CORSO I.F.T.S.

"TECNICHE PER LA PROGETTAZIONE E LA GESTIONE DI DATABASE"

Matricola 2014LA0033

DISPENSE DIDATTICHE
MODULO DI "PROGETTAZIONE SOFTWARE"

Dott. Imad Zaza

Lezione del 21/07/2014

# Diagramma di sequenza

- è utilizzato per definire la logica di uno scenario (specifica sequenza di eventi) di un caso d'uso (in analisi e poi ad un maggior livello di dettaglio in disegno)
- è uno dei principali input per l'implementazione dello scenario
- mostra gli oggetti coinvolti specificando la sequenza temporale dei messaggi che gli oggetti si scambiano
- è un **diagramma di interazione**: evidenzia come un caso d'uso è realizzato tramite la collaborazione di un insieme di oggetti

# Alcune definizioni

- Un diagramma di sequenza è un diagramma che descrive interazioni tra oggetti che collaborano per svolgere un compito
- Gli oggetti collaborano scambiandosi messaggi
- Lo scambio di un messaggio in programmazione ad oggetti equivale all'invocazione di un metodo

# Sequence diagram: componenti

- **Condizione:** è associata ad un messaggio: solo se è soddisfatta il messaggio viene generato [ha disponib]
- **Iterazione:** indica che un messaggio viene inoltrato più volte ad oggetti diversi di uno stesso gruppo
- **Ritorno:** indica il valore restituito all ' oggetto chiamante e non un nuovo messaggio; può essere omesso <- - - -
- **Distruzione:** deallocazione di un oggetto come metodo interno o tramite messaggio da un altro oggetto X

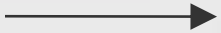
# Sequence diagram: messaggi

- In generale un **messaggio** rappresenta il trasferimento del controllo da un oggetto ad un altro
- Se l'oggetto che invia il messaggio rimane in attesa che l'oggetto ricevente ritorni, si ha un messaggio **sincrono**



- Se invece l'oggetto che invia il messaggio prosegue la propria elaborazione in parallelo all'oggetto chiamato, siamo in presenza di un messaggio **asincrono**.

# Scambio Messaggi Sincroni (1/2)



- collega freccia di invocazione e freccia di ritorne a chiamato. La freccia è etichettata col nome del metodo invocato, e opzionalmente i suoi parametri e il suo valore di ritorno
- Il chiamante attende la terminazione del metodo del chiamato prima di proseguire
- Il life-time (durata, vita) di un metodo è rappresentato da un rettangolino che collega freccia di invocazione e freccia di ritorno

# Scambio Messaggi Sincroni (2/2)

- Life-time corrisponde ad avere un record di attivazione di quel metodo sullo stack di attivazione
- Il ritorno è rappresentato con una freccia tratteggiata
- Il ritorno è sempre opzionale. Se si omette, la fine del metodo è decretata dalla fine del life-time

# Scambio Messaggi Asincroni

- Si usano per descrivere interazioni concorrenti
- Si disegna con una freccia aperta

  da chiamante a chiamato. La freccia è etichettata col nome del metodo invocato, e opzionalmente i suoi parametri e il suo valore di ritorno
- Il chiamante non attende la terminazione del metodo del chiamato, ma prosegue subito dopo l'invocazione
- Il ritorno non segue quasi mai la chiamata

# Esecuzione condizionale di un messaggio

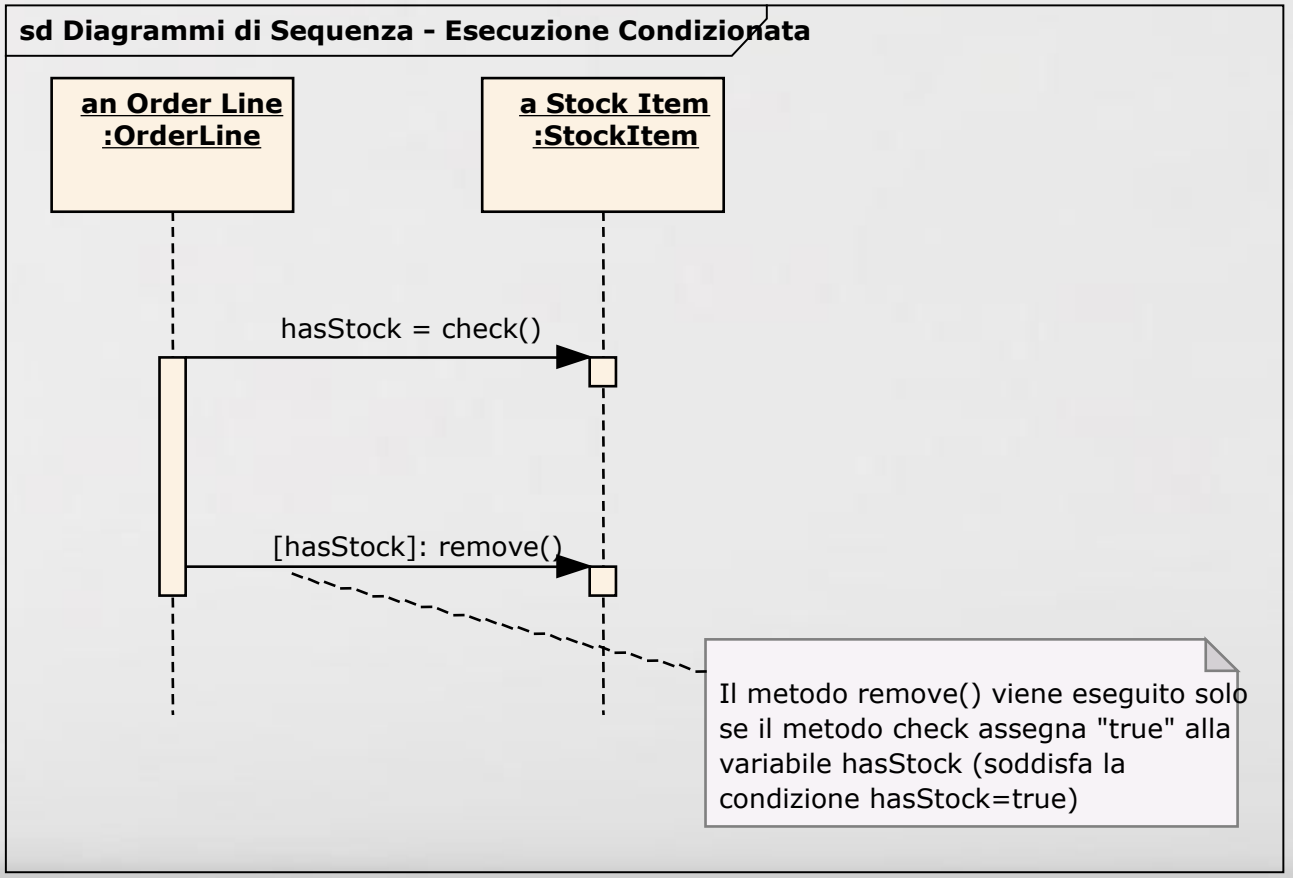
- L'esecuzione di un metodo può essere assoggettata ad una condizione. Il metodo viene invocato solo se la condizione risulta verificata a run-time

- Si disegna aggiungendo la condizione, racchiusa tra parentesi quadre, che definisce quando viene eseguito il metodo

- Sintassi:

[cond] : nomeMetodo()

# Esecuzione condizionale di un messaggio - esempio




# Iterazione di un messaggio

- Rappresenta l'esecuzione ciclica di messaggi
- Si disegna aggiungendo un * (asterisco) prima del metodo su cui si vuole iterare
- Si può aggiungere la condizione che definisce l'iterazione
- La condizione si rappresenta tra parentesi quadre. Sintassi completa:

[cond] : * nomeMetodo()

# Iterazione di un blocco di messaggi

- Rappresenta l'esecuzione ciclica di più messaggi
- Si disegna raggruppando con un blocco (riquadro, box) i messaggi (metodi) su cui si vuole iterare
- Si può aggiungere la condizione che definisce l'iterazione sull'angolo in alto a sinistra del blocco
- La condizione si rappresenta al solito tra parentesi quadre

# “Auto-Chiamata” (Self-Call)

- Descrive un oggetto che invoca un suo metodo (chiamante e chiamato coincidono)
- Si rappresenta con una “freccia circolare” che rimane all’interno del life time di uno stesso metodo

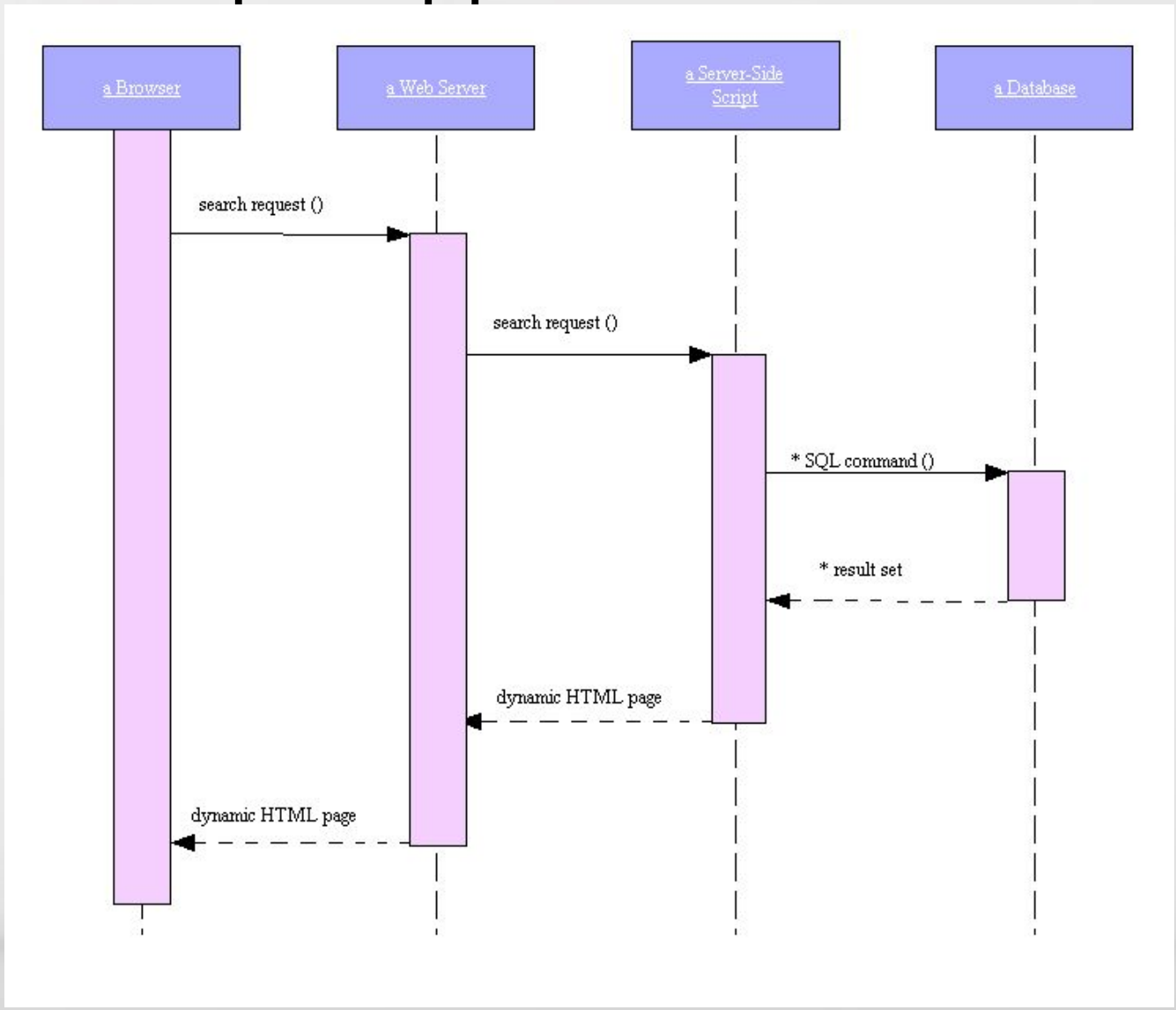
# Costruzione di un oggetto

- Rappresenta la costruzione di un nuovo oggetto non presente nel sistema fino a quel momento
- Messaggio etichettato new, create,…
- L'oggetto viene collocato nell'asse temporale in corrispondenza dell'invocazione nel metodo new (o create…)

# Eliminazione di un oggetto

- Rappresenta la distruzione di un oggetto presente nel sistema fino a quel momento
- Si rappresenta con una X posta in corrispondenza della life-line dell'oggetto
- Da quel momento in poi non è legale invocare alcun metodo dell'oggetto distrutto

# Esempio Applicazione Web

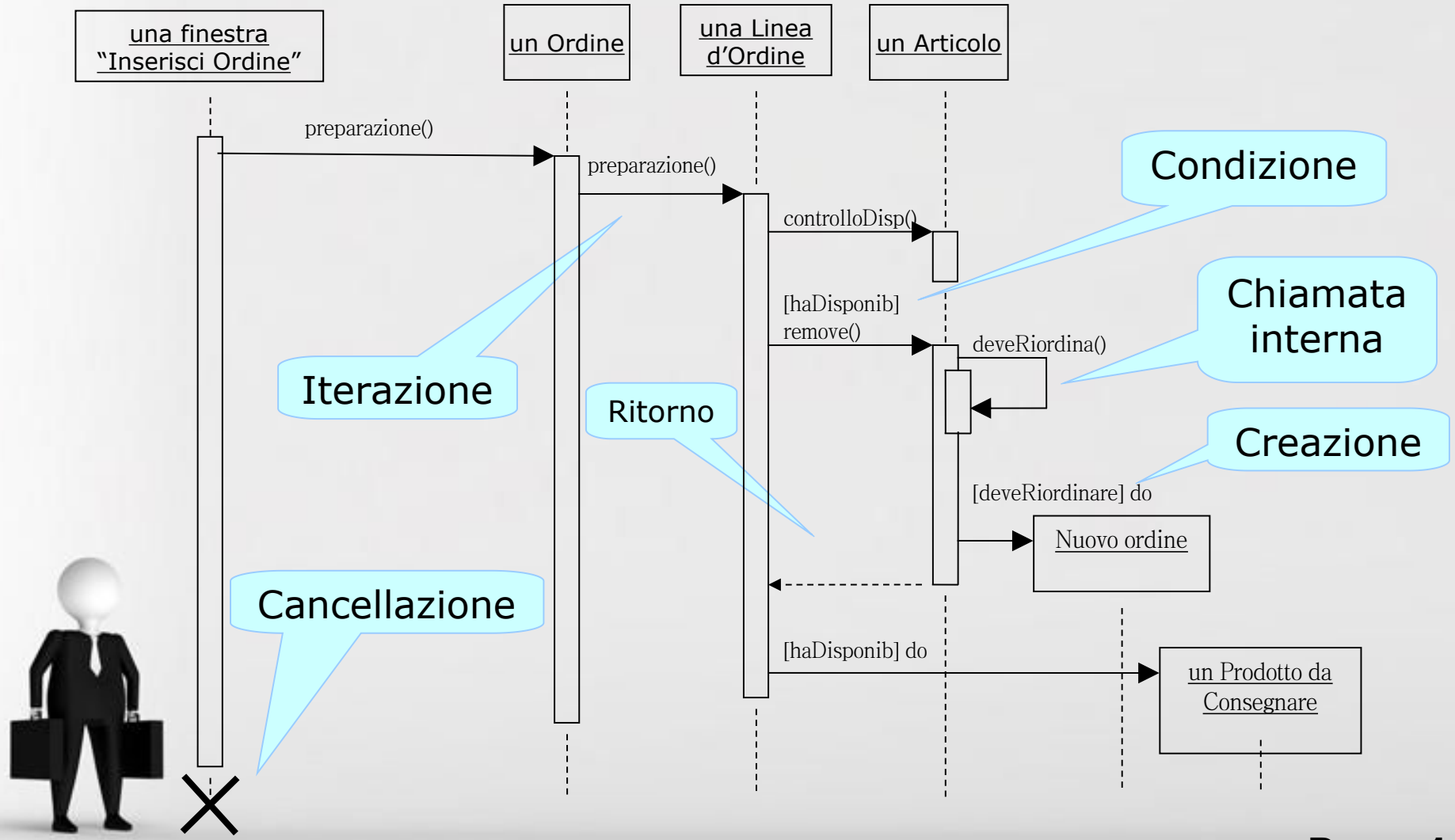



# Esercizio: ordine prodotto

Supponiamo di dover illustrare il seguente caso:

- La finestra Inserisci Ordine manda un "messaggio di preparazione" ad un Ordine
- L'Ordine invia messaggi di preparazione a tutte le Linee d'Ordine contenute nell'Ordine
- Ciascuna Linea controlla la disponibilità del proprio Articolo:
  - se è presente lo rimuove dal magazzino e crea un prodotto da consegnare
  - se la disponibilità del prodotto è scesa al di sotto di una certa soglia l'oggetto Articolo genera una richiesta di un nuovo ordine

# Esercizio: soluzione



# Esercizio

- Costruiamo un diagramma di sequenza per il seguente use case
  - Una finestra di tipo Order Entry invia il messaggio “prepare” ad un Ordine (Order)
  - L’ordine invia il messaggio “prepare” ad ogni sua linea (Order Line)
  - Ogni linea verifica gli elementi in stock (Stock Item)
  - Se il controllo ha esito positivo, la linea rimuove l’appropriata quantità di elementi in stock e crea un’unità di delivery (DeliveryItem)
  - Se gli elementi in stock rimanenti scendono al di sotto di una soglia di riordino, viene richiesto un riordino (ReorderItem)

# Alcuni suggerimenti finali

- Assicurarsi che i metodi rappresentati nel diagramma siano gli stessi definiti nelle corrispondenti classi (con lo stesso numero e lo stesso tipo di parametri)
- Documentare ogni assunzione nella dinamica con note o condizioni (ad es. il ritorno di un determinato valore al termine di un metodo, il verificarsi di una condizione all'uscita da un loop, ecc.)
- Mettere un titolo per ogni diagramma (ad es. "sd Diagrammi di Sequenza – Eliminazione di un Oggetto" )

# Alcuni suggerimenti finali

- Scegliere nomi espressivi per le condizioni e per i valori di ritorno
- Non inserire troppi dettagli in un unico diagramma (flussi condizionati, condizioni, logica di controllo)
- Non bisogna rappresentare tutto quello che si rappresenta nel codice …
- Se il diagramma è complesso, scomporlo in più diagrammi semplici (ad es. uno per il ramo if, un altro per il ramo else, ecc.)

# Collaboration diagram

- Evidenzia la collaborazione fra gli oggetti, ma lo svolgimento temporale complessivo risulta meno chiaro rispetto ai sequence diagrams.
- I componenti del diagramma sono sempre **oggetti** e **messaggi** ma questa volta non è modellato esplicitamente in maniera grafica lo scorrere del tempo. Gli oggetti sono disposti liberamente, e in modo da rendere chiare le interazioni più importanti
- Ogni oggetto ha il formato

  *NomeOggetto: NomeClasse*

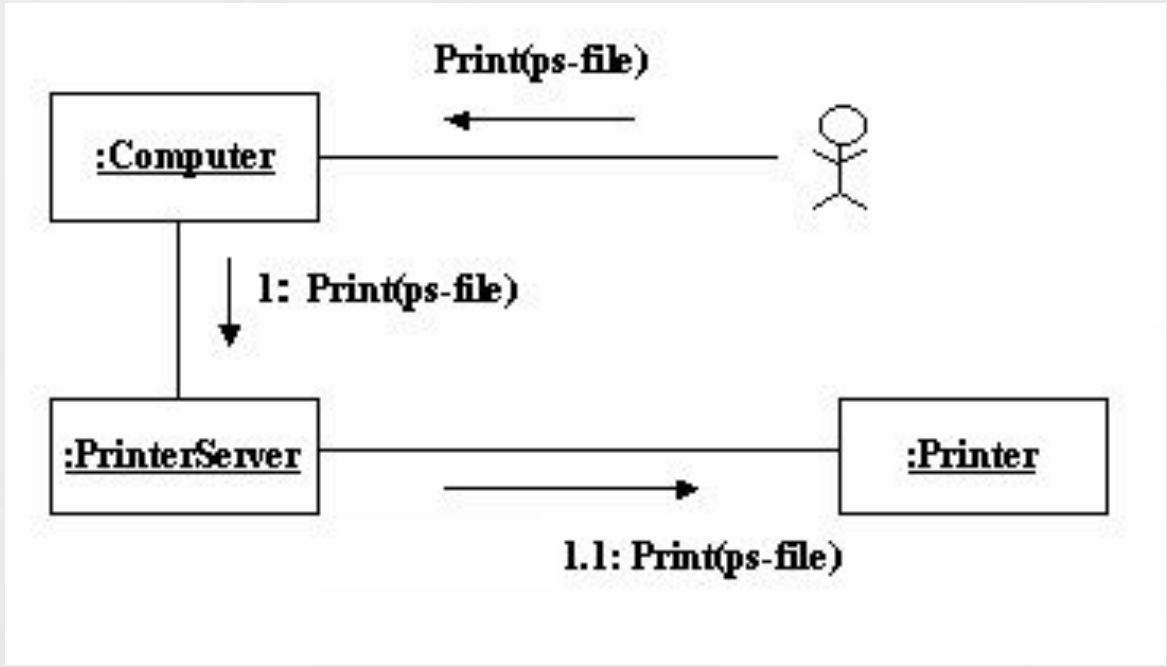
  ma è possibile indicare solo l'uno o l'altro

# Collaboration diagram

- A differenza del sequence diagram, i messaggi sono numerati, in modo da evidenziare esattamente la sequenza

- I messaggi sono accompagnati da frecce per capire quale oggetto invia il messaggio e quale lo riceve

