Design Pattern

Capitolo 10, sezione 10.4 esercizi 10.3 e altri

Cosa sono i design pattern

- Quando si e' cominciato a lavorare seriamente con i linguaggi ad oggetto, la gente si e' resa conto che si presentavano ad ogni programma, dei problemi ricorrenti.
- Un gruppo di 4 programmatori (la banda dei 4, GoF(Gang of Four) ha cercato di formulare una lista dei 23 **problemi piu ricorrenti (e delle loro soluzioni)** e cosi' nel 1994 sono nati i Design Patterns.

Idea degli architetti

- L'idea di design pattern si deve agli architetti (vedi libro di Alexander, 1977)
- Vedi libro

Quali design pattern vedremo

- Singleton
- Facade
- Visitor
- MVC a informatica III B
-
- Materiale: su wikipedia o su bruce Eckel

Singleton 10.4 (pag 291)

- Pattern *creazionale*
- A single instance of the class
- Quando ho bisogno di una unica istanza di una classe

- In Java
  - private constructor
  - static member

```
class A{
   private A(){...}
   public static A instance = new A();

}
```

Esercizio

- Implementata Math come Singleton invece che usare metodi statici
- Diverse varianti:
  - Creato subito
  - Accesso all'istanza con metodo
    - Creazione quando richiesta

Facade

- Structural pattern
- Raggruppo in un singolo oggetto più oggetti di classi distinte e fornisco un accesso a più alto livello agli oggetti sottostanti
- In genere poi passo le richieste agli oggetti sottostanti.
- Il client usa solo la facade

Esempio

Visitor Pattern es. 10.3

Synopsis

- Represent an **operation** to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Visitor Pattern

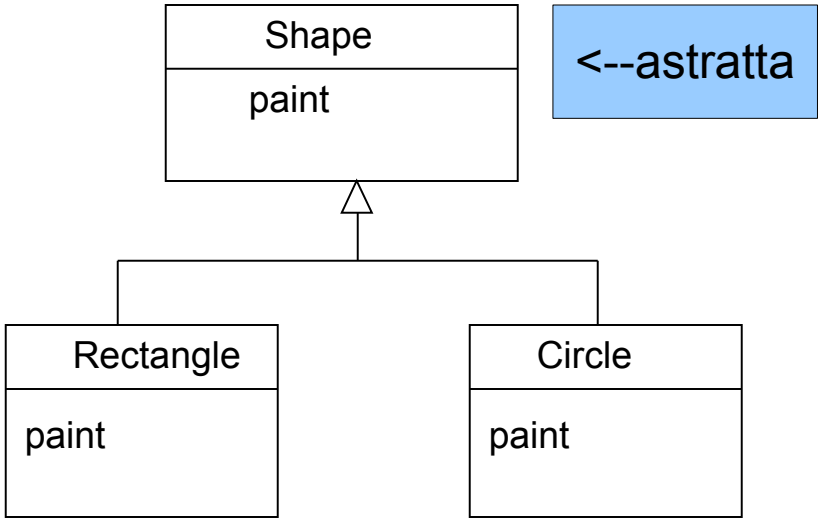
- Problem
  - Operations on collections of objects may not apply to all objects, or apply differently to different objects
- Context
  - Object interfaces are fixed and diverse
  - Need to allow new operations, without polluting” their classes with these operations.
- Solution
  - Represent operations to be performed as visitors, with the interface of every visitor representing the different kinds of objects

Context

- You should use the Visitor pattern when:
  - Many distinct and unrelated operations need to be performed on an object structure, and you want to avoid “polluting” their classes with these operations.
  - The classes defining the object structure rarely change, but you often want to define new operations over the structure.
  - An object contains many classes of objects with differing interfaces.

