Programmazione a Oggetti
Modulo B

Dott. Alessandro Roncato

18/02/2014

Riassunto

- Pure Fabrication
- Simple Factory
- Factory Method
- Abstract Factory

Soluzione 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?

Soluzione tipica: Singleton

```
public class XXX {
    public qualcheMetodo() {
        Conto conto=Factory.getSingleton().load(id);
  }
}
```

- Visibilita' globale
- Unica istanza: generalmente Factory ha bisogno di una sola istanza .

  Quindi Factory e' un caso particolare di Pure Fabrication e di solito si utilizza un Singleton

Soluzione statica

```
public class XXX {
    public qualcheMetodo() {
        Conto conto=Factory.load(id);
  }
}
```

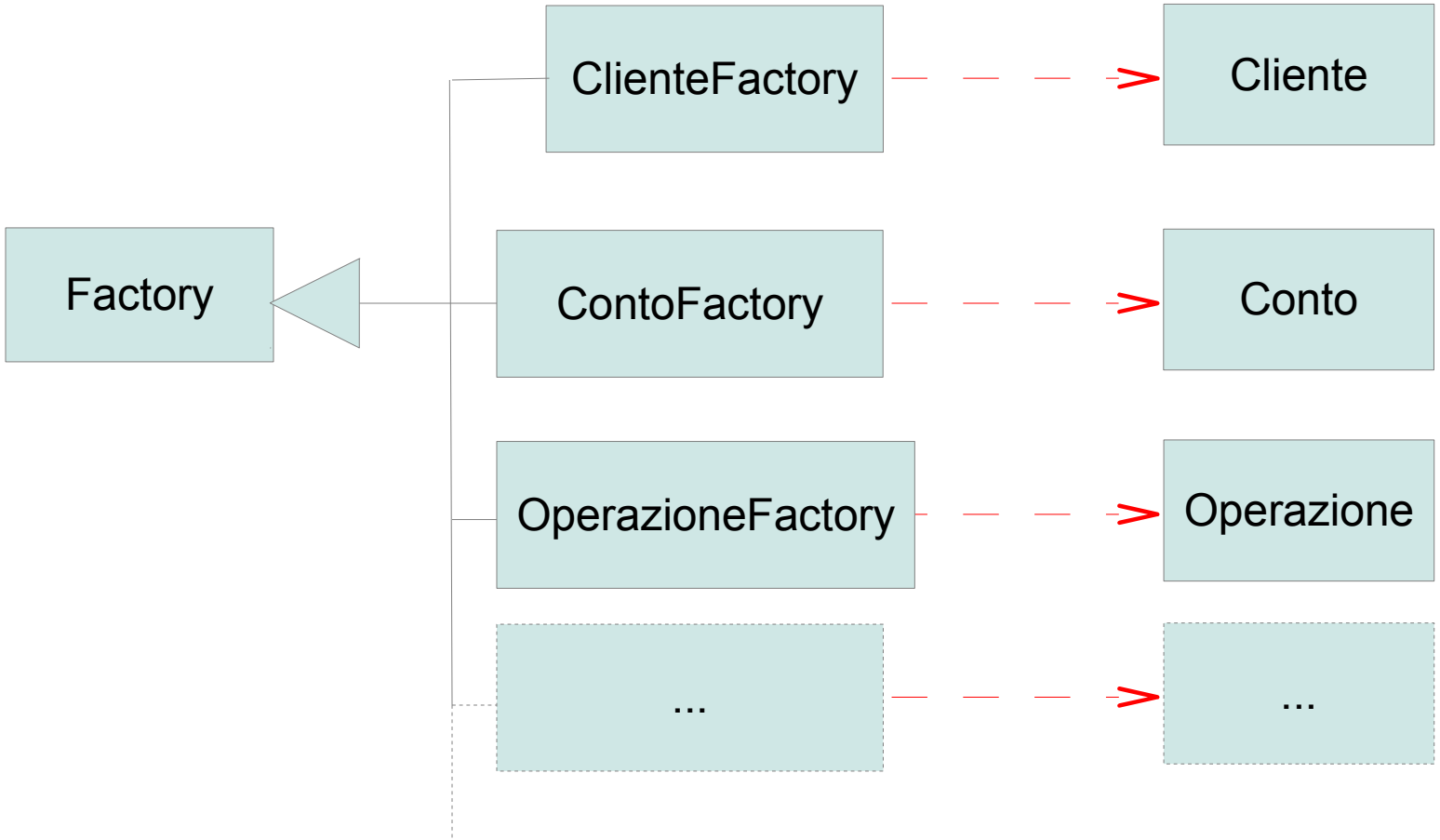
- Devo dichiarare load statico
- Visibilita' globale
- Nessuna istanza.

