

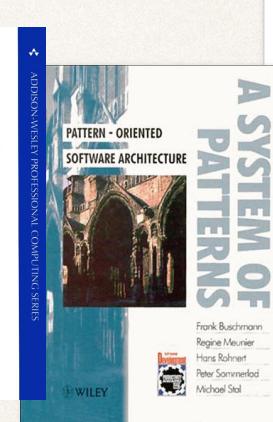
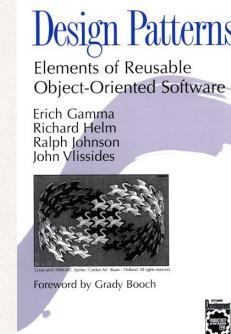
Design Pattern & Application Framework

Alberto Gianoli

Univ. Ferrara - Corso di Laurea in Informatica

Dove?

- ❖ Sommerville, qualcosa cap. 18
- ❖ Pressman, appena citati nel cap. 12
- ❖ molti libri sull'argomento



- ❖ online:
 - ❖ <http://www.javacamp.org/designPattern/>
 - ❖ <http://www.javaworld.com/columns/jw-java-design-patterns-index.html>
 - ❖ <http://www.oodesign.com/>
 - ❖ <http://www.fluffycat.com/java/patterns.html>

Il termine “pattern”

pattern (ˈpætərn)

noun

1. **a repeated decorative design:** *a neat blue herringbone pattern.*

- ✧ an arrangement or sequence regularly found in comparable objects or events: *the house had been built on the usual pattern*
- ✧ **a regular and intelligible form or sequence discernible in certain actions or situations:** *a complicating factor is the change in working patterns.*

2. **a model or design used as a guide in needlework and other crafts** (see model)

- ✧ **a set of instructions to be followed in making a sewn or knitted item.**
- ✧ **a wooden o metal model from which a mold is made for a casting.**
- ✧ **an example for others to follow:** *he set the pattern for subsequent study.*
- ✧ **a sample of cloth or wallpaper.**

Design pattern: la storia

- ❖ C. Alexander: architetto, introduce il concetto nel contesto della progettazione urbanistica e architettonica (1975)
- ❖ Beck e Cunningham (1987) applicano l'idea alla progettazione del software
- ❖ Termine divenuto popolare grazie a un libro: “Design Patterns” di Gamma, Helm, Johnson, Vlissides (GoF: gang of four)

La definizione

- ❖ **Design Pattern: soluzione progettuale assodata (ossia convalidata dal suo utilizzo con successo in più di un progetto) per un problema ricorrente in un determinato contesto (condizioni al contorno che vincolano la scelta della soluzione)**
- ❖ permettono riuso dell'esperienza di progettazione (propria o altrui)
- ❖ conducono a sistemi più contenibili
- ❖ aumentano la produttività
- ❖ linguaggio comune per comunicare scelte progettuali (se i pattern sono opportunamente codificati e identificati)

Motivazione

Prendiamo un esempio

- ❖ **Contesto:** una casa (ambiente) dove vivere
 - ❖ **Problema:** proteggere la casa dalla pioggia
 - ❖ **Soluzione:**
 - ❖ vai ad abitare in una caverna
 - ❖ vai ad abitare in una tenda
- costruisci una casa con un tetto



E' una soluzione, ma non è una ricetta: la soluzione è avere un tetto, ma ci sono molti modi di farlo

Detto in altro modo: “concept reuse”

- ❖ Se ci limitiamo a riusare un programma, o un componente sviluppato in un'altra occasione, dobbiamo seguire le stesse decisioni progettuali fatte da chi ha sviluppato originariamente quel codice
- ❖ Questo può implicare delle limitazioni al riuso
- ❖ Quindi a noi interessa “riusare” in senso più astratto
 - ❖ l'approccio utilizzato è descritto in modo indipendente (dal linguaggio, dall'architettura, ...)
 - ❖ noi svilupperemo l'implementazione secondo le nostre decisioni progettuali

Cosa non sono

- ❖ Cos'è una libreria di codice?
 - ❖ In OO può essere una struttura di classi per risolvere certi problemi
- ❖ Quindi le librerie e i framework sono anch'essi Design Patterns?
 - ❖ No, sono una specifica implementazione, e linkiamo il loro codice
 - ❖ No, al limite loro si basano su dei Design Pattern

Ricapitolando

Cosa sono

- ✧ costituiscono un vocabolario comune per i progettisti
- ✧ sono una notazione abbreviata per comunicare in modo efficace principi complessi
- ✧ aiutano a documentare l'architettura
- ✧ catturano parti critiche di un sistema in forma compatta
- ✧ mostrano più di una soluzione
- ✧ descrivono astrazioni software

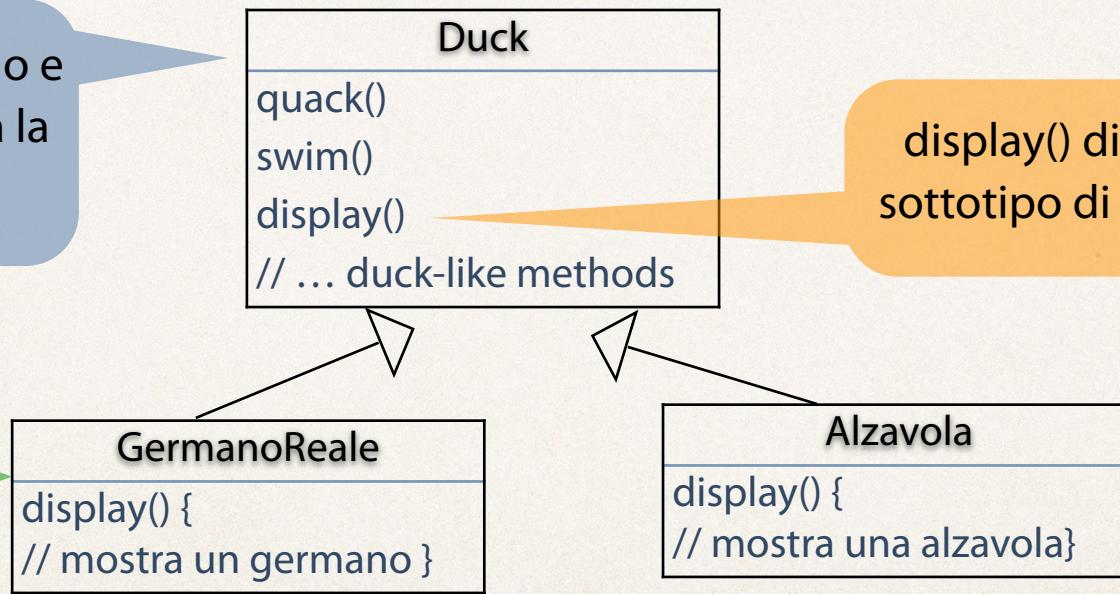
Cosa non sono

- ✧ non sono una soluzione precisa di problemi progettuali
- ✧ non risolvono tutti i problemi progettuali

Facciamo il punto: perché l'ereditarietà non basta?

- Stiamo lavorando su un nuovo gioco di simulazione sviluppato in linguaggio OO: la simulazione di uno stagno (*SimDuck*)
- Vogliamo mostrare i vari tipi di anatre che nuotano e fanno “quack”

Tutte le anatre nuotano e fanno quack: ci pensa la superclasse

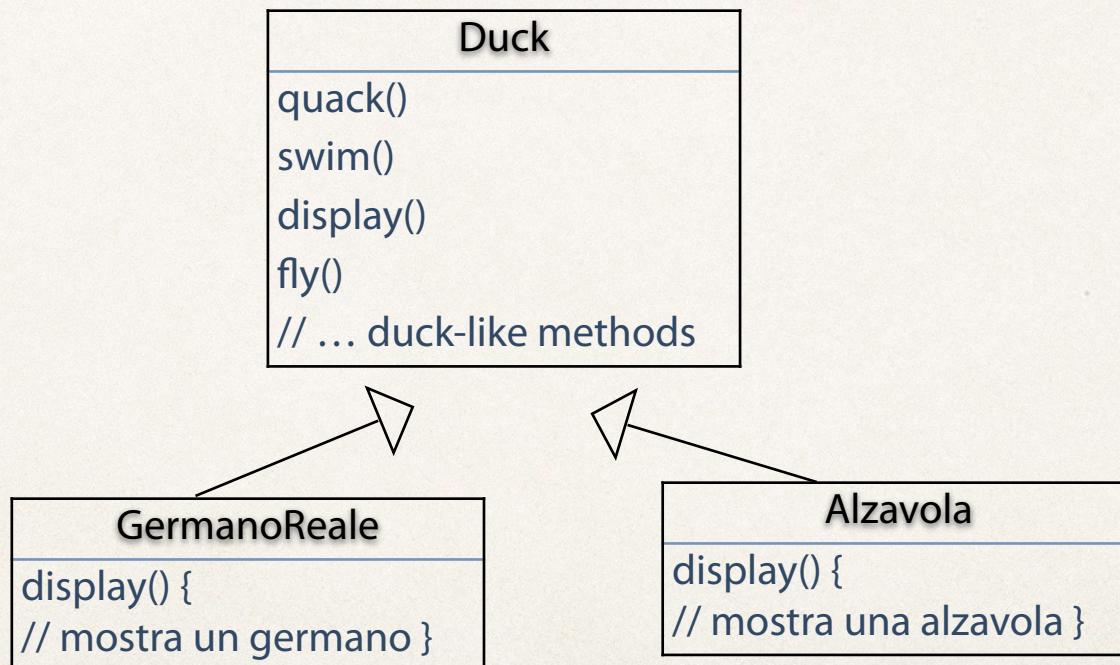


display() di tipo abstract: ogni sottotipo di anatra ha il suo look

Ogni sottotipo deve implementare il suo display()

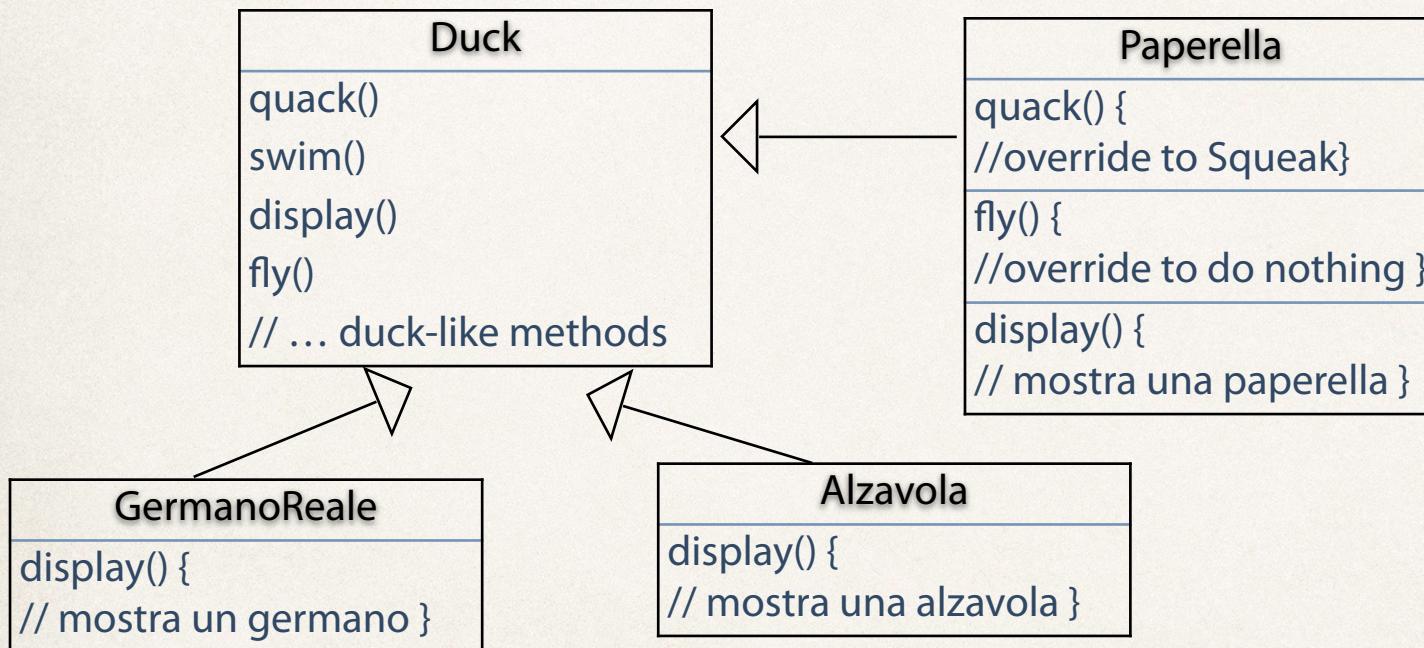
Facciamo il punto: perché l'ereditarietà non basta?

- nuova versione: le anatre devono anche volare....
 - creiamo il metodo nella superclasse e siamo a posto..



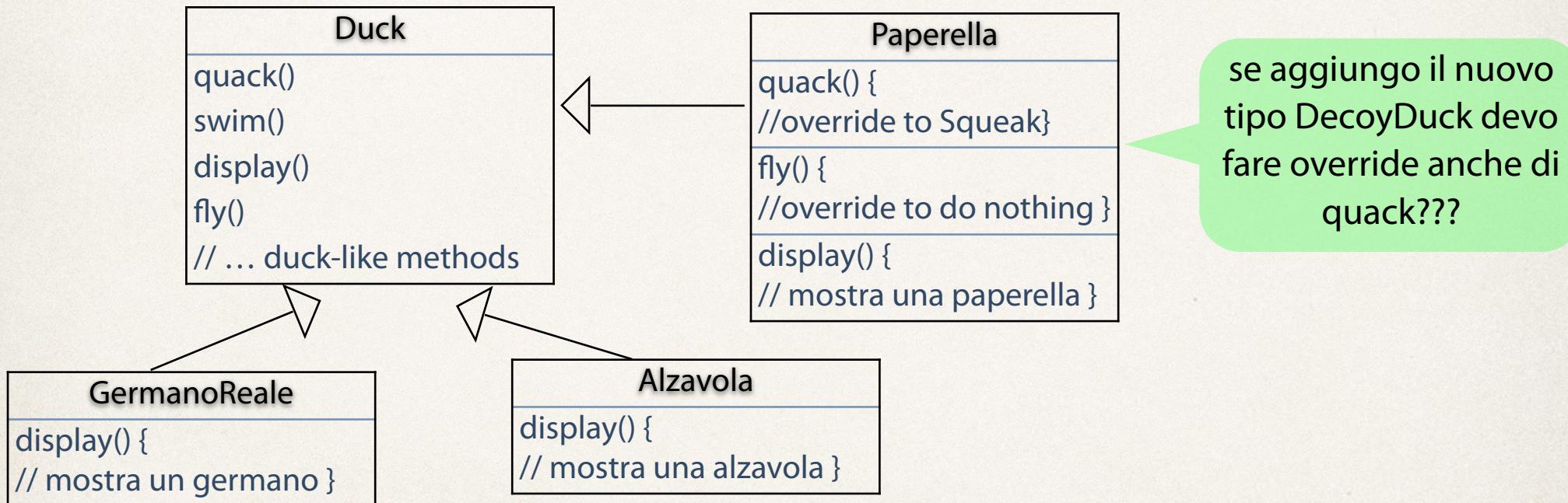
Facciamo il punto: perché l'ereditarietà non basta?

- ehm ehm.... veramente nel nostro gioco ci sono anche le paperelle di gomma... e le paperelle di gomma non volano



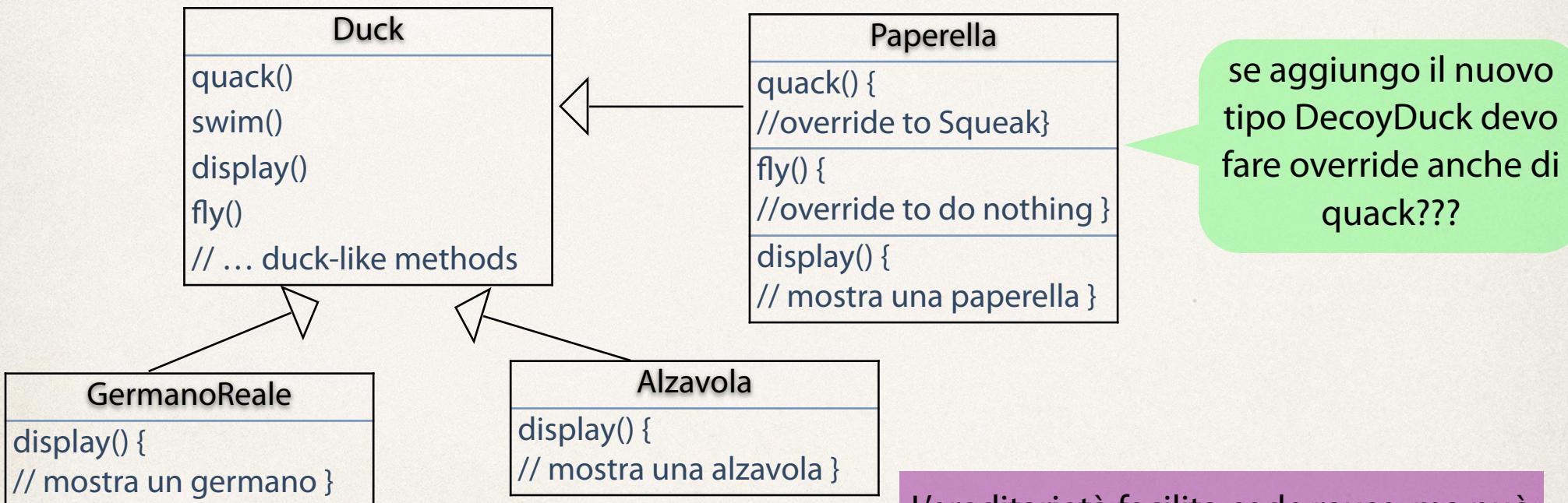
Facciamo il punto: perché l'ereditarietà non basta?

- ehm ehm.... veramente nel nostro gioco ci sono anche le paperelle di gomma... e le paperelle di gomma non volano



Facciamo il punto: perché l'ereditarietà non basta?

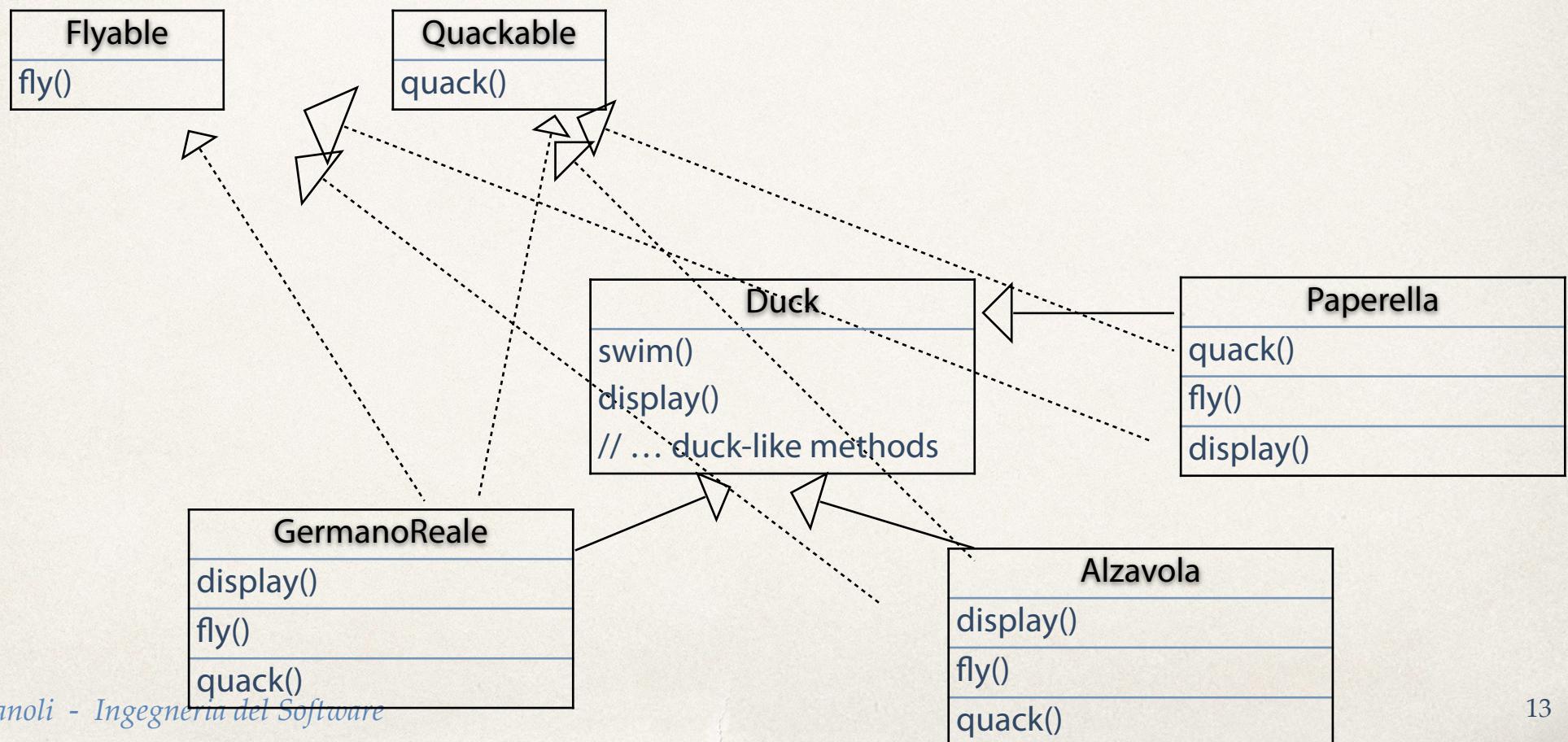
- ehm ehm.... veramente nel nostro gioco ci sono anche le paperelle di gomma... e le paperelle di gomma non volano



L'ereditarietà facilita code reuse, ma può complicare enormemente le cose quando si parla di maintenance...