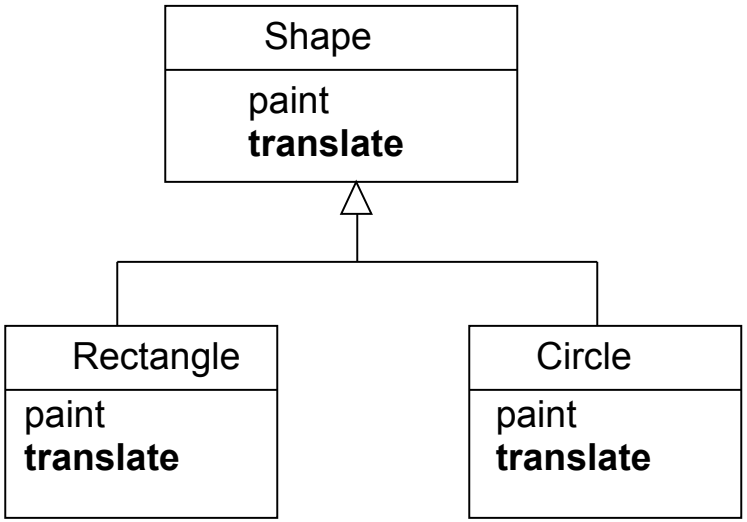
Example: problem

- Imagine you have a program that deals with geometric figures: rectangles and circles
- You want to add an **operation**, for example
  - You want to **translate** of shape



Possibile soluzione 1

- aggiungo l'operazione ad ogni classe:



CONTRO:
- Devo modificare le classi originali
- Se ho 10 operazioni devo modificare le classi
- Ho il codice sparso in tutte le classi

Soluzione 2

- Creo una classe (singleton) che rappresenta l'operazione:

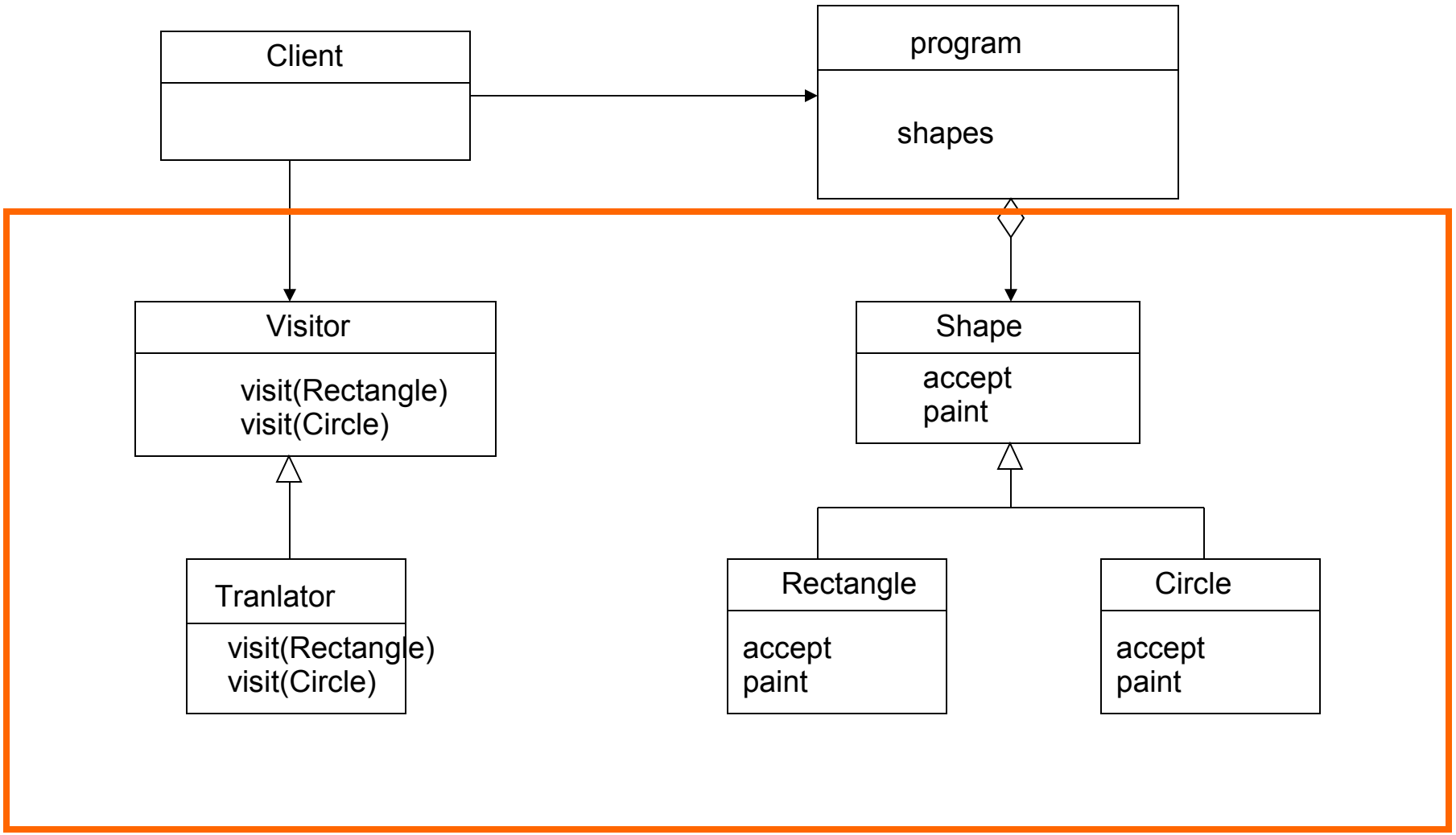
```
Class Translate{
   process(Shape s){...}
   process(Rectangle e){...}
   process(Circle e){...}
   }
   ******
```

