Design Pattern

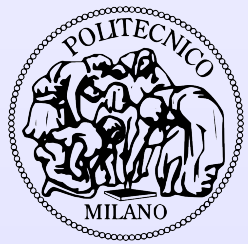
Design Principle

immagini da *SOLID Motivational Posters*, by Derick Bailey

Single Responsibility Principle

Single Responsibility Principle

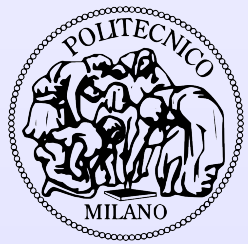
A class should have only one reason to change.



Open Close Principle

Open Close Principle

Le entità del software (le classi, i moduli e le funzioni) devono essere *aperte all'estensione* e *chiuse alle modifiche*

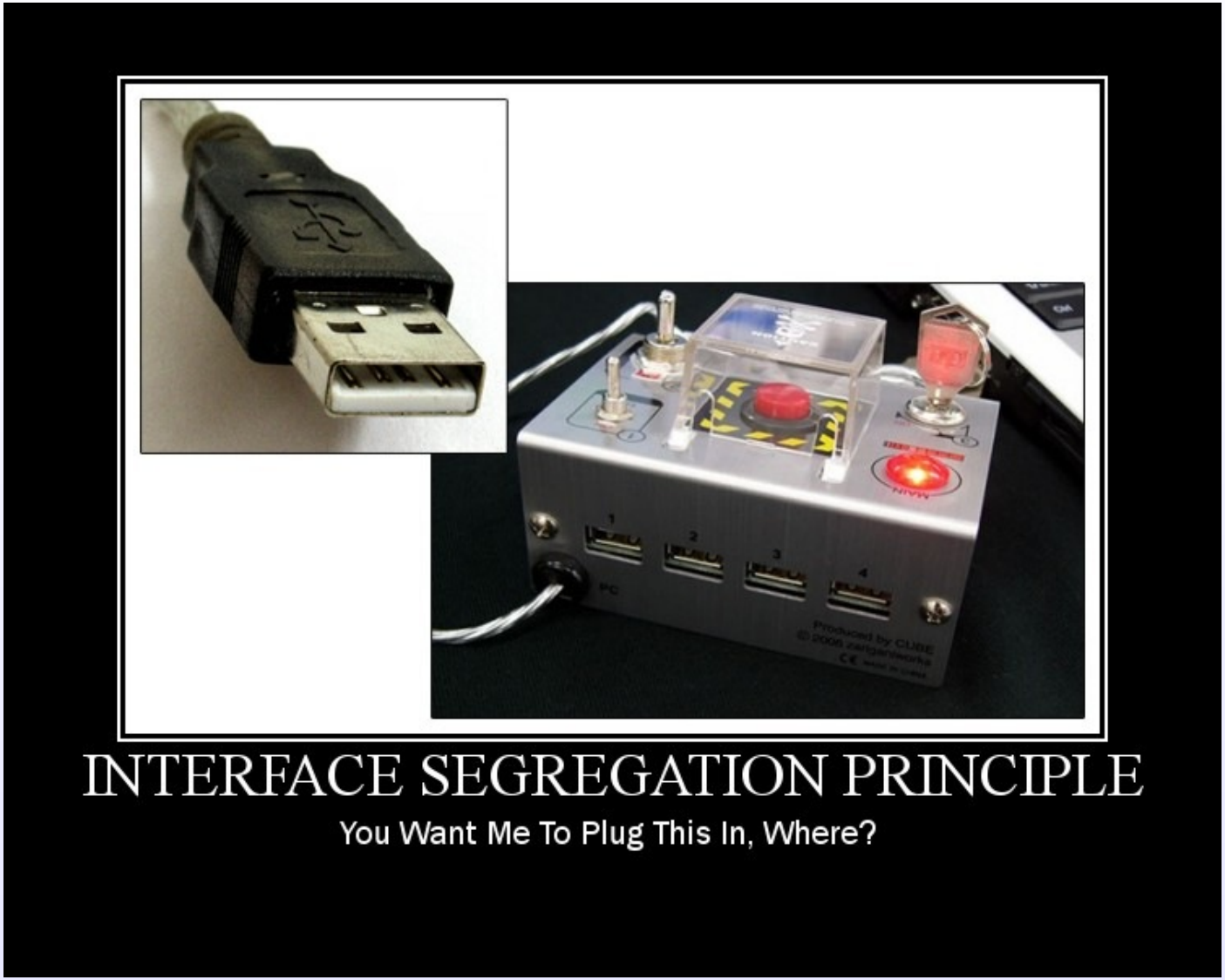


Liskov's Substitution Principle

Liskov's Substitution Principle

I tipi derivati devono essere
totalmente **sostituibili**
ai rispettivi tipi base.

