Programmazione a Oggetti
Modulo B

Lezione 2

Dott. Alessandro Roncato

4/04/2013

Riassunto

Introduzione alla POO

Come dividere il codice

Dipendenza

Relazioni tra classi

UML: diagramma classi

Diagramma delle classi => codice

Il diagramma delle classi serve per capire come vanno “costruite” le classi. Come abbiamo detto, essendo UML semiformale, non definisce tutti i dettagli.

Nonostante questo, il diagramma delle classi implica dei vincoli nel codice che andiamo a scrivere.

Diagramma delle classi => codice


```
public class Conto {
   Cliente intestatario;

    public Conto(Cliente i){
       intestatario=i;
   }
   …
}
```

```
public class Cliente {
   Collection<Conto> conti;

    public Cliente(){
       conti = new ….;
   }
   …
}
```

Diagramma delle classi => codice

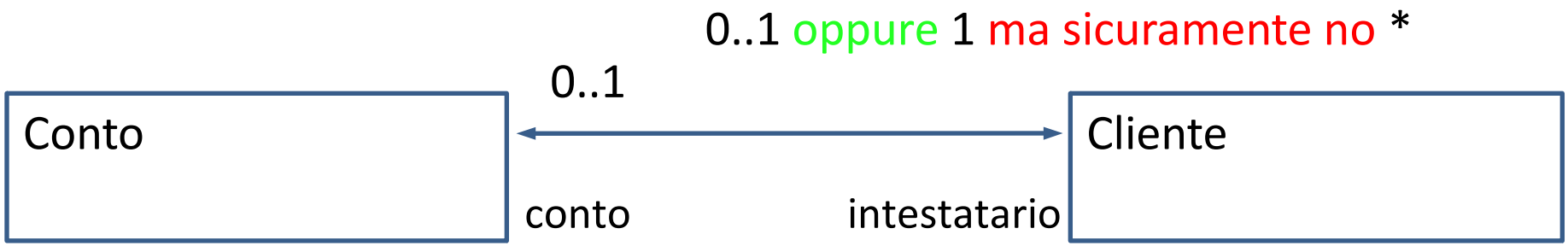

```
public class Conto {
   Cliente intestatario;

    public Conto(){
        …
   }
   …
   public void setIntestatario(…
}
```

```
public class Cliente {
   Conto conto;

    public Cliente(){
        …
   }
   …
   public void setConto(…
}
```

codice => Diagramma delle classi

```
public class Conto {
   Cliente intestatario;

    public Conto(Cliente c){
       intestatario=c;
   }
   …
   public void setIntestatario(…

}
```

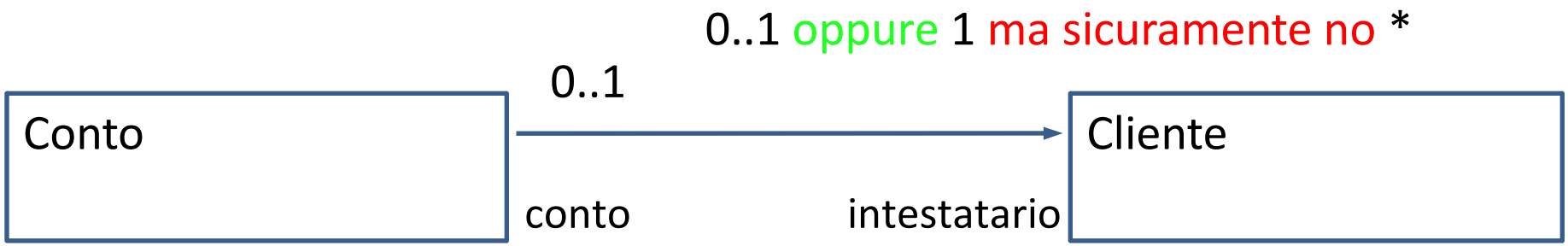
```
public class Cliente {
   Conto conto;

    public Cliente(){
       …
   }
   …
   public void setConto(…
}
```



codice => Diagramma delle classi

```
public class Conto {
    Cliente intestatario;

     public Conto(Cliente c){
        intestatario=c;
    }
    …
    public void setIntestatario(…

}
```

```
public class Cliente {


     public Cliente(){
        …
    }
    …

}
```



Esame !

Sicuramente un esercizio sul passaggio da Diagramma delle classi a codice e/o viceversa!

Diagrammi di sequenza

Scopo dei diagrammi di sequenza è illustrare come oggetti diversi collaborano per risolvere un certo problema.

