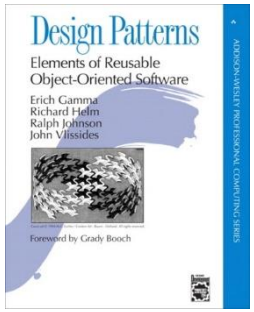
Software Design Patterns (parte 1)

Ingegneria del Software a.a. 2022-2023

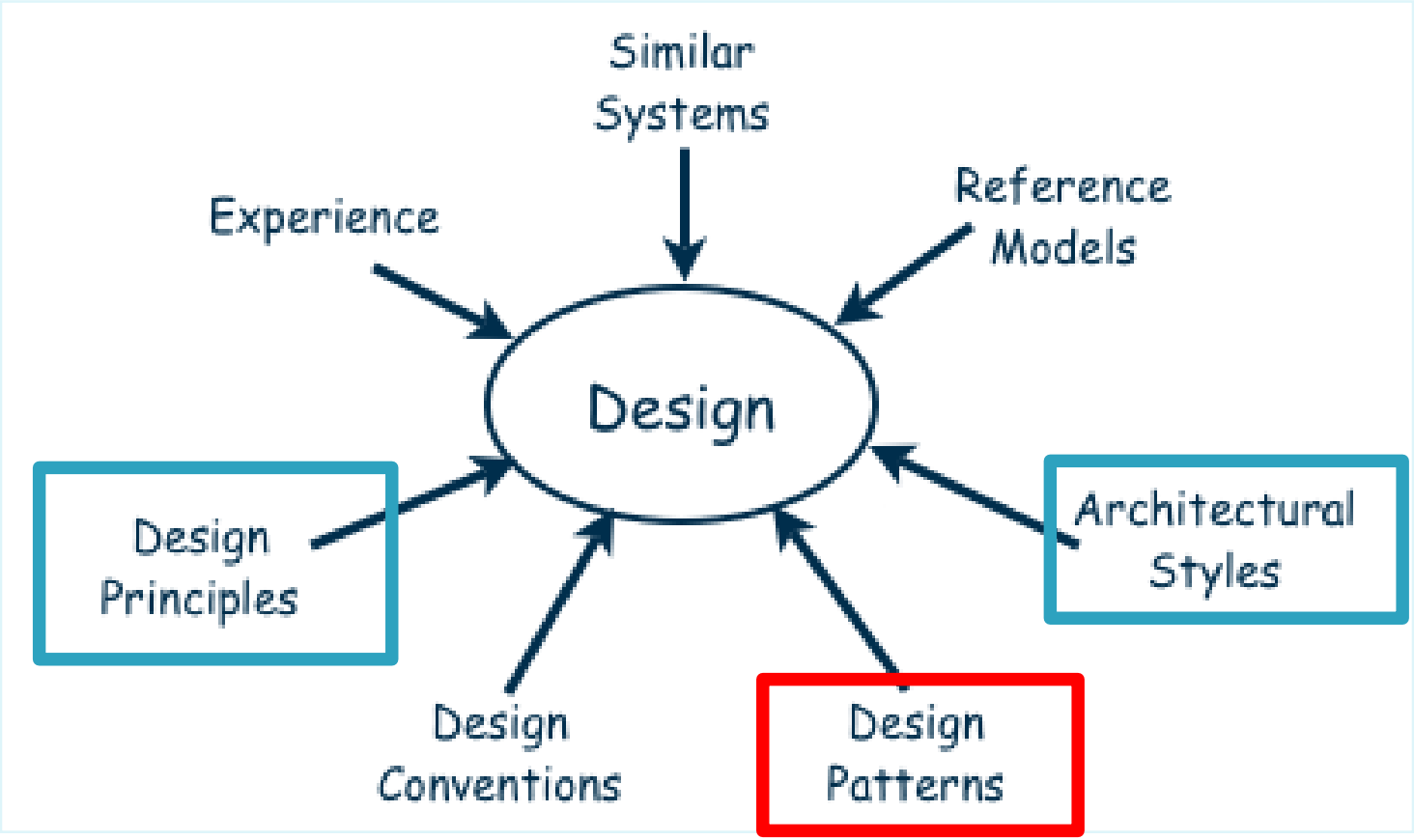
Agenda

- Cos'è un **Design Pattern**?
  - Differenza con Framework e Libreria
- Pattern elementari **GRASP**
  - General Responsability Assignment Software Patterns
- Elementi definiti nei design pattern:
  - Nome, Problema, Soluzione, Conseguenze
- **Catalogo** di design patterns **GoF**:
  - Creational Patterns, Structural Patterns, Behavioral Patterns
- GoF patterns che vedremo nel corso (**7/23**):
  - Factory Method, Abstract Factory, Adapter, Facade, Template Method, Observer, State



coursera

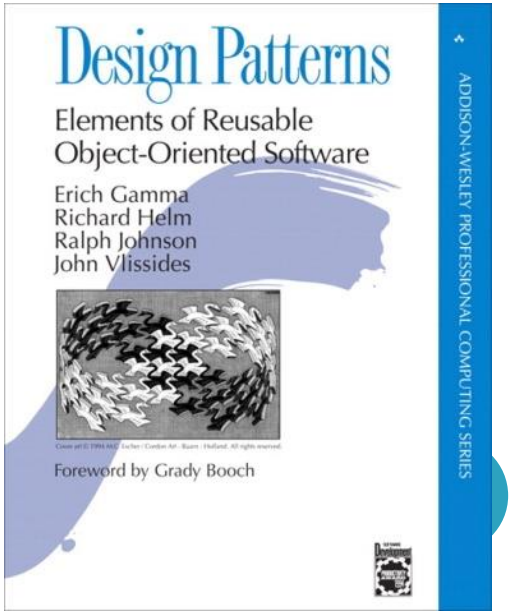
Design: Un Processo Creativo?



Design pattern

Coppia problema/soluzione che costituisce unità di riuso

- Una **soluzione elegante** (con un **nome**) di uno **specifico problema** di design/programmazione OO che costituisce un **unità di riuso**
  - Soluzioni ampiamente “levigate” ed usate (esperienza)
  - Non le soluzioni che vengono in mente per prime
  - Non basate su un particolare linguaggio OO
- L’idea di design pattern deriva da **Kent Beck** (famoso anche per Extreme Programming)
- Formalizzati nel libro autorevole e di larga diffusione GoF (Gang Of Four), 1995
- Attualmente molti cataloghi di pattern, in genere relativi ad:
  - un particolare **dominio applicativo**
    - Es. Game programming patterns
  - una particolare **tipologia di applicazioni**
    - Es. Web applications, Mobile and Enterprise



Design pattern: a cosa servono?

- Aiutano ad applicare i principi di **buona** progettazione OO
  - Es. high cohesion - low coupling
- Aiutano a creare “buoni” design
  - Possono essere applicati sia durante la **modellazione** che la **codifica**
- Facilitano la comunicazione tra sviluppatori

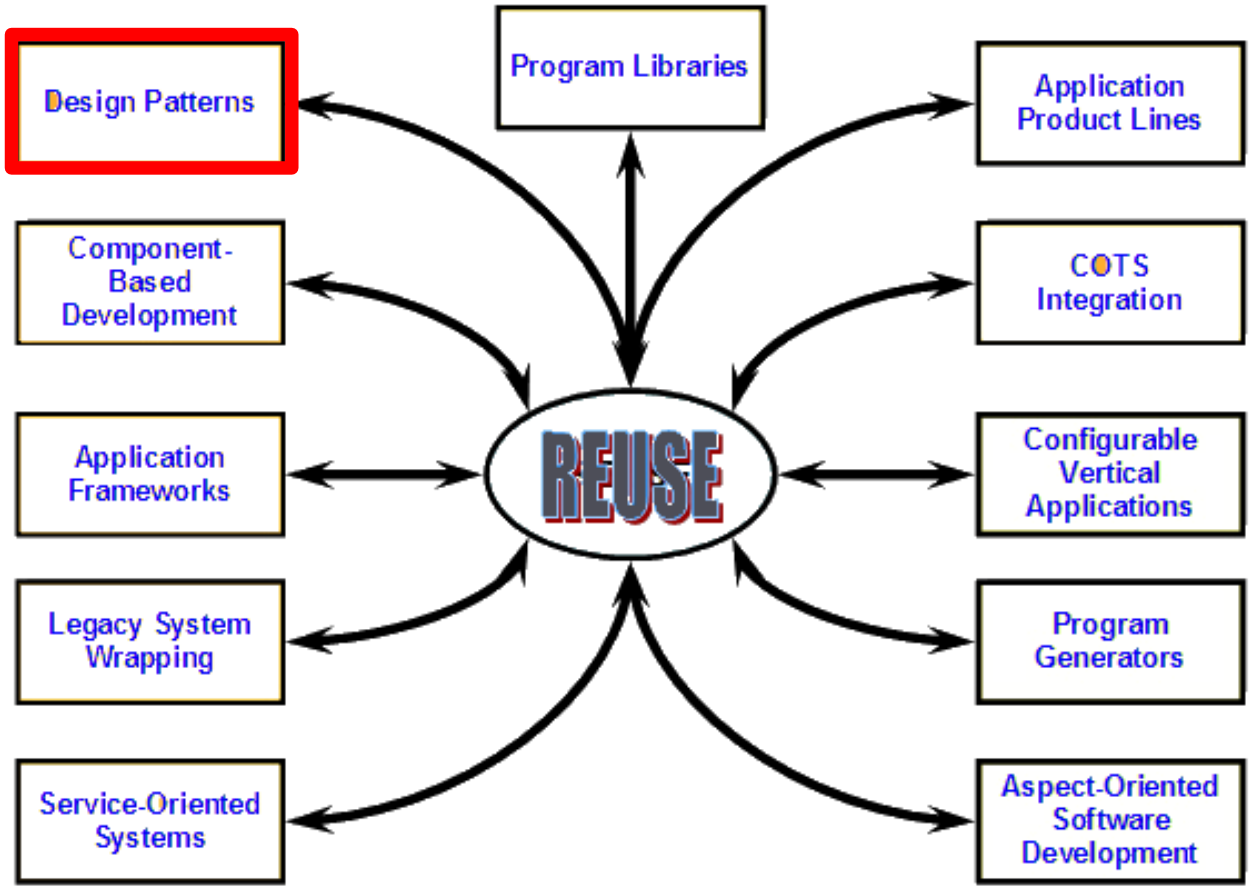
Jack il sottosistema della persistenza deve esporre i suoi servizi come una ***Facade***. Useremo un ***Abstract Factory*** per i ***Data Mapper***, e i ***Proxy*** per la ***materializzazione pigra (*Lazy initialization*)*** ...