- Devo stare attento al single dispatch. Esempio
- Shape e = new Circle()
- `Translate`.getInstance().process(e) ---
  - Va prender process(shape) quale codice eseguo??
- Dovrei metter tanti **instanceof** ...

Soluzione - visitor

- The best way to do this is to have a (generico) visitor come in and performs the operation
- Ogni classe è in grado di accettare il visitor
- L'operatore/visitatore farà l'operazione translate quando visita l'oggetto

Visitor Example



lato Visitable

- Le classi della gerarchia devono essere visitabili:

```
interface Visitable {
  void accept(Visitor v);
}
```

- ogni classe originale deve implementare visitable (un metodo accept):

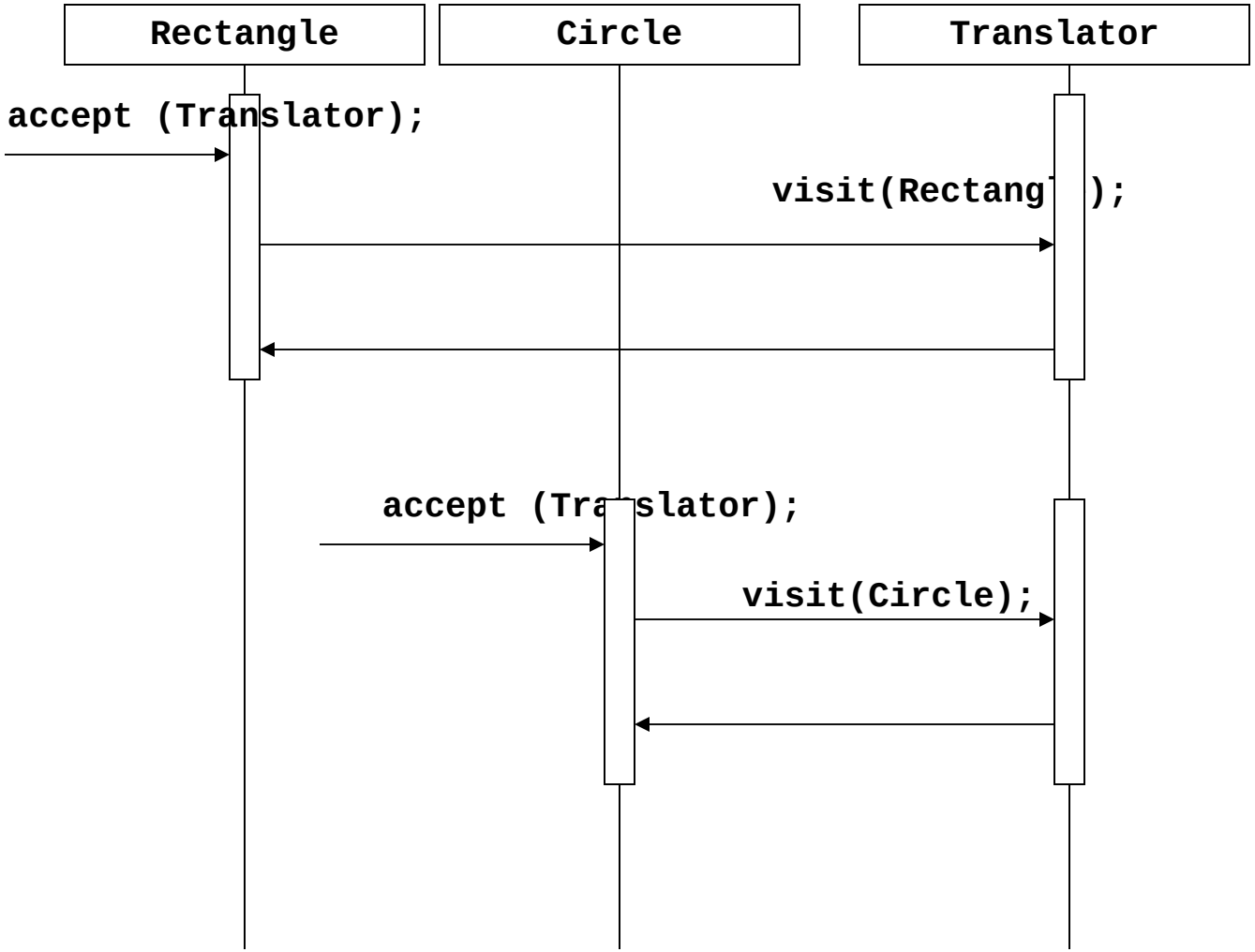
```
class Shape implements Visitable{
