# Principi di Design ...e qualche Pattern

Ugo Landini
Senior Java Architect
Sun Microsystems

# Costo del software



- Il costo del software e' influenzato dai cambiamenti nel software stesso. Il software non si usura come l'hw, ma si deteriora.
- Due possibili approcci metodologici
  - Prevedere i cambiamenti (Plan Driven)
  - Rendere i cambiamenti parte del processo di produzione (Agile)
- I due approcci si avvalgono soprattutto di:
  - Design Patterns
  - Refactoring

# Principi di design



Sia i Design Patterns che il refactoring trovano le loro radici comuni in alcuni principi fondamentali:

- **OCP** Open Closed Principle
- **DIP** Dependency Inversion Principle
- **LSP** Liskov Substitution Principle
- **SRP** Single Responsibility Principle

# OCP

- Open/Closed Principle
  - Bertrand Meyer, 1988
- Un modulo deve essere aperto e chiuso.
  - Aperto alle estensioni
  - Chiuso alle modifiche
- Deve essere dunque possibile modificare il comportamento di una classe senza toccare il codice esistente
- E' la strategia da seguire

# LSP



- Liskov Substitution Principle
  - Barbara Liskov, 1988
- Tipi e sottotipi devono essere sostituibili
- Un metodo che usi un oggetto di classe B deve essere in grado di usare un oggetto di classe D, derivata da B, senza modifiche
- Rende possibile **OCP**

# DIP



- Dependency Inversion Principle
  - Robert C. Martin, 1996
- I Moduli di alto livello non devono dipendere dai moduli di basso livello
- Le astrazioni non devono dipendere dai dettagli: i dettagli devono dipendere dalle astrazioni
- E' il meccanismo per l'utilizzo corretto di **LSP** al fine di raggiungere l'**OCP**
- Ultimamente va molto di moda, è il principio guida di Spring, Hivemind, Picocontainer

# SRP



- Single Responsibility Principle
  - Robert C. Martin, 1996
- Una classe dovrebbe avere un solo motivo per cambiare
- Altrimenti noto come “alta coesione”
- Ci aiuta nell'assegnazione di responsabilità ad altri oggetti (nuovi o preesistenti)
- Contribuisce a rendere possibile **OCP**

# Basta paroloni!



- Fine della teoria!
- Facciamo un po' di esempi pratici, analizzando delle violazioni dei principi, ossia dei design **SPORCHI, BRUTTI, CATTIVI.** Le violazioni vanno dalle evidenti alle sottili
- **ATTENZIONE**: nelle prossime presentazioni si faranno domande sui principi esposti che vi permetteranno di vincere favolosi gadget!

# Un Design OK?



```
public class Impiegato {

  public void lavora () {
    ...
    v = new Vector();
    v.add("task1");
    ...
  }

  public Vector v;

}
```

# Un Design OK?



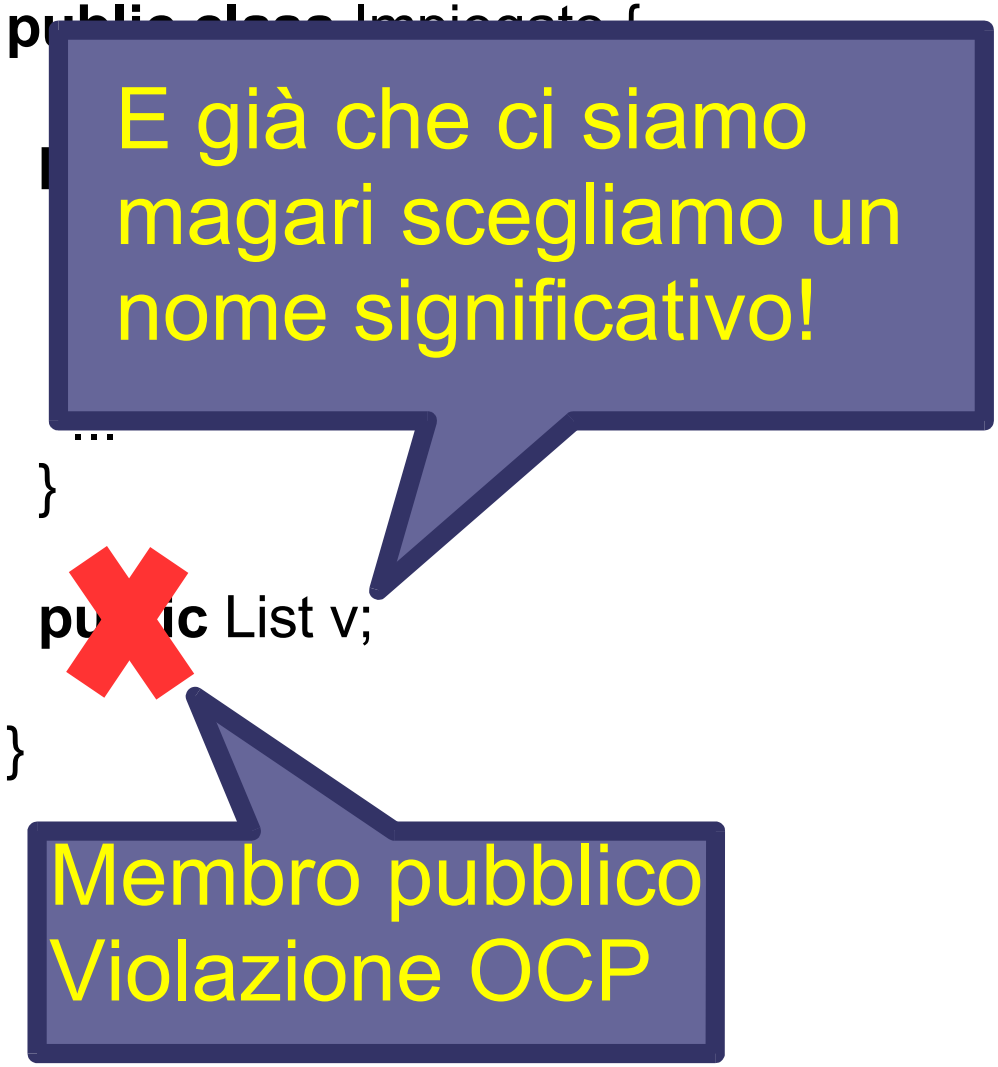
```
public cla

  public v
    ...
    v = new Vector();
    v.add("task1");
    ...
  }

  public Ve or v;

}
```