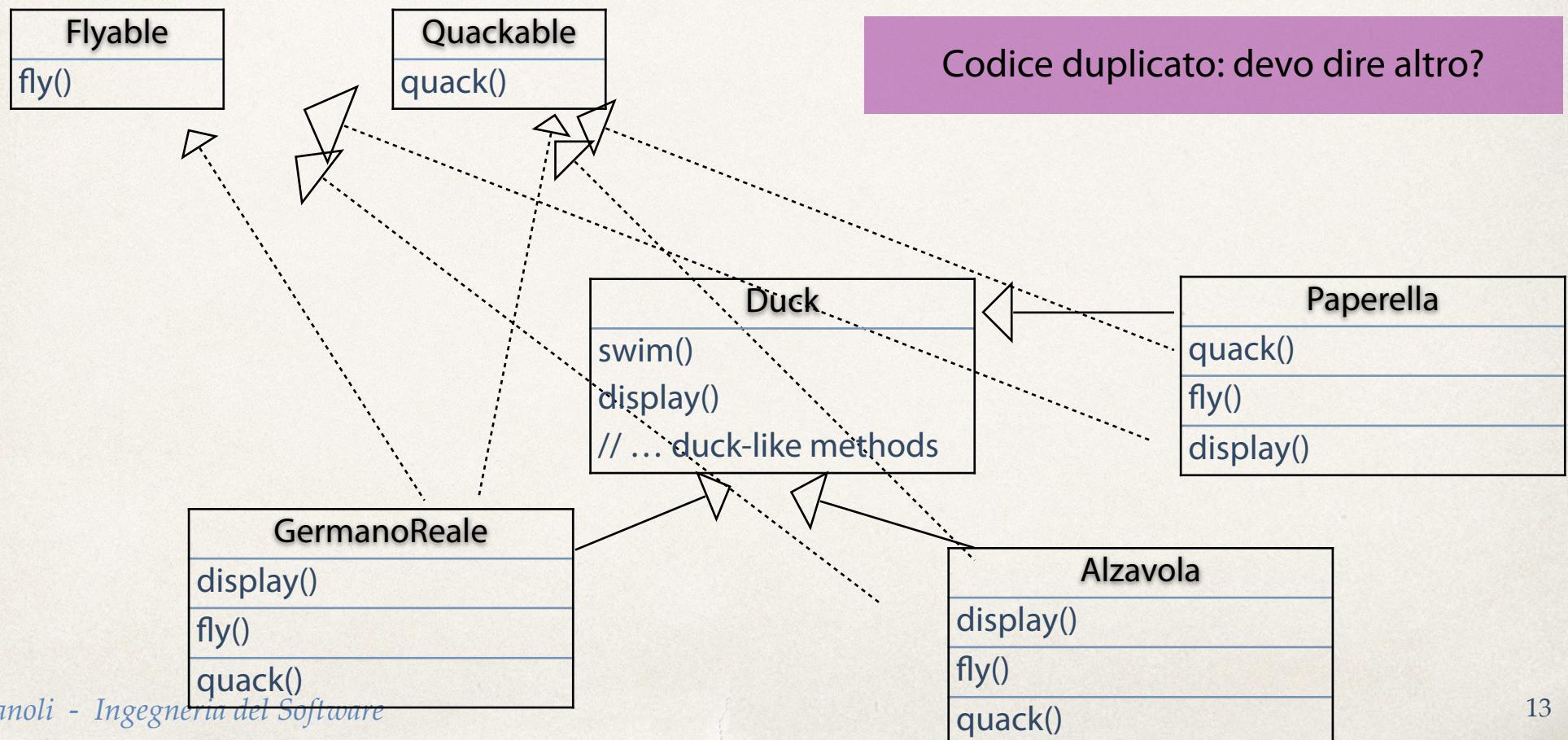
Facciamo il punto: perché l'ereditarietà non basta?

- Uhmmmm... se usassi una interfaccia?



Facciamo il punto: perché l'ereditarietà non basta?

- Uhmmmm... se usassi una interfaccia?

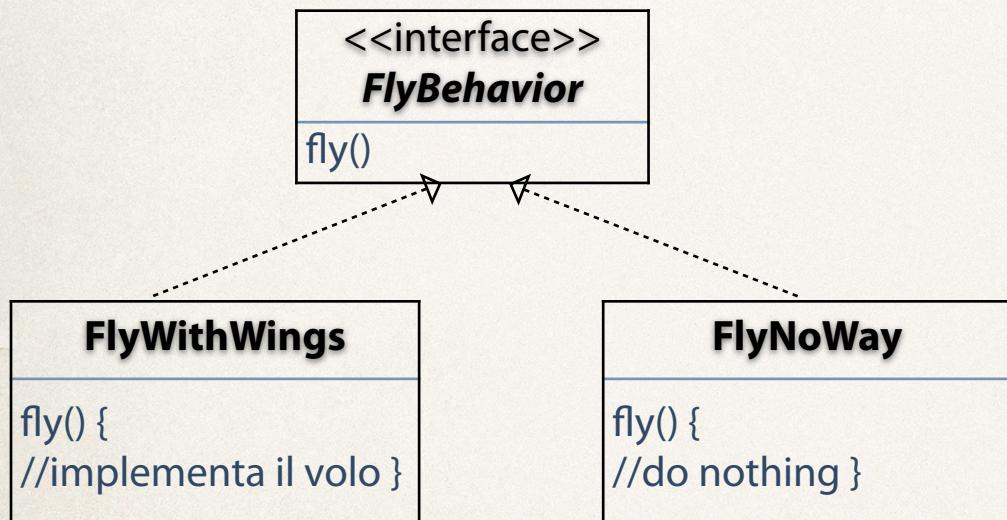


Facciamo il punto: perché l'ereditarietà non basta?

- ❖ Il comportamento dell'anatra può cambiare notevolmente tra le sottoclassi, e non è appropriato per tutte le sottoclassi di avere quei comportamenti
 - ➡ quindi l'ereditarietà non ci può aiutare
- ❖ **Design principle:** identificate gli aspetti dell'applicazione che variano, e separateli da quelli che non variano

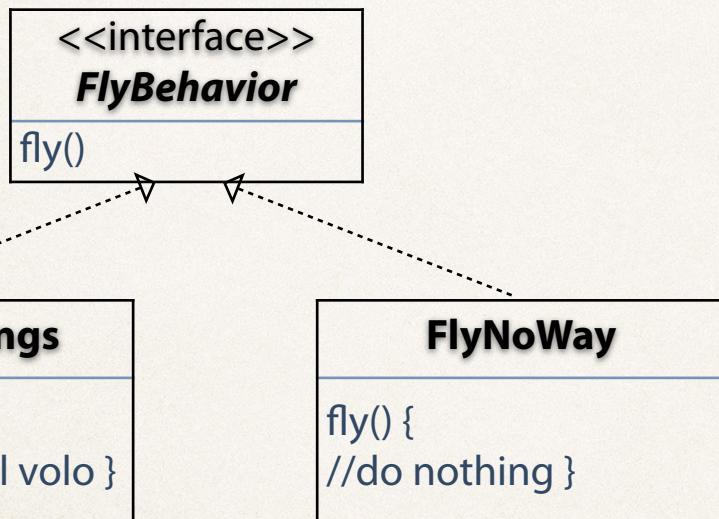
Facciamo il punto: perché l'ereditarietà non basta?

- Visto che il “comportamento” cambia, lo incapsuliamo in un suo oggetto



Facciamo il punto: perché l'ereditarietà non basta?

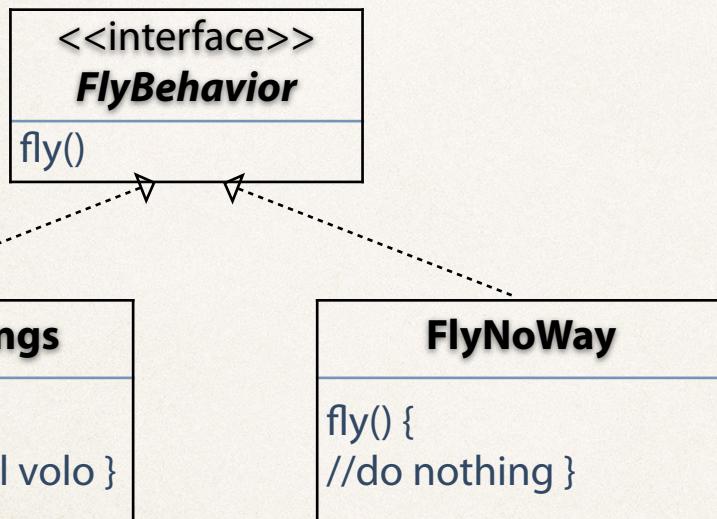
- Visto che il “comportamento” cambia, lo incapsuliamo in un suo oggetto



Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
performQuack()
swim()
display()
performFly()
// ... duck-like methods

Facciamo il punto: perché l'ereditarietà non basta?

- Visto che il “comportamento” cambia, lo incapsuliamo in un suo oggetto

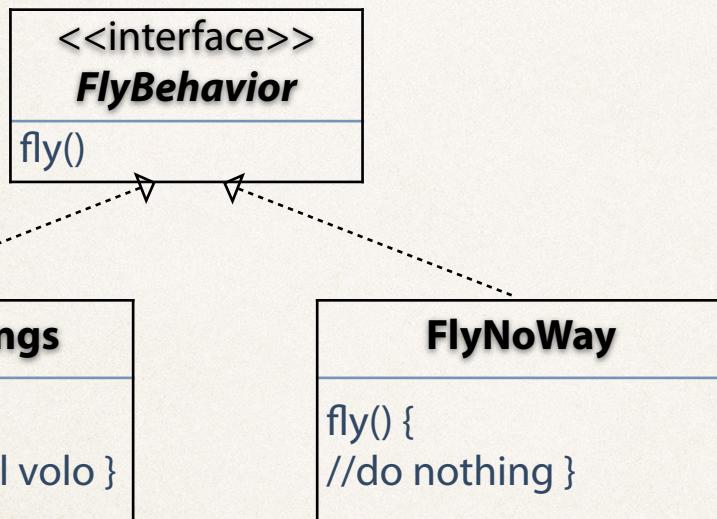


Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
performQuack()
swim()
display()
performFly()
// ... duck-like methods

```
public class Duck{
    FlyBehavior flyBehavior;
    ...
    public void performFly() {
        flyBehavior.fly();
    }
}
```

Facciamo il punto: perché l'ereditarietà non basta?

- Visto che il “comportamento” cambia, lo incapsuliamo in un suo oggetto



Ora la classe ha una referenza a qualcosa che implementa il comportamento: non gestisce il comportamento ma lo delega all'oggetto di cui ha il riferimento

Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
performQuack()
swim()
display()
performFly()
// ... duck-like methods

```
public class Duck{
    FlyBehavior flyBehavior;
    ...
    public void performFly() {
        flyBehavior.fly();
    }
}
```

Facciamo il punto: perché l'ereditarietà non basta?

```
public class GermanoReale extends Duck {  
    public GermanoReale() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
    ...  
}
```

Il constructor del germano reale "sa" quale comportamento deve usare quando viene istanziato

Facciamo il punto: perché l'ereditarietà non basta?

```
public class GermanoReale extends Duck {  
    public GermanoReale() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
    ...  
}
```

Il constructor del germano reale "sa" quale comportamento deve usare quando viene istanziato

Per completare l'opera potrei anche aggiungere alla superclasse Duck un metodo setFlyBehavior per modificare a run time il comportamento

Facciamo il punto: perché l'ereditarietà non basta?

```
public class GermanoReale extends Duck {  
    public GermanoReale() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
    ...  
}
```

Il constructor del germano reale "sa" quale comportamento deve usare quando viene istanziato

Per completare l'opera potrei anche aggiungere alla superclasse Duck un metodo setFlyBehavior per modificare a run time il comportamento

Design
Principle

- program to an interface, not an implementation

Facciamo il punto: perché l'ereditarietà non basta?

Riassumendo:

- abbiamo incapsulato le parti variabili in nuove classi
- abbiamo usato una relazione “has-a” (cioè una composizione) per combinare insieme le varie classi
- **Design principle:** preferite la composizione rispetto all'ereditarietà

Elementi essenziali di un Design Pattern

- ❖ **Nome**
 - ❖ un Design Pattern deve essere identificato in modo univoco per favorire la comunicazione (e conviene che il nome renda l'idea di cosa fa il pattern)
- ❖ **Problema affrontato**
 - ❖ descrizione del contesto in cui è possibile applicare il Pattern
- ❖ **Soluzione**
 - ❖ per la progettazione OO è espressa in termini di classi, responsabilità e collaborazioni
- ❖ **Conseguenze**
 - ❖ risultati dell'applicazione del pattern (pro e contro), da valutare nella scelta tra più alternative
- ❖ **Struttura:** diagramma UML

Come classificarli

- ✿ GoF descrivono i Design Pattern classificandoli in base a due caratteristiche
 - ✿ **purpose:** campo di applicazione del pattern (Pattern di Creazione, Strutturali e Comportamentali)
 - ✿ **scope:** indica se il pattern specifica relazioni tra classi oppure tra oggetti

N.B.: quando si parla di Design Pattern, “abstract class” e “interfaccia” spesso sono la stessa cosa

Classificazione

Defer object creation to another class/object

Describe ways to assemble

Describe algorithms and flow control

	Purpose		
	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Scope	Abstract Factory	Adapter	Chain of Responsibility
	Builder	Bridge	Command
	Prototype	Composite	Iterator
	Singleton	Decorator	Mediator
		Facade	Memento
Object		Prosy	Flyweight
			Observer
			State
			Strategy
			Visitor

Classificazione

Strutturali

- si concentrano sul come classi e oggetti sono combinati per formare strutture più grandi
- pattern strutturali *di classi*, usano ereditarietà per comporre interfacce o implementazioni
- pattern strutturali *di oggetti* descrivono modi di comporre oggetti per realizzare nuove funzionalità

Classificazione

Behavioral

- ✿ pattern che identificano metodi di comunicazione comuni tra oggetti e li realizzano
- ✿ aumentano la flessibilità implementando la comunicazione
- ✿ sono legati all'interazione e alla responsabilità

Classificazione

Creational

- ❖ astraggono il processo di istanziazione (creazione)
- ❖ aiutano a rendere il sistema indipendente da come gli oggetti sono creati, composti e rappresentati (riduce accoppiamento e aumenta flessibilità)
- ❖ nascondono (information hiding) la conoscenza di quali sono le classi concrete effettivamente istanziate
- ❖ nascondono dettagli sulla creazione e sulla composizione degli oggetti (p.e. parametri usati al momento della creazione)

Singleton

Singleton

Il problema

- ❖ talvolta è necessario garantire che una classe abbia una unica istanza, accessibile attraverso un unico punto di accesso
 - ❖ p.e. quando la classe mantiene informazioni di stato che devono essere condivise da più parti del programma e non è corretto / efficiente che queste informazioni siano duplicate
- ❖ passare attraverso `new` non va bene

Singleton

Il problema

- ❖ talvolta è necessario garantire che una classe abbia una unica istanza, accessibile attraverso un unico punto di accesso
 - ❖ p.e. quando la classe mantiene informazioni di stato che devono essere condivise da più parti del programma e non è corretto / efficiente che queste informazioni siano duplicate
- ❖ passare attraverso `new` non va bene

... ne rimarrà solamente uno!



Singleton

```
public MyClass {  
    private MyClass() {}  
}
```

Singleton

```
public MyClass {  
    private MyClass() {}  
}
```

hmmm.. la sintassi è legale, ma non ha
molto senso: il constructor è privato
quindi non riesco a istanziare la classe

Singleton

```
public MyClass {  
    private MyClass() {}  
}
```

hmmm.. la sintassi è legale, ma non ha
molto senso: il constructor è privato
quindi non riesco a istanziare la classe

```
public MyClass {  
    public static MyClass  
        getInstance() {}  
}
```

Singleton

```
public MyClass {  
    private MyClass() {}  
}
```

hmmm.. la sintassi è legale, ma non ha molto senso: il constructor è privato quindi non riesco a istanziare la classe

```
public MyClass {  
    public static MyClass  
        getInstance() {}  
}
```

hmmm.. visto che è un metodo statico, posso fare `MyClass.getInstance()`

Singleton

```
public MyClass {  
    private MyClass() {}  
}
```

hmmm.. la sintassi è legale, ma non ha molto senso: il constructor è privato quindi non riesco a istanziare la classe

```
public MyClass {  
    public static MyClass  
        getInstance() {}  
}
```

hmmm.. visto che è un metodo statico, posso fare `MyClass.getInstance()`

```
public MyClass {  
    private MyClass() {}  
    public static MyClass  
        getInstance() {  
            return new MyClass();  
        }  
}
```

Singleton

```
public MyClass {  
    private MyClass() {}  
}
```

hmmm.. la sintassi è legale, ma non ha molto senso: il constructor è privato quindi non riesco a istanziare la classe

```
public MyClass {  
    public static MyClass  
        getInstance() {}  
}
```

hmmm.. visto che è un metodo statico, posso fare `MyClass.getInstance()`

```
public MyClass {  
    private MyClass() {}  
    public static MyClass  
        getInstance() {  
            return new MyClass();  
        }  
}
```

Ahaaa! `MyClass.getInstance()` mi instanzia l'oggetto. Ora devo fare in modo di poterne creare solo uno...

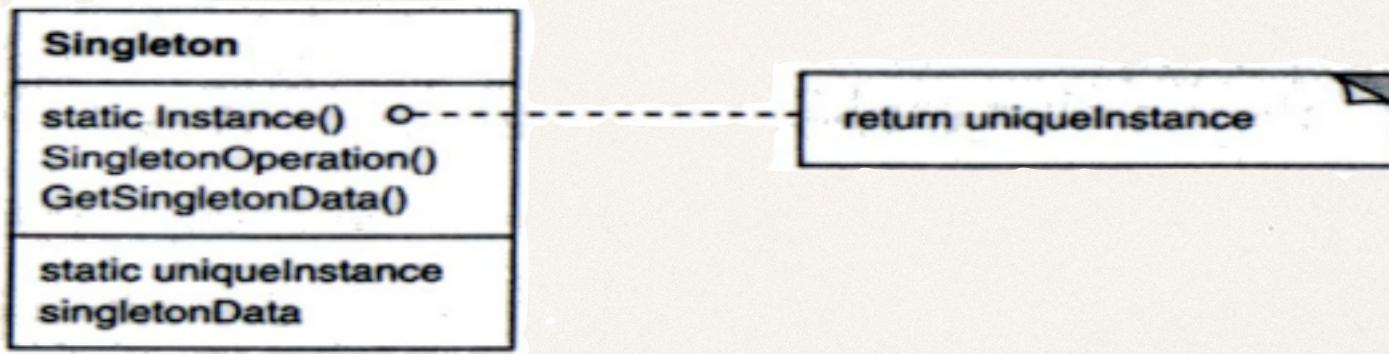
Singleton

Soluzione

- ❖ si rende il costruttore della classe privato, in modo che non sia possibile creare direttamente istanze (al di fuori del codice della classe)
- ❖ si fornisce un metodo “static” per ottenere l'unica istanza, che viene conservata in un campo static privato della classe
- ❖ varianti
 - ❖ l'istanza può essere creata all'inizializzazione del programma oppure la prima volta che viene richiesta
 - ❖ l'istanza, se necessario, può appartenere a una sottoclassificazione della classe singleton (vedi poi Factory Method)

Singleton

- Esempio di struttura



Singleton

Conseguenze

- ⊕ accesso controllato all'unica istanza
- ⊕ non occorre usare variabili globali per accedere all'unica istanza
- ⊕ è semplice estendere (mediante subclassing) la classe singleton senza modificare il codice che la usa
- ⊕ se serve, è semplice passare da una singola istanza a un numero diverso di istanze

Singleton

- Esempio di implementazione

- N.B.
- Java ≥ 5
 - Thread safe

```
import java.util.*;
public class Singleton {
    private volatile static Singleton uniqueInstance;
    // altri dati importanti
    private Singleton() {}

    public static Singleton getInstance() {
        if(uniqueInstance==null) {
            synchronized(Singleton.class){
                if(uniqueInstance==null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

Factory Method

... program to an interface, not to a implementation ...

- ❖ quando usiamo new stiamo certamente istanziando una classe concreta, quindi una implementazione e non una interfaccia
- ❖ quando abbiamo un insieme di classi concrete spesso capita questo
- ❖ abbiamo varie possibili classi da cui decidiamo a tempo di esecuzione quale vogliamo istanziare
- ❖ in una situazione simile sappiamo già che in caso di modifiche dobbiamo riesaminare tutto il codice

```
Duck duck;  
if (picnic) {  
    duck=new MallardDuck();  
} else if (hunting) {  
    duck=new DecoyDuck();  
} else if (in athTub) {  
    duck=new RubberDuck()  
}
```

NON SIAMO "CLOSED FOR MODIFICATIONS"

Factory Method

Il problema

- una classe ha bisogno di creare un oggetto (“prodotto”) che implementa una interfaccia, ma vuole evitare che dipenda da una specifica implementazione concreta tra quelle disponibili
- oppure, una classe vuole delegare alle sottoclassi la creazione di determinati oggetti

Chi deve essere responsabile di creare gli oggetti quando la logica di creazione è complessa e si vuole separare la logica di creazione per avere maggiore coesione?
Inventiamoci un oggetto chiamato Factory che gestisce la creazione...

Factory Method

Es.: negozio che vende pizze..

```
Pizza orderPizza(){
    Pizza pizza = new Pizza();

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Factory Method

ma ci sono tanti tipi di pizza...

Es.: negozio che vende pizze..

```
Pizza orderPizza(){  
    Pizza pizza = new Pizza();  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

```
Pizza orderPizza(String type){  
    Pizza pizza;  
    if(type.equals("marin")) {  
        pizza=new MarinaraPizza();  
    } else if(type.equals("marghe")) {  
        pizza=new MargheritaPizza();  
    } else if...  
    }  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Factory Method

ma ci sono tanti tipi di pizza...

Es.: negozio che vende pizze..

```
Pizza orderPizza(){  
    Pizza pizza = new Pizza();  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Per essere flessibili, vogliamo che sia una classe astratta o una interfaccia, ma non possiamo instanziare direttamente nessuna di queste

```
Pizza orderPizza(String type){  
    Pizza pizza;  
    if(type.equals("marin")) {  
        pizza=new MarinaraPizza();  
    } else if(type.equals("marghe")) {  
        pizza=new MargheritaPizza();  
    } else if...  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;
```

Factory Method

Rimandiamo la decisione al momento dell'esecuzione

ma ci sono tanti tipi di pizza...

Es.: negozio che vende pizze..

