Programmazione a Oggetti
Modulo B

Lezione 11

Dott. Alessandro Roncato

11/03/2013

Riassunto

Diagramma classi

Polimorfismo

Interfacce

Ereditarietà multipla

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 negozio
    // piu'
    // 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

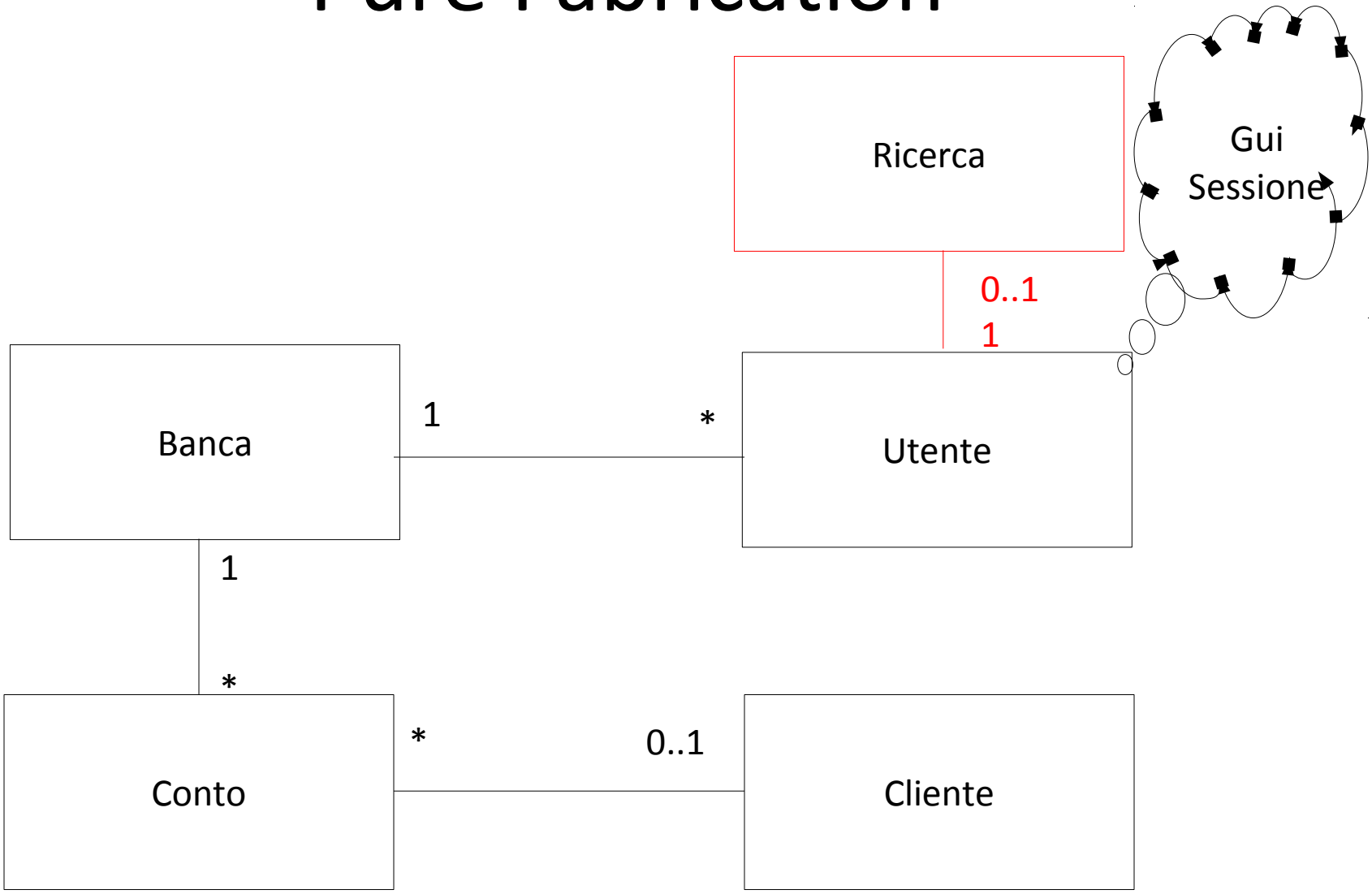
- Facciamo gestire le ricerche da una nuova classe inventata “Ricerca”
- (Questa classe non è proprio una pura invenzione perché come concetto la Ricerca esiste)
- Vedremo poi altri esempi in cui inventiamo nuove classi più di sana pianta

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,...
   }
}
```

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,...
   }
}
```

