (CCC)

Elements of Aspect Languages

Why Aspects are a Solution

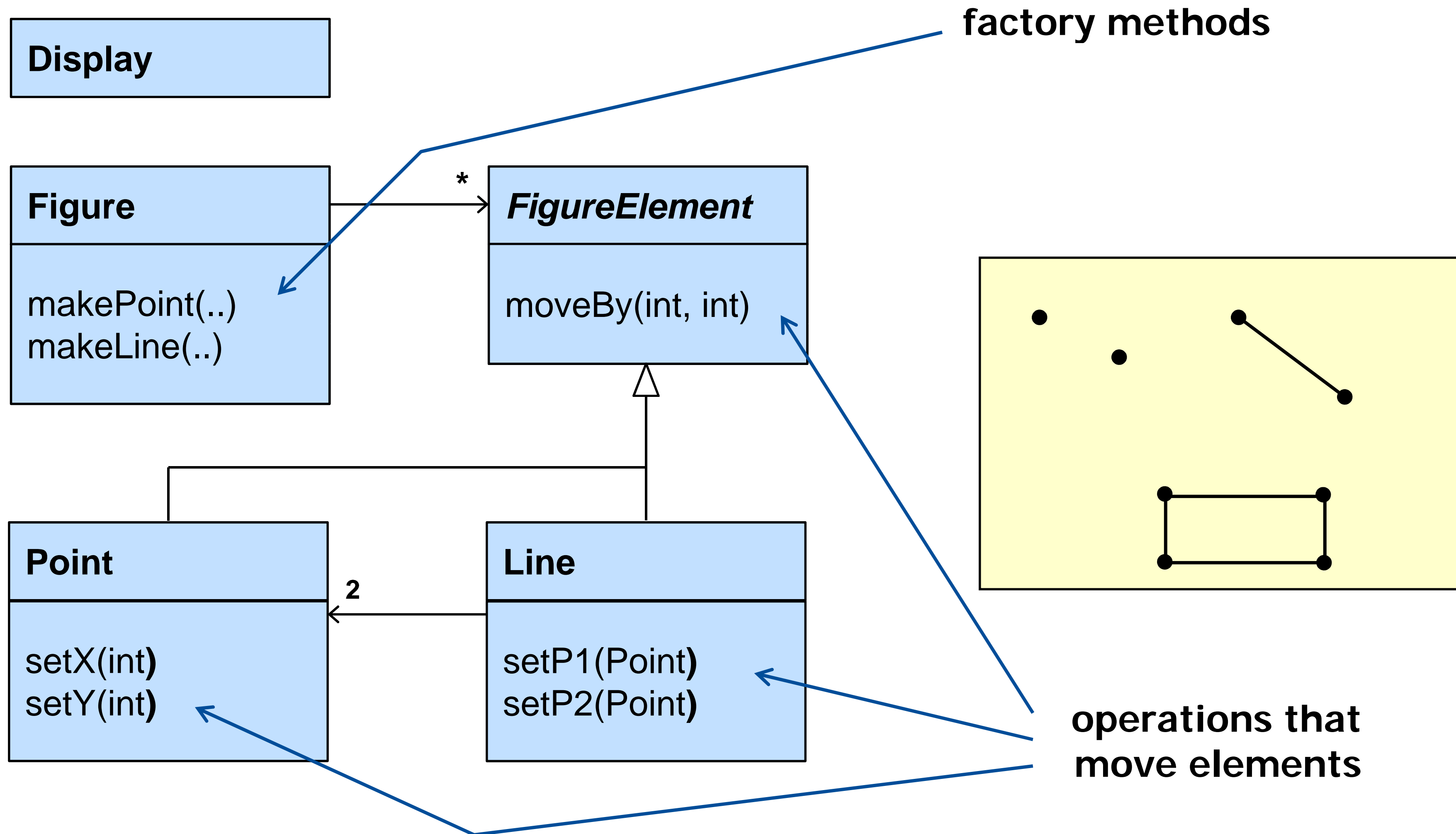
Schedule for Today

- Introduction by example
  - The evolution of a simple figure editor
    - ⇨ Scattered concerns are difficult to evolve
  - The evolution of an aspect of the simple figure editor
    - ⇨ Separated concerns are easy to evolve
  - A typical piece of code from a business application
    - ⇨ Tangled code is hard to understand
- Separation of crosscutting concerns
  - Separation of concern
  - Scattering and Tangling as deviations from the 1-1 vision
  - The nature of crosscutting concerns: “Everywhere where … do …”
  - Quantifications allows for obliviousness:
    Joinpoint Model, Pointcuts, Advice, [Intertype Declarations,] Aspects, Weaving
  - The expected benefits: Ease of change and evolution, Configurability, Reuse

Introduction by example

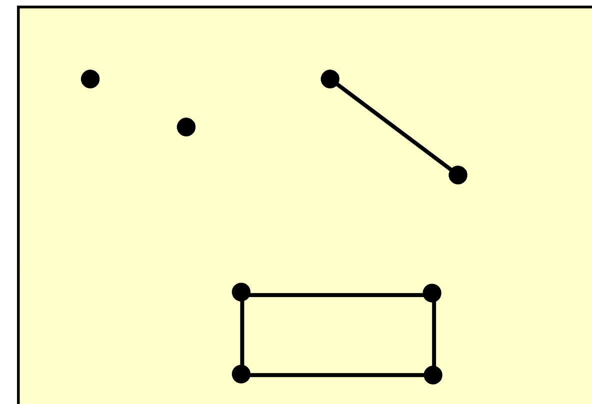
- The evolution of a simple figure editor
  - ⇨ Scattered concerns are difficult to evolve
- The evolution of an aspect of the simple figure editor
  - ⇨ Separated concerns are easy to evolve
- A typical piece of code from a business application
  - ⇨ Tangled code is hard to understand

A simple figure editor (Class diagram)



A simple figure editor (Java code)

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;

  }
  void setP2(Point p2) {
    this.p2 = p2;

  }
}

class Point {
   private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;

  }
  void setY(int y) {
    this.y = y;

  }
}
```

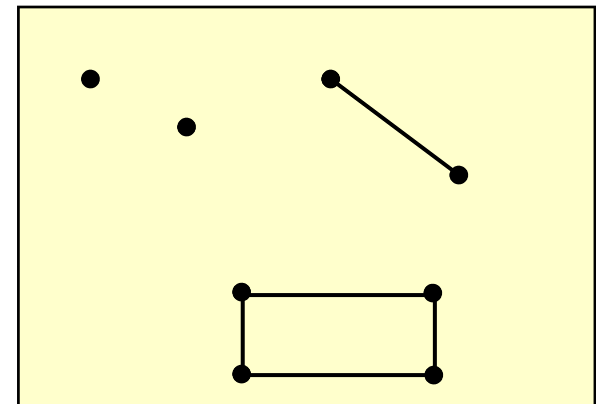


The evolution of the simple figure editor

Scattered concerns are difficult to evolve

Display updating

- collection of figure elements
  - that move periodically
  - must refresh the display as needed
  - complex collection
  - asynchronous events



The Java code again

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;

  }
  void setP2(Point p2) {
    this.p2 = p2;

  }
}

class Point {
   private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;

  }
  void setY(int y) {
    this.y = y;

  }
}
```

Now we will try to extend this code.

Adding display updating

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update();
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update();
  }
}

class Point {
   private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;

  }
  void setY(int y) {
    this.y = y;

  }
}
```

Task:

Update the display when a Line changes.

Adding more display updating

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update();
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update();
  }
}

class Point {
   private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    Display.update();
  }
  void setY(int y) {
    this.y = y;
    Display.update();
  }
}
```

Task:

Update the display when a Point changes.