In pratica, illustrano la *sequenza* di invocazioni dei diversi *metodi* nei diversi *oggetti* per un certa ***interazione*** dell'utente con l'applicazione stessa

Esempio

Diagramma di sequenza che descrive come gli oggetti collaborano quando il cassiere registra un prelievo su un conto corrente

Lo scopo è “ipotizzare” come potrebbe avvenire la collaborazione

I diagrammi vengono usati come strumento veloce per schematizzare il “funzionamento”

Diagramma di sequenza prelievo

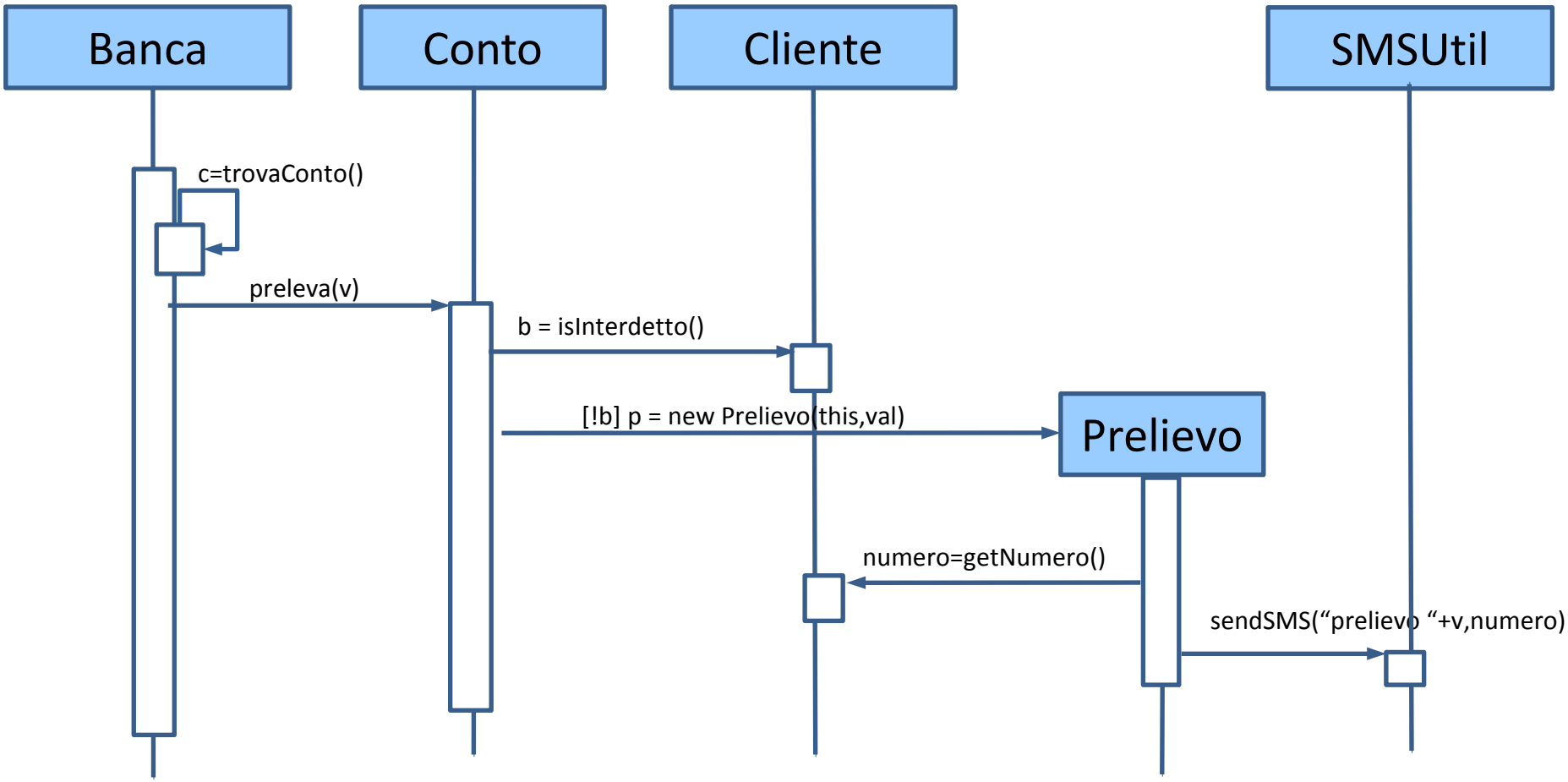
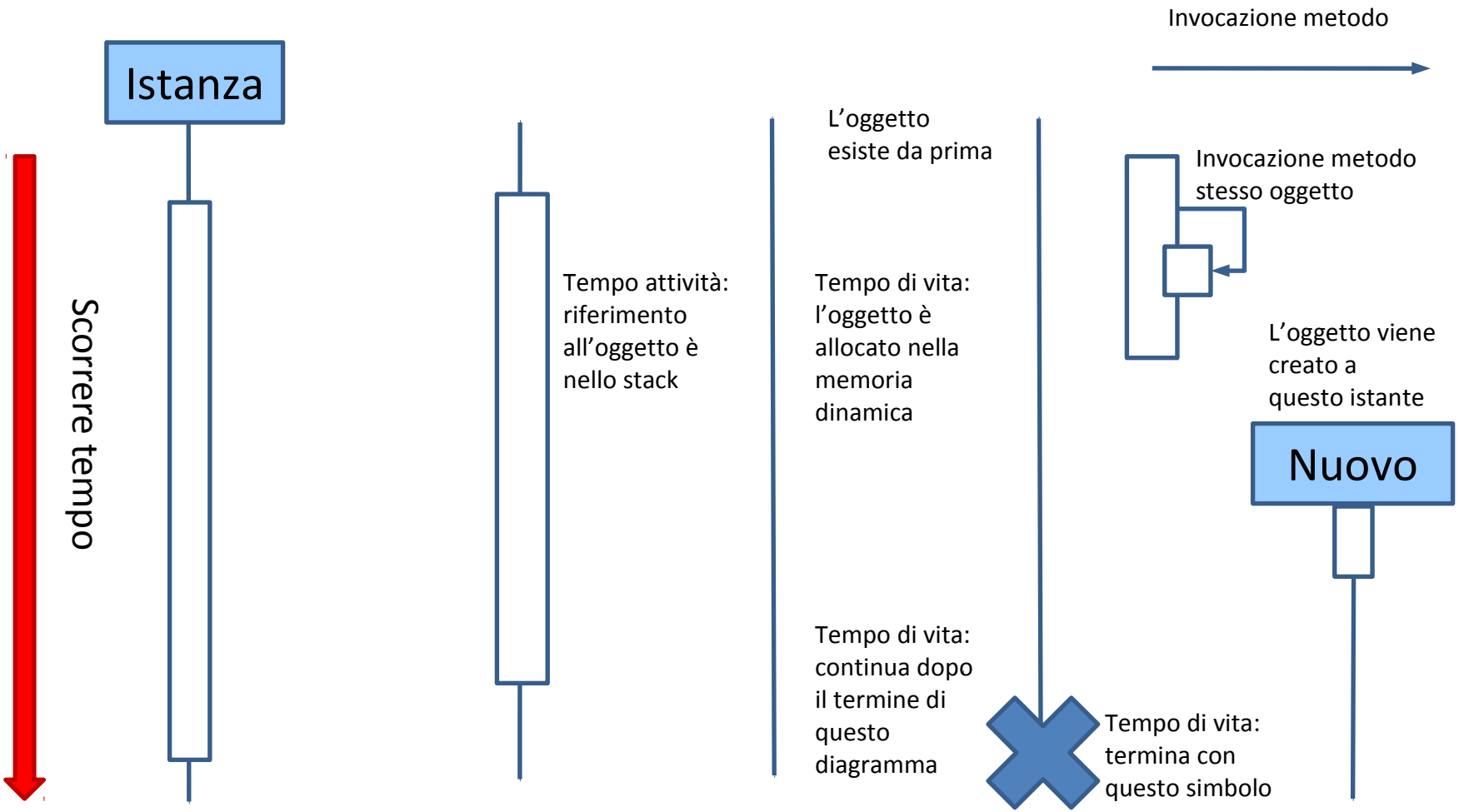
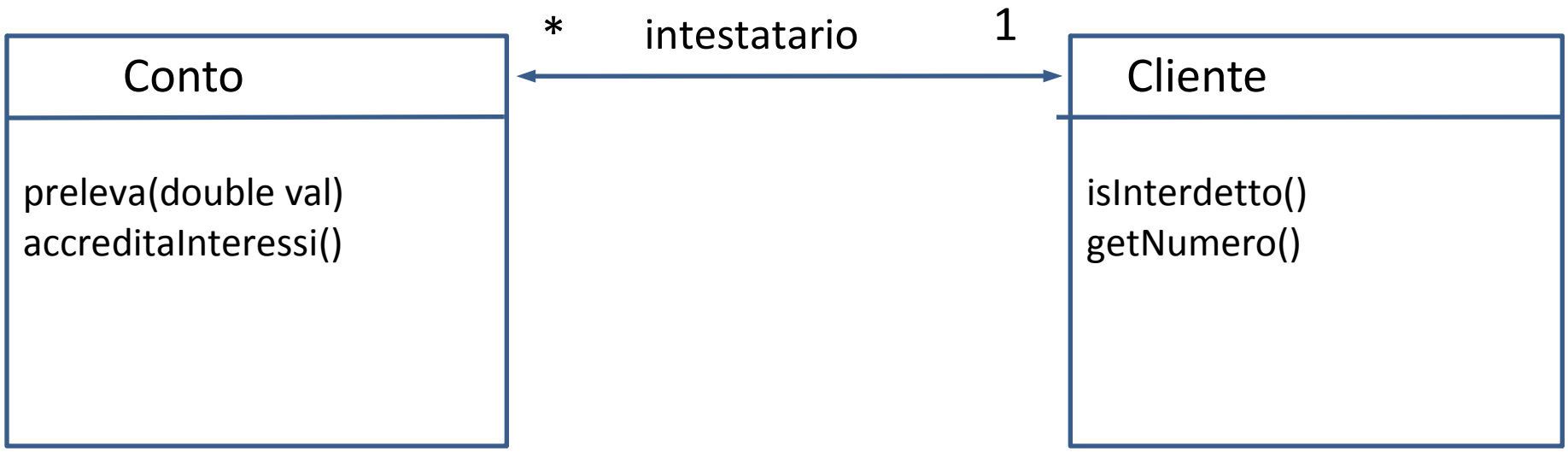