  Non puo' sfruttare il polimorfismo (vedremo)

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

Polimorfismo + Factory



Polimorfismo + Factory

```
public abstract class Factory {
 public Object load(int id){
   Class.classForName("Driver");
   Connection con=DriverManager.get.Connection(...);
   Statement sta = con.createStatement();
   ResultSet rs = sta.executeQuery(getQuery(id));
   if (rs.next())
      return new loadFromRow(rs);
   return null;
 }
 abstract public String getQuery(int id);
 abstract public Object loadFromRow(ResultSet rs);
}
```

Polimorfismo + Factory

```
public class ContoFactory extends Factory{
  public String getQuery(){
    return "SELECT * FROM conti WHERE id="+id;
  }
  public Object loadFromRow(ResultSet rs){
    int numero=rs.getString("numero");
       …
    return new Conto(numero,...);
  }
  //+ singleton
}
```

Polimorfismo + Factory

```
public class ClienteFactory extends Factory{
  public String getQuery(){
    return "SELECT * FROM clienti WHERE id="+id;
  }
  public Object loadFromRow(ResultSet rs){
    String nome=rs.getString("nome");
       …
    return new Cliente(nome,...);
  }
  //+ singleton
}
```

Dimensione metodi

- Nell'applicazione del Polimorfismo abbiamo spezzato l'implementazione del metodo load in piu' metodi: load, loadFromRow, getQuery.
- In generale questo e' un bene perche':
  - Metodi piu' semplici da capire
  - Favoriscono ulteriormente il polimorfismo
  - Possono piu' facilmente essere riusati
  - Piu' facile da trovare un nome significativo

A cosa serve Polimorfismo?

```
public class GestioneConto {

  public Set caricaConti(){
    Set result = new HashSet();
    for(int id=0;id...;id++) {
        result.add(ContoFactory.getSingleton().load(id);
    }
    return result;
  }
}
```

A cosa serve Polimorfismo?

```
public class GestioneClienti {

  public Set caricaClienti(){
    Set result = new HashSet();
    for(int id=0;id...;id++) {
        result.add(ClienteFactory.getSingleton().load(id);
    }
    return result;
  }
}
```

A cosa serve Polimorfismo?

```
public class Gestione {
  public Set carica(Factory f){
    Set result = new HashSet();
    for(int id=0;id...;id++) {
        result.add(f.load(id);
    }
    return result;
  }
  public Set caricaConto(){
    return carica(ContoFactory.getSingleton());
  }
  public Set caricaCliente(){
    return carica(ClienteFactory.getSingleton());
  }
}
```

Reflection

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

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.