# Collaboration diagram: esempio

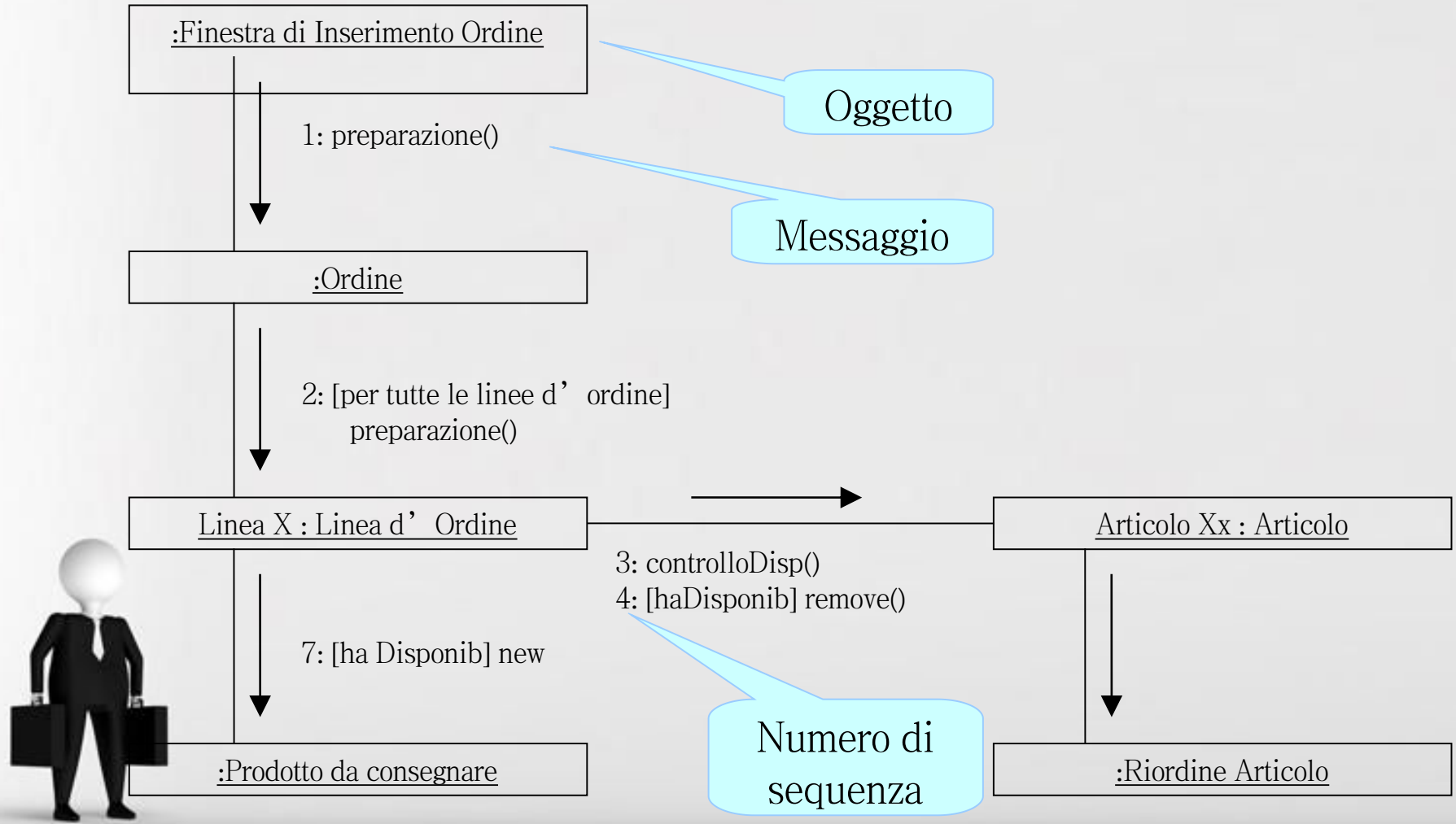



# Esercizio: ordine prodotto

Supponiamo di dover illustrare il seguente caso:

- La finestra Inserisci Ordine manda un "messaggio di preparazione" ad un Ordine
- L'Ordine invia messaggi di preparazione a tutte le Linee d'Ordine contenute nell'Ordine
- Ciascuna Linea controlla la disponibilità del proprio Articolo:
  - se è presente lo rimuove dal magazzino e crea un prodotto da consegnare
  - se la disponibilità del prodotto è scesa al di sotto di una certa soglia l'oggetto Articolo genera una richiesta di un nuovo ordine

# Esercizio: soluzione

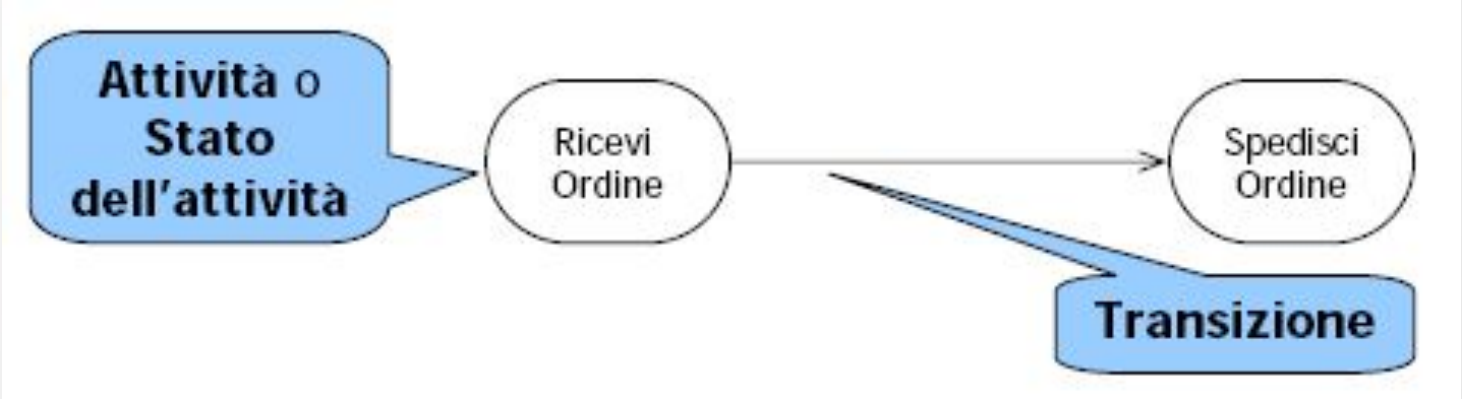


# Activity Diagram

- Servono per la modellazione di workflow e per descrivere processi con una forte componente di computazione parallela
- Il componente principale è **l'Attività**: si può definire in maniera informale come il "fare qualcosa", cioè un processo del mondo reale (es. compilare un ordine) o l'esecuzione di una procedura software (es. metodo di una classe)
- Il diagramma consiste in una sequenza di attività e supporta l'esecuzione di cicli,l'esecuzione parallela e quella condizionale

# Activity Diagram

- Oltre alle Attività le altre componenti sono le **Transizioni**
- Le transizioni non hanno condizioni e sono innescate semplicemente dal termine dell'attività precedente

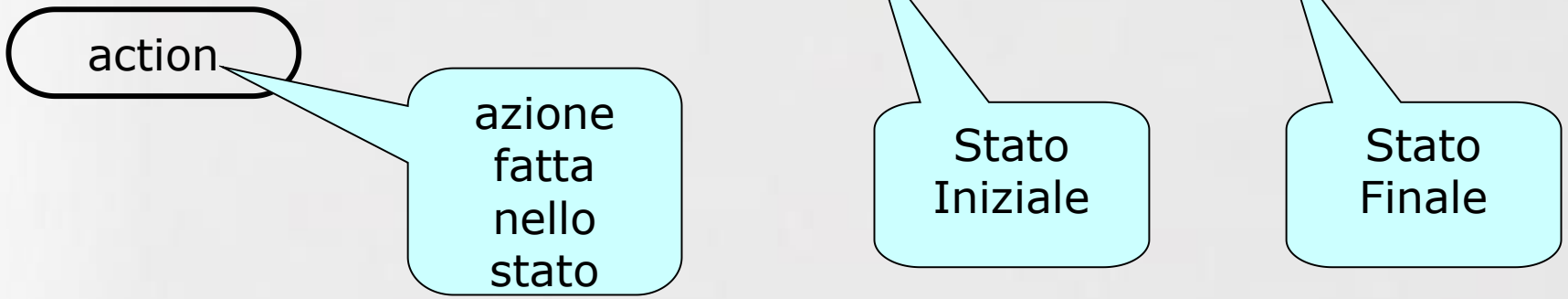



# Activity Diagram

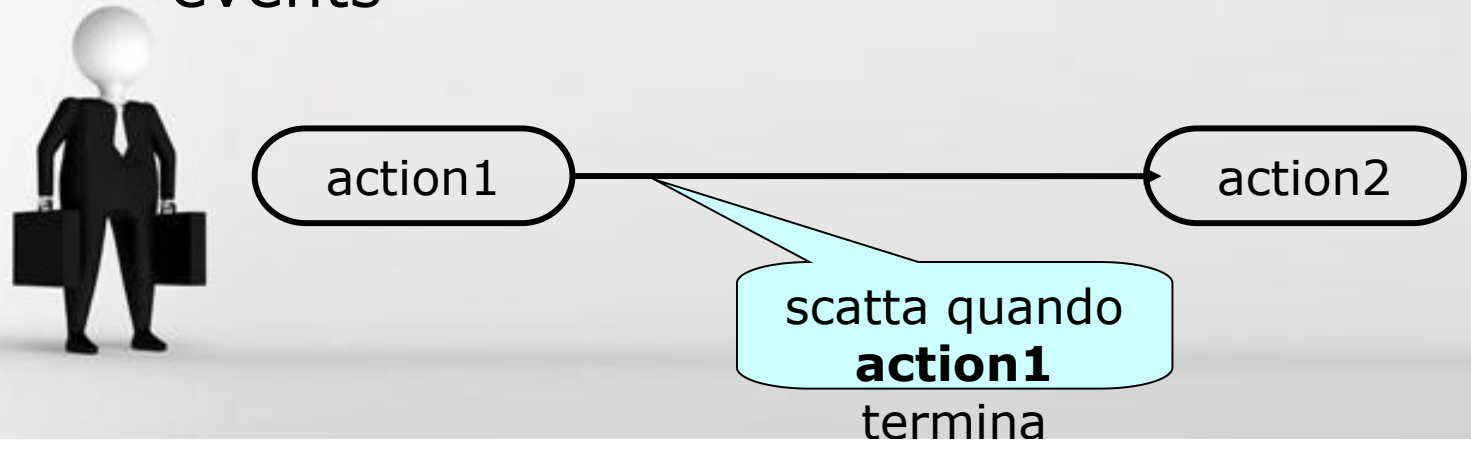
- Un Activity Diagram può essere associato:
  - A una classe
  - All'implementazione di un'operazione
  - Ad uno Use Case

# Activity Diagram: Componenti

- Action state

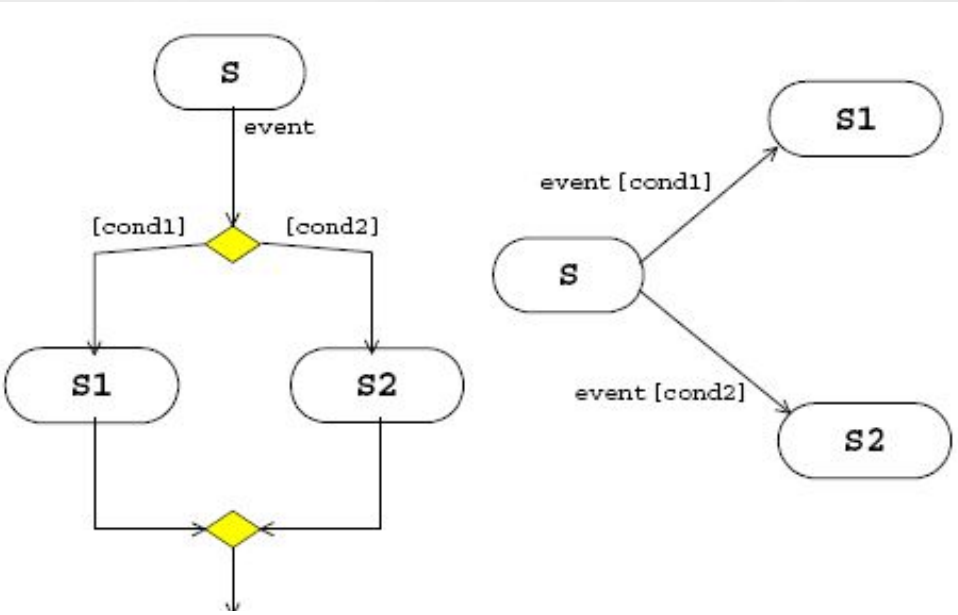



Transizioni scatenate da completion events



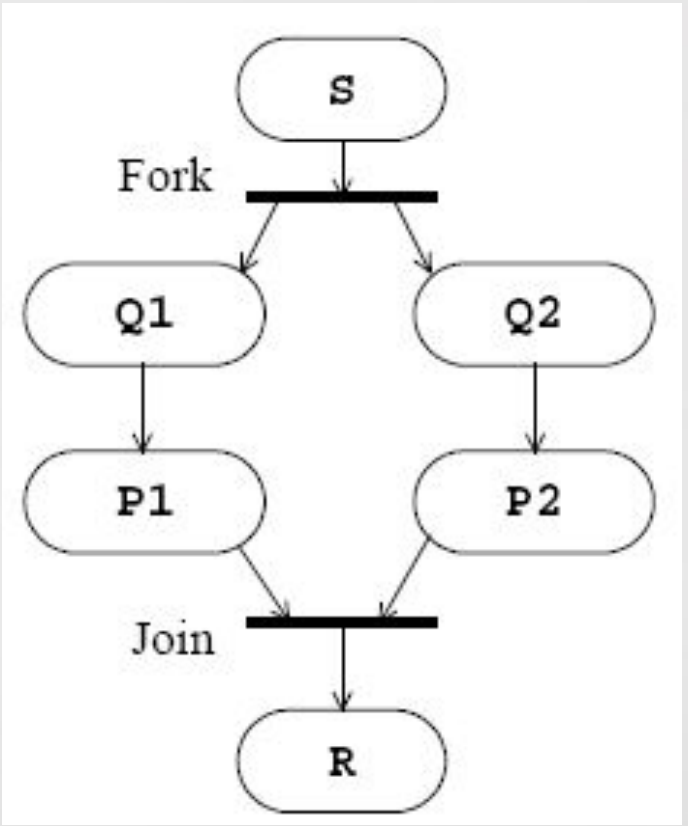
# Activity Diagram: Branch & Merge

- **Branch**: diramazione con una transizione entrante e più di una uscente, con condizioni mutuamente esclusive
  - Non è possibile che valgano entrambe (comportamento ambiguo) o nessuna comportamento bloccato)
  - Se le condizioni sono più di due deve comunque esserne verificata ogni volta una e una soltanto
- **Merge**: giunzione con più transizioni entranti ed una sola uscente, termina il blocco aperto dal Branch

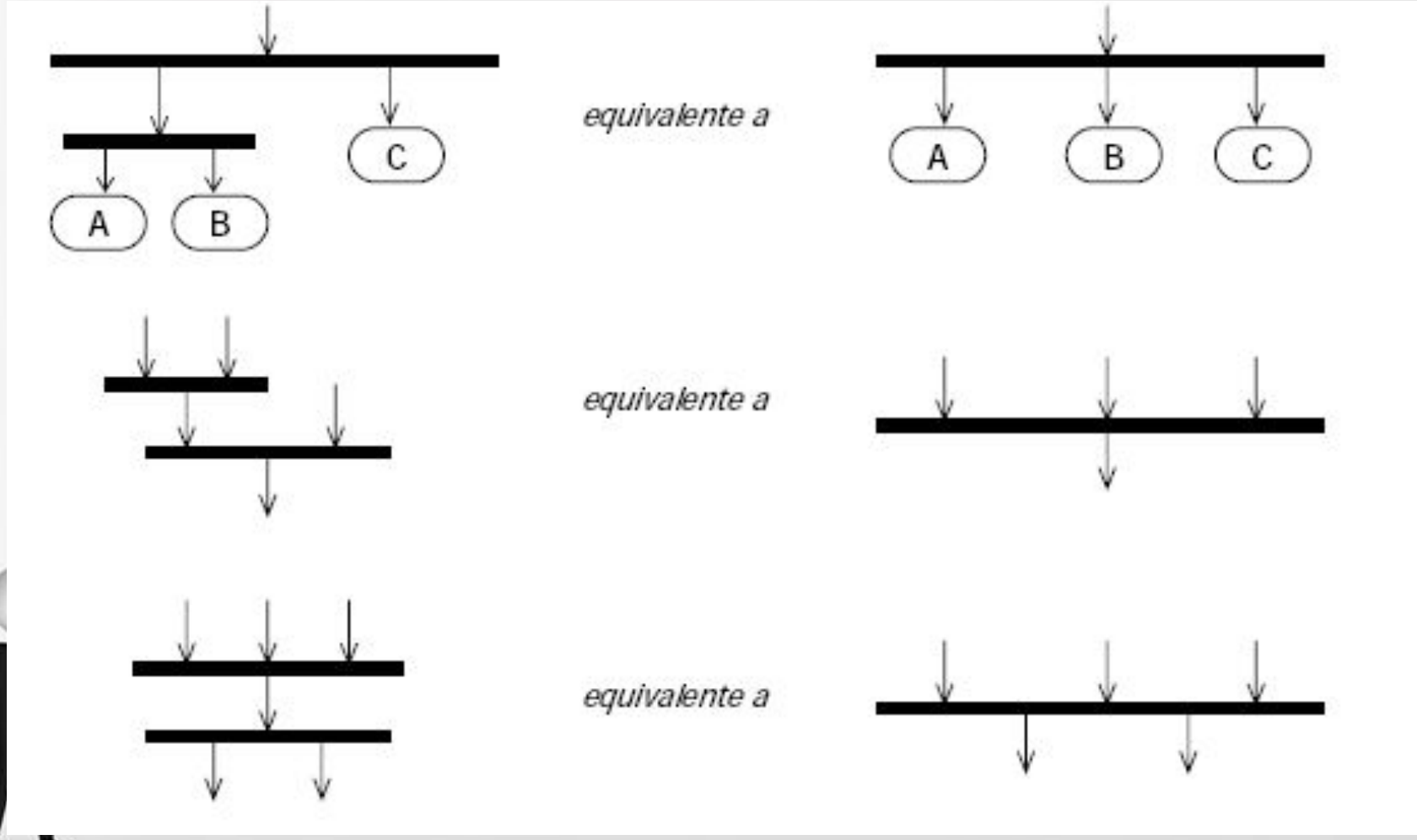


# Activity Diagram: Fork e Join

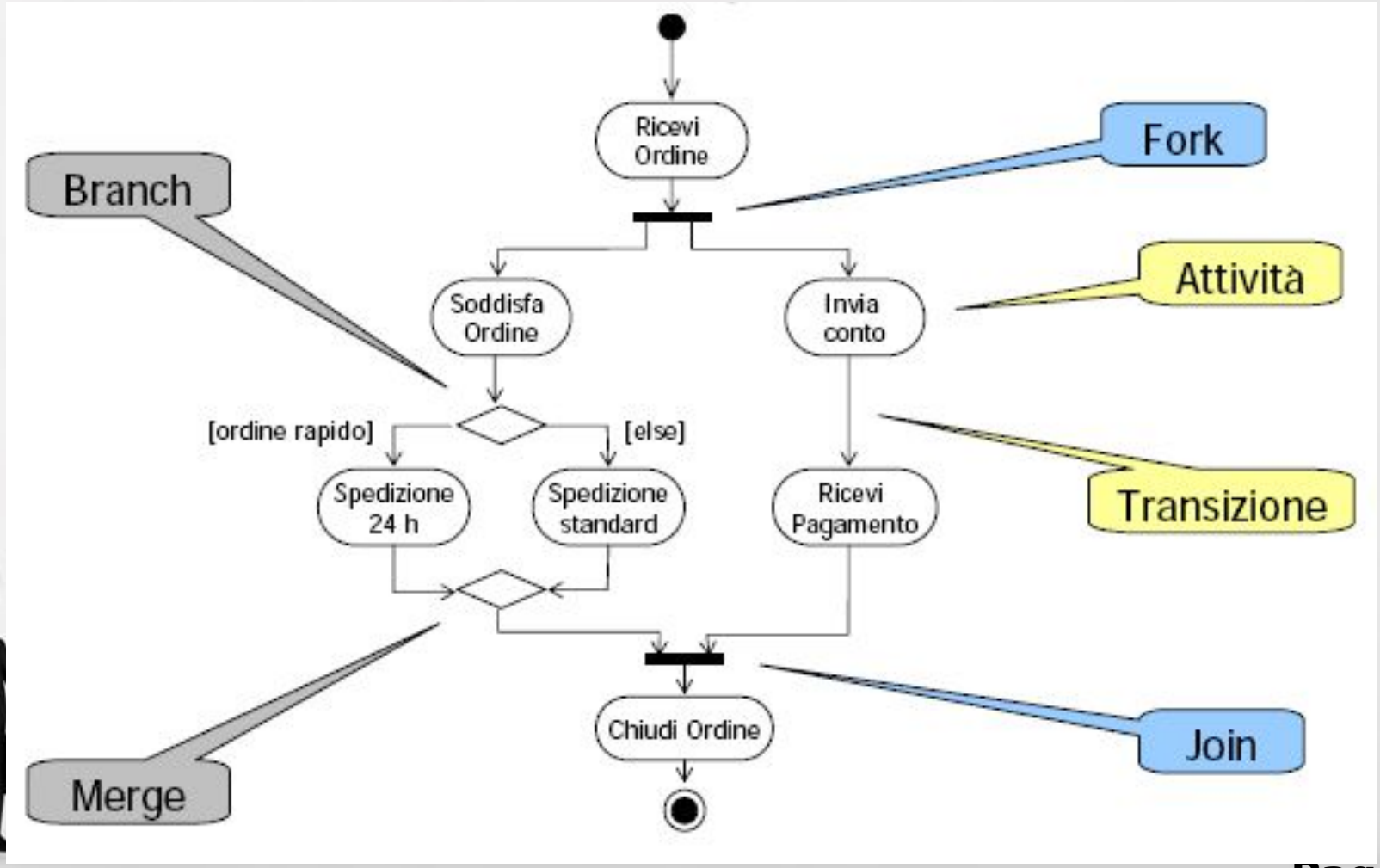
- **Fork**: divisione con una transizione entrante e più di una uscente che si eseguono in parallelo
- **Join**: unione con più transizioni entranti ed una sola uscente che può scattare solo dopo che sono terminate le attività degli stati corrispondenti alle transizioni entranti
- **Fork e Join si devono corrispondere!**