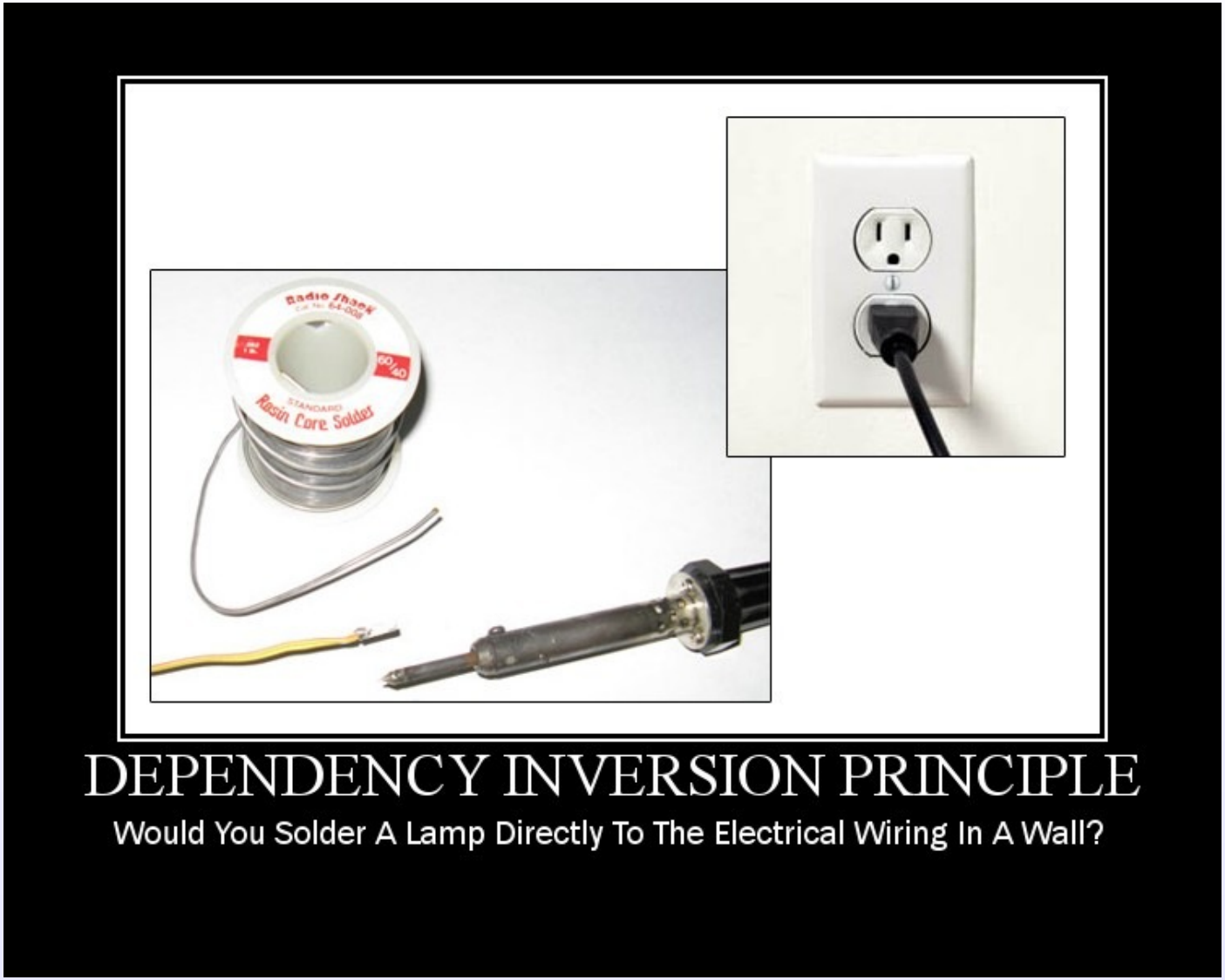
Interface Segregation Principle



Interface Segregation Principle

Nessun utilizzatore deve dipendere da metodi che non usa. Interfacce grandi devono essere spezzettate in interfacce piu' piccole e specifiche.

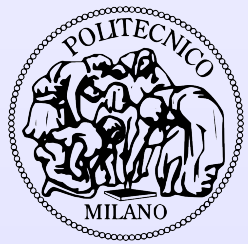
Dependency Inversion Principle



Dependency Inversion Principle

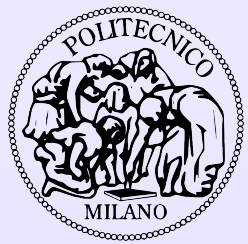
I moduli di alto livello non devono dipendere direttamente dai moduli di basso livello.
Entrambi devono dipendere da delle astrazioni.

Le astrazioni non devono dipendere i dettagli.
I dettagli devono dipendere dalle astrazioni.



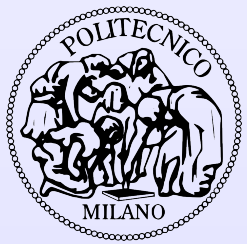
Design Pattern

- *Pattern creazionali*: riguardano il processo di creazione di oggetti
- *Pattern strutturali*: hanno a che fare con la composizione di classi ed oggetti
- *Pattern comportamentali*: Si occupano di come interagiscono gli oggetti e distribuiscono fra loro le responsabilita'

