Programmazione a Oggetti
Modulo B

Lezione 3

Dott. Alessandro Roncato

10/02/2014

Riassunto

Diagrammi di sequenza

Diagrammi dei casi d'uso

Pattern Information Expert

Pattern Creator

Moodle

Link: moodle.unive.it

Corso: prog_og

Chiave: PO2012

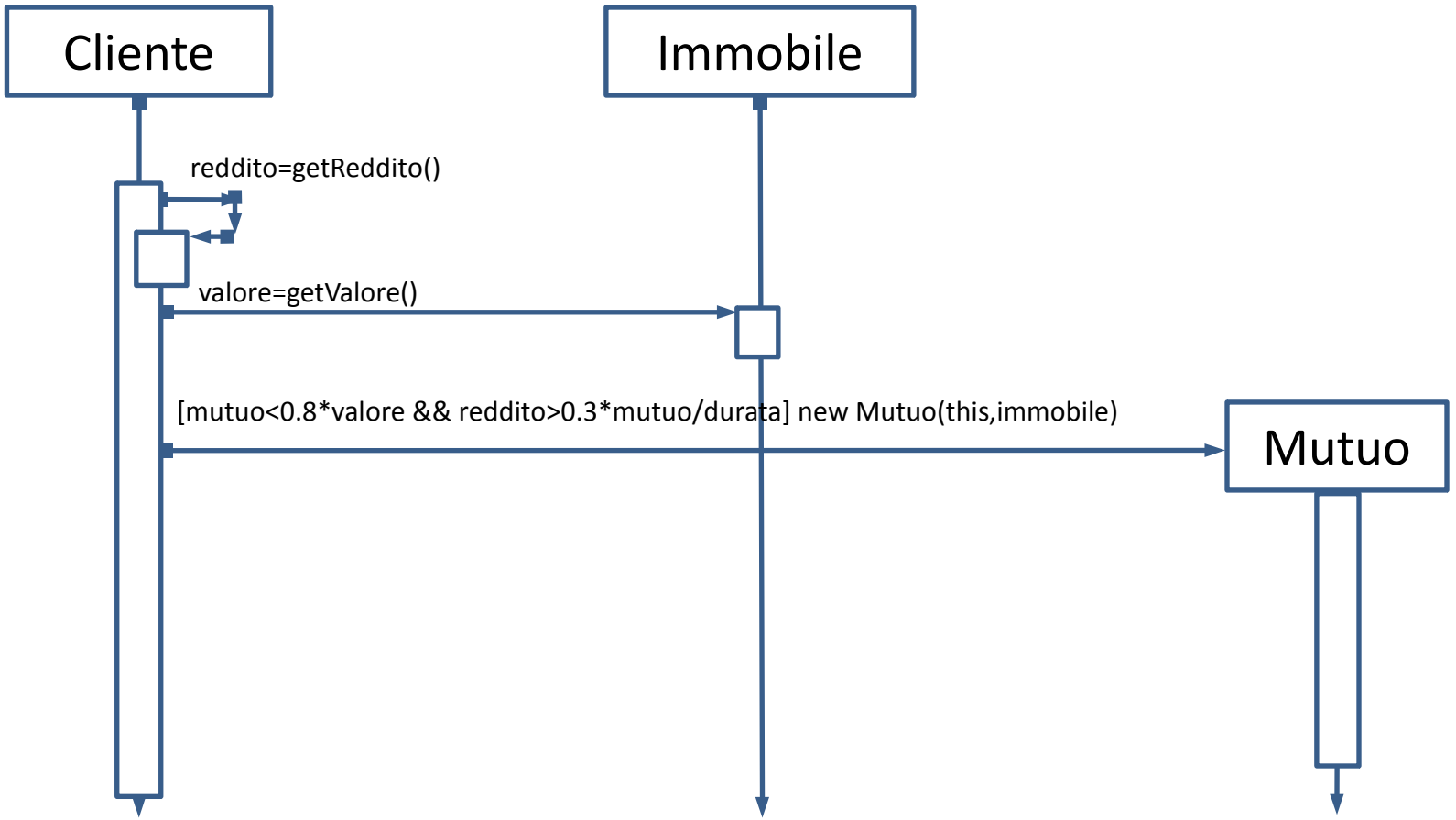
Inizia Mutuo

- Chi crea gli oggetto Mutuo?
- Non possiamo applicare C1 e C2;
- Per quanto riguarda C3 è applicabile a due differenti oggetti: Cliente e Immobile.

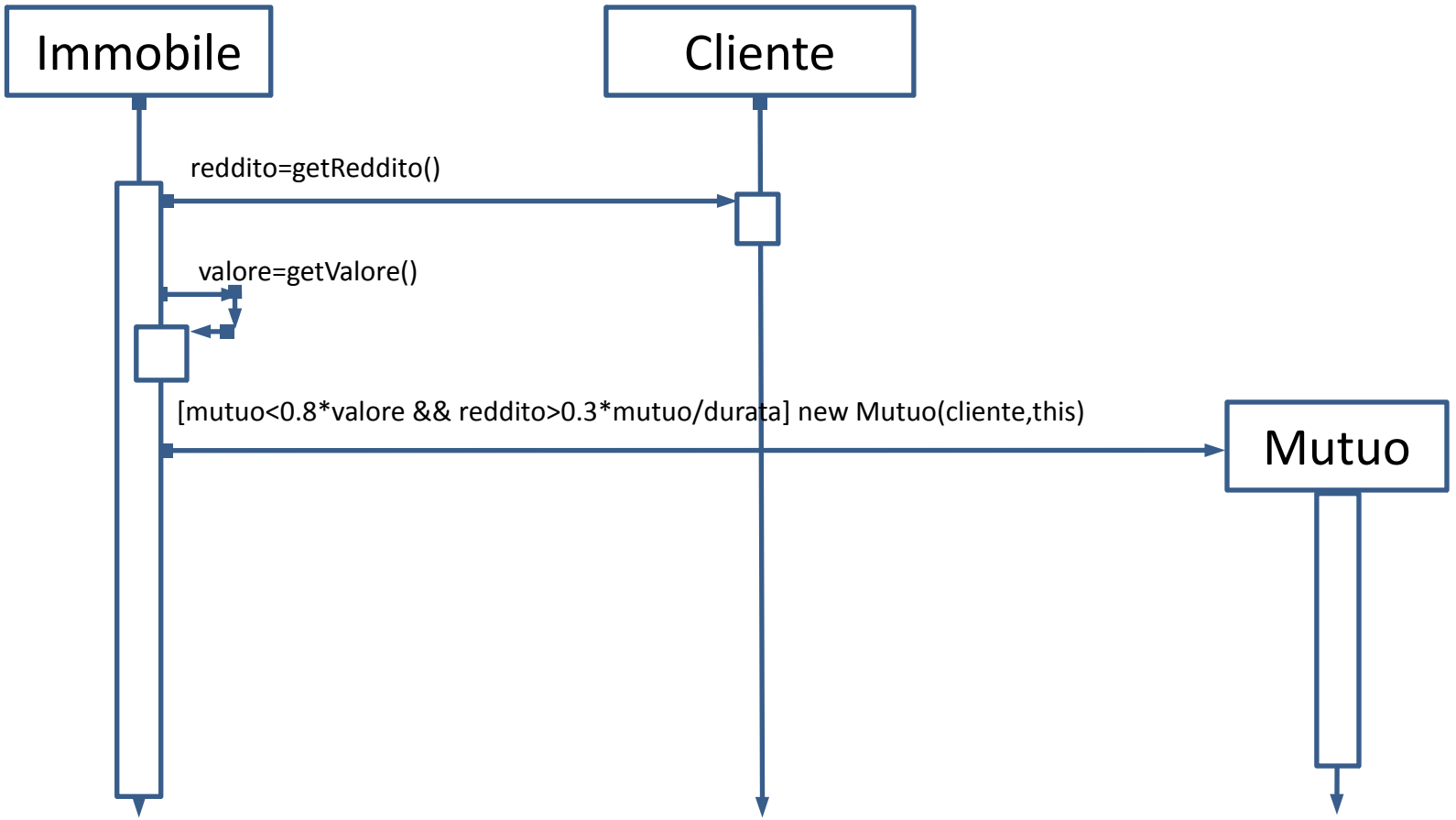
Inizia Mutuo

- Se non ci sono altri oggetti che memorizzano Mutuo per forza dobbiamo scegliere tra i due se vogliamo applicare il pattern Creator.
- Oppure vedremo che si possono applicare altri Pattern (esempio Factory).
- Esaminiamo i due Creator: Cliente e Immobile

Creator Cliente



Creator Immobile



Confronto

- Non c'è molta differenza nelle due scelte.
- In caso di parità è conveniente rimandare la scelta a quando si avranno ulteriori dettagli.
- Si continua quindi con gli altri casi d'uso.