Ehh? Cosa accidenti hai detto? Parla come mangi!

Altre unità di riuso: il panorama

Non l'unica!

[da Sommerville]



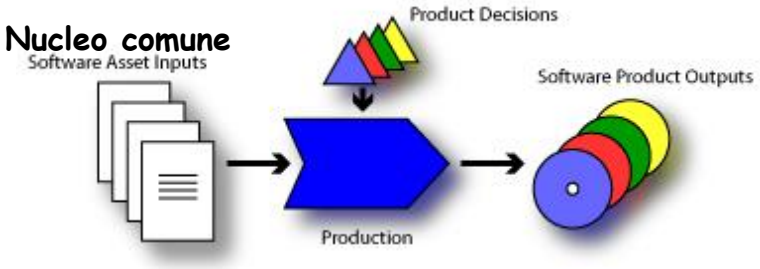
Perché il riuso?

È una best practice …

Altre unità di riuso

- **Librerie**
- **Componenti riusabili (COTS)**
  - Parti di applicazione riusabili con interfaccia ed implementazione (non visibile) [vedere TAP]
- **Framework**
  - Applicazione "con buchi" (assumono il controllo dell'applicazione)
    - Spesso per capire come funziona un framework bisogna conoscere i Design pattern ...
- **Product line**
  - **Famiglie di prodotti** sviluppate fin dall'inizio in contemporanea, con nucleo comune (core asset base)
    - Es. Visual Paradigm
- **Modelli**
  - es. del dominio, dei requisiti, del design

COTS = (Commercial) Off-the-Shelf component



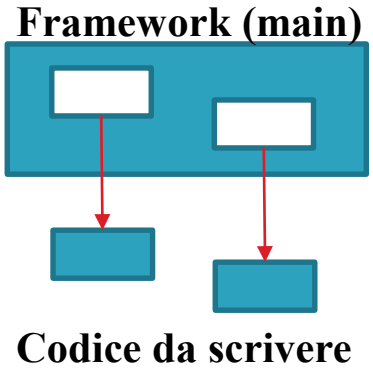
Esempio (di probabile) product line: Visual Paradigm

"Reuse the main body and write code it calls"

FRAMEWORK

- Un framework è un insieme di classi e interfacce cooperanti che realizzano un **design riusabile** e **customizzabile** per uno specifico dominio applicativo o tipologia di app
  (GUI-based app, web applications, accounting systems, …)

- Dettano l'architettura:
  - **Inversione del flusso di controllo**
- Customizzati creando sottoclassi specifiche dell'applicazione



- **Esempi di framework nel mondo Java:**
  - Struts / Struts 2 – per Web app
  - Spring – per JAVA app (anche Web)
  - Hibernate – per gestire oggetti persistenti

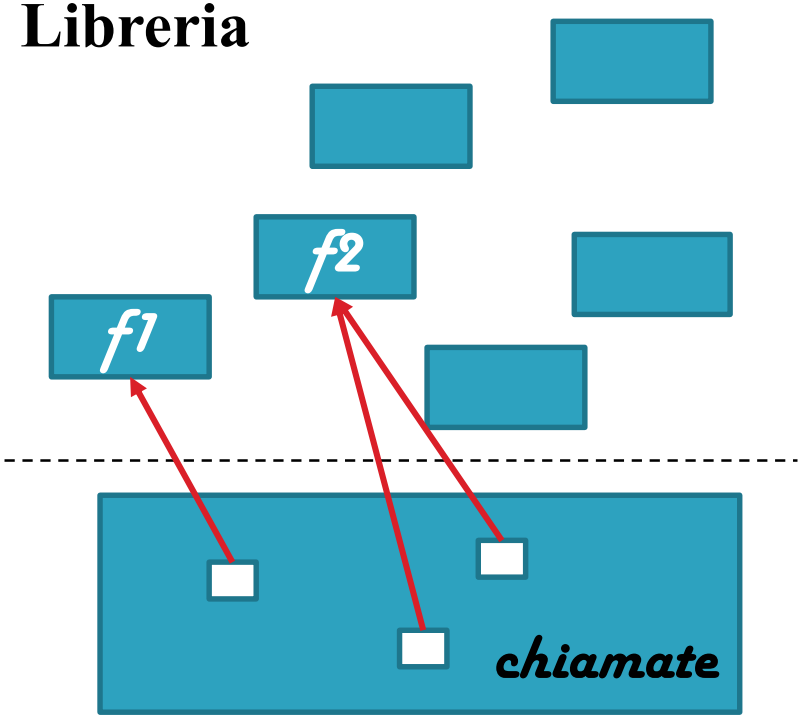
Struts²

spring

HIBERNATE

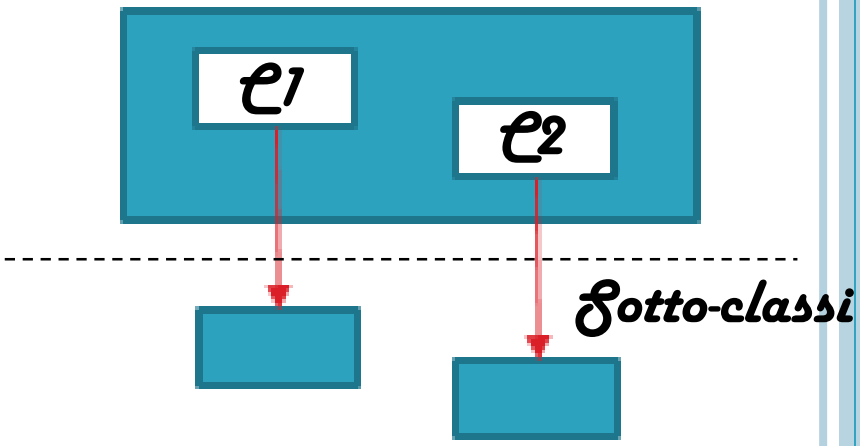
Libreria vs. Framework

Libreria



Codice da scrivere (main)

Framework (main)



Codice da scrivere

Framework vs. 'XYZ'

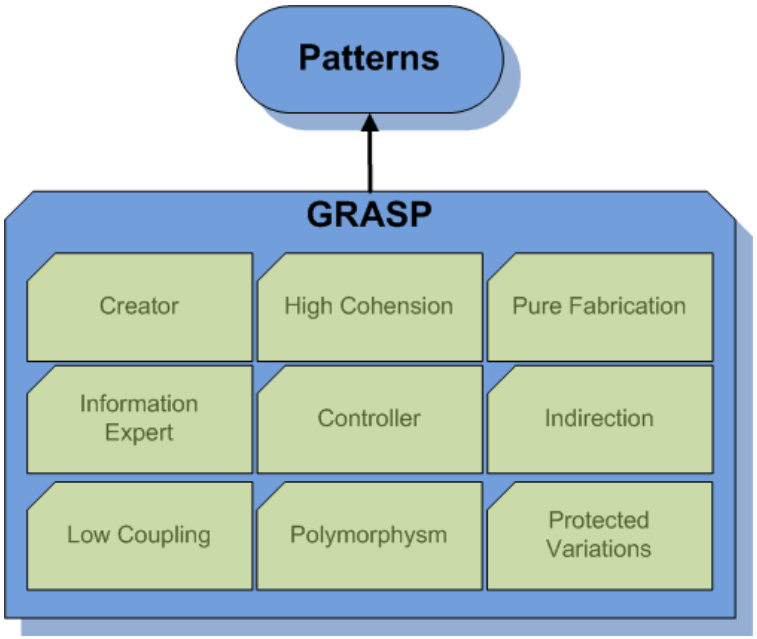
- **Framework vs. libreria software:**
  - **Libreria:** lo sviluppatore di un'applicazione chiama le operazioni delle classi di un libreria
  - **Framework:** lo sviluppatore di un'applicazione scrive le classi (in particolare le operazioni) che vengono chiamate (usate) dal framework
    - "lo sviluppatore riempie i buchi"

