Metodologie di Programmazione

Lezione 30: Design pattern (parte 2)

Lezione 30: Sommario

UNITELMA SAPIENZA

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

- Observer pattern
- Factory pattern

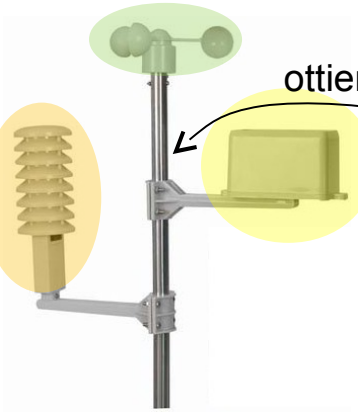
Un secondo caso concreto: monitoraggio del tempo

- Supponiamo di dover costruire una stazione di monitoraggio del tempo basata su Internet
- Basata sul nostro tipo TempoAggiornato che tiene traccia delle condizioni di tempo
  - Temperatura
  - Umidità
  - Pressione
- Vogliamo un'applicazione che visualizzi in tempo reale:
  1. Le attuali condizioni
  2. Statistiche sul tempo
  3. Una semplice previsione del tempo
- Vogliamo rilasciare un'API per permettere ad altri sviluppatori di scrivere i loro programmi basati sui nostri dati (pagando per ogni chiamata all'API!)

Schema dell'applicazione di monitoraggio del tempo

Dati forniti dalla stazione del tempo

Ciò che **NOI** implementiamo



stazione del tempo

ottiene dati da

Tempo Aggiornato

visualizza

dispositivo con display

Come realizzare la classe TempoAggiornato?

- Implementiamo 4 metodi:

TempoAggiornato

getTemperatura();
getUmidità();
getPressione();
misurazioniCambiate();

- I primi tre metodi ("**getter**") restituiscono le misurazioni più recenti per temperatura, umidità e pressione
- La classe sa come aggiornarle
- Ogni volta che vengono aggiornate le misurazioni viene chiamato il metodo misurazioniCambiate
  - Così possiamo aggiornare lo schermo del visualizzatore

Una prima implementazione di TempoAggiornato


```
public class TempoAggiornato
{
    Display displayCondizioniAttuali;
    Display displayStatistiche;
    Display displayPrevisioni;

    public void misurazioniCambiate()
    {
        double temp = getTemperatura();
        double umidita = getUmidita();
        double press = getPressione();

        displayCondizioniAttuali.aggiorna(temp, umidita, press);
        displayStatistiche.aggiorna(temp, umidita, press);
        displayPrevisioni.aggiorna(temp, umidita, press);
    }

    // altri metodi
    // ...
}
```

ottiene le misurazioni più recenti

aggiorna i tre display

Tuttavia l'interfaccia sembra uguale per tutti i display!

Stiamo codificando implementazioni **CONCRETE**
(nessun modo di aggiungerne altre senza cambiare il programma)