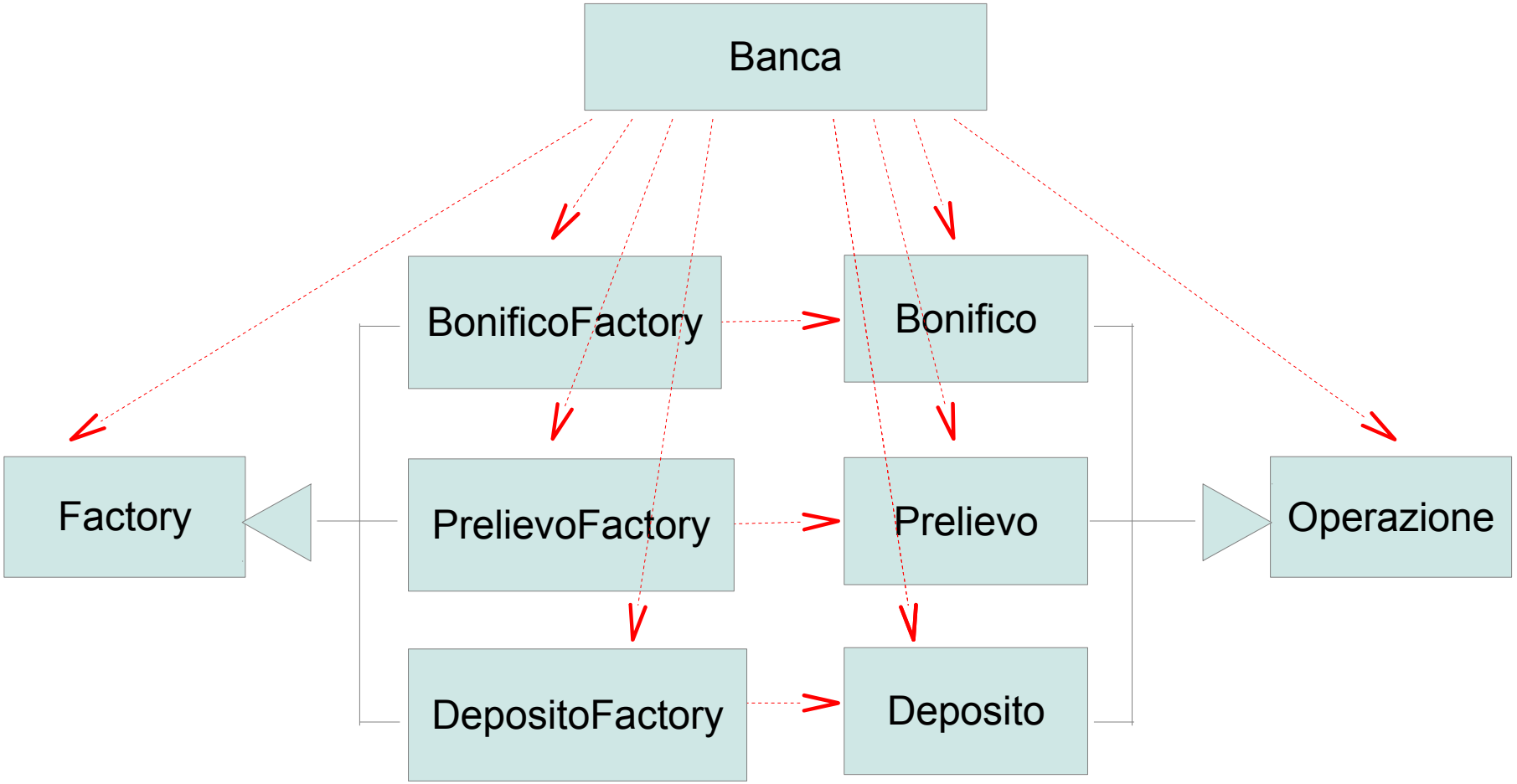
Protect Variations

- 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