Diagramma di sequenza notazione



Aggiornamento Diagramma classi



Completamento Diagramma classi

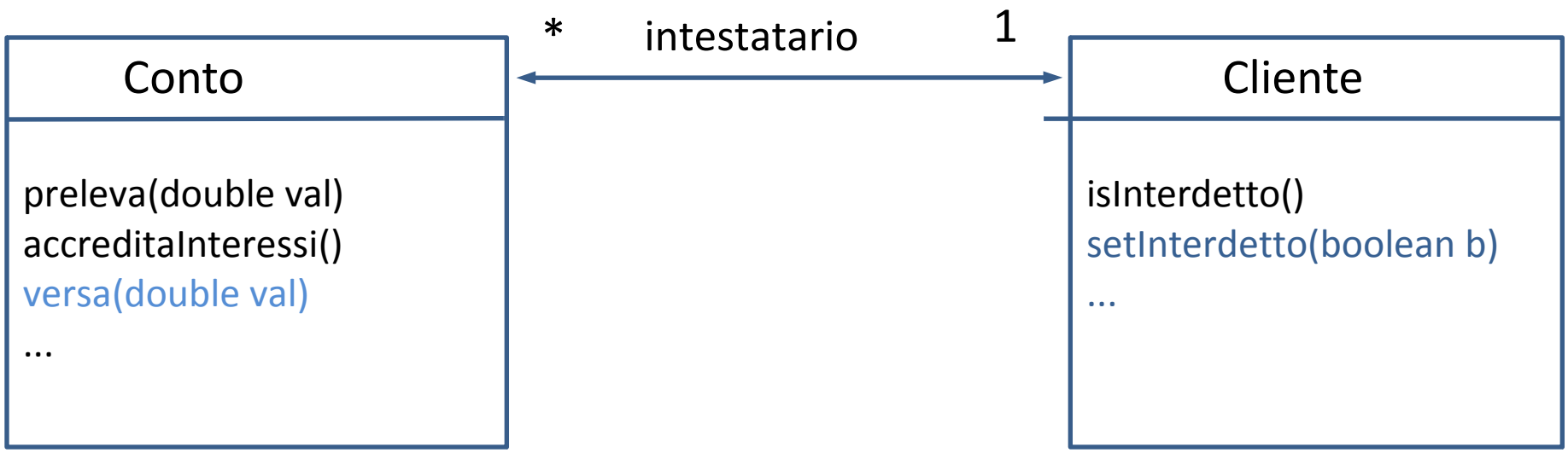


Diagramma casi d'uso

Servono per capire quali sono le interazioni dell'utente con l'applicazione

Bisogna capire quali sono i ruoli necessari con cui gli utenti utilizzaranno l’applicazione (es. cassiere, direttore, etc.)