```
Pizza orderPizza(){  
    Pizza pizza = new Pizza();  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Per essere flessibili, vogliamo che sia una classe astratta o una interfaccia, ma non possiamo instanziare direttamente nessuna di queste

```
Pizza orderPizza(String type){  
    Pizza pizza;  
    if(type.equals("marin")) {  
        pizza=new MarinaraPizza();  
    } else if(type.equals("marghe")) {  
        pizza=new MargheritaPizza();  
    } else if...  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;
```

Factory Method

Rimandiamo la decisione al momento dell'esecuzione

ma ci sono tanti tipi di pizza...

Es.: negozio che vende pizze..

```
Pizza orderPizza(){  
    Pizza pizza = new Pizza();  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Per essere flessibili, vogliamo che sia una classe astratta o una interfaccia, ma non possiamo instanziare direttamente nessuna di queste

```
Pizza orderPizza(String type){  
    Pizza pizza;  
    if(type.equals("marin")) {  
        pizza=new MarinaraPizza();  
    } else if(type.equals("marghe")) {  
        pizza=new MargheritaPizza();  
    } else if...  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Serve istanziare la classe concreta giusta. E ogni tipo di pizza deve implementare l'interfaccia di Pizza

Factory Method

Per star dietro alla
concorrenza bisogna
aggiungere nuovi tipi di
pizza....

Factory Method

Per star dietro alla concorrenza bisogna aggiungere nuovi tipi di pizza....

```
Pizza orderPizza(String type){  
    Pizza pizza;  
    if(type.equals("marin")) {  
        pizza=new MarinaraPizza();  
    } else if(type.equals("marghe")) {  
        pizza=new MargheritaPizza();  
    } else if(type.equals("vegetariana")) {  
        pizza=new VegetarianaPizza();  
    } else if...  
    }  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Factory Method

Per star dietro alla concorrenza bisogna aggiungere nuovi tipi di pizza....

```
Pizza orderPizza(String type){  
    Pizza pizza;  
    if(type.equals("marin")) {  
        pizza=new MarinaraPizza();  
    } else if(type.equals("marghe")) {  
        pizza=new MargheritaPizza();  
    } else if(type.equals("vegetariana")) {  
        pizza=new VegetarianaPizza();  
    } else if...  
    }  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Questo parte cambia ogni volta

Factory Method

Per star dietro alla concorrenza bisogna aggiungere nuovi tipi di pizza....

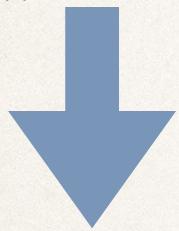
```
Pizza orderPizza(String type){  
    Pizza pizza;  
    if(type.equals("marin")) {  
        pizza=new MarinaraPizza();  
    } else if(type.equals("marghe")) {  
        pizza=new MargheritaPizza();  
    } else if(type.equals("vegetariana")) {  
        pizza=new VegetarianaPizza();  
    } else if...  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Questo parte cambia ogni volta

Questo parte non cambia ogni volta

Factory Method

Per star dietro alla concorrenza bisogna aggiungere nuovi tipi di pizza....



Il codice dovrebbe essere chiuso per le modifiche, e aperto per le estensioni...
Togliamo la parte variabile e incapsuliamola in un altro oggetto:
SimplePizzaFactory!

```
Pizza orderPizza(String type){  
    Pizza pizza;  
    if(type.equals("marin")) {  
        pizza=new MarinaraPizza();  
    } else if(type.equals("marghe")) {  
        pizza=new MargheritaPizza();  
    } else if(type.equals("vegetariana")) {  
        pizza=new VegetarianaPizza();  
    } else if...  
}  
  
pizza.prepare();  
pizza.bake();  
pizza.cut();  
pizza.box();  
return pizza;
```

Questo parte cambia ogni volta

Questo parte non cambia ogni volta

Factory Method

Soluzione

- ❖ si incapsula la creazione degli oggetti in un metodo
- ❖ varianti
 - ❖ il metodo può essere nella stessa classe che deve usare gli oggetti oppure essere un metodo “static” di una classe diversa
 - ❖ il metodo può essere “abstract” e quindi richiedere che una sottoclassifiche definisca l’implementazione, oppure essere concreto e fornire una implementazione di default (ad esempio basato su configurazione esterna)

Factory Method

```
public class SimplePizzaFactory {  
    public Pizza createPizza(String type) {  
        Pizza pizza=null;  
  
        if(type.equals("marinara")) {  
            pizza = new MarinaraPizza();  
        } else if(type.equals("margherita")) {  
            pizza = new MargheritaPizza();  
        } else if(type.equals("vegetariana")) {  
            pizza = new VegetarianaPizza();  
        } else if...  
        }  
        return pizza;  
    }  
}
```

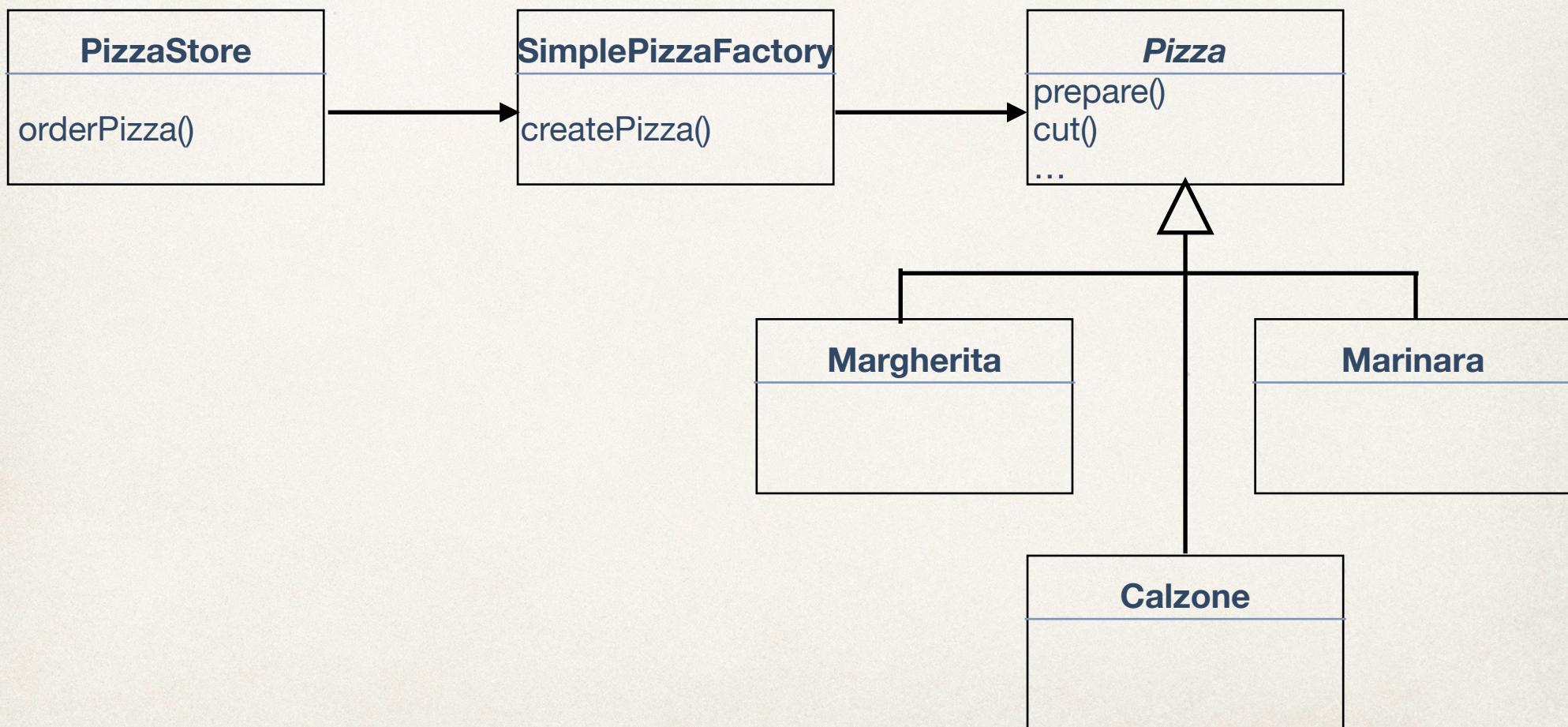
Questo è il metodo che tutti i clients useranno per istanziare una nuova pizza

Factory Method

Ripuliamo il codice.
Notate che abbiamo fatto
sparire l'operatore new:
non c'è più una istanzia-
zione concreta. Al suo
posto c'è il metodo
create di un oggetto
factory

```
public class PizzaStore {  
    SimplePizzaFactory factory;  
  
    public pizzaStore(SimplePizzaFactory factory){  
        this.factory = factory;  
    }  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza;  
  
        pizza = factory.createPizza(type);  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza;box();  
        return pizza;  
    }  
}
```

Factory Method



Factory Method

- ❖ Abbiamo avuto successo e tutti vogliono una nostra pizza: ci espandiamo in altre città!
- ❖ Però dobbiamo adattarci
 - ▶ a Roma vogliono la pizza molto sottile
 - ▶ a Palermo la vogliono sottile e croccante
 - ▶ a Bari vorrebbero anche la scamorza come formaggio
 - ▶
- ❖ come fare?

Factory Method

Umm.... potremmo togliere SimplePizzaFactory e creare diverse factory. Poi componiamo i PizzaStore con le factory opportune.
Verrebbe una cosa così

```
RomaPizzaFactory romaFactory = new RomaPizzaFactory();  
PizzaStore romaStore = new PizzaStore(romaFactory);  
romaStore.order("margherita");
```

```
BariPizzaFactory bariFactory = new BariPizzaFactory();  
PizzaStore bariStore = new PizzaStore(bariFactory);  
bariStore.order("margherita");
```

Factory Method

- ❖ Tutto a posto quindi?....
- ❖ Ogni Store fa le cose a modo suo: qualcuno non taglia la pizza, oppure usano cartoni diversi, o comunque fanno le cose in modo diverso
- ❖ Noi vorremmo che il codice che “fa la pizza” fosse legato al PizzaStore, e che fosse sempre lo stesso, pur facendo pizze in modo diverso.
- ❖ Insomma, vorremmo rimettere createPizza() dentro a PizzaStore, ma rimanere flessibili....
- ❖ abstract method???

Factory Method

Proviamo a sfruttare il fatto che un metodo astratto deve essere implementato da ogni classe derivata

```
public abstract class PizzaStore {  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza;  
  
        pizza = createPizza(type);  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
  
        return pizza;  
    }  
  
    abstract createPizza(String type);  
}
```

L'oggetto factory ora si trova qui

Factory Method

```
public class RomaPizzaStore extends PizzaStore {  
    Pizza createPizza(String item) {  
        if(item.equals("margherita")) {  
            return new RomaStyleMargheritaPizza();  
        } else if(item.equals("marinara")) {  
            return new RomaStyleMarinaraPizza();  
        } else ...  
    }  
}
```

- estendiamo PizzaStore, per cui stiamo ereditando orderPizza() (e non solo)
 - dobbiamo implementare createPizza() perchè in PizzaStore è astratto
 - creiamo le classi concrete, Roma style
- N.B.: il metodo orderPizza() della superclasse non ha idea del tipo di pizza che sarà creata; sa solo come cuocerla, tagliarla e metterla nel cartone

Factory Method

- Un “factory method” gestisce la creazione di oggetti e la incapsula in una sottoclasse
- Questo disaccoppia il codice della superclasse dalla creazione dell’oggetto nella classe derivata.

abstract Product factoryMethod(String type)

Un factory method è astratto:
le classi derivate gestiscono
la creazione dell’oggetto

Isola il client (il codice della
superclasse, es: orderPizza())
dal sapere quale tipo di
concrete Product viene creato

Ritorna un Product che poi
viene usato nei metodi
definiti nella superclasse

Un factory method può
essere parametrizzato
(oppure no) per
scegliere tra i diversi tipi
di un Product

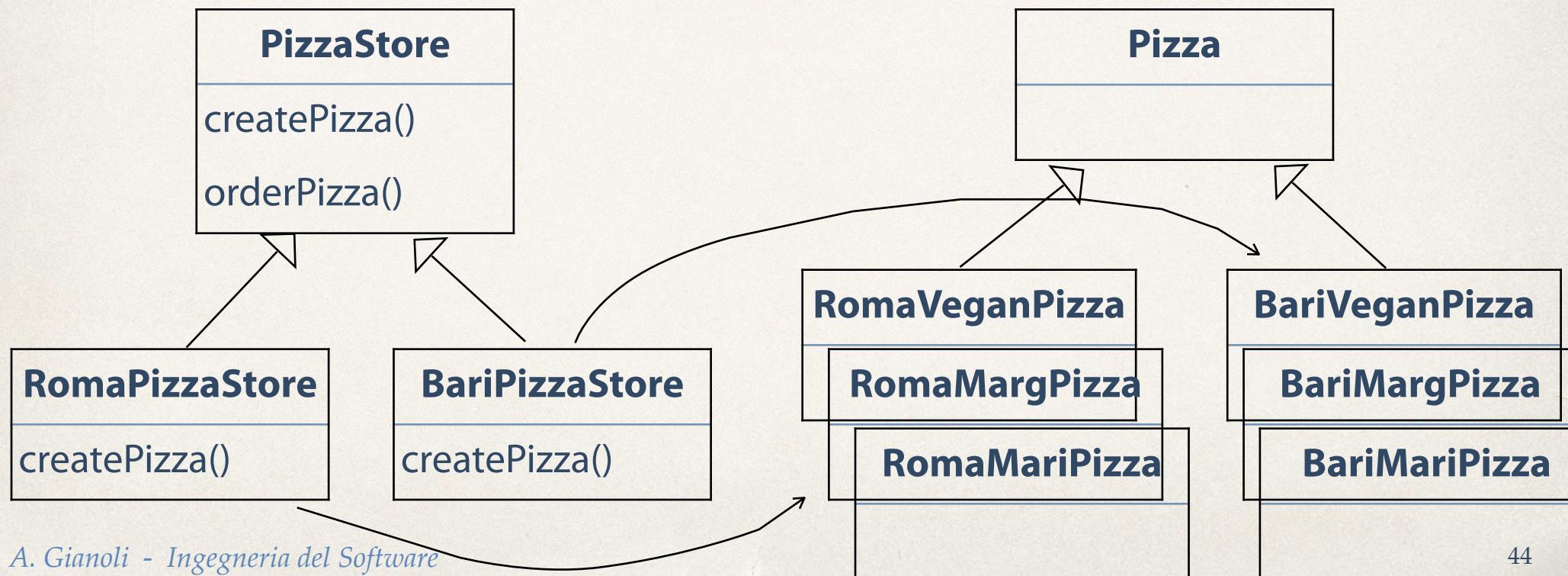
Factory Method

Creator classes

- ❖ abstract creator
- ❖ concrete creators

Product classes

- ❖ abstract general product
- ❖ concrete products



Factory Method

comparate le due implementazioni

Dependent Pizza Store

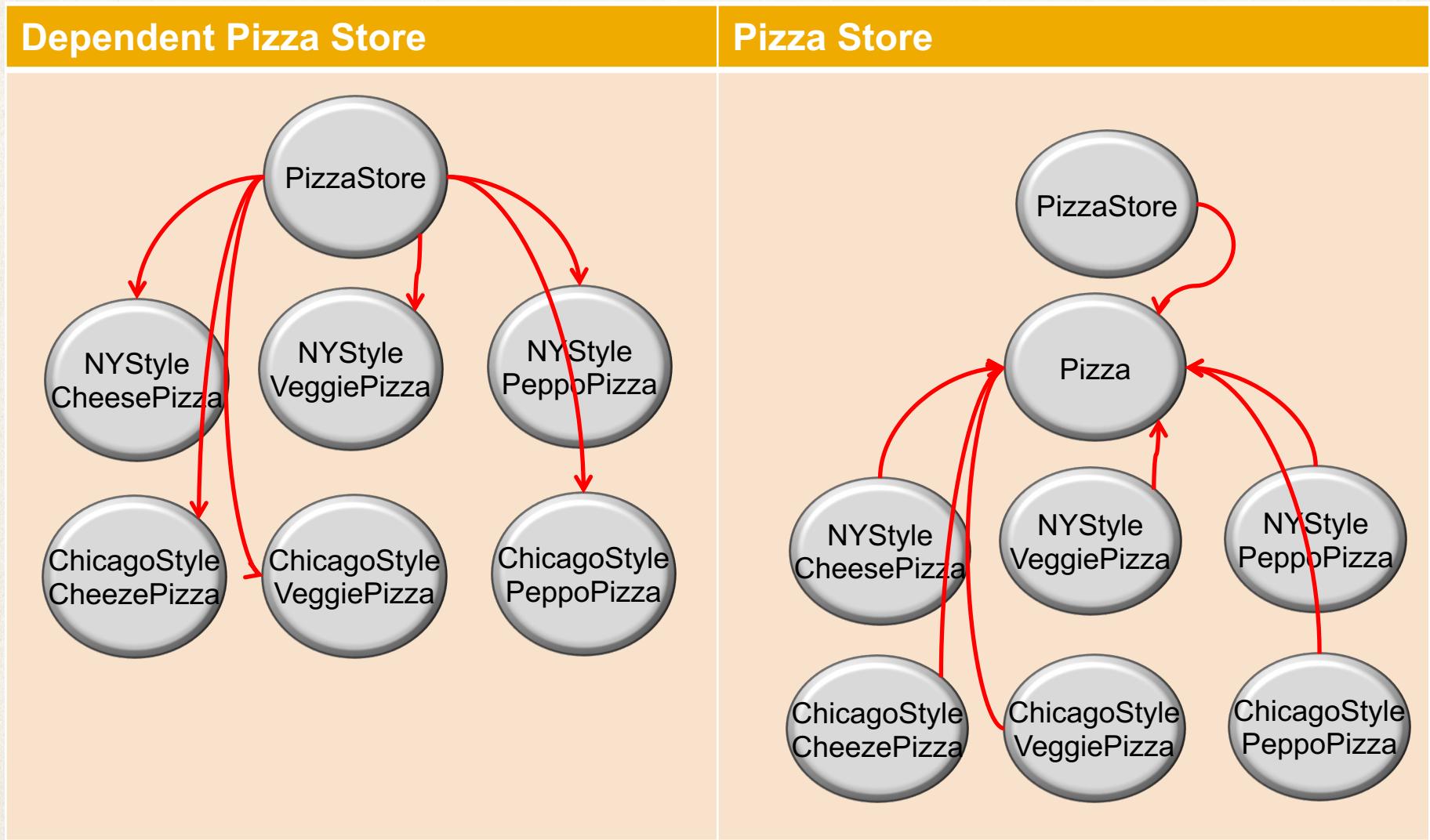
```
public class DependentPizzaStore {  
    public Pizza createPizza(String style, String type) {  
        Pizza pizza = null;  
        if (style.equals("NY")) {  
            if (type.equals("cheese")) {  
                pizza = new NYStyleCheesePizza();  
            } else if (type.equals("veggie")) {  
                pizza = new NYStyleVeggiePizza();  
            } else if (type.equals("pepperoni")) {  
                pizza = new NYStylePepperoniPizza();  
            }  
        } else if (style.equals("Chicago")) {  
            if (type.equals("cheese")) {  
                pizza = new ChicagoStyleCheesePizza();  
            } else if (type.equals("veggie")) {  
                pizza = new ChicagoStyleVeggiePizza();  
            } else if (type.equals("pepperoni")) {  
                pizza = new ChicagoStylePepperoniPizza();  
            }  
        } else {  
            System.out.println("Error: invalid type of pizza");  
            return null;  
        }  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
}
```

Pizza Store

```
public abstract class PizzaStore {  
    abstract Pizza createPizza(String item);  
    public Pizza orderPizza(String type) {  
        Pizza pizza = createPizza(type);  
        System.out.println("--- Making a " + pizza.getName() + " ---");  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
  
    public class NYPizzaStore extends PizzaStore {  
        Pizza createPizza(String item) {  
            if (item.equals("cheese")) {  
                return new NYStyleCheesePizza();  
            } else if (item.equals("veggie")) {  
                return new NYStyleVeggiePizza();  
            } else if (item.equals("clam")) {  
                return new NYStyleClamPizza();  
            } else if (item.equals("pepperoni")) {  
                return new NYStylePepperoniPizza();  
            } else return null;  
        }  
    }  
    ...  
}
```

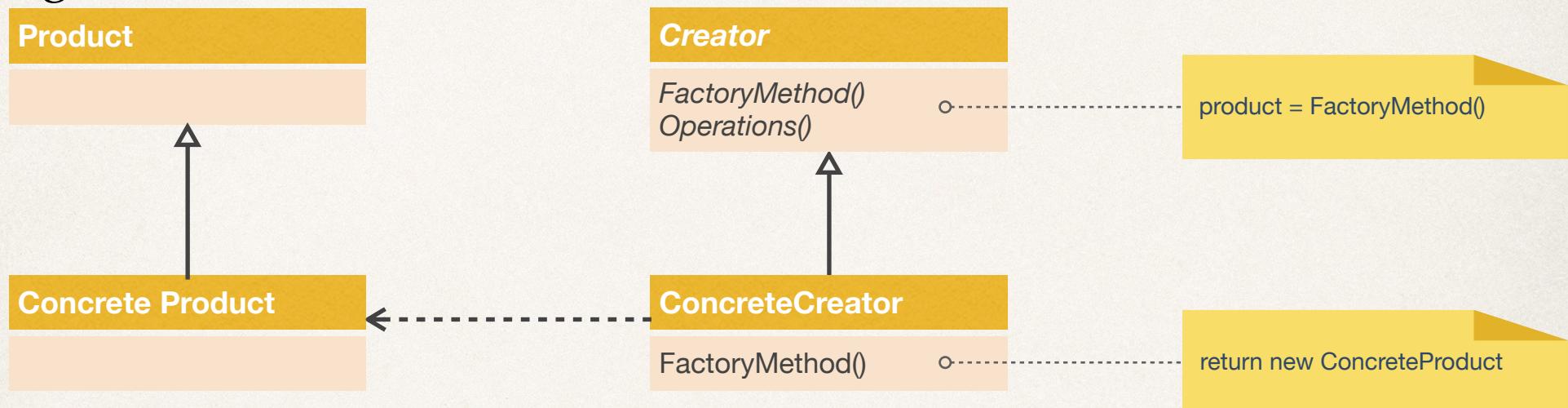
Factory Method

comparate le dipendenze



Factory Method

⊕ Diagramma UML



Creator

- Contains methods to manipulate products, since the steps and procedures are generic
- Creation of product is abstracted

Concrete Creator

- Implements concrete product creation
- Each Concrete Creator knows its respective concrete product

Concrete Products

- Implements abstract Product
- Contains specific implementations for the abstract methods/step declared in the Product Interface

Abstract Factory

Voglio creare diverse famiglie di classi tra loro correlate, nascondere le loro classi concrete, assicurare creazione e gestione di famiglie consistenti di oggetti

Il problema

- una classe ha bisogno di creare una serie di oggetti (prodotti) che implementano delle interfacce correlate, ma si vuole evitare che dipenda da una specifica implementazione concreta tra quelle disponibili
- il sistema deve essere configurato con famiglie multiple di prodotti, dove una famiglia è progettata per essere usata nel suo insieme
- si vuole distribuire una libreria di prodotti rivelando solo le interfacce e non le classi concrete che le implementano

Abstract Factory

Soluzione

- ❖ si definisce una interfaccia AbstractFactory con metodi per creare diversi prodotti, e una o più classi concrete che implementano questa interfaccia in riferimento a una singola famiglia di prodotti
- ❖ a run-time si costruisce un'istanza di una “concrete factory” che viene usata per creare i prodotti
 - ❖ N.B.: la creazione di questa istanza può essere a sua volta realizzata con un Factory Method

Abstract Factory

- ❖ Torniamo al nostro business di pizze
- ❖ Consideriamo la pizza come composta da varie parti (pasta, salsa, formaggio, verdure, ...)
- ❖ in ogni città vogliamo usare la versione locale di queste parti