Fra l'altro Vector è una legacy collection

Classe concreta
Violazione DIP

# Un Design OK?




E già che ci siamo magari scegliamo un nome significativo!
}
public List v;
}
Membro pubblico
Violazione OCP

# Un Design OK

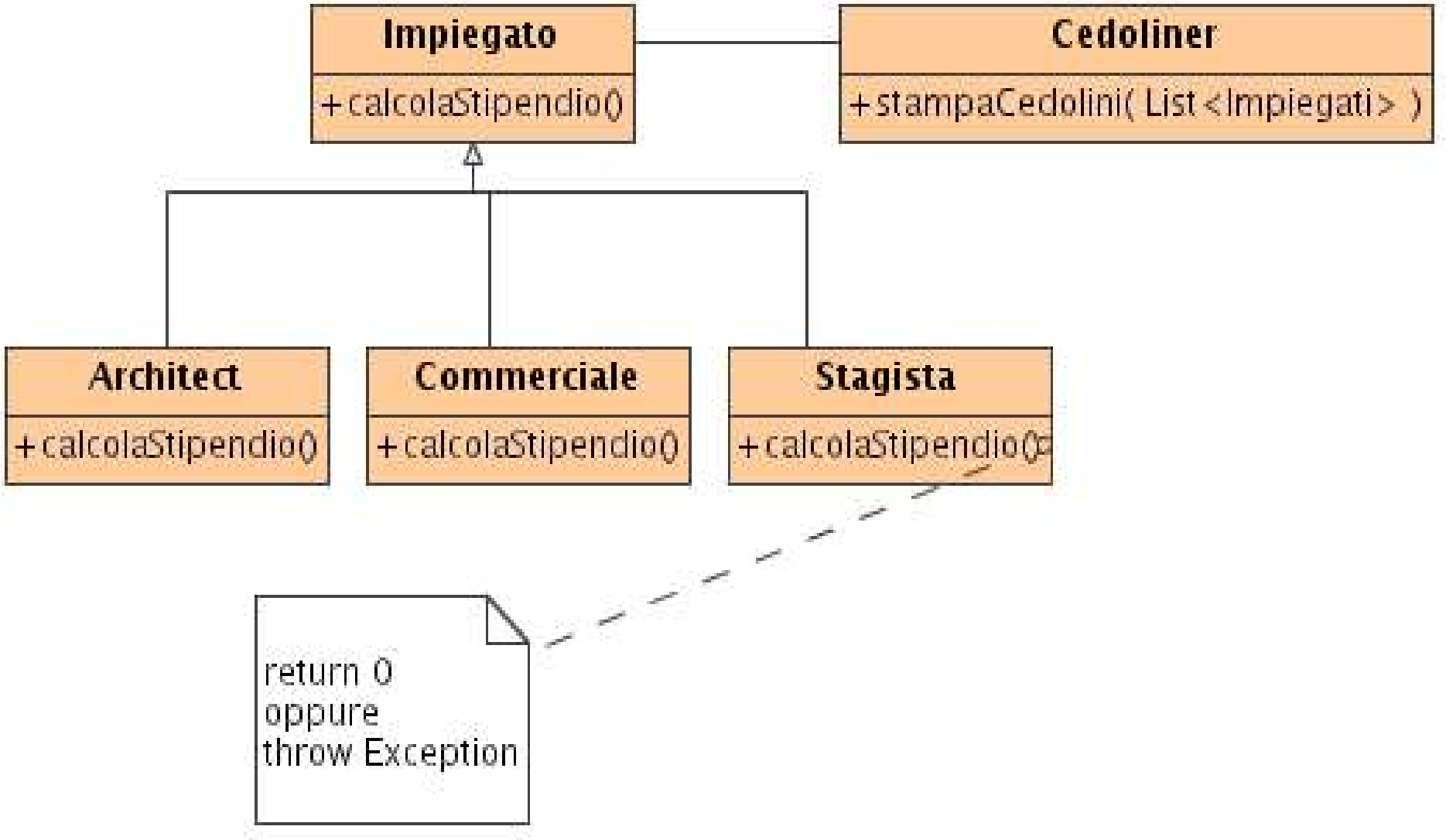


```
public class Impiegato {

  public void lavora () {
    ...
    tasks = new ArrayList();
    tasks.add("task1");
    ...
  }

  private List task

}
```

Unico accoppiamento fra Impiegato e ArrayList, come si può togliere?

