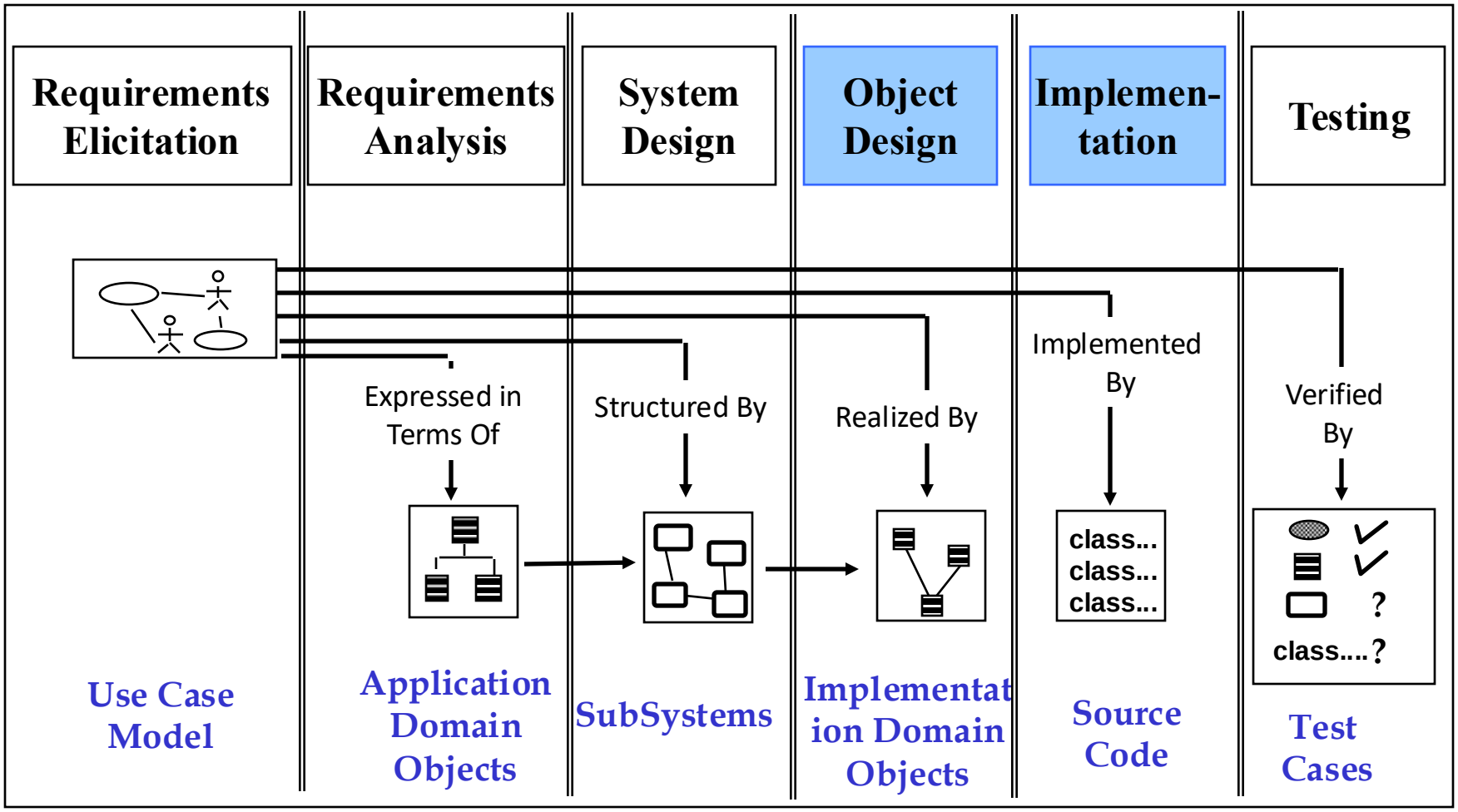
UNIVERSITÀ DEGLI STUDI DI SALERNO
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Chapter 10: Mapping Models to Code

Corso di Ingegneria del Software

Attività del Ciclo di Vita del Software



Object Design

- L'Object Design include:
  - Riuso: impiego dell'ereditarietà, di componenti off-the-shelf e dei design pattern per poter riutilizzare soluzioni esistenti.
  - Specifica dei servizi: descrizione precisa delle interfacce di ogni classe.

Oggetto della scorsa lezione

  - Ristrutturazione del modello ad oggetti: il modello di object design viene trasformato per migliorare la sua comprensibilità ed estensibilità.
  - Ottimizzazione del modello ad oggetti: trasformiamo il modello di object design in modo tale da soddisfare criteri di prestazione, quali tempo di risposta o utilizzo della memoria.

Oggetto di questa lezione

Alcuni problemi (1)

- Una volta specificate le interfacce delle classi e raffinata la relazione esistente tra le classi è possibile implementare il sistema che realizza i casi d'uso specificati durante l'analisi dei requisiti ed il system design.
- Gli sviluppatori possono incontrare diversi problemi di integrazione:
  - parametri non documentati possono essere stati aggiunti alle API in seguito a modifiche nei requisiti;
  - attributi addizionali possono essere stati aggiunti all'object model ma non nello schema di memorizzazione persistente (comunicazione inefficiente).
- Di conseguenza, il codice che si va ad implementare potrebbe essere lontano dal progetto originario e difficile da comprendere.

Alcuni problemi (2)

- Gli sviluppatori spesso eseguono trasformazioni sul modello ad oggetti:
  - trasformare localmente il modello degli oggetti per migliorarne modularità e prestazioni;
  - trasformare le associazioni del modello ad oggetti in collezioni di riferimenti ad oggetti, in quanto i linguaggi di programmazione non suportano il concetto di associazione;
  - scrivere codice per identificare e gestire le violazioni dei contratti se il linguaggio di programmazione non supporta i contratti;
  - rivedere la specifica dell'interfaccia per soddisfare nuovi requisiti richiesti dal cliente.
- Tutte queste attività non sono particolarmente complesse ma presentano aspetti ripetitivi e meccanici che possono indurre lo sviluppatore ad introdurre errori.

Una possibile soluzione

- Utilizzare un approccio disciplinato può evitare la degradazione del sistema quando lo sviluppatore deve:
  - ottimizzare il modello delle classi;
  - mappare le associazioni in collezioni;
  - mappare i contratti delle operazioni in eccezioni;
  - mappare il modello delle classi in uno schema di memorizzazione persistente.
- Vedremo un insieme di tecniche che consente di ridurre gli errori che si introducono durante le varie trasformazioni.

Outline

- **Tipi di trasformazioni**
  - Trasformazioni del modello a oggetti.
  - Refactoring.
  - Forward Enginering.
  - Reverse Enginering.
- **Tecniche di trasformazione di modello e forward engineering**
  - Ottimizzazione del modello delle classi.
  - Mapping delle associazioni su collezioni di oggetti.
  - Mapping dei contratti delle operazioni su eccezioni.
  - Mapping del modello delle classi su uno schema di memorizzazione persistente.