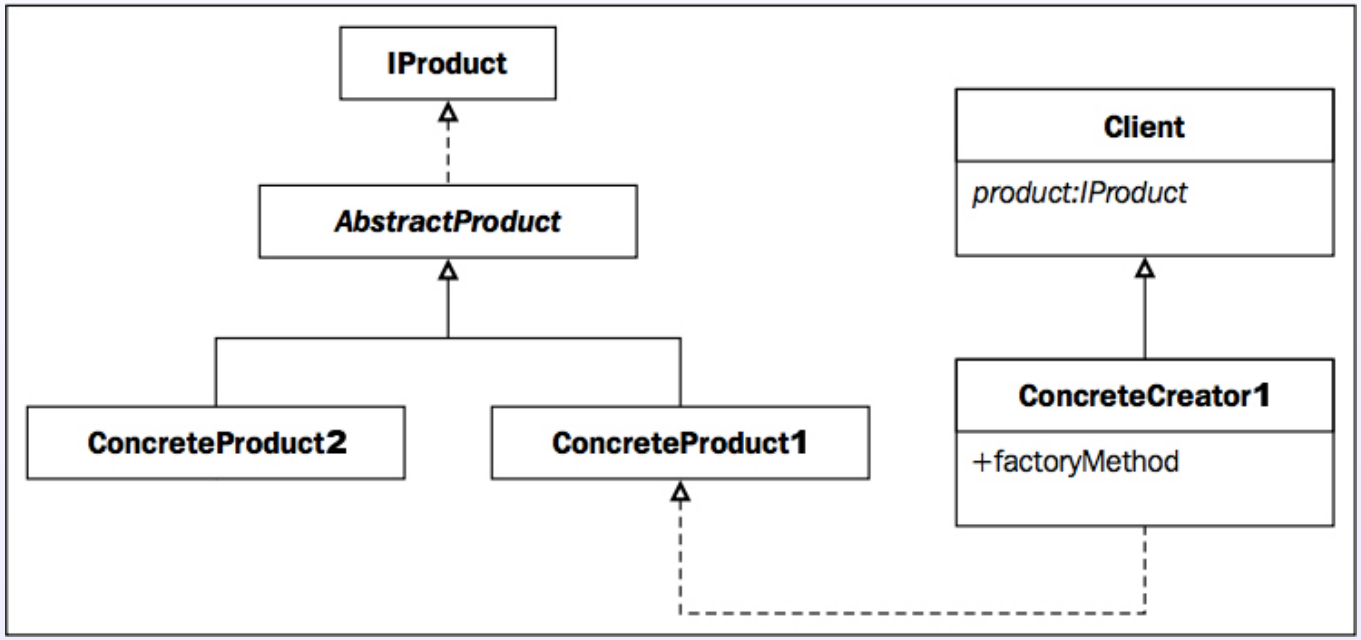


Pattern creazionali

- *Abstract Factory*: Crea una istanza tra diverse famiglie di classi
- *Factory Method*: Crea un'istanza tra diverse classi derivate
- *Singleton*: Una classe che puo' avere una singola istanza
- *Prototype*: Un'istanza totalmente inizializzata per essere copiata o clonata
- *Builder*: Separa la costruzione di un oggetto dalla sua rappresentazione



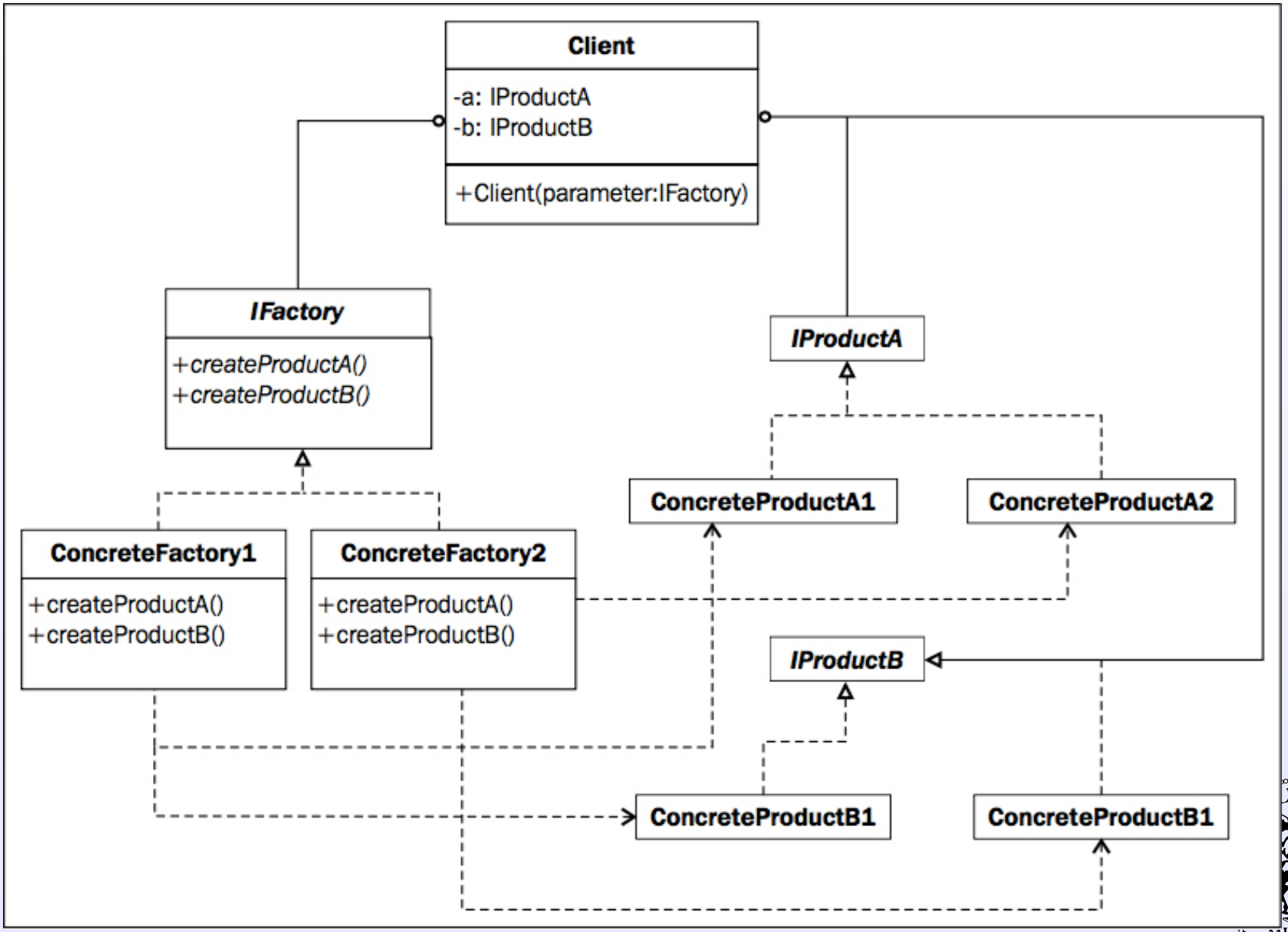
Factory method pattern



Immagini e tabella da

Factory Method e Abstract Factory: differenze e analogie, by Salvatore Cordiano

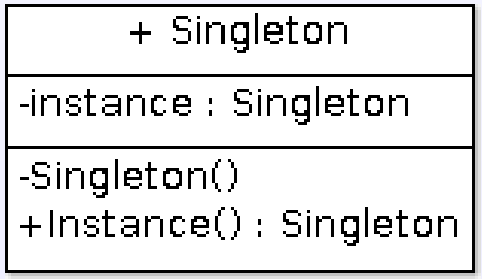
Abstract factory pattern



Factory method vs *Abstract factory*

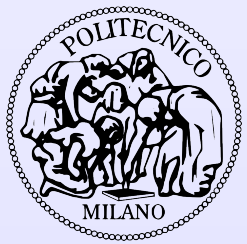
Factory Method	**Abstract Factory**
E' un metodo.	E' un'interfaccia per creare oggetti correlati. Abstract factory e' implementato spesso (non per forza) attraverso factory method.
Crea un solo tipo di oggetto.	Crea famiglie di oggetti.
Per creare nuovi oggetti dello stesso tipo e' sufficiente creare una sottoclasse del Client. Per creare nuove famiglie di oggetti e' necessario creare una sottoclasse di IFactory.	Per creare nuovi oggetti della stessa famiglia e' necessario ridefinire l'interfaccia di IFactory.

