Programmazione a Oggetti
Modulo B

Lezione 5

Dott. Alessandro Roncato

17/02/2014

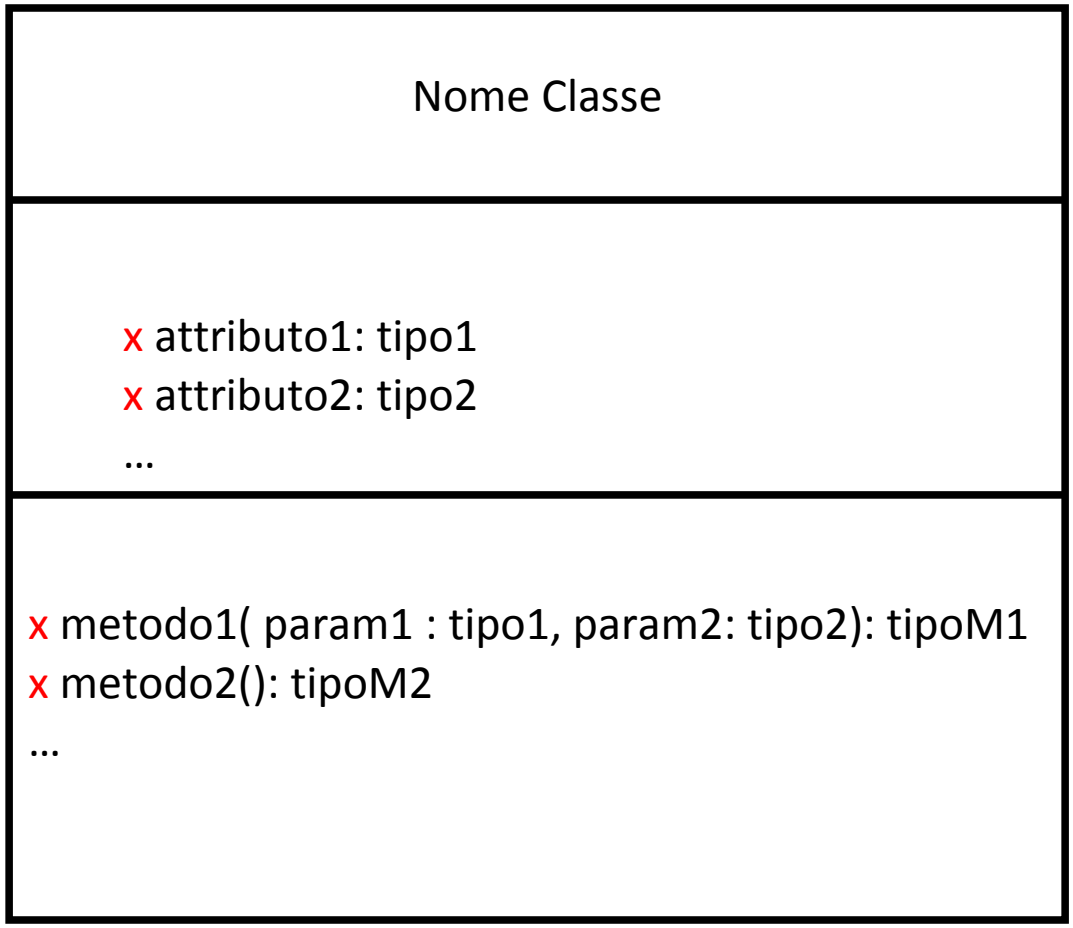
Riassunto

Pattern Null Object

Pattern Singleton

Esempi su come ridurre dipendenza

Diagramma classi



Visibilità

X=

+ public

- private

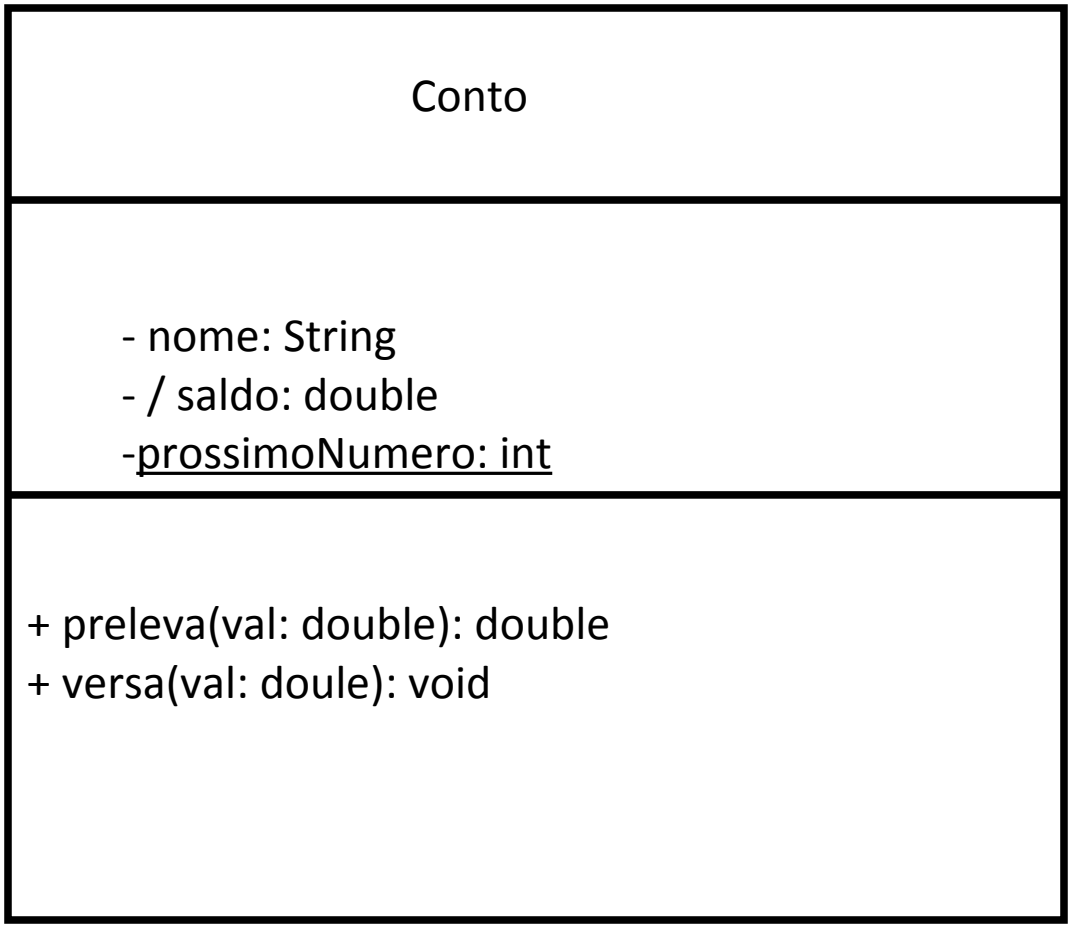
protected

~ package

/ derivate

<u>static</u>

Diagramma classi



Ricerca di un cliente

La responsabilità di gestire la ricerca può essere gestita in due modi:

1) da uno (o più) metodo(i) di una classe esistente
2) da una classe a se stante.

Classe esistente

- Un oggetto “globale” (visibile globalmente) gestisce la ricerca
- Per esempio l'oggetto Banca può gestire la ricerca
- Problemi :

1) bassa coesione

2) una ricerca per ogni oggetto a cui è associata (potrebbero essere poche o troppe!). Se associata al Banca, abbiamo un'unica ricerca.

Esempio

```
public class Banca {
  Set<Cliente> risultatoRicerca;
  ...
  public Banca(){
    //inizializzazione della banca
    // +
    // inizializzazione ricerca
  }
  //metodi Banca più metodi ricerca
  public void cercaClienti(String query){...}

  public Set<Client> getRicerca(){
         return risultatoRicerca();}

  public void raffinaRicerca(String query){...}
}
```

Bassa coesione
Difficile il riuso
Alta dipendenza

Svantaggi

- Bassa coesione: cosa hanno a che fare i metodi di Banca con quelli della ricerca dei Clienti?
- Riuso difficile: come posso riusare le funzionalità della ricerca senza la classe Banca?
- Dipendenza: aumentano le classi che hanno bisogno di accedere alla classe Banca (in più chi deve accedere alle ricerche)

Pure Fabrication

P: come assegnare le responsabilità in modo da ottenere Low Coupling and High Coesion se I.E. NON è appropiato?