Un caso d'uso rappresenta una singola interazione

Elenco Casi d'uso

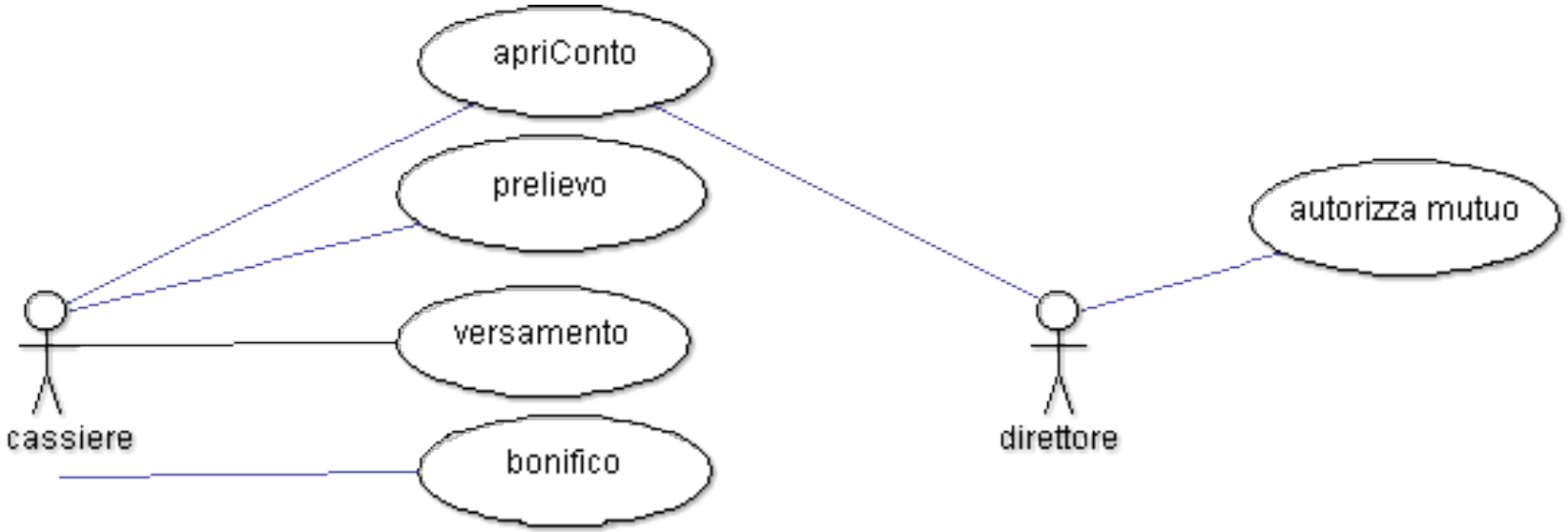


Diagramma casi d'uso

Attori (ruoli) sono soggetti esterni all'applicazione che stiamo descriveno (generalmente persone ma potrebbero essere sistemi/applicazioni esterne)

Casi d'uso: rappresentano una azione atomica gestita dall'applicazione.

I casi d'uso non corrispondono alle funzionalità: 1 funzionalità-> più casi d'uso

Come procedere

Abbozzare il diagramma delle classi

Elencare tutti casi d'uso

Abbozzare i diagrammi di sequenza per ogni casi d'uso

Aggiornare e completare il diagramma delle classi:

- Elencare i metodi
- Eventualmente aggiungere le classi mancanti

Come procedere 2

Per il momento lasciare da parte i problemi di interazione con la grafica o il parser.

I diagrammi faranno quindi riferimento al "Modello" dell'applicazione

Utilizzare Carta e matita per disegnare i diagrammi (NON usare applicazioni)

Varie

Per testare il codice che vedremo a lezione è possibile usare qualsiasi IDE; In laboratorio sono disponibili solo Netbeans, Eclipse e Jedit.

Nota che Netbeans, Eclipse e Jedit sono disponibili su tutte le piattaforme (Win, Mac, e Linux)

Per evitare problemi di compatibilità gli esempi di codice saranno indipendenti dalle piattaforme

Varie

Applicazione per disegnare con UML:

http://argouml-downloads.tigris.org/jws/argouml-latest-stable.jnlp

Design Patterns

I Design Patters sono soluzioni di progettazione standard e ben collaudate che possono essere usate in contesti diversi.

Vedremo oggi i seguenti:

- Information Expert
- Creator
- Null Object

Information Expert

D: Come assegnare le responsabilità?

R: All'oggetto che ha le informazioni per farlo!