Creare un meccanismo flessibile

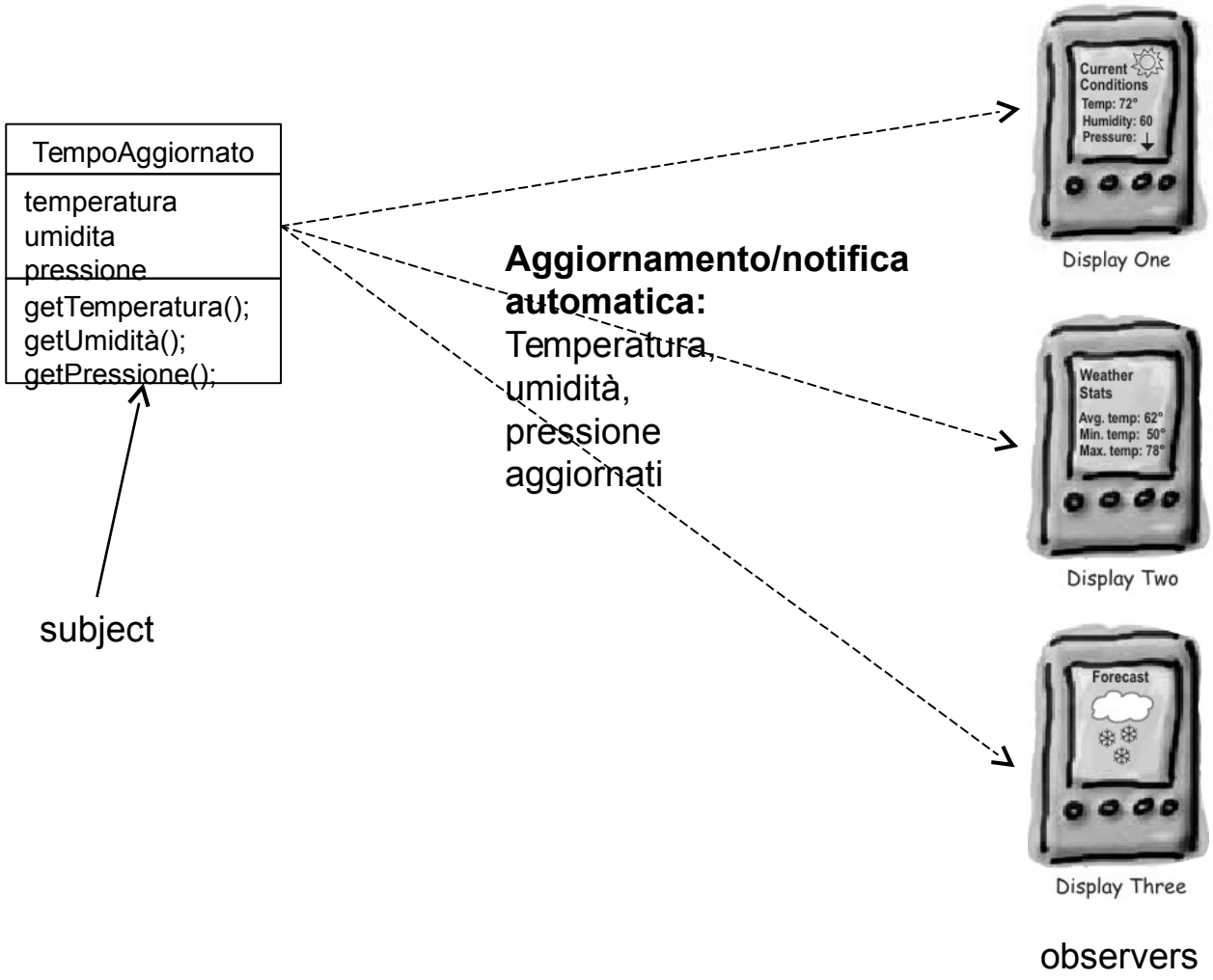
- Come possiamo rendere questo meccanismo flessibile in modo da non dover "cablare" l'elenco dei dispositivi che vogliono ricevere le informazioni aggiornate?