# Combinazione di Fork e Join



# Activity diagram: Esempio

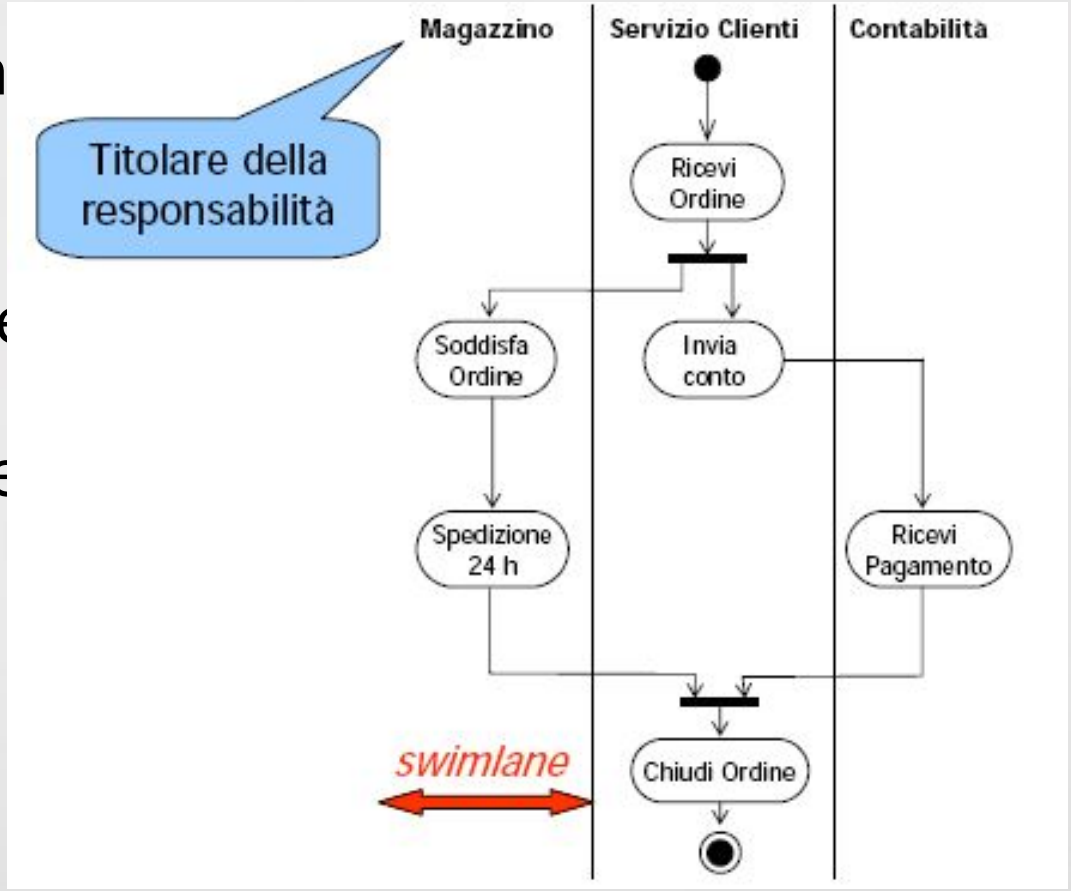


# Swimlanes

- I diagrammi delle attività documentano bene ciò che accade,ma non chi fa che cosa
- Un modo di risolvere il problema, indicando le responsabilità per le attività, è l'uso delle **swimlanes**
- Le swimlanes sono corsie verticali separate da linee continue, in cui ogni corsia è coperta dalla responsabilità di una particolare classe (o sottosistema)

# Swimlanes

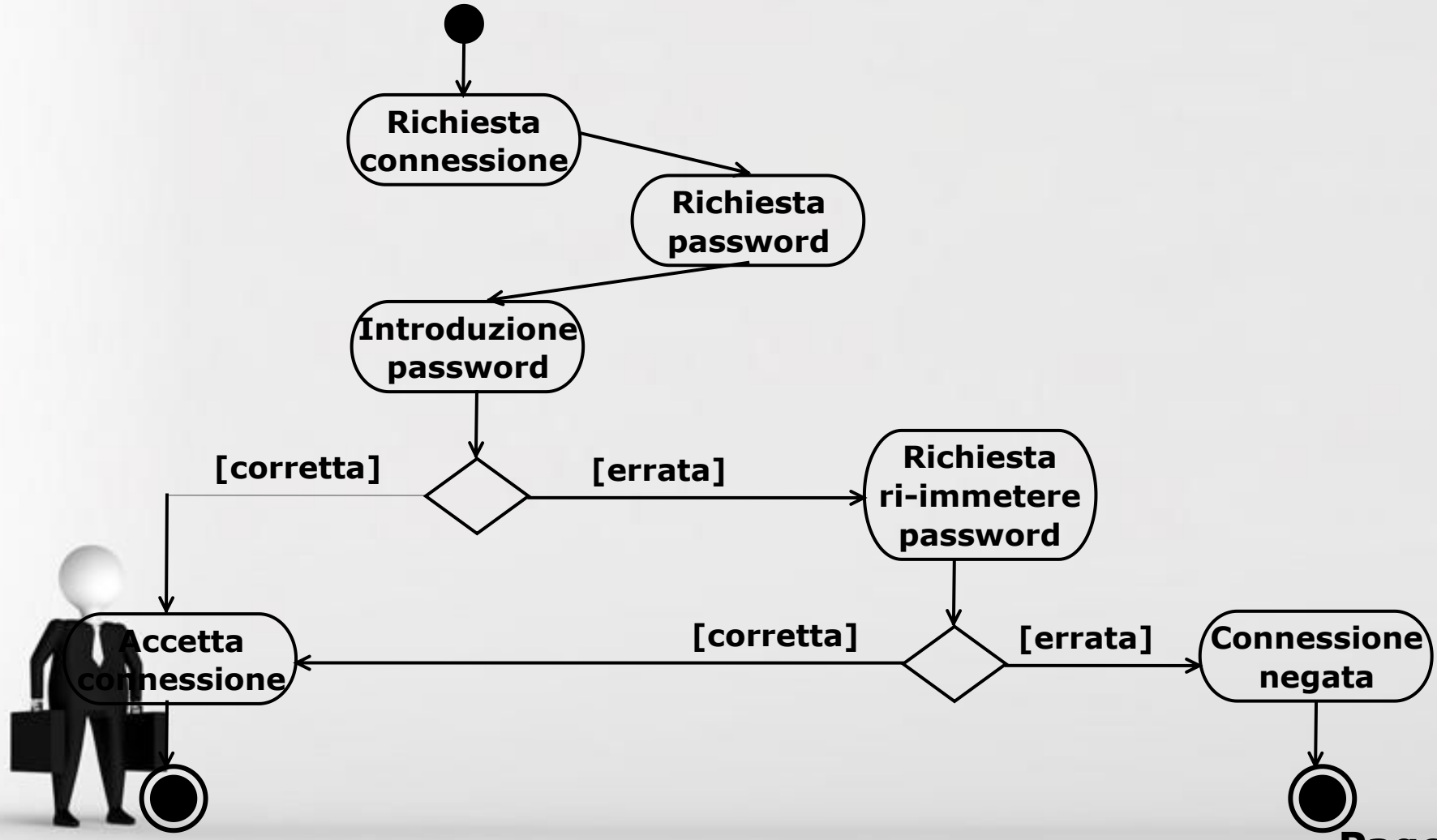
- Ogni attività deve essere contenuta in una singola swimlane
- Le transizioni invece possono attraversare le linee verticali di demarcazione



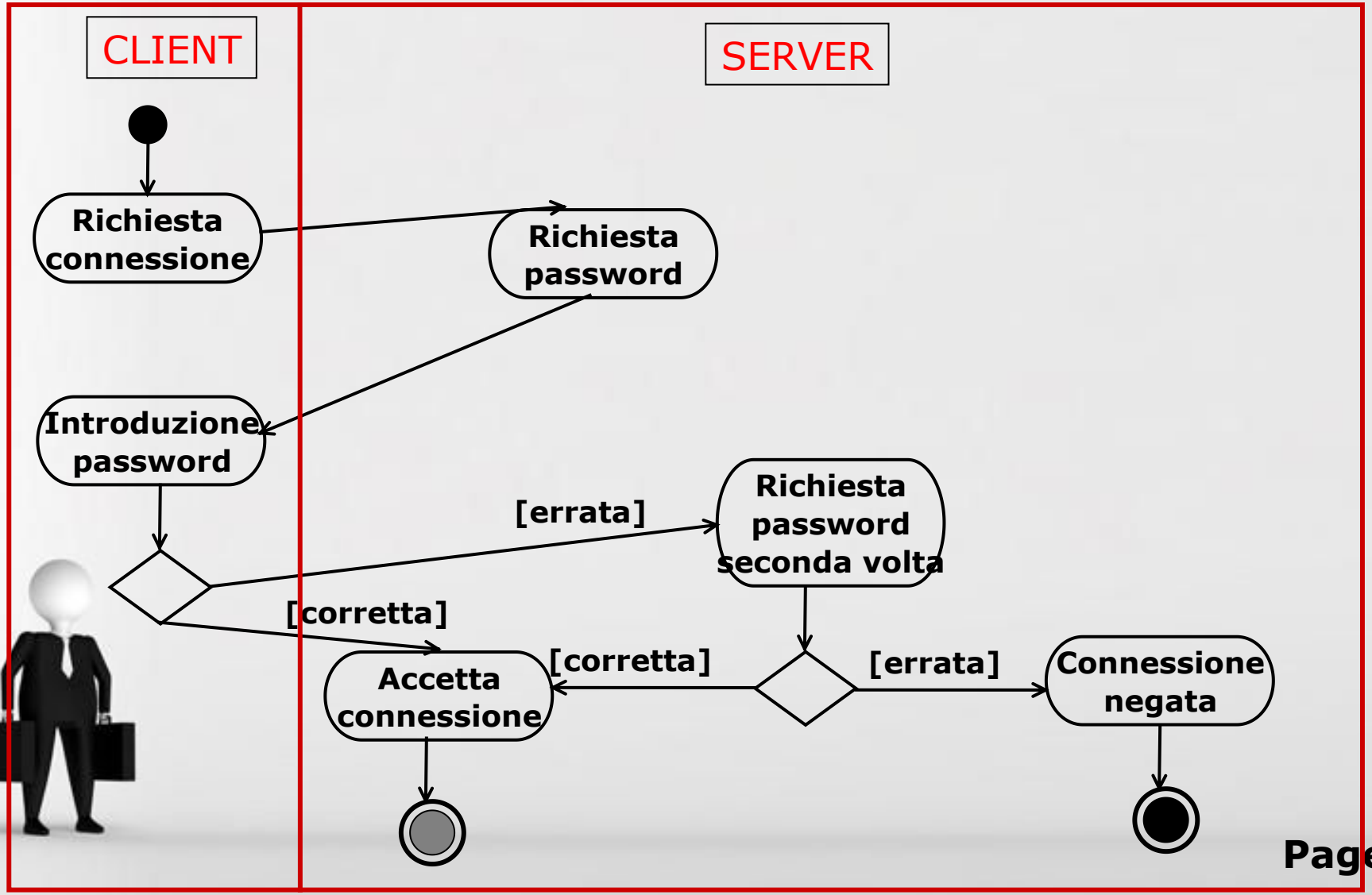

# Esercizio

- Sistema di autenticazione ad un servizio:
  - Richiesta password
  - Verifica password

# Esercizio: soluzione(1)



# Esercizio: soluzione(2)



# Design Patterns

# Perchè

I problemi incontrati nello sviluppare grossi progetti software sono spesso ricorrenti e prevedibili.

- I design pattern sono schemi utilizzabili nel progetto di un sistema.
- Permettono quindi di non inventare da capo soluzioni ai problemi già risolti, ma di utilizzare dei "mattoni" di provata efficacia.
- Inoltre, un bravo progettista sa riconoscerli nella documentazione o direttamente nel codice, e utilizzarli per comprendere in fretta i programmi scritti da altri. Quindi:
  - forniscono un vocabolario comune che facilita la comunicazione tra progettisti;
  - svantaggio potenziale: possono rendere la struttura del codice più complessa del necessario. Di volta in volta bisogna decidere se adottare semplici soluzioni ad hoc o riutilizzare pattern noti;
  - pericolo di "overdesign".

# Vantaggi

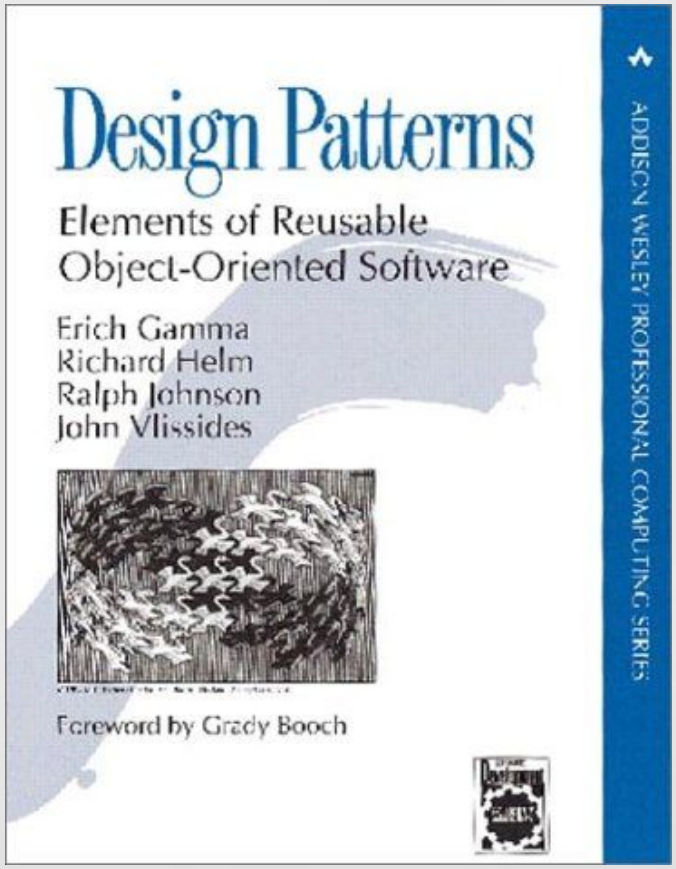
- Notevole aumento della capacità di produrre software riutilizzabile;
- Si danno allo sviluppatore strumenti utili per la modellazione di nuovi sistemi;
- Si aumenta la documentazione e la chiarezza;
- Si aumenta la velocità di sviluppo;
- Si aumenta la robustezza del software;
- Si aumenta la flessibilità e l'eleganza del software.

# Pattern

- Il termine "pattern" fu introdotto dall'architetto austriaco Christopher
- Alexander negli anni '70 (per la pianificazione di costruzioni in ambienti urbani).
- Nel 1987 Cunningham e Beck adattarono l'idea di Alexander per guidare programmatori inesperti in Smalltalk.
- Erich Gamma, tesi di dottorato, 1988-1991.
- Dal 1990 al 1992 la famosa Gang of Four (Gamma, Helm, Johnson,
- Vlissides) incominciò la stesura di un catalogo di pattern.
- Nel 1995 la Gang of Four pubblicò "Design Patterns - Elements of Reusable Object-Oriented Software" con la descrizione di 23 pattern.

# Gang of Four

- E' il libro da cui è nato tutto, l'unica Bibbia
- Le figure sono riprese da qui




# AntiPattern

La definizione di antipattern è stata coniata dalla "Gang of Four" per indicare i tipici problemi che incorrono i programmatori nella scrittura del codice. Vi è un elenco esteso di "trappole" in cui cade il programmatore, eccone alcune:

- Coltellino svizzero quando il programmatore prevede un numero elevato di funzionalità, la maggior parte inutile.
- Fede cieca quando il programmatore presume la correttezza di un codice e non prevede alcun controllo.
- Azione a distanza quando elementi del programma che interagiscono sono posti a distanza non controllando gli effetti della modifica di una parte sull'altra.
- Reinventare la ruota quando il programmatore rinuncia ad adattare un modulo esistente riscrivendolo da capo e inserendo, presumibilmente, errori.
- Spaghetti code quando si un uso eccessivo di costrutti per il controllo del flusso rendendo praticamente illeggibile un codice e propenso agli errori.
- Secondo gli autori, l'uso dei pattern contribuisce nell'evitare queste trappole.

# Definizione

Un pattern è l'astrazione di un problema che si verifica nel nostro dominio, rappresentandone la soluzione in modo che sia possibile riutilizzarla per numerosi altri contesti (Christopher Alexander).

Descrizione di classi ed oggetti comunicanti adatti a risolvere un problema progettuale generale in un contesto particolare. (Gamma, Helm, Johnson, Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley).

# Struttura di un pattern

Nel libro dei GoF ogni pattern è descritto nel seguente modo:

- **Descrizione**: una breve descrizione dell'obiettivo del pattern.
- **Esempio**: si presenta un problema la cui soluzione si ottiene tramite l'applicazione del pattern.
- **Descrizione della soluzione offerta dal pattern**: si descrive testualmente l'architettura del pattern e come questa si applica al problema.
- **Struttura del pattern**: diagramma di classi in UML della struttura generica del pattern.
- **Applicazione del pattern**: offre un diagramma UML delle classi del problema, presenta l'abbinamento delle classi del problema con le classi che descrivono la struttura concettuale del pattern, descrive l'implementazione del codice Java, presenta e commenta gli output dell'esecuzione.
- **Osservazioni sull'implementazione in Java**: presenta gli aspetti particolari che riguardano l'implementazione del pattern in Java.

# Proprietà

I pattern:

- Costituiscono un vocabolario comune per i progettisti;
- Sono una notazione abbreviata per comunicare efficacemente principi complessi;
- Aiutano a documentare l'architettura software;
- Catturano parti critiche di un sistema in forma compatta;
- Mostrano più di una soluzione;
- Descrivono astrazioni software;
- NON costituiscono una soluzione precisa di problemi progettuali;
- NON risolvono tutti i problemi progettuali;
- NON si applicano solo alla progettazione OO, ma anche ad altri domini.

# Nome

Per identificare un pattern si utilizza un nome.

- Il nome del pattern è molto utile per descrivere il problema, la sua soluzione ed il suo uso;
- Esso è composto da una o due parole;
- Bisogna cercare di omogeneizzare i vocabolari personali di tutti i colleghi.

# Problema

Un pattern si applica per risolvere un problema che si presenta nella fase di modellazione.

- Descrive quando applicare un pattern definendo il contesto ed il dominio di appartenenza;
- In generale include la lista di condizioni che devono essere valide per poter giustificare l'uso di un determinato pattern.

# Soluzione

Il pattern coinvolge componenti con particolari vincoli.

- Descrive gli elementi che verranno usati durante la modellazione;
- Descrive le relazioni e le responsabilità degli elementi;
- E’ importante capire che la soluzione non rappresenta una specifica implementazione o caso d’uso ma un modello che si applica a differenti situazioni.

# Conseguenze

L’applicazione di un pattern mostra vantaggi e svantaggi.

- Raccoglie l’elenco dei tempi e dei risultati;
- E’ importante quando si devono prendere decisioni di modellazione;
- Descrive varie metriche, i costi ed i tempi in relazione ai benefici che il pattern introdurrebbe.

# Scelta

Esistono numerosi pattern più o meno adatti al problema da modellare.

- Esistono numerosi cataloghi di pattern;
- Solitamente sono descritti attraverso una notazione comune "Design Language";
- E’ importante reperire il pattern adeguato per il proprio specifico dominio;
- Considerare come un pattern risolve il problema: ogni pattern affronta il probelma con una soluzione originale;
- Considerare il suo intento: l’obiettivo del programma deve essere lo stesso del pattern;
- Studiare le interazioni tra pattern: i pattern generalmente sono composizionali;
- Considerare come deve variare il progetto: diversi approcci con diversi pattern.

# In java !?

I pattern sono utilizzati abbondantemente dalle classi standard di Java.

Iteraror, Observer e molti altri sono stati introdotti già dalle prime versioni.

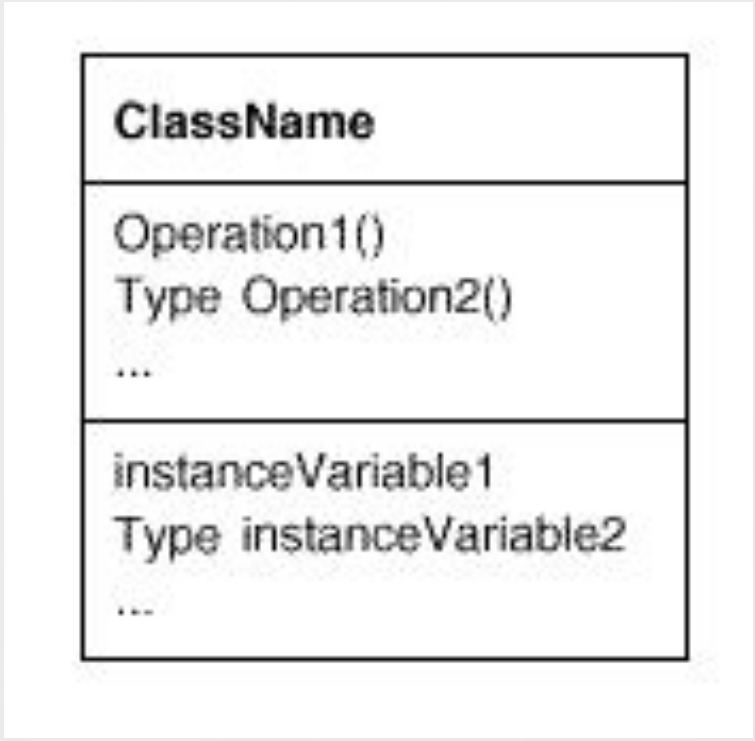
I pattern calzano perfettamente e traggono i maggiori benefici dal polimorfismo e dall'ereditarietà dei linguaggi di programmazione orientati agli oggetti.