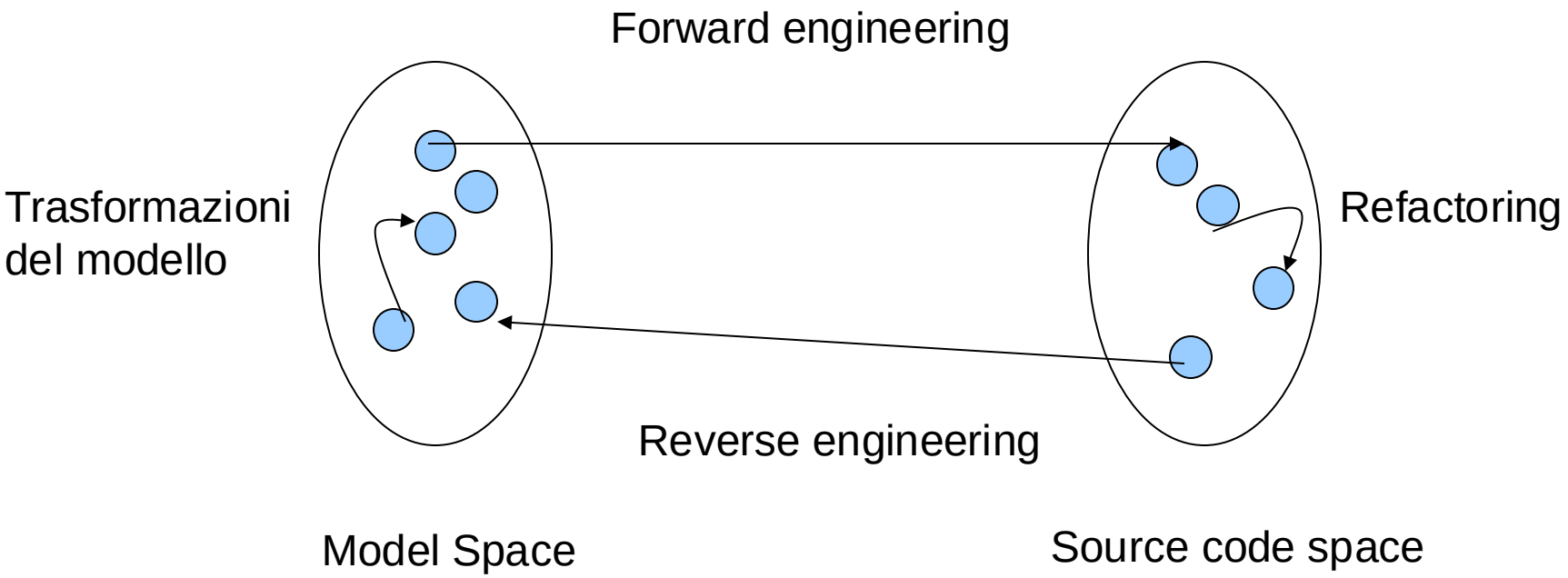
Trasformazioni

- Una **trasformazione** ha lo scopo di migliorare un aspetto del modello (ad es. la sua modularità) e preservare allo stesso tempo tutte le altre proprietà (ad es. le funzionalità).
- Generalmente una trasformazione
  - è localizzata
  - impatta su un numero ristretto di classi, attributi e operazioni
  - viene eseguita in una serie di semplici passi.

Tipi di trasformazioni

- Le **trasformazioni di modello** operano sul modello a oggetti.
  - Es.: conversione di un attributo semplice *(es: un indirizzo rappresentato da una stringa)* in una classe *(es: una classe con via, codice postale, comune, provincia e nazione)*
- **Refactoring:** trasformazioni che operano sul codice sorgente.
- **Forward Engineering:** produce un template del codice sorgente corrispondente al modello ad oggetti.
  - Molti costrutti (associazioni, attributi,...) possono essere meccanicamente mappati nei costrutti del codice sorgente (es. classi e dichiarazioni di campi in java), mentre il corpo dei metodi ed altri metodi privati vanno aggiunti dagli sviluppatori.
- **Reverse Engineering**: produce un modello a oggetti che corrisponde al codice sorgente.
  - Nonostante il supporto dei CASE è necessario un forte intervento umano per ricreare un modello accurato.

I 4 tipi di trasformazioni



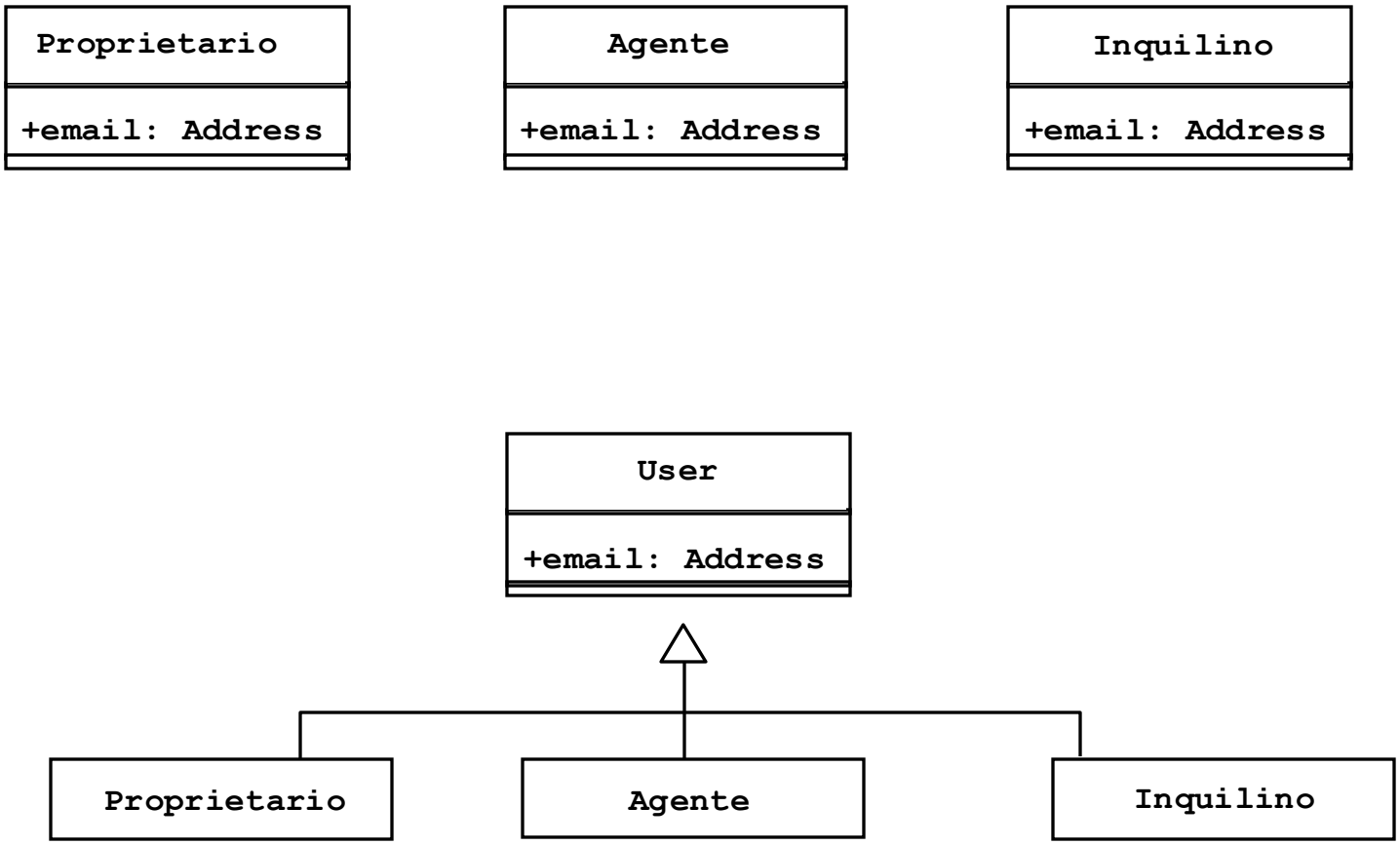
Tipi di trasformazioni

1. **Trasformazioni del modello a oggetti**
2. Refactoring
3. Forward Enginering
4. Reverse Enginering

Trasformazioni di modello

- Input: un modello ad oggetti.
- Output: un altro modello ad oggetti.
- Scopo: semplificare o ottimizzare il modello originale.
- Trasformazioni:
  - aggiungere, eliminare o rinominare classi, associazioni o attributi;
  - aggiungere o eliminare informazioni dal modello.
- Applicare una trasformazione di modello è un' attività meccanica…
- …tuttavia decidere quale trasformazione applicare e su quali classi farlo, richiede esperienza.