Possiamo utilizzare l'Observer Pattern

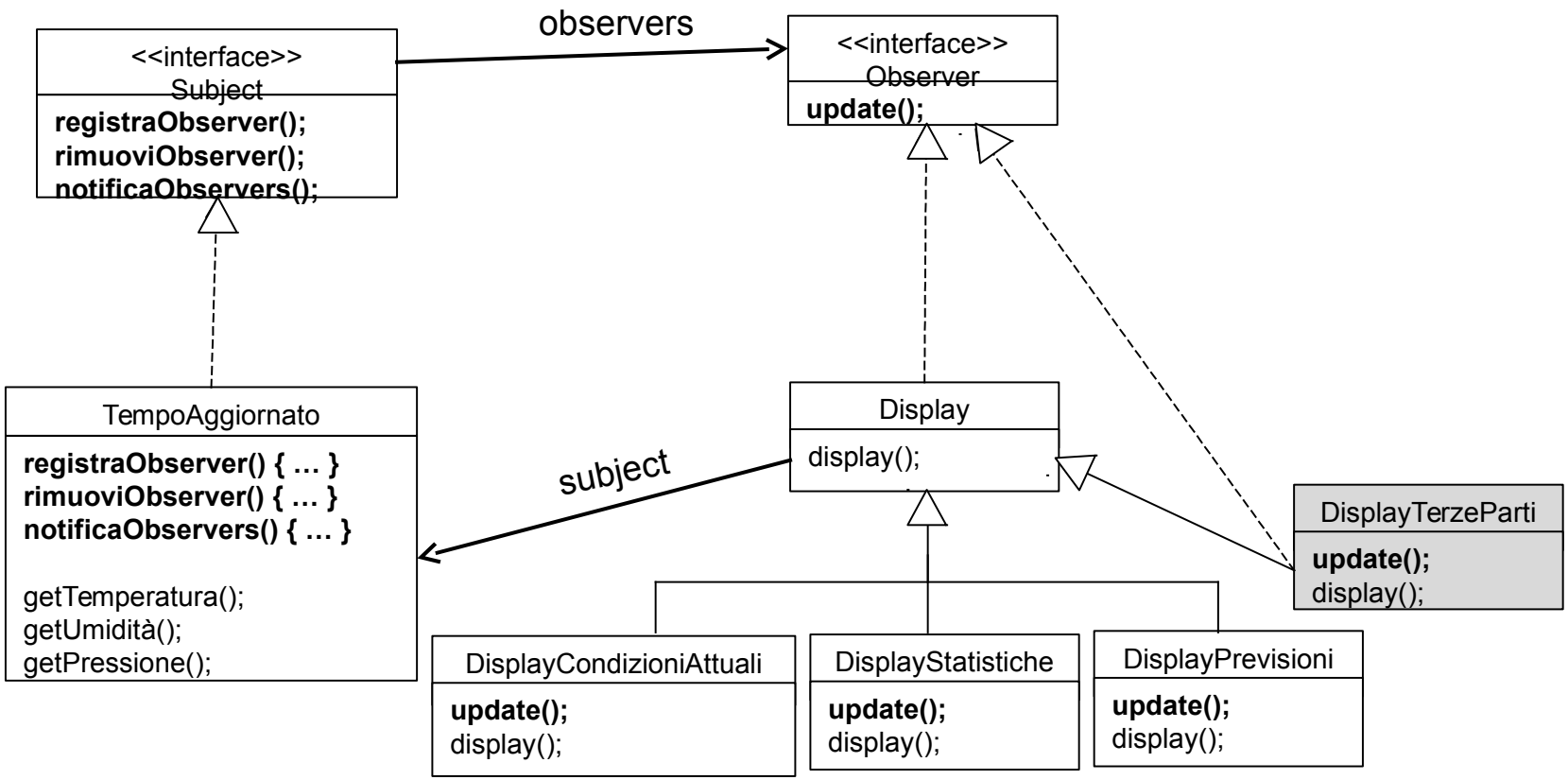
Come l'abbonamento a un settimanale:

1. Panini comics stampa Topolino
2. Ti abboni a un **determinato** settimanale e, ogni volta che viene stampata una nuova edizione, ti viene consegnata
3. Cancelli l'abbonamento quando non vuoi più ricevere le nuove copie
4. L'editore rimane nel business: altre persone continuano ad abbonarsi e a cancellare l'abbonamento

Publisher + Subscriber = Observer Pattern



Publisher + Subscriber = Observer Pattern



Sporchiamoci le mani: le interfacce


```
public interface Subject
{
    void registraObserver(Observer o);
    void rimuoviObserver(Observer o);
    void notificaObservers();
}

public interface Observer
{
    void update(double temp, double umidita, double press);
}
```

Sporchiamoci le mani: le implementazioni (1)


```
public class TempoAggiornato implements Subject
{
    private double temperatura;
    private double umidita;
    private double pressione;
    private ArrayList<Observer> observers = new ArrayList<Observer>();

    public void misurazioniCambiate()
    {
        notificaObservers();
    }

    @Override
    public void registraObserver(Observer o)
    {
        observers.add(o);
    }

    @Override
    public void rimuoviObserver(Observer o)
    {
        observers.remove(o);
    }

    @Override
    public void notificaObservers()
    {
        for (Observer o : observers) o.update(temperatura, umidita, pressione);
    }

    // altri metodi qui
}
```

Sporchiamoci le mani: le implementazioni (2)


```
abstract public class Display implements Observer
{
    abstract public void display();
}

public class DisplayCondizioniAttuali extends Display
{
    private double temperatura;
    private double umidita;
    private double pressione;
    private Subject subject;

    public DisplayCondizioniAttuali(Subject subject)
    {
        this.subject = subject;
        subject.registraObserver(this);
    }

    public void display()
    {
        System.out.println("Condizioni attuali: temp="+temperatura
                +", umidità="+umidita+", pressione = "+pressione);
    }

    public void update(double temp, double umidita, double press)
    {
        this.temperatura = temp;
        this.umidita = umidita;
        this.pressione = press;
        display();
    }
}
```

C

In realtà sono già implementati in Java!

