

Meta-Program and Meta-Programming

- **What is a Meta-Programming ?**

The creation of procedures and programs that automatically construct the definitions of other procedures and programs.

- **First example the Turing machine**

With the Universal Turing Machine, named after Alan Turing, it has been proved that it is possible to program a machine to imitate the behavior of any other machine.

Meta-Program and Meta-Programming

- **What is a Meta-Program ?**

- A program which modifies or generates other programs. A compiler is an example of a metaprogram: it takes a program as input and produces another (compiled) one as output. Genetics programming allows to generate new programs in simulating biological evolution.
- Meta-programs in general are programs that create, control or make decisions about programs, such as when and how to run them, preferred and unpreferred programs, and strategic choices of fall-back or alternative programs.
- Well-known examples : compiler, language parser, genetics programs ...

Why do we need Meta-Programming ?

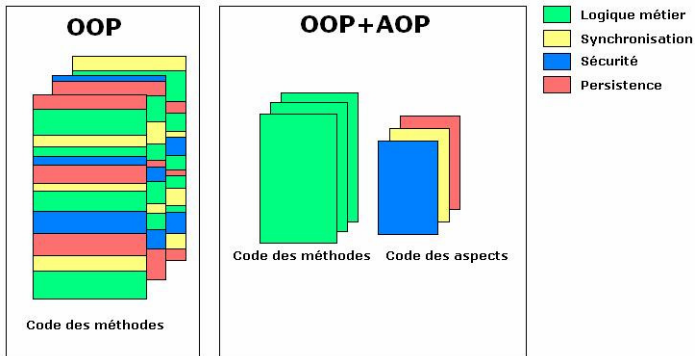
- Object Oriented Design has some interests : capability to capture data semantics, complexity management, abstract level ...

Why do we need Meta-Programming ?

- Object Oriented Design has some interests : capability to capture data semantics, complexity management, abstract level ...
- Limitations: the main principle is to tell that all the tasks to accomplish can be assumed by one object and the methods are always into classes.
- **BUT!** it exist several cases where put methods into classes cause code duplication because sometimes we need more *linear* programming than *hierarchical* programming.

Design example

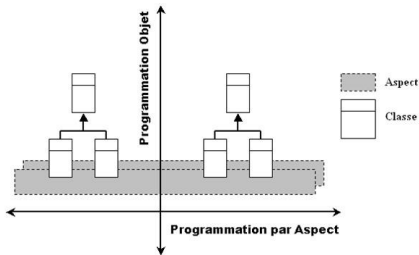
Some cases of transversal concerns : log management, verification for input parameters, exception treatments



...

How to write horizontal sharing of behaviors?

- In the Object design we can separate the business objects from the technical objects.
- Object oriented design encapsulate concerns into single entity.
- But some concerns defy these forms of encapsulation. Software engineers call these crosscutting concerns, because they "cut" across multiple modules in a program.
- So we need to build a **separation of concerns** and **AOP** give a solution for the **cross-cutting concerns** between several classes.



What is AOP?

- The beginning of the story : Gregor Kiczales and his team at the Palo Alto Xerox Parc published their works on writing programs in the 90's.
- Aspect Oriented Programming give a new way to share behaviors, or controls without breaking encapsulation.
- They wrote **AspectJ** which implements AOP for Java applications.
- One of the multiple web site with example :

<https://medium.com/@jdvp/aspect-oriented-programming-in-android-159054d52757>

AOP Principles

- Calling to a technical module from a business one or another technical modules is not direct. For example calling trace management module. This aspect is specified in a self-working way.
- But if the aspect defined explicitly where it interacts with te business module, this just put the problem in another place.
- The **solution** is a system of rational expressions to precise where are the executing point of the different aspects.

AOP Vocabulary

- **Join point**: points in a running program where additional behavior can be usefully joined. A join point needs to be addressable and understandable by an ordinary programmer to be useful
- **Pointcut**: the place where the join points are put. Determine whether a given join point matches.
- **Advice**: code to run at a join point. It can run **before**, **after** and **around** join points.
- **Weaver**: the tool to link aspect code and code of class methods. It can work before or during the compilation time, before or during the runtime.
- **Crosscutting concerns**: secondary requirements shared by several classes.

Example

```
class RealSquareRootExample {
    public static void main(String[] args) {
        System.out.println("sqrt(13.0) is " + Math.sqrt(13.0));
        System.out.println("sqrt(9.0) is " + Math.sqrt(9.0));
        System.out.println("sqrt(-4.0) is " + Math.sqrt(-4.0));
    }
}

aspect EnsureRealSquareRoot {
    before(double d) : call(static double Math.sqrt(double)) && args(d) {
        if(d < 0.0)
            throw new IllegalArgumentException("Positive arguments to sqrt() only, please!");
    }
}
```

Editor Example

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

before(): move() {
    System.out.println("about to move");
}

after() returning: move() {
    System.out.println("just successfully moved");
}

pointcut setXY(FigureElement fe, int x, int y):
    call(void FigureElement.setXY(int, int))
    && target(fe)
    && args(x, y);

after(FigureElement fe, int x, int y) returning: setXY(fe, x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ").");
}
```

The new Observer

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }
    pointcut changes(Point p): target(p) && call(void Point.set*(int));

    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }
}
```