Trasformazioni di modello: esempio



L'attributo ridondante email viene eliminato creando una superclasse.

Tipi di trasformazioni

1. Trasformazioni del modello a oggetti.
2. **Refactoring.**
3. Forward Enginering.
4. Reverse Enginering.

Refactoring

- Il **refactoring** è una trasformazione del codice sorgente che ne migliora la leggibilità o la modificabilità senza cambiare il comportamento del sistema.
- Si focalizza su di uno specifico campo o metodo di una classe.
- Al fine di non cambiare il comportamento del sistema il refactoring deve essere attuato mediante piccoli passi incrementali intervallati da test:
  - l'utilizzo di test driver per ogni classe incoraggia lo sviluppatore a modificare il codice per migliorarlo.

Refactoring: esempio

- La trasformazione del modello di oggetti vista prima corrisponde ad una sequenza di tre refactoring:
  - **Passo 1:** spostare il campo **email** dalle sottoclassi alla superclasse.
  - **Passo 2:** spostare il codice di inizializzazione del campo **email** dalle sottoclassi alla superclasse.
  - **Passo 3:** spostare i metodi che manipolano il campo **email** dalle sottoclassi alla superclasse.

Refactoring: esempio – passo 1

Passo 1: spostare il campo **email** dalle sottoclassi alla superclasse.

1. Assicurarsi che i campi email siano equivalenti nelle tre classi.
2. Creare una classe pubblica **User.**
3. Rendere **User** il padre di **Agente**, **Inquilino**, **Proprietario.**
4. Aggiungere un campo email alla classe **User.**
5. Rimuovere il campo email da **Agente**, **Inquilino**, **Proprietario.**
6. Compilare e testare.

Prima del refactoring

```
public class Proprietario {
   private String email;
   //…
}
public class Agente {
    private String email;
    //…
}
public class Inquilino {
    private String email;
    //…
}
```

Dopo il refactoring

```
public class User {
      private String email;
      //…
}
public class Proprietario extends User{
      //…
}
public class Agente extends User{
     //…
}
public class Inquilino extends User{
     //…
}
```

Refactoring: esempio – passo 2

Passo 2: spostare il codice di inizializzazione del campo **email** dalle sottoclassi alla superclasse.

1. Aggiungere il costruttore ***User(String email)*** alla classe ***User***
2. Assegnare al campo ***email*** il valore passato nel parametro
3. Aggiungere la chiamata ***super(email)*** al costruttore della classe Proprietario
4. Compilare e testare
5. Ripetere i passi 2-4 per le classi ***Agente*** e ***Inquilino***

Refactoring: esempio – passo 2

Prima del refactoring

```
public class User {
      private String email;
}

public class Proprietario extends User{
   public Proprietario(String email) {
                this.email=email;
                 //…
      }
}

public class Agente extends User{
   public Agente(String email) {
                   this.email=email;
                   //…
      }
}

public class Inquilino extends User{
   public Inquilino(String email) {
                   this.email=email;
                   //…
      }
}
```

Dopo il refactoring

```
public class User {
   private String email;
   public User (String email) {
   this.email=email
   }
}

public class Proprietario extendsUser{
   public Proprietario(String email){
            super(email);
             //…
      }
}

public class Agente extends User{
      public Agente(String email)
   {super(email);//…}
}

public class Inquilino extends User{
      public Inquilino(String email) {
          super(email);
        //…
       }
}
```

Refactoring: esempio – passo 3

Passo 3: spostare i metodi che manipolano il campo **email** dalle sottoclassi alla superclasse.

1. Esaminare i metodi della sottoclasse *Proprietario* che usano il campo *email* e selezionare quelli che non usano campi o operazioni specifiche di *Proprietario*.
2. Copiarli nella superclasse *User* e ricompilarla.
3. Cancellare questi metodi dalla sottoclasse *Proprietario*.
4. Compilare e testare.
5. Ripetere i passi 1-4 per le sottoclassi *Agente* e *Inquilino*.

Refactoring: trasformazione del modello a oggetti

- Si può notare che il refactoring
  - include molti più passi della corrispondente trasformazione del modello ad oggetti;
  - alterna attività di testing con attività di trasformazione.
- Motivazione: Il codice sorgente include molti più dettagli -> più possibilità di introdurre errori.

Tipi di trasformazioni

1. Trasformazioni del modello a oggetti
2. Refactoring
3. **Forward Enginering**
4. Reverse Enginering

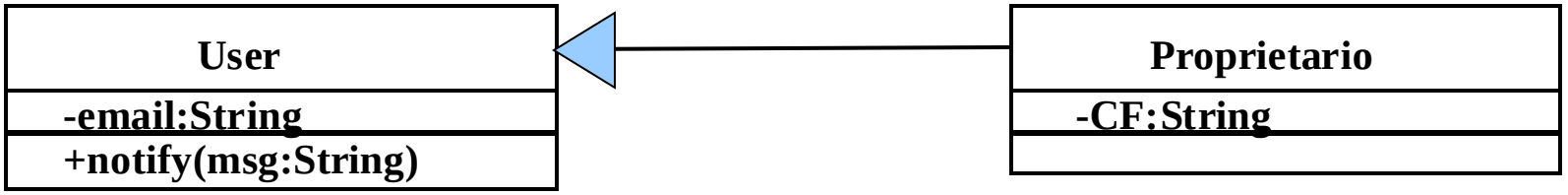
Forward Engineering

- **Input:** un insieme di elementi del modello.
- **Output:** un insieme di istruzioni di codice sorgente.
- **Obiettivi:**
  1. mantenere una forte corrispondenza fra il modello di design ad oggetti ed il codice;
  2. ridurre il numero di errori introdotti durante l'implementazione;
  3. ridurre gli sforzi di implementazione.

Forward Engineering: considerazioni

- Ogni **classe** del diagramma **UML** è mappata in una **classe JAVA**.
- La **relazione di generalizzazione** UML è mappata in **una istruzione extends** (della classe Proprietario).
- Ogni **attributo** del modello **UML** è mappato in **un campo privato** della classe Java e **due metodi pubblici** per settare e visualizzare i valori del campo.
- Gli sviluppatori possono raffinare il risultato della trasformazione con comportamenti aggiuntivi (es: controllare se un campo è positivo).
- Il codice risultante da una trasformazione di questo tipo è sempre lo stesso (eccetto i nomi degli attributi).
- Se le classi sono progettate in modo adeguato si introducono meno errori.

Forward Engineering: esempio



Codice sorgente ottenuto con forward engineering

```
public class User {
    private String email;
    public String getEmail() {
          return email;
    }
    public void setEmail(String value){
          email=value;
    }
    public void notify(String msg) {
    //…
    }
//…
}
```

```
public class Proprietario extends User{
    private String CF;
    public String getCF() {
          return CF;
    }
    public void setCF (String value) {
          CF=value;
    }
    //…..
}
```

I metodi per settare *email* e *CF* sono pubblici, i campi che rappresentano gli attributi sono privati

Tipi di trasformazioni

1. Trasformazioni del modello a oggetti
2. Refactoring
3. Forward Enginering
4. **Reverse Enginering**

Reverse Engineering

- **Input:** un insieme di elementi di codice sorgente.
- **Output:** un insieme di elementi del modello.
- **Obiettivo:** ricreare il modello per un sistema esistente.
- **Motivazioni:**
  - Il modello è andato smarrito o non è mai stato realizzato.
  - Il modello non è più allineato con il codice sorgente.

Reverse Engineering (2)

- La trasformazione di tipo Reverse Engineering è la trasformazione inversa del Forward Engineering:
  - crea una classe UML per ciascuna classe;
  - aggiunge un attributo per ciascun campo;
  - aggiunge un'operazione per ciascun metodo.
- Il reverse engineering non crea necessariamente il modello originario (forward engineering può far perdere informazioni come le associazioni).
- È supportata da CASE tool ma richiede comunque l'intervento dello sviluppatore per ricostruire un modello il più possibile vicino a quello originario.