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,...);
   }
  }
}
```

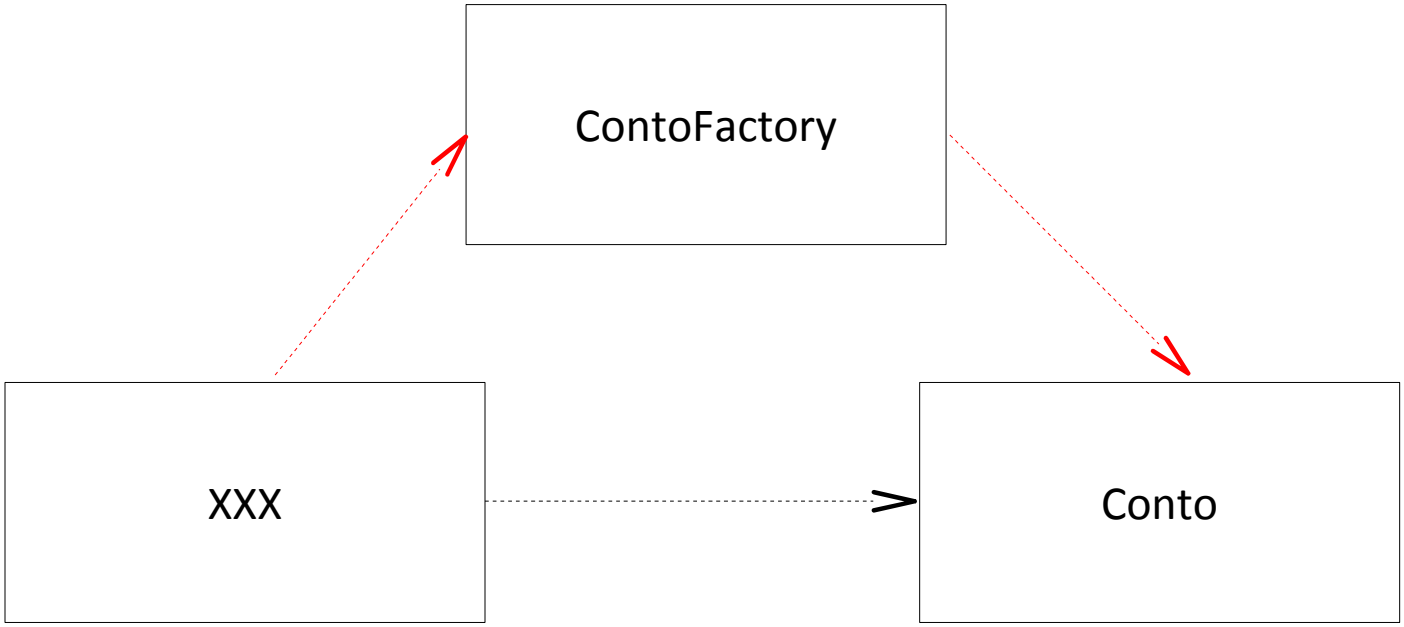
Uso Factory

```
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)

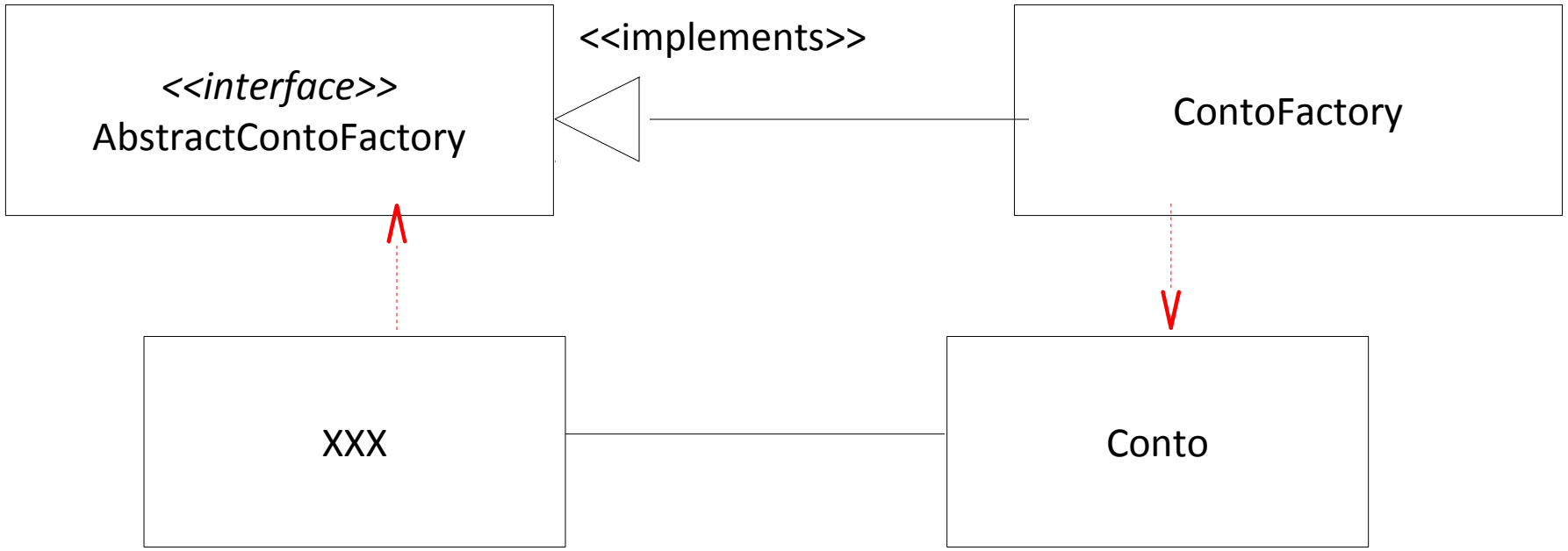
Factory method

```
public class Conto {

   private Conto() {
      …
  }
  public static Conto createNew(){
   return new Conto();
 }
  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();

}
```