Questo è uno dei pattern più “naturali” nella programmazione ad oggetti e dovrebbe essere la prima ipotesi di soluzione

Riduce la dipendenza e di solito aumenta la coesione

Esempio

- Cominciamo dal caso d'uso “fine mutuo”
- Oggetti Candidati: Banca, Cassiere, Conto, Cliente, Mutuo

Cassiere

- Pro:
  - ~~Nell'applicazione attiva la procedura per terminare il mutuo~~
  - Ha alcune informazioni necessarie
- Contro:
  - Nella PO gli oggetti sono passivi, quindi non è importante chi fa l'operazione, ma chi la “subisce”
  - Non ha molte informazioni necessarie

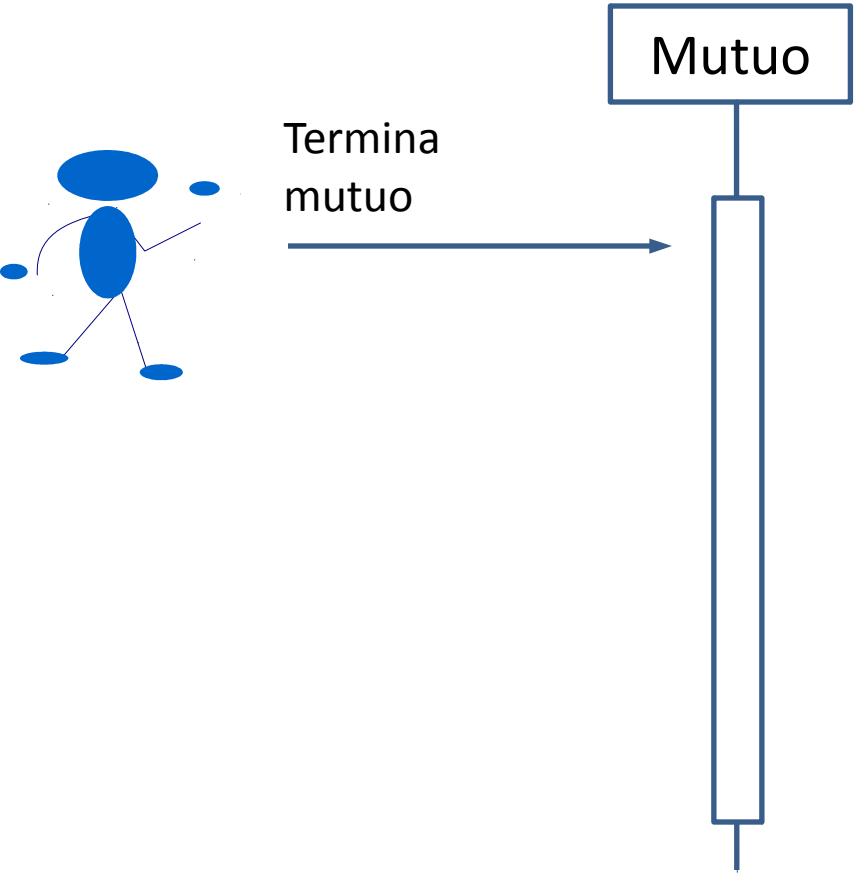
Termina mutuo?

- Banca: Ha alcune informazioni necessarie MA non ha molte altre informazioni necessarie
- Conto: ha molte informazioni
- Cliente: non ha molte informazioni
- Mutuo: ha tutte le informazioni per farlo
- Prelievo, Bonifico, etc. nessuna informazione
-

Mutuo

- E' l'oggetto che subisce l'operazione
- Ha tutte delle informazioni necessarie
-
- E' la nostra scelta secondo il pattern Information Expert

Diagramma di sequenza



E adesso???

Termina mutuo

Cosa bisogna fare per terminare il mutuo:

- Verificare che il mutuo sia ancora attivo (ovvero che non sia già stato chiuso;
- Se ci sono rate non saldate, bloccare l'operazione;
- Eliminare il bonifico automatico per il pagamento delle rate;
- Impostare il mutuo come chiuso

Verificare mutuo attivo

Chi fa questa verifica?

Chi ha le informazioni per farlo?

Il Mutuo stesso in maniera semplice può fare questa verifica

Il Mutuo può anche controllare che le rate siano state pagate

Esempio

```
public class Mutuo {
private Cliente cliente;
private Immobile immobile;
private BonificoAutomatico bonificoAutomatico;
private Calendar inizio;
private Calendar fine;
private Array<Rata> rateDaPagare;
public boolean isChiuso ()
{
     return fine!=null;

}
  public boolean isSaldato (){
     return rateDaPagare.size()==0;
  }
  ...
}
```

Eliminare Bonifico

L'oggetto Conto è quello che può fare questa cosa.

```
public class Conto {
private Array<BonificoAutomatico>
   bonificiAutomatici;
...

   public void elimina(BonificoAutomatico ba){
     bonificiAutomatici.remove(ba);
   }
   ...
```

Eliminare Bonifico

Ma come trovare l'oggetto Conto dal Mutuo?
Attraverso l'oggetto BonificoAutomatico!