```
public interface PizzaIngredientFactory {  
    public Dough createDough();  
    public Sauce createSauce();  
    public Cheese createCheese();  
    public Veggies[] createVeggies();  
    ...  
}
```

Prima di tutto creiamo una interfaccia per la factory che creerà tutti i nostri ingredienti per una pizza

Abstract Factory

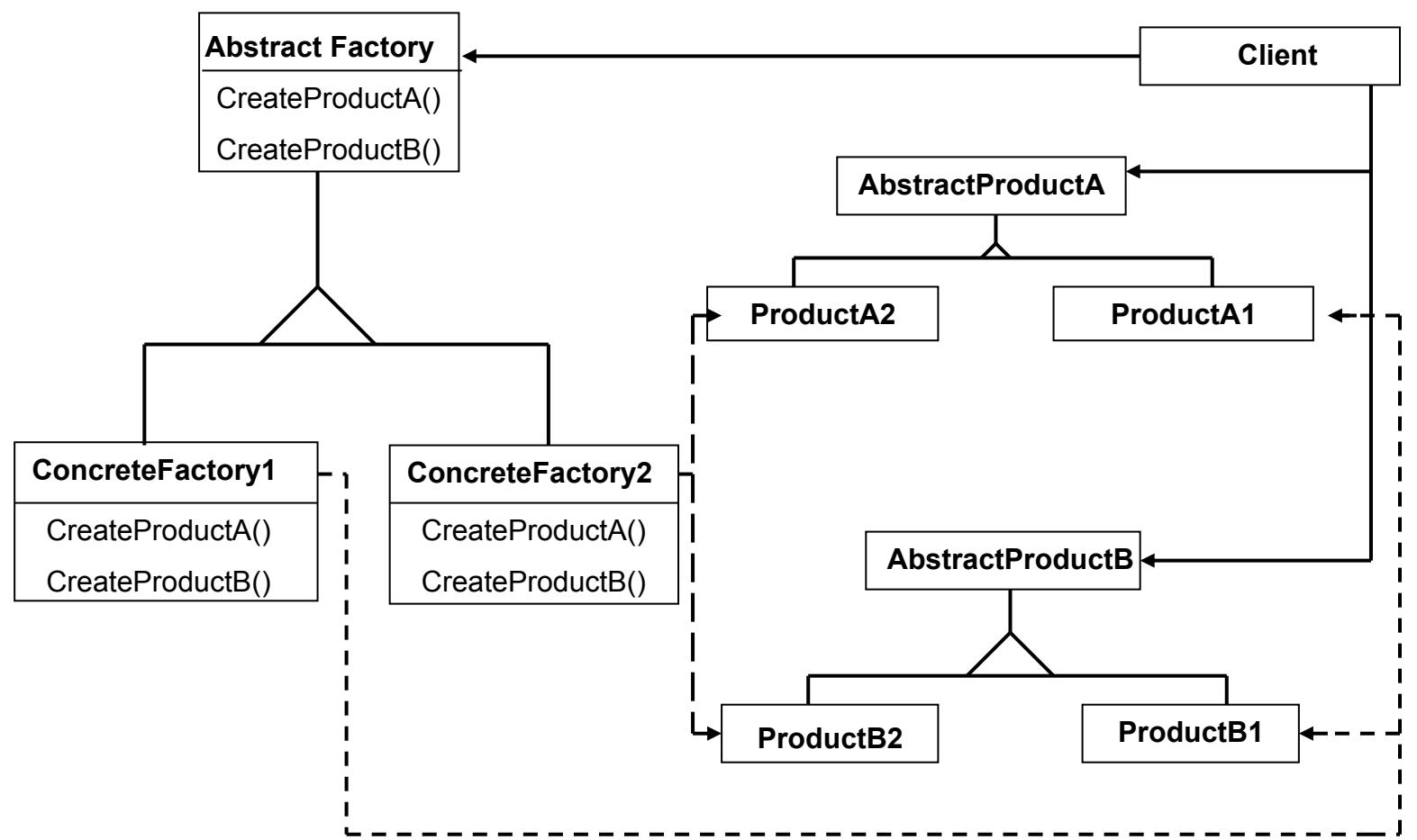
```
public class RomaIngredientFactory
    implements PizzaIngredientFactory {
    public Dough createDough() {
        return new ThinDough();
    }
    public Sauce createSauce() {
        return new FreshSauce();
    }
    public Cheese createCheese() {
        return new MozzarellaCheese();
    }
    public Veggies[] createVeggies() {
        Veggies veggies[ ] = { new Garlic(), new Onion(),
                             new Mushroom() };
        return veggies;
    }
    ...
}
```

Abstract Factory

```
public class RomaPizzaStore extends PizzaStore {  
    protected Pizza createPizza(String item) {  
        Pizza pizza = null;  
        PizzaIngredientFactory ingredientFactory =  
            new RomaIngredientFactory();  
        if(item.equals("marg")) {  
            pizza = new RomaStyleMargeritaPizza(  
                RomaIngredientFactory)  
        }  
        ...  
    }  
}
```

Abstract Factory

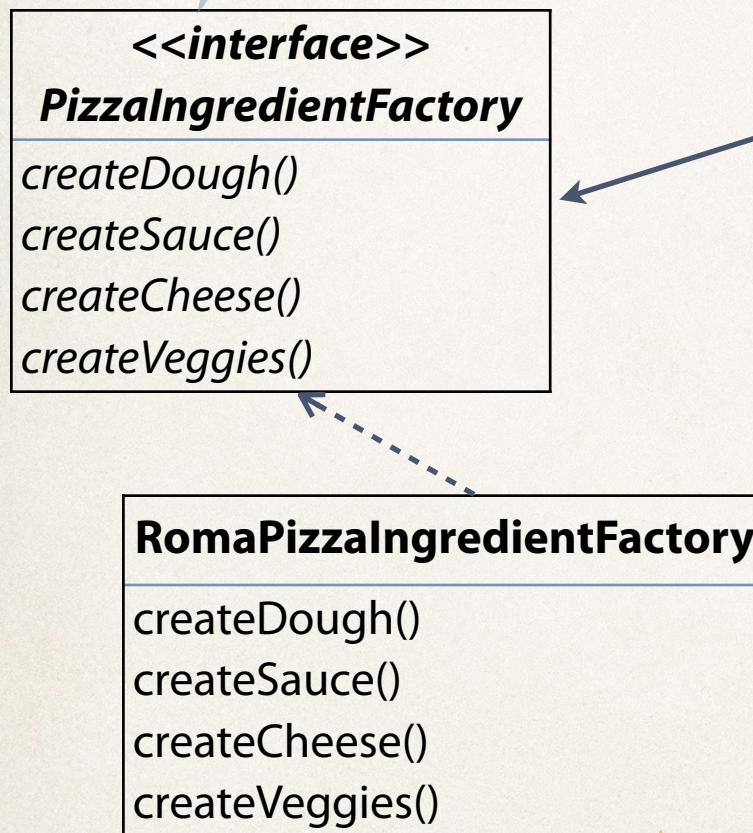
- Esempio di struttura



Abstract Factory

abstract factory: definisce un insieme di prodotti correlati per fare una pizza

NB: sia Abstract Factory che Factory creano oggetti. Factory lo fa tramite inheritance, mentre Abstract Factory lo fa tramite composition!



concrete factory: sa creare il tipo giusto di oggetto per la sua versione

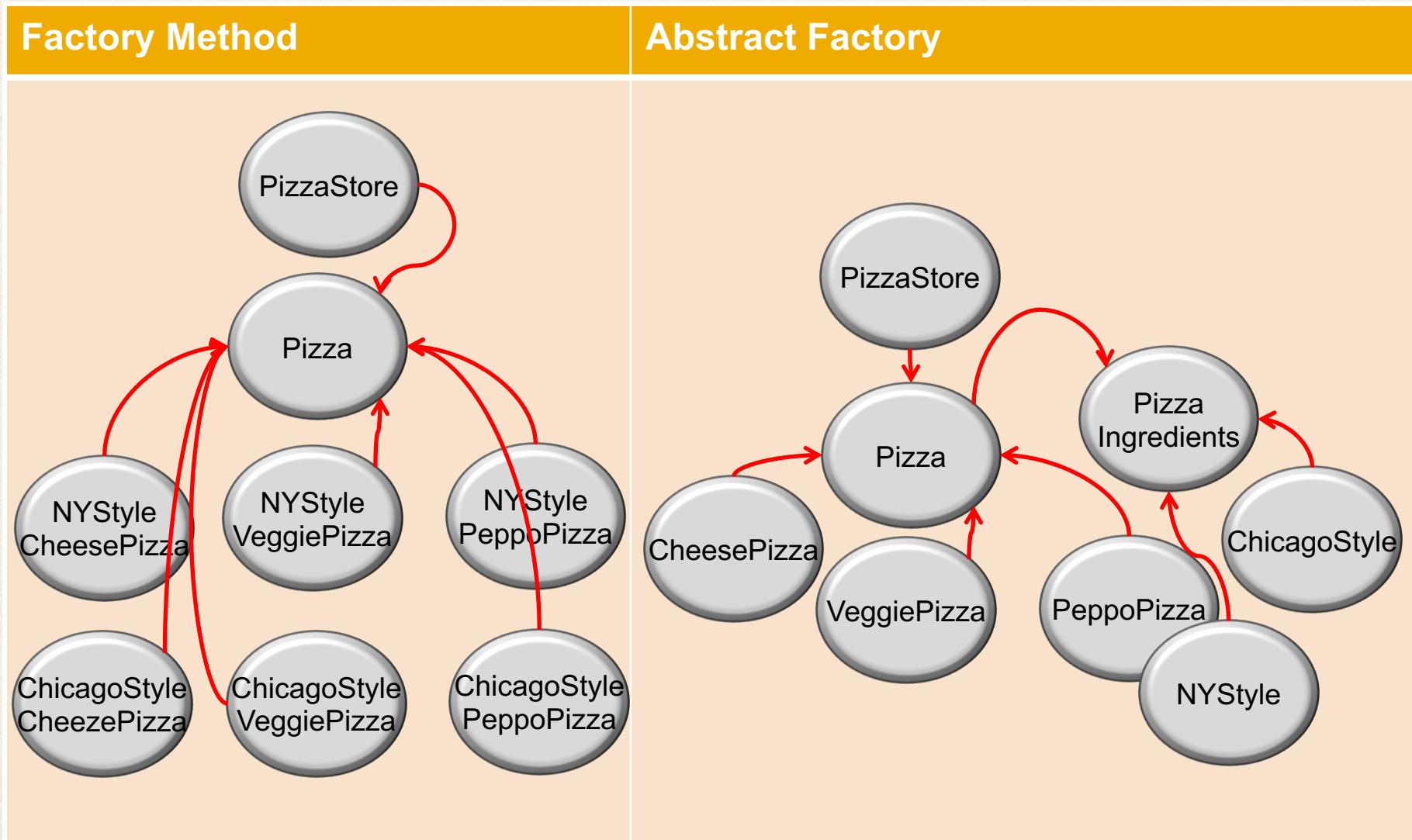
i metodi di una Abstract Factory spesso sono implementati come dei metodi "Factory"

Abstract Factory

Conseguenze

- ⊕ nasconde le classi concrete; solo la factory sa quale classe concreta viene istanziata
- ⊕ consente di sostituire facilmente una famiglia di prodotti con un'altra
- ⊕ garantisce che i prodotti usati insieme siano della stessa famiglia
- ⊕ **PROBLEMA:** rende onerosa l'aggiunta di nuovi prodotti a una famiglia, in quanto bisogna modificare simultaneamente tutte le famiglie e tutte le “concrete factories”

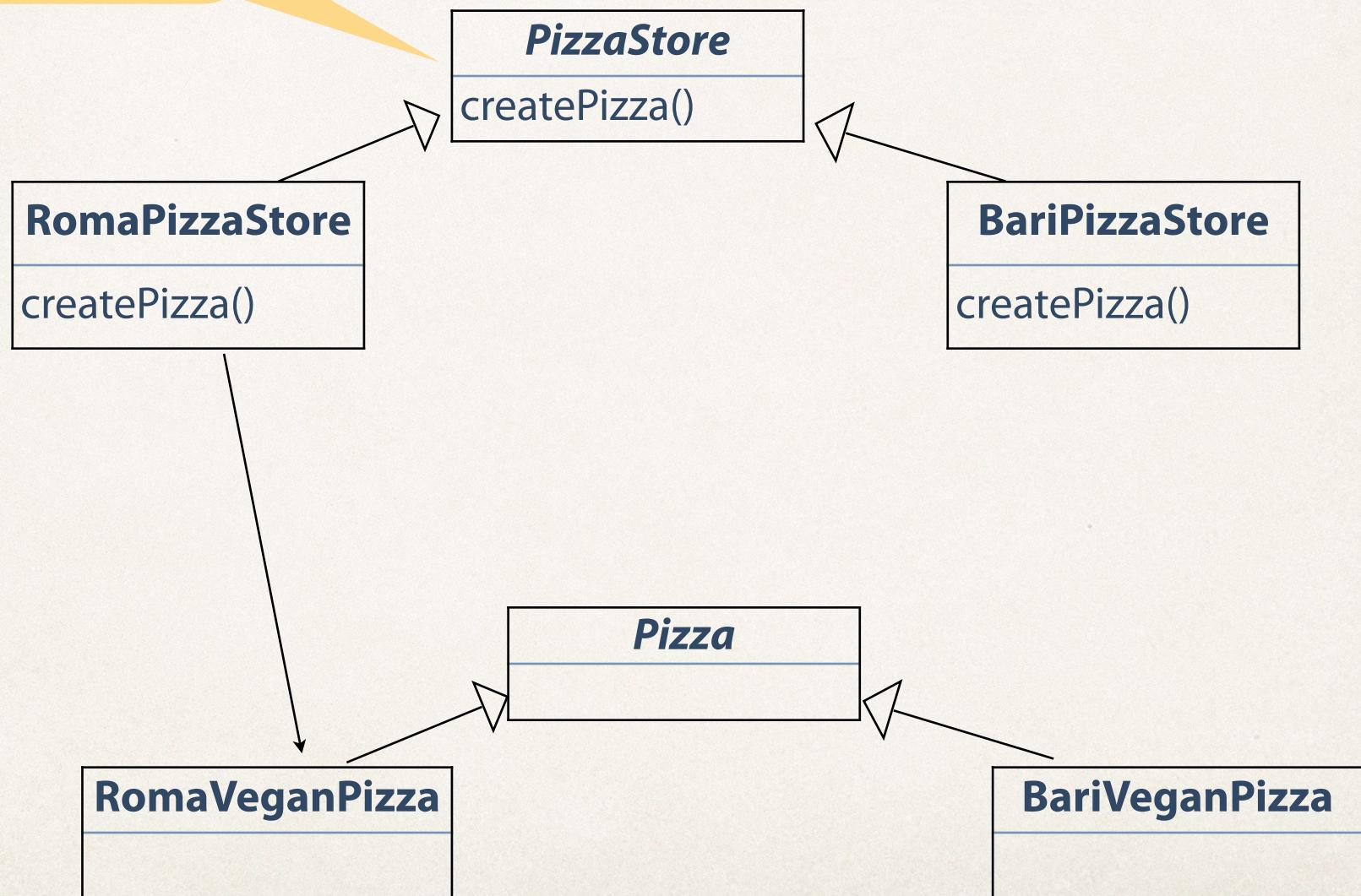
Factory vs Abstract Factory



Factory vs Abstract Factory

Factory method:
vogliamo creare un
prodotto che varia
con la regione.

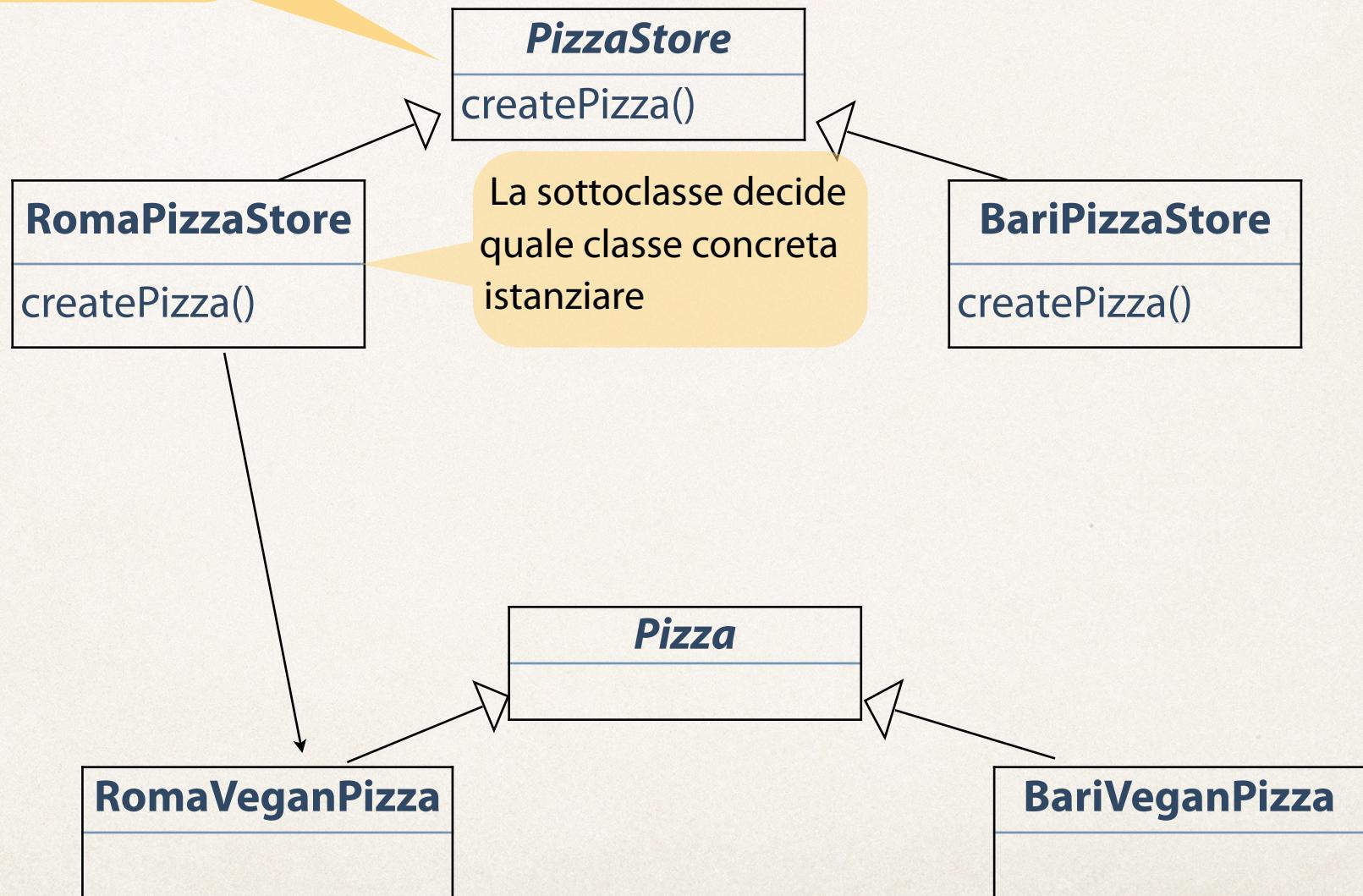
Fornisce l'interfaccia



Factory vs Abstract Factory

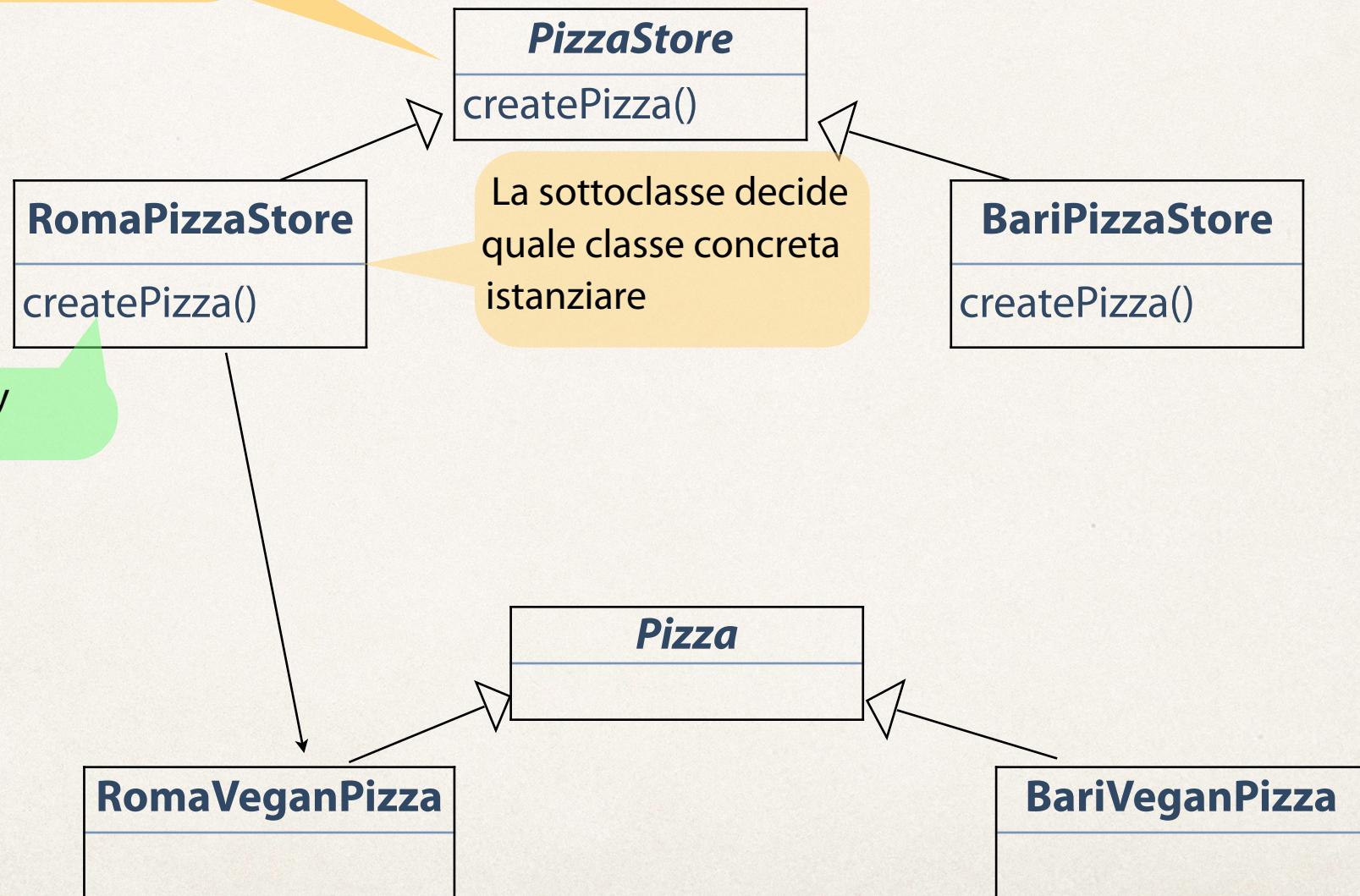
Factory method:
vogliamo creare un
prodotto che varia
con la regione.

Fornisce l'interfaccia



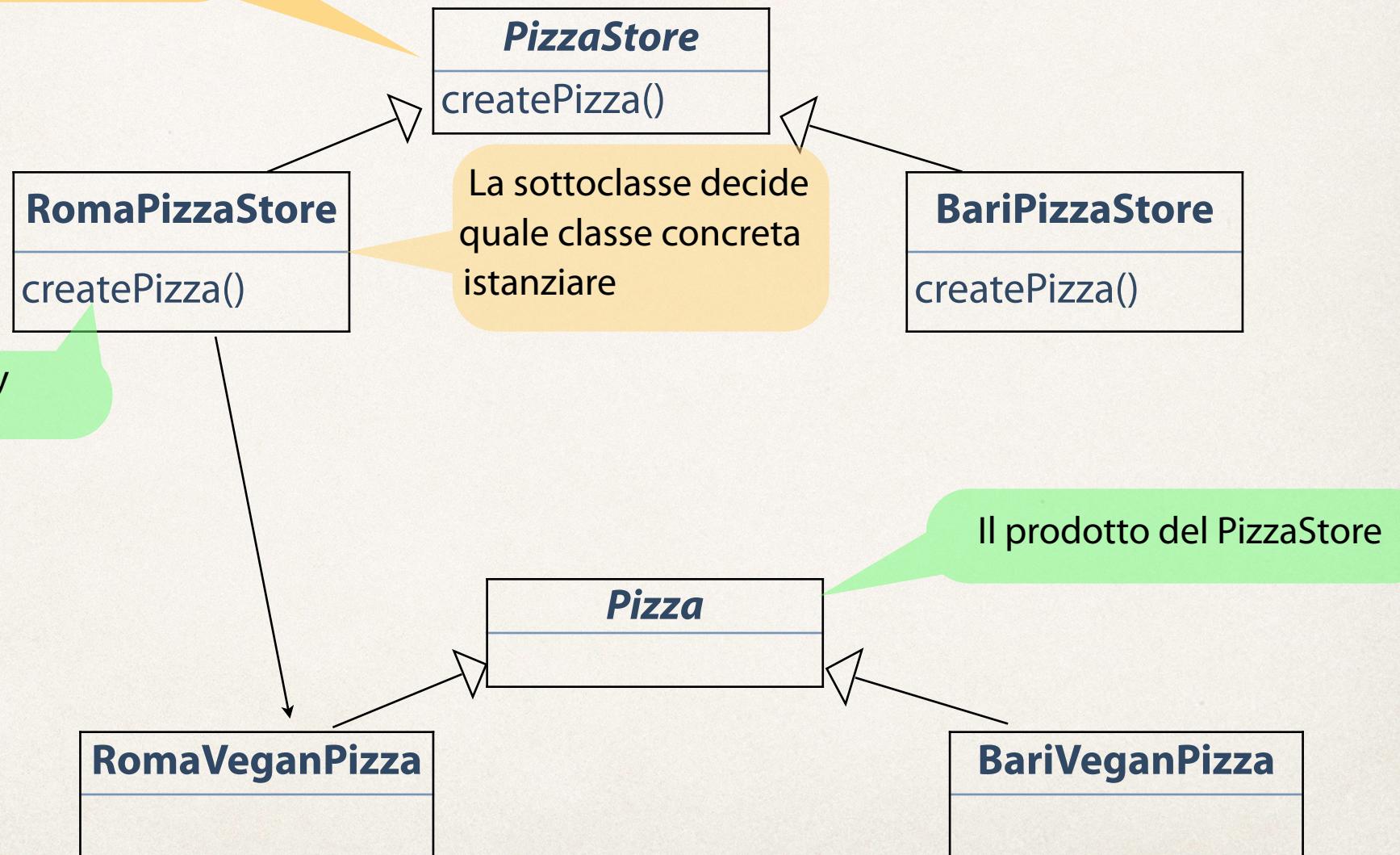
Factory method:
vogliamo creare un
prodotto che varia
con la regione.
Fornisce l'interfaccia

Factory vs Abstract Factory



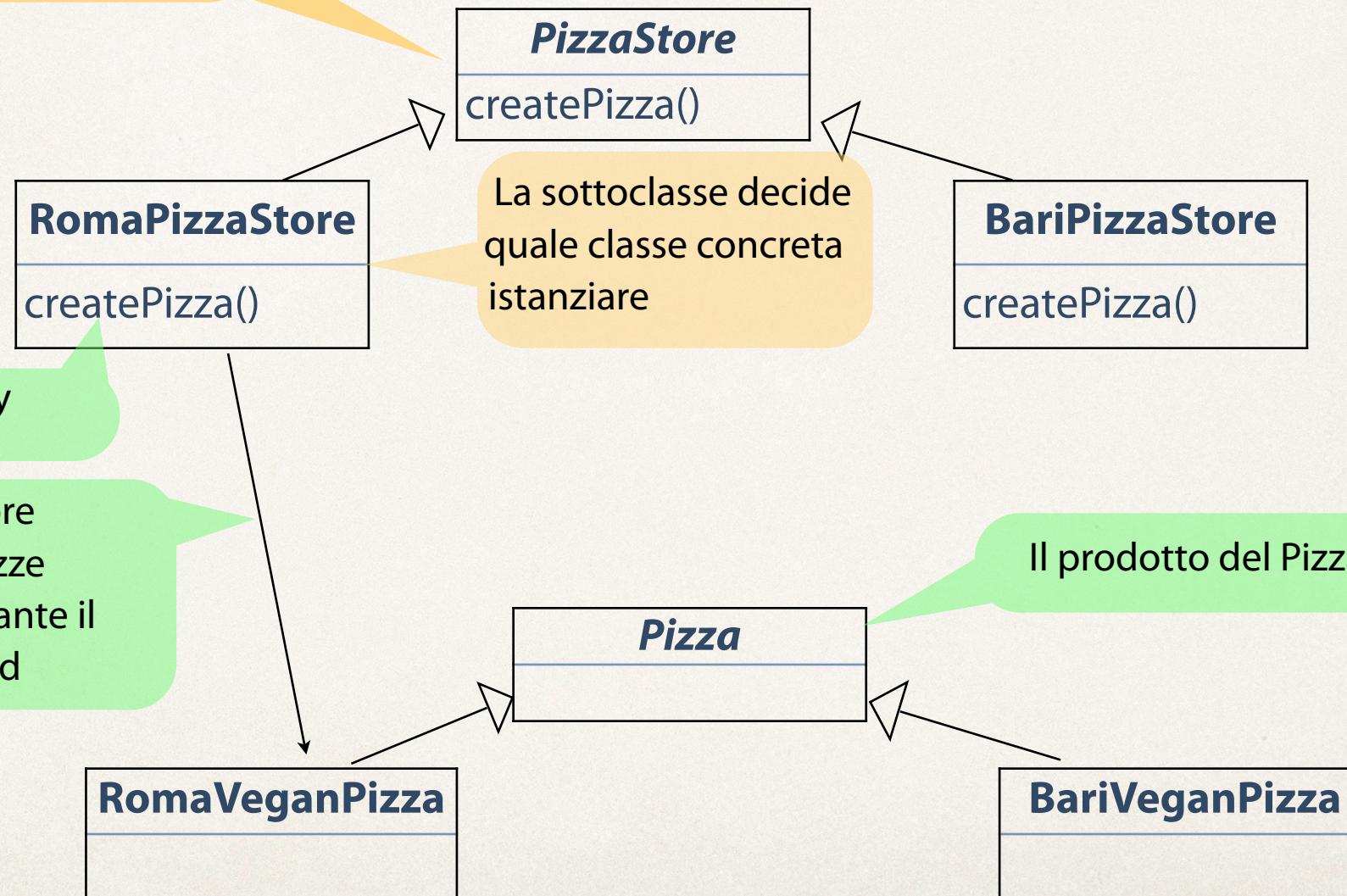
Factory method:
vogliamo creare un
prodotto che varia
con la regione.
Fornisce l'interfaccia

Factory vs Abstract Factory

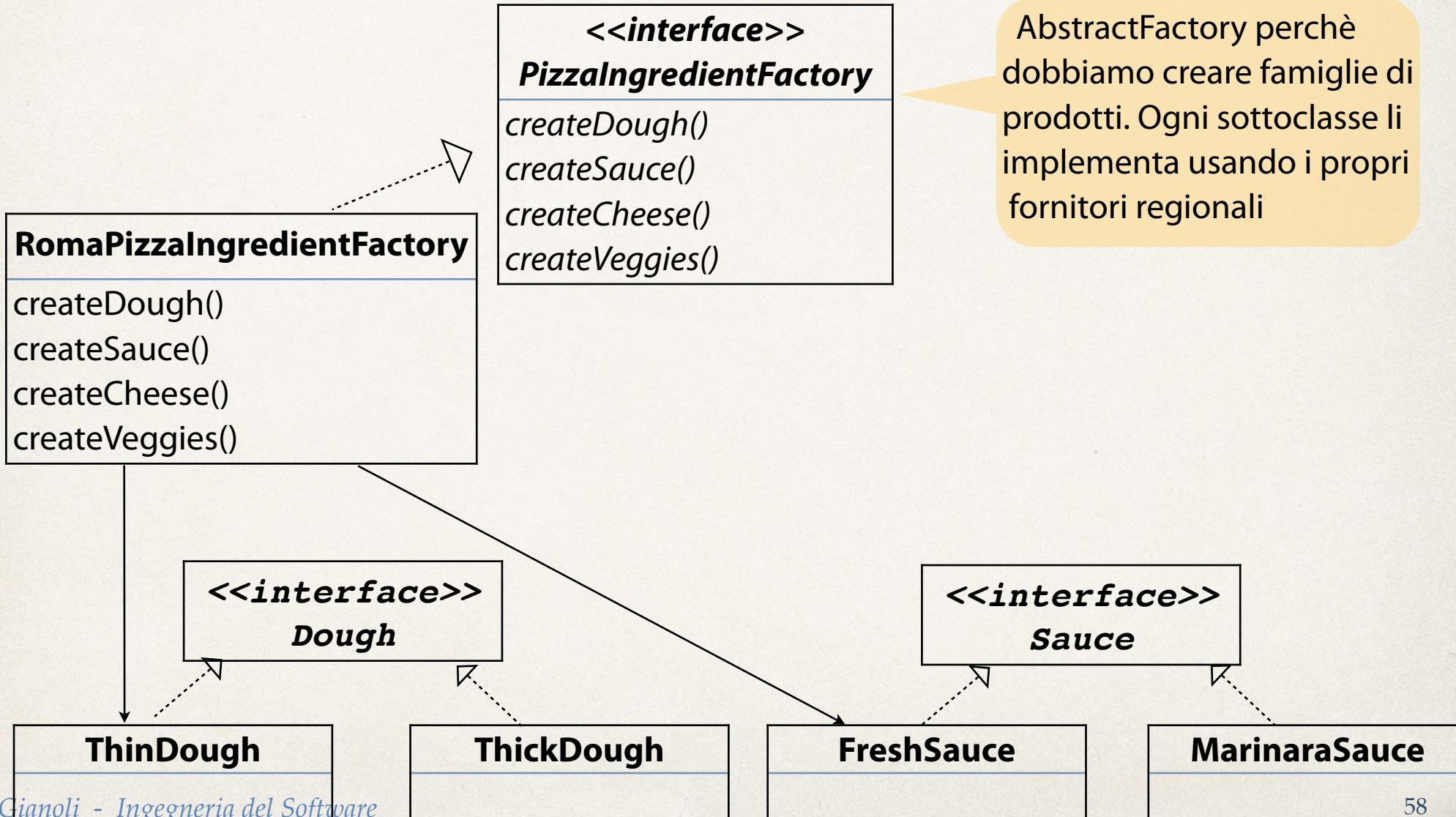


Factory method:
vogliamo creare un
prodotto che varia
con la regione.
Fornisce l'interfaccia

Factory vs Abstract Factory

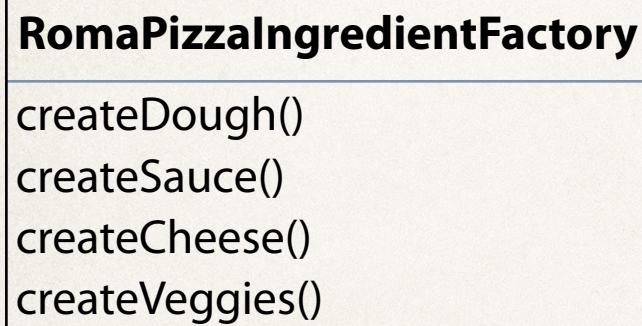


Factory vs Abstract Factory



Factory vs Abstract Factory

Ogni sottoclasse
concreta crea una
famiglia di prodotti



**<<interface>>
PizzalnredientFactory**