     abstract accept(Visitor v);
}
class Rectangle extends Shape{
     accept(Visitor v){ v.visit(this);}
}
```

lato visitor

- Translator deve essere un visitor

```
public interface Visitor {
    public void visit(Rectangle m);
    public void visit(Circle m);
}
public class Translator implements Visitor{
    public void visit(Rectangle m){
      System.out.println("translation of r");
     }
    public void visit(Circle m){
      System.out.println("tarnslation of c");
   }
```

Visitor Interactions



Cosa succede con il single dispatch

- Shape e = new Circle();
- Visitor v = new Translator();
- e.accept(v)

-> va a prendere l'accept di Circle !!!

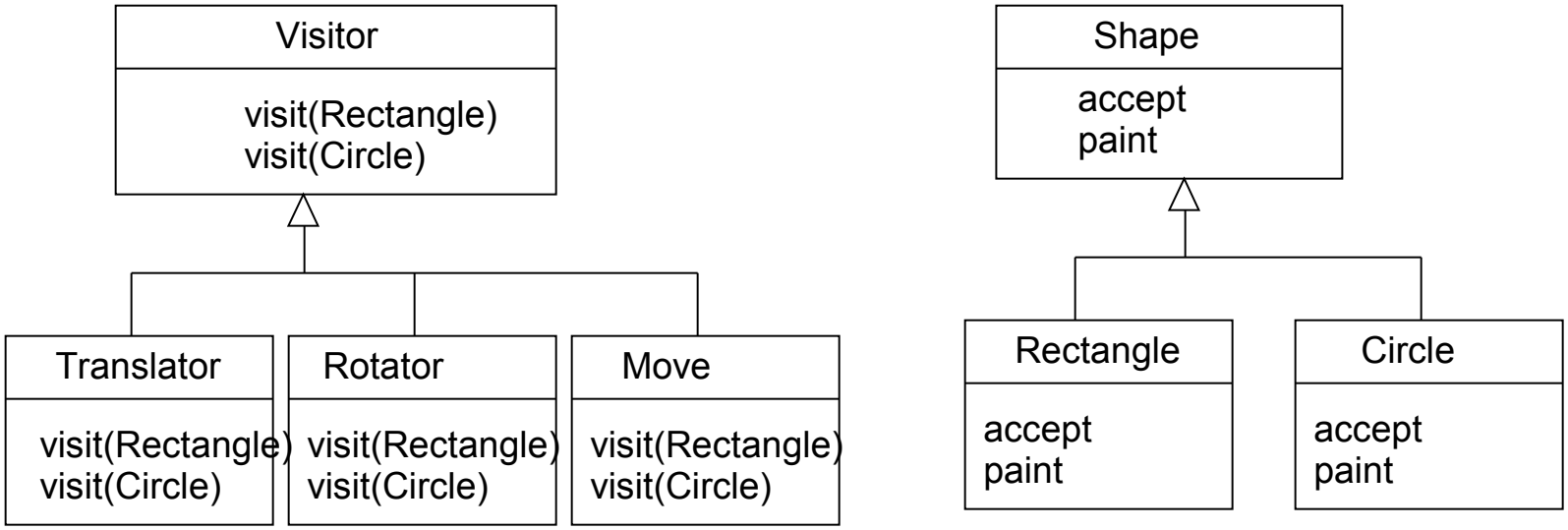
Chimato anche “double dispatch”