- **Design pattern vs. framework**
  - più astratti
    - vanno implementati/adattati non si possono usare "as-is"
  - più piccoli (come elementi architetturali)
  - di solito meno specializzati (vale per pattern GoF), **non relativi a dominio applicativo specifico o tipologia di app**
    - Es. Design pattern 'Controller' si può usare sia in Mobile che Web app

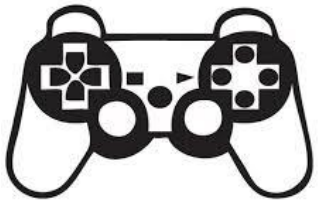
DESIGN PATTERN ELEMENTARI: GRASP

- I pattern Grasp sono pattern per **«l'assegnazione di responsabilità»** nel software
- Costituiscono **il fondamento** per la progettazione di sistemi OO
  - Chi sa sviluppare OO li applica senza conoscerli ....

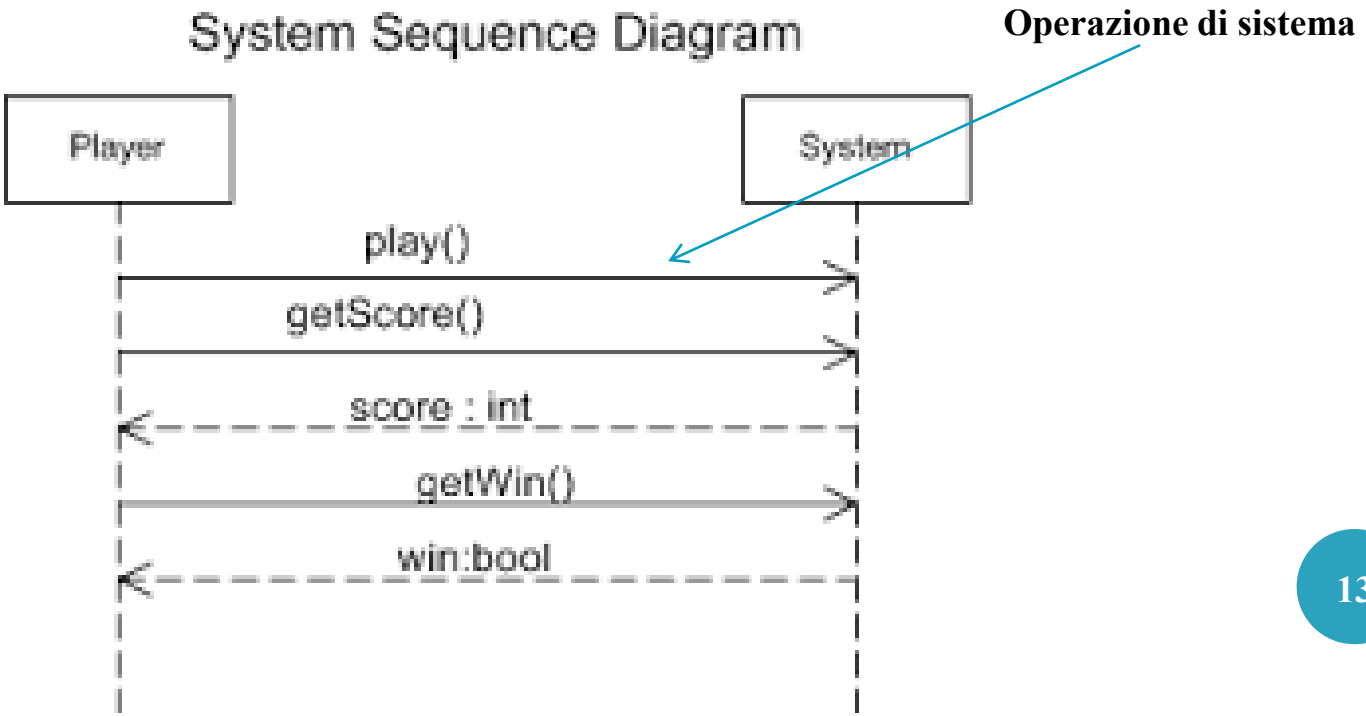
Applicare UML e i pattern
a cura di Luca Cabibbo
Pearson; 3° edizione



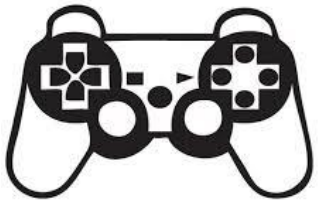
Pattern Controller (1)



- **Problema**: quale è il primo oggetto oltre lo strato di UI che riceve e coordina un **operazione di sistema?**
  - Operazione di sistema = evento di input principale nel sistema
  - Sono i msg (**verso System**) che compaiono nei SSD



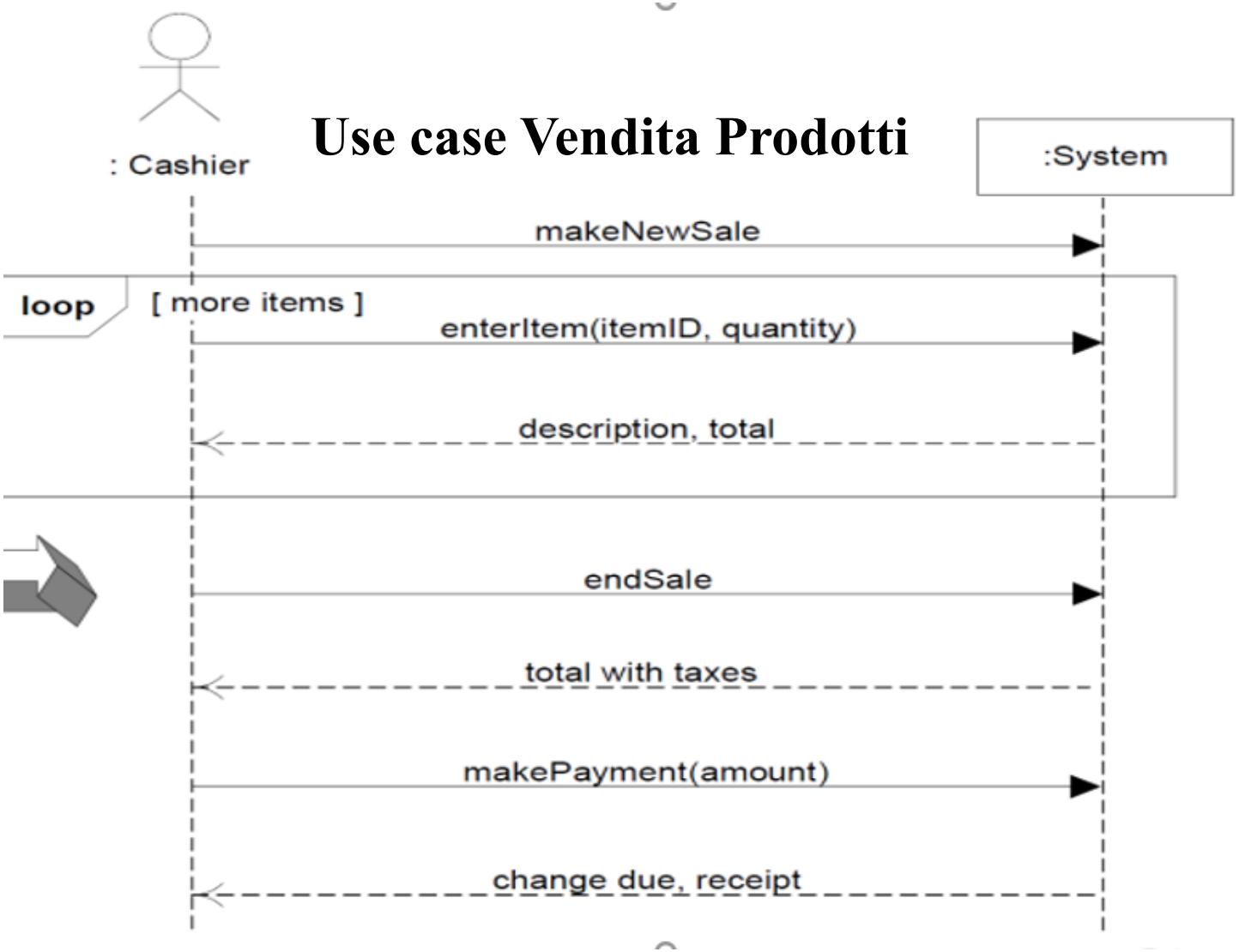
PATTERN CONTROLLER (2)