```
createDough()  
createSauce()  
createCheese()  
createVeggies()
```

AbstractFactory perchè
dobbiamo creare famiglie di
prodotti. Ogni sottoclasse li
implementa usando i propri
fornitori regionali

**<<interface>>
Dough**

**<<interface>>
Sauce**

ThinDough

ThickDough

FreshSauce

MarinaraSauce

Factory vs Abstract Factory

Ogni sottoclasse
concreta crea una
famiglia di prodotti

RomaPizzalnredientFactory

createDough()
createSauce()
createCheese()
createVeggies()

**<<interface>>
PizzalnredientFactory**

createDough()
createSauce()
createCheese()
createVeggies()

AbstractFactory perchè
dobbiamo creare famiglie di
prodotti. Ogni sottoclasse li
implementa usando i propri
fornitori regionali

I metodi per creare
prodotti di una
AbstractFactory spesso
sono implementati con
un FactoryMethod

**<<interface>>
Dough**

ThinDough

ThickDough

**<<interface>>
Sauce**

FreshSauce

MarinaraSauce

Factory vs Abstract Factory

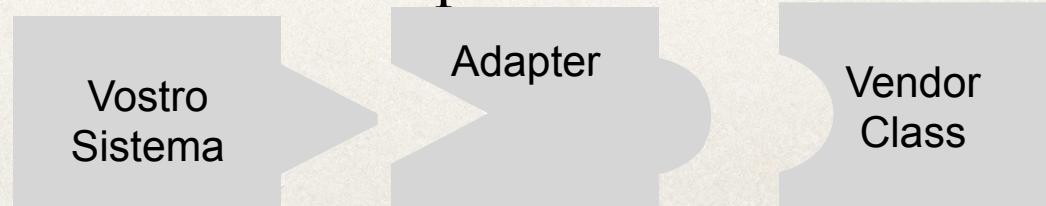
- ❖ Factory: si basa su ereditarietà, la creazione è delegata alla sottoclasse che implementa il factory method per creare gli oggetti
 - ❖ lo scopo è permettere a una classe di rimandare l'istanziare alle sue sottoclassi
- ❖ AbstractFactory: si basa su composizione, la creazione è implementata in metodi esposti nella factory interface
 - ❖ lo scopo è creare famiglie di oggetti correlati senza dover dipendere dalle loro classi concrete

Adapter

Detto così non si coglie la differenza...

Il problema

- ❖ Può essere sia a livello di classe che di oggetto
 - ❖ classe: occorre utilizzare una classe già disponibile (Adaptee) insieme a una libreria di classi sviluppata in modo indipendente; la libreria richiede una particolare interfaccia (Target) che non è presente nell'Adaptee
 - ❖ oggetto: occorre utilizzare una oggetto già disponibile(Adaptee) insieme a una libreria di classi sviluppata in maniera indipendente; la libreria richiede una particolare interfaccia (Target) che non è presente nell'Adaptee



Adapter

Soluzione

- ❖ classe
 - ❖ si crea una nuova classe (Adapter) che implementa l'interfaccia Target ed eredita l'implementazione della classe Adaptee
 - ❖ l'implementazione nell'Adapter dei metodi di Target richiama i metodi di Adaptee
- ❖ oggetto
 - ❖ si crea una nuova classe (Adapter) che implementa l'interfaccia Target e contiene un riferimento a un oggetto della classe Adaptee (tipicamente passato al costruttore)
 - ❖ l'implementazione nell'Adapter dei metodi di Target richiama i metodi dell'oggetto Adaptee

Adapter

```
class MyExistingServiceClass {  
    public void show() {  
        System.out.println("Inside");  
    }  
}  
interface ClientInterface {  
    void display();  
}
```

Soluzione

- ❖ classe
 - ❖ si crea una nuova classe (Adapter) che implementa l'interfaccia Target ed eredita l'implementazione della classe Adaptee
 - ❖ l'implementazione nell'Adapter dei metodi di Target richiama i metodi di Adaptee
- ❖ oggetto
 - ❖ si crea una nuova classe (Adapter) che implementa l'interfaccia Target e contiene un riferimento a un oggetto della classe Adaptee (tipicamente passato al costruttore)
 - ❖ l'implementazione nell'Adapter dei metodi di Target richiama i metodi dell'oggetto Adaptee

Adapter

```
class MyExistingServiceClass {  
    public void show() {  
        System.out.println("Inside");  
    }  
}  
interface ClientInterface {  
    void display();  
}
```

Soluzione

- classe

- si crea una classe che estende la classe Target ed implementa l'interfaccia ClientInterface
- l'implementazione degli metodi di ClientInterface

```
class MyNewClassAdapter extends MyExistingServiceClass  
    implements ClientInterface {  
    void display() {  
        show();  
    }  
}
```

- oggetto

- si crea una nuova classe (Adapter) che implementa l'interfaccia ClientInterface e contiene un riferimento a un oggetto della classe Adaptee (tipicamente passato al costruttore)
- l'implementazione nell'Adapter dei metodi di ClientInterface richiama i metodi dell'oggetto Adaptee

Adapter

```
class MyExistingServiceClass {  
    public void show() {  
        System.out.println("Inside");  
    }  
}  
interface ClientInterface {  
    void display();  
}
```

Soluzione

- classe

- si crea un Target ed implementa i metodi del

