



# Aspect Oriented Programming

Mariano Ceccato

ITC-Irst

Centro per la ricerca  
Scientifica e Tecnologica

[ceccato@itc.it](mailto:ceccato@itc.it)



---

# Outline

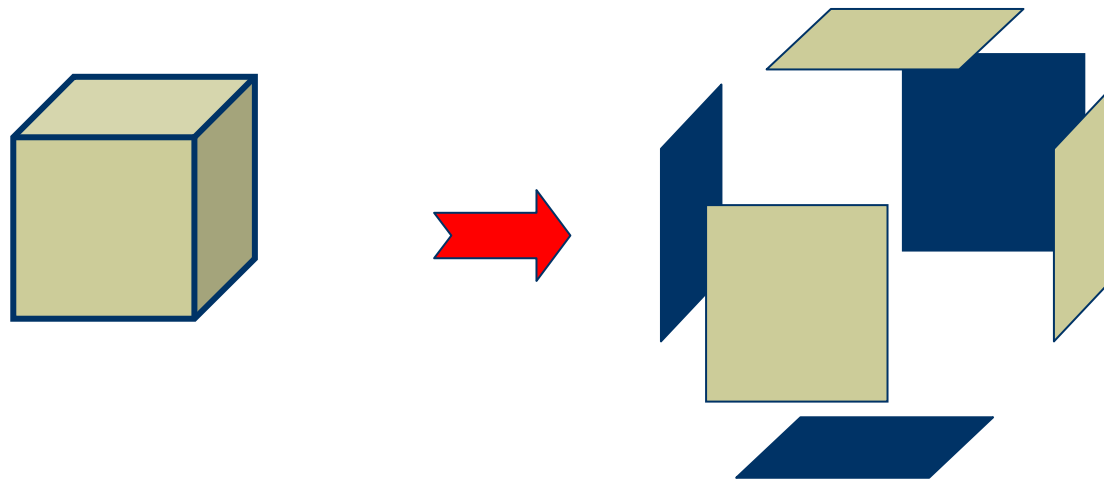
---



- ◆ Motivations
- ◆ Aspect Oriented Programming
- ◆ Language – AspectJ

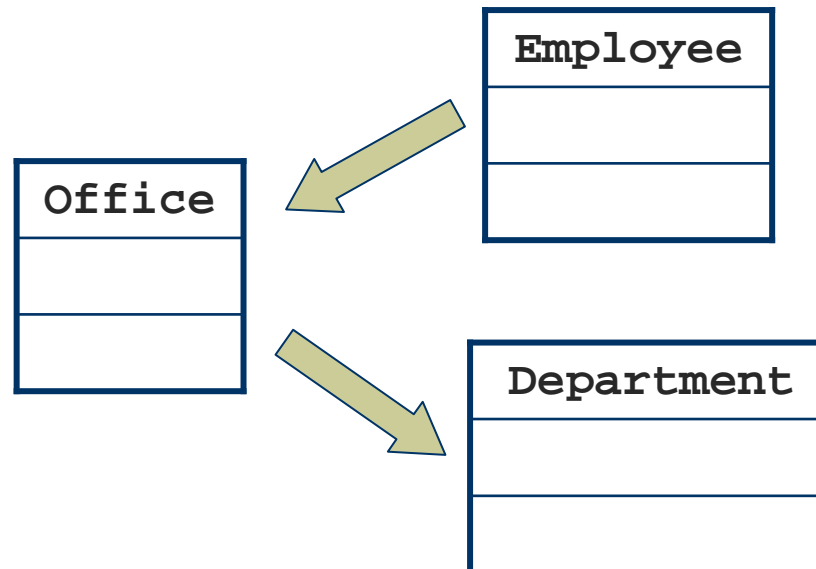
# Modularity

- ◆ Every programming language supports source code modularity
- ◆ Humans cope with complex problems dividing them into many simpler sub-problems.