Visitor Example

- Se volessi aggiungere una nuova operazione, come ad esempio la rotazione?
- Definisco un nuovo visitor senza modificare le classi figure

BASTA AGGIUNGERE UN ALTRO VISITOR

Visitor Example



Esercizi

- Esercizio 10.3 con Expression
- prova le tre soluzioni
  - Con metodi nelle classi
  - Con classe esterna
  - Con visitor

Consequences

- Positive
  - Visitor makes adding new operations easy, simply add a new visitor that implements that operation.
  - Visitor gathers related operations and separates unrelated ones.

Consequences

- Negative
  - Visitor is not good for the situation where "visited" classes are not stable. Every time a new Composite hierarchy derived class is added, every Visitor derived class must be amended
  - Often encapsulation is broken because the element class is forced to provide public operations that access internal state.
  - If using an existing system changes will be required to existing code.

Related Patterns

- Iterator
  - The iterator pattern is an alternative to the Visitor pattern when the object structure to be navigated has a linear structure.
- Composite
  - The visitor pattern is often used with object structures that are organized according to the composite pattern.

Come fare restituire un valore da un visitor

Normalmente il visitor non restituisce niente. E se devo calcolare qualcosa: come fare? due alternative

1. modifica di visit
   - definire i metodi visit che restutiscano un valore
   - ad esempio restituiscano un Object
     - `Object visit(X…);`
   - poi faccio il cast sapendo cosa effettivamente restituisce
2. aggiungere un campo e un metodo
   - campo result, che viene settato alla fine della visita
   - getResult che restituisce il risultato della visita

visitor e generics

- L'alternativa è dichiarare il Visitor generico rispetto il tipo che restituisce:

```
public interface Visitor <T> {
    public T visit(Rectangle m);
    public T visit(Circle e);
}
// if the visitor returns a String
public class GetInfo implements Visitor<String>{
    public String visit(Rectangle m){
        return "rettangolo");   }

....
```

Generic Visitable

- And a generic Visitable with a generic method

```
interface Visitable{
 public <T> T accept(Visitor<T> ask);
}
class Circle implements Visitable{
 public <T> T accept(Visitor<T> ask){
        return ask.visit(this);
 }
} ...
```

Visitor pattern e single dispatch

- Alcune volte si vuole ovviare al problema del single dispatch con il visitor pattern
- Realizza quello che si dice un “double dispatch”

Reflection

- Nota che in Java si può usare la reflection per implementare il double dispatch invece che l'uso del pattern !!!

Idee per il linguaggi diversi

- Multijava

  http://multijava.sourceforge.net/

  MultiJava is an extension to the Java programming language that adds open classes and symmetric multiple dispatch.

- Nice
  - http://nice.sourceforge.net/index.html