- **Problema**: quale è il primo oggetto oltre lo strato di UI che riceve e coordina un **operazione di sistema?**
  - Operazione di sistema = evento di input principale nel sistema
  - Sono i msg (**verso System**) che compaiono nei SSD

- **Soluzione**: assegna **la responsabilità** ad una classe che rappresenta una delle seguenti scelte:
  - **'Sistema piccolo'**: classe che rappresenta il sistema complessivo (o dispositivo su cui è eseguito il software)
  - **'Sistema grande':** classe relativa **al caso d'uso** all'interno del quale si verifica l'evento. Spesso chiamata <UseCaseName>Handler o <UseCaseName>Controller

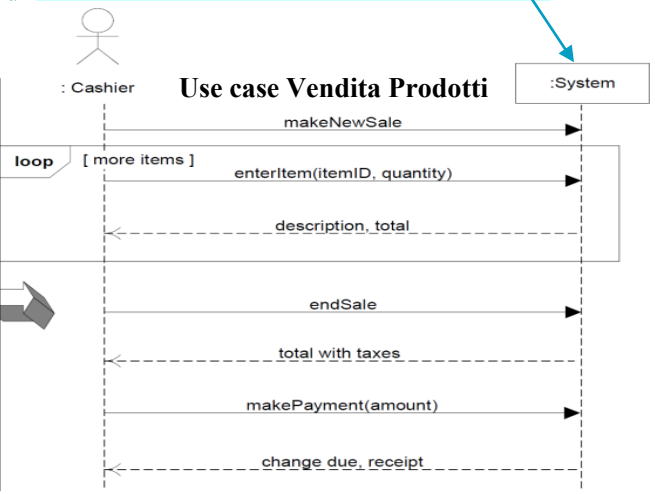
SISTEMA REGISTRATORE DI CASSA



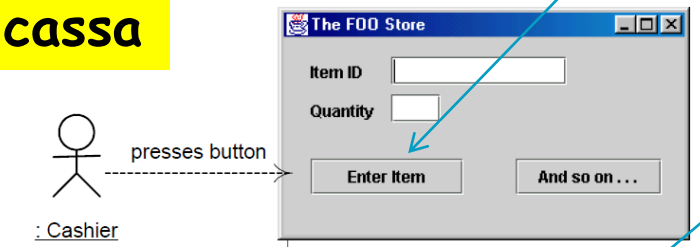
ESEMPIO PATTERN CONTROLLER

Sistema registratore di cassa

Problema: chi è il controller dell'operazione di sistema enterItem ?



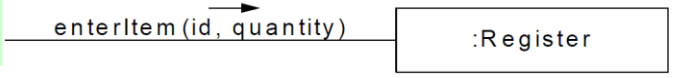
Al pulsante è associato un ActionListener



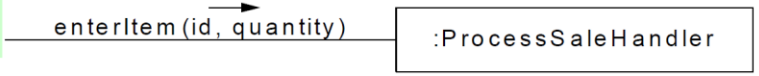
SaleJFrame (Java Swing)

Il metodo actionPerformed di **SaleJFrame** gestisce l'evento "Enter item" prelevando "id" e "quantity" dalla form e chiamando l'operazione

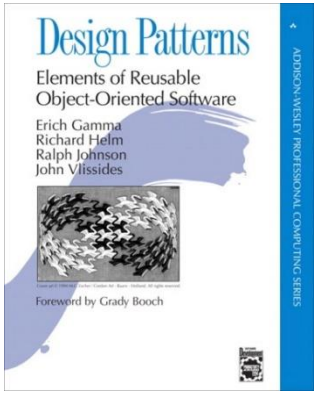
1° Soluzione: Register, un dispositivo fisico su cui viene eseguito il sw



2° Soluzione: ProcessSaleHandler, un controller di caso d'uso



Design Pattern classici



Quelli del “libro” originale GoF

- 23 Design Pattern

Tre categorie:

- **Creazionali**
  - creazione degli oggetti (in modo controllato)
- **Strutturali**
  - composizione di classi ed oggetti (strutture tipiche)
- **Comportamentali**
  - come classi (oggetti) interagiscono tra di loro e si distribuiscono le responsabilità

Spazio dei Design Pattern

Creazione	*Struttura*	*Comportamento*
Factory Method **Abstract Factory** Builder Prototype Singleton	**Adapter** Bridge Composite Decorator **Facade** Flyweight Proxy	Interpreter **Template Method** Chain of rensponsibility Command Iterator Mediator Memento **Observer** **State** Strategy Visitor

Vedremo a IS

https://dellabate.wordpress.com/2013/04/02/gof-patterns-visitor/

Spazio dei Design Pattern

Creazione	*Struttura*	*Comportamento*
Factory Method **Abstract Factory** Builder Prototype Singleton	**Adapter** Bridge Composite Decorator **Facade** Flyweight Proxy	Interpreter **Template Method** Chain of rensponsibility Command **Iterator** Mediator Memento **Observer** **State** Strategy **Visitor**

Descrizione di un Pattern

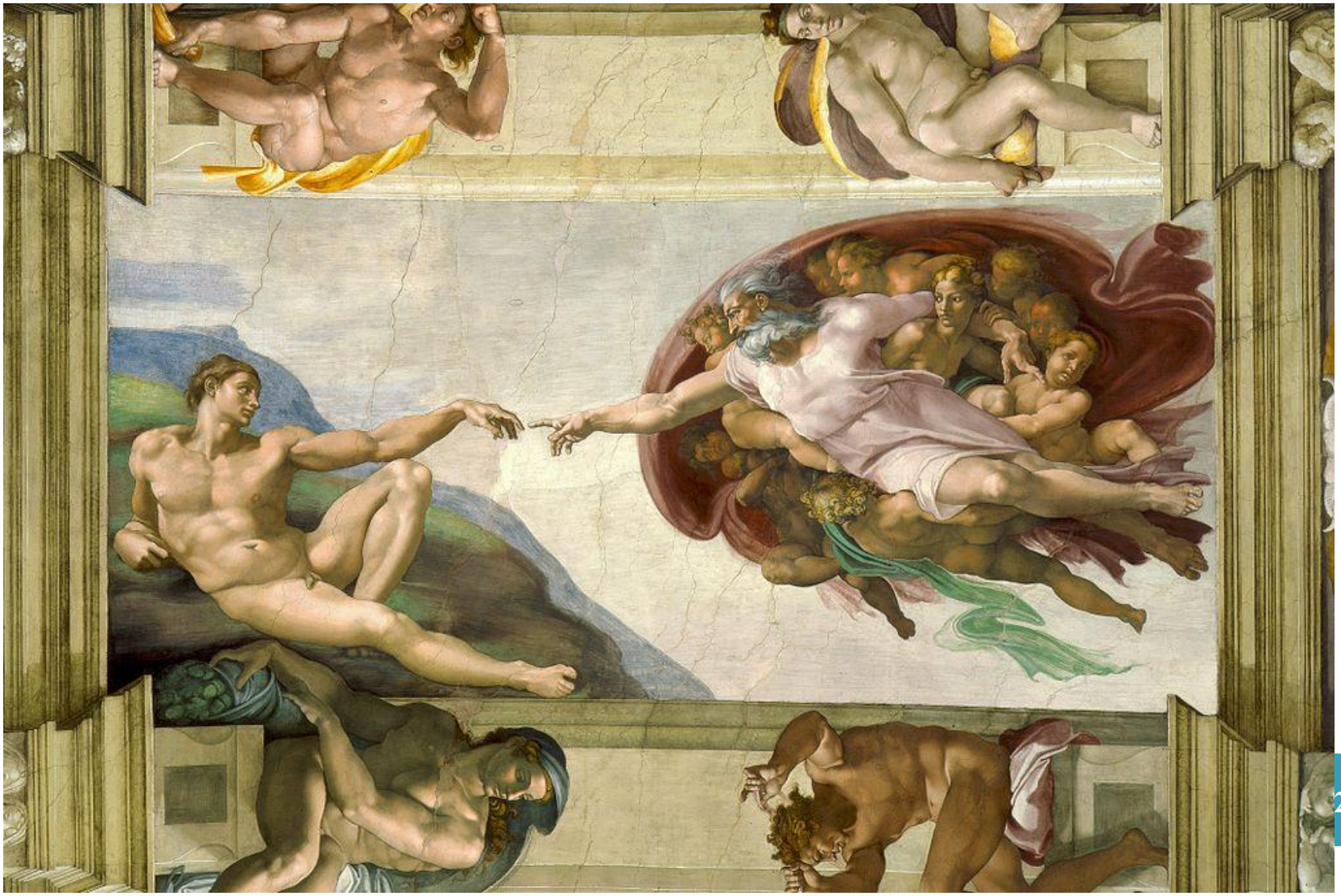
- Template piuttosto preciso!
  - Versione semplificata (solo parti fondamentali)

- **Nome (una/due parole)**
  - Significativo/esplicativo, che aiuta a capire a cosa serve il pattern
- **Problema**
  - descrive **quando** applicare il pattern
  - può contenere anche delle precondizioni per poterlo applicare
- **Soluzione**
  - descrive **astr**
    le loro relazioni, responsabilità e collaborazioni
- **Conseguenze**
  - Pro/contro della sua applicazione

Non sempre precisa e completa, è uno sketch

Iniziamo con ...