Telling the display what changed

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update(this);
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update(this);
  }
}

class Point {
   private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    Display.update(this);
  }
  void setY(int y) {
    this.y = y;
    Display.update(this);
  }
}
```

Task:

Pass the changed object, so that the update routine can determine what part of the display needs to be updated

Problems with this approach

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update(this);
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update(this);
  }
}

class Point {
   private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    Display.update(this);
  }
  void setY(int y) {
    this.y = y;
    Display.update(this);
  }
}
```

- update calls are tangled in the code
- No location for “display updating”
- “what is going on” is less explicit
- Evolution is cumbersome
  - It involves changes in all classes
  - You have to track and change all callers

Towards a solution

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update(this);
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update(this);
  }
}

class Point {
   private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    Display.update(this);
  }
  void setY(int y) {
    this.y = y;
    Display.update(this);
  }
}
```

Each line belongs to the same „aspect“

universitätbonn
ROOTS

The evolution of an aspect of the simple figure editor

Separated concerns are easy to evolve

Adding display updating (AspectJ)

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}

class Point {
   private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

Task:

Update the display when a Line changes.

```
aspect DisplayUpdating {

  pointcut move():
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)) ||
  after() returning: move() {
    Display.update();
  }
}
```

Adding more display updating (AspectJ)

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}

class Point {
   private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

Task:

Update the display when a Point changes.

```
aspect DisplayUpdating {

  pointcut move():
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)) ||
    call(void Point.setX(int))   ||
    call(void Point.setY(int));

  after() returning: move() {
    Display.update();
  }
}
```

Telling the display what changed (AspectJ)

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}

class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

Task:

Pass the changed object, so that the update routine can determine what part of the display needs to be updated

```
aspect DisplayUpdating {

  pointcut move(FigureElement fe):
    target(fe) &&
    (call(void Line.setP1(Point))          ||
     call(void Line.setP2(Point))          ||
     call(void Point.setX(int))            ||
     call(void Point.setY(int)));


  after(FigureElement fe) returning: move(fe) {
    Display.update(fe);
  }
}
```

Extending the aspect

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}

class Point {
   private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

Task:

The moveBy method on the superclass should update the display too.

```
aspect DisplayUpdating {

  pointcut move(FigureElement fe):
    target(fe) &&
    (call(void FigureElement.moveBy(int, int) ||
     call(void Line.setP1(Point))             ||
     call(void Line.setP2(Point))             ||
     call(void Point.setX(int))               ||
     call(void Point.setY(int)));


  after(FigureElement fe) returning: move(fe) {
    Display.update(fe);
  }
}
```

Advantages

```
aspect DisplayUpdating {

  pointcut move(FigureElement fe):
    target(fe) &&
    (call(void FigureElement.moveBy(int, int) ||
     call(void Line.setP1(Point))             ||
     call(void Line.setP2(Point))             ||
     call(void Point.setX(int))               ||
     call(void Point.setY(int)));


  after(FigureElement fe) returning: move(fe) {
    Display.update(fe);
  }
}
```

- clearly arranged display updating module
  - all changes in single aspect
  - evolution is modular

A typical piece of code from a business application

Tangled code is hard to understand

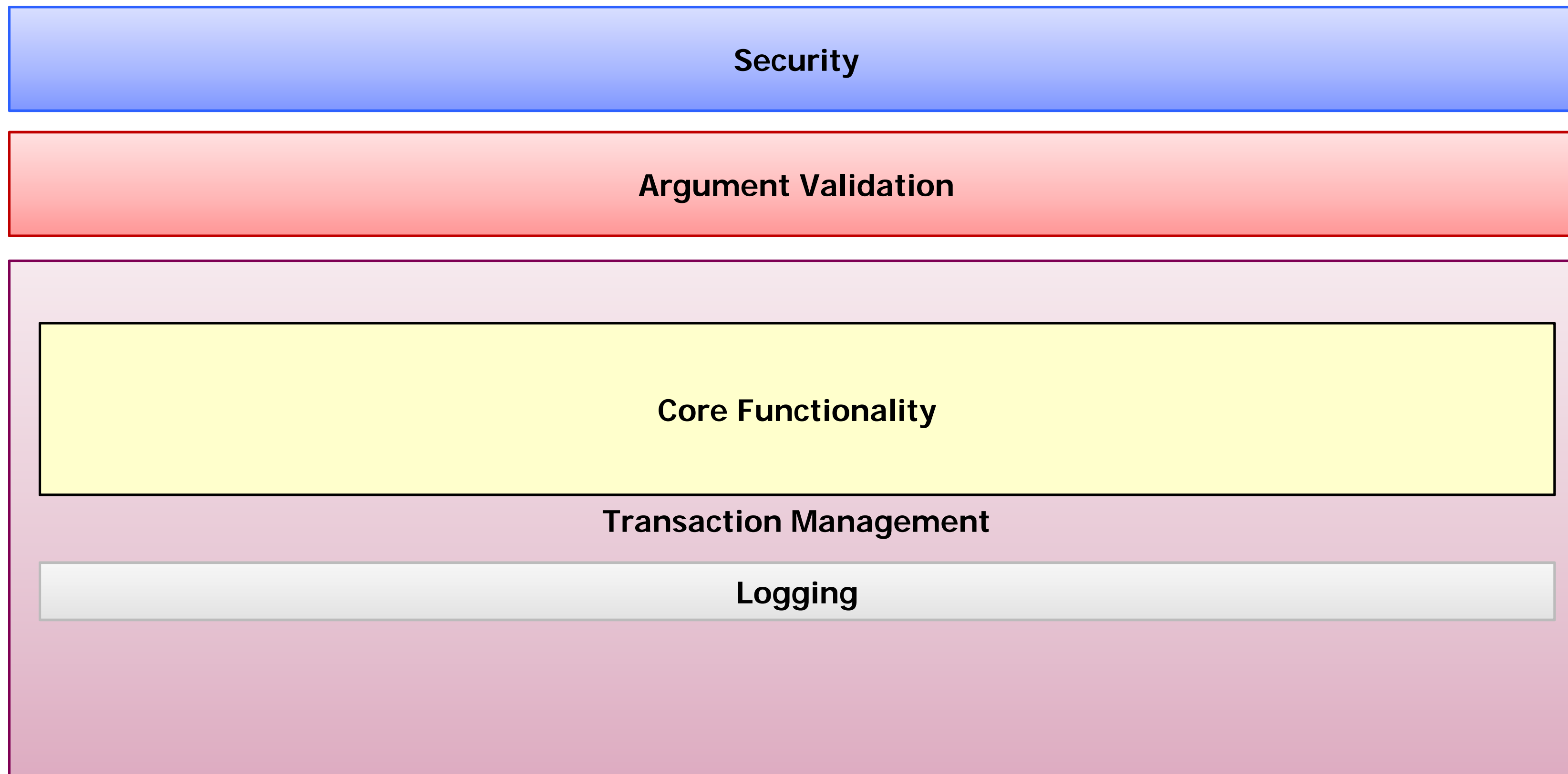
A typical piece of code from a business application.

```
void transfer(Account fromAccount, Account toAccount, int amount) throws Exception {
        if (!getCurrentUser().canPerform(OP_TRANSFER)) {
                throw new SecurityException();
        }

        if (amount < 0) {
                throw new NegativeTransferException();
        }

        Transaction tx = database.newTransaction();
        try {
                if (fromAccount.getBalance() < amount) {
                        throw new InsufficientFundsException();
                }
                fromAccount.withdraw(amount);
                toAccount.deposit(amount);

                tx.commit();

                systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount, amount);
        }
        catch(Exception e) {
                tx.rollback();
                throw e;
        }
}
```

A typical piece of code from a business application: The core functionality.

```
// transfer method for a simple banking application
void transfer(Account fromAccount, Account toAccount, int amount) {
        if (fromAccount.getBalance() < amount) {
                throw new InsufficientFundsException();
        }
        fromAccount.withdraw(amount);
        toAccount.deposit(amount);
}
```

The core functionality … and more.



The core functionality … and more: Code tangled with different concerns.