Abstract Factory

```
public class ContoFactory implements
AbstractContoFactory
{
  public Conto createNew(){
    ...

  }
}
```

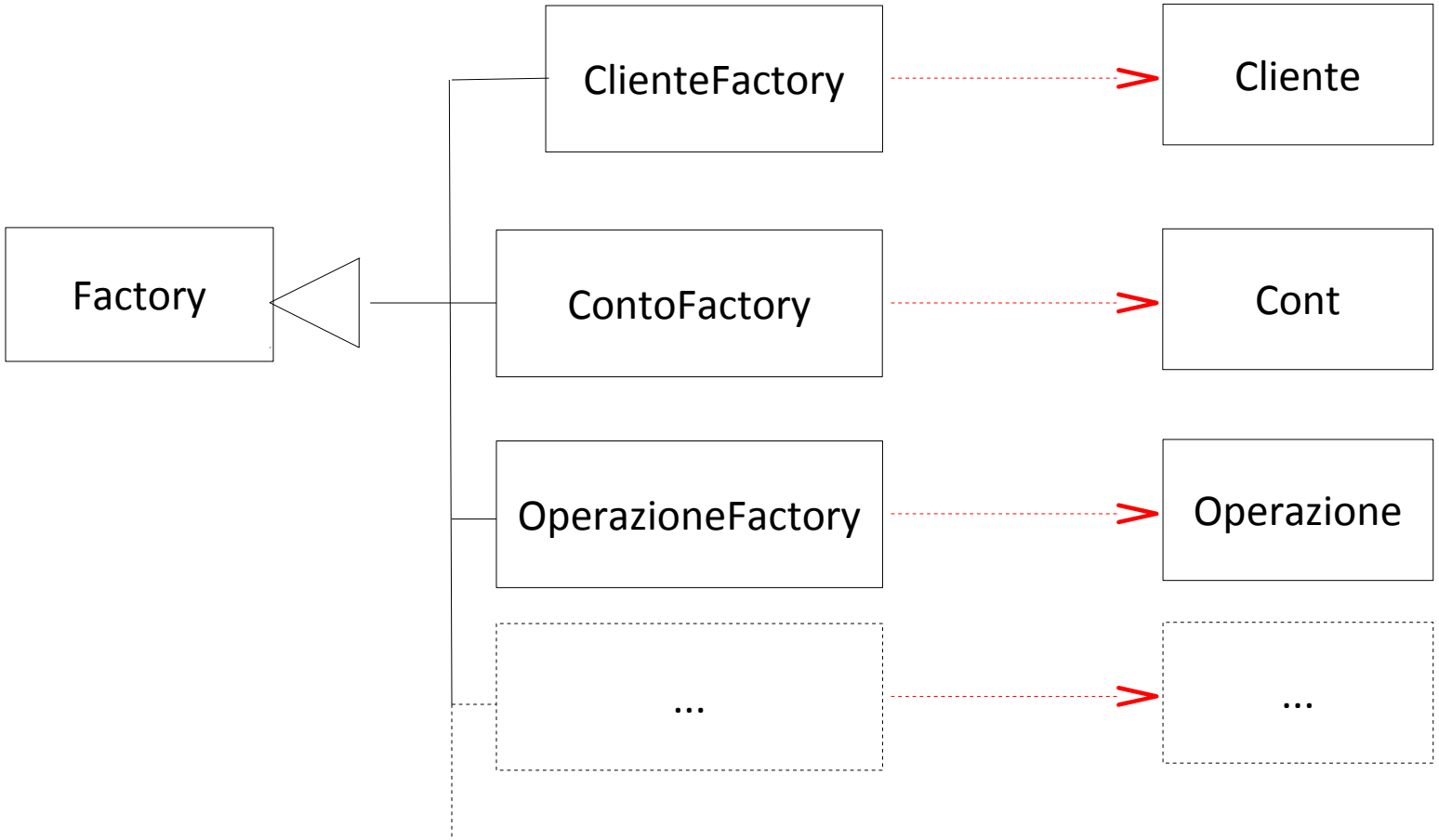
Uso Abstract Factory

```
public class XXX {


   public void qualcheMetodo() {

      Conto conto=...createNew();
     //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 ConcreteFactory 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

Polimorfismo + Factory



Reflection + Factory

Reflection