Singleton pattern



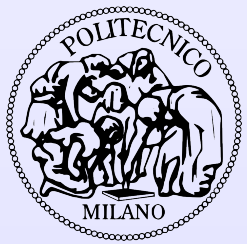
Esercizio 1

- Implementare un gioco contenente un labirinto.
- Il labirinto deve essere composto da locali, porte e muri.
- Ogni locale ha quattro pareti, nord sud est ovest che possono essere occupati da un muro o da una porta.

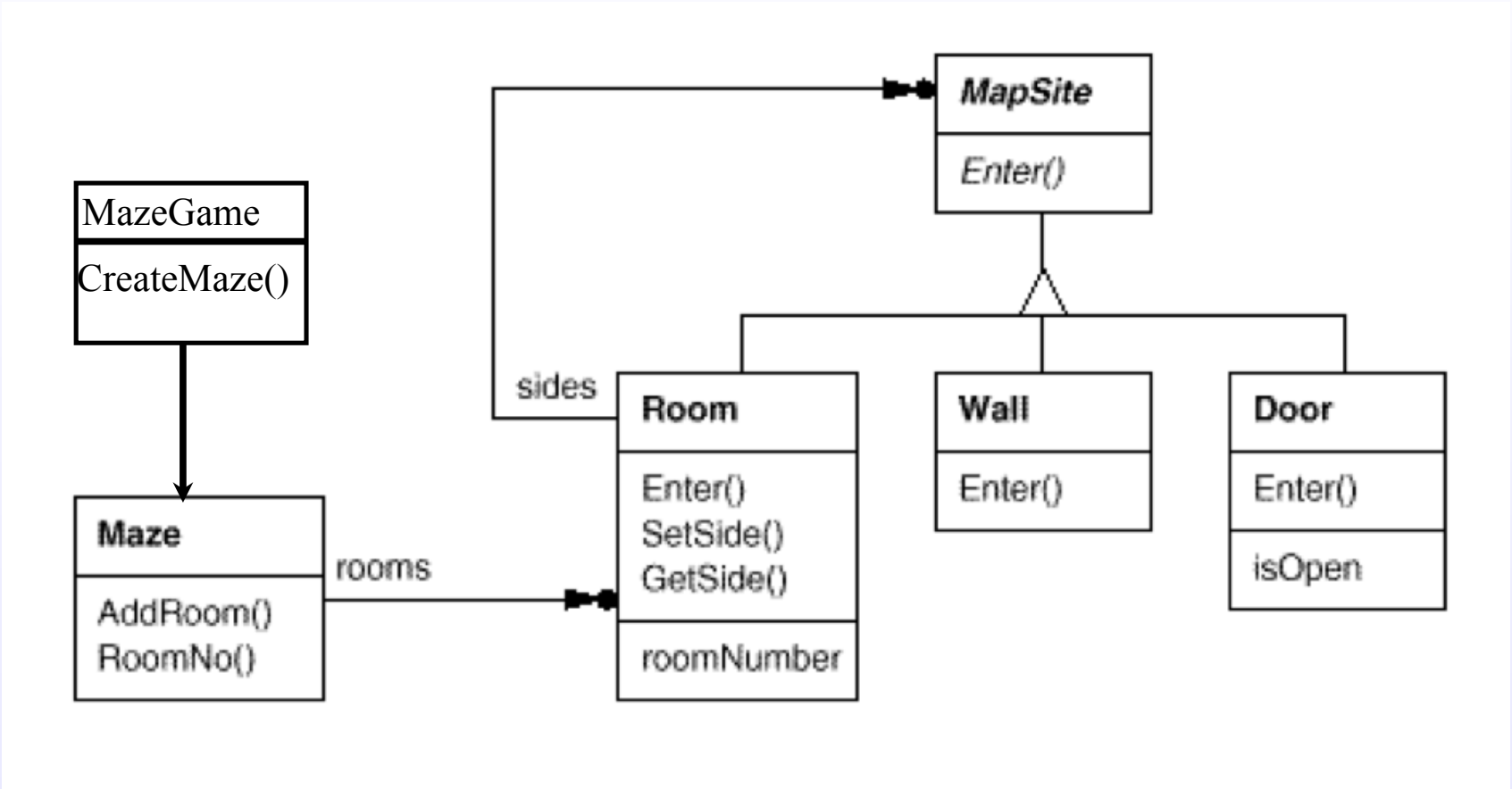


Esercizio 1

- Deve essere possibile creare labirinti di vario tipo.
- Deve essere possibile anche creare un tipo di labirinto bombato
- Per esempio un labirinto incantato deve contenere dei locali incantati, delle porte parlanti, etc. . .



Esercizio 1


MapSite
Enter()
MazeGame
CreateMaze()
sides
Room
Enter()
SetSide()
GetSide()
roomNumber
Wall
Enter()
Door
Enter()
isOpen
Maze
AddRoom()
RoomNo()
rooms

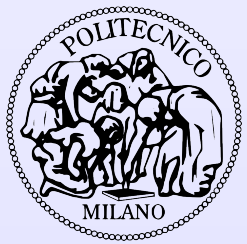

POLITECNICO
MILANO

Esercizio 1

- Implementiamo la funzione `MazeGame:CreateMaze()` per creare un labirinto con due stanze e una porta.
- Modifichiamo la funzione per creare un labirinto bombato, che contiene una porta con una bomba.