```
void transfer(Account fromAccount, Account toAccount, int amount) throws Exception {
        if (!getCurrentUser().canPerform(OP_TRANSFER)) {
                throw new SecurityException();
        }

        if (amount < 0) {
                throw new NegativeTransferException();
        }

        Transaction tx = database.newTransaction();
        try {
                if (fromAccount.getBalance() < amount) {
                        throw new InsufficientFundsException();
                }
                fromAccount.withdraw(amount);
                toAccount.deposit(amount);

                tx.commit();

                systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount, amount);
        }
        catch(Exception e) {
                tx.rollback();
                throw e;
        }
}
```

Separation of crosscutting concerns

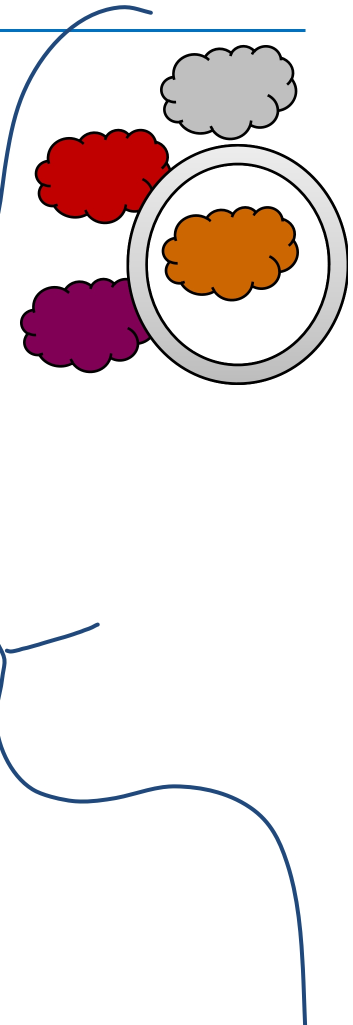
- Separation of concern
- Scattering and Tangling as deviations from the 1-1 vision
- The nature of crosscutting concerns:
  “Everywhere, where … do …”
- Quantifications allows for obliviousness:
  Joinpoint Model, Pointcuts, Advice, [Intertype Declarations,] Aspects, Weaving
- The expected benefits:
  Ease of change and evolution, Configurability, Reuse

Separation of Concerns

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects.

We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day […] But nothing is gained – on the contrary – by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns" […]

[Edsger W. Dijkstra, "On the role of scientific thought", 1974]

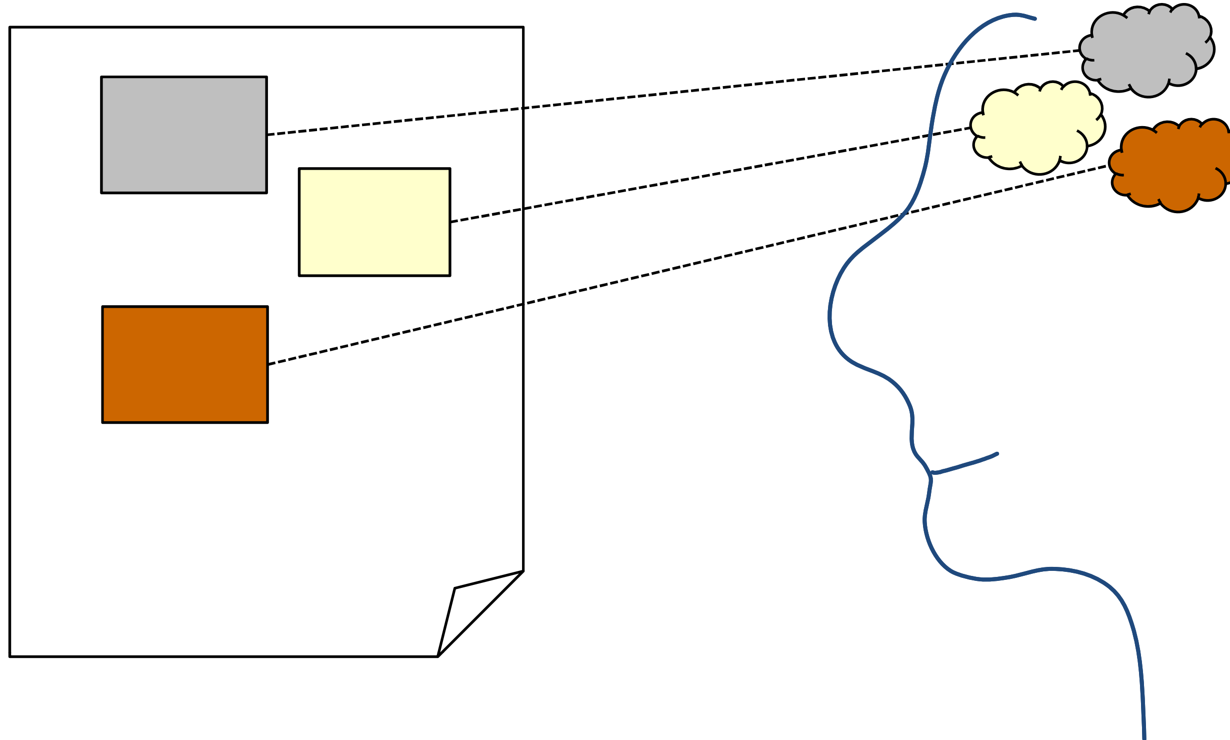


Separation of Concerns, “Definitions”

- Concern:
  “Something the developer needs to care about” (e.g. functionality, requirement,..)

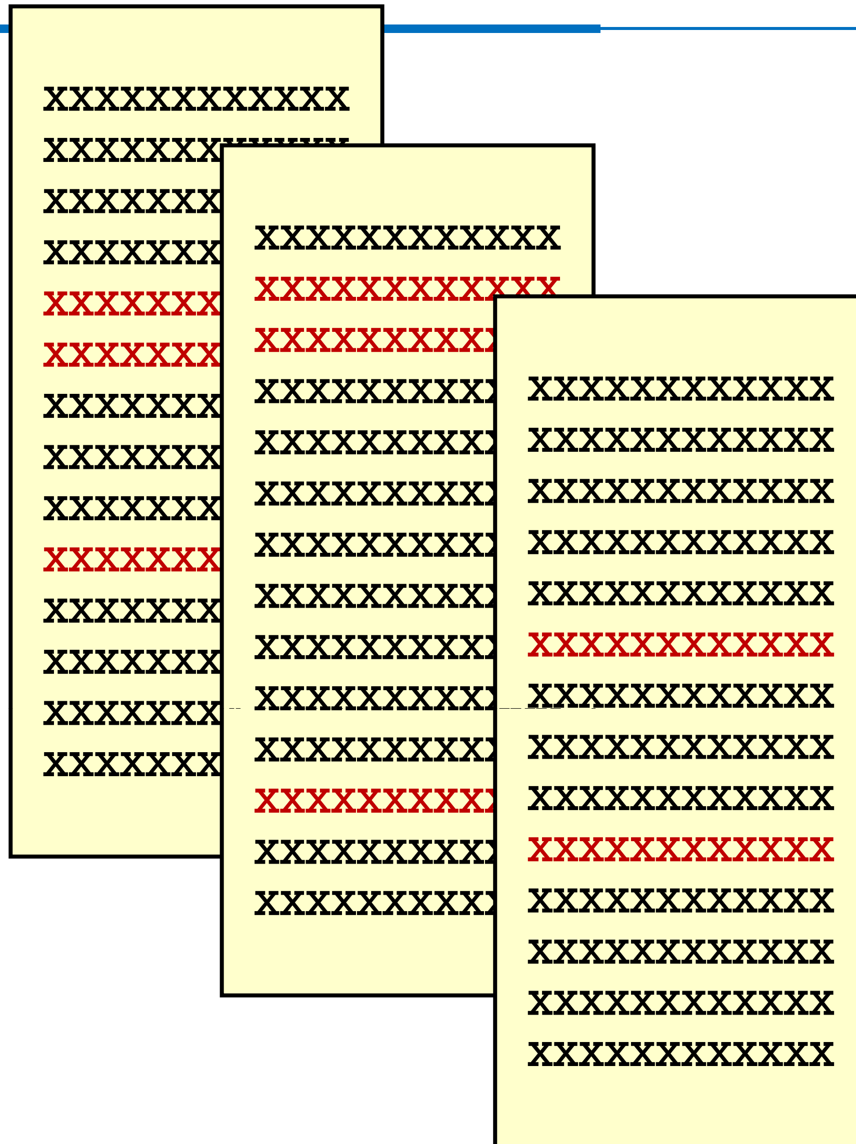
- Separation of concerns:
  Handle each concern separately, ideally by representing them with different modules.

Ideal: Concern and Module in 1-1 Relation



- → Understandability
- ← Natural implementation
- ← Easy identification of parts to change

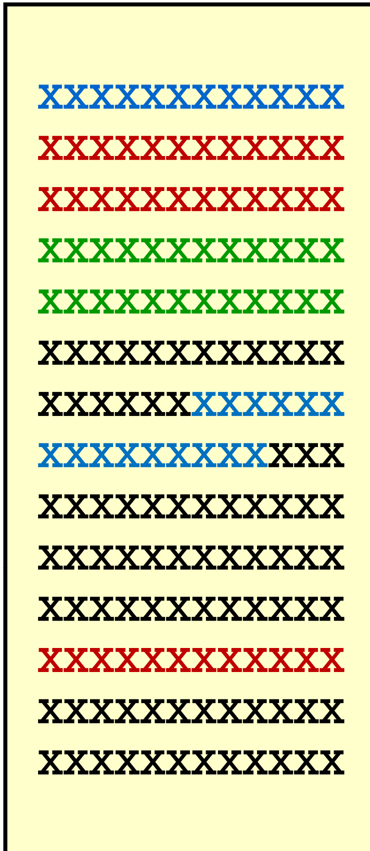
Scattering



- Code addressing one concern is spread across different regions of a program.

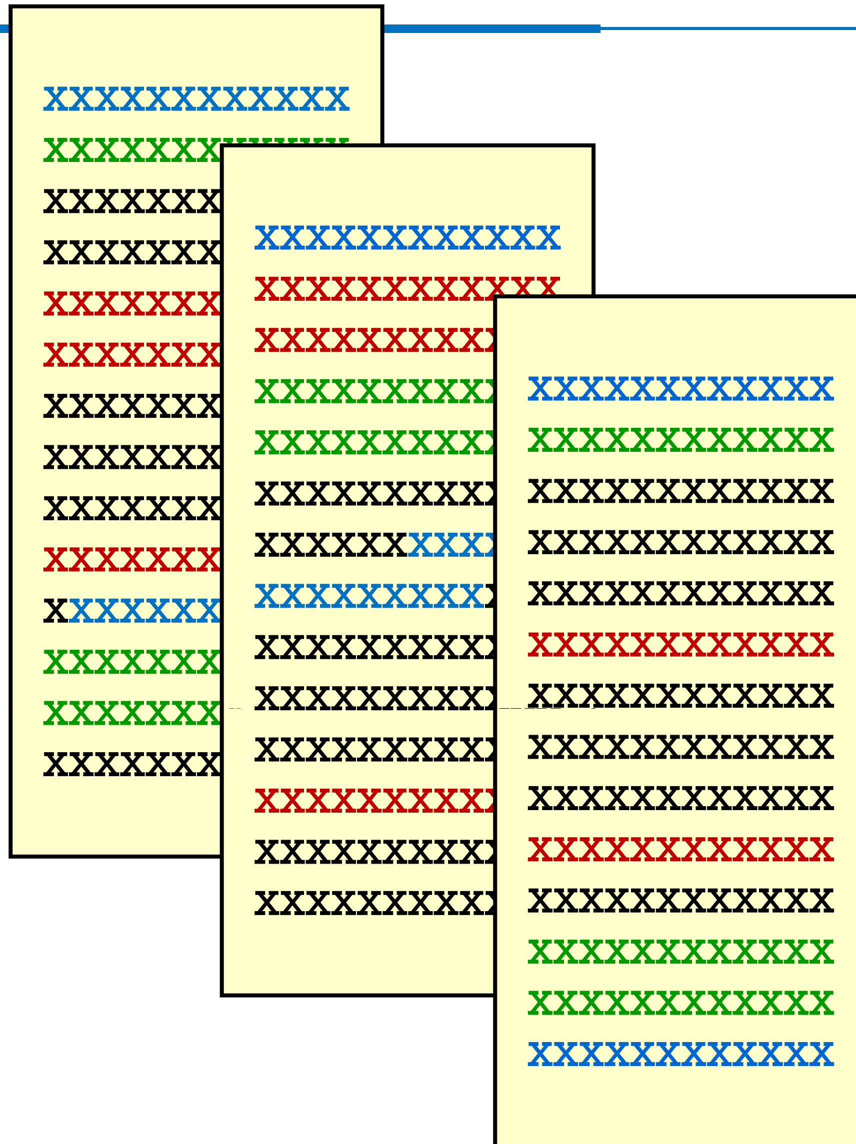
universitätbonn ROOTS

Tangling



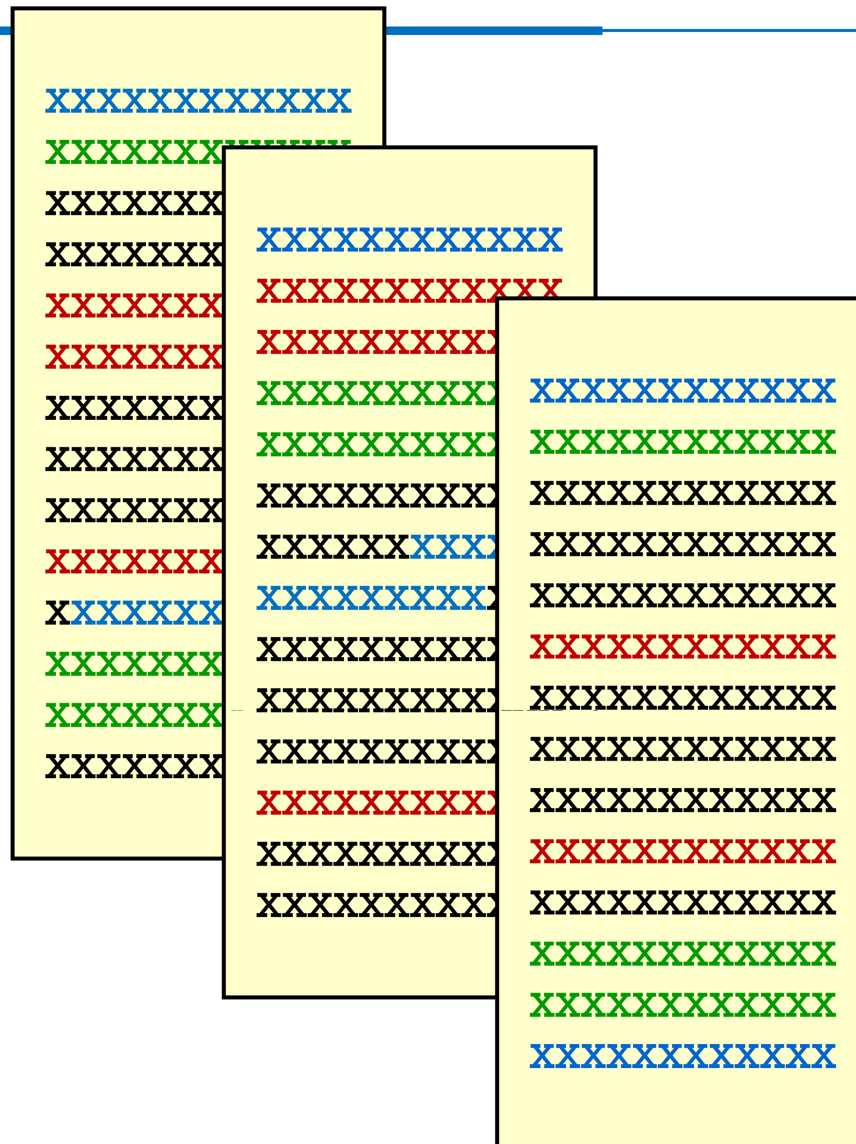
- Code in one region addresses different concerns.