Esercizio

Scrivete le due implementazioni delle due soluzioni Creator per il caso d'uso Inizia Mutuo

Null Object

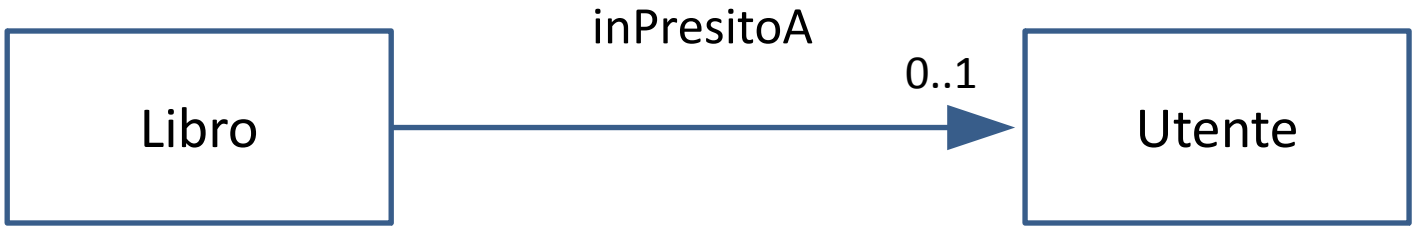
Vediamo ora un nuovo problema che si verifica quando abbiamo un'associazione del tipo:

In pratica all'oggetto A può essere associato un oggetto B oppure no.

Libro in prestito a?

Vediamo ora un nuovo esempio per capire meglio il problema

Per verificare chi è l'utente che ha in prestito un libro dobbiamo prima di tutto capire come rappresentare quando un Libro non è in prestito.



Esempio

```
public class Libro {
    Utente inPrestitoA;
    …
    public Utente getInPrestitoA()
    {
      return inPrestitoA;
    }
}
```

Esempio (alternativa)

```
public class Libro {
    Utente inPrestitoA;

   public boolean isInPrestitoA(Utente u)
    {
       return inPrestitoA.equals(u);
    }
}
```

Pro e contro 2 scelte

- getInPrestitoA in genere è la scelta migliore nella programmazione a oggetti (estensibilità, oltre a trovare facilmente il valore inPrestitoA, possiamo trovarne anche altre proprietà di interesse dell'Utente)
- per contro, isInPrestitoA gestisce facilmente i libri non in prestito

Libri non in prestito

Come rappresentare l'Utente dei Libri non in prestito?

- La scelta più ovvia è quella di utilizzare il valore `null`
- In questo modo si deve sempre controllare che il riferimento sia non nullo

Esempio null

```
public class Libro {
    Utente inPrestitoA=null;

   public boolean isInPrestitoA(Utente u)
    {
       if (inPrestitoA==null)
        return u==null;
       else
        return inPrestitoA.equals(u);
    }
}
```

isInPrestito

La modifica riguarda solamente il metodo stesso, mentre il resto delle classi non viene modificato.

Mentre, il metodo getInPrestitoA non viene modificato, ma ...

getInPrestitoA

Più “scomoda” è la prima alternativa, quella in cui ritorniamo l'Utente che ha in prestito il Libro.

Nel caso il Libro non sia in prestito, viene ritornato il riferimento null.

Quindi chi usa il metodo pubblico deve essere a conoscenza di questo!

Quindi ANCHE il codice delle altre classi deve ricordasarsi di questo fatto!

Esempio null

```
public class Biblioteca {
    Utente u = libro.getInPrestitoA()[
    if (u==null)
        …
       else
        ,,,
    }
}
```

Esempio null

```
public class Scaffale {
    Utente u = libro.getInPrestitoA()[
    if (u==null)
        …
       else
        ,,,
    }
}
```

Pattern Null Object

D: Come rappresentare le associazioni senza elementi?

R: Si crea un oggetto denominato Null Object e si usa il riferimento a questo oggetto per indicare la mancanza di elementi

Esempio Null Objetc

```
public class Libro {
   Utente inPrestitoA=UtenteNullo.istanza;
   public Utente getInPrestitoA()
   {
     return inPrestitoA;
   }
   public boolean tornaInBiblioteca(Utente uu){
      if (uu.equals(inPrestitoA))
         inPrestitoA=UtenteNullo.istanza;
      return inPrestitoA==UtenteNullo.istanza;

  //o inPrestitoA.equals(UtenteNullo.istanza);
    }
…
}
```

Esempio Null Objetc

```
public class UtenteNullo extends Utente {
    public UtenteNullo(){}
    public static UtenteNullo istanza=
        new UtenteNullo();
    public void metodo(Argomento a){}
}
```

Attenzione: Null Object !=null. Null Object è un oggetto a tutti gli effetti e con questo pattern rappresenta l'oggetto speciale diverso da tutti gli altri che rappresenta un oggetto vuoto

Vantaggi Null Object

Nel codice di tutta l'applicazione possiamo evitare di controllare se il puntatore ritornato da getInPrestitoA sia diverso da null.

```
l.getInPrestitoA().equals(utente)
```

Invece di

```
if (l.getInPrestitoA()!=null)
   c.getInPrestitoA().equals(utente)
```

Svantaggi: una classe in più!

Esame

Attenzione: all'esame scritto almeno un esercizio ha a che fare con (una versione modificata di) questo pattern.

ListaVuota

AlbertoVuoto

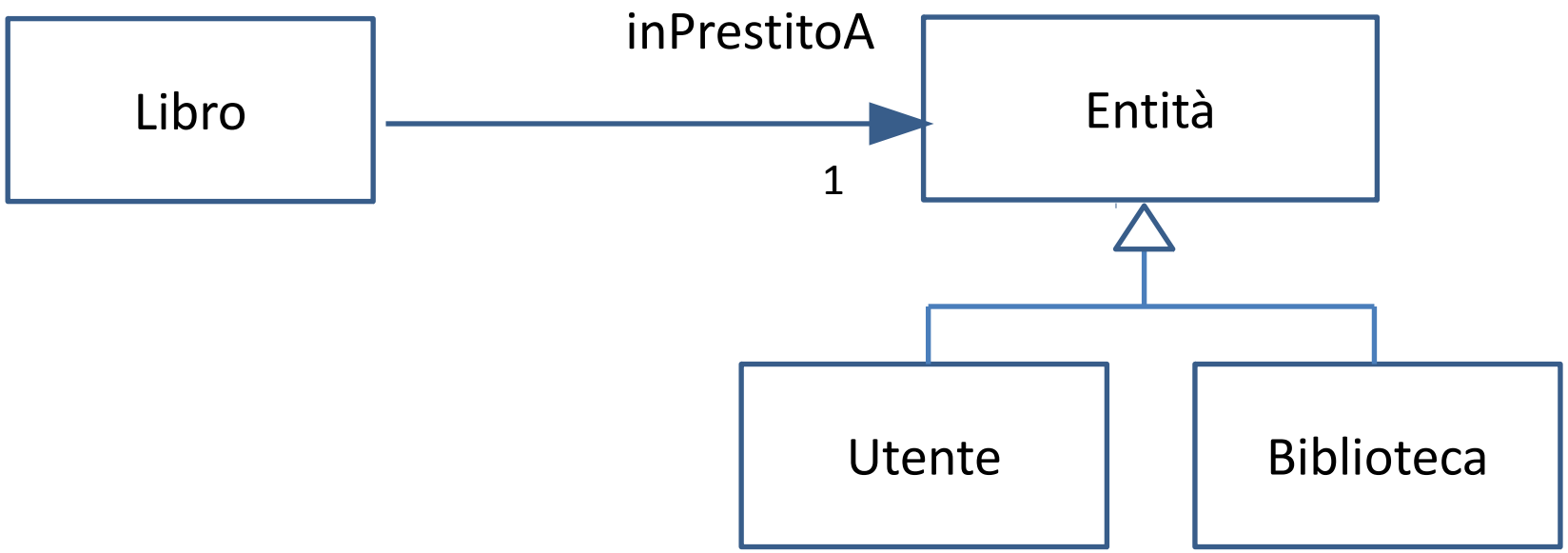
L'idea di fondo è sempre la stessa.

