Considerazioni Su Proxy

- Affinché la logica del Proxy esegua al momento giusto, le classi client devono interagire con i Proxy, anziché i RealSubject. Se le classi clienti usano direttamente la classe RealSubject bypassano il Proxy

```
Subject s = new Proxy();
```

- Nel caso del Reference Monitor, come assicurare che ciascuna chiamata arrivi al ProtectionObject passando prima dal ReferenceMonitor?
- Se occorresse la funzionalità offerta da un Proxy su varie classi occorrerebbe implementare una classe Proxy per ciascuna di tali classi, e si avrebbe una proliferazione di classi

Considerazioni Su Object-Oriented

- Tramite le tecniche Object-Oriented, l'implementazione di una funzionalità (per es. logging) può risultare sparsa tra varie classi
  - Es., all'interno di vari metodi di varie classi sono presenti istruzioni del tipo:

```
log.writeLog("Changed from "+oldval+" to "+newval);
```

- Ovvero, stabilita una certa decomposizione in classi di un sistema, alcune "parti" (concern) non possono che essere implementate tramite codice sparso su varie classi e non come un'unica classe
  - Un concern è un insieme di funzionalità del sistema
- Si definisce questa impossibilità come tirannia della decomposizione predominante

Concern

- Un sistema software realizza vari requisiti (funzionalità)
- Nella fase di progettazione, i vari requisiti sono partizionati e si identificano i moduli che devono corrispondere ad essi
  - Es. per gestione ordini: accounting, processing, presentazione risultati
  - Più in generale: logging, business logic, persistenza, sicurezza, etc.
- 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

Case Study Su Concern

- Studi fatti su Apache Tomcat hanno mostrato che
  - 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 sparso (o scattered) su tutte/tante classi
- Si dice che questi concern sono trasversali (o crosscutting)
- I corrispondenti moduli presentano tangling, ovvero vari concern sono mischiati all'interno dello stesso modulo

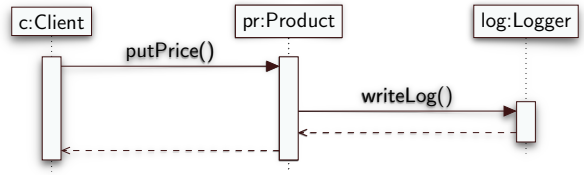
Esempio

- Realizzare un sistema software che sia in grado di
  - 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) {
    // ...
  }
}
```


Classe Product

```
// Classe che tiene informazioni sui prodotti
public class Product {
  private double price;
  private Logger log;        // codice per Log
  public Product() {
    log = new Logger();      // codice per Log
    price = 0.0;
  }
  public void putPrice(double p) {
    log.writeLog("Price changed from "+price+" to "+p); // codice per Log
    price = p;
  }
  public double getPrice() {
    return price;
  }
}
```



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`
- Per soddisfare il requisito sulla registrazione, si mischia (tangling) il codice di `Logging` e il codice di `Product`
- Per evitare il tangling, la classe `Product` non dovrebbe avere invocazioni a metodi di `Logger`
- Questo è quello che avviene con OOP

Soluzione Aspect-Oriented

- `Product` dovrebbe essere priva di qualunque interazione con `Logger`. Il metodo di `Logger` dovrebbe essere chiamato per ciascun metodo di `Product` la cui esecuzione deve essere registrata
- La soluzione è possibile tramite AOP
- 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*
- Un aspetto è un modulo che consiste di advice e pointcut ed è collegato a certi punti di una classe chiamati join point

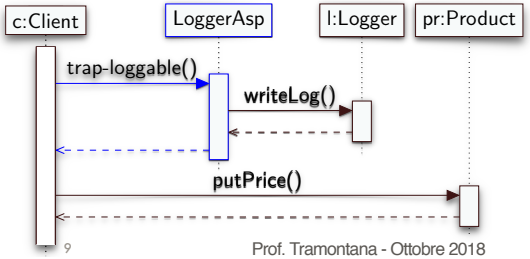
Aspetto LoggerAsp

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

  // definisce il pointcut, ovvero i punti da catturare (intercettare)
  pointcut loggable() : call(public void Product.putPrice(*));

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

```
public class Product {
  private double price = 0.0;
  public void putPrice(double p) {
    price = p;
  }
  public double getPrice() {
    return price;
  }
}
```



Altri Concern

- Altri requisiti (concern) potrebbero essere aggiunti, es.
  - Registrare il tempo necessario alla variazione del costo
  - Autenticare la richiesta di variazione (access control)
  - Contare il numero di prodotti restanti
- Si avrebbero altri concern nella stessa classe, che si mischiano fra loro. Oppure, altre classi (per es. sottoclassi) che implementano le parti aggiuntive, quindi: esplosione del numero di classi, ripetizioni di codice, concern sparso su più classi
- Il codice di `Product` si complica e diventa difficile da comprendere, più soggetto a contenere errori, più difficile da spiegare e cambiare, non riusabile
- In generale, i concern crosscutting soffrono dei suddetti problemi, se sono implementati con la programmazione OO

Join Point

- Uno join point è un punto all'interno dell'esecuzione di un programma, es.
  - Invocazione (chiamata) metodo
  - Esecuzione metodo
  - Costruzione oggetto (chiamata a new)
  - Accesso a campo (lettura e scrittura)
  - Eccezione
  - Test condizionali, etc.
- Uno join point è il punto dove interviene il crosscutting concern, ovvero, dove interviene un aspetto

Pointcut

- Un pointcut è il costrutto che seleziona i join point e colleziona il contesto di esecuzione per i join point selezionati
- Es. di pointcut
  - L'esecuzione del metodo `putPrice()` della classe `Product`
  - L'esecuzione dei costruttori che prendono un argomento `int`
- Ovvero
  - Un pointcut specifica le regole di selezione
  - Un join point soddisfa le regole specificate

Advice

- Un advice è il codice che è eseguito quando un join point è selezionato da un pointcut
- Il codice di un advice è simile a quello di un metodo: incapsula la logica da eseguire quando si raggiunge un join point
- Il codice di un advice può essere eseguito prima/dopo/al posto del join point raggiunto e indicato dal pointcut

Esempio

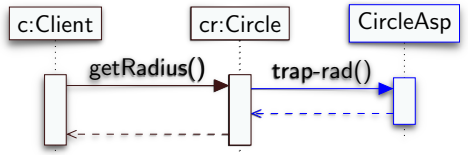
- Esempio di join point: esecuzione del metodo `getRadius()`
- In un aspetto `CircleAsp`, un pointcut `rad()` cattura l'esecuzione del metodo `getRadius()` della classe `Circle`