Scattering and Tangling



- Scattering
  - Code addressing one concern is spread across different regions of a program.
- Tangling
  - Code in one region addresses different concerns.
- Scattering and tangling tend to appear together.
  - They describe different facets of the same problem.

Cost of scattered and tangled code



- Redundancy
  - Same or similar fragments of code in many places
- Difficult to understand and reason about
  - Non-explicit structure
  - The big picture isn't clear
- Difficult to change
  - Find all the code involved
  - ... and be sure to change it consistently
  - ... and be sure not to break it by accident
  - Not much help from OO tools

Cross-Cutting Concerns: „Every time … do …“

- **Logging**
  “write something on the screen/file every time the program does X”
- **Error Handling**
  “if the program does X at points like L then do Y”
- **Persistence**
  “every time the program modifies the variable v in class C, then dump a copy to the DB”
- **User Interfaces**
  “every time the program changes its state, make sure the change is reflected on the screen”
- **Synchronization**
  “every time the state of an object is read, first make sure it can’t be changed by another thread”

Adapted from Shmuel Katz (& Johan Brichau & Theo D′Hondt), Introduction into AOSD 2006

Other Cross-Cutting Concerns

- Systemic, “Non-Functional”
  - security (authorization, auditing, encryption)
  - logging, debugging, error handling
  - synchronization and transactions
  - usability (GUI features)
  - distribution, fault tolerance, monitoring
  - persistence and many more non-functional requirements

- Functional
  - business rules and constraints
  - traversal of complex object graphs
  - accounting mechanisms (timing and billing)
  - view update (observer)
  - filter
  - aggregation mechanisms (component gluing)

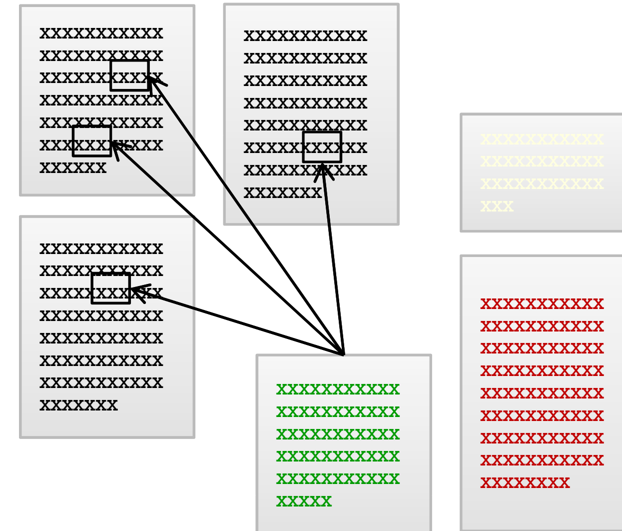
Quantification and Obliviousness

AOP can be understood as the desire to make quantified statements about the behavior of programs and to have these quantifications hold over programs that have no explicit reference to the possibility of additional behavior.

[Robert E. Filman, Daniel P. Friedman]

The Four Technical Ingredients of AOP

1. Joinpoint model: common frame of reference to define the scope of cross-cutting concerns

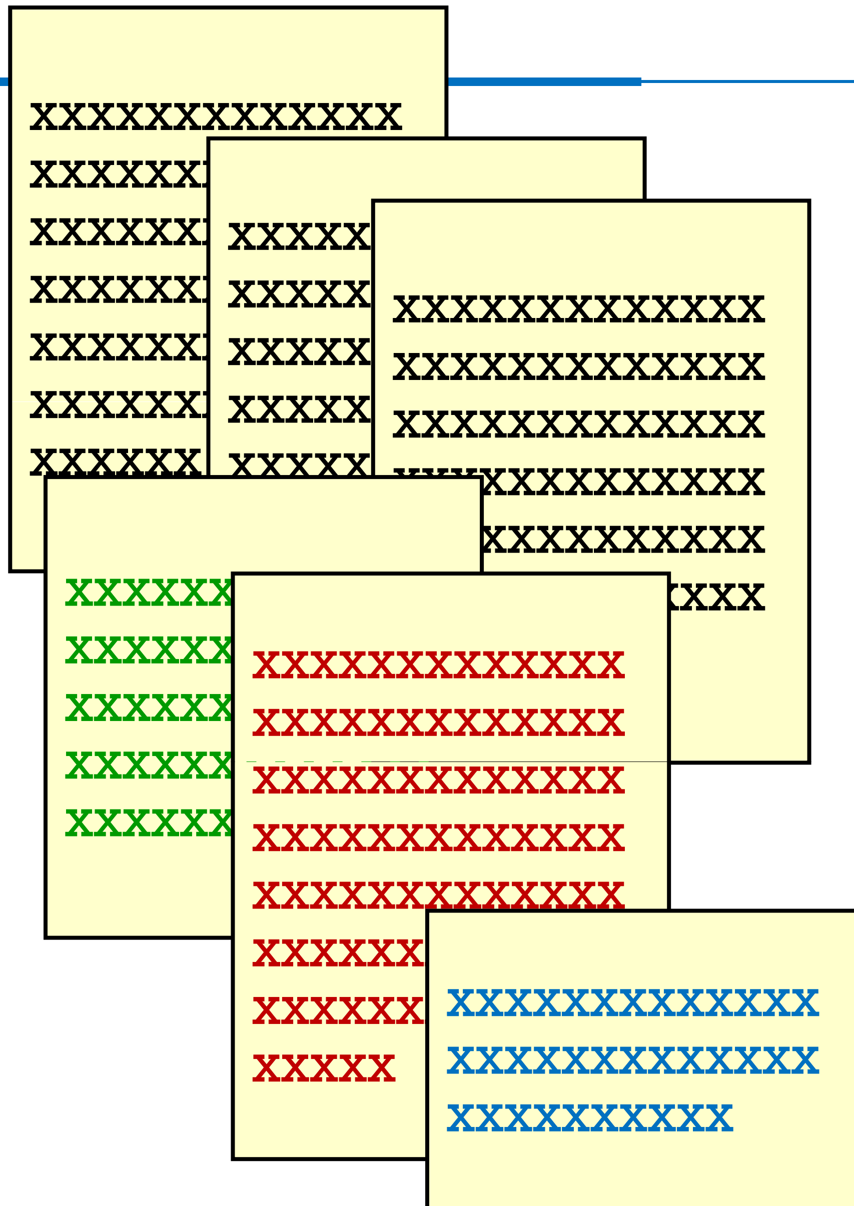