- Mediante la classe java.util.Observable:

Method Summary	
void	addObserver(Observer o) Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
protected void	clearChanged() Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the hasChanged method will now return false.
int	countObservers() Returns the number of observers of this Observable object.
void	deleteObserver(Observer o) Deletes an observer from the set of observers of this object.
void	deleteObservers() Clears the observer list so that this object no longer has any observers.
boolean	hasChanged() Tests if this object has changed.
void	notifyObservers() If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.
void	notifyObservers(Object arg) If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.
protected void	setChanged() Marks this Observable object as having been changed; the hasChanged method will now return true.

Method Summary	
void	update(Observable o, Object arg) This method is called whenever the observed object is changed.

Che principio abbiamo utilizzato?

Principio di design:
progetta in modo da accoppiare gli oggetti che interagiscono tra loro

- L'Observer Pattern permette di comunicare uno stato o un evento a un insieme di oggetti in modo il più possibile svincolato

Esercizio: Borsa (1)

- Si considerino le seguenti cinque classi:

```
public class Borsa
{
    private Analizzatore analizzatoreStatistiche;
    private Analizzatore analizzatoreTrend;

    // chiamato ogni volta che una nuova transazione è disponibile
    public void nuovaTransazione(Transazione t)
    {
        analizzatoreStatistiche.analizza(t);
        analizzatoreTrend.analizza(t);
    }
}
```

```
abstract public class Analizzatore
{
    abstract public void analizza(Transazione t);
}
```

```
public class AnalizzatoreStatistiche extends Analizzatore
{
    private List<Transazione> transazioni = new ArrayList<Transazione>();

    @Override
    public void analizza(Transazione t)
    {
        transazioni.add(t);
    }

    @Override
    public String toString()
    {
        double importo = 0.0;
        for (Transazione t : transazioni) importo += t.getImporto();
        return "IMPORTO MEDIO DELLE TRANSAZIONI = "+importo/transazioni.size();
    }
}
```

Esercizio: Borsa (2)


```
public class AnalizzatoreTrend extends Analizzatore
{
    private Transazione ultimaTransazione;
    private double trendImporto;

    @Override
    public void analizza(Transazione t)
    {
        if (ultimaTransazione == null) trendImporto = t.getImporto();
        else trendImporto = t.getImporto()-ultimaTransazione.getImporto();

        ultimaTransazione = t;
    }

    @Override
    public String toString()
    {
        return "TREND = "+trendImporto;
    }
}
```

```
public class Transazione
{
    private String compagnia;
    private double importo;

    public Transazione(String compagnia, double importo)
    {
        this.compagnia = compagnia;
        this.importo = importo;
    }

    public String getCompagnia() { return compagnia; }
    public double getImporto() { return importo; }
}
```

- Si mostrino le criticità e si ristrutturi il codice utilizzando il pattern Observer

Creare oggetti

- Finora siamo stati abituati a creare oggetti mediante la parola chiave new

```
Anatra anatra = new AnatraDiPlastica();
```

Codice flessibile grazie al polimorfismo

Istanza di una classe concreta

- Tuttavia se fino a tempo di esecuzione **non sappiamo** quale anatra istanziare, dobbiamo scrivere qualcosa del genere:

```
if (diPlastica) anatra = new AnatraDiPlastica();
else if (siPesca) anatra = new AnatraEsca();
else if (aCasa) anatra = new AnatraDomestica();
```

Ricorda: separa ciò che cambia da ciò che rimane uguale

Principio di design: identifica gli aspetti della tua applicazione che variano e separali da quelli che rimangono uguali

- In questa situazione ciò che cambia è la classe concreta da istanziare

La classe Pizzeria

UNITELMA SAPIENZA

```
public class Pizzeria
{
    public Pizza ordinaPizza(String nome)
    {
        Pizza pizza = null;

        // istanzia la classe sulla base del tipo concreto
        // passato in input come stringa
        if (nome.equalsIgnoreCase("margherita"))
            pizza = new PizzaMargherita();
        else if (nome.equalsIgnoreCase("capricciosa"))
            pizza = new PizzaCapricciosa();
        else if (nome.equalsIgnoreCase("funghi"))
            pizza = new PizzaFunghi();
        else if (nome.equalsIgnoreCase("vegetariana"))
            pizza = new PizzaVegetariana();

        // queste operazioni sono uguali per tutti:
        // ma ogni pizza sa come "prepararsi"
        pizza.impasta();
        pizza.cuoci();
        pizza.taglia();
        pizza.inscatola();

        return pizza;
    }
}
```

Ma se voglio aggiungere un'altra pizza? Devo aggiornare il codice!
MALE, MOLTO MALE!

