Brief Introduction into Aspect Oriented Programming (AOP)

Anette Lang\textsuperscript{1}  Florian Kronenberg\textsuperscript{2}

\textsuperscript{1} University of Bonn
\textsuperscript{2} RWTH Aachen

30-06-07 / XP Know-How Meeting
Outline

1. Motivation
   - Limited Modularity in Object-Oriented Programming
   - Logging as a typical example for Scattering and Tangling

2. Aspect-Oriented Programming
   - Modularisation of Cross-cutting Concerns
   - AOP Building Blocks

3. AspectJ
   - Overview
   - Example: Have Fun Logging with AspectJ
   - Advice Pointcut Types
Our Goals.

- efficient software development requires modularity, also called separation of concerns
- history of encapsulating logically related source code into functional modules:
  - binary code
  - assembler language (instructions)
  - Procedural Programming (procedures)
  - Object Oriented Programming (classes)
- modules provide a certain functionality through well-defined interfaces offering the following advantages:
  - changes can be done locally
  - system structure is understandable
  - maintenance is easy
  - modules are reusable
Limitations of object oriented programming (OOP)

- natural decomposition of **core concerns (components)** into modules [e.g. use cases, classes]
- some concerns do not fit into this scheme: they are still part of many different modules
Limitations of object oriented programming (OOP) (contd.)

- using OOP:
  - **Scattering:** code implementing one crosscutting concern (aspect) is spread throughout the whole source code [e.g., logging]
  - **Tangling:** code implementing crosscutting concerns is interwoven with code implementing the local module
  - ⇒ code implementing crosscutting concerns cannot be easily encapsulated by using means of OOP [tyranny of the dominant decomposition]
  - ⇒ the larger the project and the many crosscutting concerns occur, the more redundant, less readable, less maintainable and less reusable the code implementing crosscutting concerns
  - interesting for XP: everyone is allowed to change every part of a project’s source code – everyone should be able to
Logging in Tomcat

- **Figure:** Logging in Tomcat

- Bad modularization of concern **logging**.
- Code referring to logging is **scattered** across most packages.
Example for Scattering and Tangling

Logging

Have a look at some methods in ProjectDoALot...

```java
public void doSomething(){
    logger.trace(enter doSomething());
    //execute doSomething()
    int i=1;
    logger.trace("leave doSomething()");
}
public void doSomethingElse(){
    logger.trace(enter doSomethingElse());
    //execute doSomethingElse()
    int i=2;
    logger.trace("leave doSomethingElse()");
}
```
Have a look at some methods in ProjectDoALot...

```java
public void doSomething(){
    logger.trace(enter doSomething());
    //execute doSomething()
    int i=1;
    logger.trace("leave doSomething()");
}

public void doSomethingElse(){
    logger.trace(enter doSomethingElse());
    //execute doSomethingElse()
    int i=2;
    logger.trace("leave doSomethingElse()");
}
```
Example for Scattering and Tangling Logging

Have a look at some methods in ProjectDoALot...

```java
public void doSomething() {
    logger.trace(enter doSomething());
    //execute doSomething()
    int i=1;
    logger.trace("leave doSomething()");
}

public void doSomethingElse() {
    logger.trace(enter doSomethingElse());
    //execute doSomethingElse()
    int i=2;
    logger.trace("leave doSomethingElse()");
}
```
Example for Scattering and Tangling

Logging

Have a look at some methods in ProjectDoALot...

```java
public void doSomething(){
    logger.trace(enter doSomething());
    //execute doSomething()
    int i=1;
    logger.trace("leave doSomething()");
}

public void doSomethingElse(){
    logger.trace(enter doSomethingElse());
    //execute doSomethingElse()
    int i=2;
    logger.trace("leave doSomethingElse()");
}
```
Example for Scattering and Tangling

Logging

```java
public void doSomethingCompletelyElse()
{
    logger.trace(enter doSomethingElse());
    //execute doSomethingCompletelyElse()
    int i=2;
    logger.trace("leave doSomethingCompletelyElse()");
}
```
Example for Scattering and Tangling

Logging

```java
public void doSomethingCompletelyElse()
{
  logger.trace(enter doSomethingElse());
  //execute doSomethingCompletelyElse()
  int i=2;
  logger.trace("leave doSomethingCompletelyElse()");
}
```
Example for Scattering and Tangling Logging

public void doSomethingCompletelyElse(){
    logger.trace(enter doSomethingElse());
    //execute doSomethingCompletelyElse()
    int i=2;
    logger.trace("leave doSomethingCompletelyElse()”);
}

- logging instructions are scattered through the methods
- in each method logging instructions and functional code are tangled

⇒ AOP might help...
Modularisation of crosscutting concerns

- regard the first decomposition as a 2-dimensional scenery
  - x-axis: program code parts (e.g., methods)
  - y-axis: modules
- enlarge the model by using a third dimension
  - z-axis: aspects
  - \((x,y): \text{possible Join Points}\)
  - \(f(x,y,z): \text{additional or different code implementing aspect z, that is executed at Join Point (x,y) (advice)}\)
  - \(\{ (x,y) | f(x,y,z) \text{ is defined} \}: \text{Pointcut} \)
Definitions

- **Joinpoint**
  - model of where to apply cross-cutting concerns to classes
  - location in the source code, where aspect implementing code can be executed
  - logging-example: method call

- **Pointcut**
  - means to identify joinpoints
  - set of Joinpoints, where an advice implementing an aspect is executed
  - logging-example: set of all method calls in the project: ProjectDoALot.*(..)