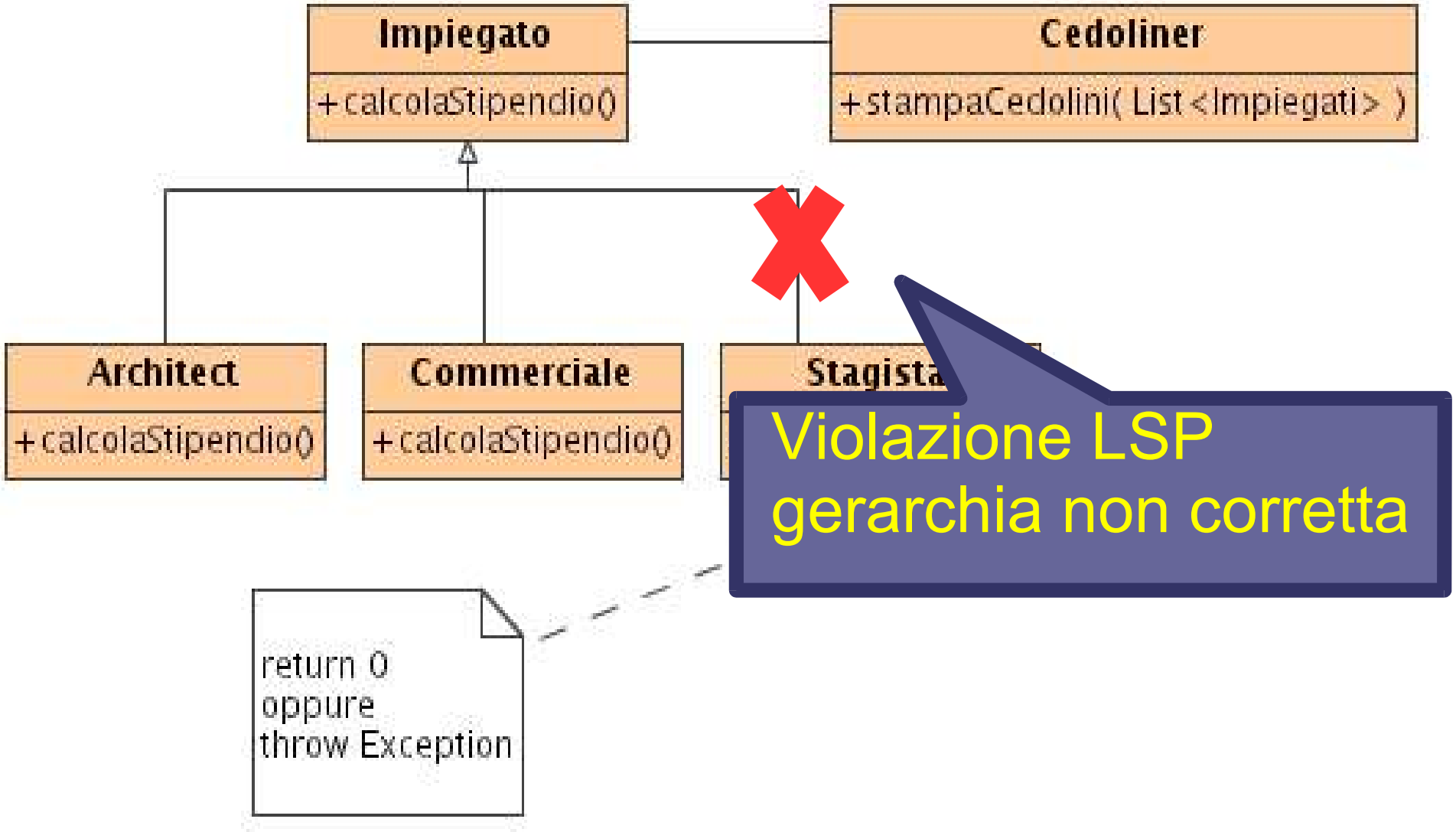
# Un Design OK?




Impiegato
+calcolaStipendio()
Cedoliner
+stampaCedolini( List<Impiegati> )
Architect
+calcolaStipendio()
Commerciale
+calcolaStipendio()
Stagista
+calcolaStipendio()
return 0
oppure
throw Exception

# Violazione di LSP





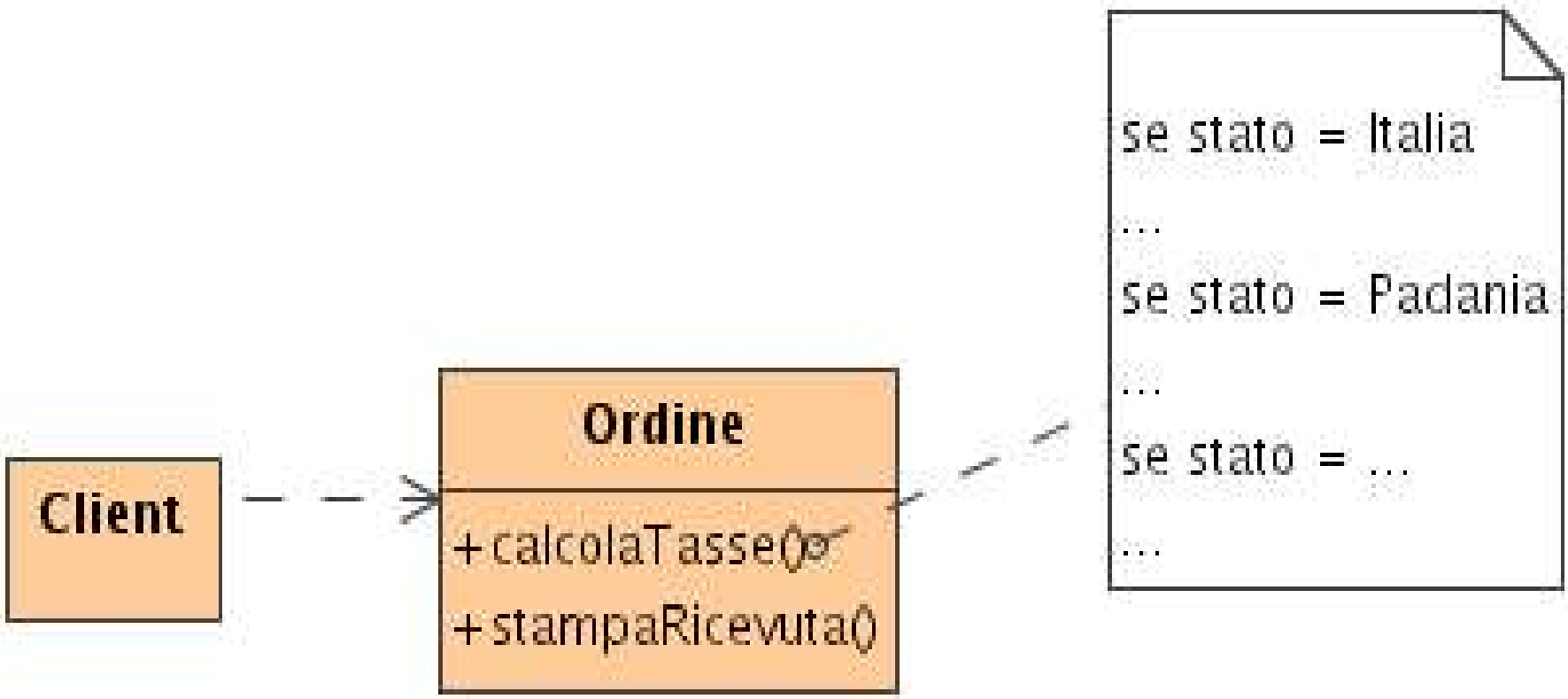
# Un Design OK?