Dobbiamo incapsulare la creazione degli oggetti

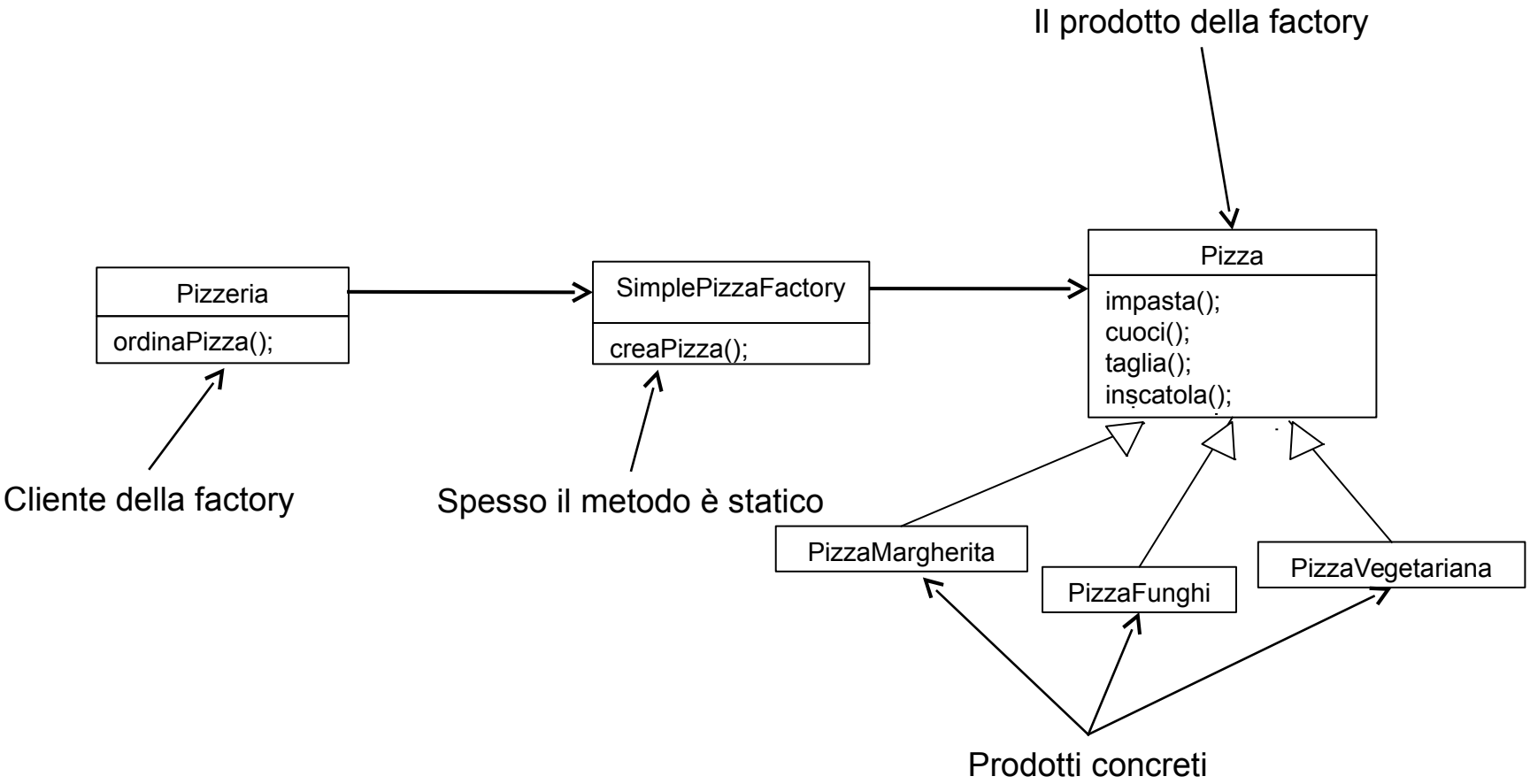
- Mediante un nuovo oggetto che chiamiamo Factory
- Factory si occupa della creazione di oggetti di un certo tipo:

```
public class PizzaFactory
{
    public Pizza creaPizza(String nome)
    {
        Pizza pizza = null;

        // istanzia la classe sulla base del tipo concreto
        // passato in input come stringa
        if (nome.equalsIgnoreCase("margherita"))
            pizza = new PizzaMargherita();
        else if (nome.equalsIgnoreCase("capricciosa"))
            pizza = new PizzaCapricciosa();
        else if (nome.equalsIgnoreCase("funghi"))
            pizza = new PizzaFunghi();
        else if (nome.equalsIgnoreCase("vegetariana"))
            pizza = new PizzaVegetariana();

        return pizza;
    }
}
```