- **Advice**
  - means to influence structure and behavior at joinpoints
  - implementation of behavior to be executed upon incurring a joinpoint
  - logging-example: System.out.println(…)

Annette Lang, Florian Kronenberg
Motivation
AOP
AspectJ
Summary

Modularisation
Building Blocks

Aspect
- Modularization of a Cross-cutting concern
- contains pointcuts and advices

Weaver
- merges core concerns and aspects into a functional system.
Overview of AspectJ

- Aspect-Oriented Programming Languages?
  - But: We are familiar with OO-languages such as Java
  - We use it for the implementation of core concerns.
  - We want to specify aspects without learning a whole new programming language.

- AspectJ
  - Aspect-oriented extension to Java
  - Java-like syntax for aspect definitions
  - AspectJ-compiler (ajc) compiles AspectJ aspects and optionally weaves them with Java classes
  - Weaving is supported at compile-time (by ajc) or at load-time (by a Java agent)
  - Powerful language for the definition of pointcuts
  - Tool-Support: tight integration with eclipse.
Overview of AspectJ

Aspect-Oriented Programming Languages?
- But: We are familiar with OO-languages such as Java
- We use it for the implementation of core concerns.
- We want to specify aspects without learning a whole new programming language.

AspectJ
- Aspect-oriented extension to Java
- Java-like syntax for aspect definitions
- AspectJ-compiler (ajc) compiles AspectJ aspects and optionally weaves them with Java classes
- Weaving is supported at compile-time (by ajc) or at load-time (by a Java agent)
- Powerful language for the definition of pointcuts
- Tool-Support: tight integration with eclipse.
Example in AspectJ

Logging

```java
public aspect Tracing{
    pointcut traceCall(): call(* ProjectDoALot.*(..));

    before(): traceCall() {
        System.out.println("enter " + thisJoinPoint + \\
"");
    }

    after(): traceCall() {
        System.out.println("leave " + thisJoinPoint + \\
"");
    }

    public void doSomething() {
        //execute doSomething()
        int i=1;
    }
}
```

Anette Lang, Florian Kronenberg
Example in AspectJ
Logging

```java
public aspect Tracing{
  pointcut traceCall(): call(* ProjectDoALot.*(..));

  before(): traceCall() {
    System.out.println("enter " + thisJoinPoint + "\n");
  }

  after(): traceCall() {
    System.out.println("leave " + thisJoinPoint + "\n");
  }

  public void doSomething() {
    //execute doSomething()
    int i=1;
  }
}
```

Anette Lang, Florian Kronenberg
Example in AspectJ

Logging

```java
public aspect Tracing{
    pointcut traceCall(): call(* ProjectDoALot.*(..));

    before(): traceCall() {
        System.out.println("enter " + thisJoinPoint + ":");
    }

    after(): traceCall() {
        System.out.println("leave " + thisJoinPoint + ":");
    }

    public void doSomething(){
        //execute doSomething()
        int i=1;
    }
}
```
Basic Advice Types

- **before**: Execute advice behavior before the joinpoint
- **after**: Execute advice behavior after the joinpoint
- **around**: Control proceeding of the joinpoint. Behavior can be added before and after joinpoint. Access to the context (e.g. parameters, return values).

![Diagram showing before, after, and around advice types with a method call and execution time axes]
Pointcuts

- defined by **pointcut expressions** in the AspectJ pointcut language
- composed from
  - primitive pointcuts
  - parameters (e.g. types, wildcards)
  - boolean operators (for combination)
- Guess what happens with these examples?:

  ```java
  target(Point) && call(int *(..))
  ```

  ```java
  call(* *(..)) && (within(Line) || within(Point))
  ```

  ```java
  within(*) && execution(*.new(int))
  ```

  ```java
  !this(Point) && call(int *(..))
  ```
Pointcuts

- defined by **pointcut expressions** in the AspectJ pointcut language
- composed from
  - primitive pointcuts
  - parameters (e.g. types, wildcards)
  - boolean operators (for combination)

Guess what happens with these examples?:

```java
target(Point) && call(int *())
```

```java
call(* *(..)) && (within(Line) || within(Point))
```

```java
within(*) && execution(*.new(int))
```

```java
!this(Point) && call(int *(..))
```
**Pointcuts**

- defined by **pointcut expressions** in the AspectJ pointcut language
- composed from
  - primitive pointcuts
  - parameters (e.g. types, wildcards)
  - boolean operators (for combination)
- Guess what happens with these examples?:

  ```java
  target(Point) && call(int *())
  ```

  ```java
  call(* *(..)) && (within(Line) || within(Point))
  ```

  ```java
  within(*) && execution(*.new(int))
  ```

  ```java
  !this(Point) && call(int *(..))
  ```
Pointcuts

- defined by **pointcut expressions** in the AspectJ pointcut language
- composed from
  - primitive pointcuts
  - parameters (e.g. types, wildcards)
  - boolean operators (for combination)

Guess what happens with these examples?:

```
target(Point) && call(int *(..))
```

```
call(* *(..)) && (within(Line) || within(Point))
```

```
within(*) && execution(*.new(int))
```

```
!this(Point) && call(int *(..))
```
A Selection of Primitive Pointcuts

- call(Signature): Aufruf einer Methode
- execution(Signature): Ausführung einer Methode
- within(Type): Innerhalb einer Klasse/Typ
- get(Signature): Lesender Zugriff auf ein Feld
- set(Signature): Schreibender Zugriff auf ein Feld
Summary

- Aspect-Oriented Programming (AOP) is more powerful than OOP in the Separation of Concerns.
- AOP helps modularizing Cross-Cutting Concerns.
- AspectJ brings AOP to the Java World