Riflessione computazionale

- AspectJ fornisce supporto alla riflessione, ad esempio
  - Per accedere al nome del metodo catturato
  - Per accedere agli argomenti passati al metodo
    - Si può accedere anche usando `args()`
  - Per avere il valore dell'istanza su cui è catturato il metodo
    - Si può accedere anche usando `target()`
- Considerazioni sull'uso della riflessione
  - Prestazioni minori per accedere al valore dell'istanza rispetto all'uso di `target()`
  - Non vi è alcun controllo di tipo
  - Il codice prodotto può essere poco chiaro
- Alcune volte è inevitabile usare la riflessione
  - Es. utile per catturare argomenti di metodi diversi con parametri diversi in numero e tipo

Oggetti this

- Su ogni advice sono disponibili gli oggetti
  - `thisJoinPoint`
  - `thisJoinPointStaticPart`
  - `thisEnclosingJoinPointStaticPart`
- Come `this`, permettono di accedere all'oggetto in esecuzione
- L'informazione contenuta nei tre oggetti può essere
  - Dinamica: si riferisce alla particolare invocazione, es. istanza e argomenti
  - Statica: non cambia al cambiare delle invocazioni e disponibile a design time, es. nome e classe del metodo invocato

`thisJoinPoint`

- `thisJoinPoint` è un oggetto di tipo `JoinPoint`
  - Fornisce accesso a informazioni dinamiche: oggetto target e argomenti
  - Permette di accedere a informazioni statiche tramite il metodo `getStaticPart()`
- `thisJoinPointStaticPart` è un oggetto di tipo `JoinPoint.StaticPart`
  - Fornisce accesso a: tipo di pointcut (`call`, `execution`, etc.), signature del join point
- `thisEnclosingJoinPointStaticPart` è un oggetto di tipo `JoinPoint.StaticPart`
  - Fornisce informazioni sul contesto che racchiude il join point, es. per una chiamata di metodo è il contesto del metodo chiamante

`JoinPoint`

- L'interfaccia `JoinPoint` fornisce i metodi
  - `getThis()` ritorna l'oggetto che sta eseguendo
    - Ritorna null se il join point è all'interno di un metodo statico
  - `getTarget()` ritorna l'oggetto target del join point
    - Ritorna null per chiamate a metodi statici
  - `getArgs()` ritorna un array in cui ogni elemento è un riferimento ad un argomento, nell'ordine in cui sono passati
    - Per join point `set()` il valore da assegnare al campo è disponibile tramite `getArgs()`
    - Per join point `get()` l'array sarà vuoto
  - `getKind()`, `getSignature()`, `getSourceLocation()` restituiscono informazioni statiche

`JoinPoint.StaticPart`

- A partire da `thisJoinPoint` possiamo ottenere una istanza di `JoinPoint.StaticPart` tramite `getStaticPart()`
- L'interfaccia `JoinPoint.StaticPart` fornisce
  - `getKind()` restituisce il tipo (kind) di join point tramite stringa, es. "method-call", "method-execution", o "field-set"
  - `getSignature()` restituisce una istanza di `Signature` per il join point in esecuzione
    - `Signature` permette di accedere a nome, tipo, etc
  - `getSourceLocation()` restituisce una istanza di `SourceLocation`
    - `SourceLocation` contiene informazioni su nome file, numero linea, etc.
- Entrambi `thisJoinPoint.getStaticPart()` e `thisJoinPointStaticPart` permettono di accedere alle stesse informazioni

Codice

```
before() : trace() {
  System.out.println(" * signature: "+thisJoinPoint.getSignature());
  System.out.print(" * this: "+thisJoinPoint.getThis());
  System.out.println(" * target: "+thisJoinPoint.getTarget());

  StringBuffer str = new StringBuffer(" * args: ");
  Object[] args = thisJoinPoint.getArgs();
  for (int length = args.length, i = 0; i < length; ++i) {
    str.append(" [" + i + "] = " + args[i]);
  }
  System.out.println(str);

  System.out.println(" * kind: "+thisJoinPoint.getKind());
  System.out.println(" * source:
"+thisJoinPoint.getSourceLocation());
}
```


Policy Enforcement

- Policy enforcement (far rispettare politiche) è il meccanismo per assicurare che le classi del sistema rispettino certe pratiche
  - Es. certe chiamate a metodi dovrebbero essere sostituite con altre chiamate per esigenze di efficienza, prestazioni, confinement, etc.

```
public aspect Detect {

   declare warning :
      call(void Logger.log(..)) :
         ”Sostituire con la chiamata a log2()";
}
```


Policy Enforcement

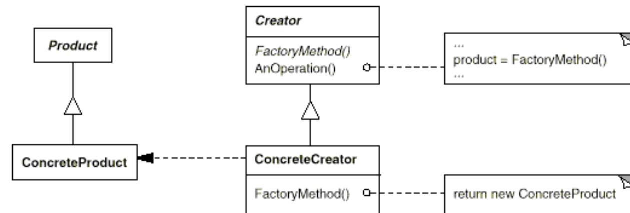
- Policy enforcement
  - Es. solo certe classi dovrebbero chiamare certi metodi di altre classi

```
public aspect Detect {

   declare error :
      (call(void ShoppingList.add(..))     ||
       call(void ShoppingList.remove(..))  ||
       call(void ReadingList.add(..))      ||
       call(void ReadingList.remove(..)))  &&
       !within(Management) :
        "Gestione errata delle operazioni di inserimento o rimozione";
}
```

Factory Method

- Intento
  - Definire una interfaccia per creare un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare
- Soluzione
  - *Product* è l'interfaccia comune degli oggetti che il FactoryMethod() crea
  - ConcreteProduct è una implementazione dell'interfaccia *Product*
  - *Creator* è l'interfaccia che dichiara il FactoryMethod() e ritorna un oggetto di tipo *Product*
  - ConcreteCreator implementa il FactoryMethod() scegliendo quale ConcreteProduct istanziare e ritorna tale istanza



Factory Method

```java
interface Shape {  //Product
  void draw();
  void fill();
}
interface ShapeCreator {  //Creator
  public Shape getShape(); //Factory Method
}
//ConcreteCreator
class CreatorA implements ShapeCreator {
  public Shape getShape() {
    //solo qui indico la classe da istanz
    return new Circle();
  }
}
class Circle implements Shape { //ConcretProd
  public void draw() {
    System.out.println("A circle ( )");
  }
  public void fill() {
    System.out.println("Filled circle (o)");
  }
}
```

```java
class Square implements Shape { //ConcretProd
  public void draw() {
    System.out.println("A Square [ ]");
  }
  public void fill() {
    System.out.println("Filled Square [X]");
  }
}
public class ShapeCreatorTest {
  public static void main(String args[]) {
    //istanzio il Concrete Creator
    ShapeCreator sc = new CreatorA();
    //ottengo una istanza di una
    //sottoclasse di Shape, non
    //decido qui quale sottoclasse
    Shape s = sc.getShape();

    s.draw();    s.fill();

    s = new Circle();
  }
}
```


Policy Enforcement

- Quando il design pattern factory method è usato
  - Dovremmo assicurarci che solo certe classi siano abilitate ad istanziare altre classi

```java
public aspect Detect {

  declare error : call(Product.new(..)) && !within(ProductFactory+) :
      "Solo ProductFactory può creare istanze di Product";
}
```

```java
public aspect Detect {

  declare error : call(Shape.new(..)) && !within(ShapeCreator+) :
     "Solo ShapeCreator può creare istanze delle sottoclassi di Shape";
}
```