Migliorabile con la Reflection!

```
public class Pizzeria
{
    private PizzaFactory pizzaFactory;

    public Pizzeria(PizzaFactory pizzaFactory)
    {
        this.pizzaFactory = pizzaFactory;
    }

    public Pizza ordinaPizza(String nome)
    {
        Pizza pizza = pizzaFactory.creaPizza(nome);

        // queste operazioni sono uguali per tutti:
        // ma ogni pizza sa come "prepararsi"
        pizza.impasta();
        pizza.cuoci();
        pizza.taglia();
        pizza.inscatola();

        return pizza;
    }
}
```

Il Simple Factory Pattern



E se avviassimo un “franchising” della nostra Pizzeria?

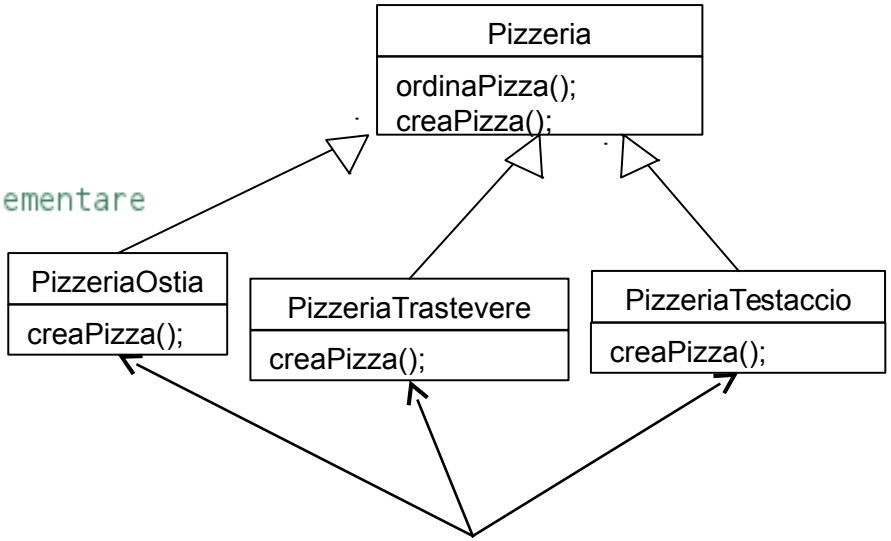
- Le diverse pizzerie si troveranno in posti diversi
- Ognuna produrrà la propria pizza
  - TrasteverePizzaFactory
  - OstiaPizzaFactory
  - TestaccioPizzaFactory
- **No problem!** La passiamo in input alla Pizzeria:

```
Pizzeria laMiaPizzeriaDiOstia = new Pizzeria(new OstiaPizzaFactory());
Pizza margherita = laMiaPizzeriaDiOstia.ordinaPizza("margherita");
```

- Tuttavia vorresti **vincolare** una factory a un determinata pizzeria
  - Altrimenti tutti (!) useranno il pizzaiolo di Ostia (il più bravo…)