- Reflection viene spesso usato assieme ad altre tecniche per ridurre la dipendenza
- Vediamo come sia possibile cercare di ridurre la dipendenza anche altro modo.

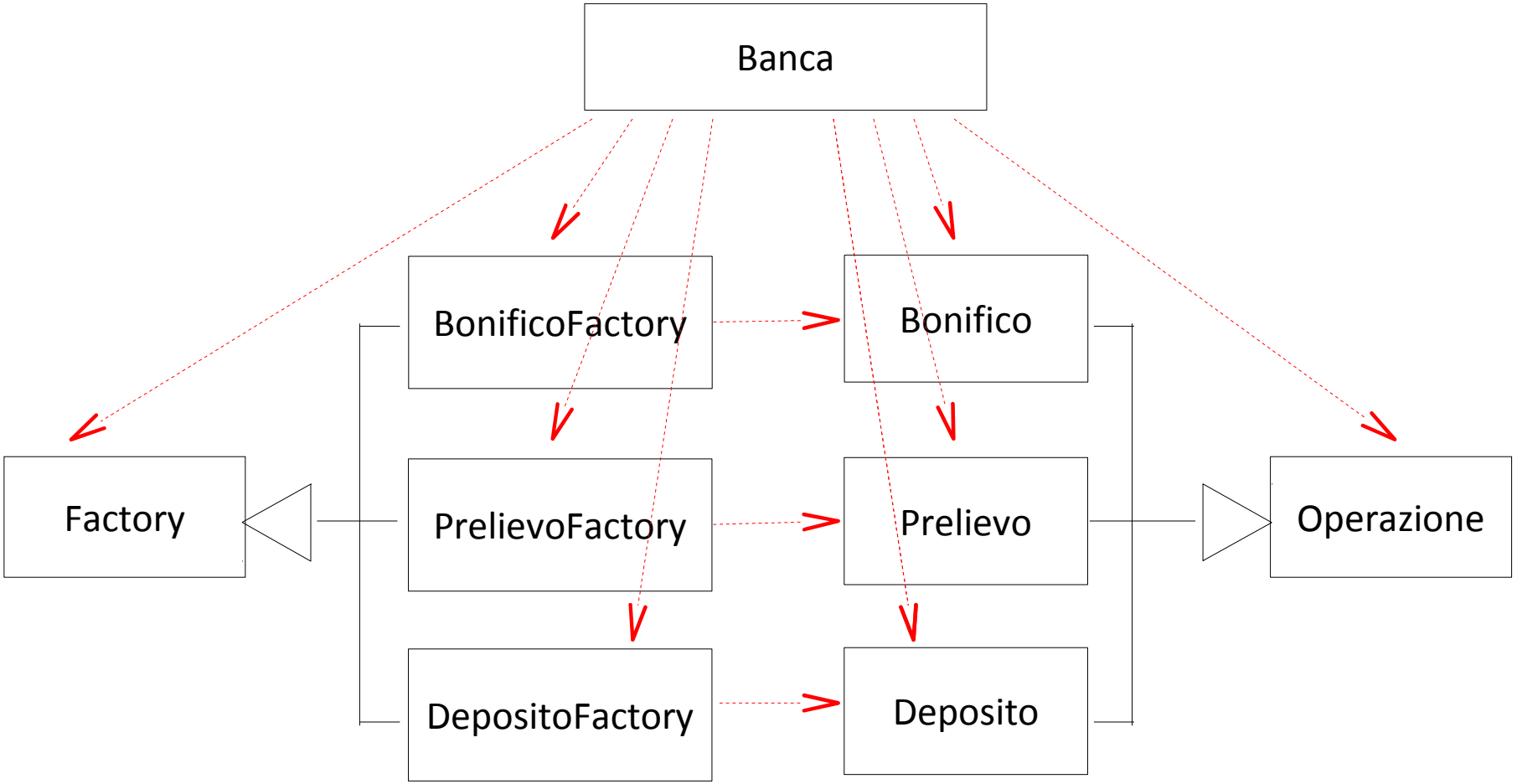
Protect Variations

- Come proteggere un oggetto/sistema dai cambiamenti indotti dagli altri oggetti/sistemi?
- Mascherando i cambiamenti degli altri oggetti tramite una “inter-faccia” stabile
- Una tecnica per mascherare è la **progettazione guidata dai dati** e un caso particolare è un file di configurazione