- *Stagista* non è sostituibile a *Impiegato*
- Nel caso si ritorni zero per *calcolaStipendio*, lo stagista riceverebbe un cedolino (errato), a meno di non gestire l'eccezionalità nelle classi deputate (*if o instanceof Stagista*???)
- Nel caso si utilizzi invece l'eccezione, dovremmo inquinare il codice con blocchi *try* o, peggio, cedere alla tentazione dell'*instanceof*

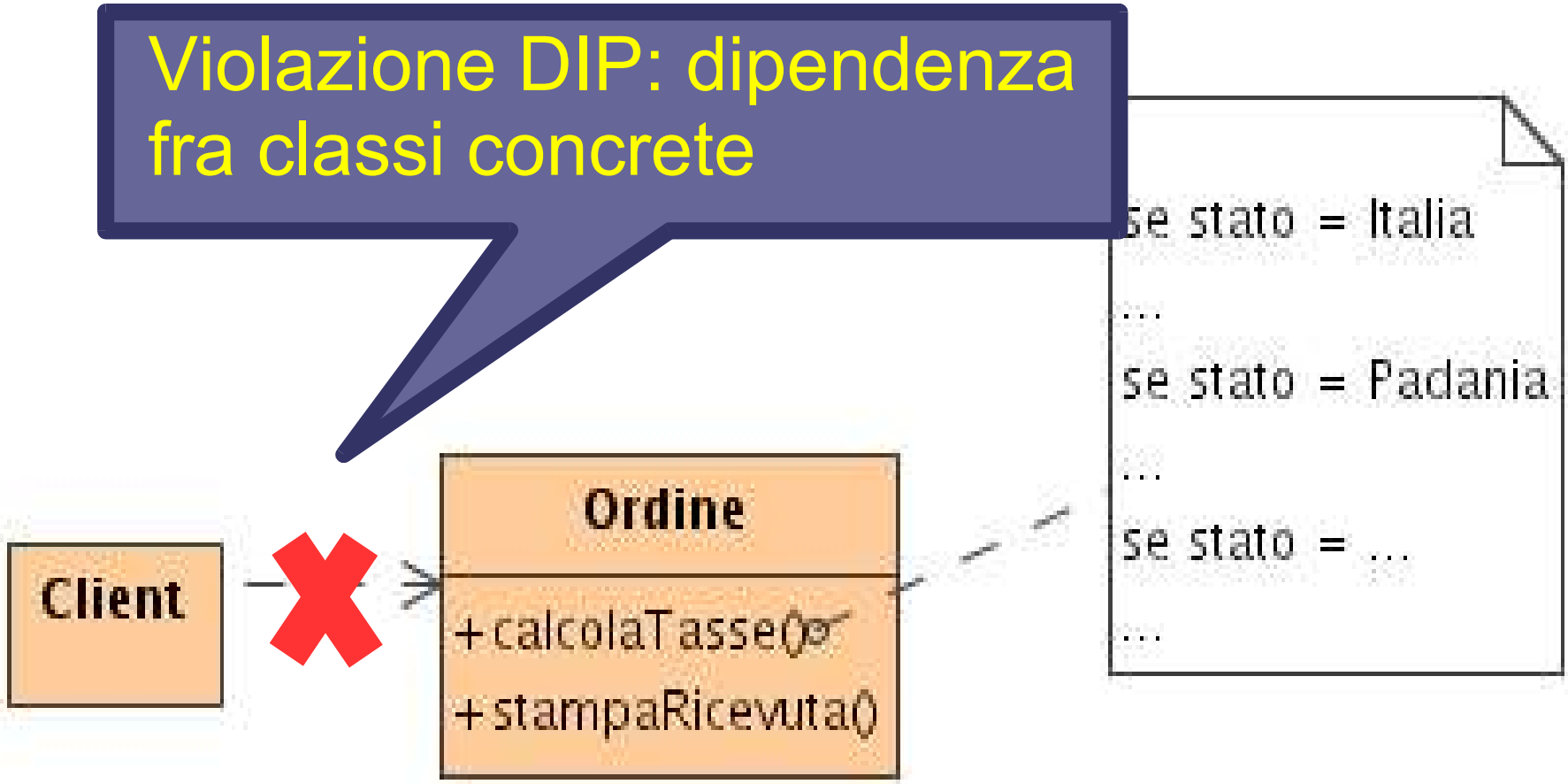
# Un Design OK?




Client
Ordine
+calcolaTasse()
+stampaRicevuta()
se stato = Italia
...
se stato = Padania
...
se stato = ...
...

# Un Design OK?




Violazione DIP: dipendenza fra classi concrete
Client
Ordine
+calcolaTasse()
+stampaRicevuta()
se stato = Italia
...
se stato = Padania
...
se stato = ...
...