Esercizio 1 - labirinto normale

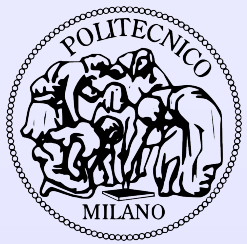
```
1  // in the class MazeGame
2  public Maze createMaze() {
3     Maze maze = new Maze();
4     Room room = new Room();
5     Room room2 = new Room();
6     Door door = new Door();
7     maze.addRoom(room);
8     maze.addRoom(room2);
9     maze.addDoor(door);
10    return maze;
11 }
```



Esercizio 1

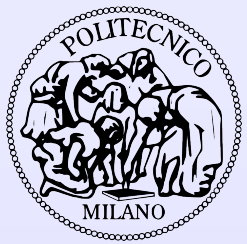
Cosa c'e' di sbagliato nella funzione precedente?

Possiamo usare questo metodo solo per creare stanze e porte in labirinti normali.
Come dobbiamo modificarlo per altri tipi di labirinti ?



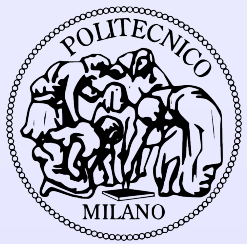
Esercizio 1 - labirinto bombato

```
1  // in the class MazeGame
2   public Maze createBombMaze() {
3      Maze maze = new BombMaze();
4      Room room = new RoomWithABomb();
5      Room room2 = new RoomWithABomb();
6      Door door = new Door();
7      maze.addRoom(room);
8      maze.addRoom(room2);
9      maze.addDoor(door);
10     return maze;
11 }
```



Esercizio 1- labirinto incantato

```
1  // in the class MazeGame
2  public Maze createEnchantedMaze() {
3     Maze maze = new Maze();
4     Room room = new EnchantedRoom();
5     Room room2 = new EnchantedRoom();
6     Door door = new DoorNeedingSpell();
7     maze.addRoom(room);
8     maze.addRoom(room2);
9     maze.addDoor(door);
10    return maze;
11 }
```



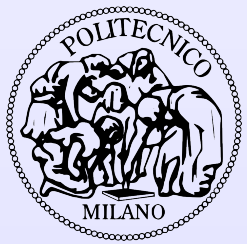
Esercizio 1 - possible problema

Possibile problema:
Il vostro gioco ha bisogno di creare delle stanze, ma non siete ancora sicuri di come queste saranno implementate e potrebbero essere estese in futuro.

Soluzione 1:

```
1 // TODO: cambiare la prossima linea quando sappiamo cosa e' una stanza, e
      come implementarla
2 Room r = new TempRoom();
3 // Nota: TempRoom e' una sottoclasse di Room.
```

Problemi??

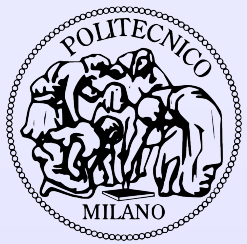


Esercizio 1 - Abstract Factory

Soluzione 2:

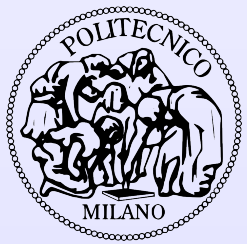
```
1 // myRoomFactory is an abstract factory!
2 Room r = myRoomFactory.createRoom();
```

Vantaggio: Settiamo myRoomFactory solo una volta, dopo di che l'istanza corretta della stanza verra' restituita!



Esercizio 1

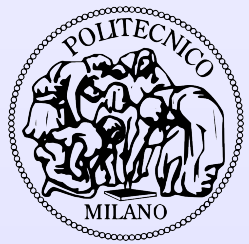
```
1   // in the class MazeGame
2   public Maze createMaze(MazeFactory factory) {
3  Maze maze = factory.createMaze();
4  Room room = factory.createRoom();
5  Room room2 = factory.createRoom();
6  Door door = factory.createDoor();
7  maze.addRoom(room);
8  maze.addRoom(room2);
9  maze.addDoor(door);
10    return maze;
11 }
```



Esercizio 1 - Abstract Factory

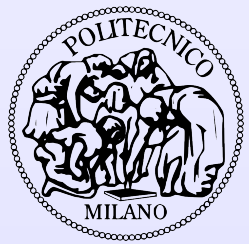
Ora possiamo usare lo stesso metodo `createMaze` in tutte tre le situazioni, bastera' passare il corretto `MazeFactory` ogni volta.

In questo esercizio MazeFactory e' una classe concreta. `EnchantedMazeFactory` **e** `BombedMazeFactory` **fanno l'override dei metodi che servono per restituire le istanze corrette.**



Esercizio 2

- Non deve essere possibile creare istanze differenti del `EnchantedMazeFactory` e `BombedMazeFactory`.



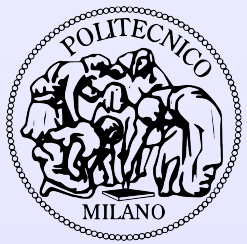
Esercizio 2 - Soluzione

Soluzione: Singleton pattern!

Per ottenere vari labirinti verra' chiamato diverse volte il metodo `createMaze` che ritornera' istanze diverse della classe `Labirinto`, ma senza avere istanze diversi dei due creatori.

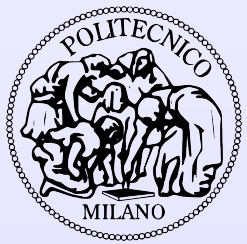
Pattern Strutturali

- *Adapter*: Adatta le interfacce di classi diverse
- *Bridge*: Separa l'interfaccia di un oggetto dalla sua rappresentazione
- *Composite*: Una struttura ad albero di oggetti semplici e composti
- *Decorator*: Aggiunge dinamicamente comportamenti ad oggetti
- *Facade*: Un'unica classe che rappresenta un intero sottosistema
- *Flyweight*: Condivide dati uguali tra oggetti dello stesso tipo
- *Proxy*: Un oggetto che rappresenta un altro oggetto



Adapter vs *Facade*

Adapter	**Facade**
Supporta un comportamento polimorfico .	Semplifica il sottosistema.