Come ridurre dipendenza

- Come possiamo sfruttare un file di configurazione per ridurre la dipendenza?
- Mettiamo nel file di configurazione tutti le informazioni per la creazione delle operazioni bancarie

Dipendenze prima

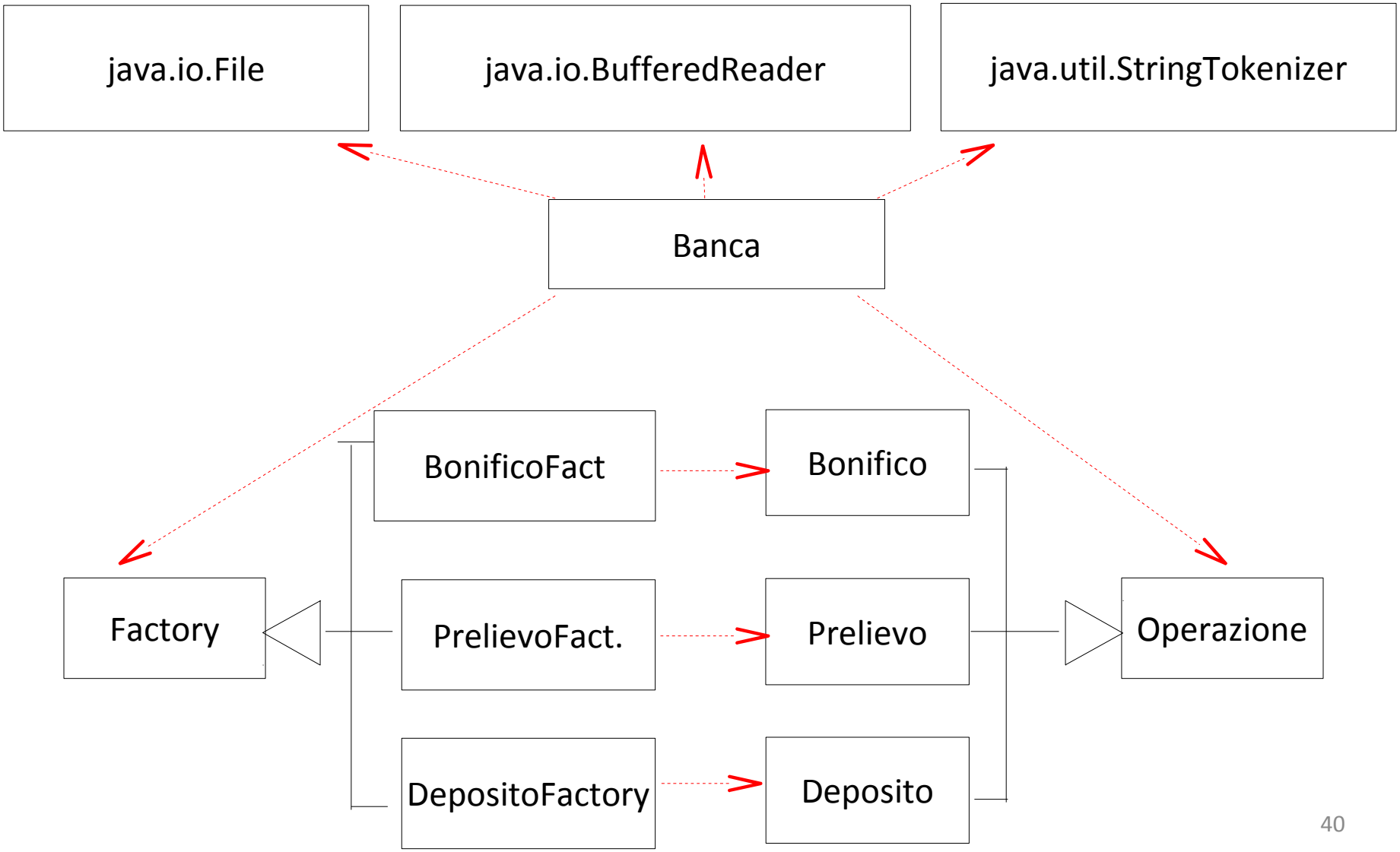


Dipendenza

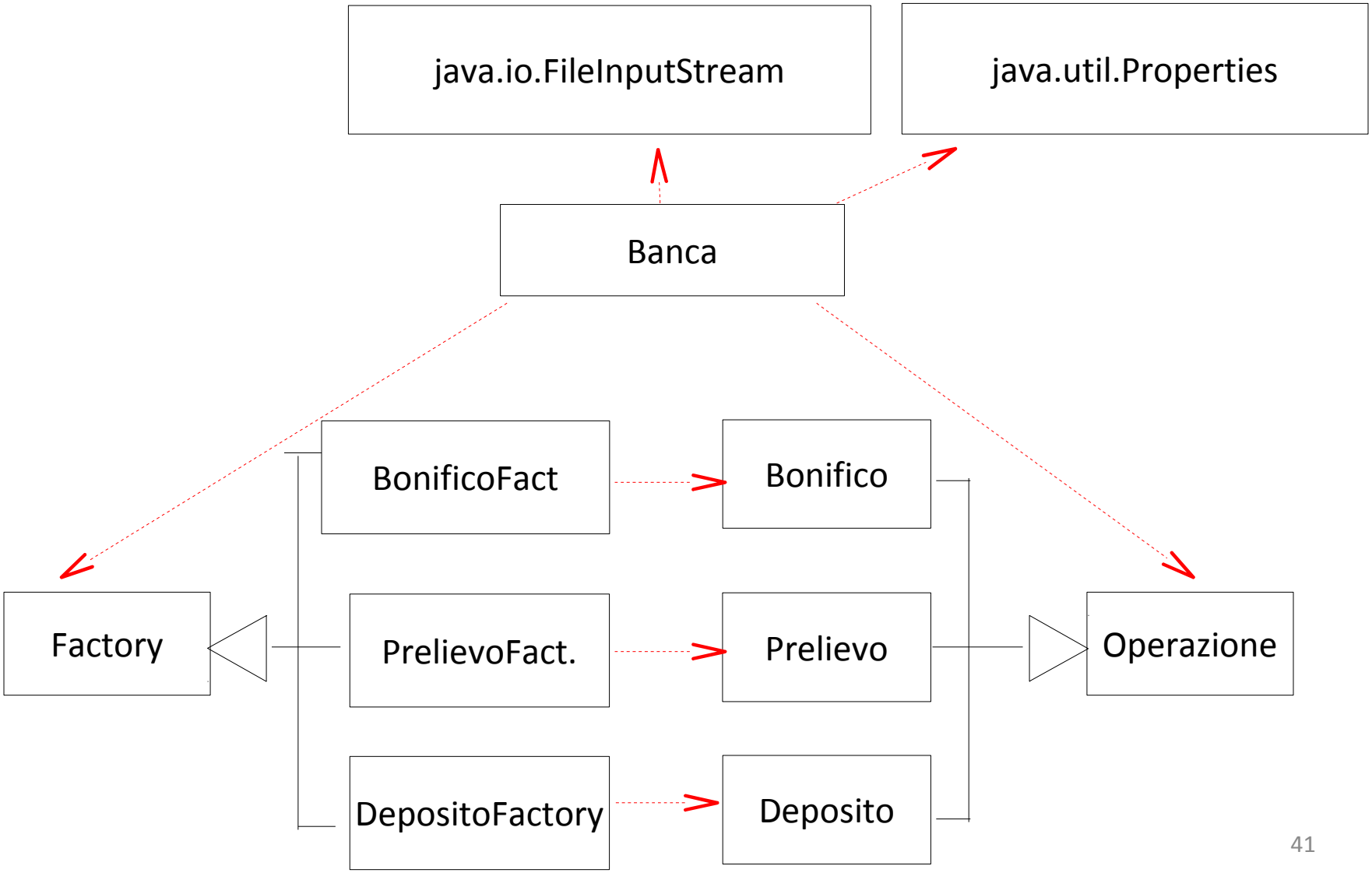
Il fatto che Banca costruisca tramite le varie Factory tutti le Operazioni crea una forte dipendenza della Banca con le altre classi:

- c'é più possibilità che Banca debba essere modificata a causa di un cambiamento alla costruzione di un oggetto Operazione che per una modifica al funzionamento della Banca stessa

Dipendenze dopo



Dipendenze dopo



Pro e contro

+/- Tolgo dipendenze ma ne Aggiungo altre

+ Ma le dipendenza con classi delle API standard di Java sono meno problematiche perché molto stabili (pochi cambiamenti in vista)

- Non tutto il comportamento risiede nel codice, bisogna sapere dell'esistenza del file esterno

Formato file .properties