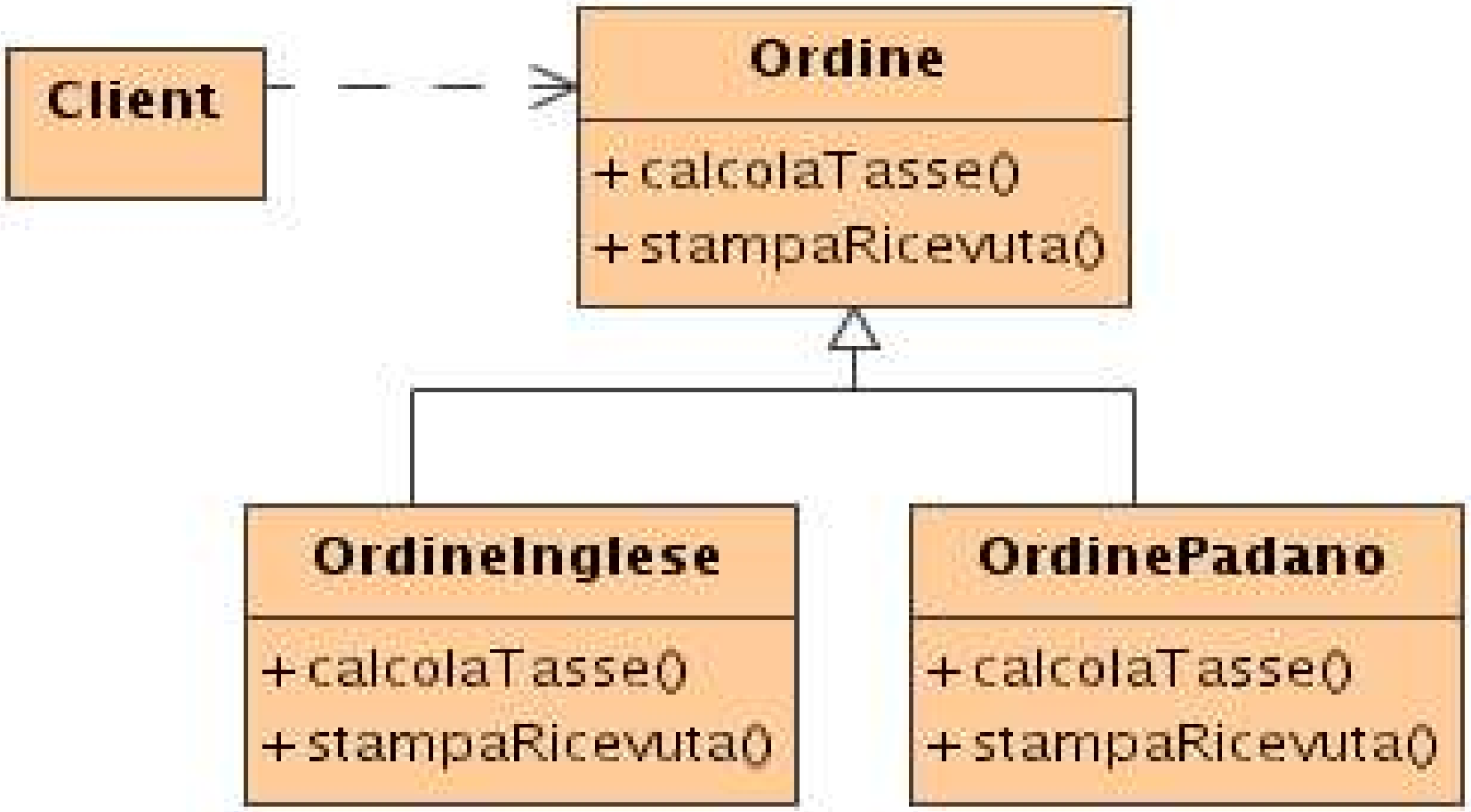
# Un Design OK?



- Violazione di DIP
  - Dipendenza fra classi concrete
- *Ordine* è fragile, non è assolutamente aperto alle estensioni e chiuso alle modifiche
  - Assenza totale di chiusure strategiche (*Strategic closures*)

# Un Design OK?




Client
Ordine
+calcolaTasse()
+stampaRicevuta()
OrdineInglese
+calcolaTasse()
+stampaRicevuta()
OrdinePadano
+calcolaTasse()
+stampaRicevuta()

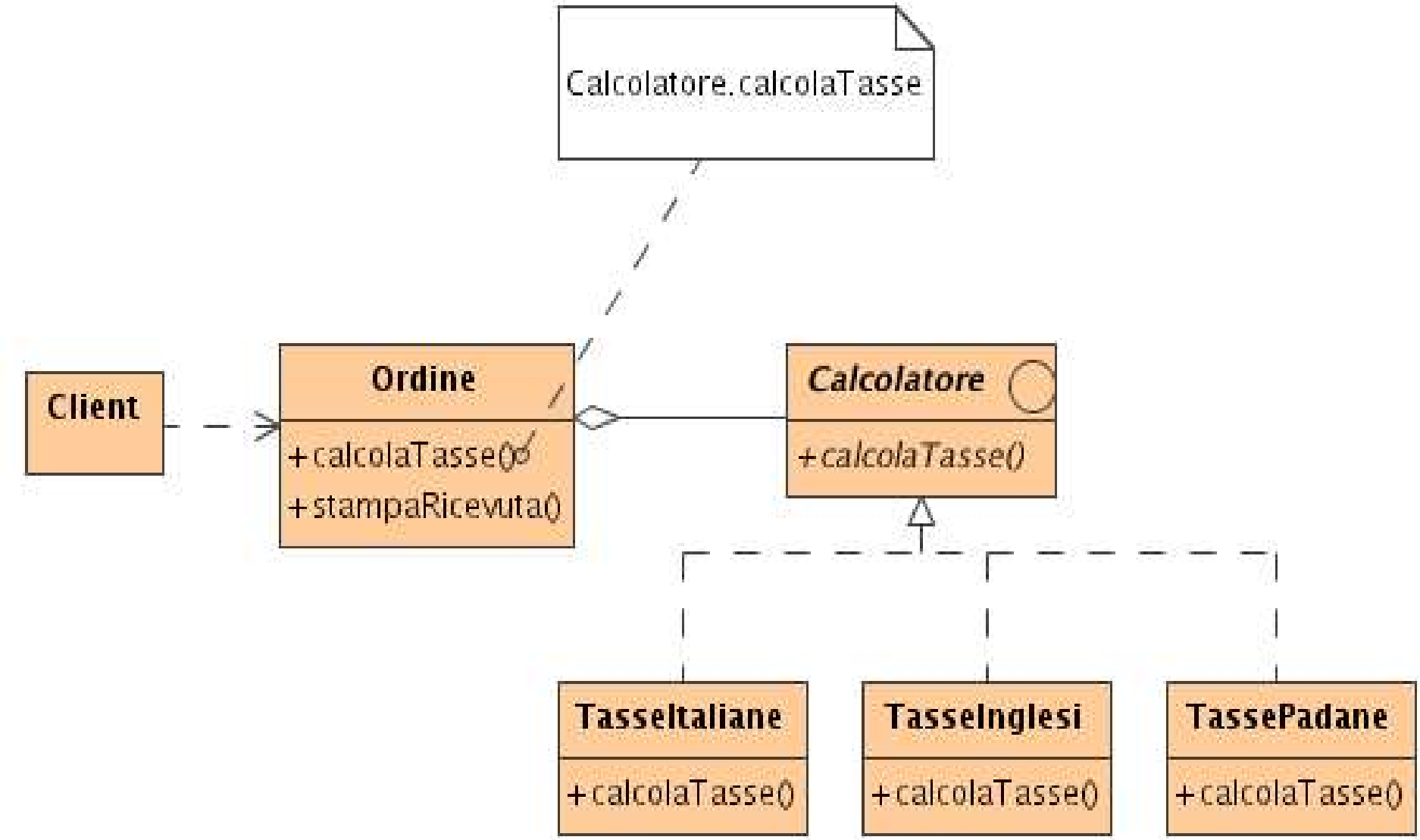
# Meglio ma...