S: assegna un insieme altamente coeso di responsabilità a una classe artificiale (di pura invenzione) che non rappresenta un concetto del dominio del problema, ma qualcosa di inventato per sostenere H.C., L.C. e il riuso.

Esempio

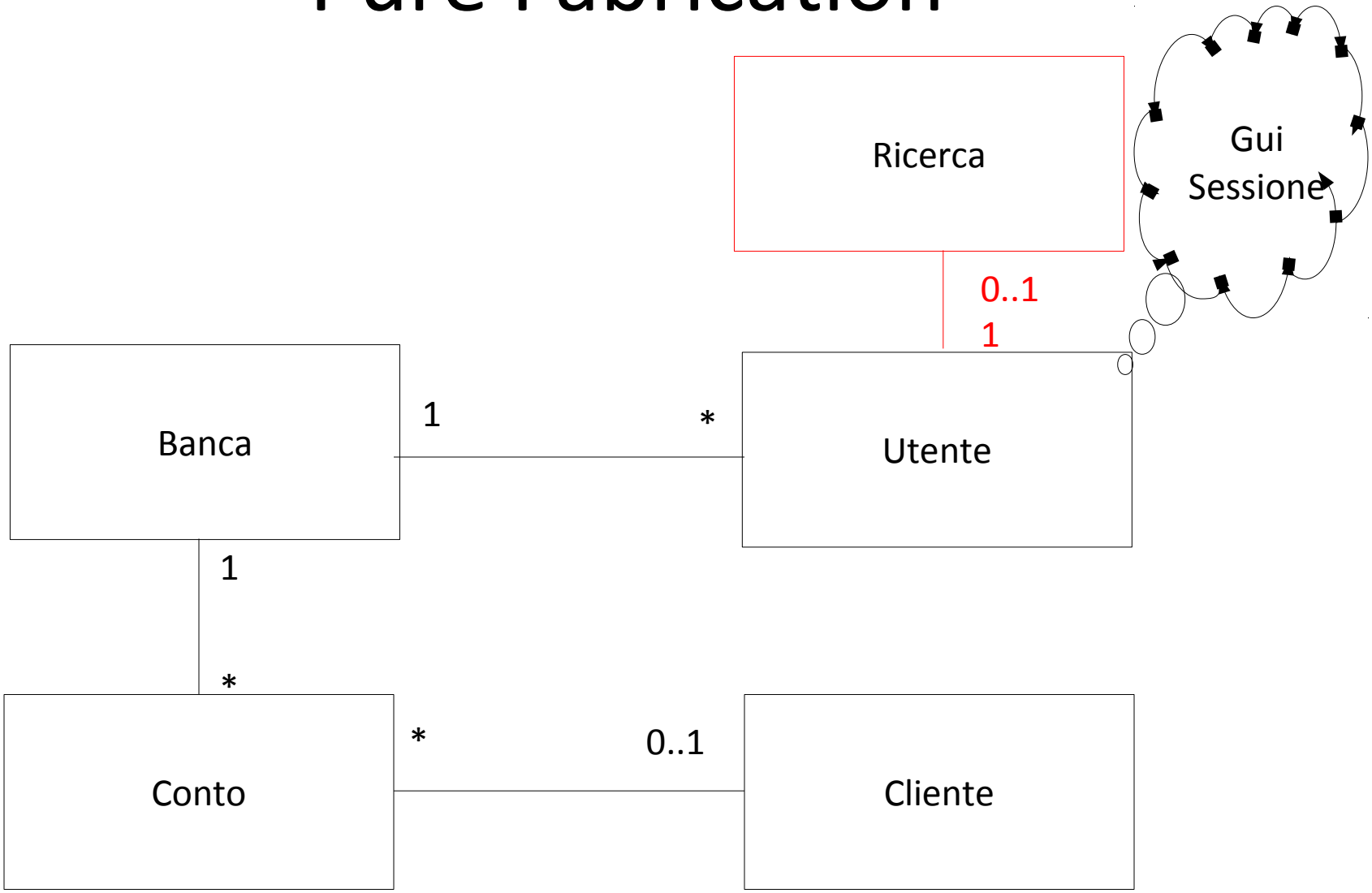
- Facciamo gestire le ricerche da una nuova classe inventata “Ricerca”
- (Questa classe non è proprio una pura invenzione perché come concetto la Ricerca esiste)
- Vedremo poi altri esempi in cui inventiamo nuove classi più di sana pianta