Libri non in prestito

Come rappresentare i Libri non in prestito?

- Una scelta alternativa è quella di ritornare come Utente la Biblioteca stessa
- Per fare questo bisogna che Biblioteca e Utente abbiano qualcosa in comune ovvero che estendano/implementino una stessa classe/interfaccia

Nel Diagramma



Nel codice

```
public interface Entita {
…
}

public class Utente implements Entita {
…
}
public class Biblioteca implements Entita {
…
}
```

Nel codice

```
public class Libro {
    Entita inPrestitoA=Biblioteca.istanza;
    public Entita getInPrestitoA()
    {
      return inPrestitoA;
    }
}
```

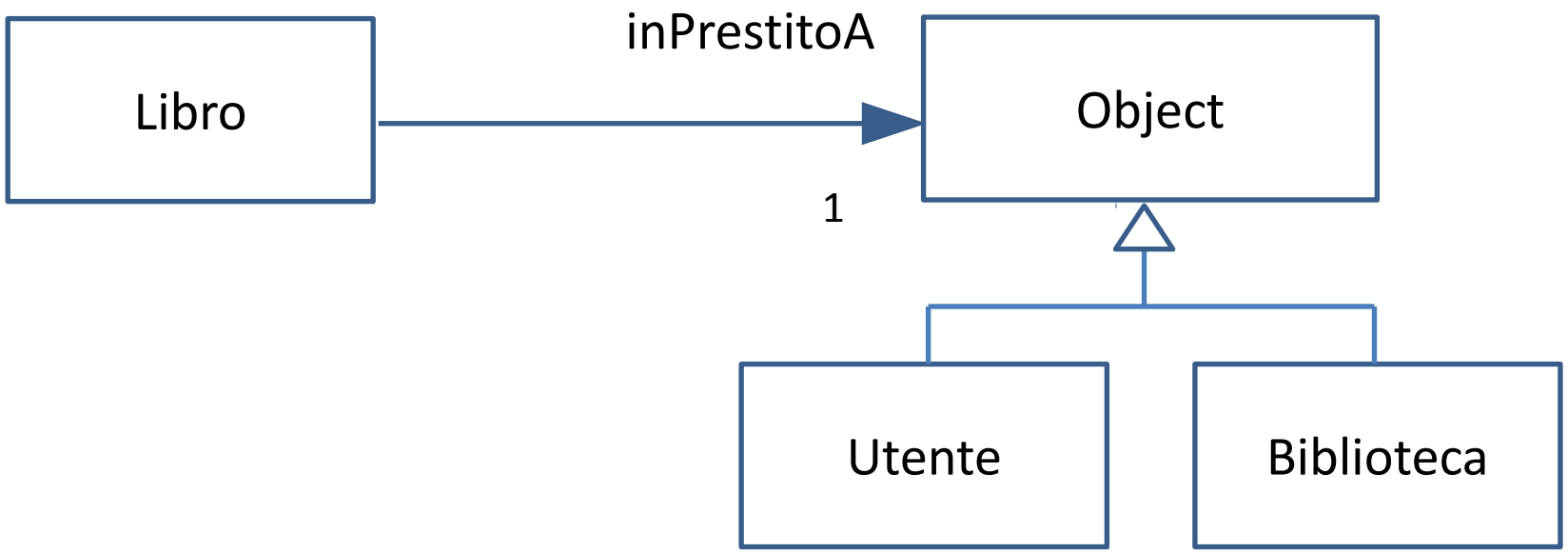
Soluzione più semplice

- Tutti gli oggetti Java estendono Object
- PRO: Non serve una classe/interfaccia in più
- CONTRO: Meno leggibile, Meno estensibile
  (non posso aggiungere metodi all'Entità, posso solo controllarne l'uguaglianza con il metodo equals)

Soluzione semplice

```
public class Libro {
    Object inPrestitoA=Bibliteca.istanza;
    public Object getInPrestitoA()
    {
      return inPrestitoA;
    }
}
```

Nel Diagramma

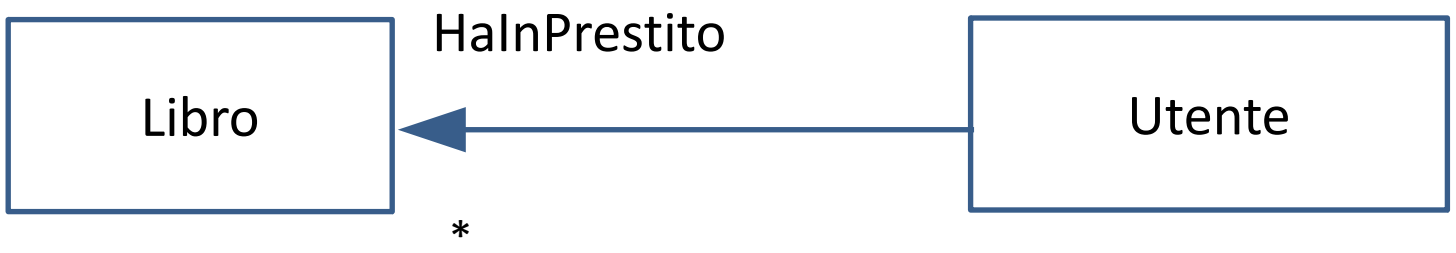


Esercizio

- Analizzare il caso in cui sia solo l'Utente a tenere traccia dei Libri che ha in prestito

- Scrivere esempio di codice;
- Disegnare i relativi diagrammi
- Confrontare Pro e Contro

Diagramma Esercizio



Attenzione

- Solitamente diagrammi UML e codice non vengono fatti nella stessa fase dello sviluppo
- Prima si fanno i diagrammi e solo successivamente si scrive il codice.
- In queste lezioni, li vediamo insieme per rendere evidente le implicazioni sul codice delle varie scelte di progettazione

Attenzione 2

- Quello che vediamo nei diagrammi ha un diretto impatto nel codice ma non lo determina univocamente.
- Vedremo come sarà possibile implementare i metodi con la stessa firma in modo diverso
- Le implementazioni viste e che vedremo sono a titolo di esempio e quelle più naturali ma non per questo le uniche corrette

Cos'è un pattern

Composto da 3 parti:

1) Nome (Esempio “Information Expert”)
2) Domanda o problema (Esempio “Come assegnare le responsabilità”)
3) Risposta o soluzione (Esempio “Agli oggetti che hanno le informazioni per farlo)

Problema

- Come gestire le funzionalità della Banca di cui:
  1) esiste un unico elemento in tutta l'applicazione
  2) usato in molti punti
- Soluzione **NON** orientata agli oggetti: `public` e `static`

Static (definizione)

```
public class Banca {
 static Array<Conto> conti=new ....;
 static int prossimoNumero=1;
 …
 static public int apriConto(Nome cliente){
   int numero=prossimoNumero++;
   conti.add(new Conto(cliente, numero));
   Return numero;
 }
 static public Conto findConto(int numero)
 {
  ...
 }
}
```

Static (uso)

```
public class XXX {

  …
 metodo( )
 {
   if (Banca.findConto(numero)!=null)

 …

   Int numero = Banca.apriConto(nome);
 …

}
```

Pattern Singleton

- D: come gestire: 1) oggetto unico e 2) usato in globalmente nell'applicazione?
- R: con un oggetto singolo con visibilità globale
- Come fare in modo che sia impossibile da duplicare da parte delle altre classi?