# Notazione: Object Modeling Technique

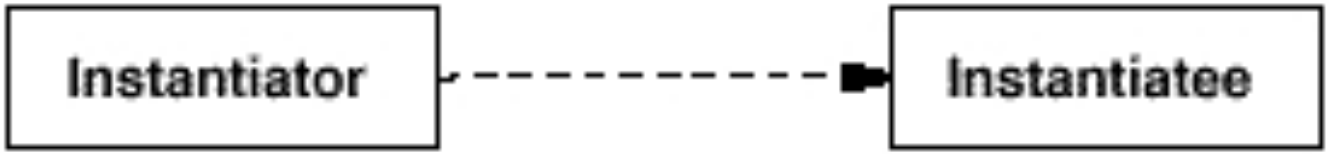
In pratica UML:

- Diagramma delle classi
- Diagramma degli oggetti
- Diagrammi di iterazione

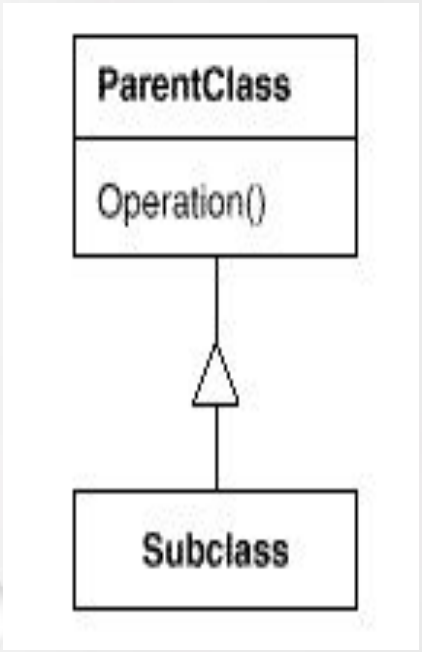
# Classi

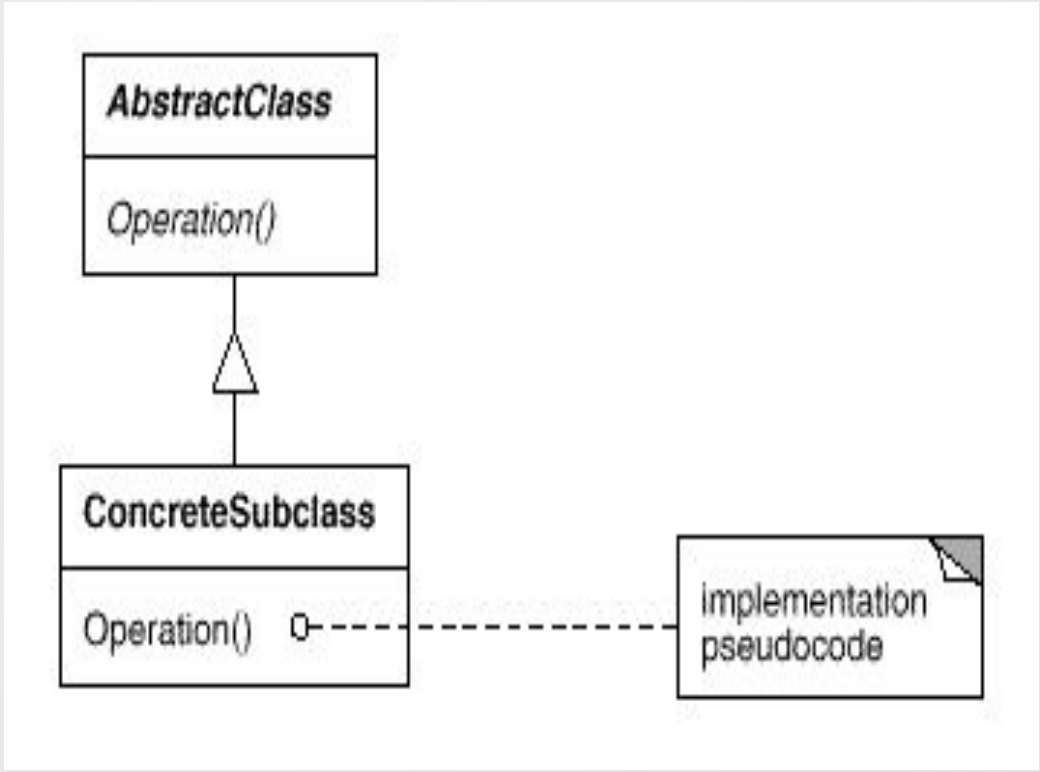

ClassName
Operation1()
Type Operation2()
...
instanceVariable1
Type instanceVariable2
...

# Istanzazione oggetti

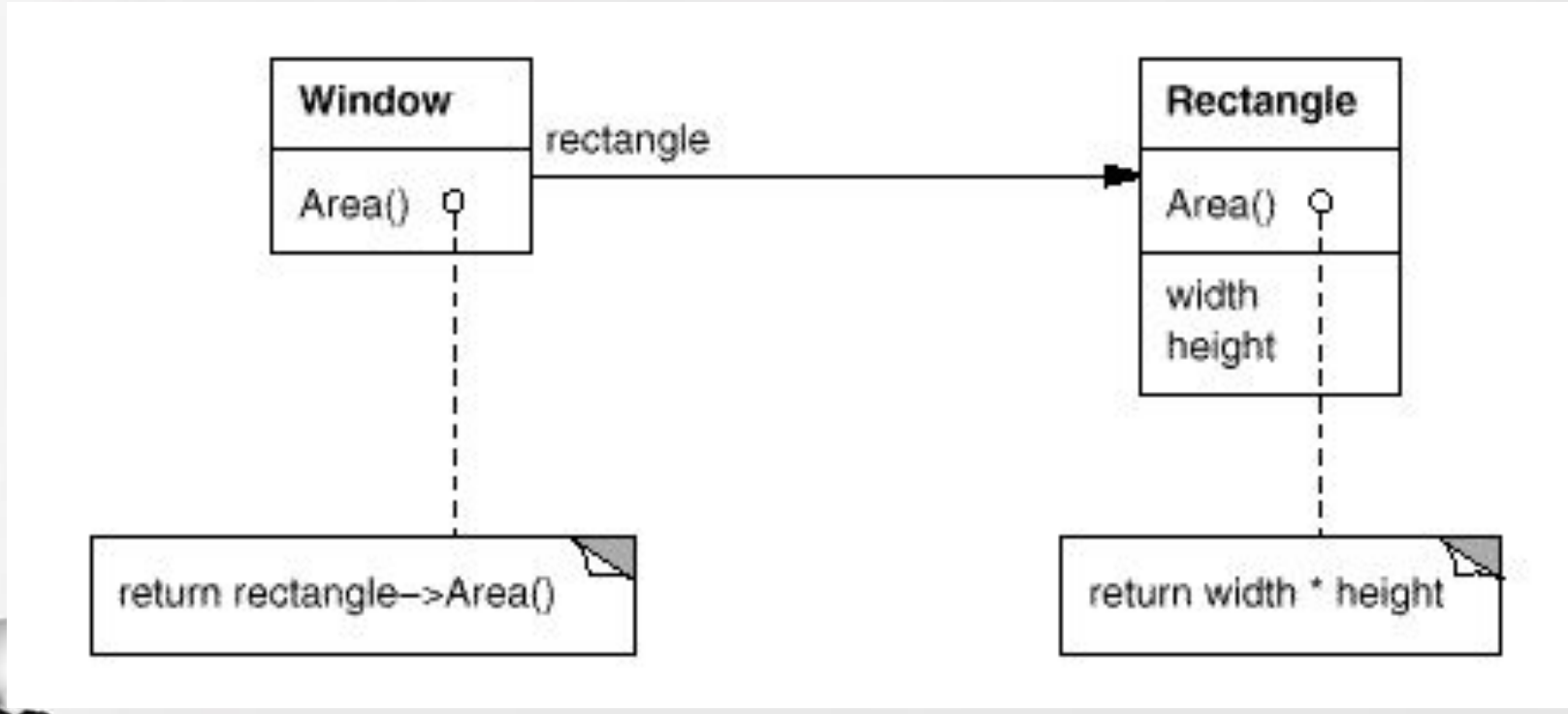



# Gerarchie e classi astratte

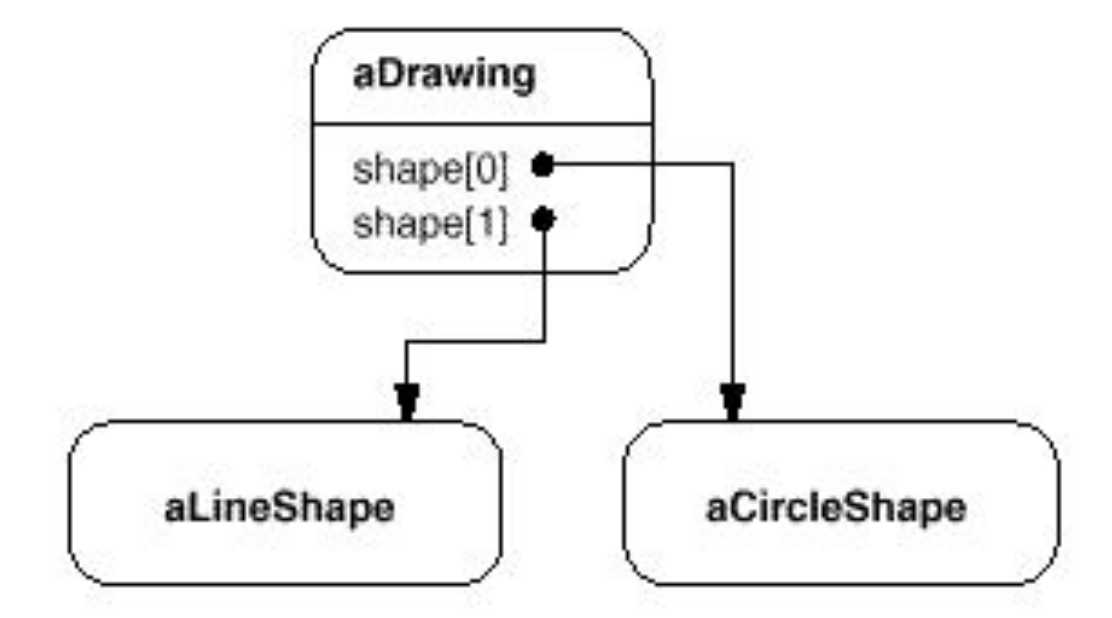


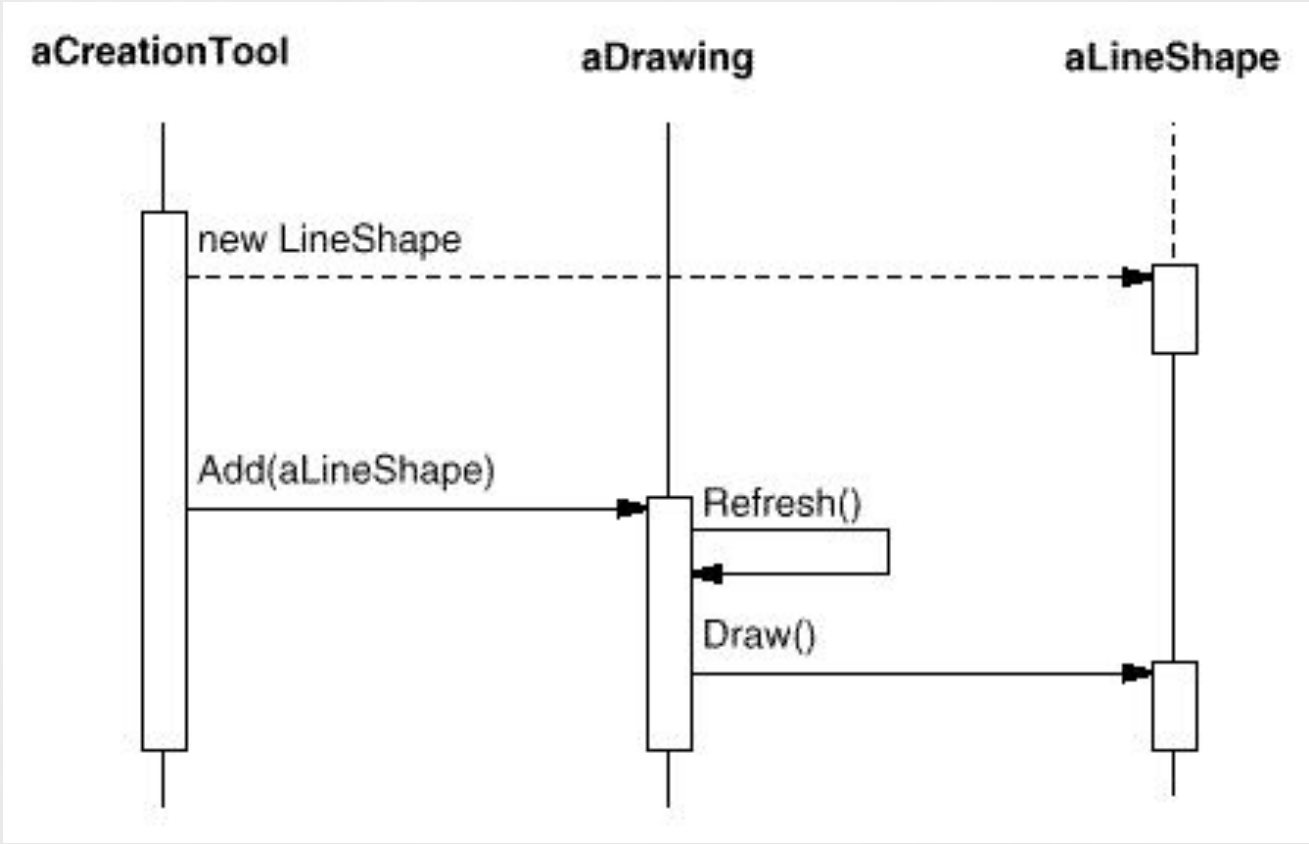

# Pseudo codice e contenitori




# Diagramma degli oggetti




# Diagramma di iterazione




# Principi sottostanti

Basarsi su classi astratte per nascondere le differenze fra le sottoclassi dei client

Classe dell'oggetto vs. Tipo del oggetto

- Le classi definiscono come un oggetto è implementato
- I tipi definiscono l'interfaccia del oggetto
- Programmare verso un interfaccia non verso un implementazione

# Principi sottostanti

- Black-box vs. white-box
  - Il riuso mediante black-box si basa sui riferimenti degli oggetti, generalmente attraverso l'istanziane delle variabili
    - Preferito in contesto di information hiding, flessibilità al runtime, eliminazione delle dipedneze implementatitive
    - L'efficienza run-time ne soffre Un elevato numero di istanze e overhead di comunicazione

- Favorire la composizione anzichè l'eriditarietà

# Principi sottostanti

- Delegare
  - Tecnica potente quando combinata con black box reuse
    - Permettere la delgazione di differenti istanza a run-time, fintantochè le istanze rispondo a messaggi simili
  - Però:
    - Quaclhe volta il codice è difficle da leggere
      - Efficienza perchè uso Black box

# Tassonimia

I design patterns possono essere raggruppati secondo il pricipale contesto di applicazione in:

- Pattern di creazione delegati alla gestione della costruzione di oggetti;
- Pattern di struttura delegati alla rappresentazione di oggetti;
- Pattern di comportamento delegati al comportamento dinamico degli oggetti.

# Tassonomia

| | |
|---|---|
| Creazionali | Pattern di creazione di nuove istanze |
| Strutturali | Pattern di strutturazione dati/comunicazione |
| Comportamentali | Gestione del comportamento negli algoritmi |

# Creazionali

- **Abstract Factory** - Crea oggetti appartenenti a famiglie di classi senza specificare le classi concrete.
- **Builder** - Separa il processo di creazione di un oggetto dalla rappresentazione definitiva.
- **Factory Method** - Crea oggetti derivanti da diversi tipi di classi.
- **Prototype** - Creazione di oggetti a partire da altri oggetti.
- **Singleton** - Una classe di cui può esistere solo un singolo oggetto.

# Strutturali

- **Adapter** - Realizza interfacce per differenti classi.
- **Bridge** - Separa l'interfaccia di un oggetto dalla sua implementazione.
- **Composite** - Oggetti composti da oggetti con la stessa struttura.
- **Decorator** - Aggiunge dinamicamente funzionalità ad un oggetto.
- **Facade** - Una singola classe che rappresenta un intero sottosistema.
- **Flyweight** - Una rappresentazione fine di istanze efficentemente condivise.
- **Proxy** - Un oggetto rappresenta un altro oggetto.

# Comportamentali

- **Chain of Responsibility** - Un modo di passare richieste tra una catena di ggetti.
- **Command** - Incapsula richieste di comandi ad un oggetto.
- **Interpreter** - Un modo di includere elementi di linguaggio in un programma.
- **Iterator** - Scansione degli elementi di una collezione.
- **Mediator** - Definisce un modo semplificato di comuncazione tra classi.
- **Memento** - Congela e ripristina lo stato interno di una classe.
- **Observer** - Un modo di notificare cambiamenti ad un insieme di classi.
- **State** - Modifica del comportamento degli oggetti al cambiamento dello stato.
- **Strategy** - Incapsulamento di algoritmi nelle classi.
- **Template Method** - Rimanda l'esatto passo di elaborazione di un algoritmo ad una sottoclasse.
- **Visitor** - Aggiunge una nuova operazione senza cambiare la classe.

# Creazionali

- Questi pattern permettono di descrivere come vengono creati gli oggetti.
- L'idea è di astrarre il modo in cui sono creati gli oggetti per dover fare esplicitamente new il meno possibile.
- Un pattern di creazione aiuta a rendere un sistema indipendente da come gli oggetti sono creati, composti e rappresentati.
- Esistono due temi ricorrenti circa i pattern di creazione:
  - Incapsulare la conoscenza circa quale classe concreta il sistema utilizzi;
  - Nascondere il modo in cui istanze di classi siano create e messe assieme.

# Factory Method

Separa la responsabilità di istanziare una classe dalla responsabilità di scegliere quale classe istanziare.

Noto come: Virtual Constructor.

# Scopo

- Il design pattern Factory Method definisce un'interfaccia (Creator) per ottenere una nuova istanza di un oggetto (Product) delegando ad una classe derivata (ConcreteCreator) la scelta di quale classe istanziare (ConcreteProduct).
- La classe ConcreteCreator che determina quale classe ConcreteProduct istanziare è stabilita a design-time attraverso l'ereditarietà, quindi questo design pattern è classificato rispetto allo scopo come rivolto alle classi.
- Rispetto al fine questo design pattern è classificato tra i pattern di creazione.

# Applicabilità

Un componente od un framework può aver bisogno di delegare al programmatore che lo utilizza la scelta di quale classe istanziare. Ad esempio:

- si può lasciare al programmatore la scelta di quale classe istanziare tra quelle di una lista predefinita di classi del framework (configurazione);
- si può lasciare al programmatore la scelta di istanziare una classe del framework di default o una nuova classe da derivata da quella di default e personalizzata dal programmatore stesso (personalizzazione);
- si può lasciare al programmatore la scelta di istanziare una nuova classe da lui realizzata (estensione);

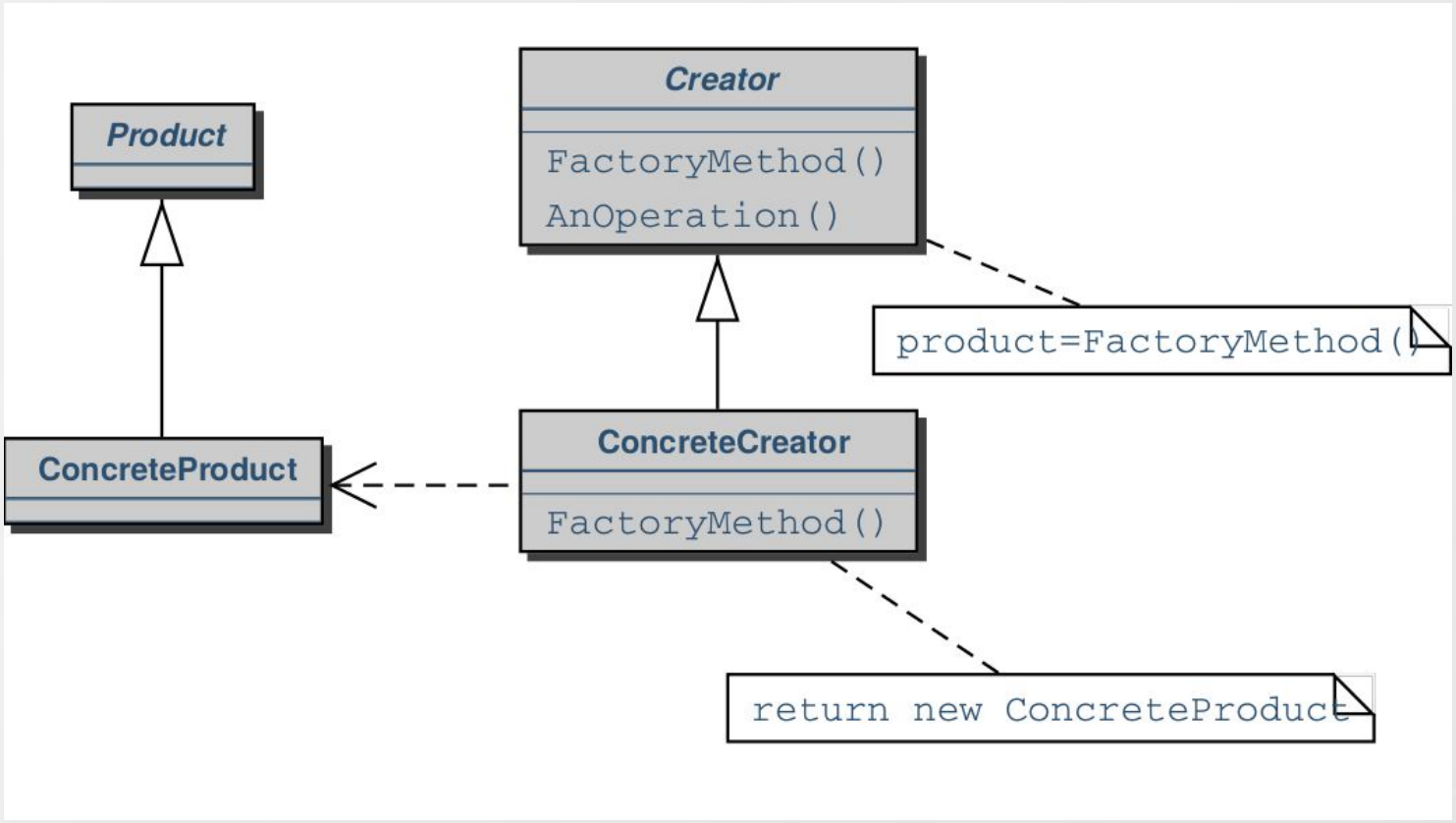
Questa necessità è assolta dal design pattern Factory Method. Esso infatti invece di richiamare direttamente il costruttore della classe da istanziare prevede l'uso di un metodo.