Un framework generale per la Pizzeria


```
abstract public class Pizzeria
{
    public Pizza ordinaPizza(String nome)
    {
        // chiama creaPizza: metodo astratto da implementare
        Pizza pizza = creaPizza(nome);

        // queste operazioni sono uguali per tutti:
        // ma ogni pizza sa come "prepararsi"
        pizza.impasta();
        pizza.cuoci();
        pizza.taglia();
        pizza.inscatola();

        return pizza;
    }

    // ogni pizzeria "concreta" sara' costretta a implementare creaPizza
    // "a modo suo" (pizza più bruciacchiata, pizza più alta, ecc.)
    abstract protected Pizza creaPizza(String nome);
}
```

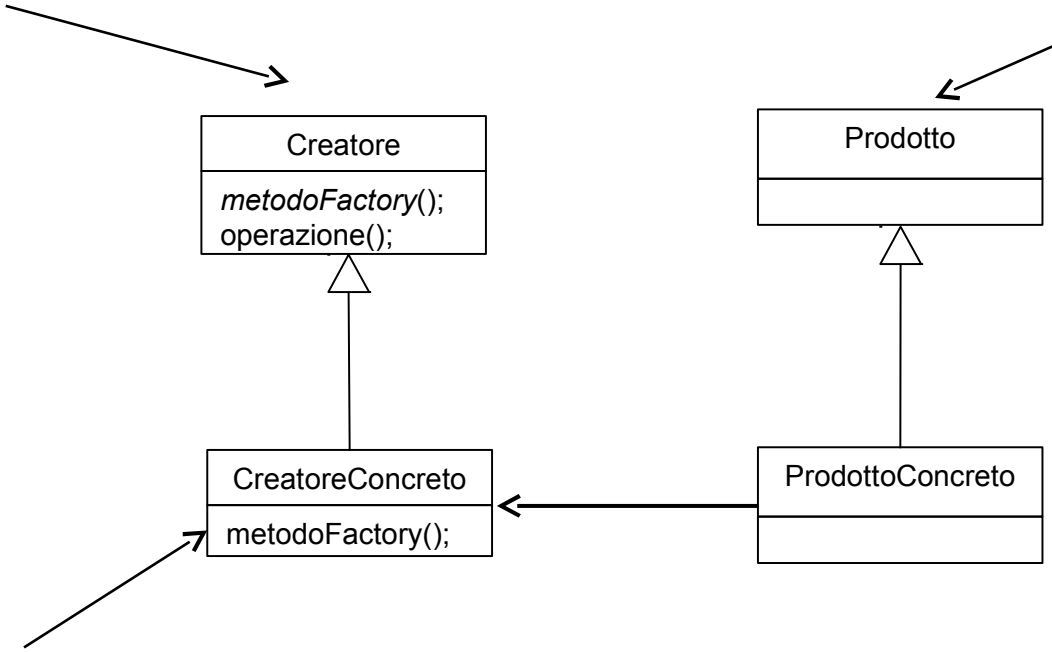


La pizzeria concreta è diventata la specifica factory

Il Factory Pattern

La classe che contiene
Le implementazioni per tutti i metodi che manipola prodotti, eccetto il metodo factory

Tutti i prodotti implementano la stessa interfaccia disponibile nella classe astratta



L'unica classe che ha la conoscenza per costruire i prodotti concreti

Che principio abbiamo utilizzato?

Principio di design:
Nel costruire oggetti, lasciando alle sottoclassi il compito di decidere quale classe istanziare

- Il Factory Pattern permette a una classe di differire l'istanziazione alle sottoclassi

Esercizio: Scrittori di libri (1)

- Si considerino le seguenti classi:

```
public class Scrittore
{
    public enum GenereLibro
    {
        AVVENTURA, GIALLO, FUMETTO
    }

    public Libro pubblica(GenereLibro genere)
    {
        Libro libro = null;

        switch(genere)
        {
            case AVVENTURA: libro = new LibroAvventura(); break;
            case GIALLO: libro = new LibroGiallo(); break;
            case FUMETTO: libro = new Fumetto(); break;
        }

        libro.impagina();

        return libro;
    }
}
```

```
public class ScrittoreDiGialli extends Scrittore
{

}
```

```
public class Fumettista extends Scrittore
{

}
```

Esercizio: Scrittori di libri (2)


```
abstract public class Libro
{
    public void impagina() { /* ... */ }
}

public class LibroGiallo extends Libro
{
}

public class LibroAvventura extends Libro
{
}

public class Fumetto extends Libro
{
}
```

- Si mostrino le criticità e si ristrutturi il codice utilizzando il pattern Factory