```
class MyNewClassAdapter extends MyExistingServiceClass  
    implements ClientInterface {  
    void display() {  
        show();  
    }  
}
```

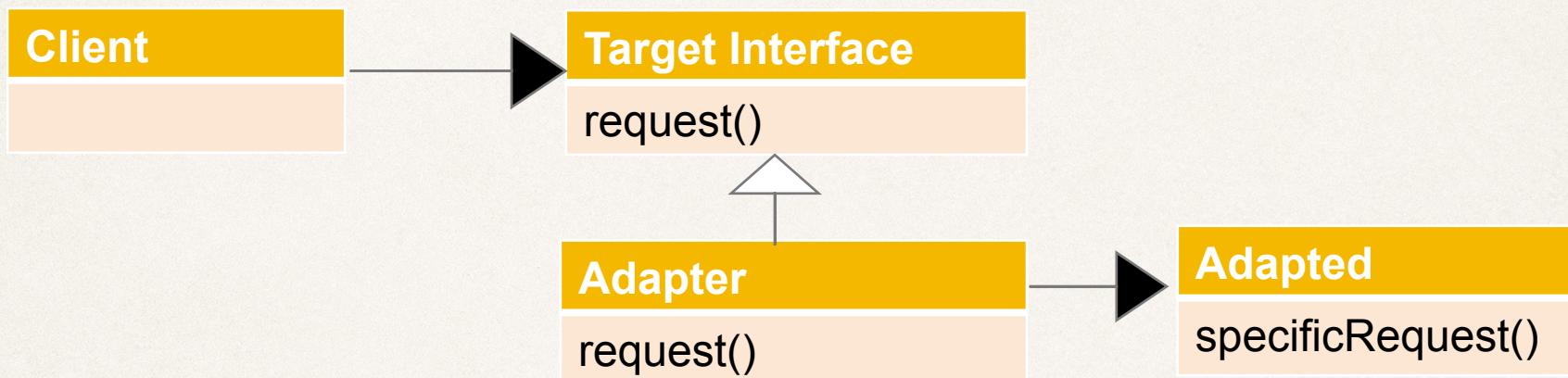
- oggetto

- class MyNewObjectAdapter implements ClientInterface {
 MyExistingServiceClass existingClassObject;

 void display() {
 existingClassObject.show();
 }
}

Adapter

- ❖ Esempio di struttura, caso di Adapter di classi



Adapter

Conseguenze

- ❖ classe
 - ❖ se Target non è una interfaccia pura (java “interface”) è necessaria ereditarietà multipla
 - ❖ se c’è una gerarchia di Adaptee occorre una gerarchia parallela di Adapter
- ❖ oggetto
 - ❖ l’adapter può essere utilizzato per oggetti della classe Adaptee e di tutte le classi derivate
 - ❖ Adapter e Adaptee rimangono due oggetti distinti: overhead di memoria

Composite

Il problema

- vogliamo trattare un insieme di oggetti come se fosse l'istanza di un unico tipo di oggetti

Lo scopo

- combinare gli oggetti in una struttura gerarchica ad albero

Motivazione

- differenziare tra foglia e nodo di solito richiede una logica complessa che può introdurre errori

Composite

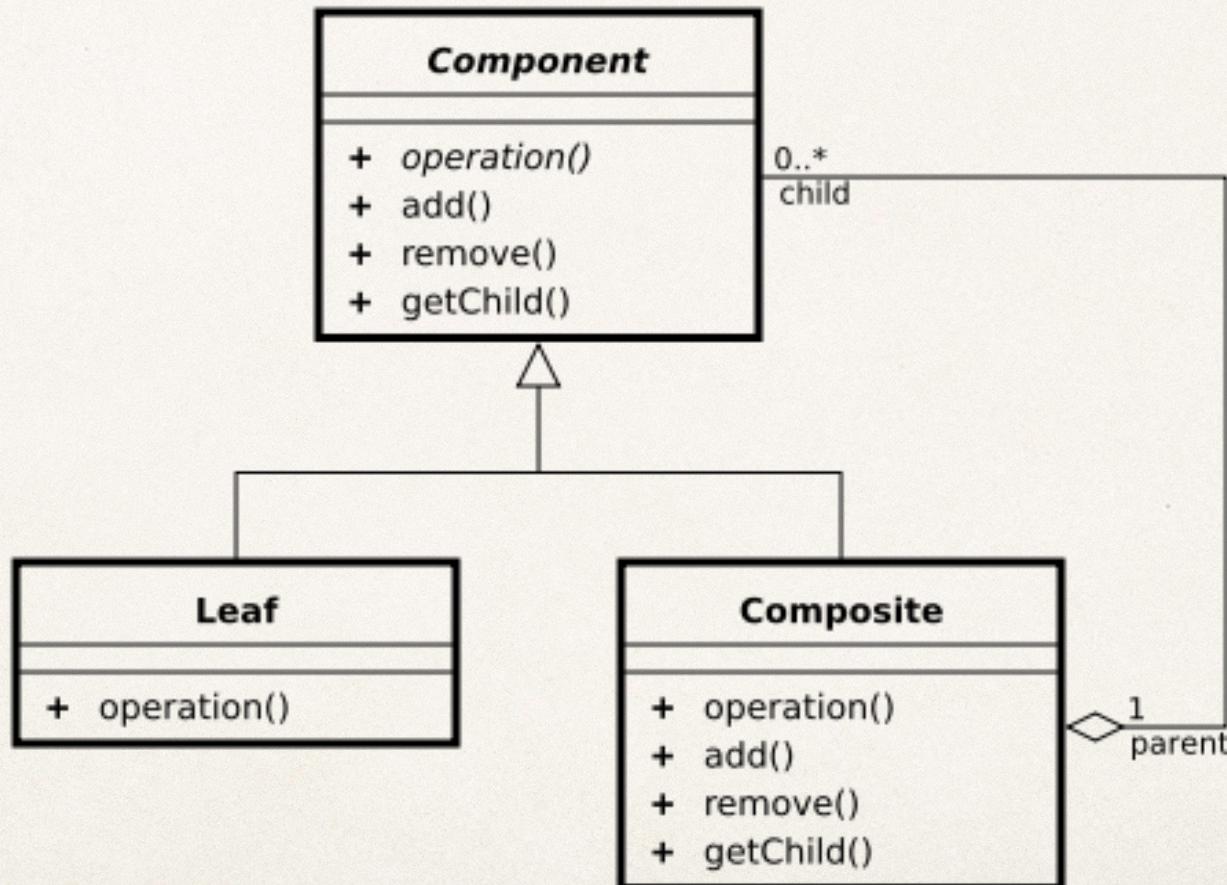
Soluzione

- una interfaccia che tratti oggetti complessi e elementari in modo uniforme

Quando usarlo:

- quando il client non ha bisogno di differenziare tra foglie e nodo (cioè hanno la stessa funzionalità)
- es: un menu strutturato - meglio un metodo “display()” che deleghi alle foglie / rami sottostanti piuttosto che dover iterare attraverso tutta la struttura

Composite



Composite

Component

- ⊕ è l'astrazione di tutti i componenti, compresi quelli composti
- ⊕ dichiara l'interfaccia degli oggetti
- ⊕ implementa il comportamento di default dell'interfaccia comune a tutte le classi
- ⊕ dichiara una interfaccia per accedere e gestire i suoi componenti figlio

Leaf

- ⊕ rappresenta una foglia dell'albero
- ⊕ implementa tutti i metodi di Component

Composite

Composite

- ❖ rappresenta un Component composito (un ramo)
- ❖ implementa metodi per manipolare i livelli sottostanti
- ❖ implementa tutti i metodi di Component, di solito delegandoli ai livelli sottostanti

Facciamo un esempio (1)

Dobbiamo progettare un editor grafico di documenti (maggiori dettagli li trovare nel libro della GoF)

Requisiti

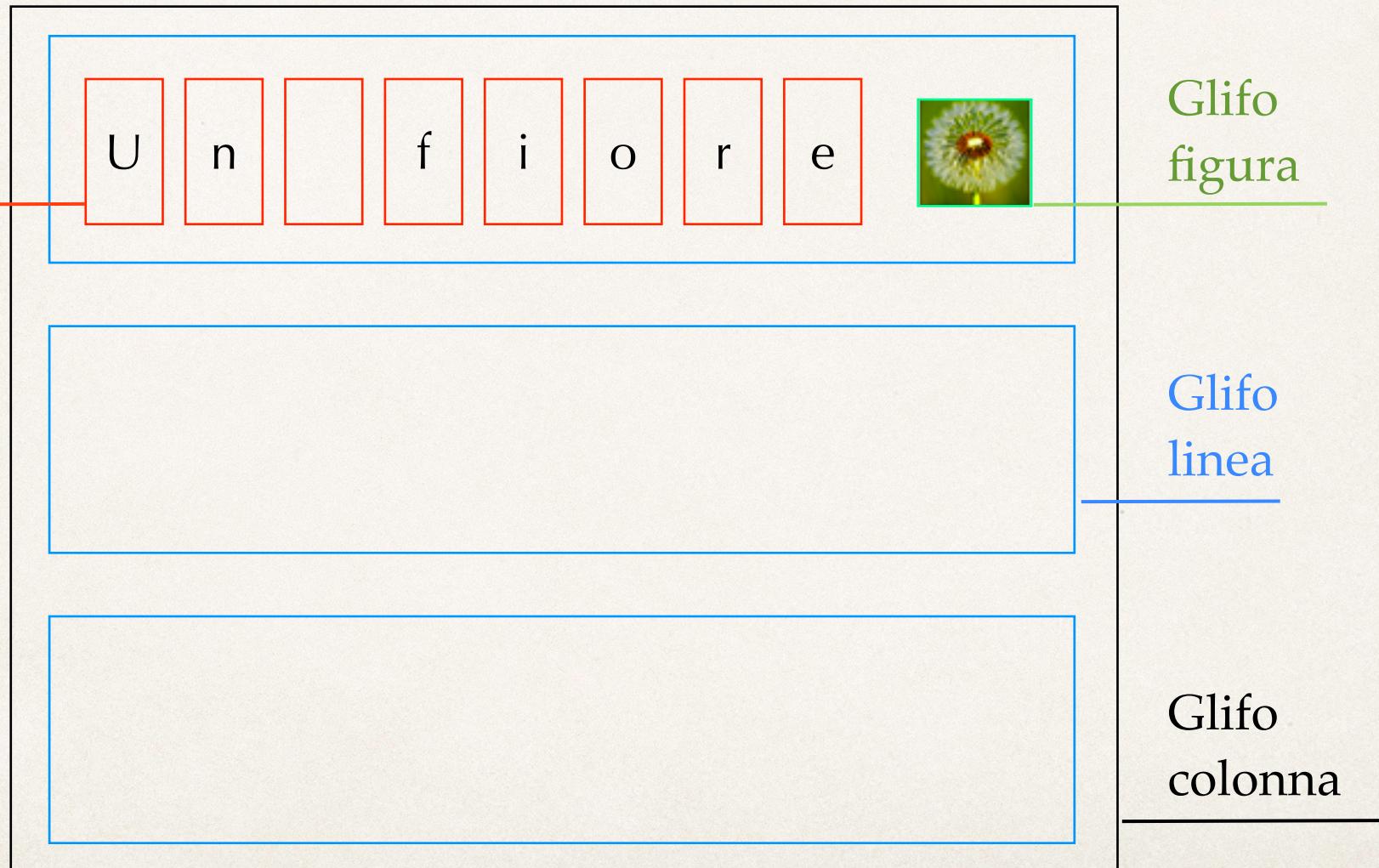
- ⊕ un documento è una sequenza di pagine con testo e grafica
- ⊕ interfaccia grafica
- ⊕ finestre multiple
- ⊕ operazioni sui testi: sillabazione, controllo vocaboli

Facciamo un esempio (1)

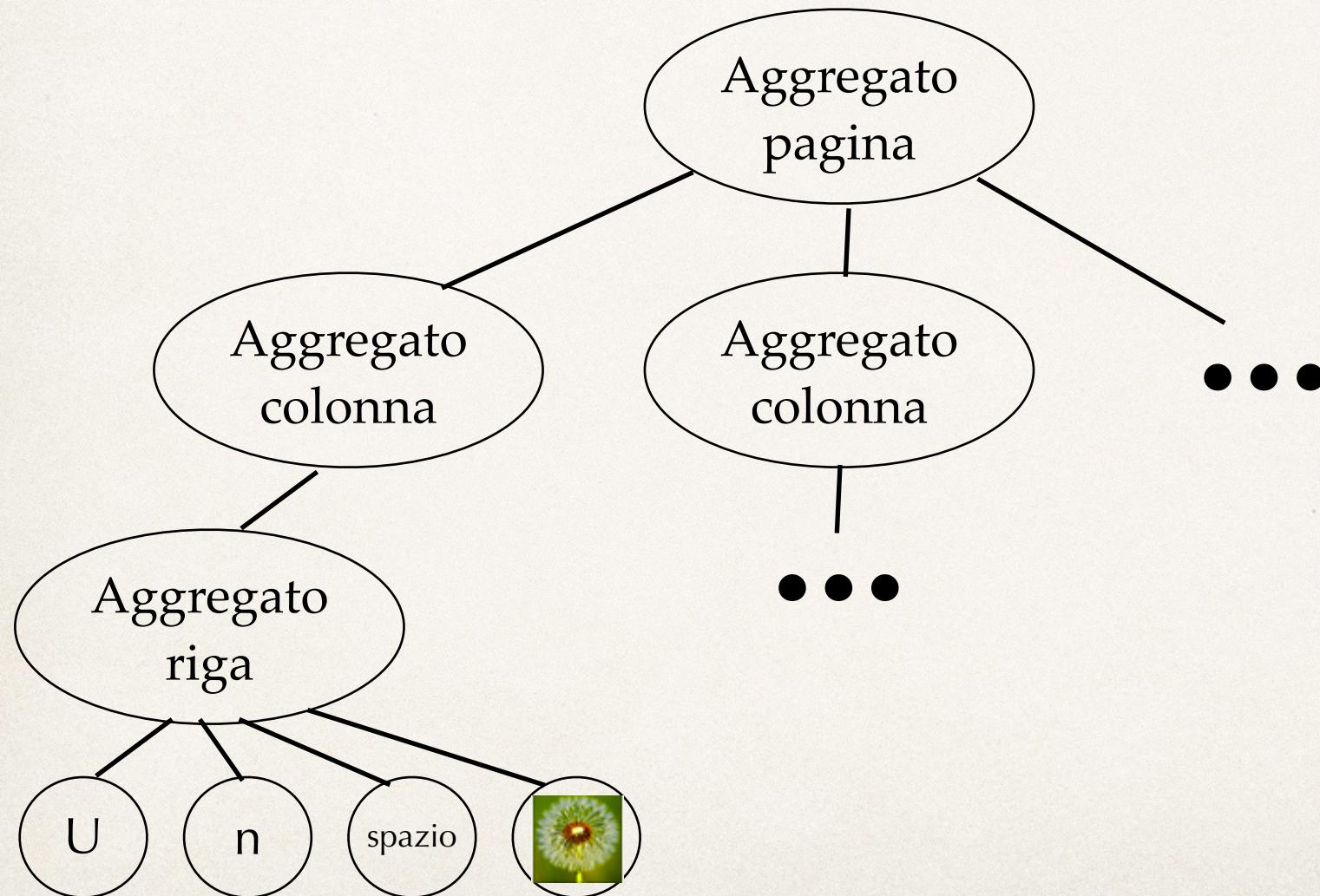
Struttura di un documento

- ✿ un documento ha una sua struttura
- ✿ documento: sequenza di pagine
- ✿ pagina: sequenza di colonne
- ✿ colonna: sequenza di linee
- ✿ linea: sequenza di glifi
- ✿ glifo: elemento primitivo; carattere, figura, rettangolo, cerchio,

Facciamo un esempio (1)



Facciamo un esempio (1)



Facciamo un esempio (1)

Vediamo le alternative progettuali

1. Classi diverse per i vari elementi primiviti: carattere, linea, colonna, pagina; elementi grafici quali quadrato, cerchio, ...
2. Una sola classe astratta per la nozione di “glifo”, che rappresenta un elemento grafico generico con una interaccia standard, che poi specializziamo in tanti modi diversi

Facciamo un esempio (1)

```
Class Glyph {  
    List children = new LinkedList();  
    Int ox, oy, width, height;  
  
    Void draw() {  
        For (g:children) g.draw();  
    }  
  
    Void insert(Glyph g) {  
        children.add(g);  
    }  
  
    Boolean intersects(int x, int y) {  
        Return (x>=ox) && (x<ox+width)  
        && (y>=oy) && (y<oy+height);  
    }  
}
```

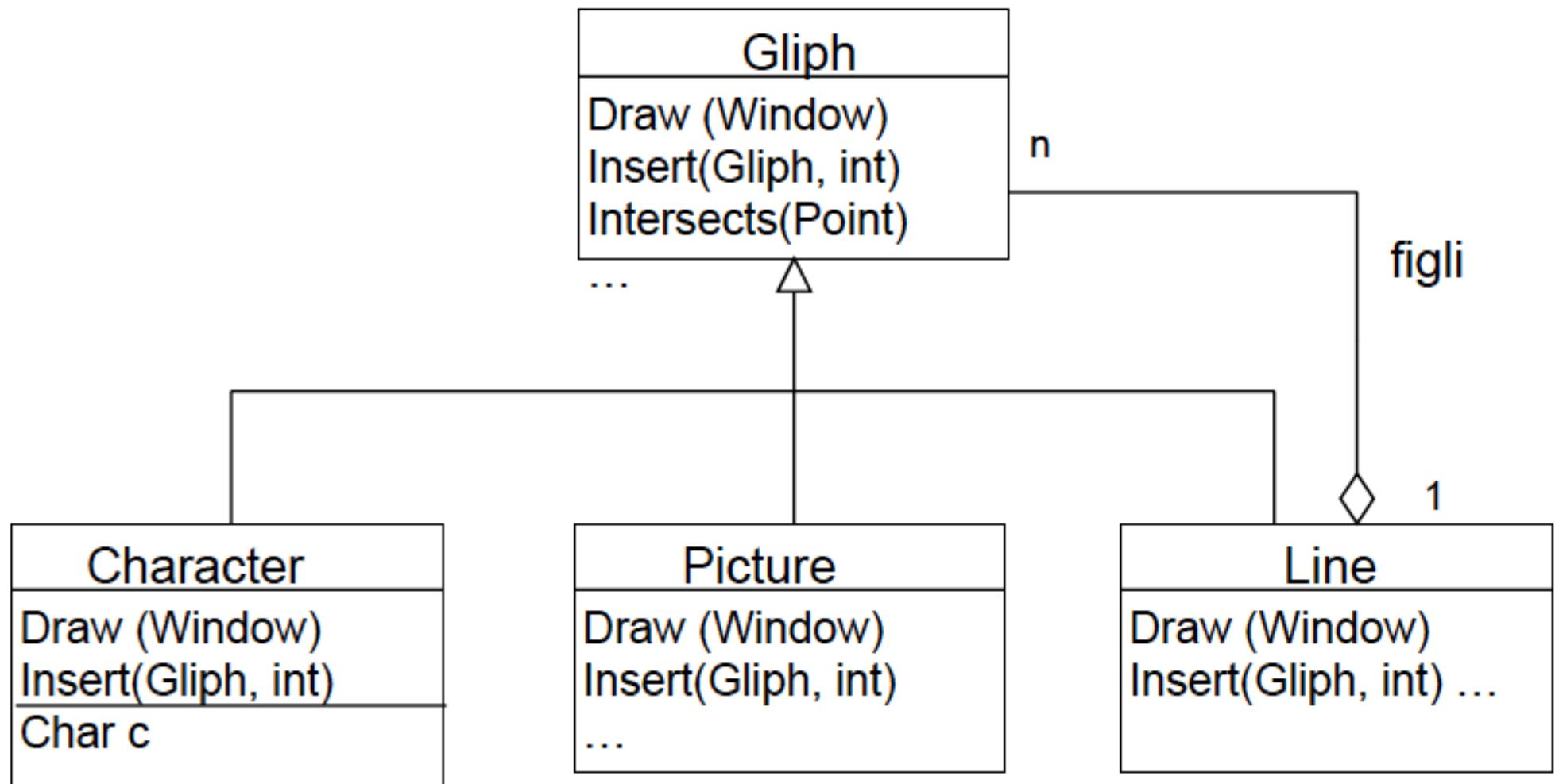
```
class Character extends Glyph {  
    char c;  
  
    public Character(char c) {  
        this.c = c;  
    }  
  
    void draw() {  
        . . . .  
    }  
}
```

Facciamo un esempio (1)