Principi di trasformazione

- Per evitare che le trasformazioni possano introdurre errori difficili da rilevare e da correggere è necessario che ogni trasformazione:
  - migliori il sistema rispetto ad un singolo obiettivo di design. Es: tempo di risposta, coerenza.
  - cambi pochi metodi o poche classi alla volta.
  - venga applicata in modo isolato rispetto agli altri cambiamenti. (una alla volta)
  - venga seguita da un passo di validazione: dopo aver effettuato una trasformazione e prima di svolgerne un'altra, bisogna validare i cambiamenti.

Outline

- **Tipi di trasformazioni**
  - Trasformazioni del modello a oggetti.
  - Refactoring.
  - Forward Enginering.
  - Reverse Enginering.

- **Attività che coinvolgono trasformazioni**
  - Ottimizzazione del modello delle classi.
  - Mapping delle associazioni su collezioni di oggetti.
  - Mapping dei contratti delle operazioni in eccezioni.
  - Mapping del modello delle classi in uno schema di memorizzazione persistente.

Attività che coinvolgono trasformazioni

- **Ottimizzazione del modello di Object Design**: attività intrapresa al fine di soddisfare i requisiti di prestazione del modello del sistema.

  Tale attività include:

  - la riduzione della molteplicità delle associazioni per velocizzare le richieste;
  - l'aggiunta di associazioni ridondanti per migliorare l'efficienza;
  - l'aggiunta di attributi derivati per migliorare i tempi di accesso agli oggetti.

Attività che coinvolgono trasformazioni

- **Realizzare associazioni:** attività necessaria per mappare associazioni in costrutti di codice sorgente (ad es. riferimenti o collezioni di riferimenti).
- **Mappare contratti in eccezioni:** attività necessaria per descrivere il comportamento delle operazioni quando i contratti sono violati.
- **Mappare i modelli delle classi in uno schema di memorizzazione:** per realizzare la strategia di memorizzazione persistente prescelta durante il system design alcune classi devono essere mappate in uno schema di memorizzazione ad esempio utilizzando flat file o lo schema del DBMS selezionato.

Attività che coinvolgono trasformazioni

- **Ottimizzare il modello di Object Design**
  - Ottimizzare i cammini di accesso (access path)
  - Collassare gli oggetti in attributi
  - Ritardare le elaborazioni costose
  - Mantenere in una struttura dati temporanea il risultato di elaborazioni costose
- Mappare le associazioni in collezioni
- Mappare contratti in eccezioni
- Mappare l'Object Model in uno schema di memorizzazione persistente

Ottimizzare il modello di Object Design

- La traduzione diretta del modello di analisi in codice sorgente è spesso inefficiente.
- Il modello di analisi si focalizza sulle funzionalità del sistema e non tiene conto degli obiettivi di design identificati durante il system design.
- Compito dell'object design: trasformare il modello ad oggetti per soddisfare gli obiettivi di design identificati durante il system design (es. tempo di risposta, tempo di esecuzione, risorse di memoria, etc.).

Attività che coinvolgono trasformazioni

- **Ottimizzare il modello di Object Design**
  - **Ottimizzare i cammini di accesso (access path)**
  - Collassare gli oggetti in attributi
  - Ritardare le elaborazioni costose
  - Mantenere in una struttura dati temporanea il risultato di elaborazioni costose
- Mappare le associazioni in collezioni
- Mappare contratti in eccezioni
- Mappare l'Object Model in uno schema di memorizzazione persistente

Ottimizzare i cammini di accesso

- Al fine di **ottimizzare i cammini di accesso** dobbiamo eliminare il ritardo ottenuto a causa:
  - dell'attraversamento ripetuto di associazioni multiple;
  - dell'attraversamento di associazioni di tipo "molti";
  - della presenza di attributi mal collocati.

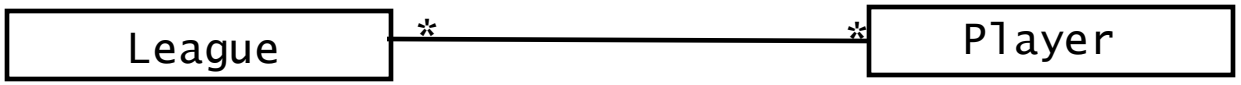
Ottimizzare i cammini di accesso

- **Attraversamento ripetuto di associazioni multiple**
  - Le operazioni più frequenti non dovrebbero richiedere molti attraversamenti di associazioni, ma dovrebbero avere un'associazione diretta fra l'oggetto che interroga e l'oggetto interrogato.
  - **Soluzione:** aggiungere l'associazione fra i due oggetti.
  - La frequenza dei path di accesso è facile da determinare nell'interface engineering e re-engineering. Mentre può essere determinata durante il testing di sistema nel caso di greenfield engineering.

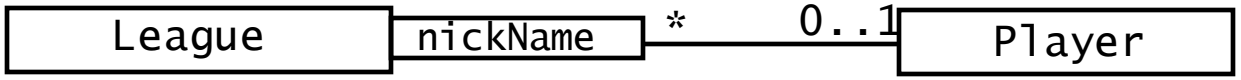
Ottimizzare i cammini di accesso (2)

- **Attraversamento di associazioni di tipo "molti"**
  - Provare a far decrescere il tempo di ricerca riducendo il tipo "molti" (*) al tipo "uno" utilizzando un' associazione qualificata.

Object design model before transformation



Object design model before forward engineering



  - Se ciò non è possibile, si può provare a ordinare o indicizzare gli oggetti sul lato "molti" per migliorare il tempo di accesso.

Ottimizzare i cammini di accesso (3)

- **Attributi mal collocati**
  - **Problema:** modellazione eccessiva. Durante l'analisi possono essere state individuate delle classi non rilevanti, i cui attributi sono chiamati solo dai metodi **set()** e **get()**.
  - **Soluzione:** si possono spostare gli attributi nella classe chiamante. Alcune classi, così, possono non servire più ed essere rimosse dal modello.

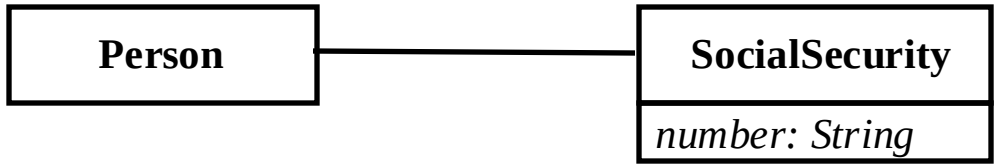
Attività che coinvolgono trasformazioni

- **Ottimizzare il modello di Object Design**
  - **Ottimizzare i cammini di accesso (access path)**
  - **Collassare gli oggetti in attributi**
  - Ritardare le elaborazioni costose
  - Mantenere in una struttura dati temporanea il risultato di elaborazioni costose
- Mappare le associazioni in collezioni
- Mappare contratti in eccezioni
- Mappare l'Object Model in uno schema di memorizzazione persistente

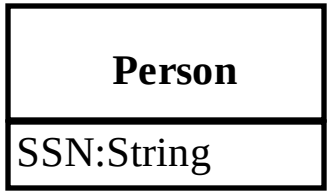
Collassare gli oggetti in attributi

- Dopo che il modello è stato ottimizzato diverse volte alcune classi possono avere pochi attributi o comportamenti.
- Queste classi, se sono associate ad una sola altra classe, possono essere collassate in attributi.