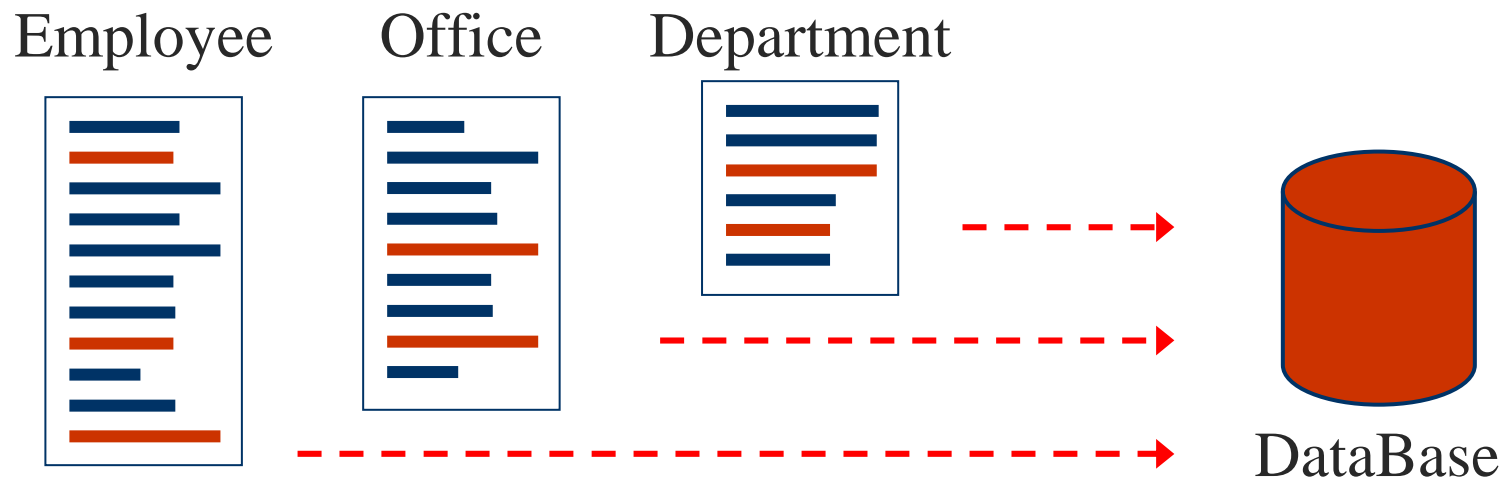
# Modularity in Object Oriented Programming

- ◆ Modules in the object oriented paradigm are the classes.
- ◆ A class should map exactly one entity in the problem domain.



# Crosscutting Concerns

- ◆ In the implementation of any complex system, there are concerns that inherently crosscut the natural modularity of the rest of the implementation.
- ◆ Examples: logging, tracing, persistency, caching, error management, synchronization...



# Example: the tracing concern

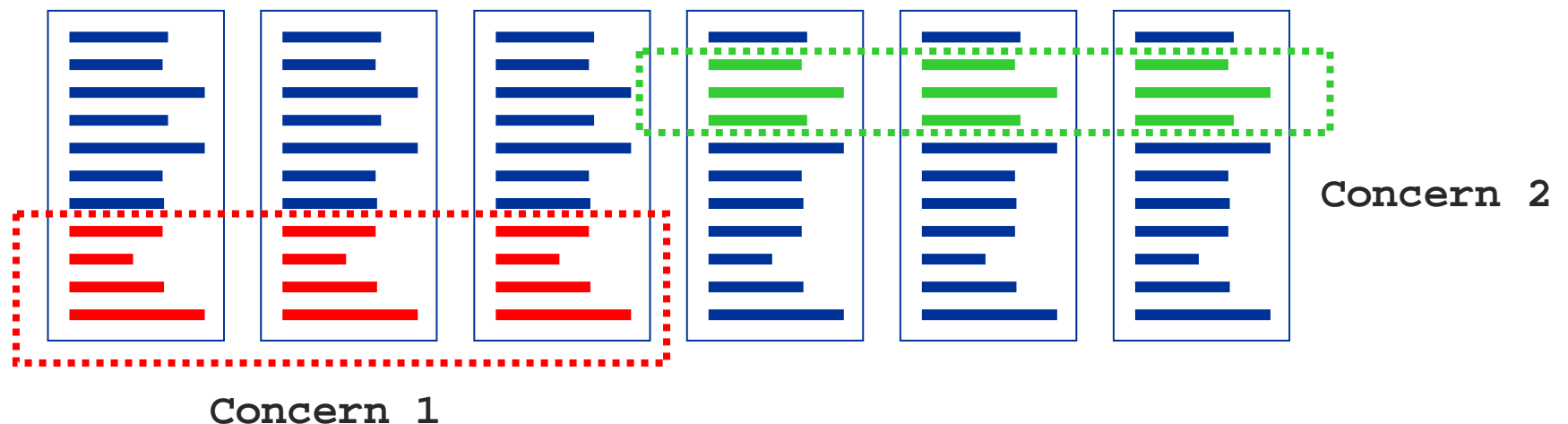
```
class Main {  
  
    public void metod1() {  
        Tracer.entry("method1()")  
        ...  
        Tracer.exit("method1()")  
    }  
  
    public void method2(int i) {  
        Tracer.entry("method2(int)")  
        ...  
        Tracer.exit("method2(int)")  
    }  
  
    public static void main(String [] args) {  
        Tracer.entry("main(String[])")  
        ...  
        Tracer.exit("main(String[])")  
    }  
}
```