```
class Picture extends Glyph {  
    File pictureFile;  
  
    public Picture(File  
                  pictureFile) {  
        this.pictureFile =  
            pictureFile;  
    }  
  
    void draw() {  
        . . .  
    }  
}
```

```
class Line extends Glyph {  
    char c;  
  
    public Line() {}  
    // inherits draw, ...  
}
```

Facciamo un esempio (1)



Decorator

Il problema

- ❖ si vuole aggiungere delle responsabilità ad un oggetto (Component) senza cambiarne l'interfaccia
- ❖ l'aggiunta deve essere fatta a runtime, oppure ci sono più aggiunte utilizzabili in combinazioni diverse; oppure si deve rimuovere a runtime le responsabilità aggiunte; insomma non è possibile usare l'ereditarietà

Decorator

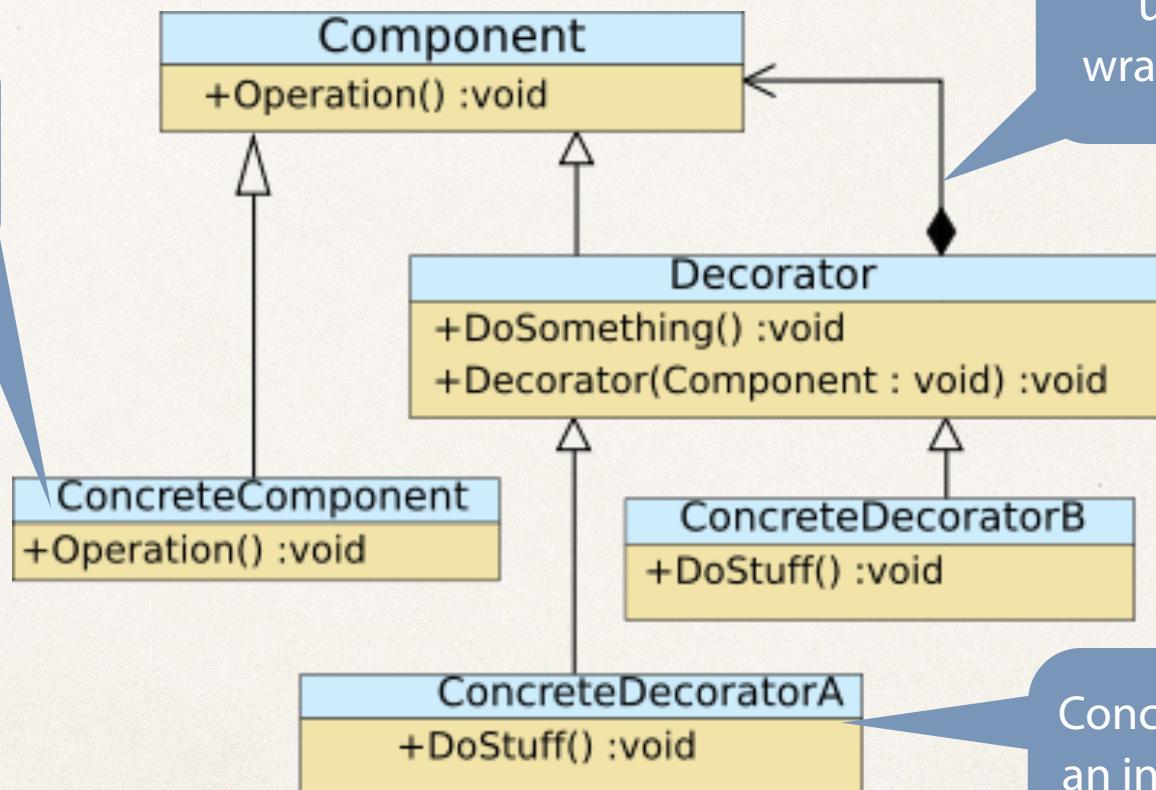
Soluzione

- si definisce una classe **Decorator** che implementa l’interfaccia di **Component**
- **Decorator** mantiene un riferimento al **Component** che viene “decorato”
- l’implementazione delle operazioni di **Component** nella classe **Decorator** richiama l’implementazione del componente decorato, ma ha la possibilità di fare delle pre o post elaborazioni

Decorator

❖ Esempio di struttura

Concrete Component
we can dynamically add
new behavior



Each component can be used on its own or wrapped by a decorator

Open-Close
Principle

- Classes should be open for extension
- Should be closed for modification

Concrete Decorator has an instance variable for the thing it decorates

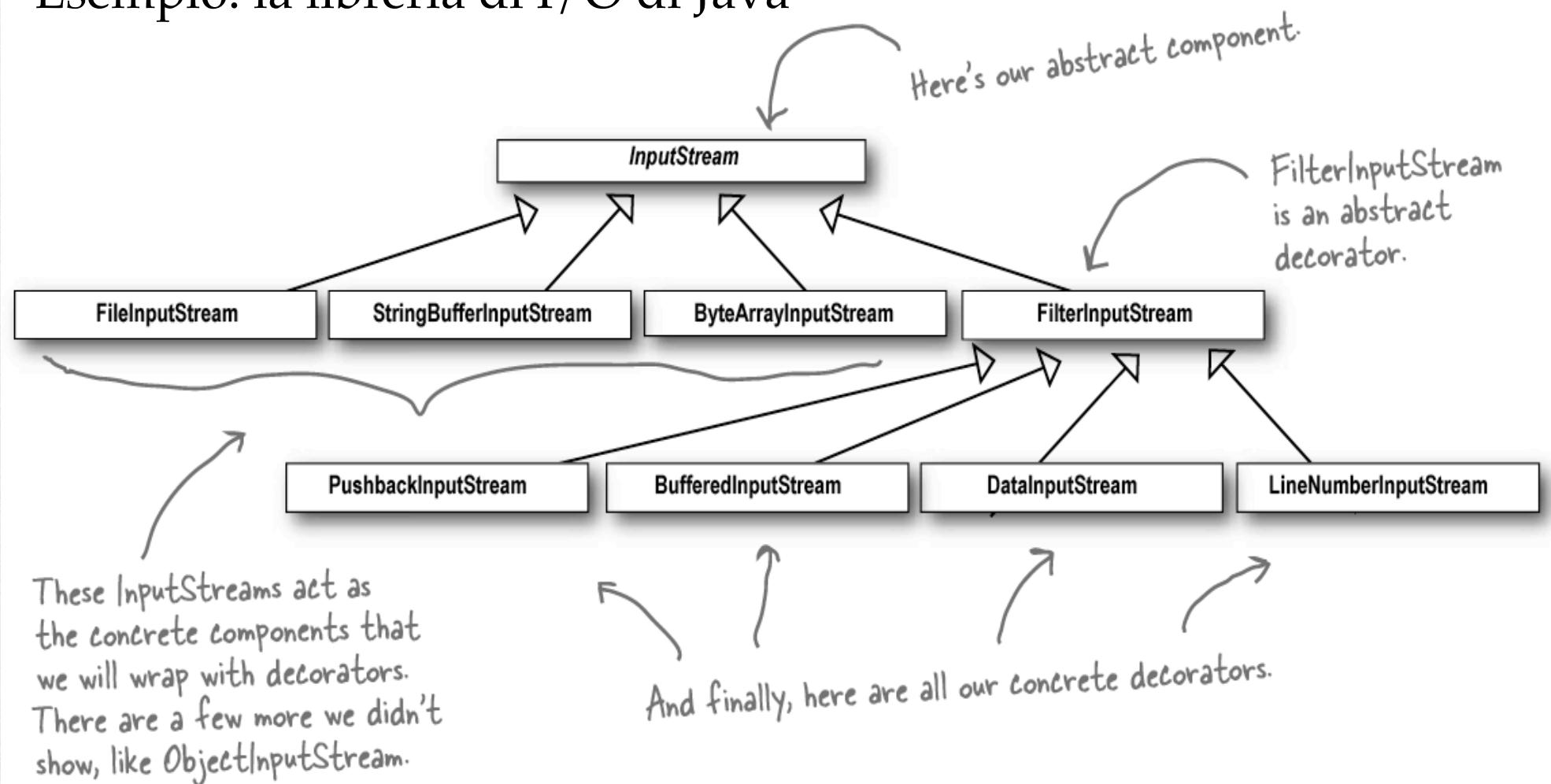
Decorator

Conseguenze

- il comportamento aggiuntivo è trasparente per gli utenti del componente (sostituendo i riferimenti al Decorator al posto dei riferimenti al componente iniziale)
- è possibile applicare più Decorator in cascata
- l'insieme dei Decorator può essere deciso a runtime (ed è possibile rimuovere un decoratore durante l'esecuzione)

Decorator

- ❖ Esempio: la libreria di I/O di Java



Decorator

```
public class LowerCaseInputStream
        extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c :
            Character.toLowerCase((char)c));
    }
    public int read(byte[] b, int offset, int len)
throws IOException {
    int result = super.read(b, offset, len);
    for (int i = offset; i<offset+result; i++) {
        b[i]=(byte)Character.toLowerCase((char)b[i]);
    }
    return result;
}
}
```

Decorator

```
public class LowerCaseInputStream
    extends FilterInputStream {
public LowerCaseInputStream(InputStream in) {
    super(in);
}

public int read() throws IOException {
    int c = super.read();
    return (c == -1 ? c :
        Character.toLowerCase((char)c));
}
public int read(byte[] b, int offset, int len)
throws IOException {
    int result = super.read(b, offset, len);
    for (int i = offset; i<offset+result; i++) {
        b[i]=(byte)Character.toLowerCase((char)b[i]);
    }
    return result;
}
```

```
public class InputTest {
    public static void main(String[] args)
throws IOException {
    int c;
    try {
        InputStream in =
            new LowerCaseInputStream(
            new BufferedInputStream(
            new FileInputStream("test.txt")));
        while((c = in.read()) >= 0) {
            System.out.print((char)c);
        }
        in.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Decorator

```
public class LowerCaseInputStream
        extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c :
            Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len)
throws IOException {
        int result = super.read(b, offset, len);
        for (int i = offset; i<offset+result; i++) {
            b[i]=(byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

Crea un FileInputStream e
decoralo, con un
BufferedInputStream e poi con il
nostro LowerCaseInputStream

```
public class InputTest {
    public static void main(String[] args)
throws IOException {
    int c;
    try {
        InputStream in =
            new LowerCaseInputStream(
            new BufferedInputStream(
            new FileInputStream("test.txt")));
        while((c = in.read()) >= 0) {
            System.out.print((char)c);
        }
        in.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Decorator

- ❖ Questo pattern è utilizzato spesso per aggiungere a un oggetto responsabilità che non riguardano “cosa” viene fatto, ma “come”:
 - ❖ logging
 - ❖ gestione delle transazioni
 - ❖ caching
 - ❖ sincronizzazione

Facciamo un esempio (2)

- ❖ Riprendiamo il nostro editor grafico
- ❖ Vogliamo aggiungere una serie di decori ai vari elementi dell'interfaccia utente (bordi, scrollbar, buttoni, ..)
- ❖ Come incorporare questo nella struttura dell'editor?

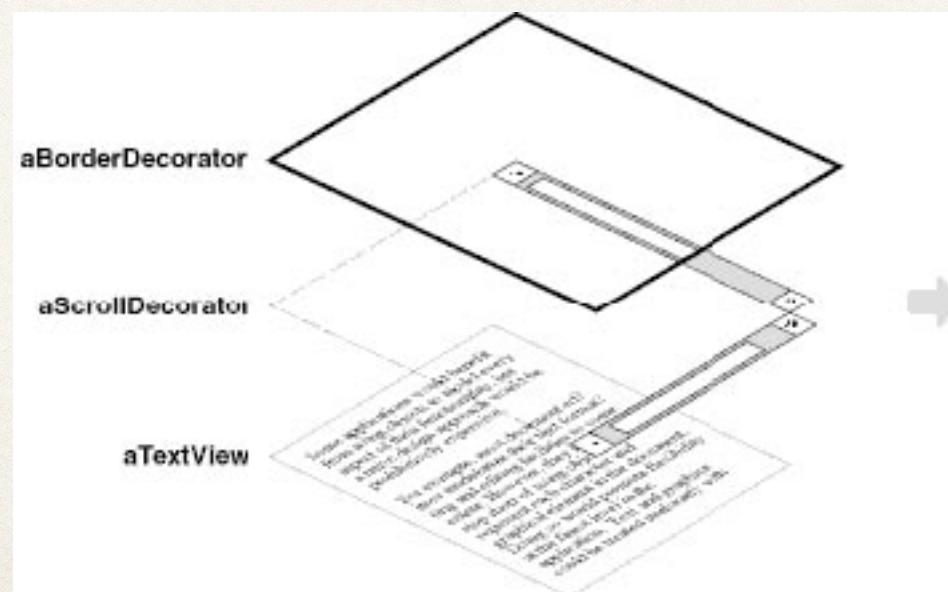
Facciamo un esempio (2)

- ❖ Potremmo usare l'ereditarietà per estendere il comportamento
 - ▶ sottoclassi di Glyph: Glyph con bordo, Glyph con scroll, Glyph bottone, Glyph con bordo e scroll, ...
 - ▶ se abbiamo n decoratori ci ritroviamo con 2^n combinazioni
- ❖ Esplosione del numero di classi!
(senza contare che la gerarchia di ereditarietà è statica...)

Facciamo un esempio (2)

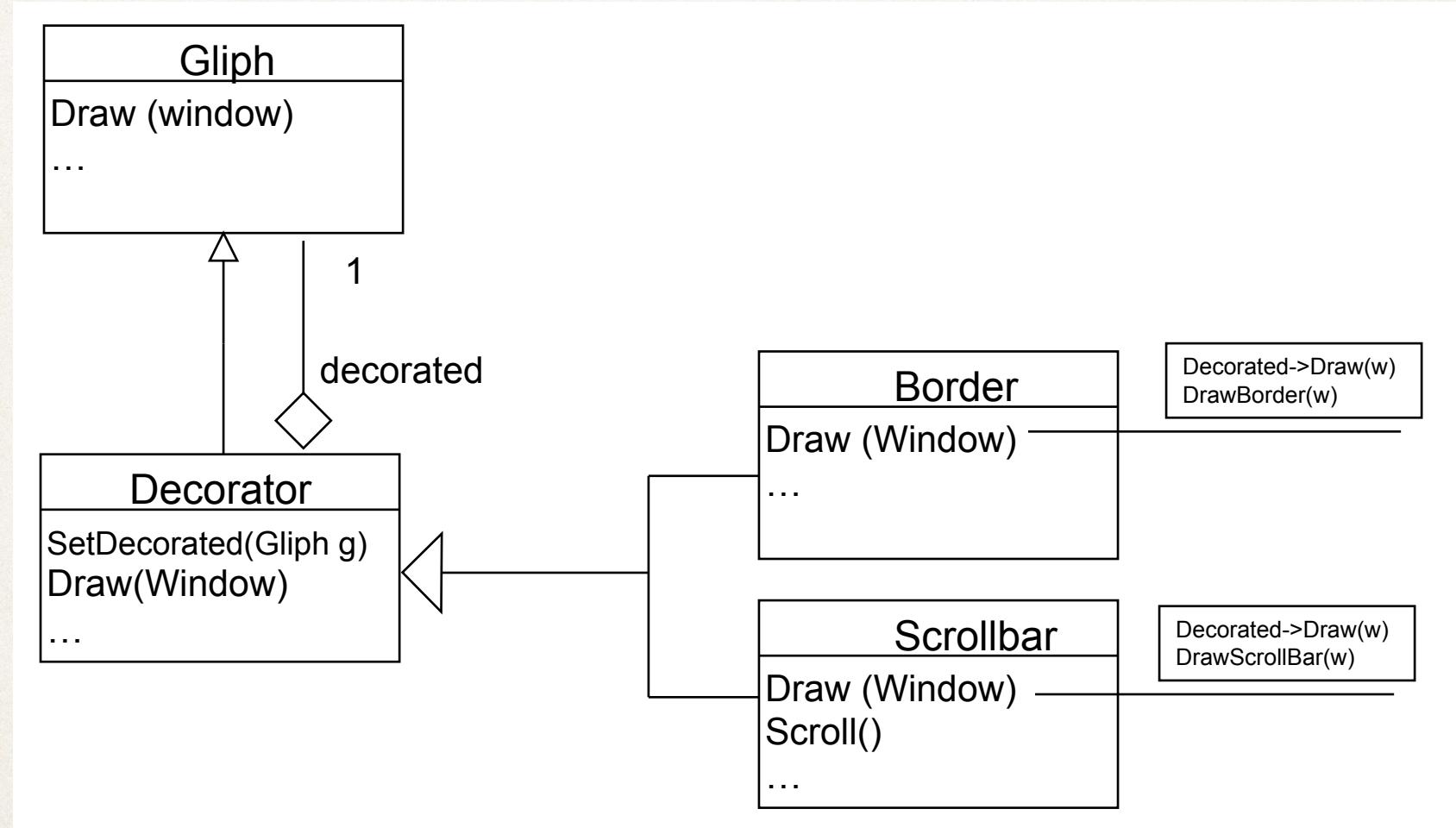
Proviamo in un altro modo

- ❖ Vogliamo che le decorazioni (bordi, scrollbar, buttoni, menu,...) possano essere combinate indipendentemente e a piacere
 - ❖ del tipo: `a=new Scrollbar(new Border(new Character))`
- ❖ Usiamo il pattern Decorator
 - ❖ che estende `Glyph`
 - ❖ ha un membro `decorator` di tipo `Glyph`
 - ❖ `Border`, `Scrollbar`, `Button`, `Menu` estendono `Glyph`



Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

Facciamo un esempio (2)



Facciamo un esempio (3)

- ⊕ Esistono diversi standard di “look-and-feel” per le interfacce grafiche
- ⊕ Differiscono per come visualizzano menu, bottoni, scrollbar, ...
- ⊕ Vogliamo supportarli tutti

Facciamo un esempio (3)

```
Menu m = new MacosMenu;
```

oppure

```
Menu m = new WindowsMenu;
```

```
Menu m;
if(style==Macos)
    m=new MacosMenu;
else if(style==...)
    ...

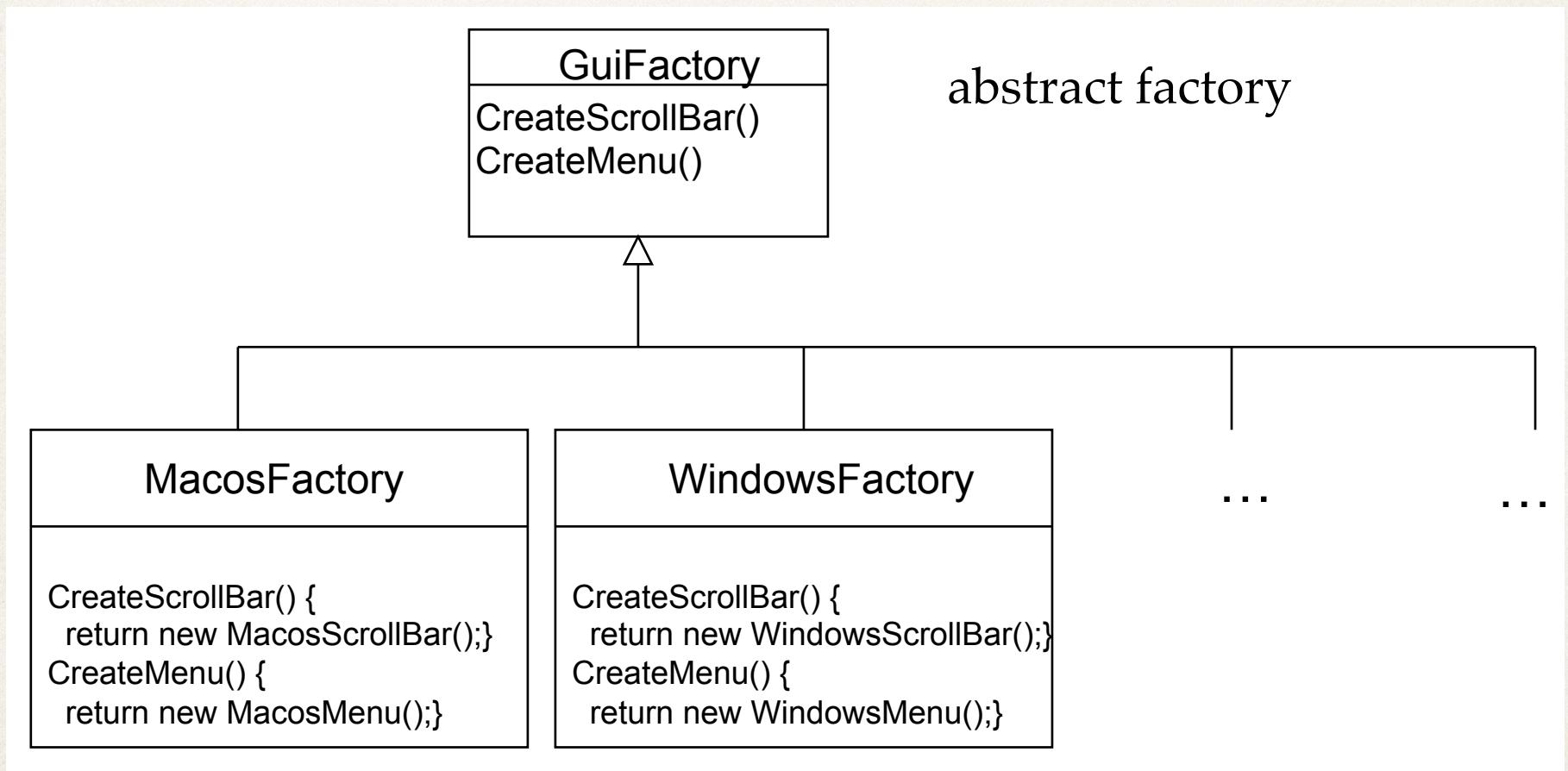
```

Uhmeeee....

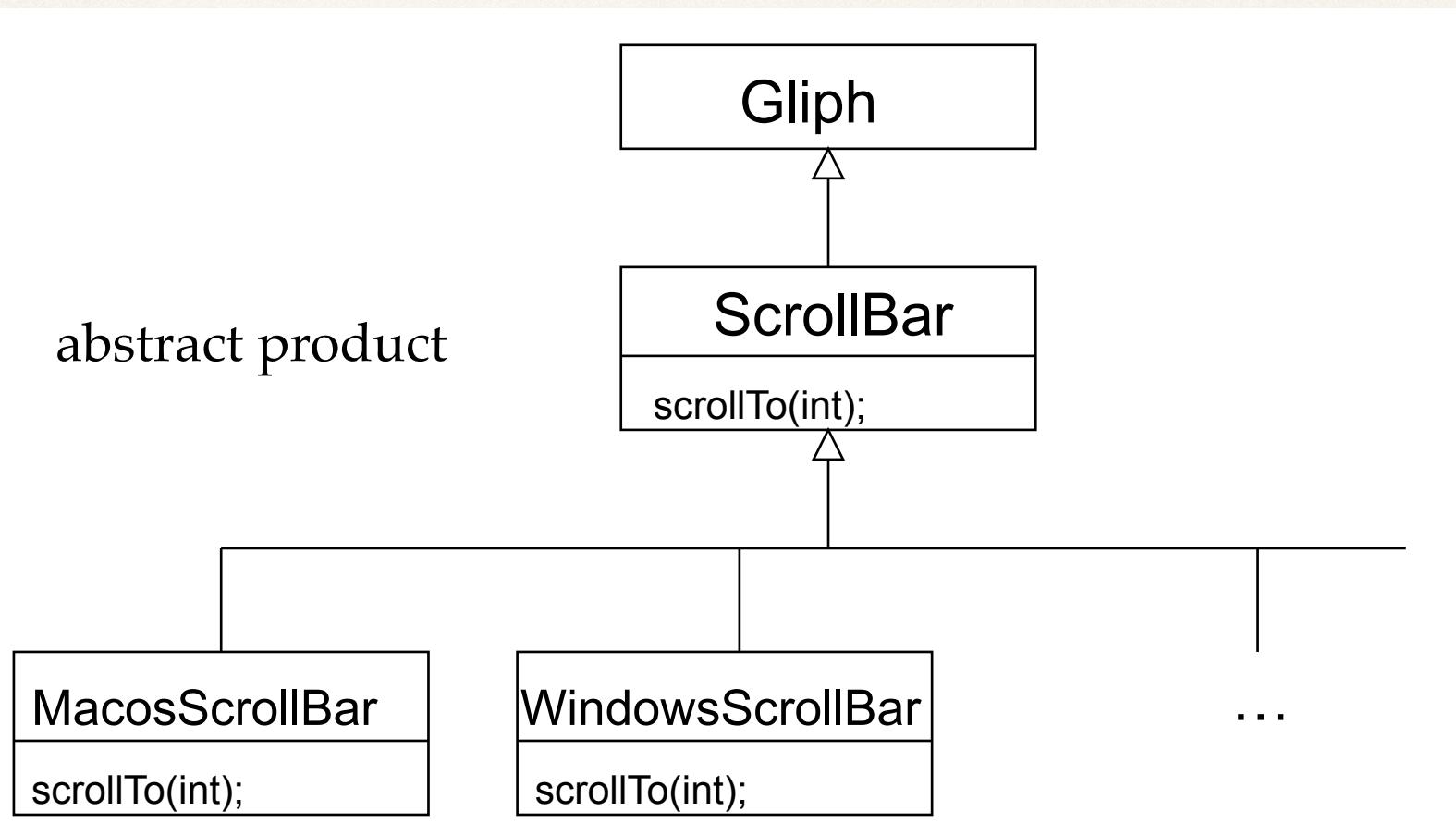
Facciamo un esempio (3)