2. Means to identify joinpoints [AspectJ: “pointcuts”]

3. Means to influence structure and behavior at joinpoints [AspectJ: “advice” and “inter-type-declarations”]

4. Means to weave everything together into a functional system

Good modularity



- Separated
  - Implementation of a concern can be treated as relatively separate entity
- Localized
  - Implementation of a concern appears in one part of program.
- Modular
  - Localized + well defined interface to the rest of the system.

The expected benefits

- Locality
  - A system concern is treated in one place, and can be easily changed
- Evolvability
  - Evolving requirements can be added easily with minimal changes to previous version
- Pluggability
  - Configurable components become practical (“On demand computing”)
- Reuse
  - Reuse of code that cuts across usual class hierarchy to augment system in many places

[Shmuel Katz]

Why „aspect“?

So, what makes an Aspect an Aspect, before we even think of programming it with AspectJ? Given the name, we choose for it, which clearly influences our perception, Aspects are **software concerns** that affect what happens in the Objects but **that are more concise, intelligible, and manageable when written as separate chapters** of the imaginary book that describes the application.

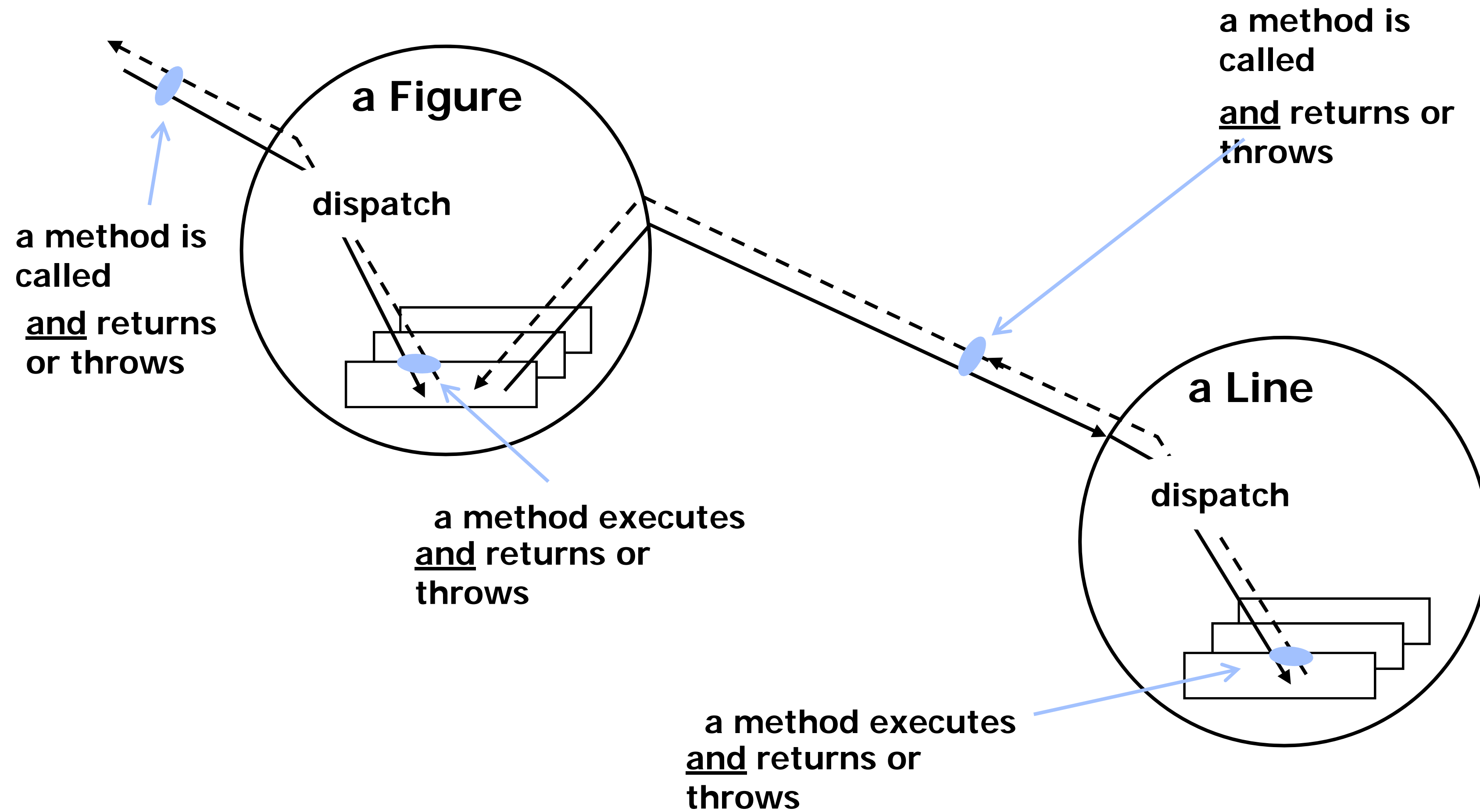
[Christina Videira Lopes]

The AspectJ language in more detail

- Joinpoint Model
- Advice: Before, After, Around
- Pointcuts: Primitive, Named, Kinds, Parameters

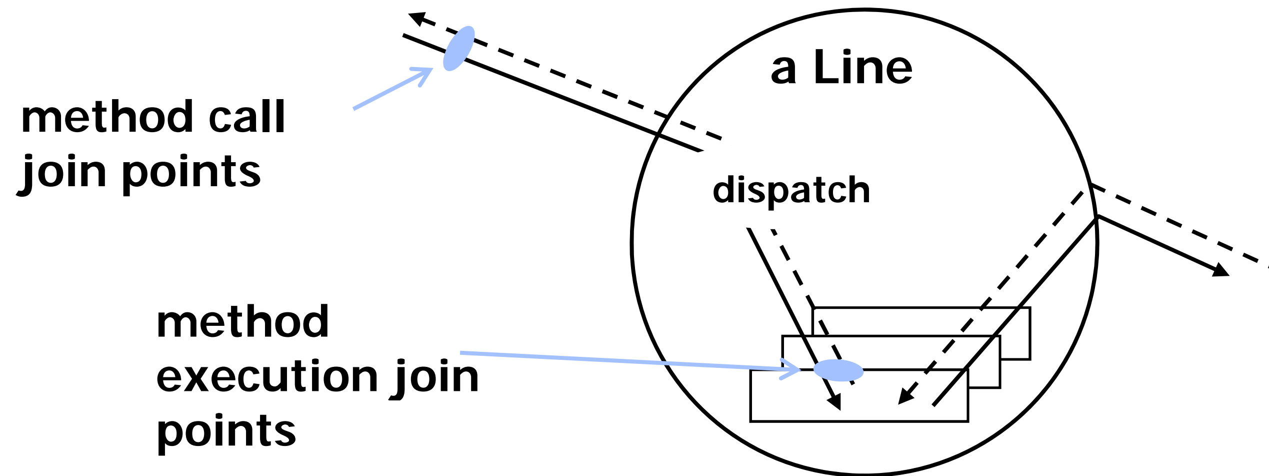
join points

key points in dynamic call graph



(c) Copyright 1998-2001 Xerox Corporation. All rights reserved.

join point terminology

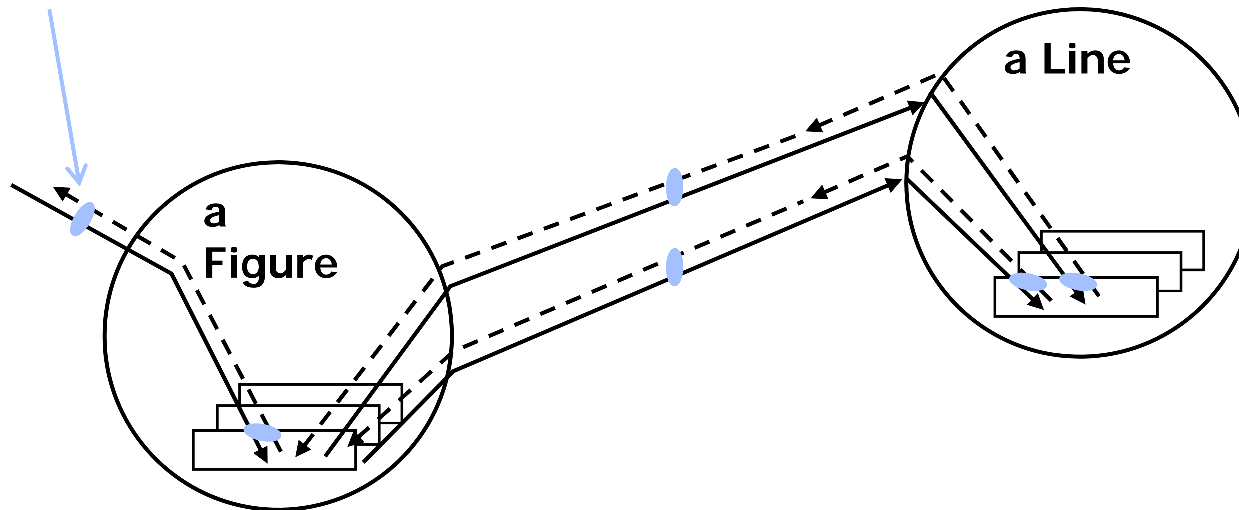


- several kinds of join points
  - method & constructor call join points
  - method & constructor execution join points
  - field get & set join points
  - exception handler execution join points
  - static & dynamic initialization join points

join point terminology

key points in dynamic call graph

all join points on this slide are within the control flow of this join point



repeated calls result in new join points

Primitive pointcuts

a pointcut is a kind of predicate on join points that:

- can match or not match any given join point and
- Optionally, can pull out some of the values at that join point

Example:

```
call(void Line.setP1(Point))
```

matches if the join point is a method call with this signature

pointcut composition

pointcuts compose like predicates, using &&, || and !

a "void Line.setP1(Point)" call

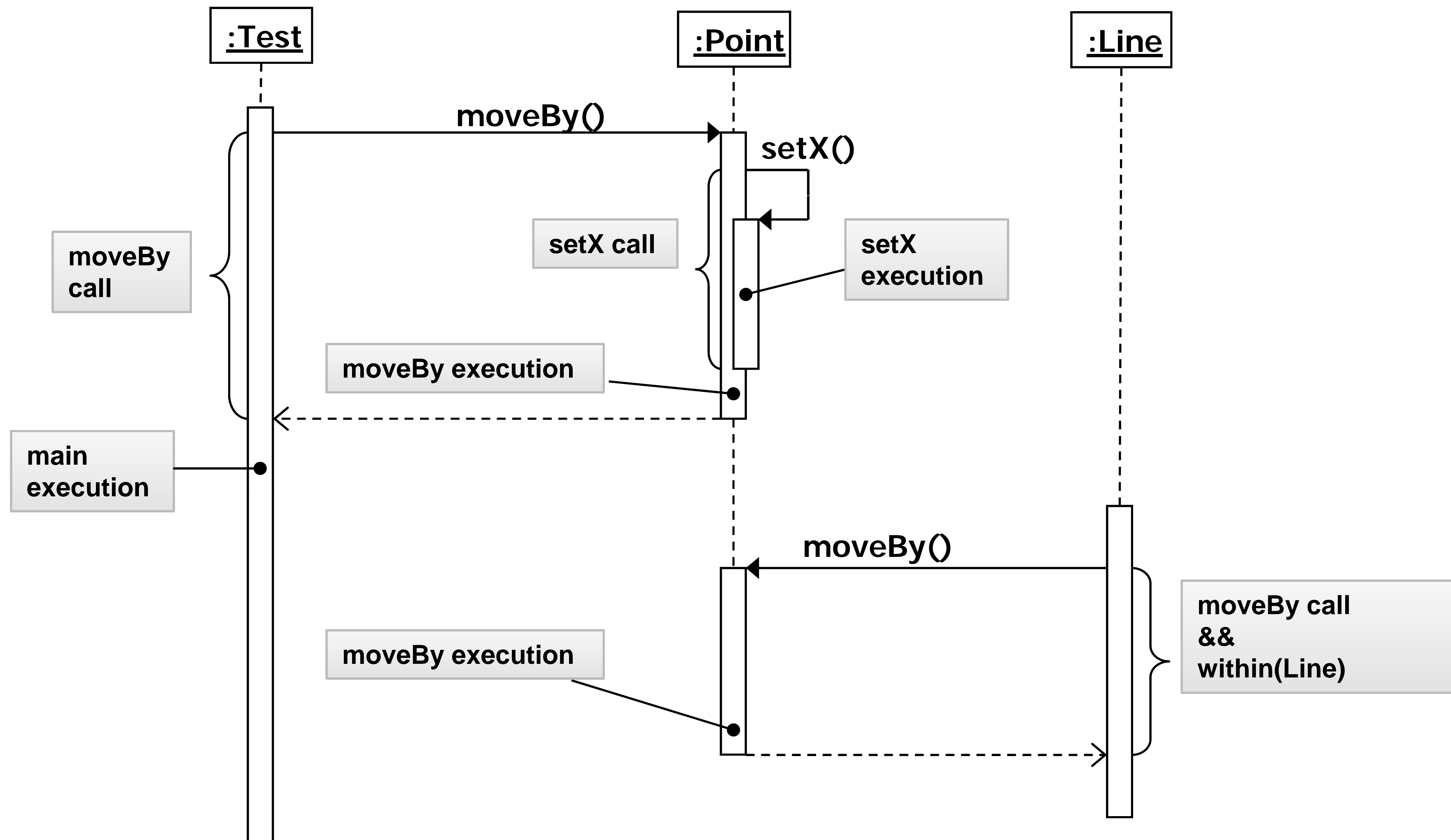
or