```
#commenti preceduti dal carattere #
!commenti preceduti dal carattere !
number=23;
stringa=Questa è una stringa
stringaSuMolteLinee=Una stringa \
su più linee
altraStringa:Quaranta
stringaConUguale=x\=y
stringaConSlash=x\\y
stringaCon2Punti=x\:y
nomeCon\:=Valore
```

File banca.properties

```
#banca.properties
#
numeroOperazioni=4
cro0=1234567890123
importo0=12.34
...
cro1=0123456789012
importo1=43.21
…
cro2=2345678901234
importo2=-23.41
...
cro3=3216549872103
importo3=32.14
```

Codice

```
try {
FileInputStream fis =
      new FileInputStream(“banca.properties”);

Properties props = new Properties();
props.load(fis);
fis.close();

String tempNum=props.getProperty(“numeroOperazioni”);
int num = Integer.parseInt(tempNum);

for (int i=0; i<num; i++) {
  Operazione op=new Operazione();
  op.setCRO(props.getProperty(“cro”+i));
  double importo =
     Double.parseDouble(props.getProperty(“importo”+i));
  op.setImporto(importo);

  …}
}
```

Cosa manca?

- Generare le Operazioni in maniera polimorfa!
- In pratica dobbiamo riuscire a creare le operazioni 0 e 1 di tipo Bonifico, mentre il prodotto 2 di tipo Prelievo e il 3 di tipo Deposito
- Serve di nuovo la “Reflection”
- La reflection è una tecnica spesso usata per implementare P.V.

File banca.properties

```
#banca.properties
#
numeroOperazioni=4
cro0=1234567890123
importo0=12.34
class0=banca.Bonifico
...
cro1=0123456789012
importo1=43.21
class1=banca.Bonifico
…
cro2=2345678901234
importo2=-23.41
class2=banca.Prelievo
...
cro3=3216549872103
importo3=32.14
class3=banca.Deposito
```

Codice

```
try {
FileInputStream fis =
      new FileInputStream("banca.properties");

Properties props = new Properties();
props.load(fis);
fis.close();

String tempNum=props.getProperty("numeroOperazioni");
int num = Integer.parseInt(tempNum);