Object design model prima della trasformazione



Object design model dopo la trasformazione



Attività che coinvolgono trasformazioni

- **Ottimizzare il modello di Object Design**
  - **Ottimizzare i cammini di accesso (access path)**
  - **Collassare gli oggetti in attributi**
  - **Ritardare le elaborazioni costose**
  - Mantenere in una struttura dati temporanea il risultato di elaborazioni costose
- Mappare le associazioni in collezioni
- Mappare contratti in eccezioni
- Mappare l'Object Model in uno schema di memorizzazione persistente

Ritardare le elaborazioni costose

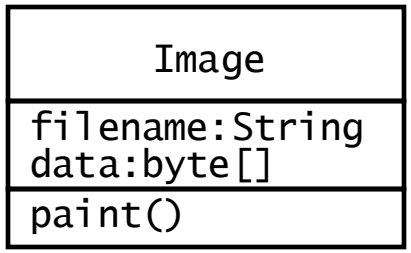
- Alcuni oggetti sono costosi da creare
  - la loro creazione può spesso essere ritardata finché il loro contenuto è effettivamente necessario.
- Esempio: considerare un oggetto che rappresenti un'immagine memorizzata come file
  - caricare tutti i pixel che costituiscono l'immagine dal file è costoso.

Ritardare le elaborazioni costose (2)

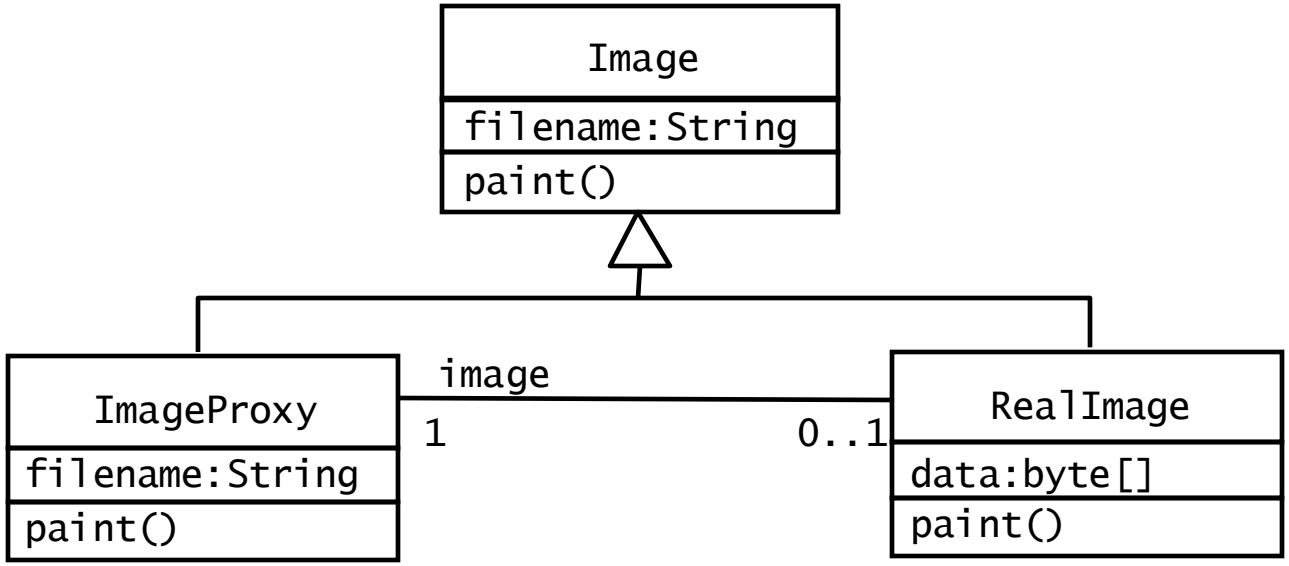
- Si utilizza il design pattern Proxy:
  - Un oggetto ImageProxy sostituisce l'oggetto Image e fornisce la stessa interfaccia dell'oggetto Image.
  - Operazioni semplici come larghezza() ed altezza() saranno gestite da ImageProxy.
  - Quando Image deve essere disegnato, ImageProxy carica i dati dal disco e crea l'oggetto RealImage.
  - Se il client non invoca la paint(), l'oggetto RealImage non è creato, risparmiando un notevole tempo di calcolo.

Ritardare le elaborazioni costose (3)

Object design model before transformation



Object design model after transformation



Tecniche di trasformazione di modello

- **Ottimizzare il modello di Object Design**
  - Ottimizzare i cammini di accesso (access path)
  - Collassare gli oggetti in attributi
  - Ritardare le elaborazioni costose
  - **Mantenere in una struttura dati temporanea il risultato di elaborazioni costose**
- Mappare le associazioni in collezioni
- Mappare contratti in eccezioni
- Mappare l'Object Model in uno schema di memorizzazione persistente

Memorizzare il risultato di elaborazioni costose

- Alcuni metodi sono invocati diverse volte, ma il loro risultato non cambia o cambia raramente.
  - Ridurre il tempo di elaborazione di tali metodi può portare una riduzione dei tempi di risposta.
- **Soluzione:** memorizzare il risultato in un attributo privato.
  - Si migliora il tempo di risposta, ma si consuma più spazio per memorizzare l'informazione ridondante.

Teniche di Forward Engineering

- **Ottimizzare il modello di Object Design**
- **Mappare le associazioni in collezioni**
  - Associazioni uno-a-uno unidirezionali e bidirezionali
  - Associazioni uno-a-molti e molti-a-molti
  - Classi associative
- Mappare contratti in eccezioni
- Mappare l'Object Model in uno schema di memorizzazione persistente

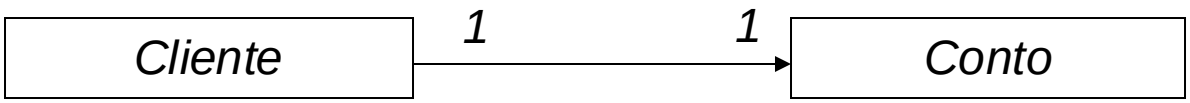
Mappare associazioni in collezioni

- Le associazioni sono concetti UML che denotano collezioni di link bidirezionali tra due o più oggetti.
- I linguaggi orientati agli oggetti non forniscono il concetto di associazione, ma forniscono i concetti di:
  - riferimento: un oggetto memorizza un "handle" verso un altro oggetto.
    - nota: i riferimenti sono unidirezionali e collegano due oggetti.
  - collezione: possono essere memorizzati, ed eventualmente ordinati, i riferimenti a diversi oggetti.
- Durante l'object design trasformiamo le associazioni in termini di riferimenti considerando:
  - la molteplicità e la direzione delle associazioni.

Teniche di Forward Engineering

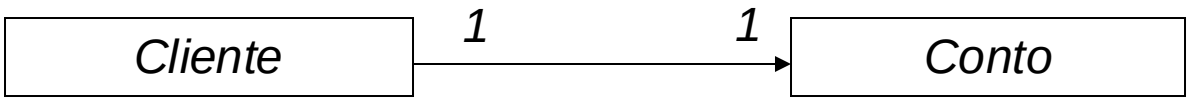
- **Ottimizzare il modello di Object Design**
- **Mappare le associazioni in collezioni**
  - **Associazioni uno-a-uno unidirezionali e bidirezionali**
  - Associazioni uno-a-molti e molti-a-molti
  - Classi associative
- Mappare contratti in eccezioni
- Mappare l'Object Model in uno schema di memorizzazione persistente

Associazioni uno-a-uno unidirezionali



- Se la classe *Cliente* chiama le operazioni della classe *Conto*, per sapere tutti i movimenti che gli sono stati addebitati e la classe *Conto* non chiama mai operazioni della classe *Cliente*, l'**associazione** è ***unidirezionale***.