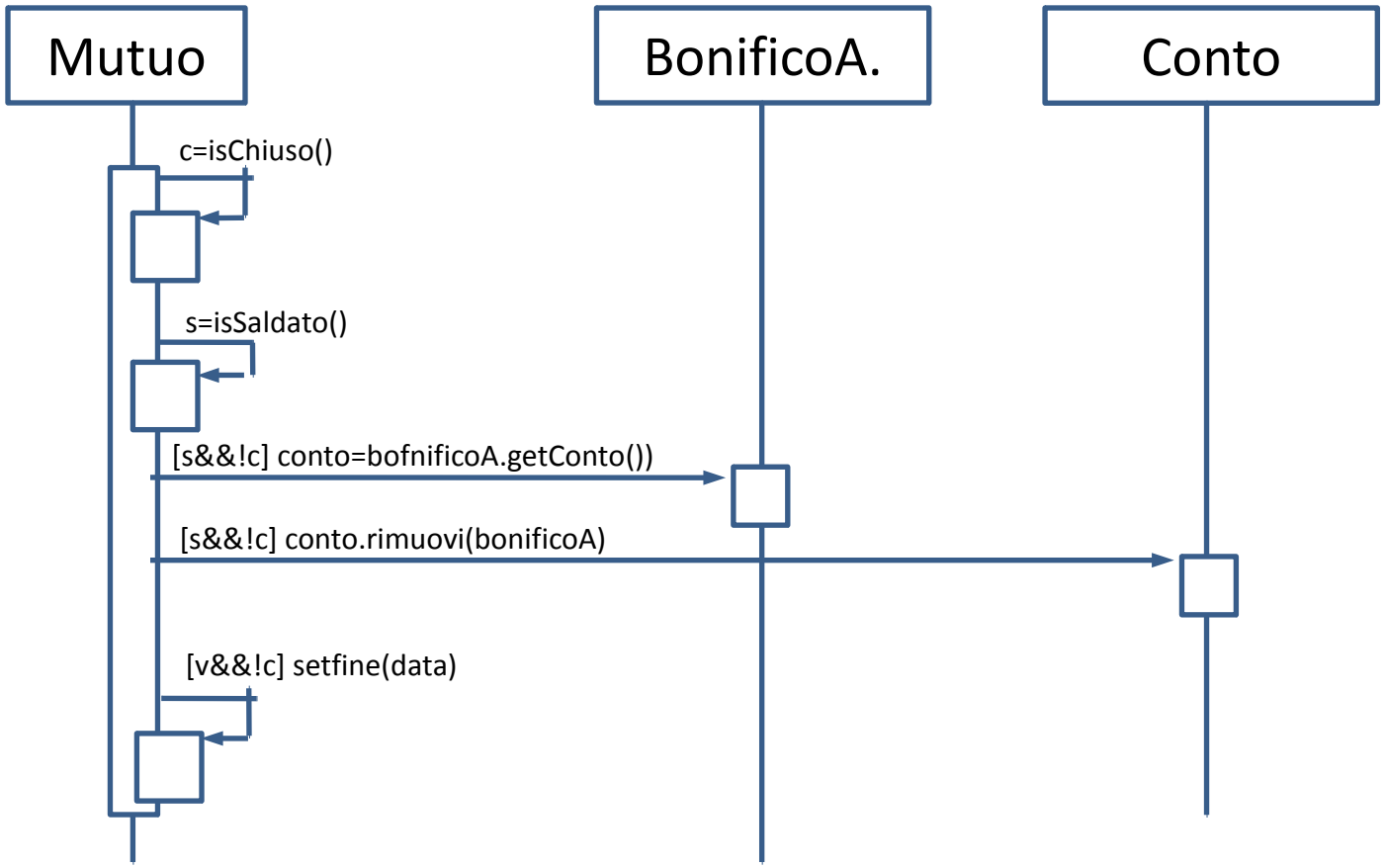
```
public class BonificoAutomatico {
private Conto conto;
private Conto benificiario;
private double importo;
private int numeroRate;
private int scadenza;
...

  public Conto getConto()
    return conto;
  }
  ...
```

Impostare la da di fine mutuo

L'oggetto Mutuo è quello che può fare questa cosa.

```
public class Mutuo {
  …
   private Calendar fine;

public void setFine(Calendar fine){
    this.fine=fine;
}

}
```

Diagramma di sequenza



Codice Java

```
public class Mutuo {
  …
  public boolean fine(Calendar data) {
        boolean c = isChiuso();
        boolean s = isSaldato();
        if (!c&&s) {
         Conto conto =
  bonificoAutomatico.getConto();
         conto.rimuovi(bonificoAutomatico);
         setFine(data);
        }
    }
}
```

Inizia Mutuo

- Abbiamo visto come trattare il caso d'uso
  Fine Mutuo
- Come ci comportiamo con il caso simmetrico
  Inizia Mutuo?
- La differenza sostanziale è che l'oggetto
  Mutuo deve ancora essere creato e quindi
  I.E. non può essere applicato

Chi crea gli oggetti?