Pattern singleton

- Costruttore privato
- Unica istanza costruita dalla classe
- Riferimento all'oggetto privato
- Metodo getter statico e pubblico
- Vari tipi di inizializzazione (preventiva, lazy, etc.)

Costruttore privato

- Gli oggetti devono essere semplici e “sicuri” da usare!
- Cosa potrebbe succedere se per errore nella nostra applicazione venissero costruiti più oggetti Banca?
- Ci sarebbe un'inconsistenza.

Unica istanza costruita

- Dato che il costruttore è privato, l'unica classe che può usarlo per creare oggetti è la classe che lo definisce.
- Quindi la classe stessa deve creare l'unica istanza dell'oggetto
- Quando farlo? 1) sempre e prima di tutto, 2) solo quando serve, 3) etc

Riferimento privato

- Lascia maggiore libertà di implementazione
  - Il momento esatto della costruzione
  - Numero e tipo oggetto

Getter pubblico

- Permette la visibilità globale dell'oggetto

Singleton

```
public class Banca {
  private static final Banca REF=new Banca(...);
  Array<Conto> conti=...;
  …
  public static Banca getSingleton()
  { return REF;}
  public int apriConto(String nome){
     Stesso codice versione static
  }
  public Conto findConto(String titolo){
     Stesso codice versione static
  }
 ...
}
```

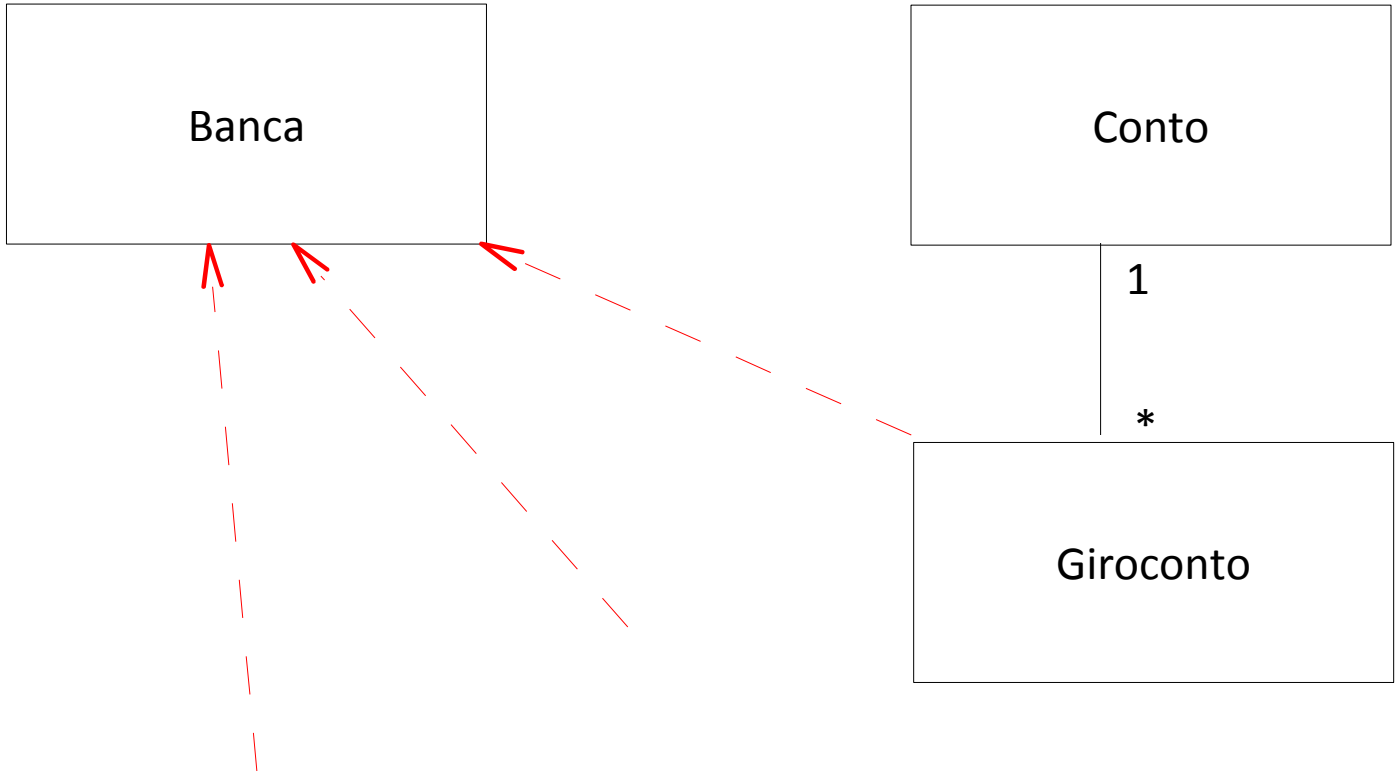
Singleton lazy

```
public class Banca {
  private static Banca ref=null;
  Array<Conto> conti=...;
  …
  public static Banca getSingleton()
  { if (ref==null)  //non thread safe
       ref=new Banca();
    return ref;}
  Come versione precedente
}
```

Singleton (uso)

```
public class XXX {
…

metodo()
 {
  if (Banca.getSingleton().findConto(numero)!=null)

 …
   Banca.getSingleton().apriConto(nome);
 …

}
```

Riferimento



Singleton pro e contro

- Più complesso di `static`
- Maggiore flessibilità (possibile cambiare tempo di istanziazione,...)
- Rispetto a static è più facile da passare da unico a molti
- Potrebbe indurre troppe dipendenze: il metodo **`public static`** è visibile in tutta l'applicazione (anche `static` stesso problema)

Singleton uso tipico

- Log
- Logica accesso a DB
- ...

- La loro caratteristica di essere unici e visibile globalmente difficilmente cambia nel tempo