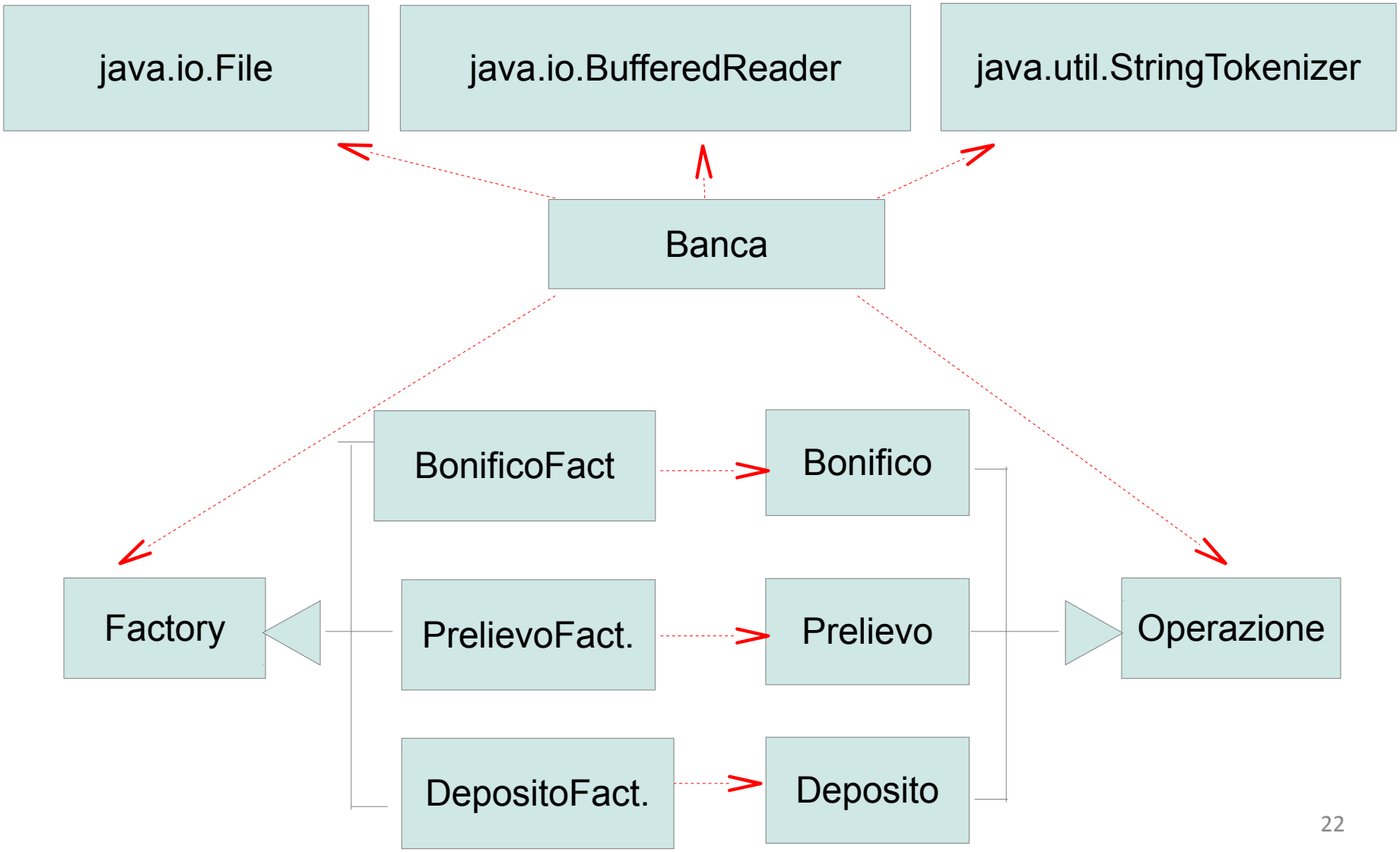


Dipendenza

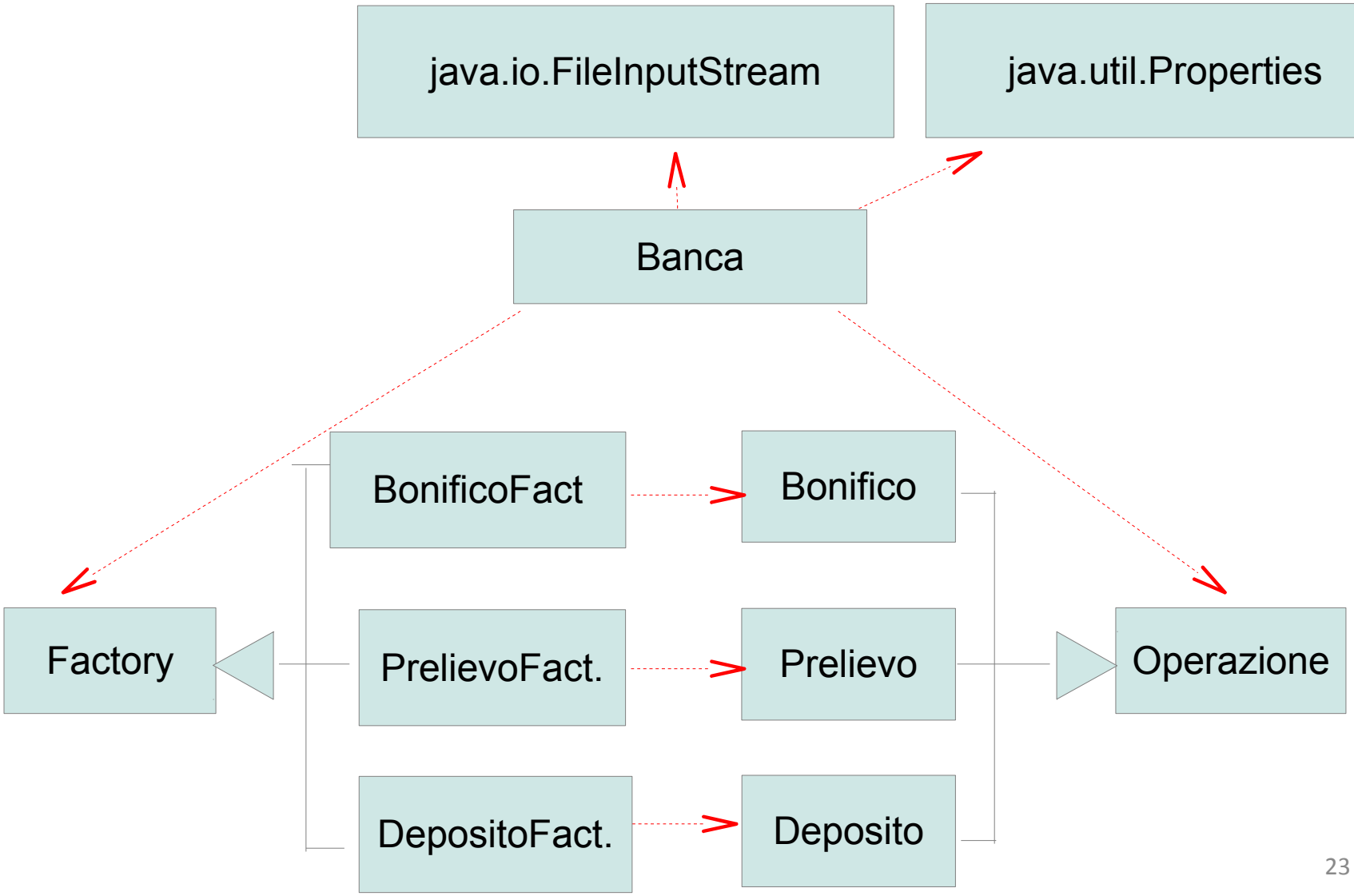
Il fatto che Banca costruisca tramite le varie Factory tutte le Operazioni crea una forte dipendenza della Banca con le altre classi:

- 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);