Iniziamo con ...

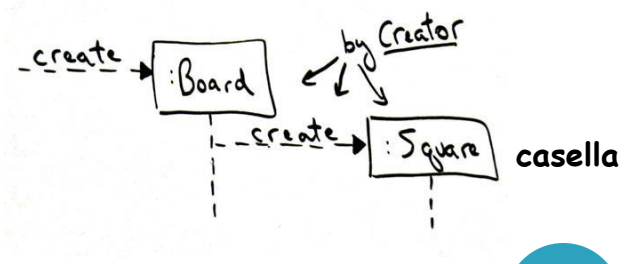


I PATTERN CREAZIONALI

- Sono pattern che hanno a che fare con la creazione di istanze

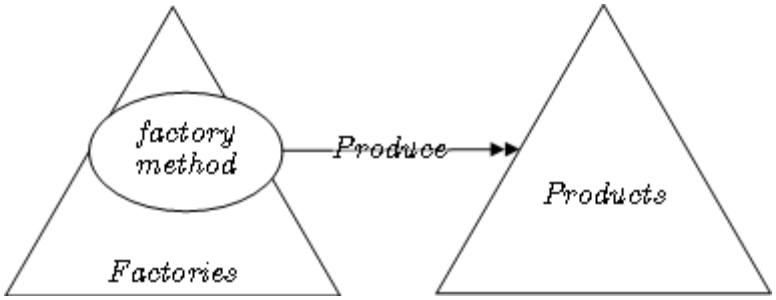
Chi crea un oggetto di tipo A?

- Di solito per risolvere questo problema ci si affida al pattern GRASP **Creator**
  - Assegna alla classe B la responsabilità di creare un oggetto di tipo A se una delle condizioni è vera (più sono vere meglio è)
    - B contiene o aggrega oggetti di tipo A
    - B utilizza strettamente A
    - B possiede i dati per l'inizializzazione di A



- Tuttavia esistono casi in cui non conviene applicare il pattern **Creator ...**

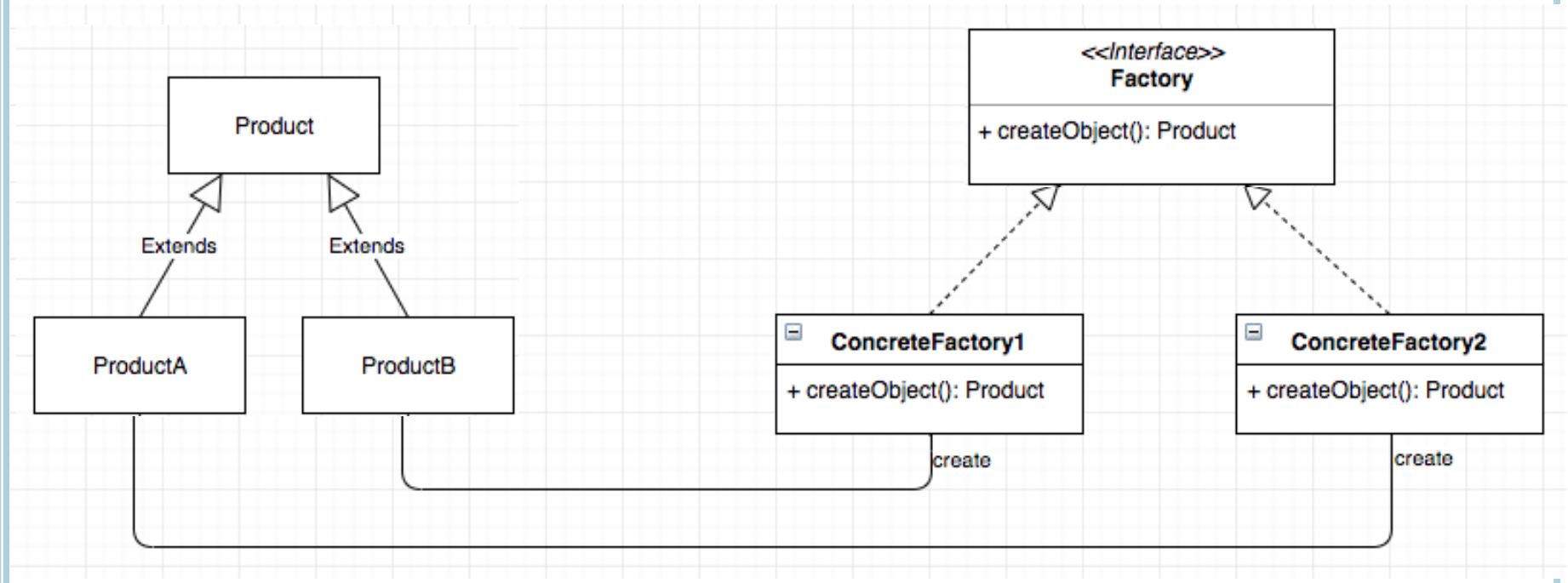
FACTORY



- Quando si vuole **nascondere la logica di creazione** perchè potenzialmente complessa
- Quando si vuole separare la logica di creazione dalla logica applicativa pura
- Perchè si vogliono introdurre strategie per la gestione della memoria per migliorare le prestazioni
  - Caching o 'riciclaggio' di oggetti

- In questi casi si ricorre al conce[illegible]tory (fabbrica), in cui **viene definito un oggetto Factory per creare gli oggetti ...**

Factory Method Design Pattern



Esempio Factory Method

```
public interface Animal { String getCall(); }
```

```
public class Cat implements Animal {
  @Override
  public String getCall() { return “Miao”; }
}
```

```
public class Dog implements Animal {
  @Override
  public String getCall() { return ”Bau”; }
}
```

```
public class Main {
   public static void main(String[] args) {
   AnimalFactory factory = new AnimalFactory();
   …
  factory.getAnimal(type).getCall();
  …
   }
}
```

```
public class AnimalFactory {
    public AnimalFactory(){}

    public Animal getAnimal (AnimalEnum type){
        Animal retval = null;
        switch (type){
            case Cat:
                retval = new Cat();
                break;
            case Dog:
                retval = new Dog();
                break;
        }
        return retval;
    }
}
```

```
public enum AnimalEnum { Cat, Dog }
```


https://italiancoders.it/factory-method-design-pattern/

ABSTRACT FACTORY

- Fornisce una soluzione **per creare famiglie di prodotti** (oggetti connessi o dipendenti tra loro), in modo che non ci sia necessità da parte dei **client** di specificare le classi concrete dei prodotti all'interno del proprio codice

Famiglia prodotti "1" (widget): **`Window1`**, **`ScrollBar1`**, **`...`**

NO!!!

Client (GUI)

```
Window w = new Window1();
...
ScrollBar s = new ScrollBar1();
```

Quando usare Abstract Factory?

- Un sistema deve essere indipendente da come i suoi prodotti **sono creati**, composti e rappresentati
- Un sistema deve essere **configurato** per **una di diverse** famiglie di prodotti disponibili
  - Ad esempio deve poter funzionare con:
    - Famiglia 1: Window1, ScrollBar1, …
    - Famiglia 2: Window2, ScrollBar2, …
- Un famiglia di prodotti collegati tra loro è progettata per funzionare assieme, e bisogna rispettare questo vincolo
    - Es. Window1 e ScrollBar1 (e non Window1 e ScrollBar2)