# Conseguenze

Maggiore modularità: la concreta gestione delle operazioni di creazione  gestione è confinata;

- Maggiore elasticità: è possibile aggiungere altri oggetti di tipo diverso senza cambiarne il loro uso;
- Maggiore flessibilità: il cliente usa diversi tipi di oggetti nello stesso modo.

# UML


Product
Creator
FactoryMethod()
AnOperation()
product=FactoryMethod()
ConcreteProduct
ConcreteCreator
FactoryMethod()
return new ConcreteProduct

# Comportamento

- Product definisce l'interfaccia dell'oggetto creato dal factory method.
- ConcreteProduct implementa l'interfaccia di Product.
- Creator dichiara il factory method che produce un oggetto di tipo Product e lo può invocare per creare un oggetto di tipo Product.
- Il creator può definire un'implementazione del factory method che produce un oggetto ConcreteProduct di default.
- ConcreteCreator ridefinisce il factory method per produrre un'istanza di un ConcreteProduct

# Esempio

Per esempio, ci sono due versioni A1 e A2 di una class A:

```
abstract class A { public abstract String getVal(); }

class A1 extends A {
    private String val;
    A1(String val) { this.val = val; }
    public String getVal() { return "A1: " + val; }
}
class A2 extends A {
    private String val;
    A2(String val) { this.val = val; }
    public String getVal() { return "A2: " + val; }
}
```

# Esempio

La fattoria permette di astrarre come si fa la scelta fra A1 e A2:

```
class AFactory {
    public static final int MAX_LENGTH = 3;
    public AFactory() { } // Costruttore
    public static boolean test(String s) {
     return s.length() < MAX_LENGTH;
     }
    public static A get(String s) {
     if (test(s)) { return new A1(s); }
           return new A2(s);
    }
}
```

# Esempio

Adesso si creano gli oggetti di tipo A attraverso Factory

```
A a = AFactory.get("ab"), b = AFactory.get("abc");
System.out.println(a.getVal());
System.out.println(b.getVal());
```

È lo stesso codice, ma adesso a.getVal() produce A1: ab, invece b.getVal() produce A2: abc.

Vantaggi
Abbiamo astratto la creazione di un oggetto di tipo A. Se si vuole cambiare come sono creati gli oggetti di tipo A bisogna cambiare solamente la classe AFactory.

# Abstract Factory

- Il design pattern Abstract Factory definisce un’interfaccia ("AbstractFactory") tramite cui istanziare famiglie (famiglia 1, 2, ...) di oggetti (AbstractProductA, AbstractProductB, ), tra loro correlati o comunque dipendenti, senza indicare da quali classi concrete (ProductA1 piuttosto che ProductA2, ...).
- La scelta delle classi concrete è delegata ad una classe derivata (ConcreteFactory1 per la famiglia 1, ConcreteFactory2 per la famiglia 2, ...).

Noto come: kit.

# Scopo

Fornire un’interfaccia per creare famiglie di oggetti dipendenti senza specificare le classi concrete.

# Applicabilità

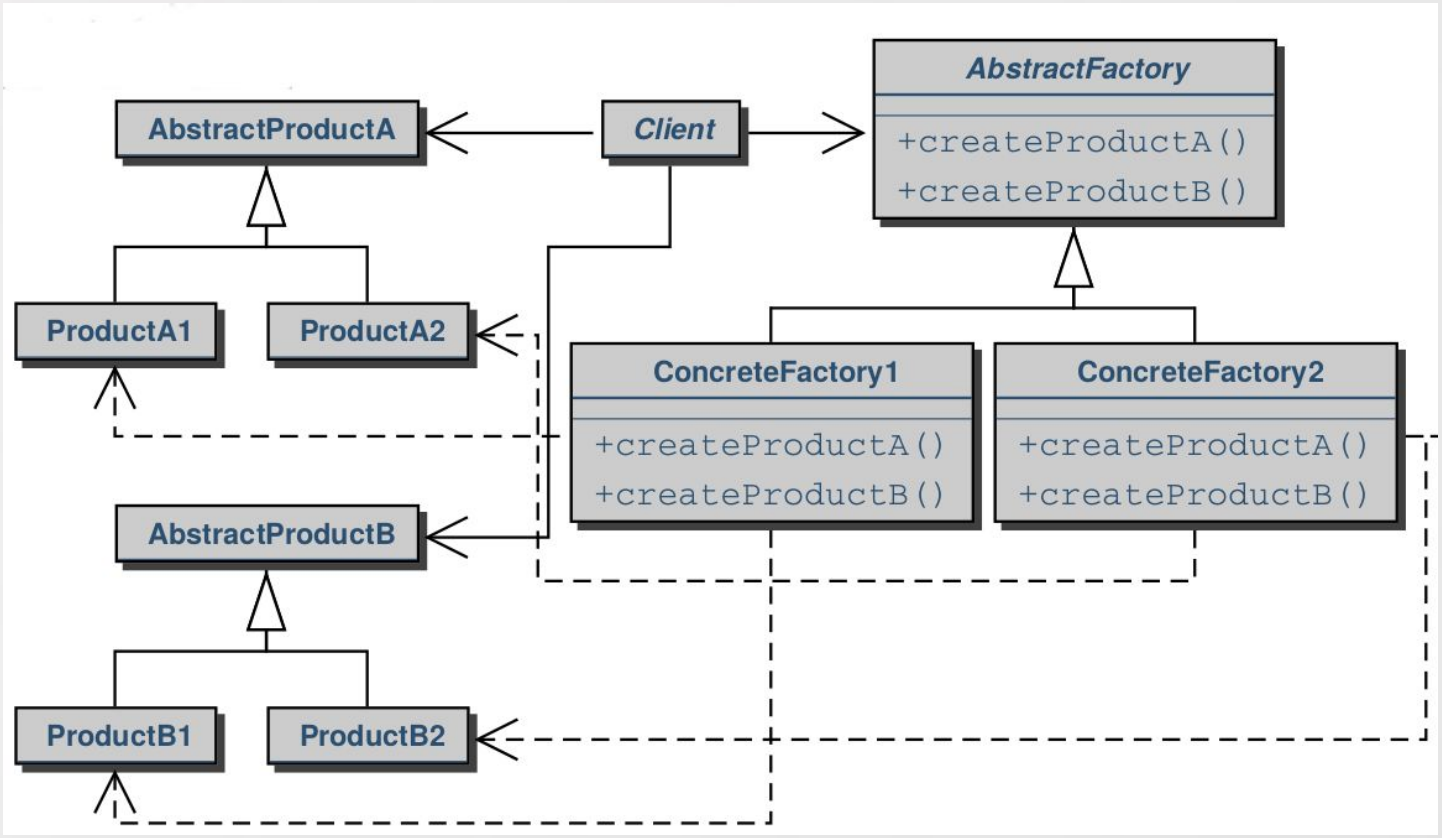
- Realizzare un sistema indipendente da come i prodotti sono creati, composti e rappresentati;
  - Il sistema deve essere configurato con famiglie multiple di prodotti;
  - Mettere a disposizione soltanto l'interfaccia, non l'implementazione, di una libreria di classi.

# Conseguenze

Isola le classi concrete;

- Facilita la portabilità;
- Aumenta la consistenza tra i prodotti;
- Per contro, inserire nuovi prodotti risulta complicato, in quanto implica cambiamenti all'Abstract Factory.

# UML


AbstractFactory
+createProductA()
+createProductB()
AbstractProductA
Client
ProductA1
ProductA2
ConcreteFactory1
+createProductA()
+createProductB()
ConcreteFactory2
+createProductA()
+createProductB()
AbstractProductB
ProductB1
ProductB2

# Java

Per esempio, date due versioni delle classe A e B:

```
abstract class A { }
class A1 extends A {
        private String val;
        A1(String val) { this.val = val; }
}

class A2 extends A {
        private String val;
        A2(String val) { this.val = val; }
}
```

# Esempio

```
abstract class B { };

class B1 extends B {
        private int val;
        B1(int val) { this.val = val; }
}

class B2 extends B {
        private int val;
        B2(int val) { this.val = val; }
}
```

# Esempio

Una fattoria ha il compito di dare un insieme compatibile di A e B:

```
abstract class AbAbstractFactory {
        public abstract A getA(String val);
        public abstract B getB(int i);
}
```

# Esempio

Nel nostro caso A1 corrisponde a B1 e A2 a B2.
Dunque ci sono due fattorie:

```
class AbAbstractFactory1 extends AbAbstractFactory {
        public A getA(String val) { return new A1(val); }
        public B getB(int i) { return new B1(i); }
}

E

class AbAbstractFactory2 extends AbAbstractFactory {
        public A getA(String val) { return new A2(val); }
        public B getB(int i) { return new B2(i); }
}
```

# Esempio

Un esempio di utilizzo di queste fattorie è il seguente:

```
AbAbstractFactory f1 = new AbAbstractFactory1();
AbAbstractFactory f2 = new AbAbstractFactory2();

A a1 = f1.getA("ab"); // crea un oggetto di tipo A1
B b1 = f1.getB(1); // crea un oggetto di tipo B1
A a2 = f2.getA("ab"); // crea un oggetto di tipo A2
B b2 = f2.getB(2); // crea un oggetto di tipo B2
```

# Singleton

Il design pattern Singleton permette di assicurare che una classe (Singleton) abbia una unica istanza e che questa sia globalmente accessibile in un punto ben noto.

Il modo più semplice di implementare questo pattern in Java è di definire una class final con tutti i metodi statici.

# Scopo

- Costruire un unico oggetto di un determinato tipo.
- Assicurarsi che una classe abbia soltanto un’istanza, e fornirne un unico punto di accesso.

# Applicabilità

Il Singleton andrebbe usato:

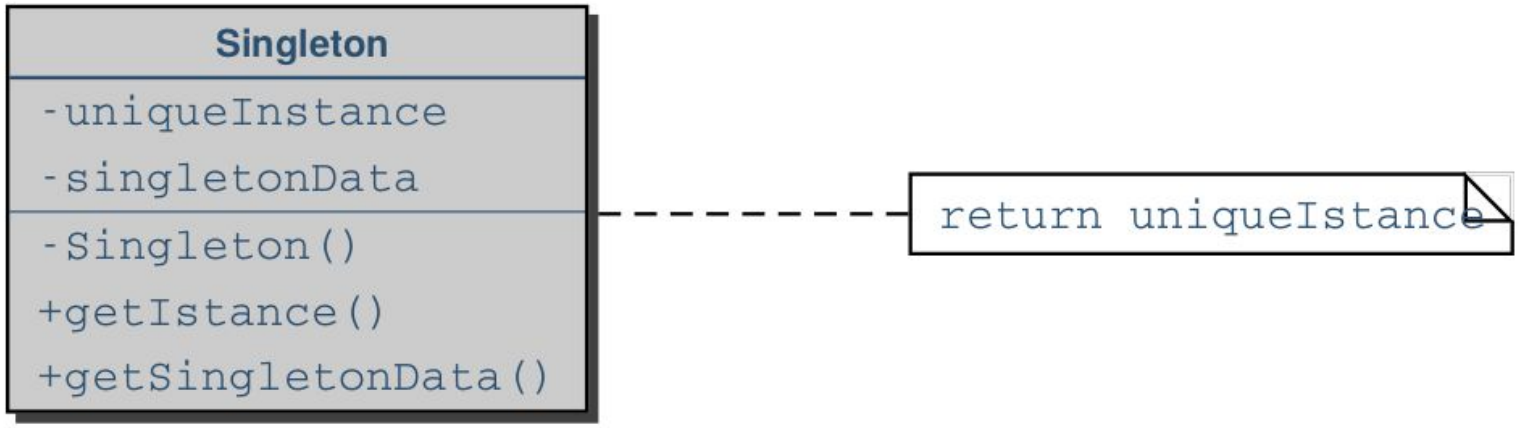
- Quando si vuole la garanzia che nel sistema vi sia una solo istanza di oggetti di un determinato tipo;
- Quando non si vuole delegare ad altri il controllo di unicità di un oggetto;
- Quando più oggetti devono condividere un unico pool di dati;
- Deve esserci esattamente una singola istanza di una classe, e deve essere accessibile da un punto di accesso ben preciso;
- Quando tale istanza deve essere estensibile tramite subclassing, i client dovrebbero poter utilizzare l'istanza estesa senza modificare il proprio codice.

# Conseguenze

Accesso controllato all'istanza singola;

- Name space ridotto;
- Raffinamento di operazioni e rappresentazione delle stesse;
- Possibilità di usare un numero variabile di istanze;
- Maggiore flessibilità rispetto all'uso degli static member;
- Quindi si ha:
  - Maggiore correttezza: anche volendo non è possibile produrre oggetti distinti;
  - Maggiore modularità: la gestione dei dati gestisti dal singleton può essere remotizzata;
  - Maggiore flessibilità: la modifica dei dati gestiti dal singleton non interferisce con l'unicità dell'stanza.

# UML


Singleton
-uniqueInstance
-singletonData
-Singleton()
+getIstance()
+getSingletonData()
return uniqueIstance

# Esempio

```
import java.io.*;

class EsempioDiSingleton { // System
      public static void main(String[] args) {
               PrintStream o1 = System.out, o2 = System.out;
               if (o1 == o2) o1.println("Single istance");
       }
}
```

Oppure

```
class EsempioDiSingleton { // Math
        public static void main(String[] args) {
               final double x = 2.0;
               System.out.printf("sqrt(%f)=%f",x,Math.sqrt(x));
}
```

# Esempio

Ponendo il costruttore con visibilità privata di fatto si impedisce la creazione di oggetti. La creazione del singleton avviene solo con l'invocazione del metodo getIstance() che produce sempre il riferimento all'unica istanza. Ad esempio:

```
class Singleton {
      private static Singleton uniqueIstance = null;
      private int singletonData;
      private Singleton() { // generazione del dato unico
      singletonData = (int)(Math.random()*10);
}

public static Singleton getSingleton() {
       if (uniqueIstance==null) uniqueIstance = new Singleton();
              return uniqueIstance;
}
public int getSingletonData() { retun singletonData; }
```

# Esempio di applicazione: Sistema di visualizzazione a finestre

Necessità di portabilità per differenti Window system

Molto simile al problema look and feel ma differenti vendors fanno widget differenti

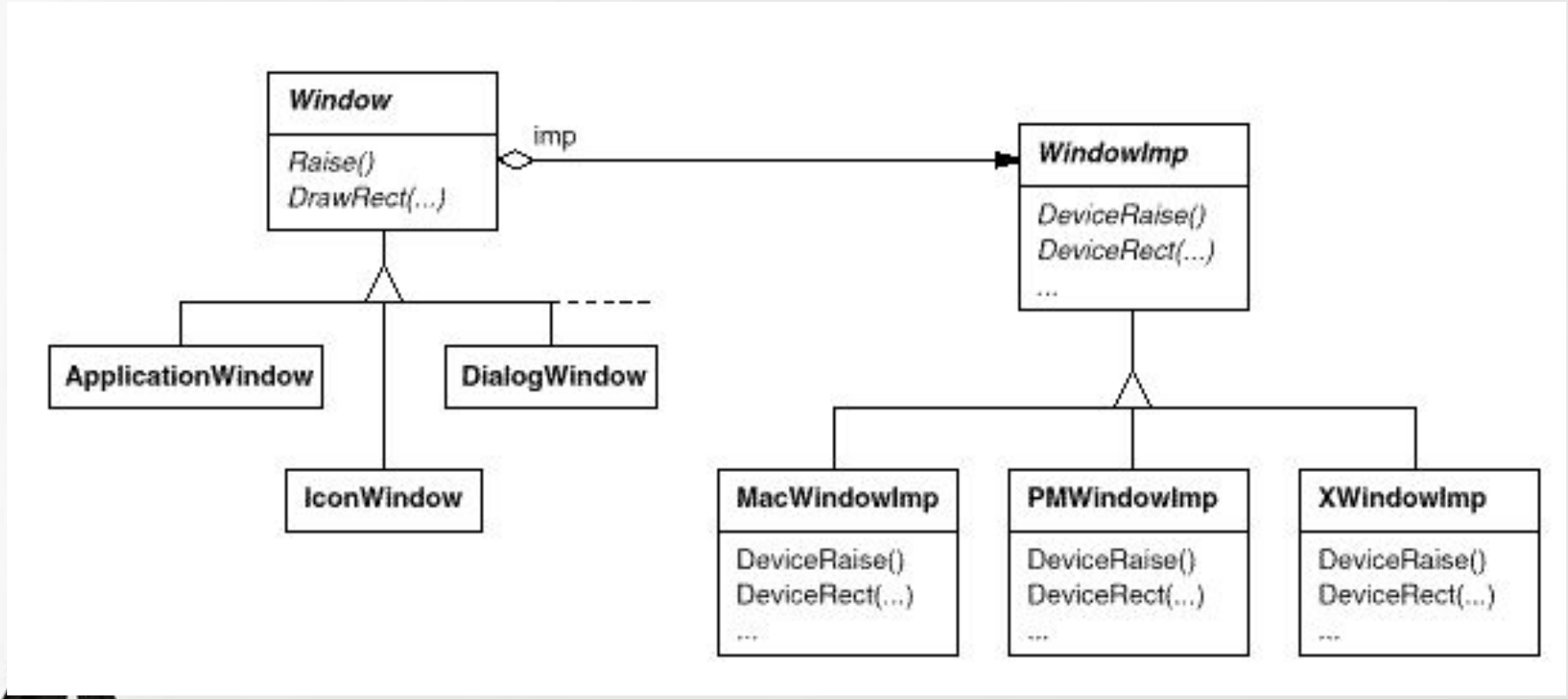
Soluzione

Definire una classe astratta Window con funzionalità base (draw, iconify, move, resize, etc.)

Definire sottoclassi concrete per ogni tipo specifico di Window (dialog, application, icon, etc.)

Definire Gerarchia WindowImp hierarchy per gestire implementazione per ogni vendor

# Esempio


Window
Raise()
DrawRect(...)
imp
WindowImp
DeviceRaise()
DeviceRect(...)
...
ApplicationWindow
DialogWindow
IconWindow
MacWindowImp
DeviceRaise()
DeviceRect(...)
...
PMWindowImp
DeviceRaise()
DeviceRect(...)
...
XWindowImp
DeviceRaise()
DeviceRect(...)
...

# Builder

- Il design pattern Builder separa la costruzione (il metodo Construct dell'oggetto Director è l'algoritmo di costruzione che compone le parti e ogni metodo di ConcreteBuilder costruisce una parte) di un oggetto complesso dalla sua rappresentazione (Product) in modo tale che lo stesso processo di costruzion (il metodo Construct dell'oggetto Director) possa essere usato per creare diverse rappresentazioni (per ogni rappresentazione ci sarà un suo ConcreteBuilder che produce un diverso Product).

- Il builder è una variante della Abstract Factory in cui l'oggetto da produrre è Composite.
  - Dunque la chiamata di un Builder può implicare la creazione di diversi oggetti. Il caso tipico è quello di un'applicazione con una interfaccia grafica. In tal caso c'è sempre un Builder che deve preoccuparsi della costruzione dell'interfaccia.

# Scopo

- Separare la costruzione di un oggetto complesso dalla relativa appresentazione.
  - Ovvero confina in una classe le operazioni per creare oggetti complessi mascherando a chi vuole ottenere oggetti i complessi meccanismi di costruzione.