```
pointcut rad() : execution(public int Circle.getRadius());
```

- Un advice può essere eseguito prima, dopo, o al posto, (before, after, around) del join point corrispondente al pointcut specificato
- Advice che esegue prima del join point, quindi prima dell'esecuzione del metodo `getRadius()`

```
before() : rad() {
  System.out.println("prima del metodo getRadius()");
}
```

```
public class Circle {
  private int radius = 5;
  public int getRadius() {
     return radius;
  }
}
```



Tipi Di Pointcut

```
call(double Product.getPrice())
```

- `call` seleziona il punto del programma in cui vi è la chiamata al metodo `getPrice` su un'istanza di `Product` (dal lato chiamante)

```
execution(double Product.getPrice())
```

- `execution` seleziona il punto del programma in cui inizia l'esecuzione del metodo `getPrice` su un'istanza di `Product` (dal lato chiamato)
- Spesso sono equivalenti, ma se bisogna intercettare operazioni su classi di libreria non modificabili, si può usare solo `call`

Weaver

- Un weaver (compilatore AspectJ) è usato per fondere insieme aspetti e classi e generare una applicazione che contiene solo costrutti tradizionali
- Elimina i costrutti del linguaggio per aspetti
- Genera un codice sorgente intermedio (senza aspetti) che viene elaborato da un normale compilatore
- Il bytecode prodotto è eseguibile da una JVM standard
- Sito per AspectJ: eclipse.org/aspectj
  - scaricare il jar e installare
  - compilare programmi AOP con: ajc <lista file java>

Esempio

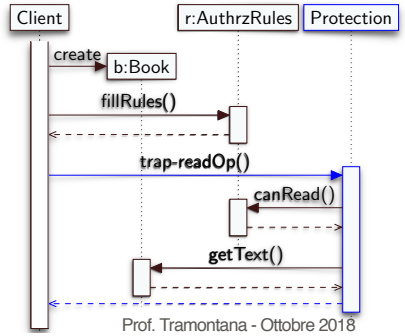
- Vogliamo che gli accessi ai metodi di una classe siano controllati
- Es. la classe Book ha metodi i cui accessi sono da controllare
- La classe `Client` fa chiamate alla classe Book
- La classe `AuthrzRules` tiene le regole di accesso
- L'aspetto `Protection` interviene prima di accedere alla classe Book
  - L'aspetto implementa il ruolo Proxy
  - Lo stesso aspetto può svolgere il ruolo di Proxy per tante classi


```
public class Client {
  private static AuthrzRules r = new AuthrzRules();
  public static void main(String[] args) {
    Book b = new Book(); // notare che si usa l'istanza di Book
    r.fillRules();
    System.out.println("Client: " + b.getText());
    b.append("Hello world.");
  }
  public static AuthrzRules getRules() {
    return r;
  }
}
public class AuthrzRules {
  private List<String> rperm = new ArrayList<String>();
  private List<String> wperm = new ArrayList<String>();
  public boolean canRead(Object o) {
    if (rperm.contains(o.getClass().getName())) return true;
    return false;
  }
  public boolean canWrite(Object o) {
    if (wperm.contains(o.getClass().getName())) return true;
    return false;
  }
  public void fillRules() {
    rperm.add("Book");
  }
}
```



```
public aspect Protection { // Sta facendo da Proxy per la classe Book
  private AuthrzRules r = Client.getRules();
  pointcut readOp(Book b) : call(String Book.getText()) && target(b);
  pointcut writeOp(Book b) : call(void Book.append(String)) && target(b);
  String around(Book b) : readOp(b) {
    System.out.println("Aspect: before read");
    if (r.canRead(b)) return proceed(b);
    System.out.println("Aspect: Read access has not been granted");
    return null;
  }
  void around(Book b) : writeOp(b) {
    System.out.println("Aspect: before write");
    if (r.canWrite(b)) proceed(b);
    else System.out.println("Aspect: Write access has not been granted");
  }
}
```

```
Output:
Aspect: before read
Client: It was a bright cold day in April,
Aspect: before write
Aspect: Write access has not been granted
```



Collezione Del Contesto

- Per collezionare il contesto di esecuzione, ovvero, in questo caso, il riferimento dell'oggetto chiamato, con b di tipo Book

`call(String Book.getText()) && target(b)`

- Seleziona tutti i join point per i quali l'oggetto su cui avviene la chiamata è un'istanza di Book o di una sua sottoclasse
- b conterrà il riferimento all'istanza di Book
- Tale riferimento è quindi passato all'advice corrispondente