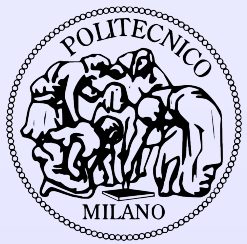


Pattern Comportamentali

- *Chain of Responsibility*: Un modo di passare una richiesta in una catena di oggetti
- *Command*: Incapsula un comando di richiesta come un oggetto
- *Interpreter*: Un modo di includere elementi del linguaggio in un programma
- *Iterator*: Accede in sequenza agli elementi di una collezione
- *Mediator*: Definisce una semplice interfaccia di comunicazione tra classi
- *Memento*: Salva e ripristina lo stato interno di un oggetto
- *Observer*: Un modo per notificare un cambiamento a un numero di classi
- *State*: Cambia il comportamento di un oggetto in base al cambiamento di stato
- *Strategy*: Incapsula un algoritmo all'interno di una classe
- *Template Method*: Delega l'implementazione di alcuni passi di un algoritmo a una sottoclasse
- *Visitor*: Aggiunge una nuova operazione ad una classe senza cambiarla

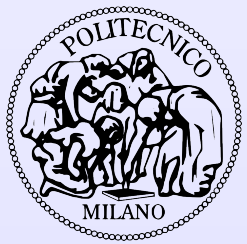
Strategy vs *State*

Strategy	**State**
Ogni classe (algoritmo) rappresenta degli algoritmi	Ogni classe (stato) rappresenta lo stato del sistema.
Ogni classe (algoritmo) gestisce solo task specifici	Ogni classe (stato) implementa (quasi) tutti i metodi per il contesto specificato .
Comportamento (tipo di algoritmo da selezionare) definito dal chiamante.	Comportamento autodefinito(definito dallo stato del sistema).

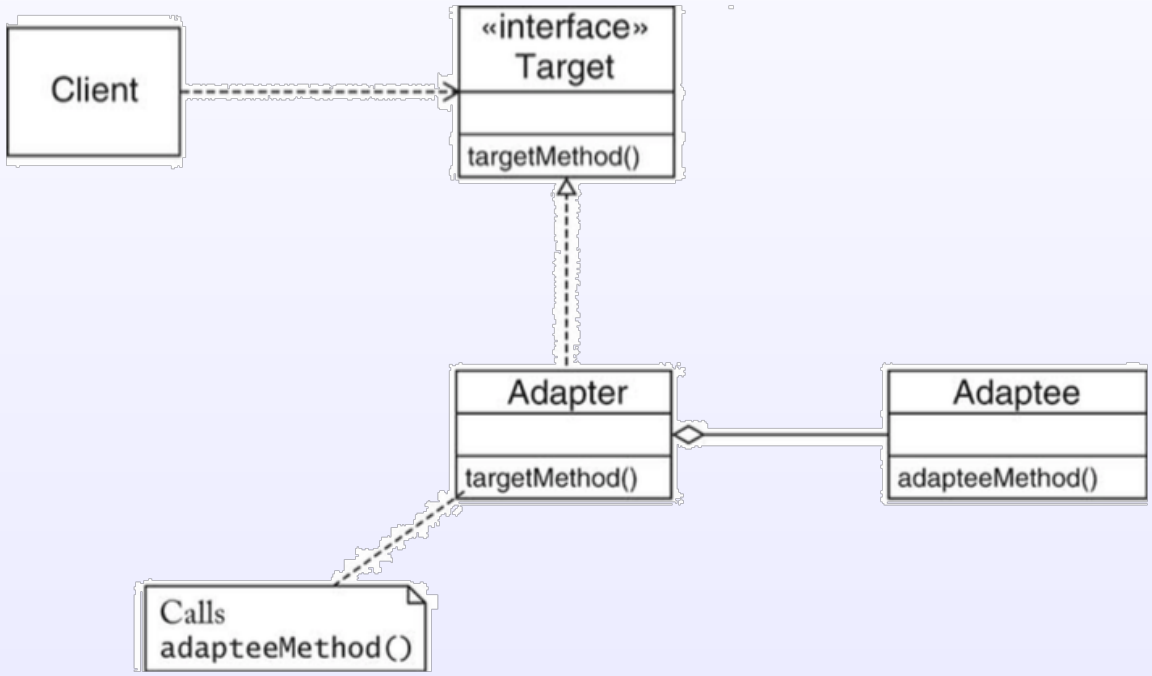


Esercizio 3

- Vogliamo sviluppare una applicazione che permetta l'utilizzo di diverse tecnologie di comunicazione, evitando il più possibile duplicazioni di codice
- come possiamo strutturare la nostra applicazione?
- supponiamo per esempio di avere la nostra logica che ci consente di eseguire un set di operazione `action(A)`, `action(B)` e `action(C)`; come possiamo supportare un client che utilizza dei comandi specificati per mezzo di Stringhe o vari tipi di comunicazione?