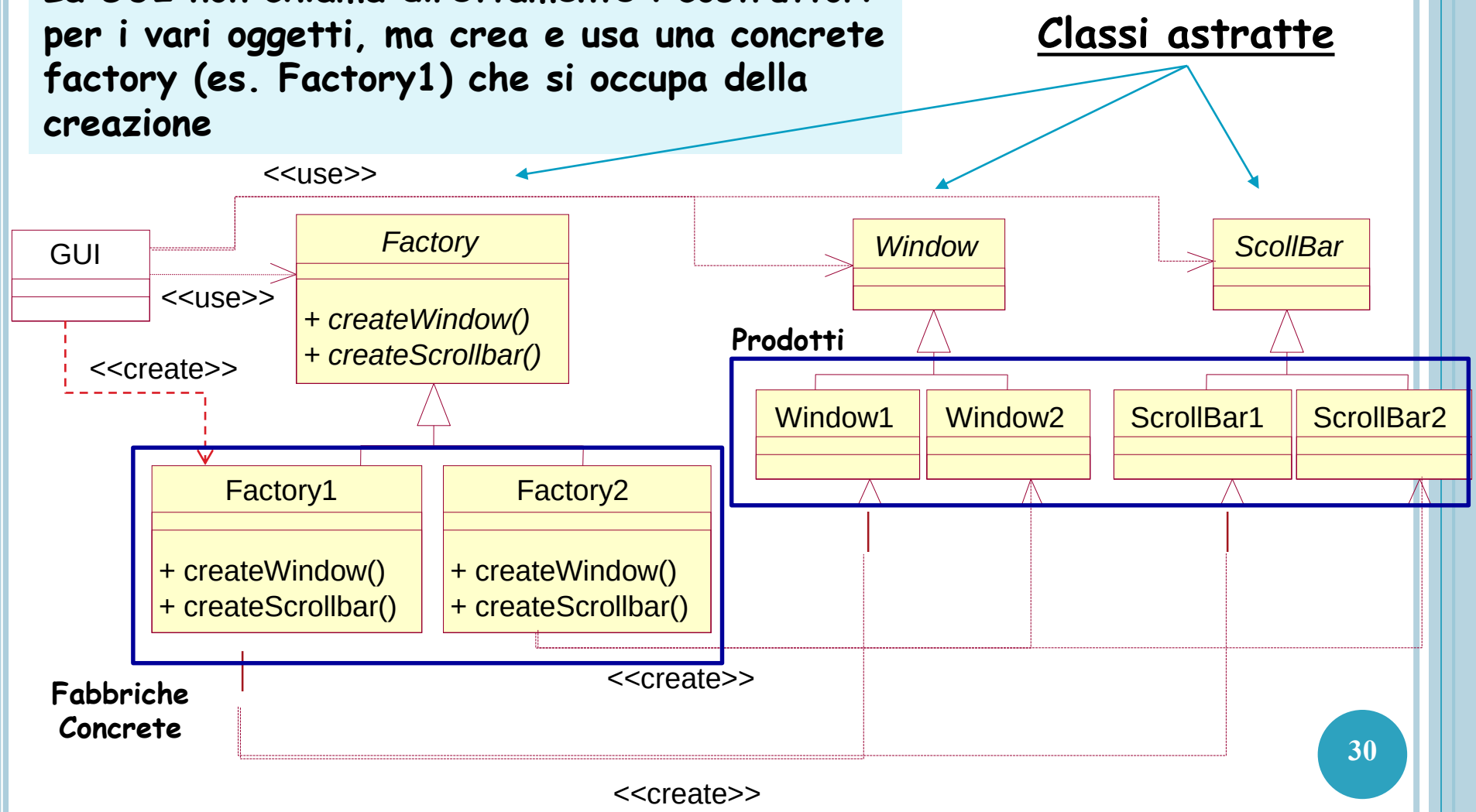
ESEMPIO/PROBLEMA

- Vogliamo progettare una GUI
  - **Multipiattaforma**
    - Se eseguita su piattaforma 1, istanzia prodotti della famiglia “1”: **`Window1`**, **`ScrollBar1, ...`**
    - Se eseguita su piattaforma 2, istanzia prodotti della famiglia “2”: **`Window2`**, **`ScrollBar2, ...`**
  - **Però che non dipenda troppo dalla piattaforma**
    - La GUI (che creerà/istanzierà quelle classi) non deve sapere quali classi concrete istanzia
    - Bisogna evitare che la GUI accoppi (sbagliando) per esempio **`Window1`** e **`ScrollBar2`**

Soluzione: usare una “fabbrica astratta”!

La fabbrica (astratta) dei prodotti

La GUI non chiama direttamente i costruttori per i vari oggetti, ma crea e usa una concrete factory (es. Factory1) che si occupa della creazione



COSA FA LA GUI?

- Senza la fabbrica:

```
Window w = new Window1();
...
ScrollBar s = new ScrollBar1();
```

- Con la fabbrica:

```
Factory f = new Factory1();
Window w = f.createWindow();
...
ScrollBar s = f.createScrollBar();
```

Differenze

Senza la "fabbrica"

- **`GUI`** deve "conoscere" **`Window1`** e **`ScrollBar1`**
- **`GUI`** crea una **`Window1`**
- Poi crea una **`ScrollBar1`**
- **`GUI`** ha la responsabilità di accoppiare correttamente **`Window1`** e **`ScrollBar1`**

Con la "fabbrica"

- **`GUI`** chiede una fabbrica della famiglia "1"
- Poi chiede alla fabbrica una **`Window`**
- Poi chiede alla fabbrica una **`ScrollBar`**
- La responsabilità di accoppiare **`Window1`** e **`ScrollBar1`** è delegata alla fabbrica

- **Separazione delle responsabilità (concerns)**
  - Disaccoppia la responsabilità della creazione e accoppiamento (nella fabbrica)…
  - … dalla responsabilità dell'uso dei prodotti (nel cliente)
- **Perché "astratta"?**
  - La GUI conosce/usa solo classi astratte dei prodotti

Applicazione dell'Abstract factory

Usando la fabbrica astratta si produce un design, e quindi un codice, migliore!

Design pattern: a cosa servono?

- Aiutano ad applicare i principi di **buona** progettazione OO
  - Es. high cohesion - low coupling
- Aiutano a creare "buoni" design
  - Possono essere applicati sia durante la **modellazione** che la **codifica**
- Facilitano la comunicazione tra sviluppatori

Diagramma Abstract Factory

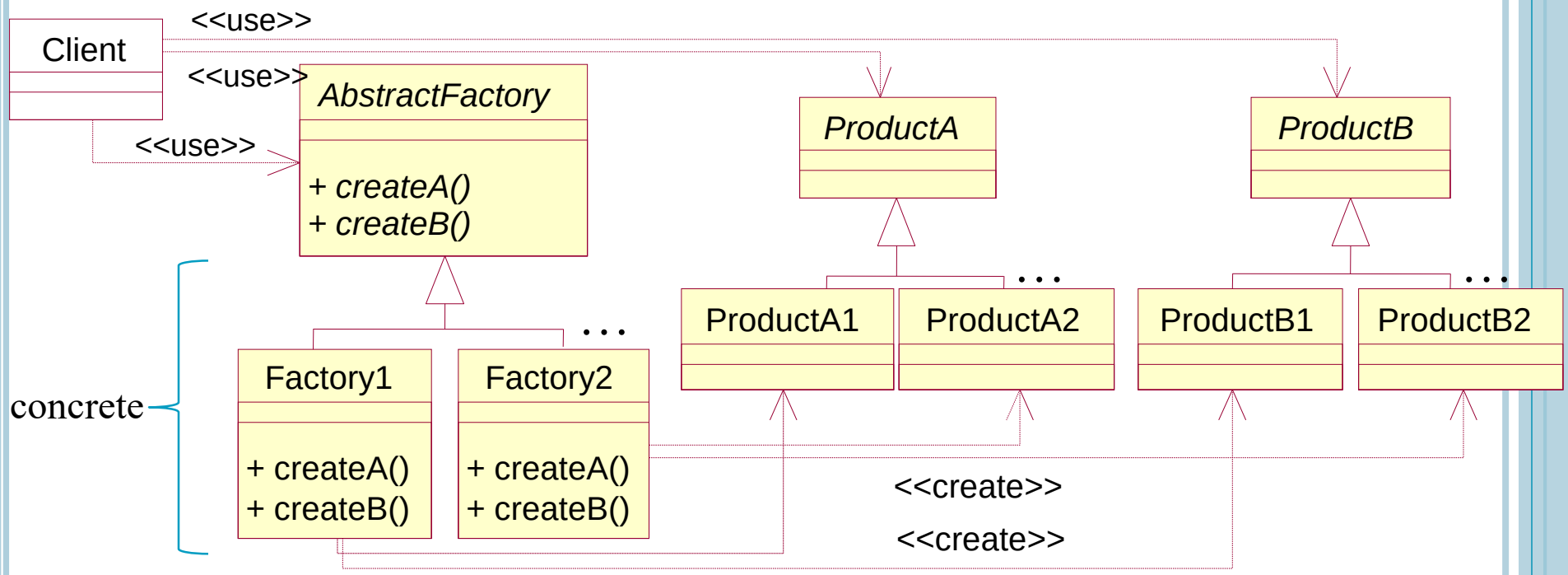
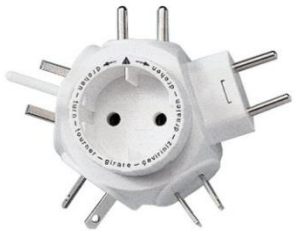


Diagramma che si trova nei libri/cataloghi ....