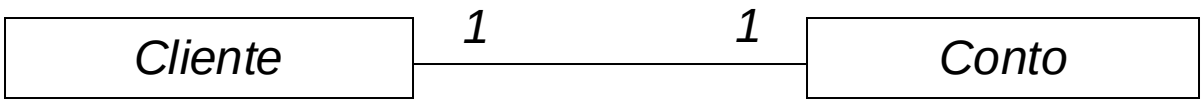
Associazioni uno-a-uno unidirezionali




```
public class Cliente {

   private Conto conto;

   public Cliente() {
        conto = new Conto();
   }

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

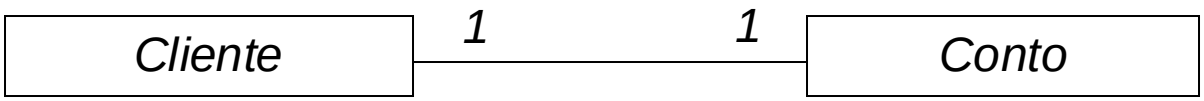
- L'associazione si traduce inserendo un campo conto nella classe Cliente che referenzia l'oggetto Conto.
- Un valore nullo nel campo conto può essere presente solo quando l'oggetto Cliente è creato.

Associazioni uno-a-uno bidirezionali



- Le **associazioni bidirezionali** sono complesse ed introducono una dipendenza reciproca fra le classi.
- Supponiamo di modificare la classe *Conto* in modo tale che il nome del *Conto* da visualizzare dipenda dal nome del *Cliente*

Associazioni uno-a-uno bidirezionali




```
public class Cliente {

   private Conto conto;

   public Cliente() {
      conto = new Conto(this);
   }

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

```
public class Conto {

   private Cliente owner;

   public Conto(Cliente owner) {
        this.owner = owner;
   }

   public Cliente getOwner() {
        return owner;
   }
}
```

I valori iniziali di conto e owner sono inizializzati e non vengono più modificati.

Associazioni unidirezionali o bidirezionali ?

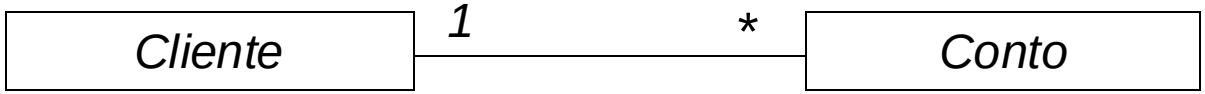
- Deve essere valutato lo specifico contesto.
- A volte le associazioni unidirezionali vengono trasformate in associazioni bidirezionali.
- È consigliabile rendere sistematicamente gli attributi privati e fornire i metodi getAttribute() e setAttribute() per accedere ai riferimenti.
- Questo minimizza le modifiche quando si passa da una associazione unidirezionale ad una bidirezionale e viceversa.

Attività che coinvolgono trasformazioni

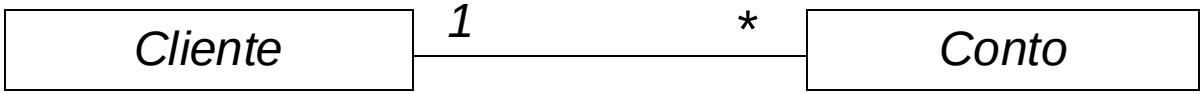
- Ottimizzare il modello di Object Design
- **Mappare le associazioni in collezioni**
  - Associazioni uno-a-uno unidirezionali e bidirezionali
  - **Associazioni uno-a-molti e molti-a-molti**
  - Classi associative
- Mappare contratti in eccezioni
- Mappare l'Object Model in uno schema di memorizzazione persistente

Associazioni uno-a-molti

- Le **associazioni uno-a-molti** non possono essere realizzate usando un singolo riferimento.
- Esempio: supponiamo che ad un cliente possano corrispondere più conti.
  - Poiché i conti non hanno un ordine specifico possiamo usare un insieme di riferimenti, *conti*, per modellare la parte "molti" dell'associazione.



Associazioni uno-a-molti (2)




```
public class Cliente {

    private Set conti;

    public Cliente() {
        conti = new HashSet();
    }

    public void addConto (Conto c) {
          conti.add(c)
          c.setOwner(this);
    }

    public void removeConto (Conto c)
    {
          conti.remove(c);
          c.setOwner(null);
    }
}
```

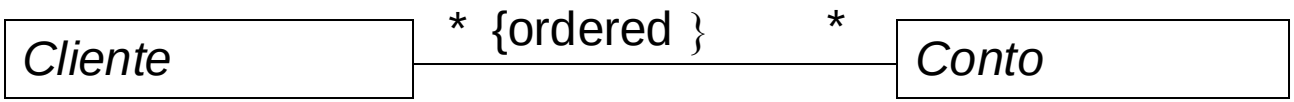
```
public class Conto {

    private Cliente owner;

    public void setOwner (Cliente newOwner){
         if (owner != newOwner) {
           Cliente old = owner;
           owner = newOwner;
           if (newOwner!= null)
                newOwner.addConto(this);
           if (old != null)
                 old.removeConto(this);
          }
    }
}
```

Associazioni uno-a-molti (3)

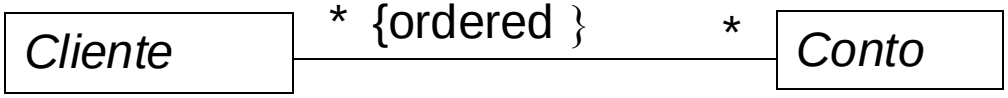
- L'associazione è stata realizzata in modo bidirezionale.
- Sono stati usati i metodi *addConto()*, *removeConto()* e *setOwner()* per aggiornare i campi *conti* e *owner* nelle classi *Cliente* e *Conto*, rispettivamente.
- Se *conti* deve essere ordinato, si usa un *List* invece di un *Set*.
- Per minimizzare i cambiamenti nelle interfacce in caso di cambiamenti sui vincoli dell'associazione è conveniente usare come tipo di ritorno per *getConto()* il tipo *Collection*, una superclasse comune di *List* e *Set*.

Associazioni molti-a-molti

- **Entrambe** le classi hanno campi che sono collezioni di riferimenti ed operazioni per mantenere queste collezioni consistenti.
- Supponiamo che un conto possa essere intestato a più clienti



Associazioni molti-a-molti (many-to-many)




```
public class Cliente {

    private List conti;

    public Cliente() {
        conti = new ArrayList();
    }

    public void addConto (Conto c) {
          if (!conti.contains(c)) {
                    conti.add(c);
                    c.addCliente(this);
          }
    }
}
```

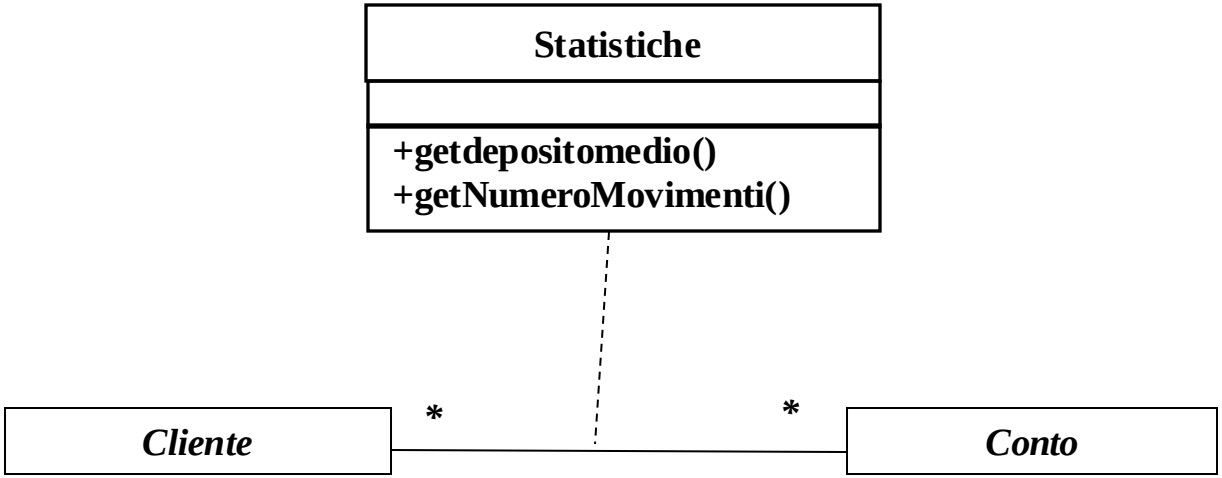
```
public class Conto {

    private List clienti;

    public Conto() {
        clienti = new ArrayList();
    }

    public void addCliente (Cliente c) {
          if (!clienti.contains(c) {
                    clienti.add(c);
                    c.addConto(this);
          }
    }
}
```