Esempio

```
public class Ricerca {
   Set<Clienti> risultato;

   public void cercaClienti(String query){...}

   public Set getRicerca(){return risultatoRicerca();

   public void raffinaRicerca(String query){...}


    }
}
```

Alta coesione
Riuso
Riduce dipendenza

Pure Fabrication



Altri esempi

- Chi è responsabile della “persistenza” degli oggetti?
- Persistenza=salvare, leggere, aggiornare, cercare gli oggetti nel filesystem o in un DB
- Per I.E. Dovrebbe essere l'oggetto stesso a farlo
- Ma questo ha gli svantaggi già visti
- Si preferisce una Pure Fabrication che gestisca la persistenza di tutti gli oggetti

Altri esempi

- Chi è responsabile della di gestire la visualizzazione?
- Per I.E. Dovrebbe essere l'oggetto stesso a farlo
- Ma questo ha gli svantaggi già visti
- Si preferisce una Pure Fabrication che gestisca la visualizzazione degli oggetti

Svantaggi Pure Fabrication

- Una classe in più
- Potrebbe essere difficile da individuare la classe responsabile (usare nomi autoesplicativi)
- Visione procedurale invece che ad oggetti (raggruppo per funzionalità e non per concetto)

  (vedremo come limitare questo ultimo problema)

Chi crea gli oggetti letti dal DB ?

```
public class Conto {
   static public Conto load(int id){
    Class.classForName("Driver");
    Connection con=DriverManager.get.Conn...
    Statement sta = con.createStatement();
    ResultSet rs = sta.executeQuery("...
    if (rs.next){
        int numero=rs.getString("numero");
        …
        return new Conto(numero,...
    }
    return null;
   }
}
```

Chi crea gli oggetti letti dal DB ?

```
public class Cliente {
   static public Cliente load(int id){
    Class.classForName("Driver");
    Connection con=DriverManager.get.Conn...
    Statement sta = con.createStatement();
    ResultSet rs = sta.executeQuery("...
    if (rs.next){
        String nome=rs.getString("nome");
        …
        return new Cliente(nome,...
    }
    return null;
   }
}
```

Svantaggi

- Logica di creazione complessa
- Poca coesione
- Forte dipendenza degli oggetti del modello da le classi di gestione del DB
- Ripetizione di codice in classi diverse

  (vedi esercizio più avanti)

Soluzione

- Specializzazione di Pure Fabrication: Factory (o anche detto Simple Factory o Concrete Factory)
- Un oggetto di pura invenzione crea gli oggetti del modello

Attenzione:

- Factory=Fabbrica
- Fabrication=Invenzione

Pattern Factory

P: chi deve creare gli oggetti quando la creazione è complessa?

S: Un oggetto di pura invenzione chiamato Factory

Factory

```
public class ContoFactory {
  public Conto load(int id){
   Class.classForName("Driver");
   Connection con=DriverManager.get.Connection(...);
   Statement sta = con.createStatement();
   ResultSet rs = sta.executeQuery("...");
   if (rs.next){
       int numero=rs.getString("numero");
       …
       return new Conto(numero,...);
   }
 }
}
```

Factory

```
public class ClienteFactory {
  public Cliente load(int id){
   Class.classForName("Driver");
   Connection con=DriverManager.get.Connection(...);
   Statement sta = con.createStatement();
   ResultSet rs = sta.executeQuery("...");
   if (rs.next){
       String nome=rs.getString("nome");
       …
       return new Cliente(nome,...);
   }
  }
}
```