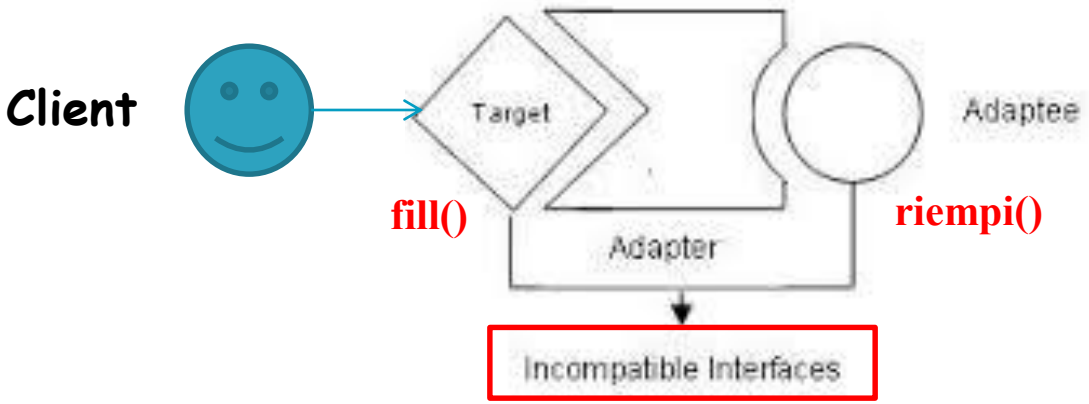
Abstract factory

- Conseguenze:
  - Isola le classi concrete 
    - Il client (la GUI nel nostro esempio) non deve sapere niente dei prodotti che userà, neanche al momento dell'istanziazione dei prodotti (ci pensa la fabbrica)
  - Rende il cambio della famiglia dei prodotti facile 
    - basta cambiare la sola classe ConcreteFactory (una linea di codice)
  - Favorisce la consistenza tra i vari prodotti di una famiglia (es. **`Window1`** e **`ScrollBar1`**) 
  - Non è facile aggiungere nuovi prodotti 
    - Può richiedere cambiamenti all'interfaccia dell'Abstract Factory e alle sue sottoclassi
    - Richiede l'aggiunta di una nuova classe ConcreteFactory e di nuovi AbstractProducts e Products

ADAPTER



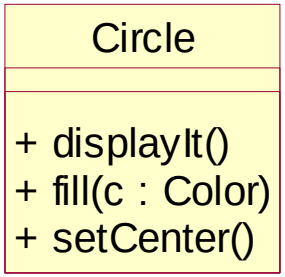
- **Converte l'interfaccia di una classe in un'altra interfaccia che il cliente si aspetta**
- Permette a delle classi di lavorare assieme, anche se non potrebbero visto che hanno interfacce incompatibili ...



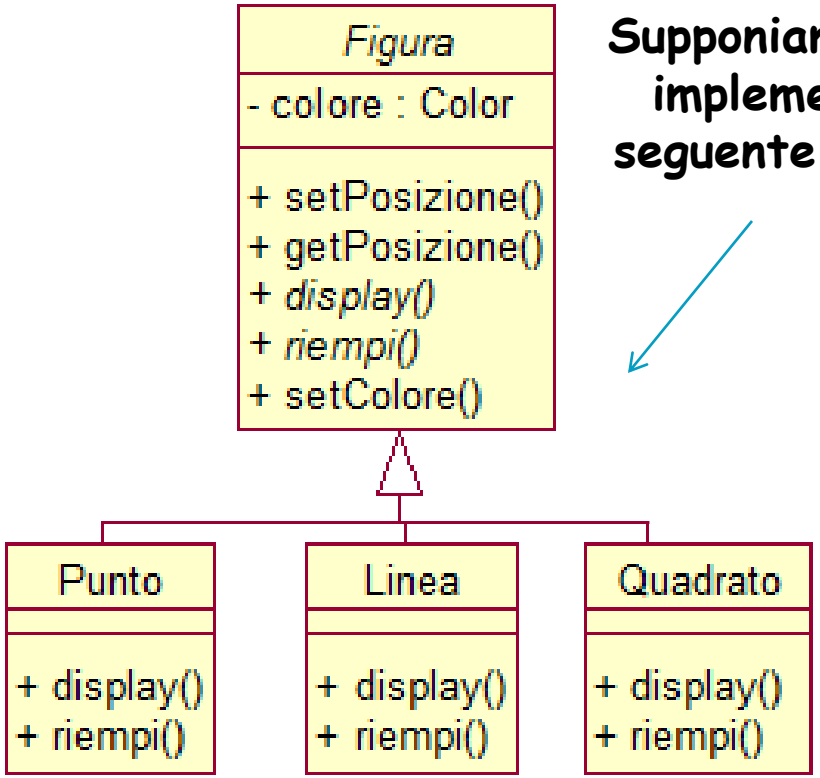
- **Problema**
  - usare una classe esistente, la cui interfaccia non è quella che il cliente si aspetta
  - si vuole creare una classe che collabora con classi non correlate, o che non si conoscono ancora

ADAPTER – ESEMPIO (1)

Supponiamo di avere anche:



Supponiamo di avere implementato la seguente gerarchia:

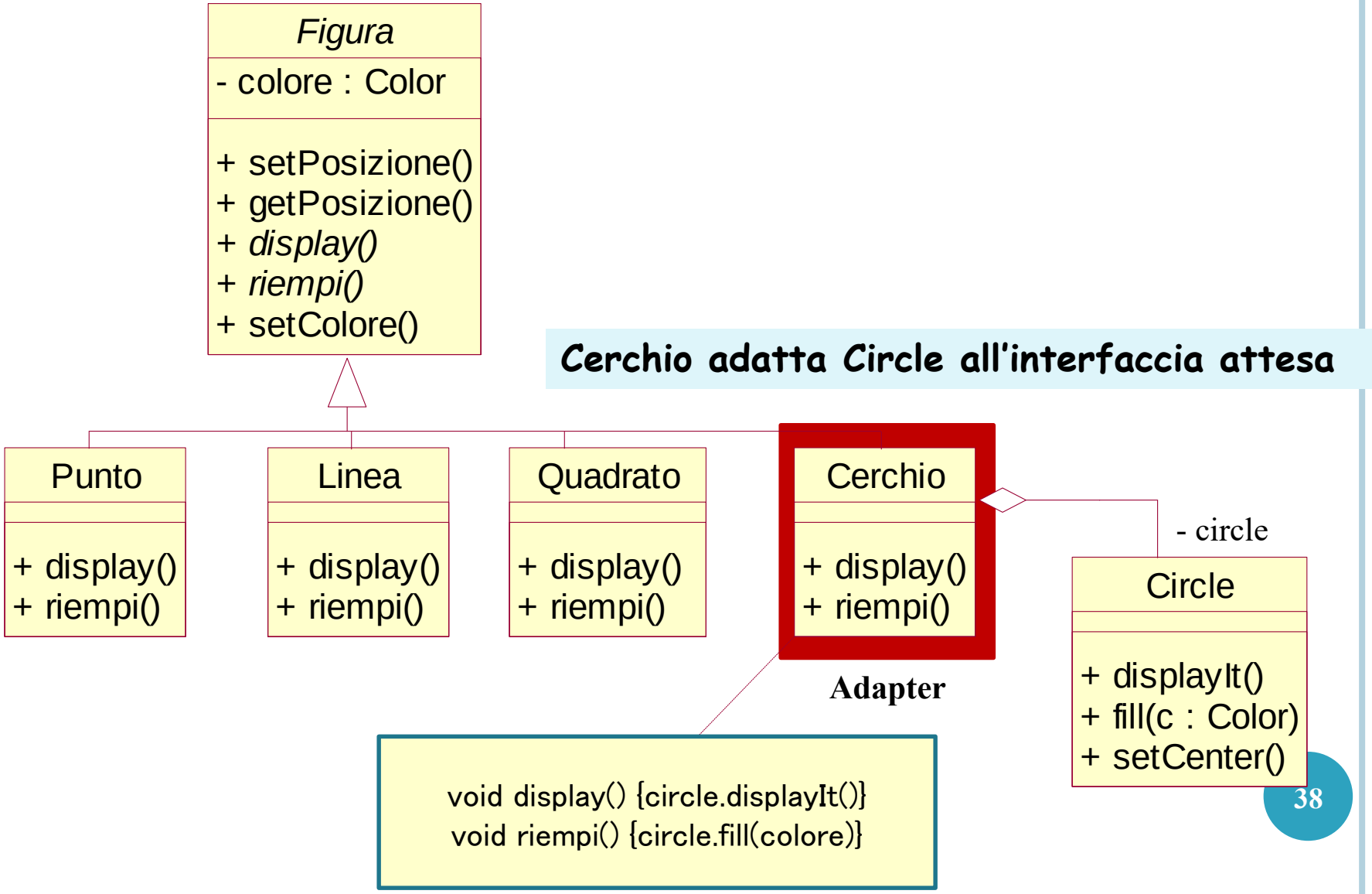


- Vogliamo aggiungere **`Cerchio`**
  - Lo implementiamo da zero?
    - Fatica inutile …
  - Usiamo la classe **`Circle`**?
    - Ha un'interfaccia diversa …

- e non possiamo modificare **`Circle`**
  - non abbiamo il sorgente
  - dobbiamo usarla così com'è (es. perchè usata da altre classi)

Soluzione: creare un adattatore e usare Circle!