```
call(void Line.setP1(Point)) ||
call(void Line.setP2(Point));
```

a "void Line.setP2(Point)" call

each time a Line receives a
"void setP1(Point)" or "void setP2(Point)" method call

Explanation of primitive pointcuts: call, execution, within



User-defined pointcuts

user-defined (also: named) pointcuts

- can be used in the same way as primitive pointcuts

name → parameters

```
pointcut move():
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point));
```

Pointcuts

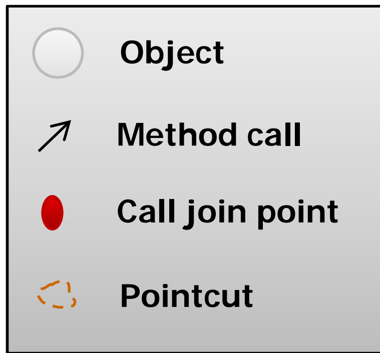
user-defined pointcut designator

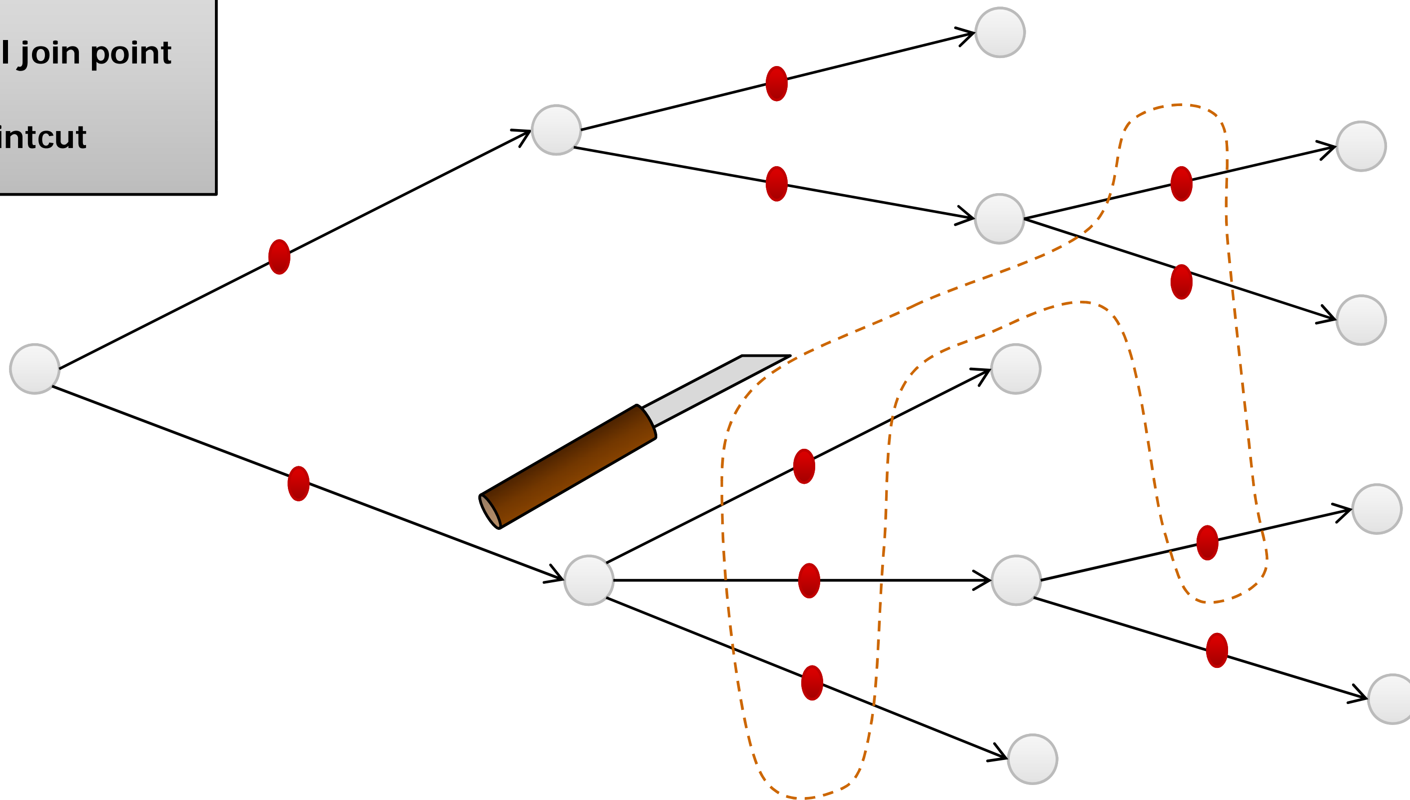
```
pointcut move():
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point));
```

primitive pointcut designator, can also be:

- call, execution
- this, target
- get, set
- within, withincode
- handler
- cflow, cflowbelow
- initialization, staticinitialization

Pointcuts: Cutting out joinpoints





Advice: Additional action to take at join point

- before — before entering join point

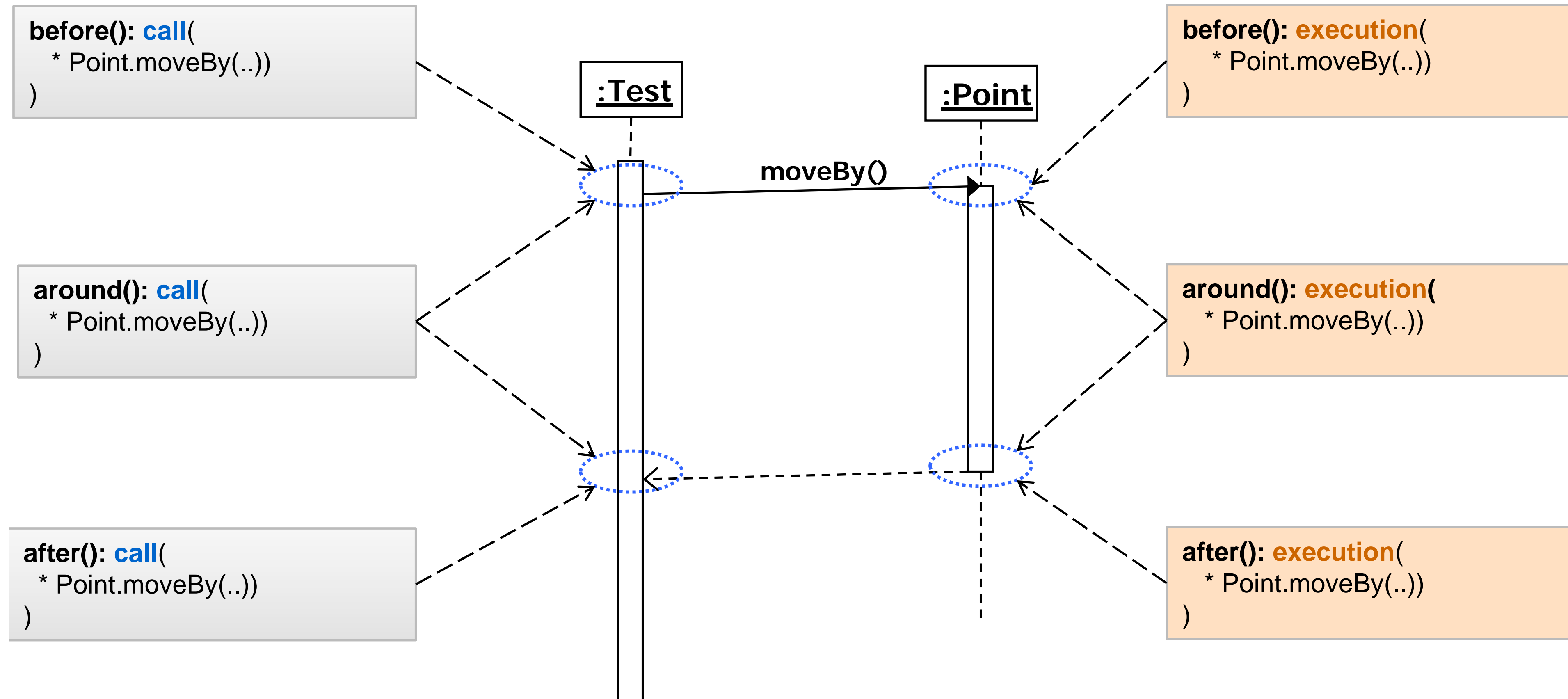
- after returning — … a value from join point
- after throwing — … a throwable from join point
- after — … returning from join point either way

- around — on arrival at join point
  (gets explicit control over when/if program proceeds)

“Proceeding” from around advice

- Syntax: <result type> proceed(arg1, arg2, …)
- Available only in around advice
- Means “run what would have run if this around advice had not been defined”

Advice types: before, after, around



Each dotted circle represents an opportunity for running advice

Parameters: Using values at join points

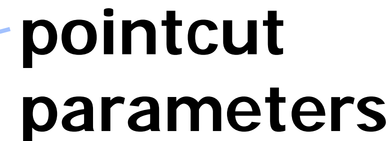
- pointcut can explicitly expose certain values as parameters
- advice can use these values

parameter mechanism being used