Alternative

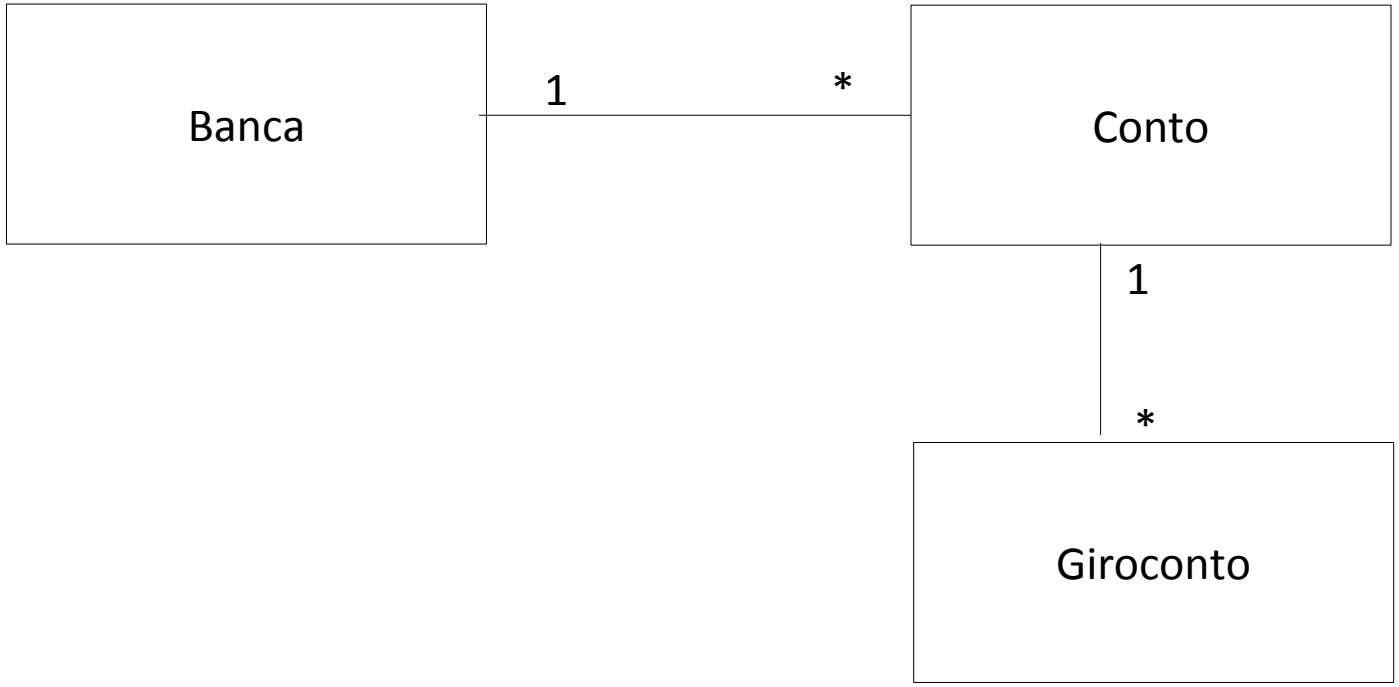
Scopo è ridurre la visibilità globale (resta unica istanza):

1) Passare un riferimento all'oggetto a tutti gli oggetti che hanno bisogno di accedere a Banca => aggiunge associazioni

   oppure

2) Utilizzare le associazioni già esistenti tra oggetti

Riferimento



Giriconto vuole accedere a Banca

Riferimento

```
public class Banca {
  Array<Conto> conti=...;

  public Conto findConto(int numero){...}
  …
  public int apriConto(String nome){
    int numero=prossimoNumero++;
    Conti.add(new Conto(nome, numero, this);
    return numero
  }
}
```

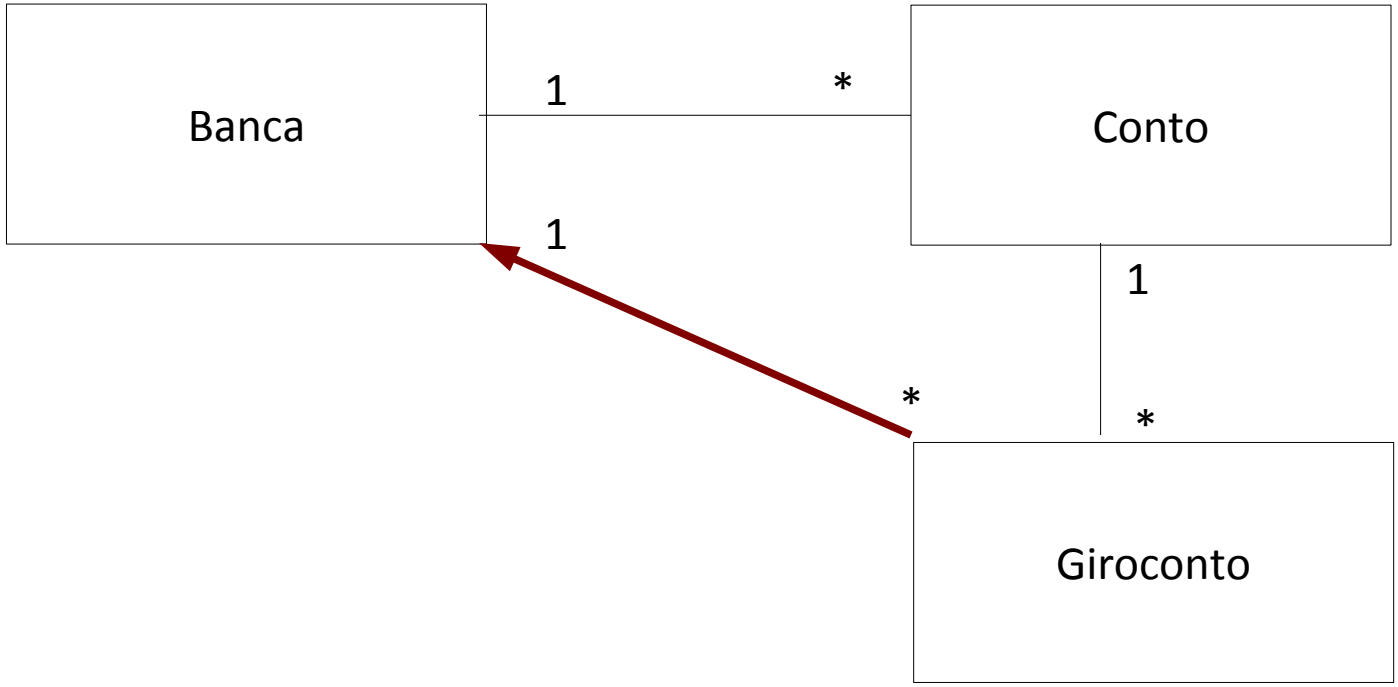
Riferimento

```
public class Conto {
  Banca banca;
  ...

 public Conto(String nome, int numero, Banca b){
     banca=b;
     ...
  }

  public void giroconta(double v, int altro)
  { …
    Giroconto g = new Giroconto(v,this,altro,banca);
   operazioni.add(g)
  }
}
```

Riferimento

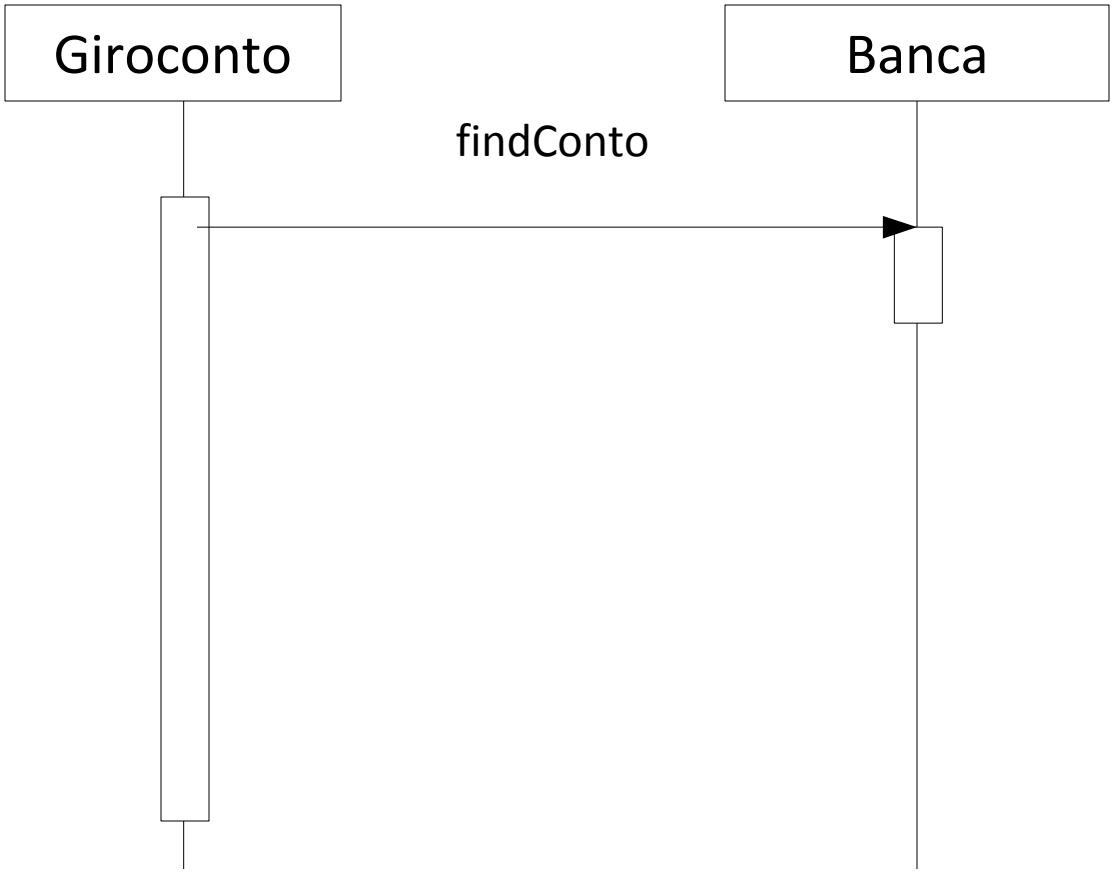
```
public class Giroconto extends Operazione {
   Conto conto;
   double importo;
   Calendar data;
   int altro;
   Banca banca;
   ...
   public Prelievo(double v,Conto c, int a, Banca b){
      importo=val; conto=c; data=now();
      banca=b;
      …
}
   public boolean perEsempio()
   { …
     Conto altro = banca.findConto(altro);
   }
```