# Example: the persistency concern

```
...
Employee emp1 = new Employee;
DataBase.store( emp1.toByteArray() );
...
Office off = new Office();
DataBase.store( off.toByteArray() );
...
emp2.updateSalary( amount );
DataBase.update( emp2.toByteArray() );
...
if (today.isHollyday() = true) {
    emp3.setPresence( false );
    //update not required
}
...
```

# Scattering

- ◆ Crosscutting concerns are affected by scattering.
  - The code implementing the same concern is spread in many different modules.
  - In general there is no language support to understand which modules have a role in the same concern.

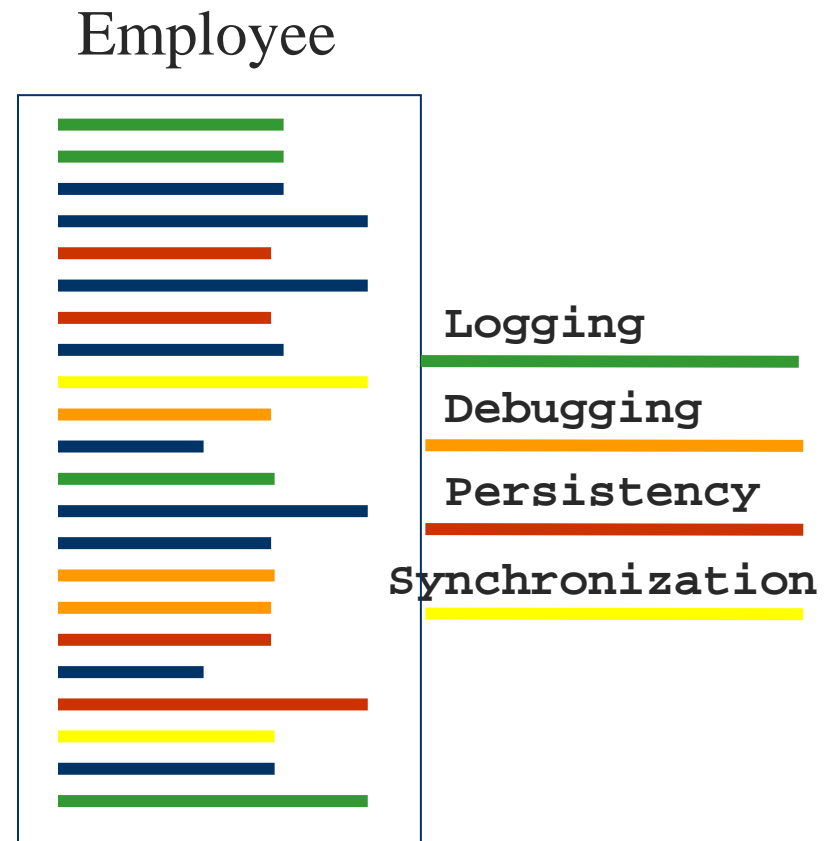


# Scattering drawbacks

- ◆ Modifying a crosscutting concern require to change a lot of modules in the same time.
- ◆ All the involved modules have to reasoned about at one time, loss of modularization benefit.
- ◆ Understanding which are all the participation modules can be not so easy.
- ◆ It can be not so easy to see the presence of a crosscutting concern (programming language doesn't support it).

# Tangling

- ◆ The code pertaining to a crosscutting concern is intermixed with the rest of the module code.
- ◆ The same module can be affected by several crosscutting concerns.
- ◆ Concerns can not be separated into different modules using standard modularization mechanisms.