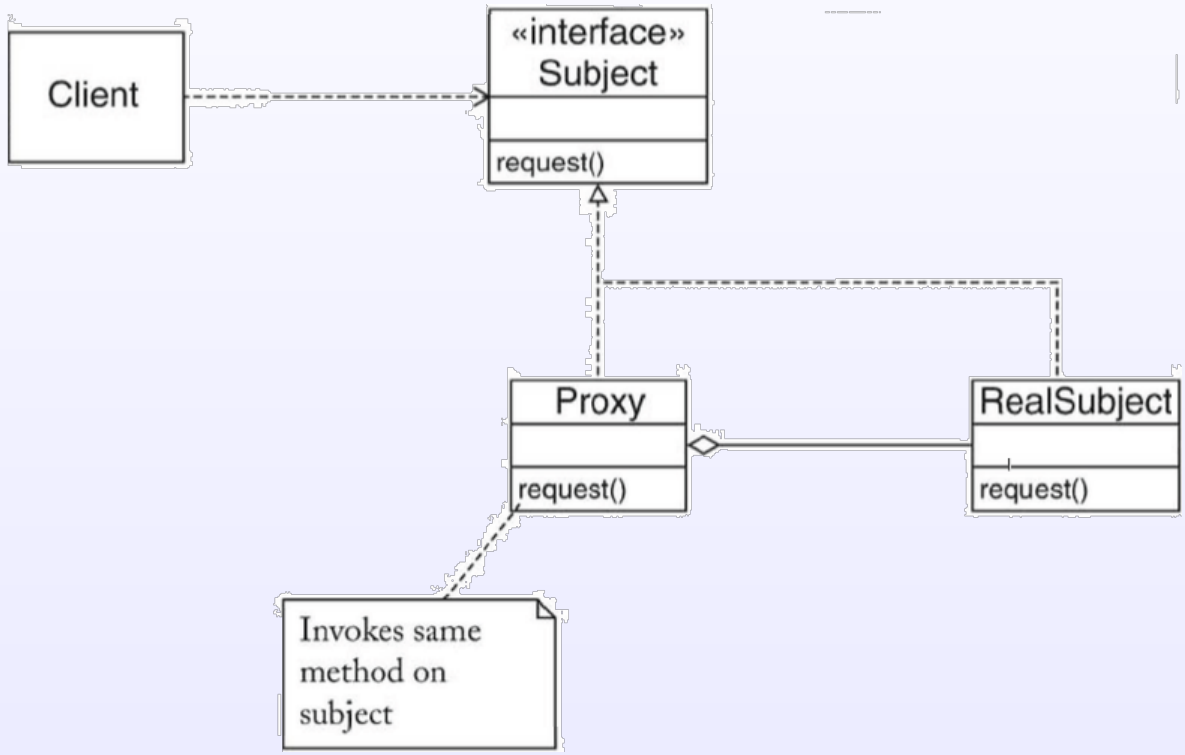
Adapter pattern

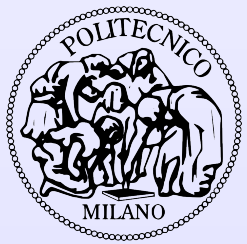


Esercizio 4

- Immaginiamo di implementare un social network.
- Data una specifica persona vorremmo mostrare come “potenziali amici" gli amici degli amici degli amici.
- Come possiamo implementarla in maniera efficace?

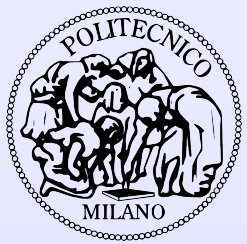
Proxy pattern



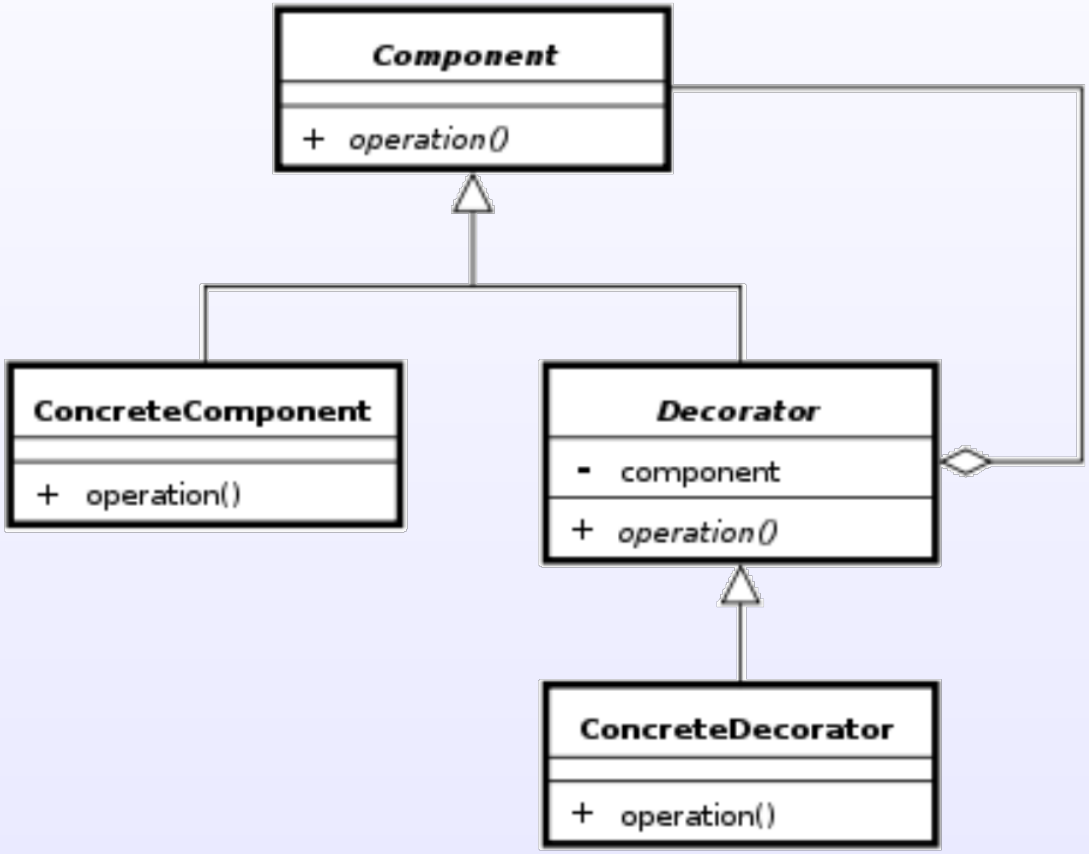


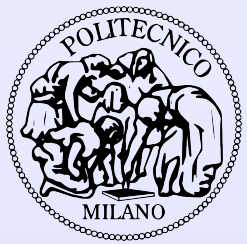
Esercizio 5

- Scrivere un applicazione per rappresentare diversi tipi di caffe' con diversi ingredienti.
- Per esempio, vorremmo aggiungere del caramello, del whisky, dello zucchero, a richiesta.