Risultato: nuova associazione



Ora Giroconto ha un'associazioen con Banca

Diagramma di sequenza

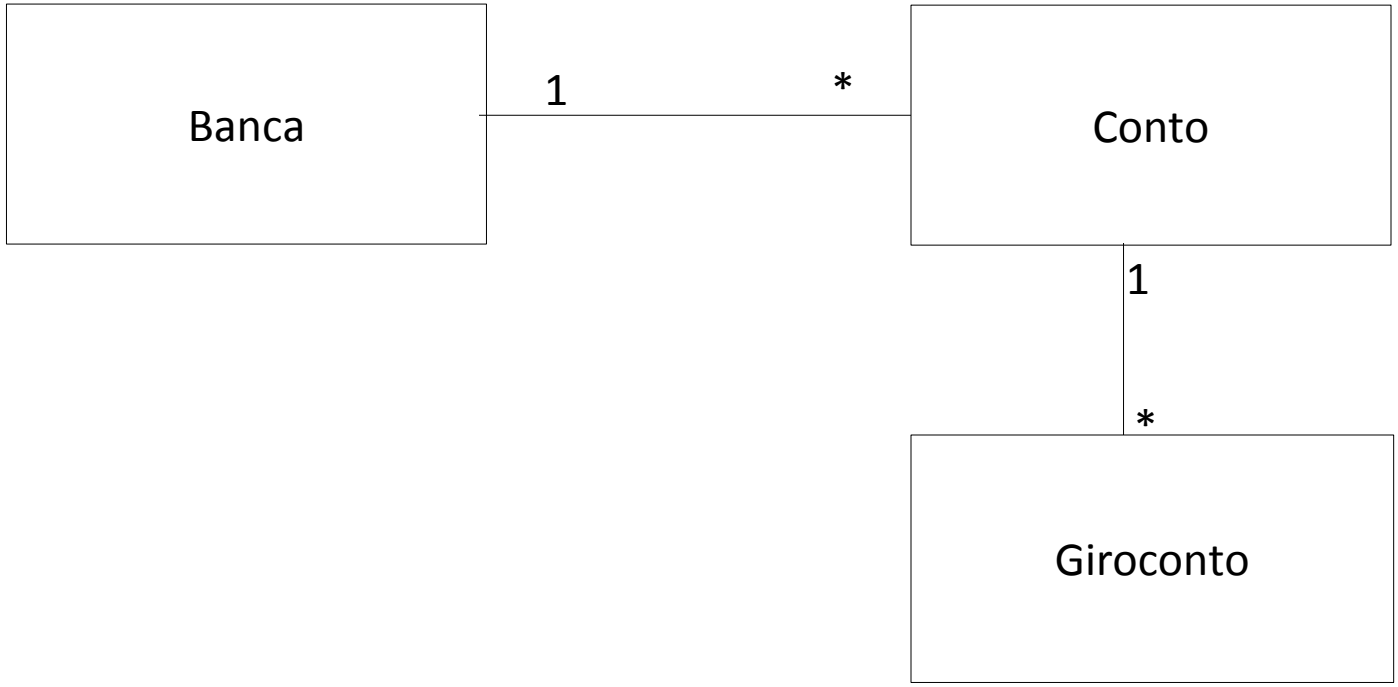


Perché riduce dipendenza?

- La Banca non è più globalmente visibile, ma solo agli oggetti che hanno il riferimento all'oggetto

  (nel nostro esempio Conto e Giroconto)
- Quindi, per esempio, è impossibile da un metodo di un'altra classe accedere direttamente alla Banca
- Ma posso farlo indirettamente usando le associazioni, vediamo come...

Usando associazioni



Giriconto vuole accedere a Banca

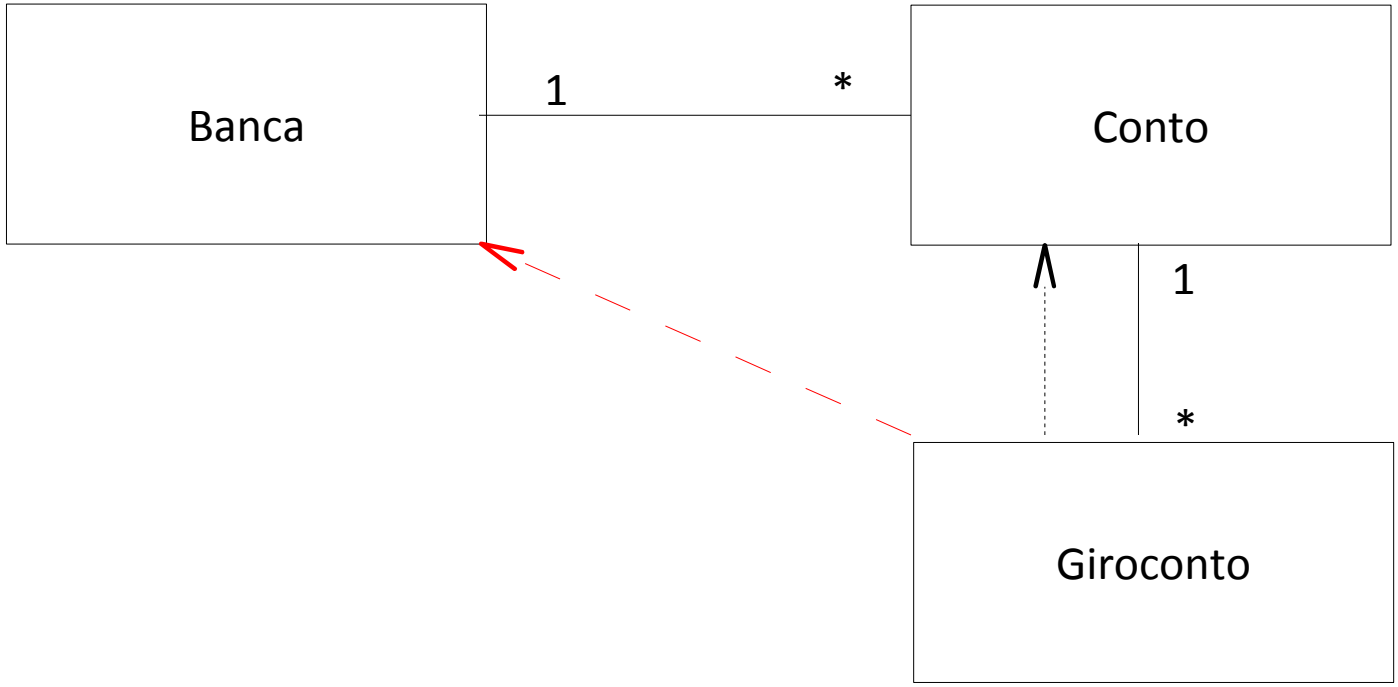
Usare le Associazioni

```
public class Banca {
… //come prima
}
public class Conto {
  …//come prima
  public void giroconta(double v, int altro)
  { …
    Giroconto g = new Giroconto(v,this,altro,banca);
   operazioni.add(g)
  }
  public Banca getBanca(){
    return banca;
  }
}
```