for (int i=0; i<num; i++) {
  Operazione op =
  Class.forName(props.getProperty("class"+i)).newInstance();
  op.loadFromProperties(props,i);
  …}
}
catch (Exception e){...}
```

Esercizio

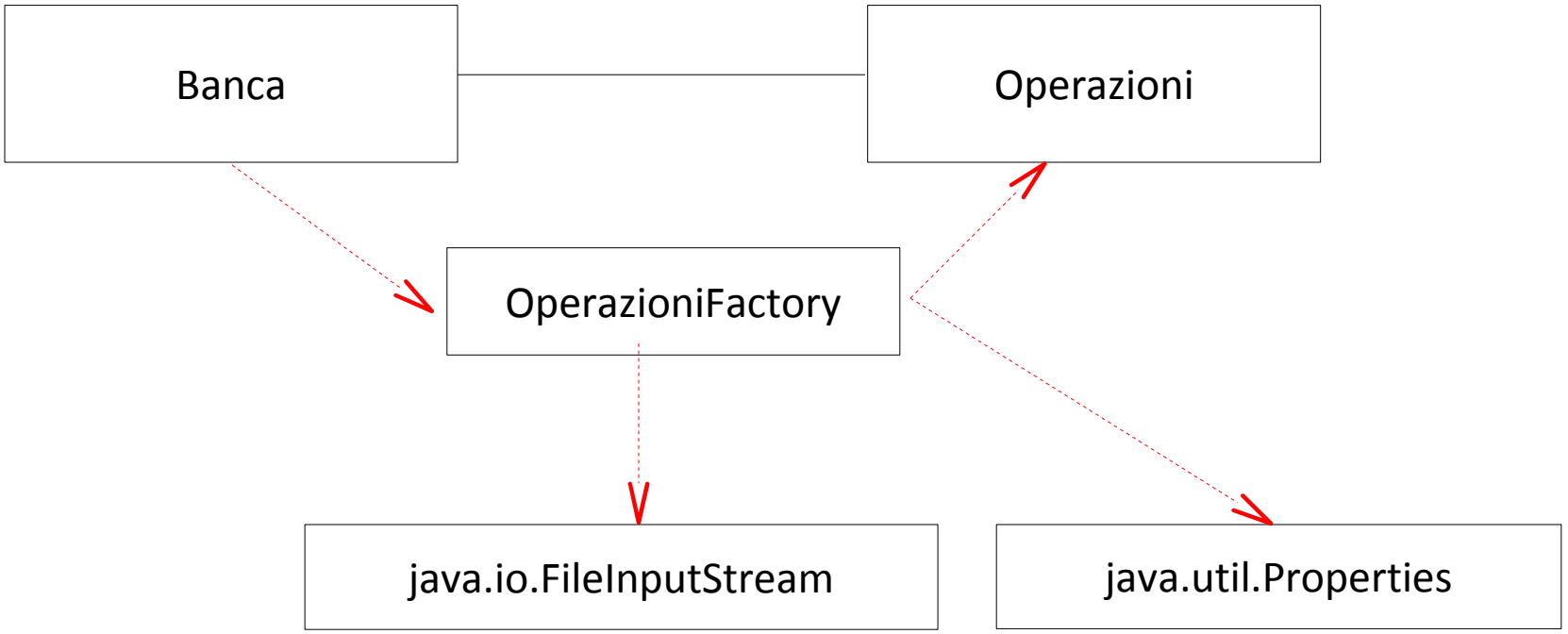
- Perché abbiamo richiamato il metodo `loadFromProperties`?
- A cosa serve il parametro `i` di questo metodo?

Protect Variation

- Più complesso
- Più lento
- Più costoso
- Senza controllo compilatore (vedi try-catch)
- Bisogna valutare se effettivamente l'interfaccia rappresenta un punto di possibile cambiamento
- La modifica può essere fatta senza bisogno di modificare il codice Java

Esercizio

Progettare e implementare una Simple Factory con Protect Variation per la creazione delle Operazioni (elencare pro e contro)



Esercizio

- Pensare a dover modificare l'applicazione dovendo aggiungere nuovi tipi di Operazioni. Quale delle scelte viste risulta la migliore?

Evoluzioni progettisti

- Alle prime esperienze, i progettisti tendono a essere troppo ottimisti e pensare che non ci saranno variazioni, bug e manutenzione e quindi la struttura del codice risulta troppo sensibile ai cambiamenti
- Successivamente, i progettisti tendono a essere troppo pessimisti e prevedono troppi punti di cambiamento e il codice risulta troppo complesso

Riflessioni sulla reflection

- La reflection è molto interessante e permette di ottenere codice molto flessibile
- Analoga alla reflection sono i Metadati per quanto riguarda i DB
- Attenzione che usando la reflection perdiamo i vantaggio di avere un linguaggio fortemente tipato e spostiamo al run-time la scoperta di possibili errori.

Riflessioni sulla reflection 2

- Tramite reflection è possibile cambiare al run-time la visibilità dei membri di una classe: è quindi possibile accedere ad un attributo che è privato
- Questo permette a molte librerie/framework di accedere in modo uniforme allo stato degli oggetti
- Il problema della mancanza di controllo del compilatore viene mitigato dal fatto che il codice è ben testato dai tanti utilizzatori

Reflection e sicurezza

Vale la pena di rimarcare che il fatto che tramite la reflection sia possibile cambiare la visibilità di attributi di una classe, ancora di più sottolinea che l'unico scopo delle visibilità private, protected e package è quello di limitare la dipendenza tra le classi e NON riguarda minimamente aspetti di sicurezza applicativa.

Quindi: per nascondere le password non ha senso usare "`private`"

Sicurezza e Java

- Spesso si sente dire che Java è un linguaggio sicuro a differenza del C.
- Questa è vero ma NON riguarda il fatto che C non ha le visibilità private, protected etc.
- Bensì, riguarda i controlli fatti al run time dalla virtual machine quando esegue il nostro codice (es. controllo indici, SecurityManager, Eccezioni etc.)

domande