- Il pattern Information Expert ci dice che le operazioni devono essere fatte dagli oggetti che sono più esperti.
- Anche nella creazione dovrebbe essere naturale che il più esperto sia l'oggetto stesso
- Ma finché non abbiamo un oggetto non possiamo applicare il pattern I.E.

Esempio static

```
public class Mutuo {

  public Mutuo(Cliente c, Immobile i) {
  …
  }

  public static crea(Cliente c, Immobile i){
    return new Mutuo(u,l);
  }

}
```

static

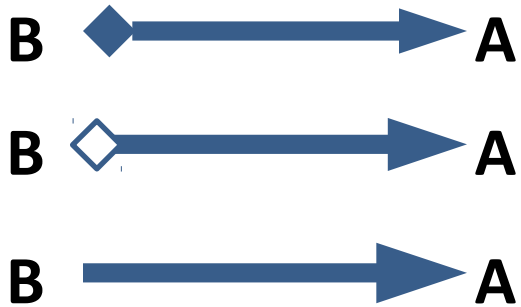
- Gli attributi e il codice statici non hanno bisogno di oggetti e quindi possono godere di vita propria
- `Mutuo.crea(cliente,immobile);`
- Non sfruttano i benefici della programmazione a oggetti
  - No polimorfismo
- Vedremo come PO riduce al minimo i metodi statici
- Sposta il problema: chi invoca crea?

Pattern Creator

D: Chi crea un oggetto A?

R: L'oggetto B che:

- C1: B contiene A
- C2: B aggrega A
- C3: B memorizza A
- C4: B che ha le informazioni per farlo



Esempio C1

- Chi crea gli oggetto Capitolo?
- Con Creator, l'oggetto il Libro è il candidato ideale per creare i Capitoli dato che Libro contiene i Capitoli e queste istanze di Capitolo non vengono usate nessun altra parte

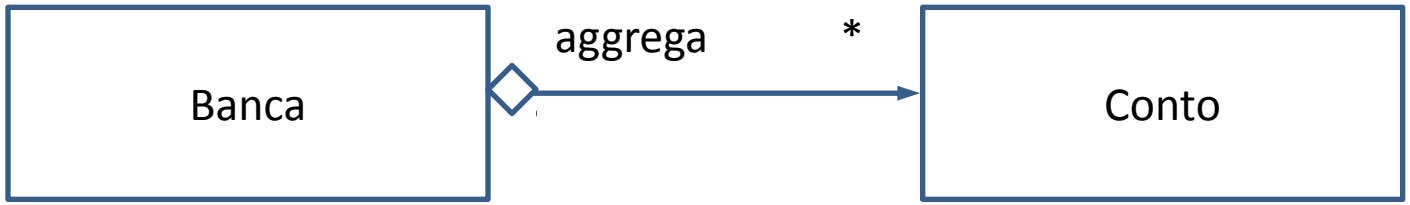
N.B. Un oggetto Capitolo ha lo stesso tempo di vita dell'oggetto Libro quindi viene creato e distrutto assieme al Libro stesso

Nel codice

```
public class Libro {
    Array<Capitolo> capitoli=new ...;
    ...
     public Libro(...){
     …
     for(...)
       capitoli.add(new Capitolo(...));

}
```

Esempio C2

- Chi crea gli oggetto Conto?
- Con Creator, l'oggetto Banca è il candidato ideale per creare l'oggetto Conto dato che la Banca lo contiene



N.B. Un oggetto Conto può essere riusato da altri oggetti .
Un oggetto Capitolo non può essere riusato da altri oggetti.

Nel codice

```
public class Banca {
    Array<Conto> conti=new ...;
    ...
     //in qualche metodo
     public int apriConto(Cliente c){
        int numero = prossimoNumero++;
         conti.add(new Conto(c,numero));
}
```

Esempio C3

- Chi crea l'oggetto Operazione?
- Con Creator, l'oggetto Conto è il candidato ideale per creare l‘Operazione dato che memorizza un riferimento all'oggetto Operazione

Nel codice

```
public class Conto {
    Array<Operazione> operazioni;
    ...
     //in qualche metodo
     operazioni.add(new Operazione(...));

}
```

domande