Usando le Associazioni

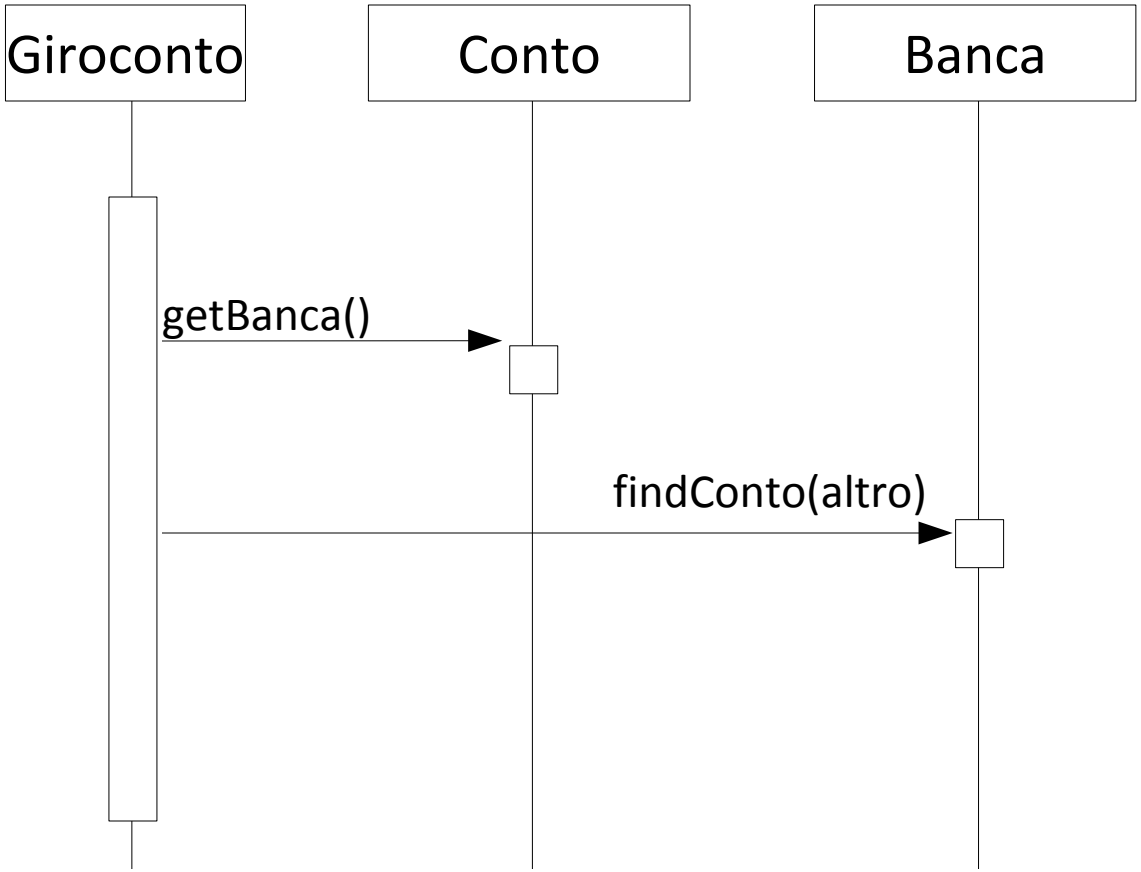
```
public class Giroconto extends Operazione {
   Conto conto;
   double importo;
   Calendar data;
   int altro;
   Banca banca;
   ...
   public Prelievo(double v,Conto c,int a, Banca b){
      importo=val; conto=c; data=now();
     banca=b;
      …
}
   public boolean perEsempio()
   { …
     Conto altro = conto.getBanca().findConto(altro);
   }
```

Usando associazioni



Giriconto accede a Banca

Diagramma di sequenza



Metodo fragile

```
conto.getBanca().findConto(altro);
```

- (Lunga) sequenza di visite
- Basta che una sola delle interfacce cambi per dover riscrivere questo metodo
- Non soddisfa legge di Demetra

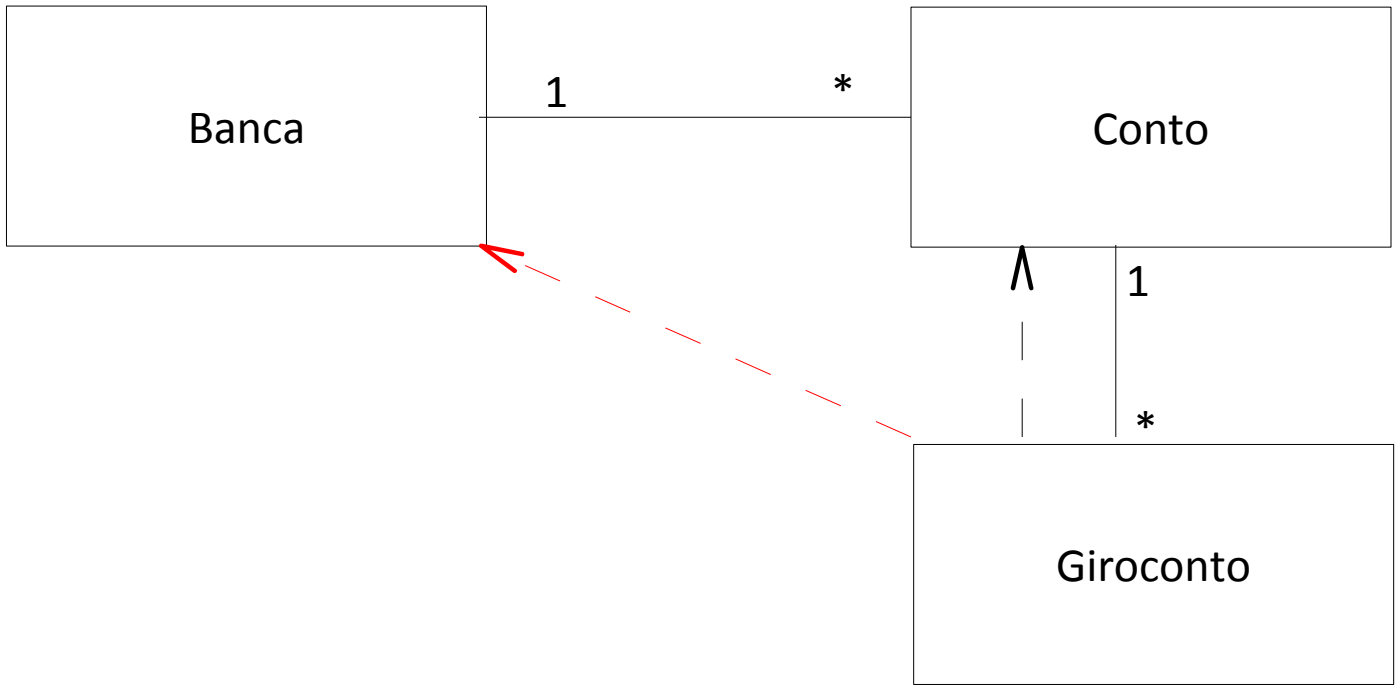
Legge di Demetra

Ogni metodo può invocare solo i metodi dei seguenti di oggetti:

1. variabili d'istanza
2. dei suoi parametri
3. di ogni oggetto che crea
4. dei suoi componenti diretti

- un oggetto NON dovrebbe invocare metodi di un oggetto ritornato da un altro metodo.