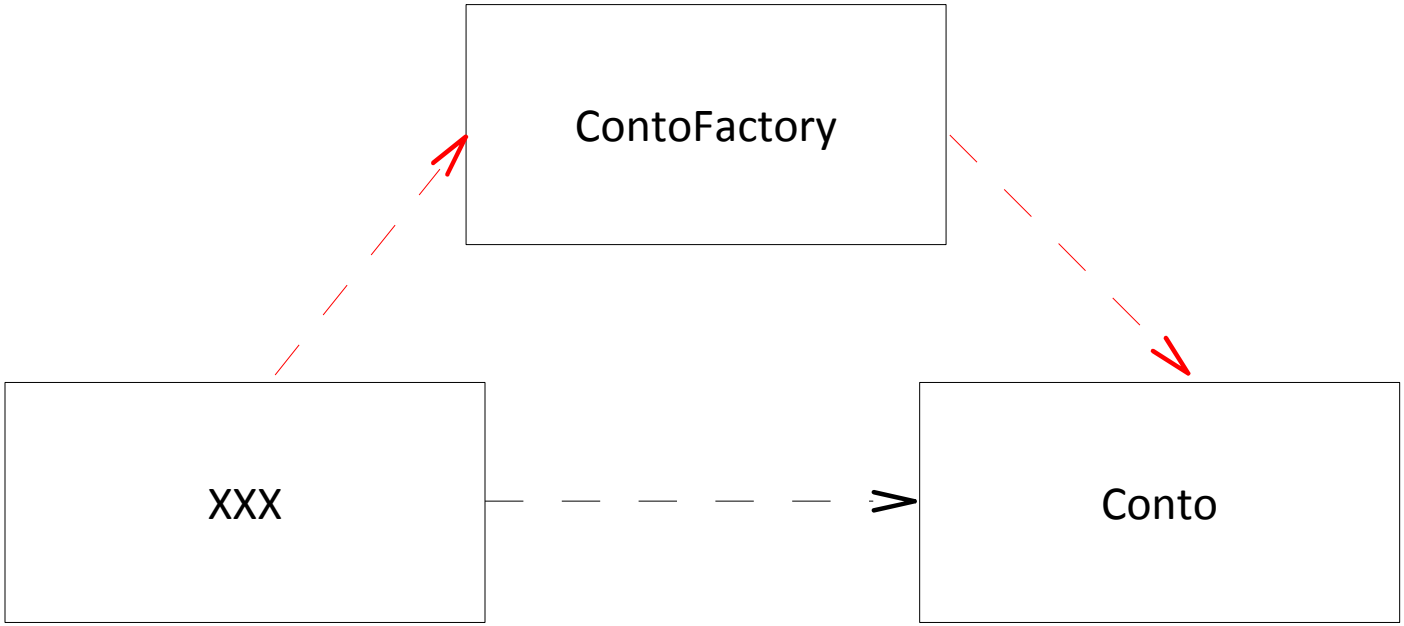
Uso Factory

```
public class XXX {


   public qualcheMetodo() {

      Conto conto=...load(id);
     //cosa metto al posto dei ...?
     //quante Factory sono necessarie?
     //che visibilità devono avere?
     //vedi esercizio
  }
}
```

Considerazioni

- Una classe in più per ogni classe del Modello
- Più coeso
- Facile il riuso?

Factory



Esercizi

- Cosa metto al posto dei puntini `...`?
- Che vantaggi e svantaggi ci sono?

- Come posso ottenere alta coesione e bassa dipendenza usando `static` invece che Factory?
- Che vantaggi e svantaggi ci sono?