- DIP è rispettato
- Chiusura strategica verso nuovi tipi di ordini
- Violazione di SRP: Ordine fa diverse cose e probabilmente ne farà altre in futuro... cosa succede se devo creare una nuova gerarchia di ordini? Non sono semplici altre chiusure strategiche

# Molto meglio!





# Molto meglio!



- DIP è rispettato, ed anche SRP
- Chiusura strategica verso nuovi tipi di calcolo
- Ininfluente dalla struttura gerarchica degli ordini
- Relazione dinamica fra *Ordine* e *Calcolatore*
- Incidentalmente questa struttura di oggetti ha un nome: **Strategy Pattern**

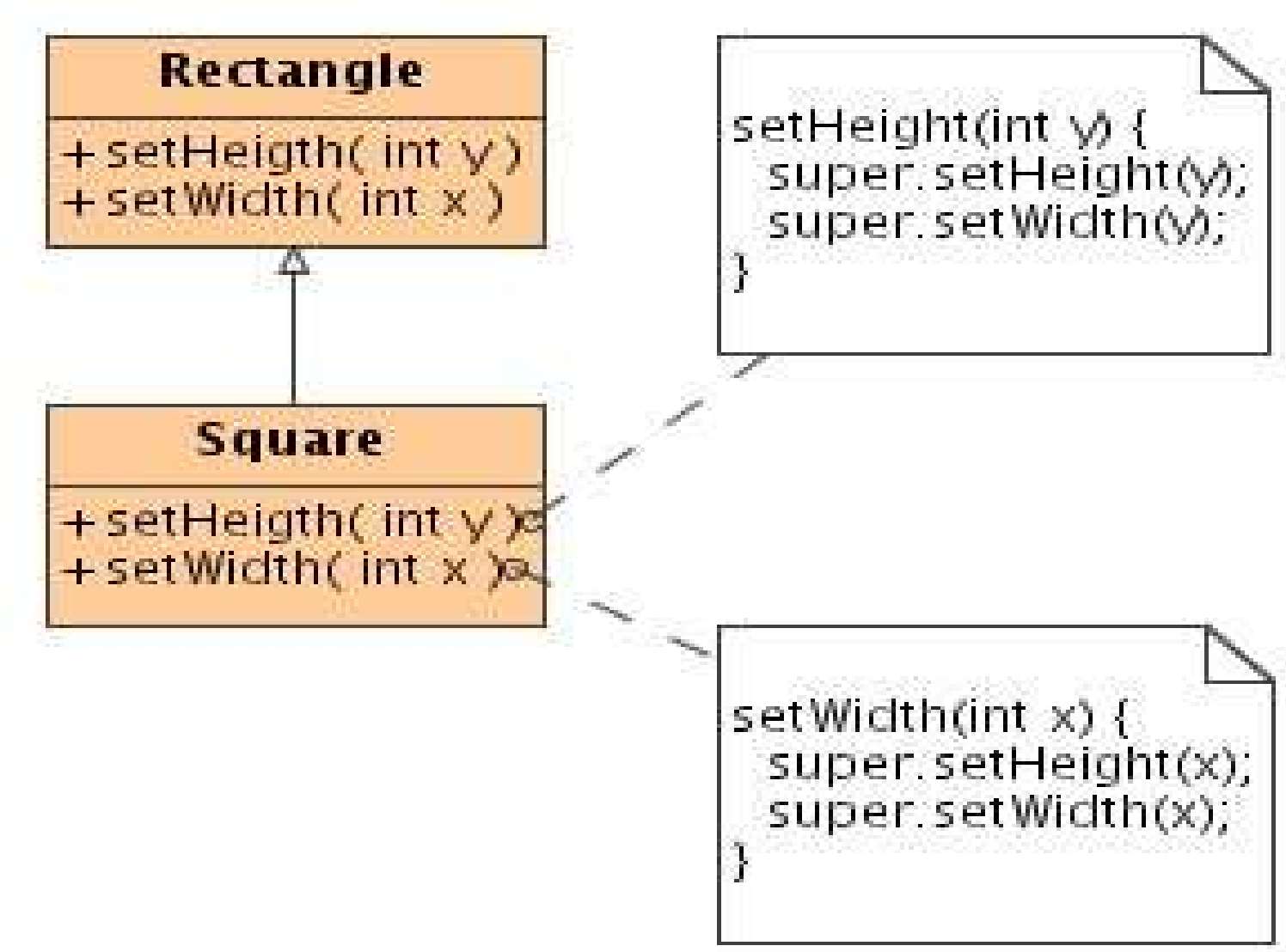
# Strategy in Java



- File e FilenameFilter
  - Il metodo *list()* di **File** (Context) delega ad un oggetto concreto (ConcreteStrategy) che implementa **FilenameFilter** (Strategy)
- Collection e Comparator
  - Il metodo *sort()* di **Collection** (Context) delega ad un oggetto concreto (ConcreteStrategy) che implementa **Comparator** (Strategy)
- Component e LayoutManager
  - Il metodo *doLayot()* di **Component** delega ad un Layout concreto (ConcreteStrategy) che implementa **LayoutManager** (Strategy)