Attività che coinvolgono trasformazioni

- Ottimizzare il modello di Object Design
- **Mappare le associazioni in collezioni**
  - Associazioni uno-a-uno unidirezionali e bidirezionali
  - Associazioni uno-a-molti e molti-a-molti
  - **Classi associative**
- Mappare contratti in eccezioni
- Mappare l'Object Model in uno schema di memorizzazione persistente

Classi associative

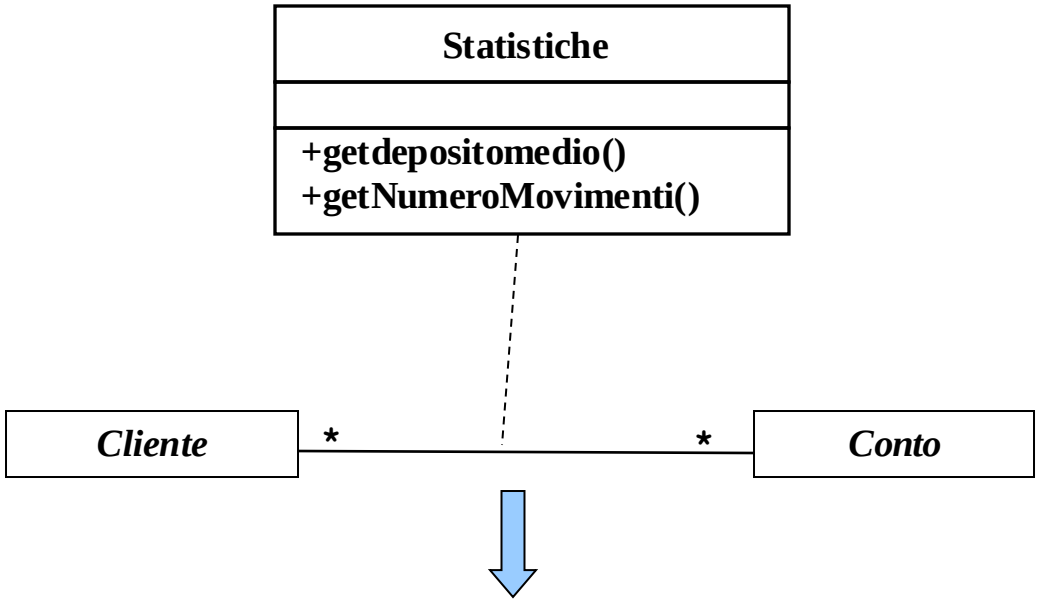
- Le classi associative sono utilizzate in UML per contenere gli attributi e le operazioni di una associazione.



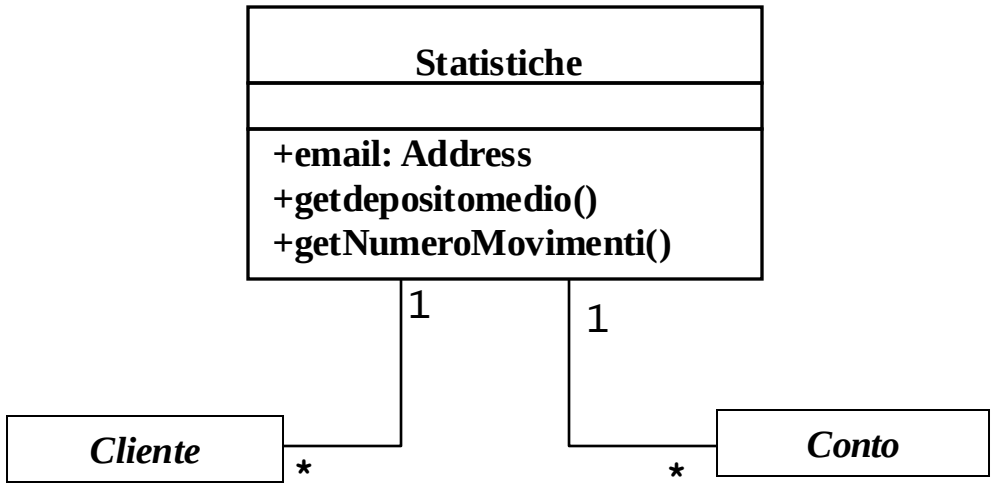
- Per realizzare tale associazione prima trasformiamo la classe associativa in un oggetto che ha più associazioni binarie, poi convertiamo le associazioni binarie in un insieme di attributi referenziati, come visto in precedenza.

Classi associative: esempio

Object design model
before transformation



Object design model
after transformation



Attività che coinvolgono trasformazioni

- **Ottimizzare il modello di Object Design**
- **Mappare le associazioni in collezioni**
- **Mappare contratti in eccezioni**
  - Le eccezioni in Java: il meccanismo try-throw-catch
  - Implementare un contratto
- Mappare l'Object Model in uno schema di memorizzazione persistente