# Applicabilità

L’algoritmo per la creazione di un oggetto complesso dovrebbe essere indipendente dalle componenti dell’oggetto stesso;

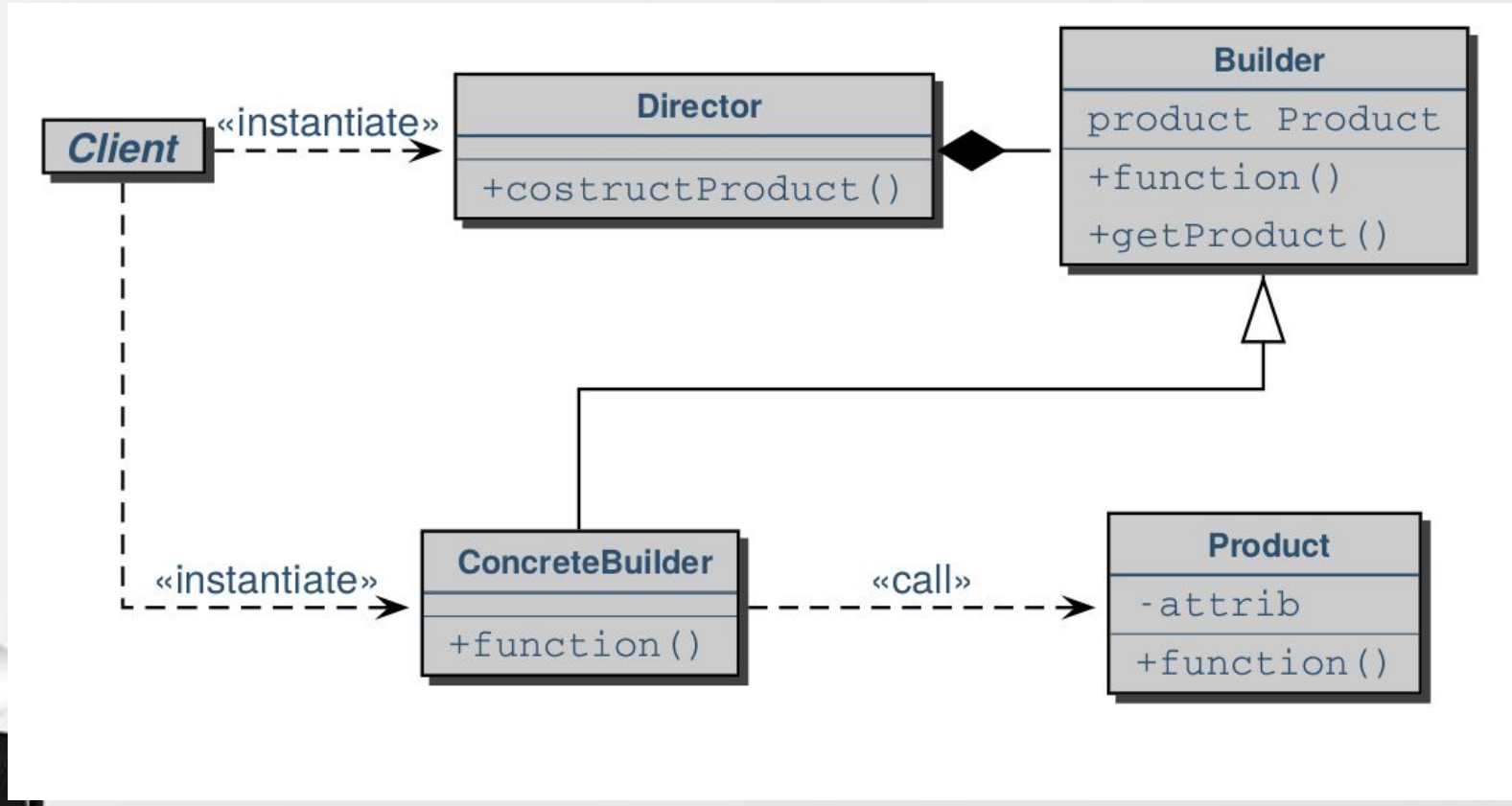
- Il processo di costruzione consente differenti rappresentazioni per l’oggetto.

# Conseguenze

E’ possibile variare la rappresentazione interna di un prodotto;

- Isola il codice per la costruzione e la rappresentazione: il Builder incapsula il modo in cui un oggetto complesso è costruito;
- Consente un miglior controllo sul processo di costruzione: il Builder consente una costruzione step-by-step del prodotto, sotto il controllo del Director.

# UML


Client
«instantiate»
Director
+costructProduct()
Builder
product Product
+function()
+getProduct()
«instantiate»
ConcreteBuilder
+function()
«call»
Product
-attrib
+function()

# Struttura

- Builder specifica l'interfaccia astratta che crea le parti dell'oggetto Product.
- ConcreteBuilder costruisce e assembla le parti del prodotto implementando l'interfaccia Builder; definisce e tiene traccia della rappresentazione che crea.
- Director costruisce un oggetto utilizzando l'interfaccia Builder.
- Product rappresenta l'oggetto complesso e include le classi che definiscono le parti che lo compongono, includendo le interfacce per assemblare le parti nel risultato finale.

# Esempio

```
/* Product */
class Pizza {
   private String base = "", salsa = "", condimento =
   "";
   public void setBase(String b) { base = d; }
   public void setSalsa(String s){ salsa = s; }
   public void setCondimento(String c) { condimento
   = c; }
}
```

# Esempio

```
/* Abstract Builder */
abstract class PizzaBuilder {
protected Pizza pizza;
public Pizza getPizza() { return pizza; }
public void createNewPizzaProduct() { pizza = new
   Pizza(); }
public abstract void buildBase();
public abstract void buildSalsa();
public abstract void buildCondimento();
}
```

# Esempio

```
/* ConcreteBuilder */
class PizzaMargherita extends PizzaBuilder {
public void buildBase() { pizza.setBase("sottile"); }
public void buildSalsa() {
   pizza.setSauce("pomodoro"); }
public void buildCondimento() {
pizza.setCondimento("acciughe");
}
}
```

# Esempio

```
/* ConcreteBuilder */
class PizzaCapricciosa extends PizzaBuilder {
public void buildBase() { pizza.setBase("spessa"); }
public void buildSalsa() { pizza.setSalsa("salsa"); }
public void buildCondimento() {
pizza.setCondimento("uova+olive+carciofini");
}
}
```

# Esempio

```
/* Director */
class Cottura {
private PizzaBuilder pizzaBuilder;
public void setPizzaBuilder(PizzaBuilder pb) {
pizzaBuilder = pb;
}
public Pizza getPizza() {
return pizzaBuilder.getPizza();
}
public void constructPizza() {
pizzaBuilder.createNewPizzaProduct();
pizzaBuilder.buildBase();
pizzaBuilder.buildSalsa();
pizzaBuilder.buildCondimento();
}
}
```

# Esempio

```
/* Client */
class Cuoco {
    public static void main(String[] args) {
    Cottura cuoce = new Cottura(); // 1
    PizzaBuilder pizzaMargherita = new PizzaMargherita(); // 2
    PizzaBuilder pizzaCapricciosa = new PizzaCapricciosa(); // 3
    cuoce.setPizzaBuilder(pizzaMargherita); // 4
    cuoce.constructPizza(); // Attiva il costruttore // 5
    Pizza pizza = cuoce.getPizza(); // 6
}
}
```

# Esempio

1. Il Cuoco (client), crea una istanza di Cottura indipendente da che cosa cuocerà
2. Il Cuoco crea una istanza di pizzaMargherita (ConcreteBuilder)
3. Il Cuoco crea una istanza di pizzaCapricciosa (ConcreteBuider)
4. Il Cuoco assegna a cuoce, istanza di Cottura, la pizzaMargherita
5. Il Cuoco invoca il costruttore di cuoce per cuocere la pizzaMarcherita
6. assegnata prima
7. Il Cuoco invoca i metodi di cuoce per accedere a funzionalità di specifiche di pizzaMargherita (Product)

# Prototype

- Il design pattern Prototype istanzia un oggetto clonandolo da un'istanza esistente.
- Esso si applica quando la creazione di un oggetto di tipo A è costosa in termini di computazione ma può essere semplificata avendo già un oggetto di tipo A.
- In Java, nella class Object è previsto un metodo clone.

# Scopo

Specificare il tipo di oggetti da creare utilizzando un'istanza prototipale, e creare nuovi oggetti copiando il prototipo.

# Applicabilità

Il pattern andrebbe usato nel caso in cui un sistema dovrebbe essere indipendente da come i prodotti sono creati, e:

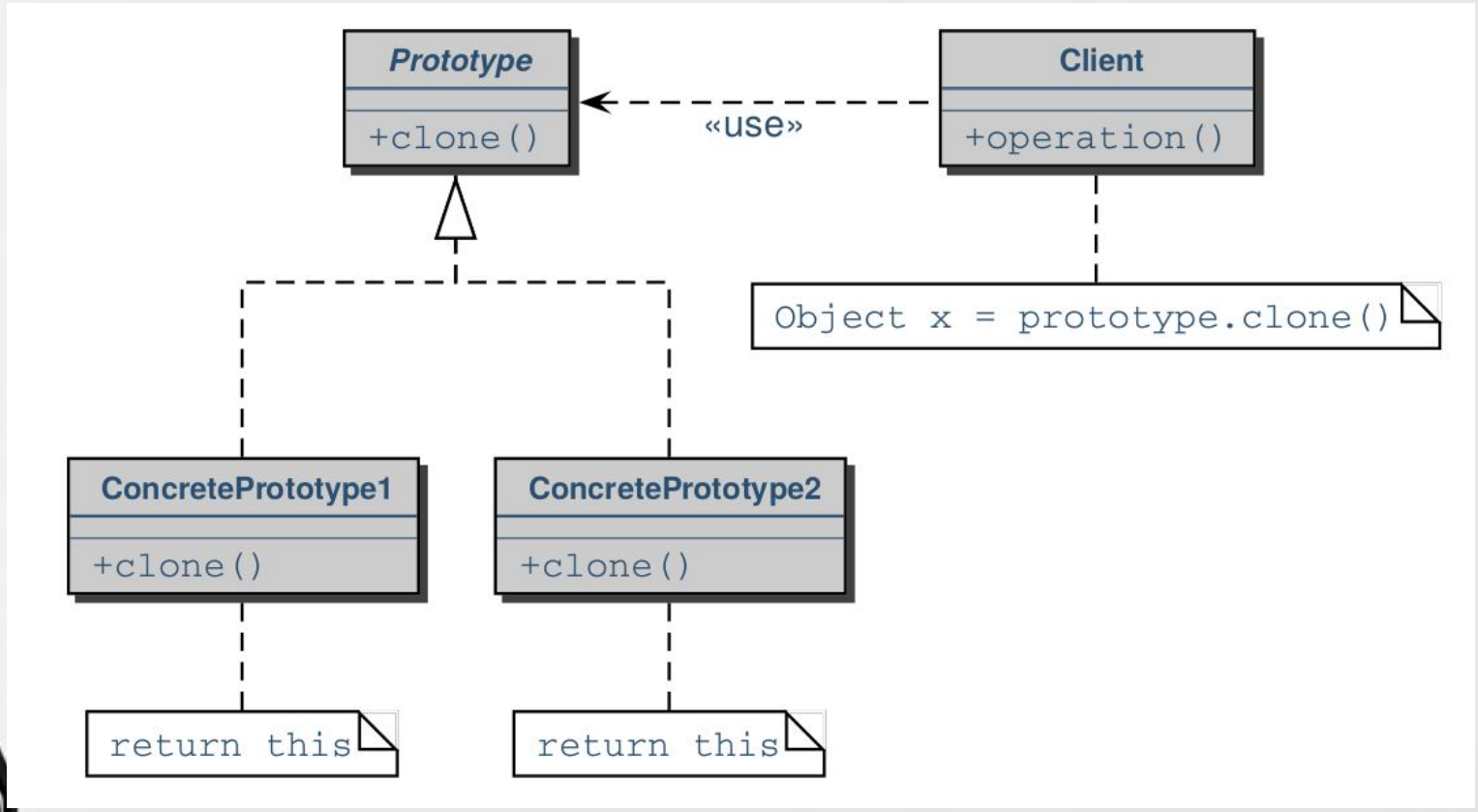
- Quando le classi da istanziare sono specificate a run-time;
- Per evitare la creazione di gerarchie di factory parallele alle gerarchie di prodotti;
- Quando le istanze di una classe possono trovarsi in soltanto una (o poche) combinazioni di stati.

# Conseguenze

(oltre a quelle dell'Abstract Factory e del Builder):

- Aggiungere o rimuovere prodotti a run-time;
- Specificare nuovi oggetti variando valori;
- Specificare nuovi oggetti variando strutture;
- Ridurre il sub-classing;
- Caricare classi dinamicamente nell'applicazione.

# UML


Prototype
+clone()
«use»
Client
+operation()
Object x = prototype.clone()
ConcretePrototype1
+clone()
ConcretePrototype2
+clone()
return this
return this

# Esempio

```
class Object {
// ...
protected Object clone() throws
   CloneNotSupportedException {
if (! (this instanceof Cloneable)) {
throw new CloneNotSupportedException();
}
// ....
}
}
```

# Esempio

- Per essere duplicato un oggetto deve implementare l'interfaccia Cloneable (che è vuota!).
- Il comportamento di default è di copiare l'oggetto ma non gli attributi.
- Per esempio, per permettere di copiare una class A fuori di A bisogna fare un overriding del metodo clone:

```
class Element implements Cloneable {
    private int i;
    int getI() { return i; }
    void setI(int i) { this.i = i; }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
}
}
```

# Esempio

Adesso si può creare un prototipo:

```
class Use {
    public operation() {
    try {
    Eelement [] array = new array[100];
    Element one = new Element();
    array[0] = one;
    one.setI(0);
    for(int i = 1, i < 100; i++) {
      array[i]=one.clone(); array[i].setI(i);
    }
    } catch (CloneNotSupportedException e) {
    }
}
```

# Strutturali

- I pattern di struttura si applicano per descrivere come è stata organizzata la struttura dei diversi oggetti. Essi riguardano il modo in cui classi e oggetti sono legati tra loro in strutture più grandi.
- Questa organizzazione può essere fatta usando l'ereditarietà (si parla allora di pattern di classi) o usando oggetti che contengono altri oggetti (si parla allora di pattern di oggetti).
- Il consiglio è di preferire sempre la seconda soluzione.

# Adapter

- Il design pattern Adatper converte l'interfaccia di una classe o di un oggetto (Adaptee) nell'interfaccia (Target) che l'utilizzatore (Client) si aspetta.
- Il design pattern Adatper permette all'utilizzatore (Client) e alla classe o oggetto (Adaptee) che altrimenti non potrebbero collaborare di lavorare insieme.
- Esso si applica quando c'è bisogno di adattare il comportamento di un oggetto B in modo tale che B propone la stessa interfaccia di un altro oggetto A.

Noto come: Wrapper.

# Scopo

si vuole adattare.

accia di una classe in un’interfaccia differente richiesta dal client: in tal modo è possibile l’interoperabilità tra classi aventi interfacce incompatibili.

Si vuole usare una classe esistente senza modificarla. Tale classe è quella da adattare (adaptee).

Il contesto in cui si vuole usare la classe adattata richiede un’interfaccia detta obbiettivo che è diversa dalla classe che si vuole adattare.

# Applicabilità

E’ consigliabile utilizzare l’adapter quando:

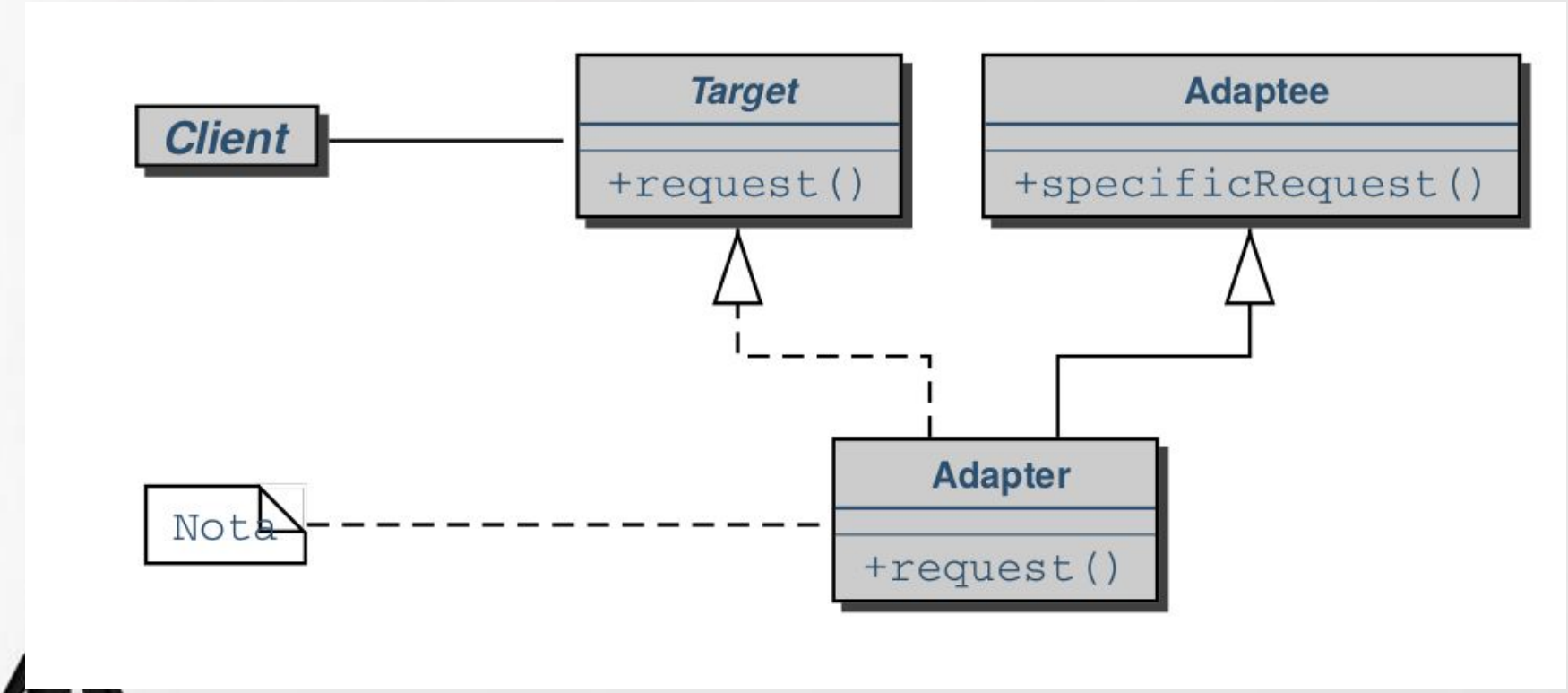
- Occorre utilizzare classi esistenti, ma le interfacce non sono compatibili con quella del client;
- Occorre creare una classe riusabile che coopera con altre classi scollegate da essa, e dunque aventi interfacce diverse;
- Occorre utilizzare diverse sottoclassi esistenti, ma non è pratico adattare l’interfaccia effettuando il subclassing di ognuna. Si può allora utilizzare un object adapter;
- Non si vuole o non si può modificare la classe da adattare.

# Conseguenze

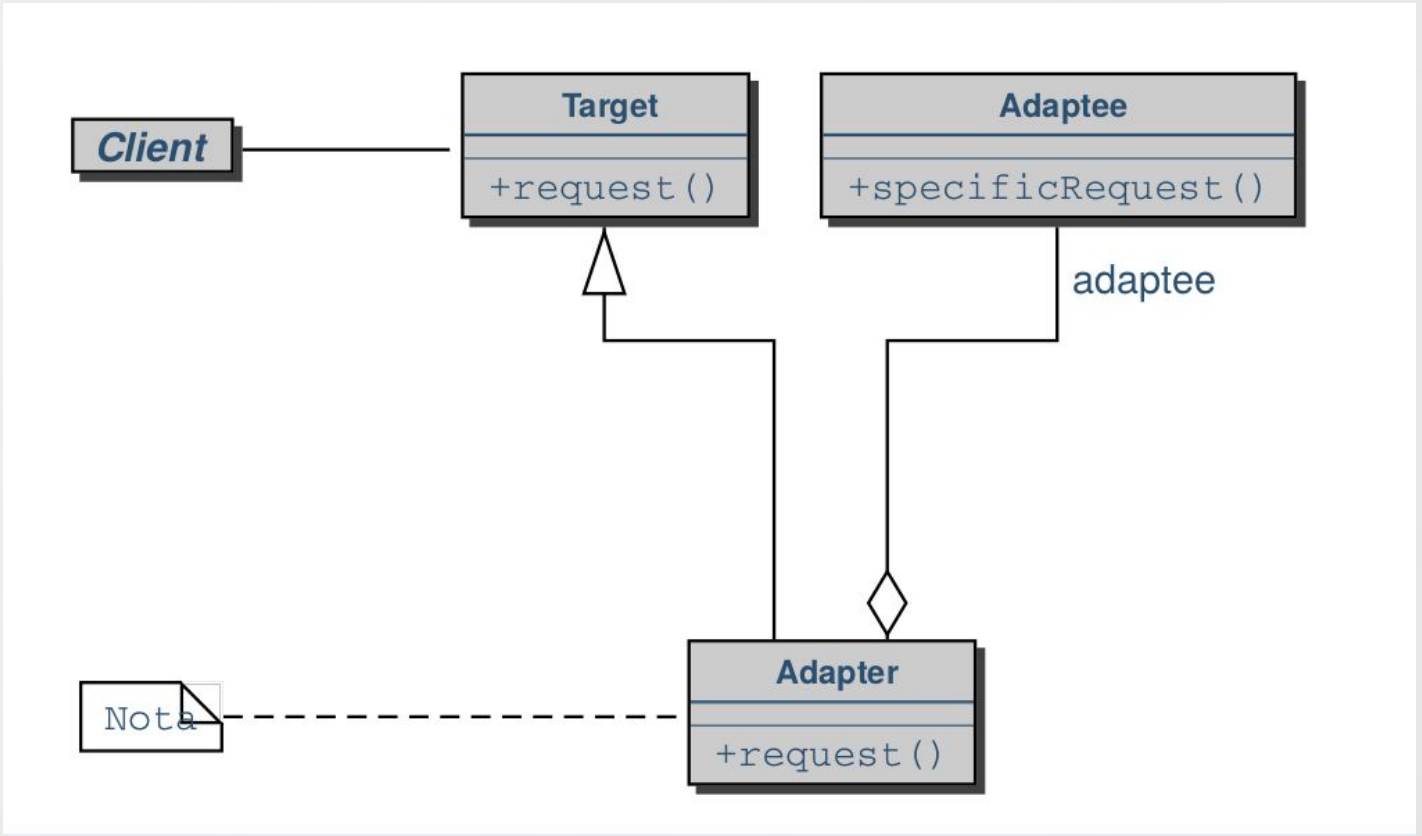
Un Class Adapter non è in grado di adattare una classe e tutte le relative sottoclassi;

- Un Class Adapter consente l'override del comportamento dell'Adaptee;
- Un Object Adapter consente ad un singolo Adapter di operare con diversi Adaptee;
- Tramite un Object Adapter l'overriding del comportamento dell'Adaptee risulta maggiormente difficoltoso: richiede il sub-classing dell'Adaptee, e richiede che l'Adapter punti a tali sub-classi piuttosto che all'Adaptee stesso.