Factory Method

- Nell'esempio di Simple Factory abbiamo visto che la Factory definisce il metodo `load` che è un esempio Factory Method.
- Possibilità di avere nomi significativi per i metodi (es. loadFromDb, loadFromFile, createNew) (invece il costruttore ha un nome imposto)

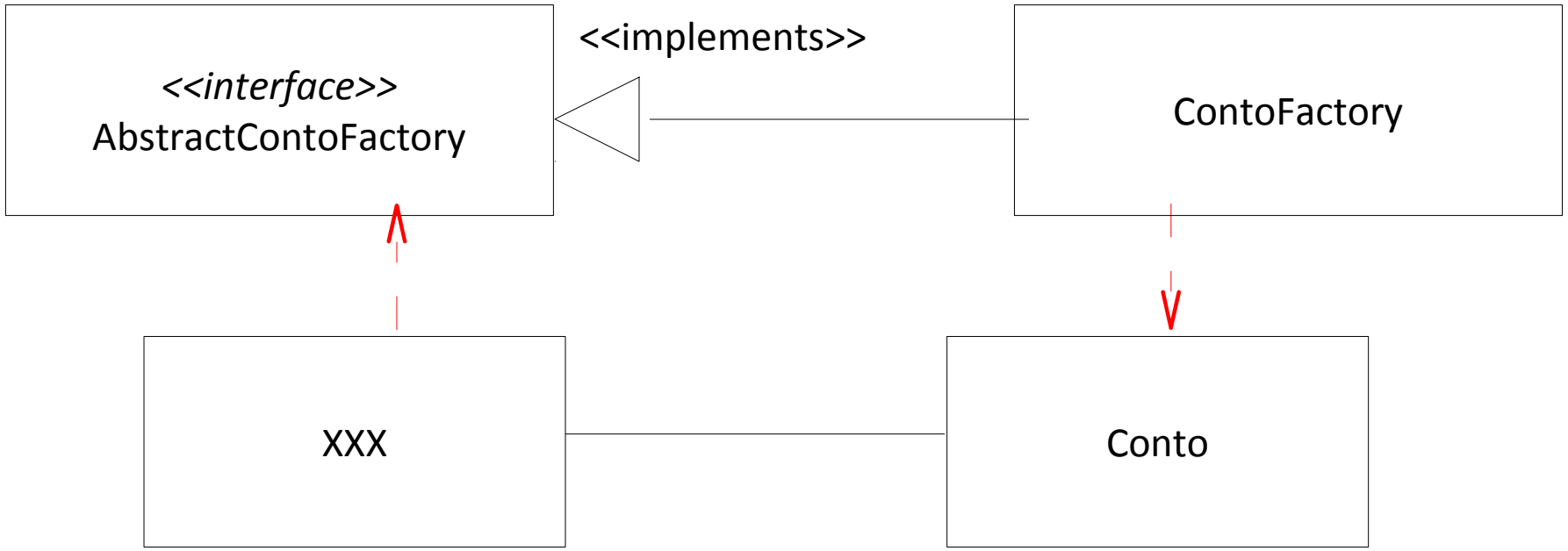
Factory method

```
public class Conto {

   private Conto() {
      …
  }
  public static Conto createNew(int numero){
   Conto res = new Conto();
   res.numero=numero;
   return res;
 }
  public static Conto loadFromDB(int id){
   ...//come già visto
 }
}
```

Abstract Factory

- A differenza del Simple Factory che è una classe, l'Abstrac Factory definisce un'interfaccia
- Ci vogliono quindi almeno una classe e un'interfaccia (prima avevo un'unica classe)
- L'interfaccia definisce almeno un factory method che poi verrà implementato da una o più classi

Abstract Factory



Abstract Factory

```
public interface AbstractContoFactory {

public Conto createNew(int numero);
public Conto loadFromDB(int id);

}
```

Abstract Factory

```
public class ContoFactory implements
AbstractContoFactory
{
  public Conto createNew(int n){
    ...
  }
  public Conto loadFromDB(int id){
  …
  }
}
```

Uso Abstract Factory

```
public class XXX {


   public void qualcheMetodo() {

      Conto conto=...createNew(int i);
     //cosa metto al posto dei ...?
     //la differenza rispetto a Simple Factory
     //dipende da cosa metto nei puntini

   }
  }
}
```

Esercizio

- Fare in modo di usare Polimorfismo e Factory in modo che il codice di accesso al DB venga riusato quanto più possibile.
- Nella applicazioni reali si usa anche la Reflection in modo da evitare addirittura la necessità delle sottoclassi e che Factory dipenda dagli oggetti della nostra applicazione

domande