Dipendenza



Giriconto accede a Banca

Come ridurre dipendenza

- Aggiungere metodi appositi
- Effetto collaterale dell'applicazione della legge di Demetra è la creazione di molti metodi “wrapper”
- Anche con la legge di Demetra posso creare dipendenza tra oggetti “lontani”: basta passarli come parametri

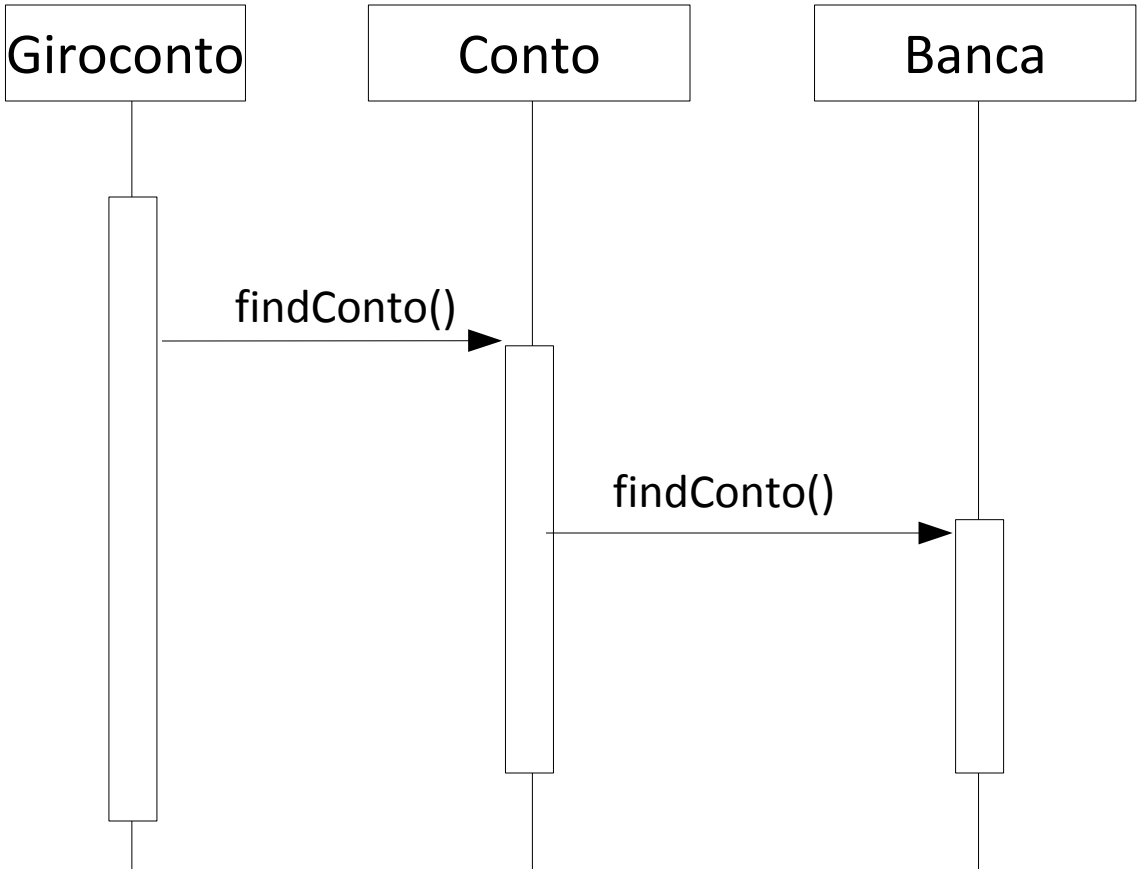
Aggiunta Metodi

```
public class Conto {
   Banca banca;
   ...
   public Conto findConto(int numero){
      return banca.findConto(numero);
   }

}
```

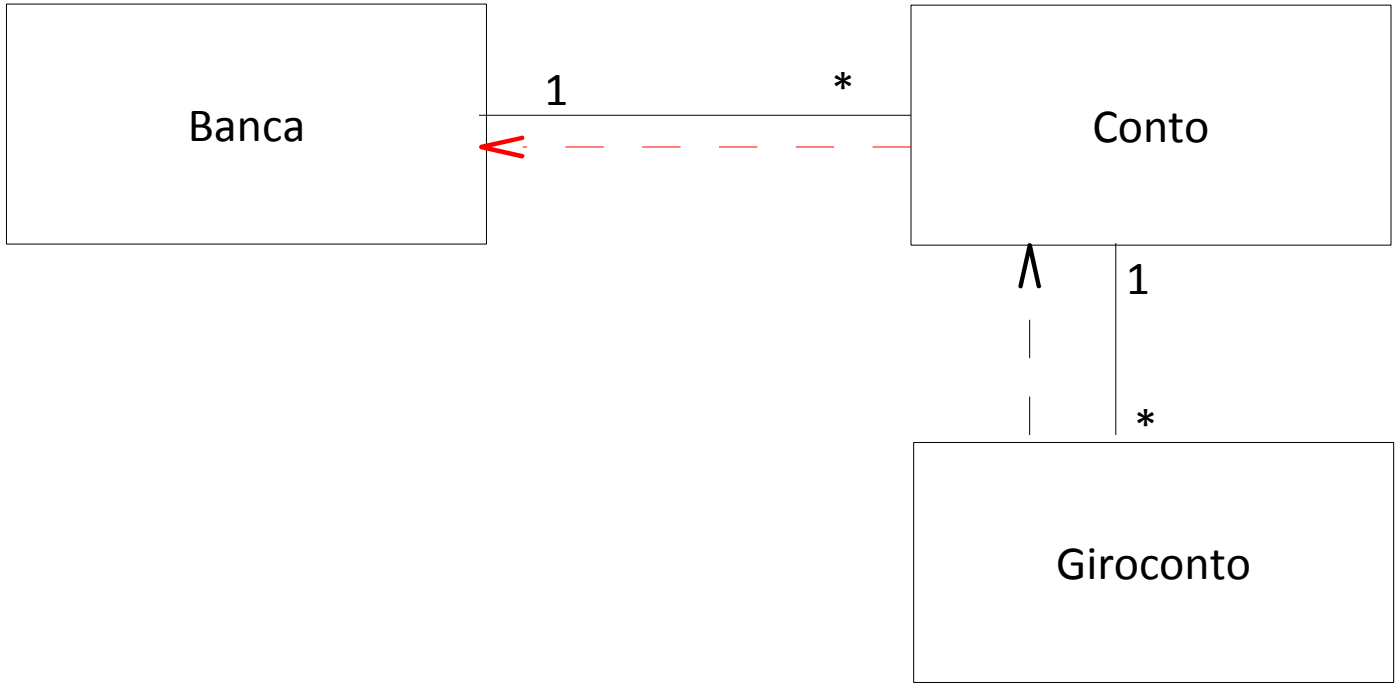
Aggiunta Metodi

```
public class Giroconto {

}
  public boolean perEsempio()
  { …
    conto.findConto(altro);
  }
}
```

Diagramma di sequenza



Dipendenza



Giriconto accede a Banca

Aggiunta metodi

+ Ho ridotto le dipendenze specie quelle lontane

- Ho aumentato la parte pubblica delle classi

- Ho reso meno coese le classi stesse facendogli fare cose che secondi IE fanno altre classi

Considerazioni generali

- Se ho bisogno di visibilità globale scelgo Singleton
- Altrimenti, cerco di ridurre le dipendenze.
- Ogni metodo ha dei pro e dei contro
- Magari riconsidero le scelte già fatte: esempio findConto lo faccio fare alla classe Conto

domande