ADAPTER – ESEMPIO (2)

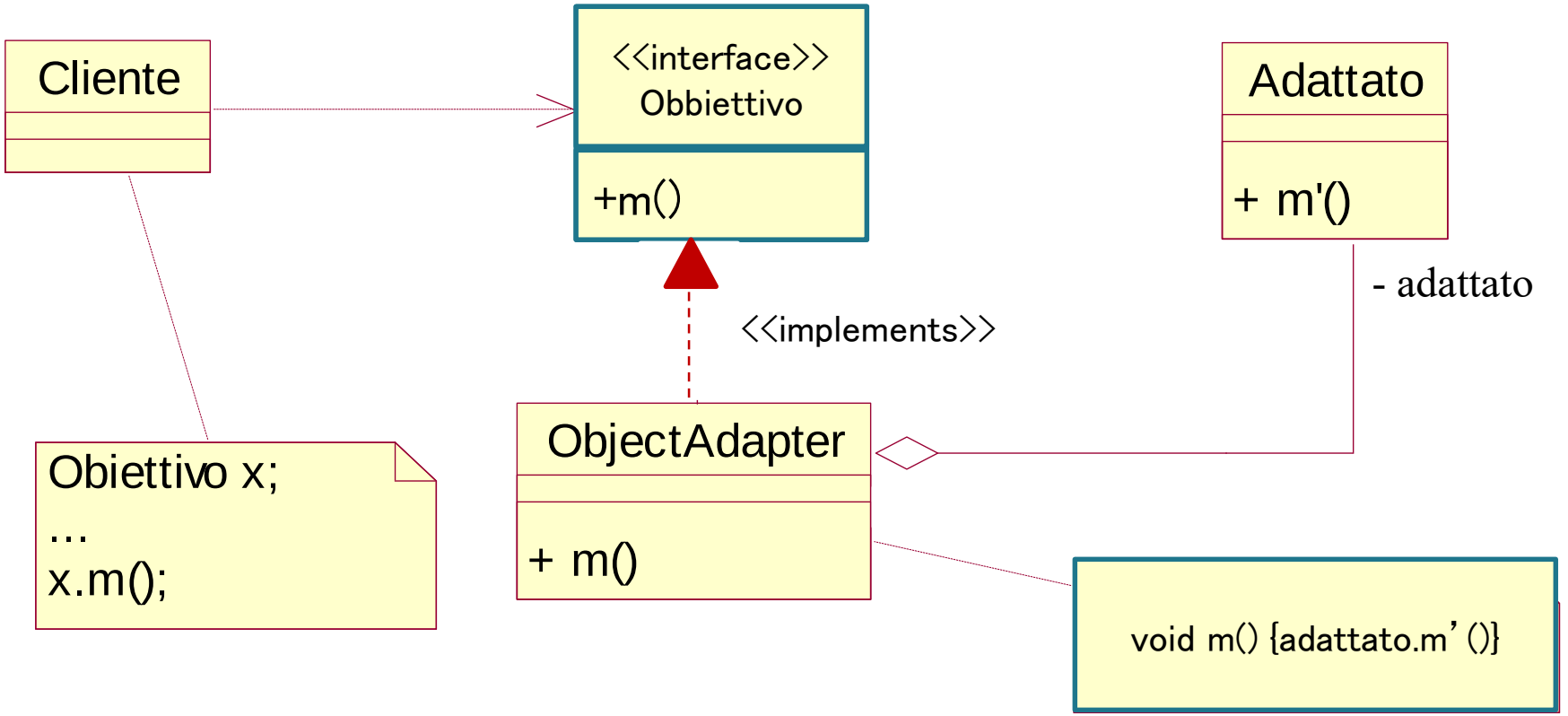


Due tipi di Adapter

- **Oggetto adattatore (Object Adapter)**
  - Basato su delega/composizione
- **Classe adattatore (Class Adapter)**
  - Basato su ereditarietà
  - L'adattatore eredita sia dall'interfaccia attesa sia dalla classe adattata
  - No eredità multipla $\Rightarrow$ l'interfaccia attesa deve essere un'interfaccia, non una classe

Il cliente vorrebbe chiamare m() però noi abbiamo una classe che fornisce m'() …

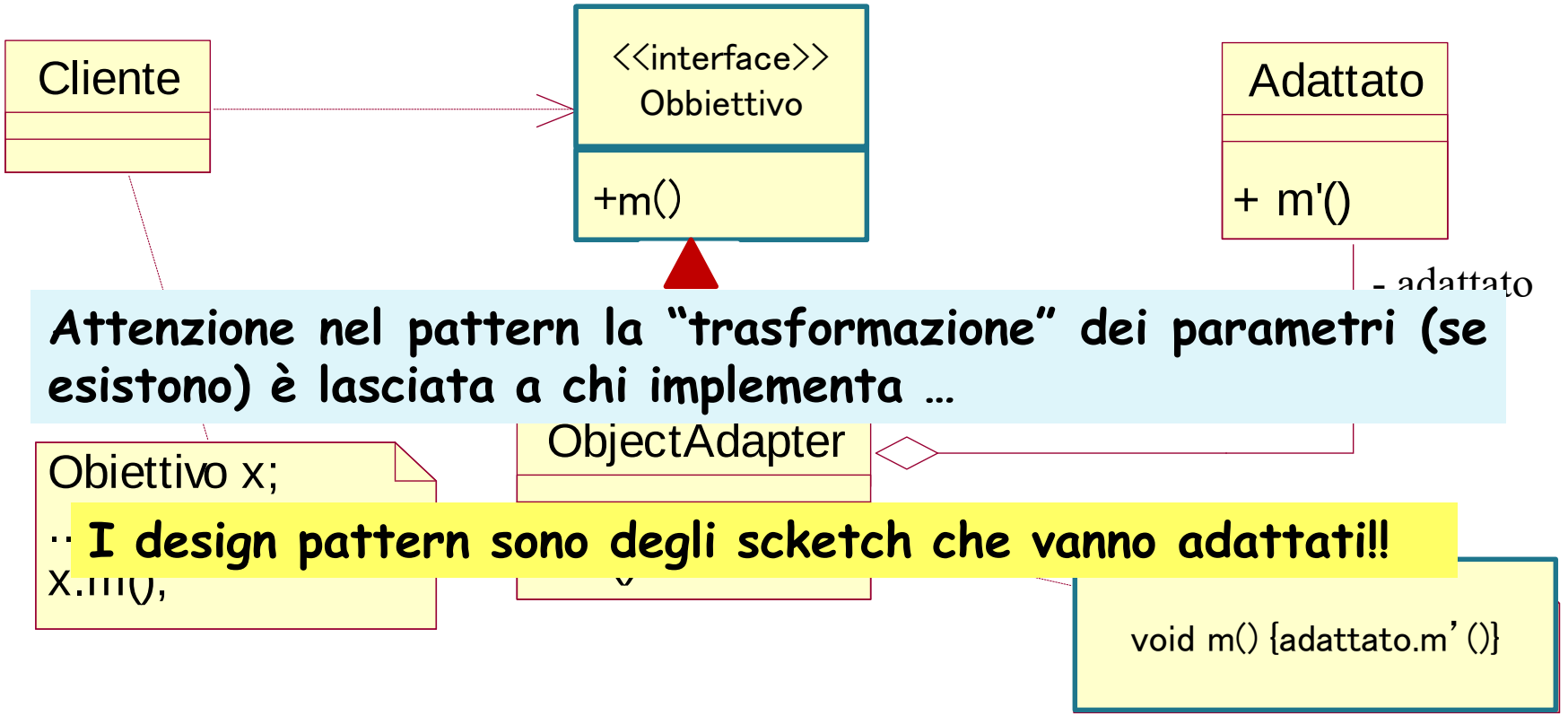
DIAGRAMMA (OBJECT) ADAPTER



• ObjectAdapter converte, ovvero adatta, l'interfaccia che il cliente si aspetta (Obbiettivo) all'interfaccia dell'adattato
• Il cliente usa ObjectAdapter come fosse l'oggetto Adattato
• ObjectAdapter possiede il riferimento a Adattato e sa come invocarlo

Il cliente vorrebbe chiamare m() però noi abbiamo una classe che fornisce m'() …

DIAGRAMMA (OBJECT) ADAPTER



• ObjectAdapter converte, ovvero adatta, l'interfaccia che il cliente si aspetta (Obbiettivo) all'interfaccia dell'adattato
• Il cliente usa ObjectAdapter come fosse l'oggetto Adattato
• ObjectAdatper possiede il riferimento a Adattato e sa come invocarlo