# UML


Client
Target
+request()
Adaptee
+specificRequest()
Adapter
+request()
Nota

# UML


Client
Target
+request()
Adaptee
+specificRequest()
adaptee
Adapter
+request()
Nota

# Esempio

Per esempio, abbiamo costruito un'applicazione che permette di usare un oggetto di tipo A:

```
class A {
    public int getX() { ... }
    public int getY() { ... }
}
```

e vogliamo che si possa usare anche un oggetto B che con qualche modifica può fare tutto quello che fa A.

```
class B {
    int getXPlusY() { ... }
    int getXMinuxY() { ... }
}
```

# Esempio

Il primo passo è di rappresentare quello che può fare A come un'interfaccia:

```
interface ACapable {
    int getX();
    int getY();
}
```

e cambiare A in conseguenza:

```
class A implements ACapable {
....
```

# Esempio

Adesso esistono due possibilità:

Class Adapter

Si può usare l'ereditarietà per adattare B:

```
class AClassAdapter extends B implements ACapable {
      public int getX() {
            return (getXPlusY() + getXMinusY()) / 2;
      }
     public int getY() {
            return (getXPlusY() - getXMinusY()) / 2;
      }
}
```

# Esempio

Object Adapter

Si può creare un nuovo oggetto che contiene l'oggetto B:

```
class AObjectAdapter implements ACapable {
      private B b;
      public int getX() {
                  return (b.getXPlusY() + b.getXMinusY()) / 2;
      }
     public int getY() {
                  return (b.getXPlusY() - b.getXMinusY()) / 2;
      }
}
```

Quest'ultima soluzione è da preferire.

# In java

- Adaptee InputStream
- Target Reader
- Adapter InputStreamReader
- Client La classe che vuole leggere testo da un flusso in ingresso
- targetMethod read
- adapteeMethod read

# Bridge

- Il bridge pattern permette di separare l'interfaccia di una classe (che cosa si può fare con la classe) dalla sua implementazione (come si fa).
  - In tal modo si può usare l'ereditarietà per fare evolvere l'interfaccia o l'implementazione in modo separato ed autonomo e sia possibile sostituire l'implementazione senza conseguenze per l'utilizzatore (Client).

Noto come: Handle/Body, Driver.

# Scopo

Disaccoppiare un’astrazione dalla relativa implementazione in maniera tale da consentire ad entrambe di variare in maniera indipendente.

# Applicabilità

Può essere conveniente utilizzare il bridge nei seguenti casi:

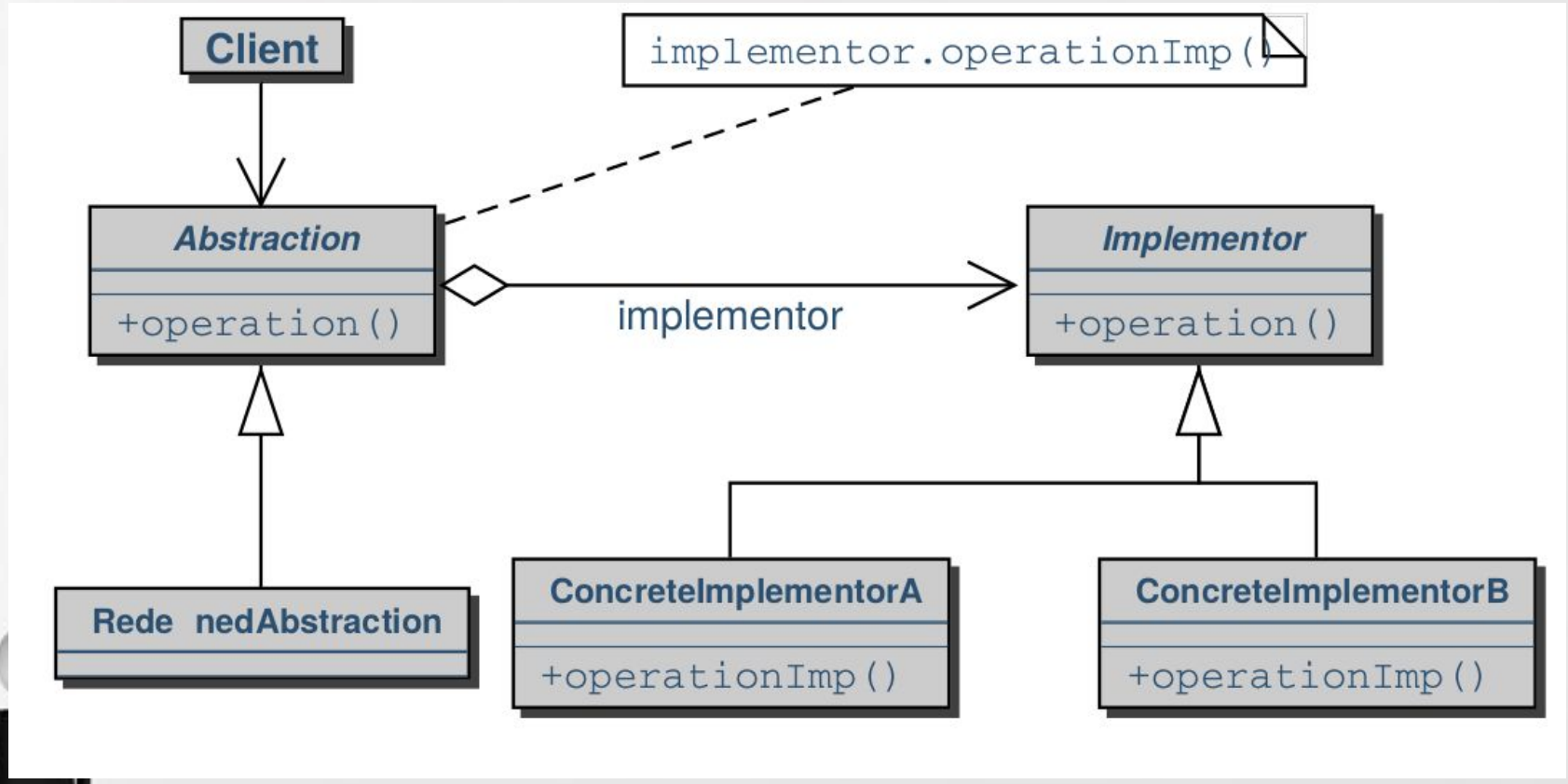
- Si desidera evitare un binding permanente tra astrazione e interfaccia;
- Occorre rendere flessibili astrazione e implementazioni mediante sub-classing;
- Le modifiche all'implementazione non dovrebbero avere impatto sui client;
- Rendere l'implementazione completamente invisibile ai client;
- Condividere un'implementazione tra oggetti multipli.

# Conseguenze

Disaccoppiare interfaccia e implementazione: ciò riduce anche necessità di ricompliazioni continue dell'Abstraction durante la fase di sviluppo e incoraggia lo sviluppo di sistemi a layer;

- Migliorare l'estensibilità;
- Nascondere i dettagli implementativi ai client.

# UML


Client
implementor.operationImp()
Abstraction
+operation()
implementor
Implementor
+operation()
Rede nedAbstraction
ConcreteImplementorA
+operationImp()
ConcreteImplementorB
+operationImp()

# Esempio

- Vogliamo realizzare funzionalità e realizzazioni per disegnare cerchi.
- La parte che riguarda la sezione implementattiva è definita con una interfaccia che permette successivamente di essere implementata con diverse classi

# Esempio

```
/* Implementor */
interface Cerchio {
public void disegnaCerchio(double x,
  double y, double raggio);
}
```

# Esempio

Due realizzazioni di implementazione

```
/* ConcreteImplementorA*/
class DisegnaCerchio1 implements Cerchio {
	public void disegnaCerchio(double x, double y, double r) {
		System.out.printf("Cerchio DisegnaCerchio1(%f,%f,&f)",x,y,r);
	}
}
```

e

```
/* ConcreteImplementorB*/
class DisegnaCerchio2 implements Cerchio {
	public void disegnaCerchio(double x, double y, double r) {
		System.out.printf("Cerchio DisegnaCerchio2(%f,%f,&f)",x,y,r);
	}
}
```

# Esempio

La parte che riguarda la sezione descrittiva è definita con una interfaccia che permette successivamente di essere estesa con ulteriori funzionalità

```
/* Abstraction */
interface Disegno {
public void disegna();
public void ridimenzionaInPercentuale(double p);
// funzione aggiunta non definita nell'implementazione
}
```

# Esempio

```
/* RefinedAbstraction */
class DisegnaCerchio implements Disegno {
        private double x, y, r;
        private Cerchio d; // associazione implementor
        DisegnaCerchio(double x, double y, double r, Cerchio i)
        {
            this.x = x, this.y = y; this.r = r, this.d = i;
        }
public void disegna() { d.disegnaCerchio(x,y,r); }
public void ridimenzionaInPercentuale(double p) { r *= p; }
}
```

# Esempio

```
/* Client */
class Bridge {
     public static void main(String [] args) {
     Disegno [] disegni = new Disegno[2];
    disegni[0] = new DisegnaCerchio(1,2,3,new DisegnaCerchio1());
    disegni[1] = new DisegnaCerchio(5,7,11,new DisegnaCerchio2());
    for (int i = 0; i < disegni.legth; i++) {
          disegni[i].ridimensionaInPercentuale(2.5);
          disegni[i].disegna();
    }
    }
}
```

# Strutturali: Composite

Scopo:

comporre ad albero elementi semplici e composti;

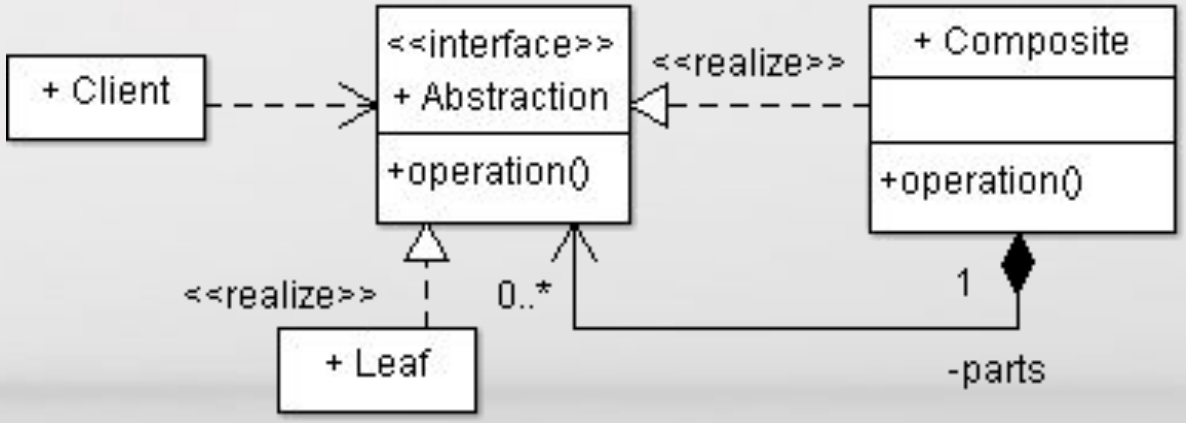
si pensi a oggetti grafici e gruppi di oggetti grafici;

Ingredienti:

gerarchia di astrazione;

classe aderente alla gerarchia composta con elementi della stessa (dell'interfaccia/superclasse astratta);

operazioni delegate agli elementi della composizione.



# Strutturali: Decorator

Scopo:

"decorare" il comportamento di una classe concreta;

rendere tale differenza trasparente al client;

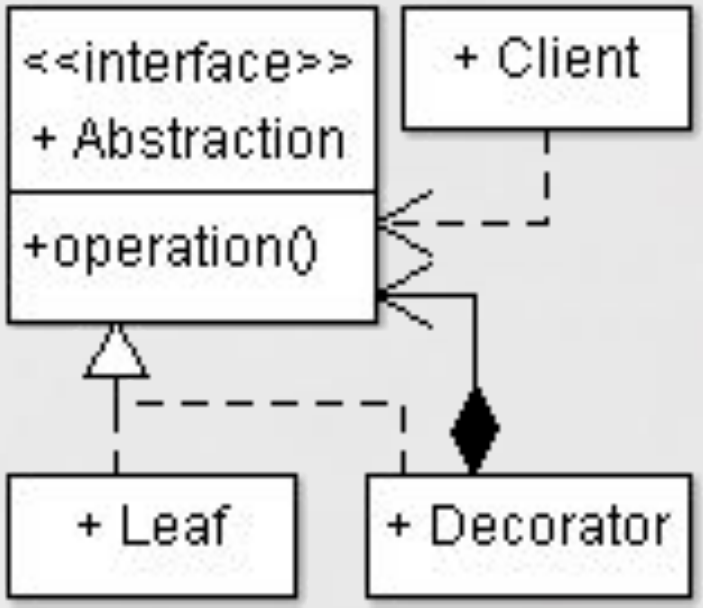
Ingredienti:

gerarchia di astrazione;

sottoclasse della classe da decorare (Leaf);

operazioni delegate alla superclasse concreta…

…successivamente modificate con codice aggiuntivo.




# Pattern Decorator

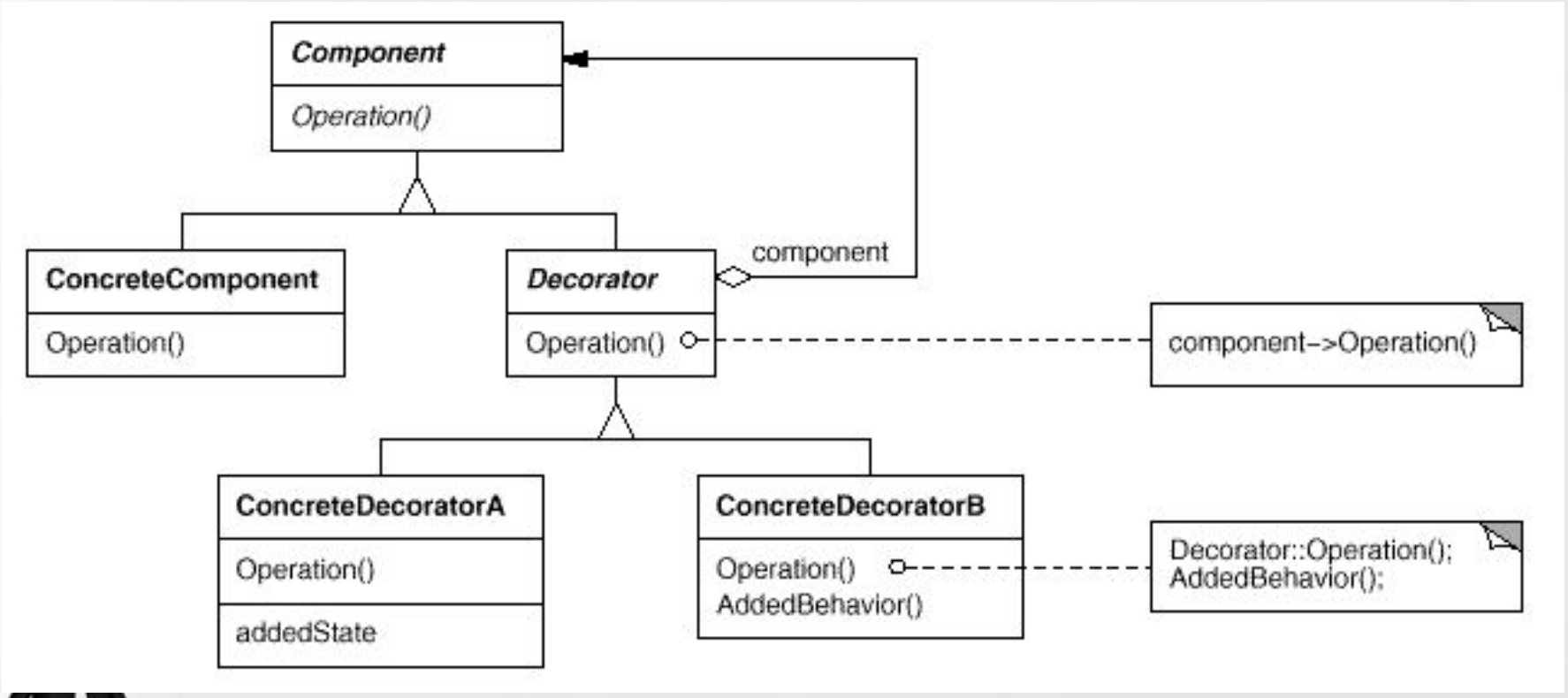
- Necessità di aggiungere responsabilità ad ogni singolo oggetto, non all'intera classe
- L'eriditarietà necessita una scelta durante la compilazione

- Soluzione
  - Racchiudere il componente in un altro oggetto che aggiunge le responsabilità
    - Racchiudere l'oggetto è detto Decorare

# Pattern Decorator

- Il Decorator si conforma all'interfaccia del componente che sta decorando dimodochè la sua presenza è trasparente ai componenti dei client
  - Il Decorator forwarda le richieste al componente e può fare azioni addizionali prima o dopo il forwarding
  - E' possibile includere decorator in maniera ricorsiva, permettendo cosi di aggiungere infinitamente responsabilità
  - E' possibile aggungere, rimouvore responsabilità dinamicamente

# Struttura


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

# Considerazioni

Vantaggi:
Pochi classi rispetto all'eriditarietà statica
Aggiunta dinamica
Le classi radici rimangono semplici

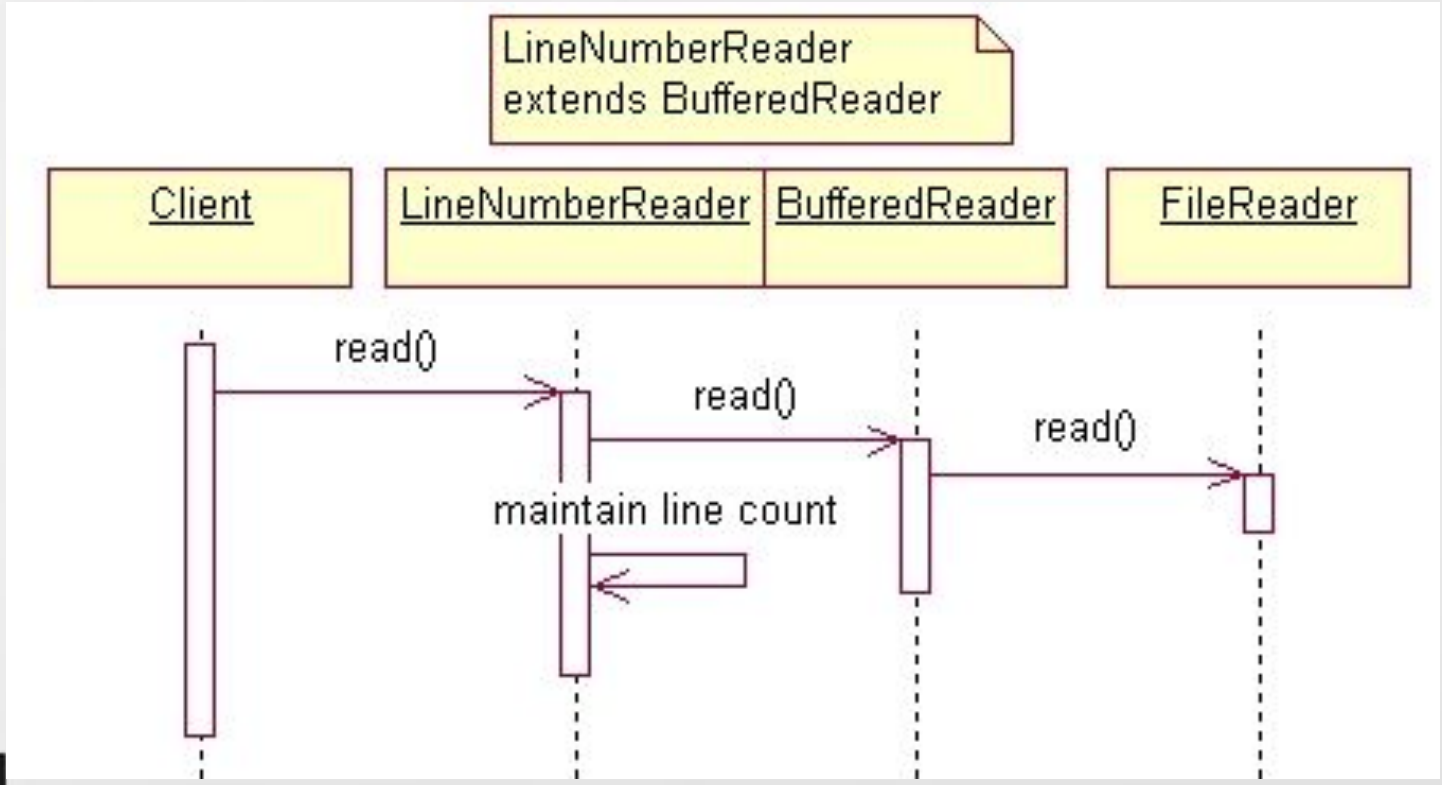
Svantaggi:
Proliferazioni di istanze al run-time

Usi:
Molto utile quando i componenti sono leggeri
altrimenti usare Strategy

# Esempio Java

```
FileReader frdr = new FileReader(filename);
LineNumberReader lrdr = new
  LineNumberReader(frdr);
String line;
while ((line = lrdr.readLine()) != null) {
    System.out.print(lrdr.getLineNumber() +
  ":\t" + line);
  }
```

# Diagramma di interazione




# Strutturali: Facade

Scopo:

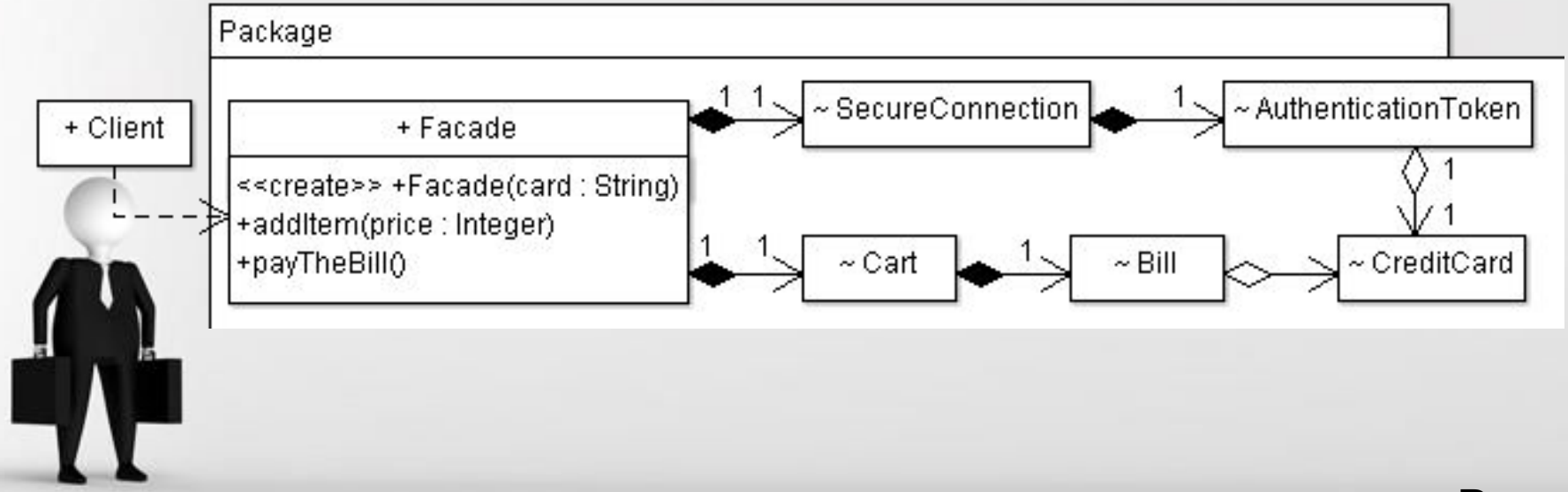
mostrare un punto di accesso semplice a un pacchetto dalle operazioni complesse;

nascondere complessi dettagli di collaborazione;

Ingredienti:

una classe che accede al contenuto del pacchetto;

operazioni semanticamente semplici orchestranti le classi presenti all'interno del pacchetto.