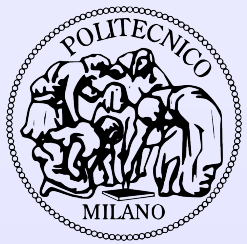
Decorator pattern



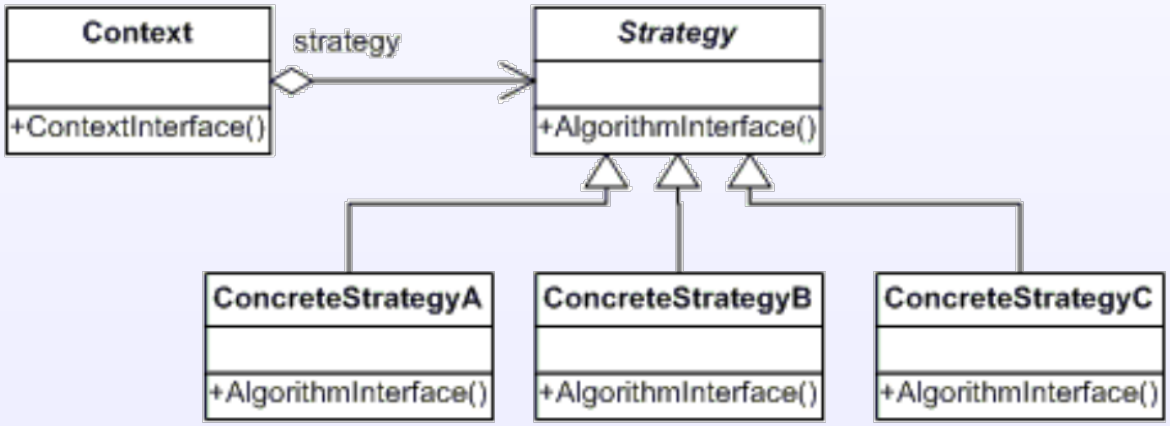


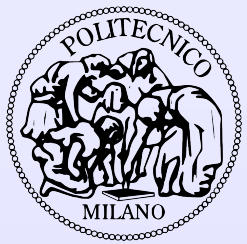
Esercizio 6

- Scrivere il codice per rappresentare un robot che puo' avere diverse strategie per gestire il suo comportamento.
- Vogliamo fare si che i comportamenti possano cambiare mentre la nostra applicazione e' in esecuzione.



Strategy pattern

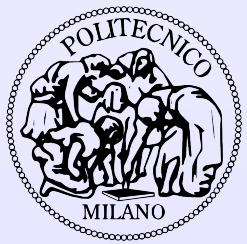




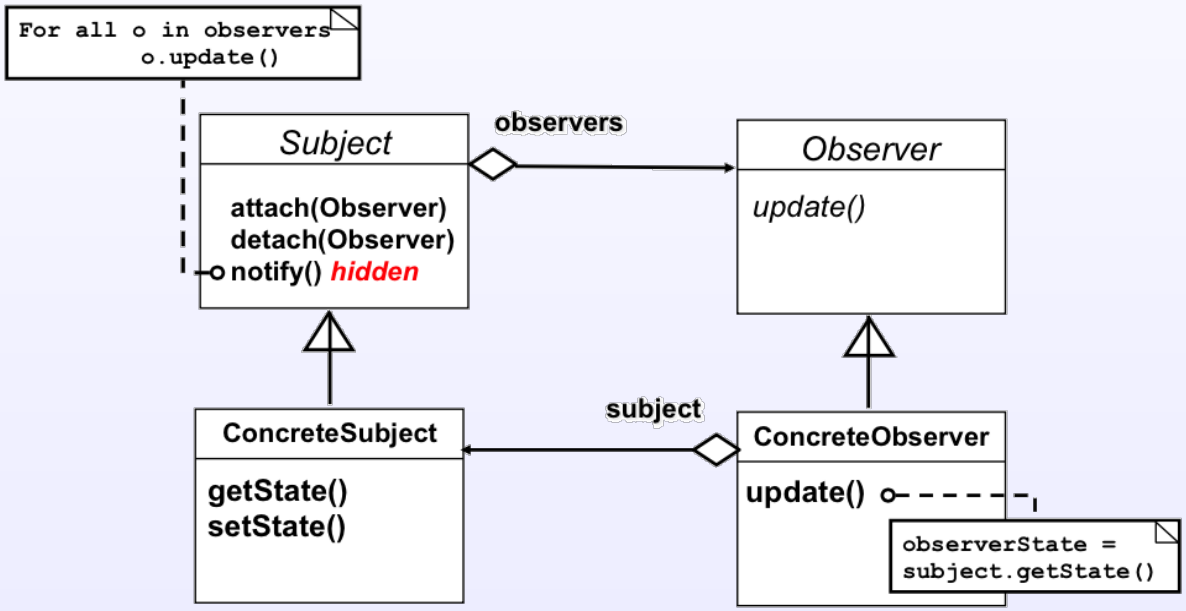
Esercizio 7

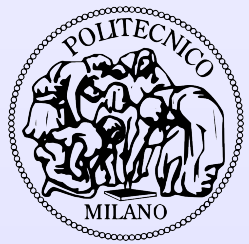
Progettare un sistema per il monitoraggio del Meteo:

- si ha a disposizione l'oggetto `WeatherData` che fornisce temperatura, umidita', pressione
- implementare tre diversi tipi di display che mostrano condizione attuale, previsioni e statistiche
- il sistema deve essere espandibile per supportare nuovi display



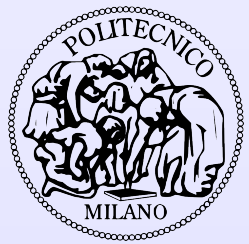
Observer pattern





Ack

Slide originarie di Alessandro Rizzi, modificate da Mattia Salnitri.
Slide su Abstract Factory parttern adattate e tradotte dalle slide di Thomas Fritz e Martin Glinz[3]



[3] https://www.ifi.uzh.ch/dam/jcr:087cc35a-627e-4032-aa50-43c8d9edc741/DesignPatterns.pdf