```
pointcut move(FigureElement figElt):
  target(figElt) &&
  (call(void FigureElement.moveBy(int, int)) ||
   call(void Line.setP1(Point))             ||
   call(void Line.setP2(Point))             ||
   call(void Point.setX(int))               ||
   call(void Point.setY(int))
  );

after(FigureElement fe) returning: move(fe) {
  <fe is bound to the figure element>
}
```

explaining parameters…

- variable in place of type name in pointcut designator
  - pulls corresponding value out of join points
- variable bound in user-defined pointcut designator
  - makes value accessible in pointcut




```
pointcut move(Line l):
   target(l) &&
   (call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)));
```

explaining parameters...

- variable bound in advice
- variable in place of type name in pointcut designator
  - pulls corresponding value out of join points
  - makes value accessible within advice

advice parameters

typed variable in place of type name

```
after(Line line): move(line) {
  <line is bound to the line>
}
```


universitätbonn ROOTS

explaining parameters...

- value is ‘pulled’ from right to left across the ‘:’ separator

```
left side : right side
```

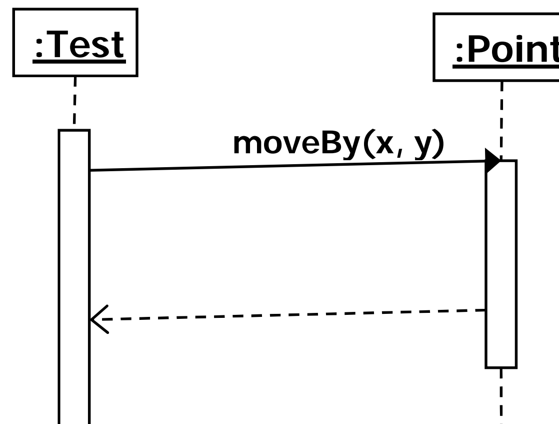
- from pointcut designators to user-defined pointcut designators

```
pointcut move(Line l):
  target(l) &&
  (call(void Line.setP1(Point)) ||
   call(void Line.setP2(Point)));
```

- from pointcut to advice

```
after(Line line): move(line) {
  <line is bound to the line>
}
```

Use the primitve pointcuts this, target, args to access state at the joinpoint




```
pointcut move(Test mover, Point moved, int dx, int dy):
   this(mover) && target(moved) && args(dx, dy) &&
   call(void Point.moveBy(int, int));
```

So much for today.
To be continued…

[Questions?]