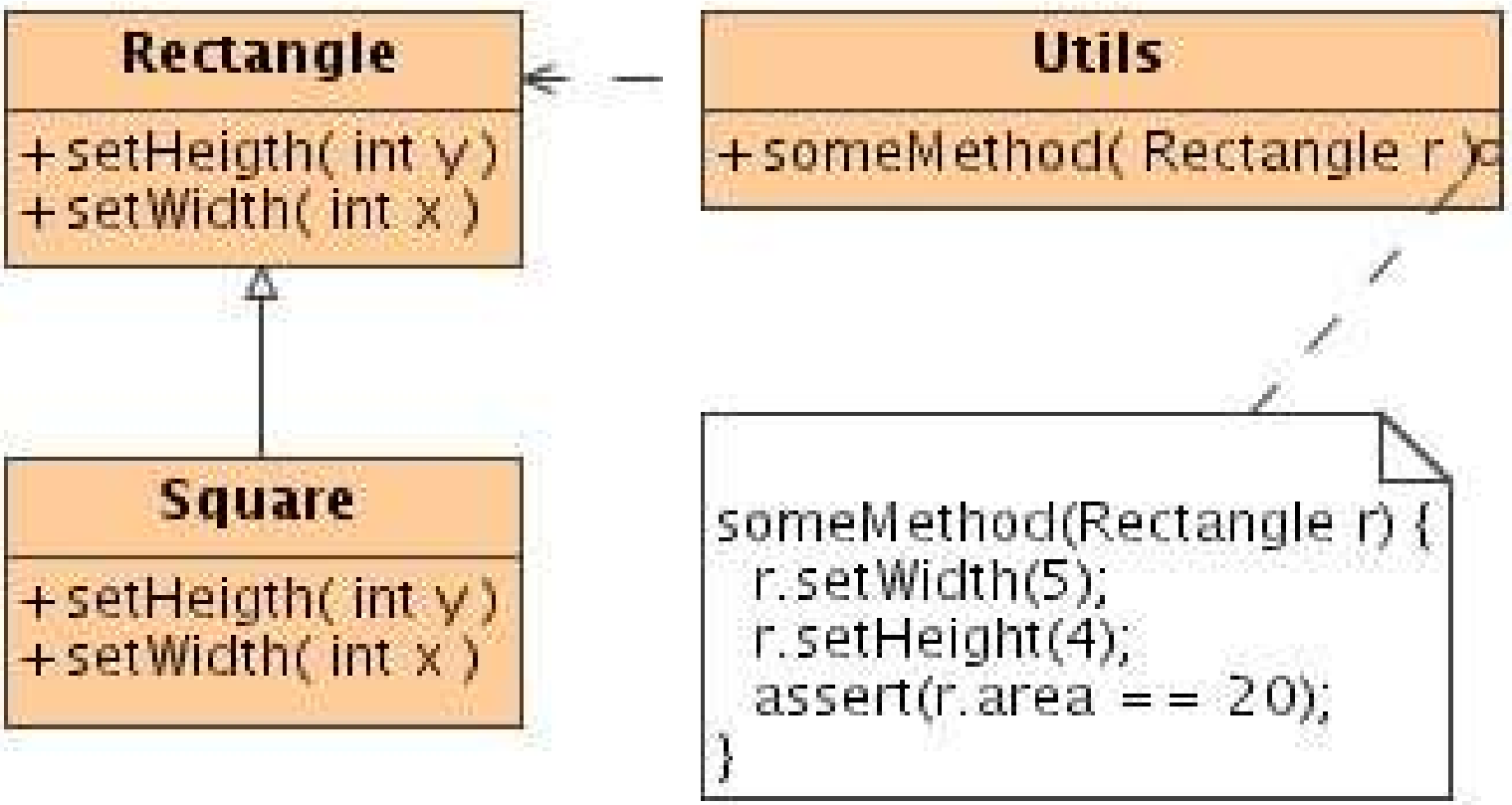
# Un Design OK?




Rectangle
+setHeigth( int y )
+setWidth( int x )
Square
+setHeigth( int y )
+setWidth( int x )
setHeight(int y) {
super.setHeight(y);
super.setWidth(y);
}
setWidth(int x) {
super.setHeight(x);
super.setWidth(x);
}

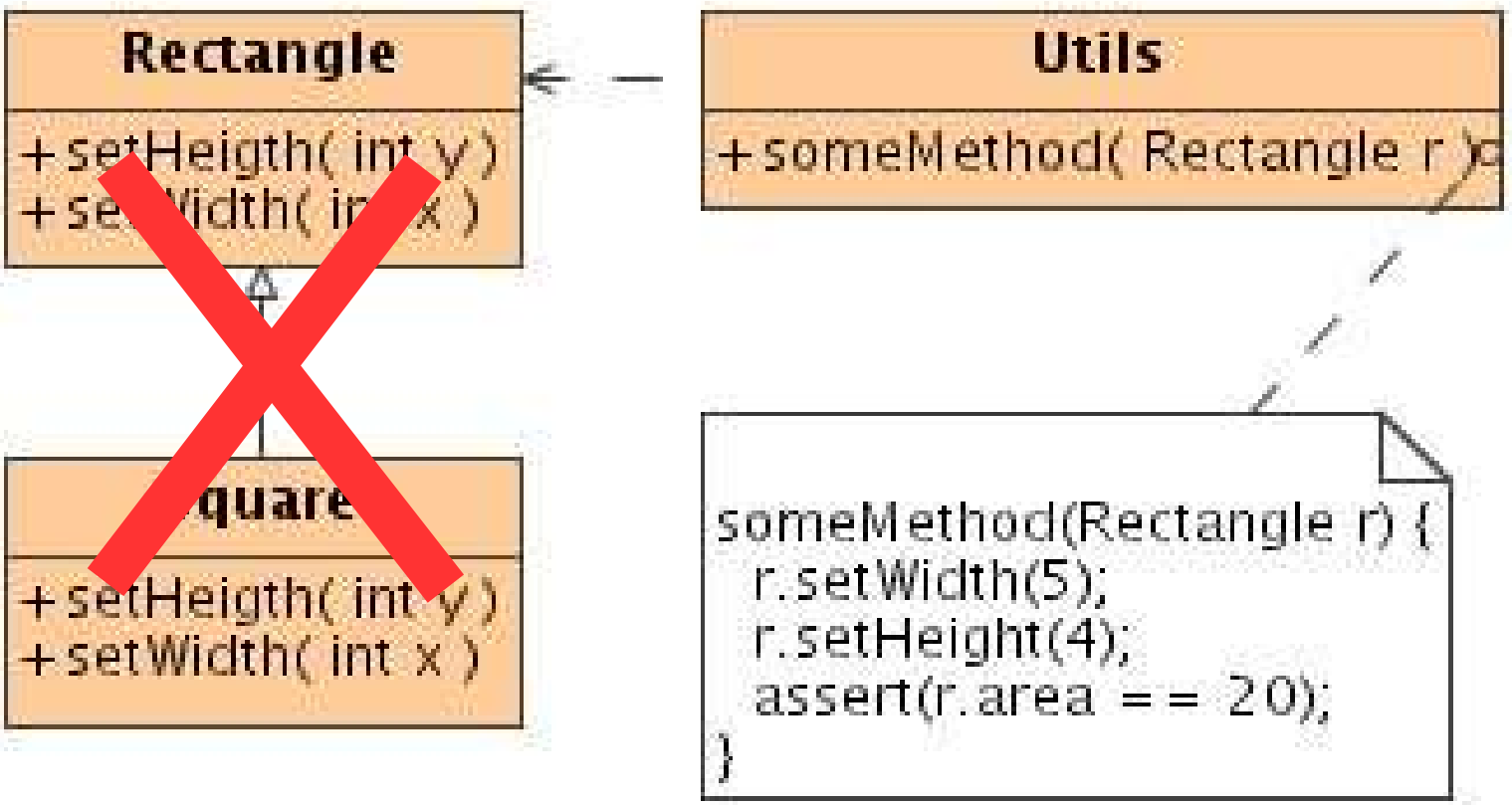
# Un Design OK?