# Comportamentali: Command

Scopo:

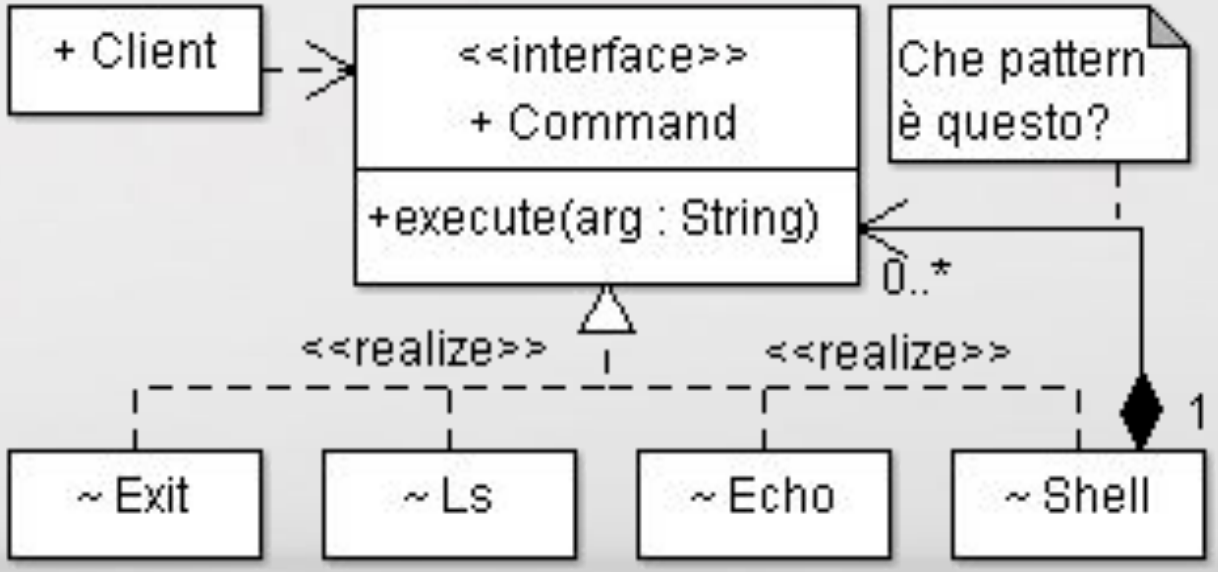
isolare una funzionalità dai suoi punti di accesso;

astrarre il concetto di operazione eseguibile;

Ingredienti:

classe astratta o interfaccia Command;

metodo astratto di esecuzione come contratto.



# Comportamentali: Iterator

Scopo:

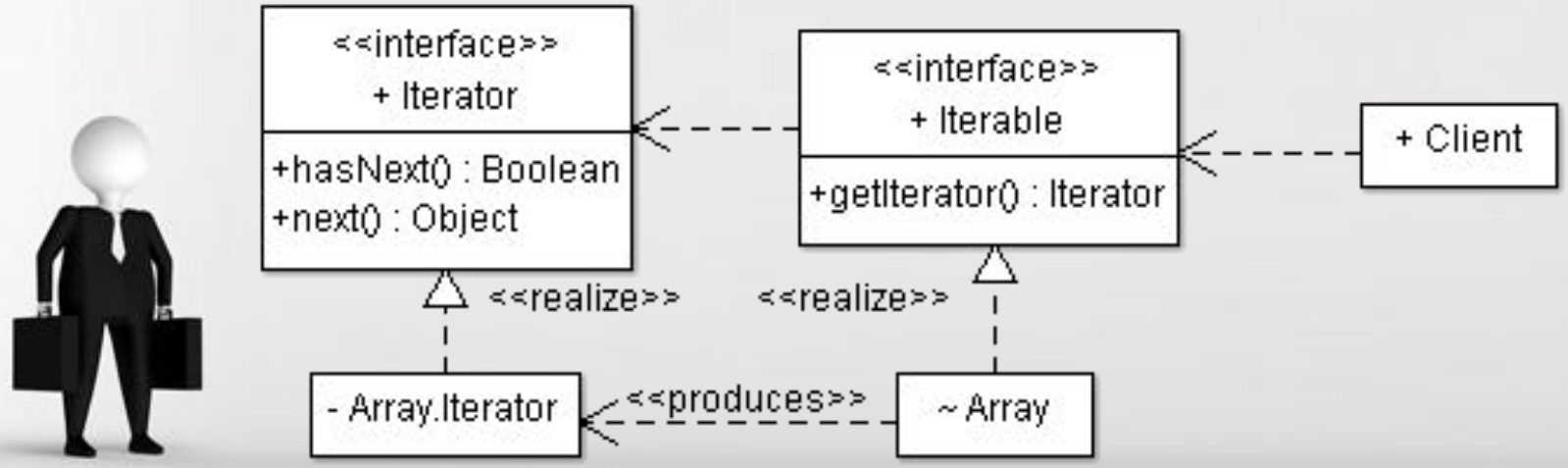
iterare sugli elementi di una struttura dati senza conoscere i dettagli della struttura stessa;

Ingredienti:

interfaccia di astrazione del concetto di iterazione;

astrazione di struttura dati che supporta l'iteratore;

strutture dati implementanti tale astrazione.



# Comportamentali: Observer

Scopo:

consentire l'astrazione della notifica di eventi;

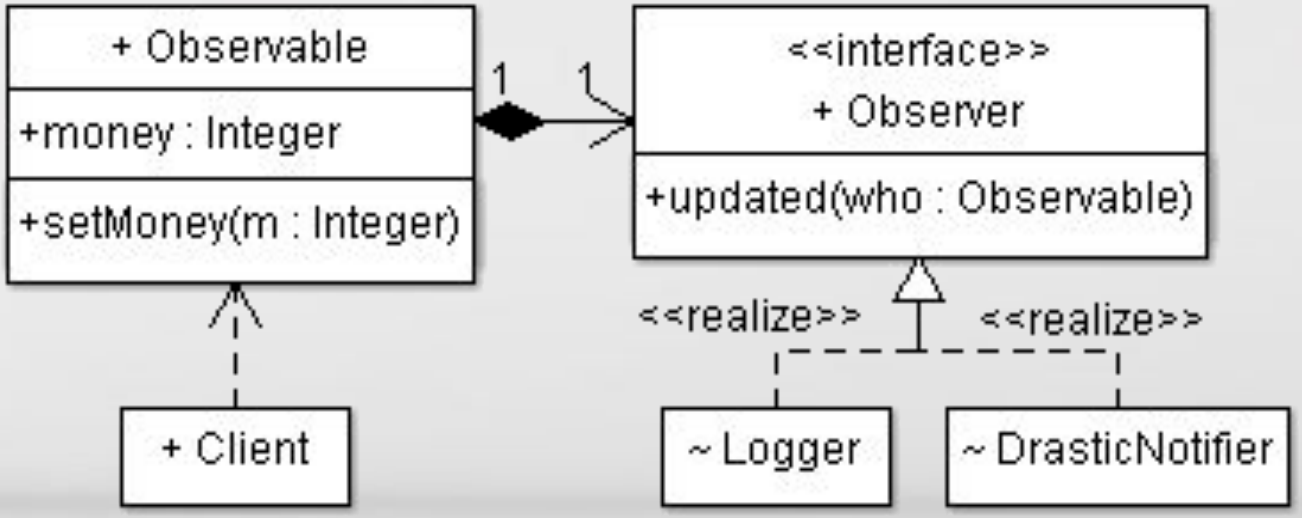
disaccoppiare chi genera l'evento da chi lo riceve;

Ingredienti:

interfaccia di notifica dell'evento (ricevente);

ascoltatori che la implementano;

classe di generazione degli aggiornamenti.



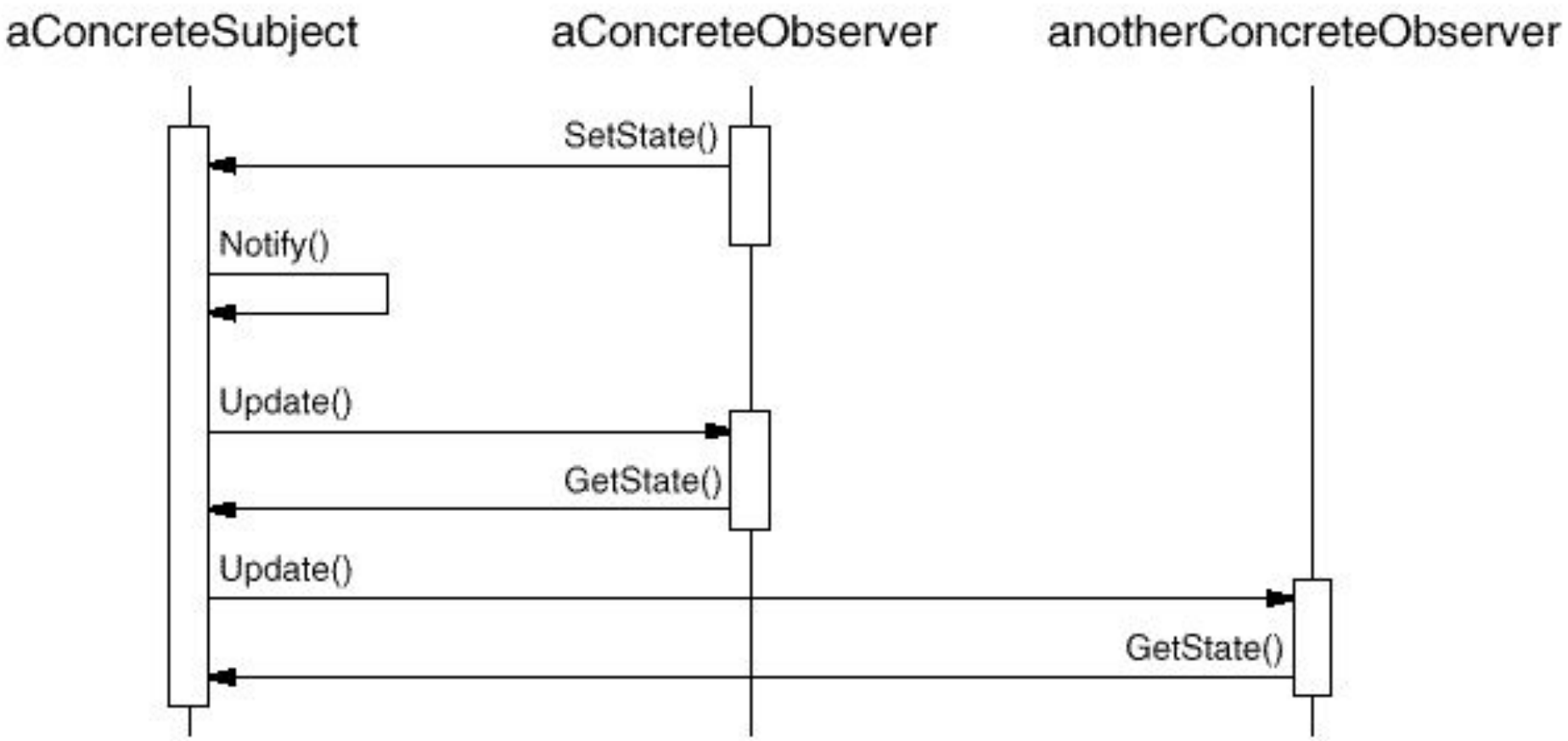
# Pattern Observer

- Problema
  - Lo stato dipendendente deve essere consistente con quello master

# Soluzione

- Definire quattro tipi di oggetti:
  - abstract subject
    - tenere una lista di dipendenti; notificare quando il master cambia
  - abstract observer
    - defiinire il protocollo di come aggiornare i dipedendenti
  - concrete subject
    - gestire i dati per i dipendenti; notificare quando il master cambia
  - concrete observers
    - prendere il nuovo stato del subject appena vi è una modifica

# Uso del Pattern Observer



# Conseguenze

- Disaccopiamento tra subject e observer
  - Subject è all'oscuro dei dipendenti
- Permettere il broadcasting:
  - Aggiunta e rimozione dinamica degli observer
- Aggiornamenti inaspettati:
  - nessun controllo da parte dei subject sulle computazioni degli observer

# Implementazione

- Memorizzare lista di Observer:
  - Generalmente nei Subject
- Osservare multi Subject
  - Aggiunta parametri ad update()

- Chi innesca l'update
  - Le operazioni di setting dello stato del subject
  - Client
    - Condizionato da errori se un observer dimentica di notificare un update

# Considerazioni

- Possibilità di riferimenti sospesi quando i subject sono cancellati
  - Poco male in linguaggi con grabge collector
  - Subject notificano della morte al Master
- Possibilità di notifica prematura
  - Tipicamente un metodo nella sottoclasse Subject chiama un metodo ereditato  il quale fa la notifica
  - Si usa il Template Pattern
    - Il metodo nella classe astratta chiama un metodo non riferenziato che è definito nella sottoclasse concreta

# Considerazioni

- Quanta informazione devono fornire i subject con le update() ?
  - Modello Push: I subject inviano tutte le informazioni che observer necessita
    - Possibilità di accopiamento subject, observer forzando un interfaccia di observer
  - Modello Pull: I subject non inviano informazioni
    - Può essere inefficiente

# Considerazioni

- Update complessi
  - Cambiare i gestori
    - tiene traccia di relazione complesse tra i subject e gli observer  e incapsula update complessi verso gli observer

# Considerazioni

- Observer su molti subject
  - Ha senso quando un observer dipende da subject. Il subject semplicimente si passa come  parametro durante un update dimodchè l'observer sa quale è il subject.
  - Bisogna che lo stato del subject sia consistente prima della notifica

# Esempi

Il modello ad eventi di standart Java e JavaBean è un esempio di Observer Pattern

# Comportamentali: Template Method

Scopo:

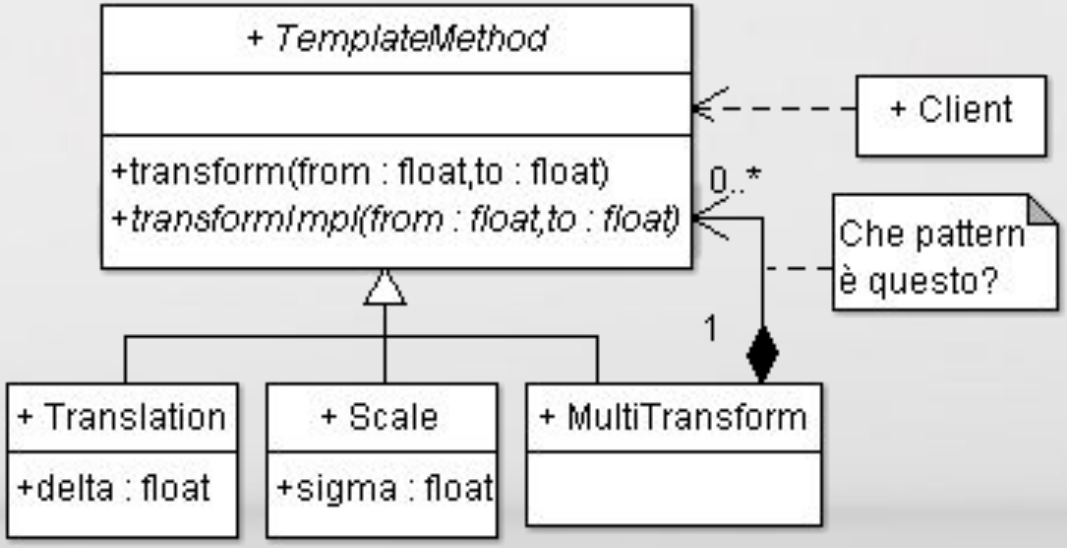
creare una gerarchia di classi in cui varia solamente una funzionalità, senza dover replicare codice;

astrarre una parte di una classe usata da sé stessa;

Ingredienti:

classe astratta con metodo astratto (protetto);

sottoclassi implementanti quel metodo.



# Comportamentali: Strategy

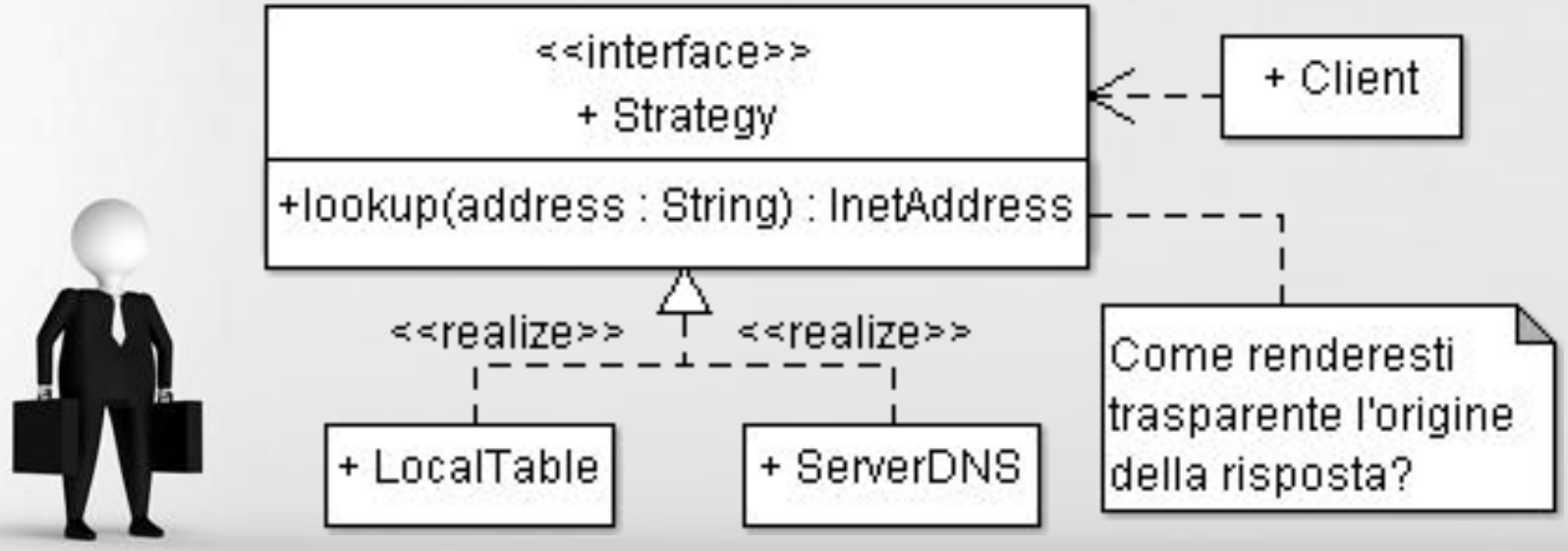
Scopo:

astrarre un algoritmo facendo sì la sua scelta avvenga a run-time (criteri vari, es.: efficienza);

Ingredienti:

interfaccia/classe astratta rappresentante l'algoritmo;

classi implementanti in concreto l'algoritmo stesso.



# Da qui?

- Abbiamo visto solamente un assaggio:
  - i design pattern GoF non sono solo questi;
  - molti altri design pattern esistono in letteratura;
  - costruire architetture usando i design pattern richiede esperienza oltre che profonda comprensione;
  - esistono altri strumenti come smell e antipattern;
  - tecniche di refactoring vengono utilizzate per trasformare un'architettura in un'altra migliore.