  …}
}
catch (Exception e){...}
```

Cosa manca?

- Generare le Operazioni in maniera polimorfa!
- In pratica dobbiamo riuscire a creare le operazioni 0 e 1 di tipo Bonifico, mentre la 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

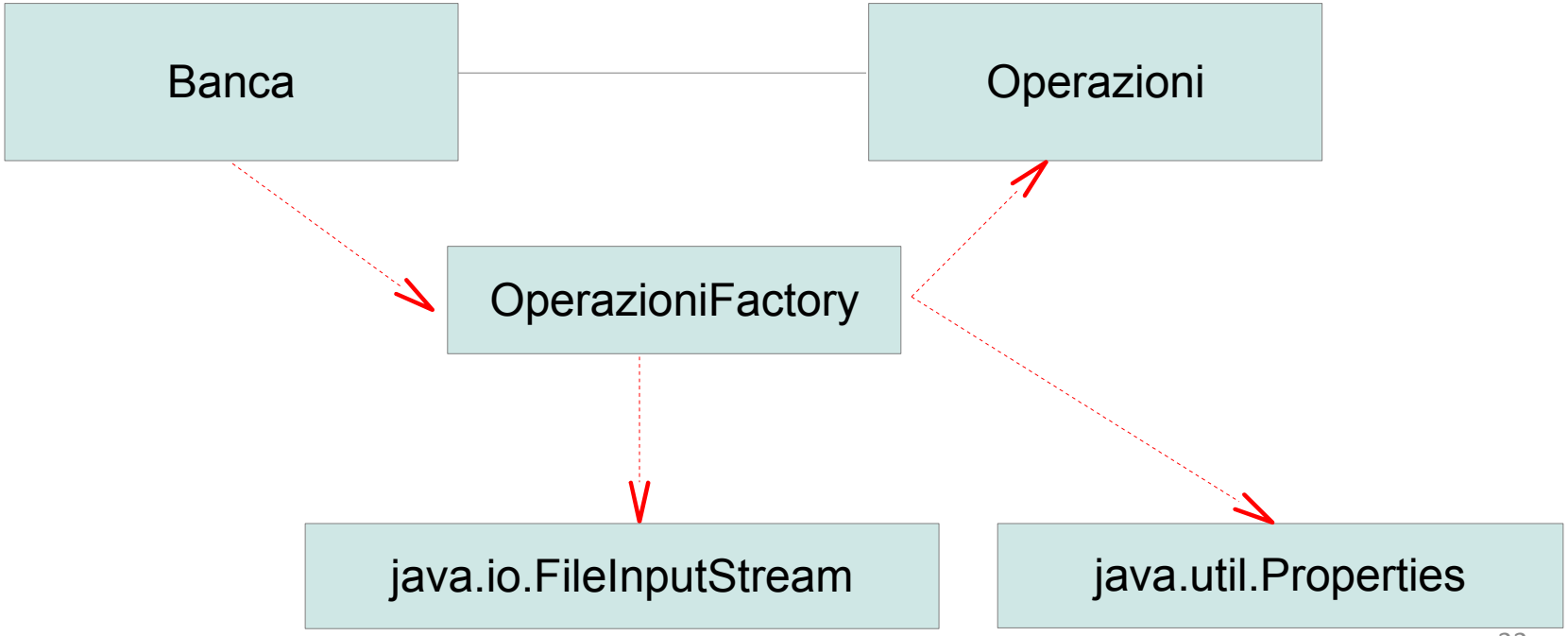
- Perché abbiamo richiamato il metodo `loadFromProperties`?
- A cosa serve il parametro `i` di questo metodo?

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.)

Esercitazione

Progettare un'applicazione per la gestione di una segreteria studenti. L'applicazione deve gestire: gli studenti (e generare numeri di matricola distinti), corsi, piani di studio, esami e esami di laurea. Dovrà gestire solo la registrazione/controllo dei dati relativi ai concetti: per esempio, degli esami non dovrà implementare l'erogazione degli esami ma solo la registrazione finale.

Esercitazione

In particolare ci sono vari tipi di corso: triennali, magistrali, v.o., master e dottorati.

Gli studenti possono sostenere solo esami presenti nel proprio corso di studio.

Nei primi due anni dei corsi triennali i piani di studio sono fissi.

Gli esami vengono inseriti dai professori titolari dei corsi.

I piani di studio dei primi 2 anni triennali vengono inseriti in automatico al momento dell'iscrizione dello studente.

Per gli anni seguenti il piano viene inserito dallo studente stesso scegliendo tra gli esami liberi.

La segreteria inserisce i dati degli studenti, dei corsi e dei piani di studi standard dei primi 2 anni, i dati dei professori.

...

Esercitazione

Produrre:

Diagramma delle classi abbozzato

Diagramma dei casi d'uso

(Disegno prototipo interfaccia utente)

Diagramma di sequenza di almeno 2 casi d'uso complessi

Diagramma dei casi d'uso con metodi e attributi

Domande?