- ❖ Conviene usare l'abstract factory
- ❖ Definiamo una classe GUIFactory
 - ▶ inseriamo nella class un metodo per ciascun oggetto dipendente dalla GUI
 - ▶ avremo un oggetto GUIFactory per ciascun tipo di GUI

Facciamo un esempio (3)



Facciamo un esempio (3)



Observer

Problema

- ❖ Spesso i cambiamenti nello stato di un oggetto (Subject) devono riflettersi su uno o più oggetti da esso dipendenti
- ❖ si vuole disaccoppiare il Subject dagli oggetti dipendenti

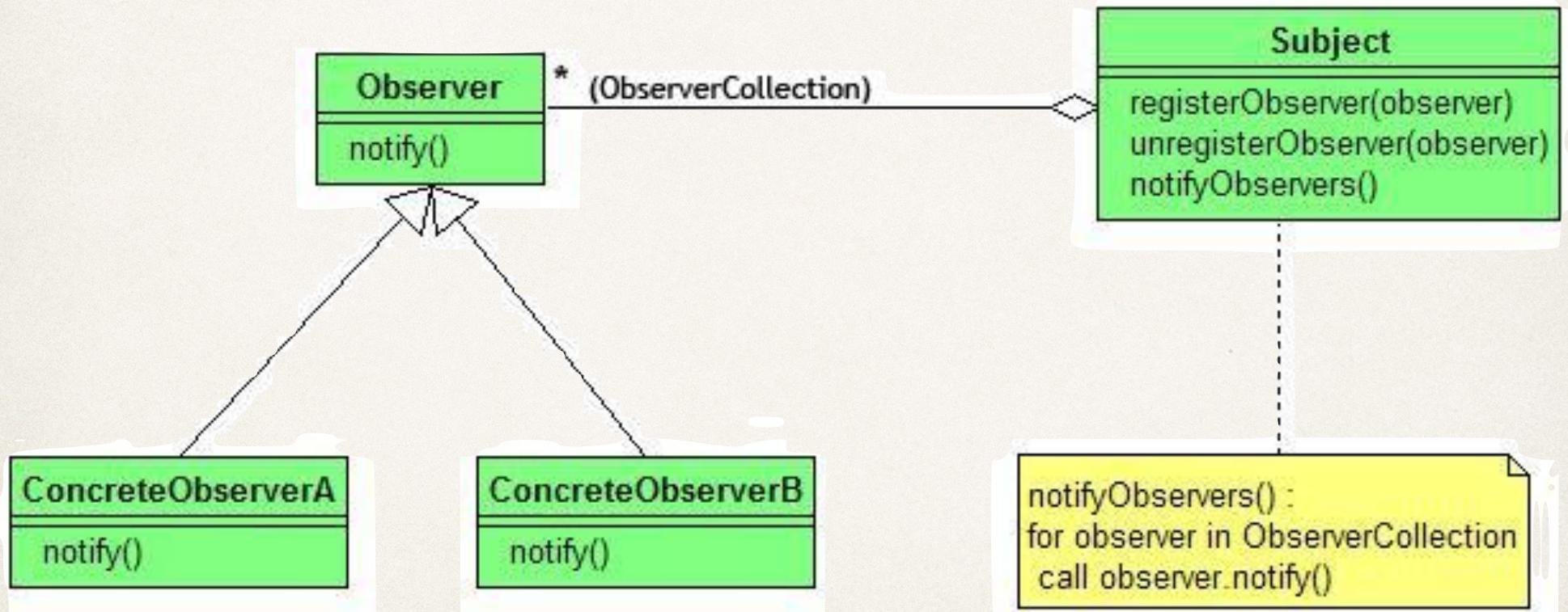
Observer

Soluzione

- ❖ Si definisce una interfaccia Observer, con un metodo che viene richiamato ad ogni modifica dello stato del Subject
- ❖ Gli oggetti (che implementano Observer) che sono interessati a un determinato Subject devono essere registrati presso il Subject con un apposito metodo
- ❖ Il Subject provvede a richiamare il metodo di notifica per tutti gli Observer registrati ogni volta che cambia il proprio stato

Observer

- Esempio di struttura



Observer

Esempio

- ❖ la libreria standard Java vi mette già a disposizione una classe (`java.util.Observable`) e un'interfaccia (`java.util.Observer`) per implementare questo pattern
 - ❖ il Subject estende Observable
 - ❖ quando un metodo modifica lo stato del Subject, deve chiamare il metodo **setChanged** per segnalare il cambiamento
 - ❖ al termine di una serie di cambiamenti occorre chiamare il metodo **notifyObservers** per avvisare gli Observer
 - ❖ ciascun Observer viene avvisato del cambiamento attraverso il metodo **update**

```
import java.util.Observable;
public class AlertCondition extends Observable {
    public static final int GREEN=0,
                           YELLOW=1,
                           RED=2;
    private int condition;
    public AlertCondition() {
        condition=GREEN;
    }
    public int getCondition() {
        return condition;
    }
    public void setCondition(int newCondition) {
        if (newCondition!=RED &&
            newCondition!=YELLOW &&
            newCondition!=GREEN)
            throw new RuntimeException("Unvalid alert condition!");
        if (newCondition != condition) {
            condition=newCondition;
            setChanged();
        }
        notifyObservers();
    }
}
```

```
import java.util.*;
import java.io.*;
import java.text.*;

public class LogAlertObserver implements Observer {
    private PrintWriter out;
    public LogAlertObserver(String fileName) throws IOException {
        FileOutputStream fos=new FileOutputStream(fileName, true);
        OutputStreamWriter osw=new OutputStreamWriter(fos, "UTF-8");
        out=new PrintWriter(osw);
    }
    public void update(Observable subject, Object arg) {
        AlertCondition alert=(AlertCondition)subject;
        DateFormat dfmt=DateFormat.getDateInstance(DateFormat.MEDIUM,
                                                    DateFormat.LONG);
        String date=dfmt.format(new Date());
        String state;
        switch (alert.getCondition()) {
            case AlertCondition.GREEN: state="GREEN"; break;
            case AlertCondition.YELLOW: state="YELLOW"; break;
            case AlertCondition.RED: state="RED"; break;
            default: state="UNKNOWN";
        }
        out.println("[ "+date+" ] the alert is: "+state);
        out.flush();
    }
}
```

Observer

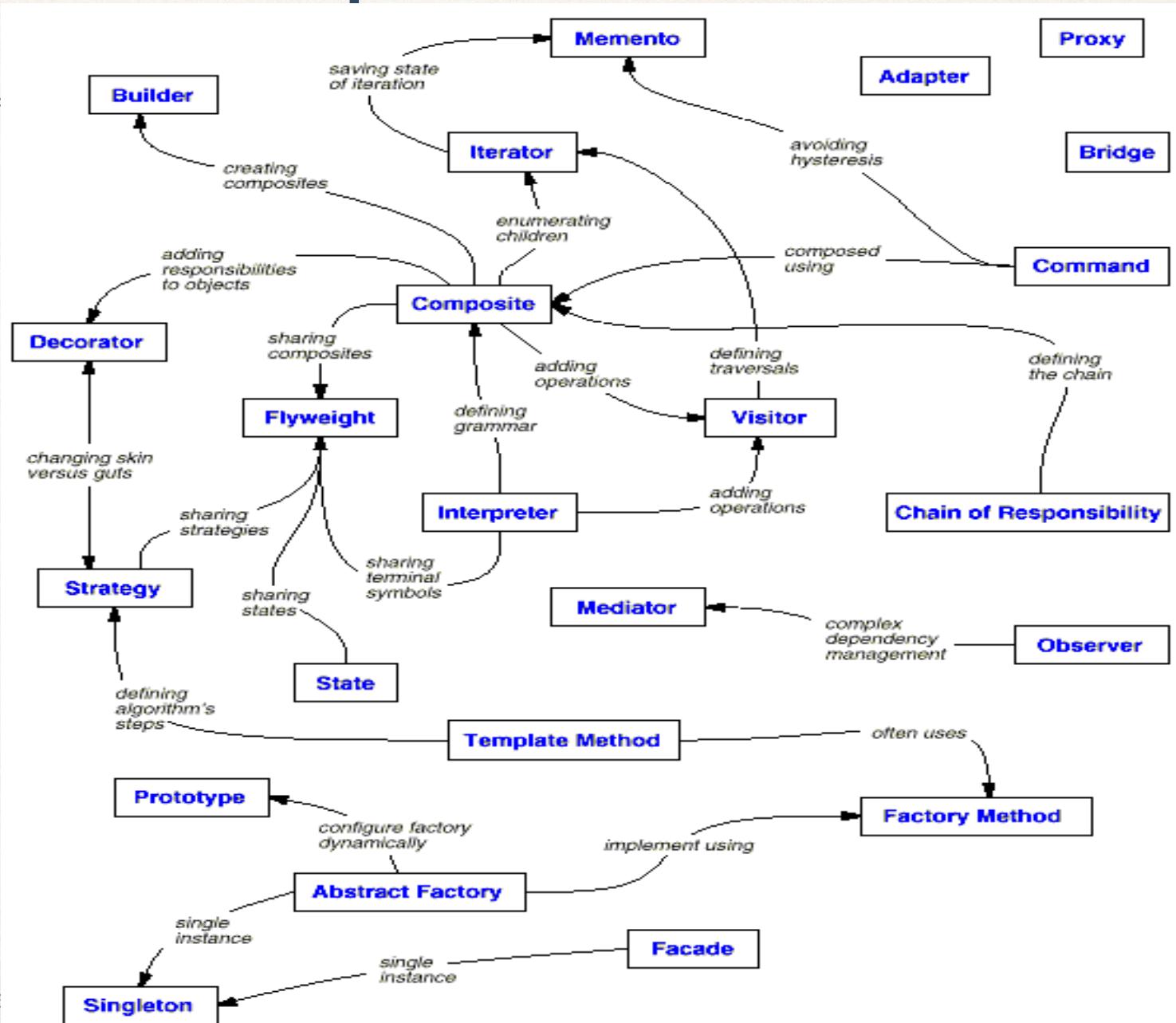
Loose Coupling

- Strive for loosely coupled designs between objects that interact

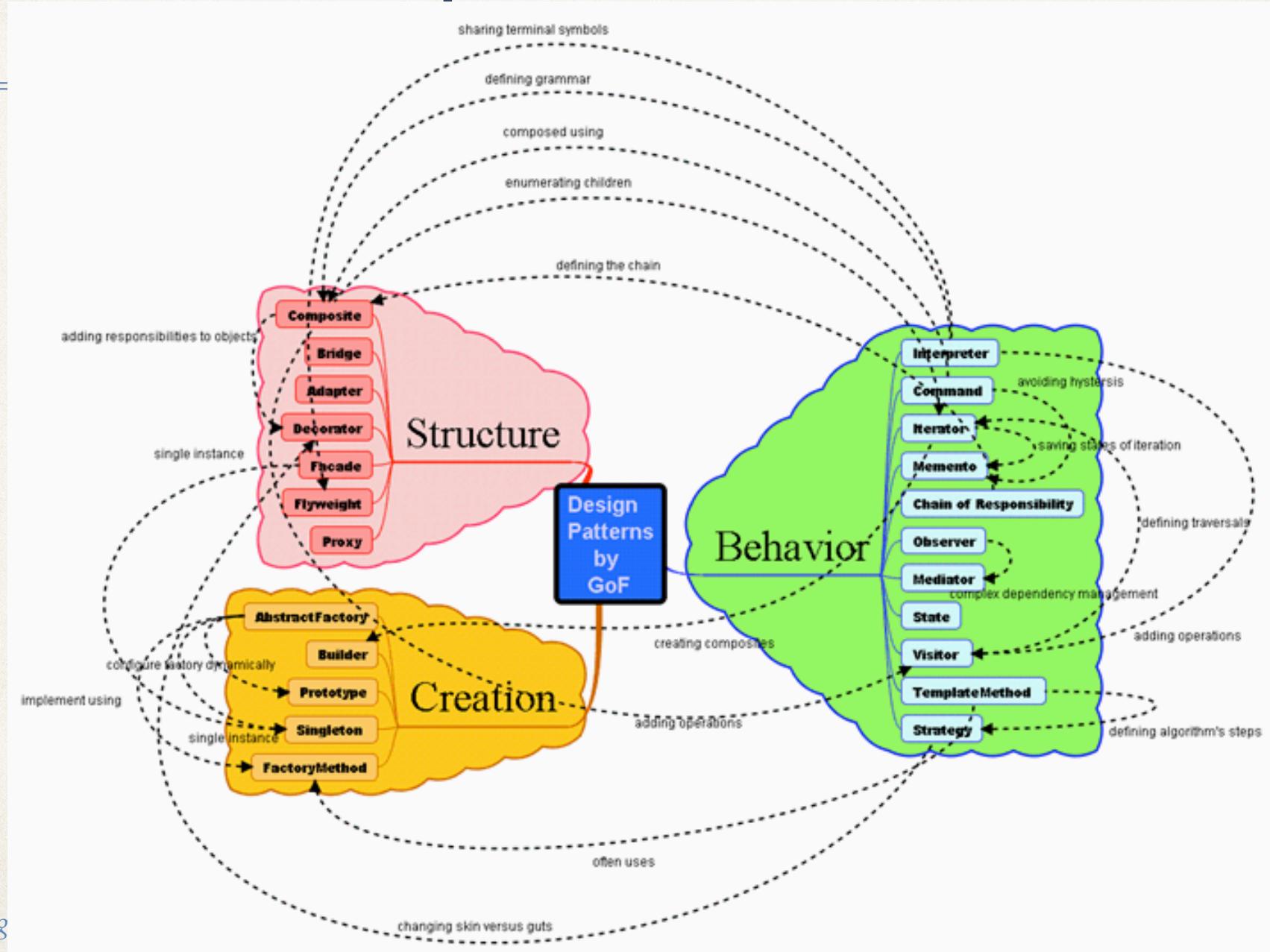
- Il Subject non si preoccupa: manderà notifiche a ogni oggetto che implementa l'interfaccia Observer
- La progettazione scarsamente accoppiata ci permette di costruire un sistema OO che può essere modificato perché minimizza l'interdipendenza tra gli oggetti
- “Pull” è considerato più corretto che “Push”
- Il framework Swing usa pesantemente questo pattern
- Il pattern Observer definisce una relazione one-to-many tra gli oggetti

Relazioni tra i pattern

mappa concettuale



Relazioni tra i pattern



Vantaggi dei Design Pattern

- ⊕ I Design Pattern permettono il riuso su larga scala di architetture software
- ⊕ Aiutano anche a documentare e a capire il sistema sviluppato
- ⊕ I Pattern

Svantaggi dei Design Pattern

- ⊕ I Pattern non portano al riuso diretto di codice
- ⊕ I Pattern possono essere ingannevolmente semplici
- ⊕ Se si è “nuovi” del settore, si può finire con l’avere troppi pattern
- ⊕ I Pattern sono validati dall’esperienza e dal confronto tra gli utilizzatori, piuttosto che da un testing esaustivo
- ⊕ Integrare i Pattern nel processo di sviluppo di un software richiede una intensa attività umana

Application Framework

- ❖ Un Framework comprende il completo design di una applicazione o di un sottosistema (al contrario del pattern che è solo uno schema di una soluzione di una classe di problemi)
- ❖ Un Framework specifica l'architettura di una applicazione e può essere customizzato per ottenere una applicazione
- ❖ Un Application Framework è uno specifico insieme di classi (spesso basate su Patterns) che cooperano strettamente tra loro e che insieme sono riusabili per una categoria di problemi
- ❖ Quando uso un Framework, riuso la parte principale del framework, e scrivo solo il codice che viene chiamato

Tipi di application frameworks

- ❖ Infrastruttura di sistema
 - ❖ supportano lo sviluppo di infrastrutture, quali comunicazioni, interfacce utente, compilatori, ...
- ❖ Integrazione di middleware
 - ❖ insiemi di standard e classi di oggetti associate che supportano lo scambio di informazioni: CORBA, COM+, Java Beans,
- ❖ Applicazioni aziendali
 - ❖ supportano lo sviluppo di applicazioni per specifici dominii di applicazione integrando conoscenza del dominioç ambito telecomunicazioni, ambito finanziario, ...

Usare i framework

- ❖ I framework sono generici e quindi devono essere “estesi” per essere usati
 - ❖ aggiungere classi concrete che ereditano operazioni dalle classi astratte del framework
 - ❖ aggiungere metodi (callbacks) che vengono chiamati in risposta ad eventi riconosciuti dal framework
- ❖ Il problema dei framework è la loro complessità: significa che serve del tempo per imparare a usarli in modo efficace

Esempio: MVC

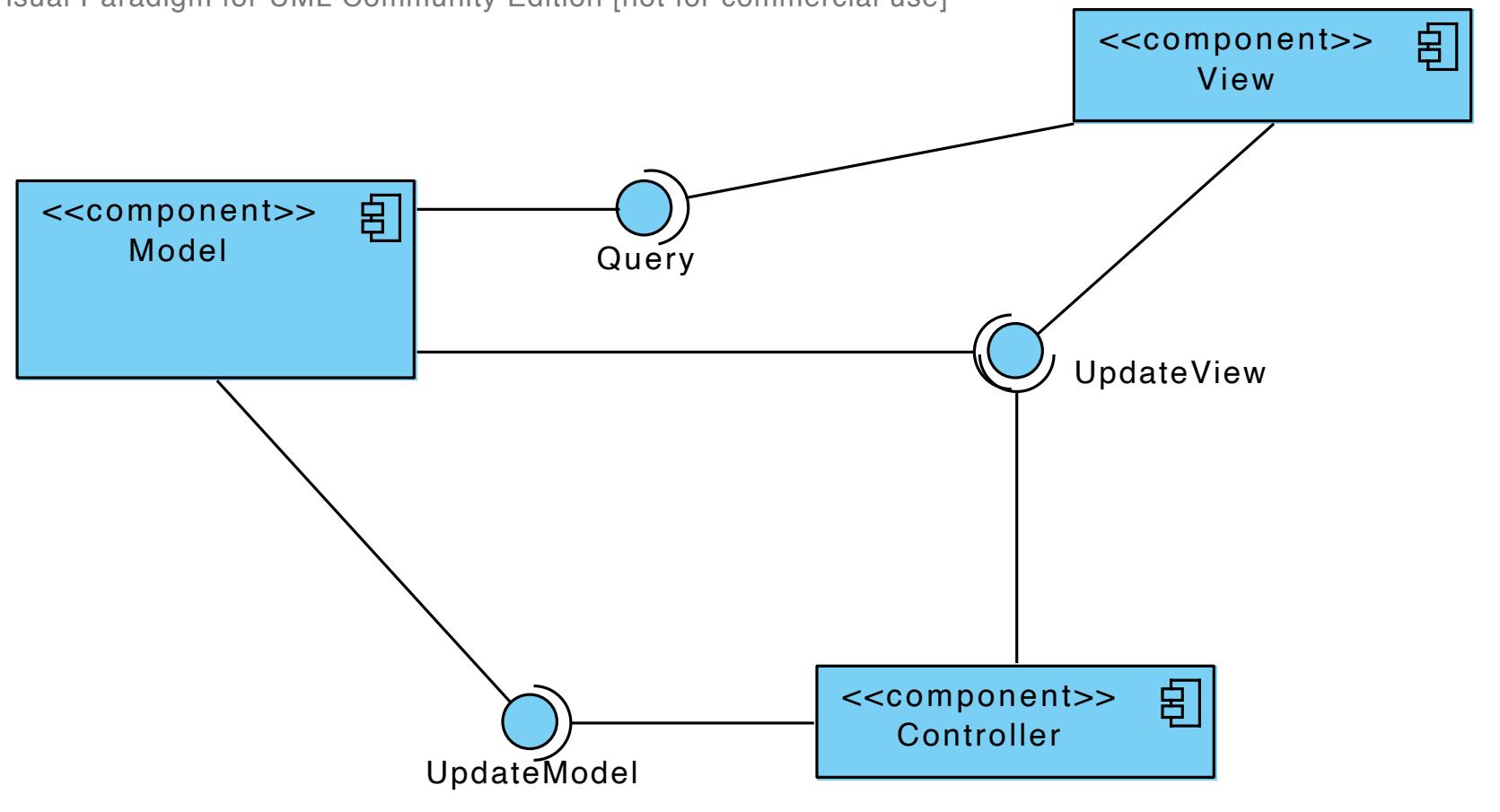
- ❖ Usato nella progettazione di GUI
- ❖ Problema: consentire la presentazione multipla di un oggetto e stili di interazione separati per ogni interazione (è il caso di tutti gli strumenti interattivi come wordprocessor, spreadsheet, grafica, ...)
 - ❖ quando i dati sono modificati attraverso una delle presentazioni, tutte le altre presentazioni sono aggiornate
- ❖ Data la definizione di framework, questi sono spesso istanziazioni di una serie di pattern: in questo caso dei pattern Observer, Strategy, Composite, ...

Esempio: MVC

- ❖ Non ha senso duplicare i dati in funzione della view che viene mostrata, anzi è meglio ci sia una unica copia dei dati da cui ricavare di volta in volta le varie view
- ❖ Model
 - ❖ componente che contiene tutti i dati gestiti dall'applicazione
- ❖ View
 - ❖ componente che ha il compito di mostrare i dati; possono essere più componenti di questo tipo (in generale uno per view)
- ❖ Controller
 - ❖ componente che riceve le richieste da parte dell'utente e le traduce in operazioni di modifica del modello

Esempio: MVC

Visual Paradigm for UML Community Edition [not for commercial use]



Esempio: MVC

- ❖ Due modi per aggiornare le view (modello semplice)
 - ❖ passivo: è l'utente che chiede di aggiornare le view (Query)
 - ❖ attivo: il modello invia una richiesta (UpdateView) alle view ogni volta che c'è un cambiamento nei dati del modello stesso
- ❖ Le operazioni di modifica dei dati sono generate dall'utente tramite il controller: da un lato aggiorna il modello (UpdateModel) e dall'altro provoca l'aggiornamento delle view (UpdateView)