Rectangle
+setHeigth( int y )
+setWidth( int x )
Utils
+someMethod( Rectangle r )
Square
+setHeigth( int y )
+setWidth( int x )
someMethod(Rectangle r) {
r.setWidth(5);
r.setHeight(4);
assert(r.area == 20);
}

# Violazione di LSP





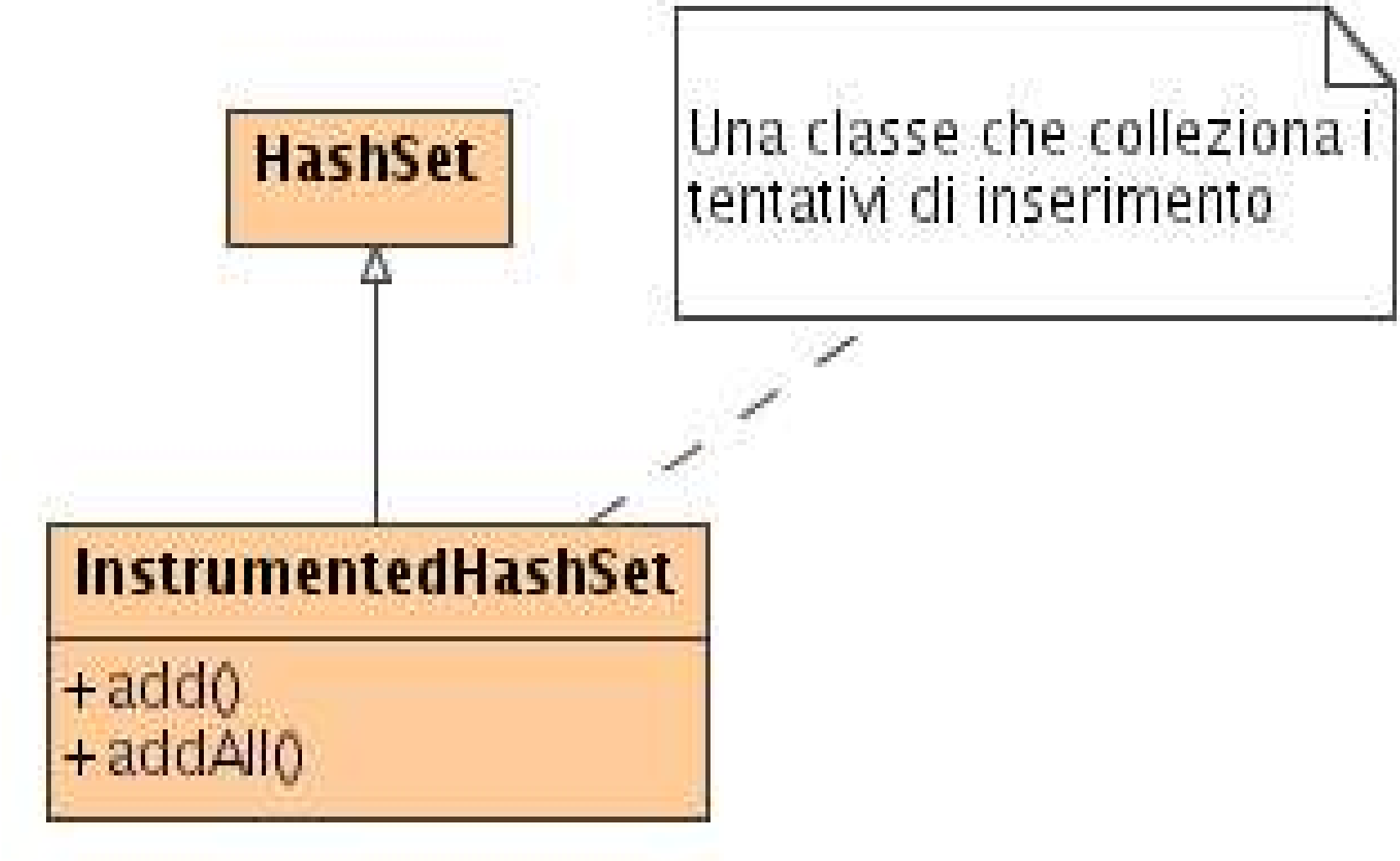
# Violazione di LSP



- *Square* non è sostituibile a *Rectangle*
- Una funzione che utilizza *Rectangle* cessa di funzionare passandogli uno *Square*
- In questo caso l'errore è da imputare allo sviluppatore di *Square*, che ha violato un invariante di *Rectangle* non esplicito: la modifica di un attributo non implica la modifica dell'altro

# Un design OK?




HashSet
Una classe che colleziona i tentativi di inserimento
InstrumentedHashSet
+add()
+addAll()

# Un design OK?



```
public class InstrumentedHashSet extends HashSet {
  //... costruttori omessi

  public boolean add(Object o) {
    counter++;
    return super.add(o);
  }

  public boolean addAll(Collection c) {
    counter += c.size();
    return super.addAll(c);
  }

  public void printHowMany(PrintStream stream) {
    stream.print("Number of elements: ");
    stream.println(counter);
  }

  private int counter;
}
```

# FBC (OCP, LSP)



- L'eredità (di implementazione) ROMPE l'incapsulamento: la classe base è fragile
- Si è costretti a rivelare dettagli implementativi per non “rompere” le sottoclassi (anche future!)
- “Favour composition over inheritance”
- Utilizzare l'ereditarietà di tipo (interfacce)
- Evitare i casi di self-use