

# Qualità del codice

- Pre-requisiti per produrre classi che possono risultare corrette e manutenibili
  - Documentazione sul codice delle classi
  - Information hiding
  - Bassa complessità per ciascun metodo e ciascuna classe
  - Alta coesione (una sola responsabilità) per ciascuna classe
  - Basso accoppiamento (coupling) tra classi
- Per produrre classi poco complesse, altamente coese e con basso accoppiamento, è utile usare
  - Stili architetturali (Blackboard, Pipe-Filter, etc.)
  - Design Pattern (Singleton, Factory Method, Adapter, Observer, etc.)
  - Refactoring

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 1

# Logging e OOP

- Tramite OOP
    - L'implementazione del logging (o di un'altra funzionalità) può risultare sparsa tra varie classi
    - Es. all'interno di vari metodi è presente una istruzione del tipo: `logObj.writeLog("Changed from "+oldval+" to "+newval);`
- Ovvero,
- Stabilita una certa decomposizione in classi di un sistema, alcune "parti di interesse" (*concern*) non possono che essere implementate tramite codice sperso su varie classi e non come un'unica classe
  - Si definisce questa impossibilità come tirannia della decomposizione predominante

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 2

# Concern

- Un sistema software realizza vari requisiti (funzionalità)
- Nella fase di design, i vari requisiti sono partizionati e i moduli corrispondenti sono identificati
  - Es. per gestione ordini: accounting, processing, presentazione risultati
  - Più in generale: logging, business logic, persistenza, sicurezza
- Separare le differenti realizzazioni dei requisiti in moduli diversi è importante (separation of concern)
  - Una soluzione modulare è una realizzazione in cui ciascun requisito è realizzato in un modulo a sé

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 3

# Case Study e concern

- Studi fatti su Apache Tomcat mostrano
  - Parsing XML -> 1 classe intera :-)
  - Pattern matching URL -> 2 classi intere :-)
  - Logging -> frammenti in tutte le classi :-)
  - Session expiration -> frammenti in tante classi :-)
- Parsing XML, pattern matching URL, Logging, session expiration sono concern
- Logging e session expiration sono concern il cui codice è sperso su tutte/tante classi
  - Si dice che questi concern sono trasversali (o crosscutting)

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 4

## Esempio

- Requisiti del sistema software da realizzare
  - Tenere dati riguardanti vari prodotti, es. nome, costo, etc.
  - Registrare ogni variazione del costo del prodotto
- Classi individuate
  - Product
  - Logger

```
// Classe che tiene i log
public class Logger {
    public void writeLog(String l) {
        // ...
    }
}
```

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 5

## Classe Product

```
// Classe che contiene informazioni su prodotti
public class Product {
    private double price;
    Logger logObj; // codice per Log
    Product() {
        logObj = new Logger(); // codice per Log
        price = 0.0;
    }
    public void putPrice(double p) {
        logObj.writeLog("Price changed from "+price+" to "+p); // codice x Log
        price = p;
    }
    public double getPrice() { return price; }
    public static void main(String args[]) {
        Product product = new Product();
        product.putPrice(5.00);
    }
}
```

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 6

## Considerazioni

- La classe Product
  - Ha la responsabilità di mantenere informazioni sul prodotto
  - Ma, deve anche avvisare la classe Logger delle variazioni di costo
- Il requisito sulla registrazione delle operazioni è sparso (scattered) tra le classi Product e Logger
- Il codice per soddisfare il requisito sulla registrazione si mischia (tangling) con il codice che realizza Product
- La classe Product non dovrebbe avere invocazioni a metodi di Logger
- Questo è quello che avviene con OOP

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 7

## Nuovi requisiti

- Altri requisiti potrebbero essere aggiunti
  - Registrare il tempo necessario alla variazione del costo
  - Autenticare la richiesta di variazione (access control)
  - Contare il numero di prodotti restanti

```
public class Product {
    public void putPrice(double p) {
        // start time
        // Log
        // check user authentication
        price = p;
        // end time
    }
    // ...
}
```

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 8

## Codice risultante

- Il codice di Product si complica e
  - Diventa difficile da comprendere
  - È più soggetto a contenere errori
  - È più difficile da spiegare e da cambiare
  - Non può essere riusato
- In generale
  - I concern crosscutting soffrono dei suddetti problemi, se sono implementati con la programmazione OO

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 9

## Altra soluzione: AOP

- Product dovrebbe essere priva di qualunque interazione con Logger
- Logger dovrebbe essere invocata per ciascun metodo di Product la cui esecuzione deve essere registrata
- Aspect-Oriented-Programming (AOP)
  - Permette di definire un *aspetto* (modulo) che specifica quali azioni compiere quando un metodo di Product è chiamato
- In generale,
  - AOP fornisce il supporto per separare quei concern che altrimenti sarebbero implementati in modo *crosscutting*

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 10

## Aspetto Logger

```
// Aspetto che implementa le operazioni per avviare il logging
public aspect Logger { // dichiarazione aspetto
    // crea una istanza che contiene i dati per il logging
    private Log logOb = new Log();

    // definisce il pointcut, cioè i metodi da catturare per la classe
    // Product
    pointcut loggable() : call(public void Product.putPrice(*));

    // definisce l'advice, cioè le operazioni da eseguire quando avviene
    // la cattura definita dal pointcut
    before() : loggable() {
        logOb.writeLog("Price changed");
    }
}
```

Ing. E. Tramontana - Aspect-Orientation - 26-Ott-06 11