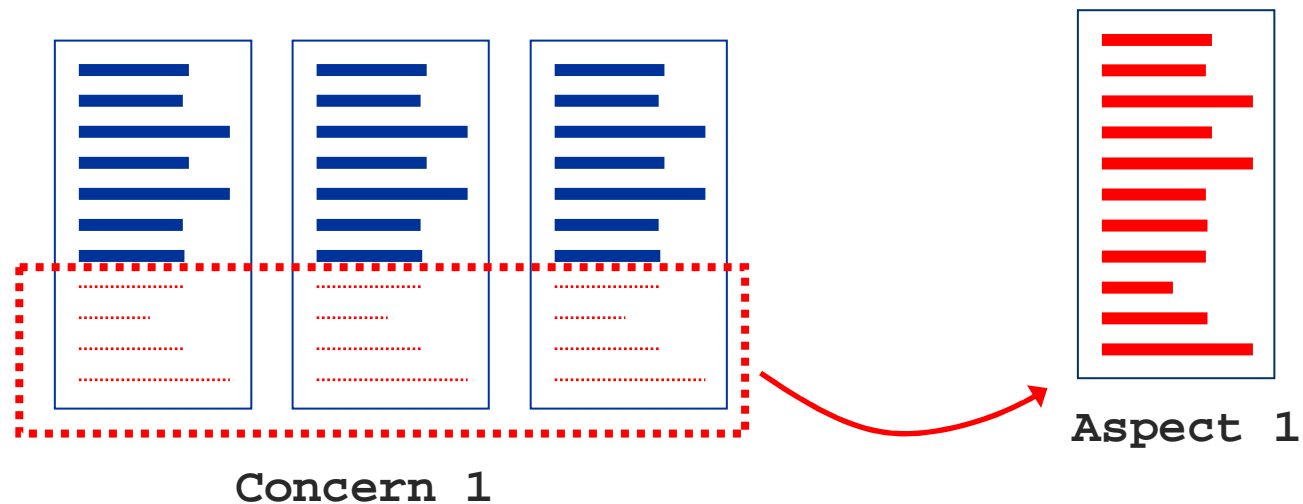


# Tangling drawbacks

- ◆ If the code of each concern is intermixed with the code of the others, it is not so simple to understand the behavior of a module
- ◆ When it is required to modify a crosscutting concern
  - Understanding which parts of the code pertain to the concern and which ones pertain to the others can be hard.
  - A change in the concern can have ripple effects that are hard to locate.

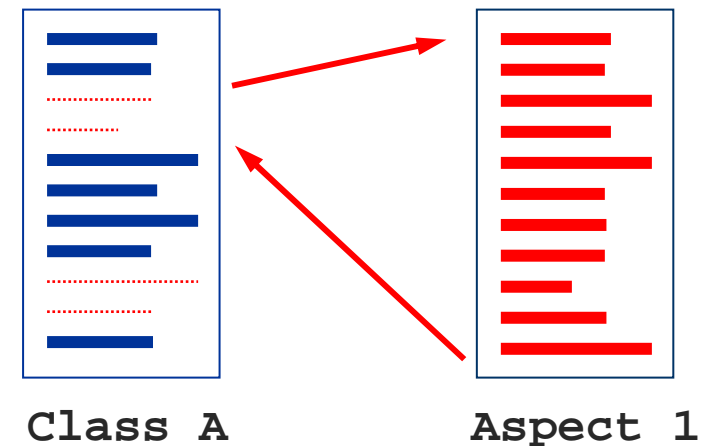
# The AOP purpose

- ◆ The purpose of Aspect Oriented Programming is to separate the principal decomposition of a module from its crosscutting concerns.
- ◆ AOP provide a new type of modules: the *aspect*. An aspect factors out all the related code fragment coming from different modules.
- ◆ The resulting code should be easy to understand and to evolve, due to the tangling and scattering elimination.



# How AOP works

- ◆ The base code has no references to the crosscutting concern.
- ◆ An aspect defines some points in the base program flow that need the concern.
- ◆ When these point are reached
  - The aspect takes the control flow
  - It executes the code of the concern
  - It returns the control to the base program
- ◆ These points are called *join points*



# AOP – Persistency

```
...
Employee emp1 = new Employee;
DataBase.store( emp1.toByteArray() );
...
Office off = new Office();
DataBase.store( off.toByteArray() );
...
emp2.updateSalary( amount );
DataBase.update( emp2.toByteArray() );
...
if (today.isHollyday() = true) {
    emp3.setPresence( false );
    //update not required
}
...
```

# AOP – Persistency

```
...  
Employee emp1 = new Employee;  
...  
Office off = new Office();  
...  
emp2.updateSalary( amount );  
...  
if (today.isHollyday() = true) {  
    emp3.setPresence( false );  
    //update not required  
}  
...
```

```
aspect Persistency {  
    //after a new Employee instace creation  
    DataBase.store( employee.toByteArray() );  
  
    //after a new Office instace creation  
    DataBase.store( office.toByteArray() );  
  
    //after a new Employee.salary change  
    DataBase.update( employee.toByteArray() );  
}
```

# Pointcuts

- ◆ A *pointcut* contains the definition of the join points that have to be intercepted.
  - There are some primitive pointcut that intercepts method/constructor invocations, field accesses...
  - More sophisticated pointcuts can be defined by combining primitive ones using set operations (union, interception ...)

```
//after a new Employee
//instance creation
DataBase.store(
    employee.toByteArray() );

//after a new Office
//instance creation
DataBase.store(
    office.toByteArray() );
```

# Advices

- ◆ An advice is the code that the aspect executes when a join point is reached.
- ◆ Depending on its definition, an advice can be executed:
  - before the reached join point,
  - after the join point or
  - instead of it (around).

```
//after a new Employee
//instance creation
DataBase.store(
    employee.toByteArray() );

//after a new Office
//instance creation
DataBase.store(
    office.toByteArray() );
```

# AspectJ

- ◆ *AspectJ* is an extension of the Java language, with aspects.
- ◆ By compiling an AspectJ project, plain java compatible bytecode is produced.
- ◆ Available on <http://eclipse.org/aspectj>
- ◆ It is the most cited and better supported aspect oriented language

# Pointcut syntax in AspectJ

```
pointcut :=  
pointcut id ( [parameters] ) : pointcut_declarator ;
```

```
primitive_pointcut :=  
call( method )  
call( constructor )  
set( field )  
get( field )  
...
```

```
pointcut_declarator :=  
primitive_pointcut  
pointcut_declarator || pointcut_declarator  
pointcut_declarator && pointcut_declarator  
! pointcut_declarator
```

Examples:

```
pointcut employeeInstanceCreation(): call( Employee.new() );
```

```
pointcut officeInstanceCreation(): call( Office.new() );
```

# Pointcut syntax in AspectJ

- ◆ Pointcut can use wildcards in order to match several different join points
  - \* matches any string
  - .. match any signature

```
pointcut instanceCreation(): call( *.new(int, int) );
```

This pointcut matches any constructor invocation with two int parameter

```
pointcut officeStringTranslation(): call( Office.toString(..) );
```

This pointcut matches invocation to the *toString* methods in class *Office* with any parameter number and type.

# Advice Syntax in AspectJ

```
advice :=  
    advice_spec : pointcut { body }
```

```
advice_spec :=  
    before ( [parameters] )  
    after ( [parameters] )  
    around ( [parameters] )
```

Example:

```
afert(): employeeInstanceCreation() {  
    ...//perform same action  
}
```



# Intertype Declaration in AspectJ



- ◆ Aspects, not only modify class behavior, but also class internal structure.
- ◆ They can change inheritance relations
- ◆ They can add new members
  - Fields
  - Methods

# Intertype declaration Examples

```
declare parents : Office implements Serializable;  
  
public int Office.anIntField ;  
  
private void Office.writeObject ( ObjectOutputStream s ) {  
    ... //body of the method  
}  
private void Office.readObject ( ObjectInputStream s ) {  
    ... //body of the method  
}
```



# Advantages



- ◆ Since a crosscutting concern is modularized into single unit, it is easier to evolve its behavior. In fact only one module has to be taken into account, the aspect.
- ◆ The comprehension of the concern is easier.
- ◆ The presence of the concern is underlined by a language feature.
- ◆ There is a clear separation between all the different concerns code

# Scenarios

## **Generative:**

- ◆ Aspect can be used to add new features to an existing application (for instance persistency).
- ◆ When the new feature is a crosscutting concern, it can be coded in an aspect. The new code doesn't tangle with the already existing one.

## **Migration:**

- ◆ In an object oriented application an existing crosscutting concern is refactored into an aspect.

# Weaving aspect code

- ◆ The AspectJ compiler weaves aspects by modifying the source code of all the intercepted classes.
  - Fields and Methods introduction correspond to a new member in the base code.
  - Advices correspond to a new methods.
  - Pointcuts correspond to invocations to the advice methods.
- ◆ The modified class source code is then compiled using the standard Java compiler.

# AOP Vs Source Transformation

## TXL

- ◆ General purpose
- ◆ Very powerful
- ◆ Quite complicated for software developer

## AOP

- ◆ Special purpose
- ◆ Some limitations
- ◆ Easier to use and to maintain



# AOP for this year project

- ◆ Look at the AspectJ language specifications.
- ◆ Download AspectJ compiler and try it.
- ◆ Extend the TXL java grammar in order to parse AspectJ source files.
  
- ◆ The final purpose is to add a new crosscutting feature to Jconsole

# References

- ◆ Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. *An overview of AspectJ*. In Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18--22 June 2001.  
<http://www.parc.com/research/csl/projects/aspectj/downloads/ECOOP2001-Overview.pdf>
- ◆ AspectJ language documentation  
<http://eclipse.org/aspectj>