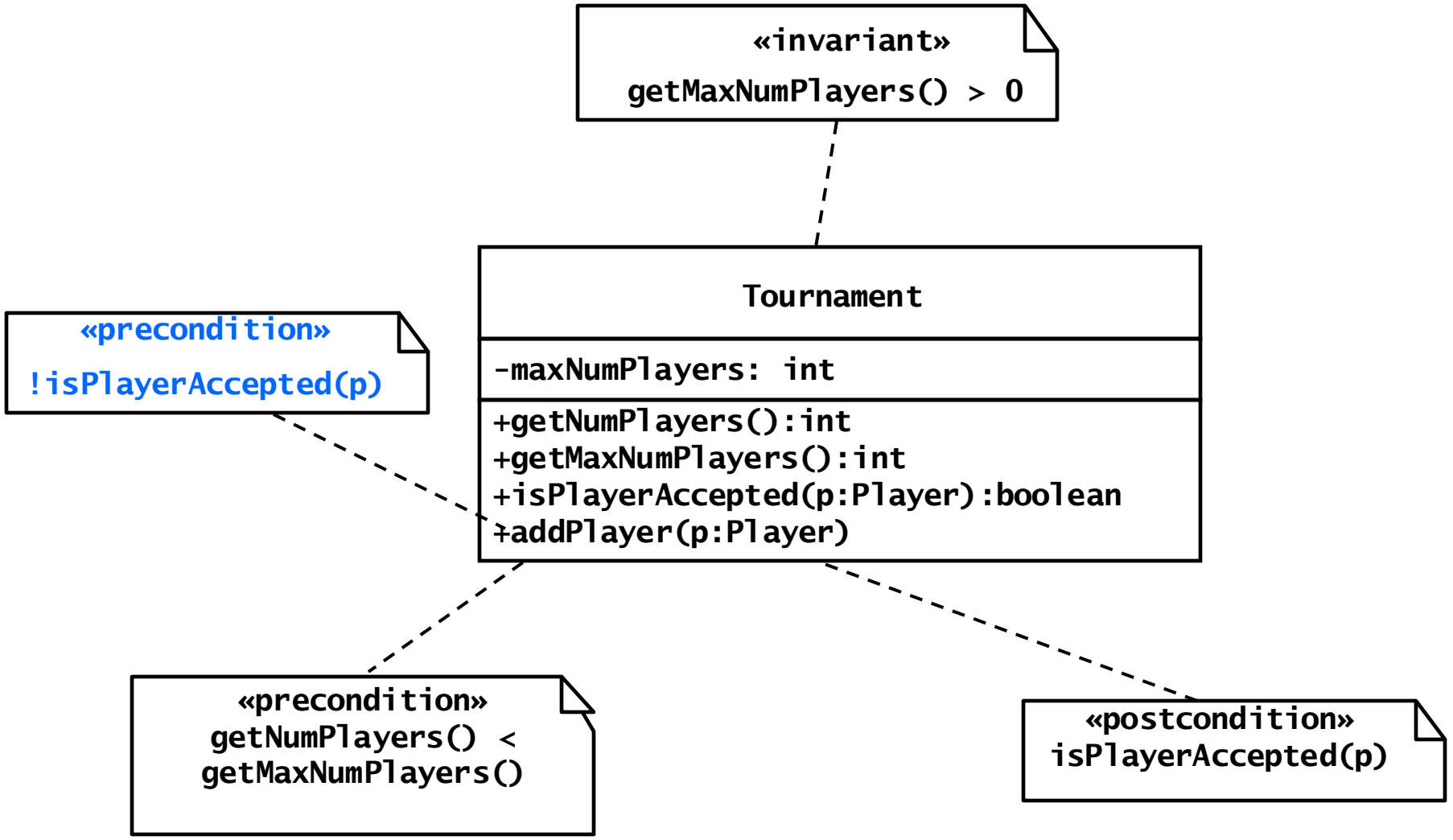
Mapping di contratti in eccezioni

- Un **contratto** è un vincolo su di una classe che deve essere soddisfatto prima di utilizzare la classe.
- Molti linguaggi di programmazione object-oriented non forniscono supporto per i contratti.
- Si utilizza il meccanismo delle eccezioni per segnalare e gestire le violazioni dei contratti.

Le eccezioni in Java

- In **Java** si solleva una eccezione con la parola chiave **throw** seguita da un oggetto eccezione.
- L'oggetto eccezione fornisce un posto dove memorizzare informazioni sull'eccezione, usualmente un messaggio di errore.
- L'effetto di lanciare un'eccezione è duplice:
  - interrompere il flusso di controllo;
  - svuotare lo stack delle chiamate finchè non si trova un blocco **catch** che gestisce l'eccezione.

Contratti per l'operazione Tournament.addPlayer()



Implementare un contratto

Per ogni operazione nel contratto:

- **controllare le precondizioni** prima dell'inizio del metodo con un test che lancia una eccezione se una precondizione non è verificata.
- **controllare la postcondizione** alla fine di ciascun metodo e lanciare una eccezione se il contratto è violato.
  - Se più di una postcondizione non è soddisfatta, lanciare una eccezione solo per la prima violazione.
- **controllare le invarianti** allo stesso modo delle postcondizioni.
- **gestire l'ereditarietà** incapsulando il codice di controllo per precondizioni e postcondizioni in metodi separati che possono essere richiamati dalle sottoclassi.

Euristiche per mappare contratti in eccezioni

Specificare tutti i contratti non è realistico, non si ha abbastanza tempo, possono essere introdotti errori, il codice può divenire complesso, le prestazione possono peggiorare.

- Si può omettere il codice di controllo per postcondizioni e invarianti:
  - è ridondante inserirlo insieme al codice che realizza la funzionalità della classe, inoltre non individua molti bug a meno che non venga scritto da un altro sviluppatore.
- Si può omettere il codice di controllo per metodi privati e protetti se è ben definita l'interfaccia del sottosistema.
- Concentrarsi sulle componenti che hanno una lunga durata:
  - oggetti Entity, non oggetti boundary associati all'interfaccia utente.
- Riusare codice per il controllo dei vincoli:
  - molte operazioni hanno precondizioni simili;
  - incapsulare il codice per il controllo degli stessi vincoli in metodi così possono condividere le stesse classi di eccezioni.

Attività di mapping

- Ottimizzare il modello di Object Design
- Mappare le associazioni in collezioni
- Mappare contratti in eccezioni
- **Mappare l'Object Model in uno schema di memorizzazione persistente**
  - Database Relazionali (in breve)
  - Mappare classi ed attributi
  - Mappare le associazioni
  - Mappare le relazioni di ereditarietà
    - Mapping orizzontale e mapping verticale

Mappare l'object model in schemi di memorizzazione persistenti

- I linguaggi di programmazione object-oriented di solito non forniscono un modo efficiente per memorizzare gli oggetti persistenti.
- È necessario mappare gli oggetti persistenti in strutture dati che possono essere memorizzate nei sistemi di gestione dei dati selezionati durante il system design (database o file).
- Se usiamo database object-oriented non devono essere effettuate trasformazioni.
- Se usiamo database relazionali o flat file è necessario:
  - mappare il modello degli oggetti in uno schema di memorizzazione.
  - fornire una infrastruttura per convertire gli oggetti in schemi di memorizzazione persistente e viceversa.

Database Relazionali (in breve)

- Uno **schema** è una descrizione dei dati (meta-modello)
- I database relazionali memorizzano sia lo schema sia i dati
- I dati persistenti sono memorizzati sotto forma di **tabelle**.
- Una tabella è strutturata in **colonne.**
- Ogni colonna rappresenta un **attributo**.
- La **chiave primaria** di una tabella è un insieme di attributi i cui valori identificano univocamente una riga della tabella.
- Insiemi di attributi che possono essere usati come chiave primaria sono detti **chiavi candidate**.
- Una **chiave esterna** è un attributo (o un insieme di attributi) che referenzia la chiave primaria di un'altra tabella.

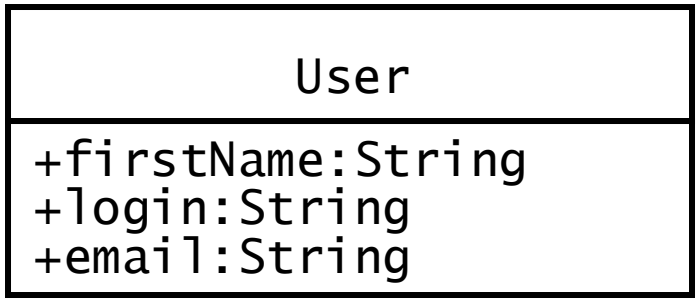
Mappare classi ed attributi

- Focalizziamo innanzitutto l'attenzione sulle **classi** ed i loro **attributi**:
  - mappiamo ogni classe in una tabella del database con lo stesso nome;
  - per ogni attributo aggiungiamo una colonna nella tabella con il nome dell'attributo della classe.
- Ogni tupla della tabella corrisponde ad un'istanza della classe.
- Mantenendo gli stessi nomi nel modello ad oggetti e nelle tabelle garantiamo la tracciabilità fra le due rappresentazioni.
- Quando mappiamo gli attributi dobbiamo selezionare i tipi di dati per le colonne della tabella:
  - ci sono dei tipi di dati per cui il mapping è intuitivo, altri per cui complesso (es: String può corrispondere al tipo text in SQL che, però, richiede una taglia fissata, text[25]).

Selezionare la chiave primaria

- Per **selezionare la chiave primaria** di una tabella possiamo scegliere tra due opzioni:
  - identificare un insieme di attributi della classe che identifichi univocamente l'oggetto;
  - aggiungere un identificatore unico (id) che identifichi univocamente l'oggetto.

Mapping the User class to a database table



User table

id:long	firstName:text[25]	login:text[8]	email:text[32]

Example for Primary and Foreign Keys

User table

Primary key

firstName	login	email
“alice”	“am384”	“am384@mail.org”
“john”	“js289”	“john@mail.de”
“bob”	“bd”	“bobd@mail.ch”

Candidate key Candidate key

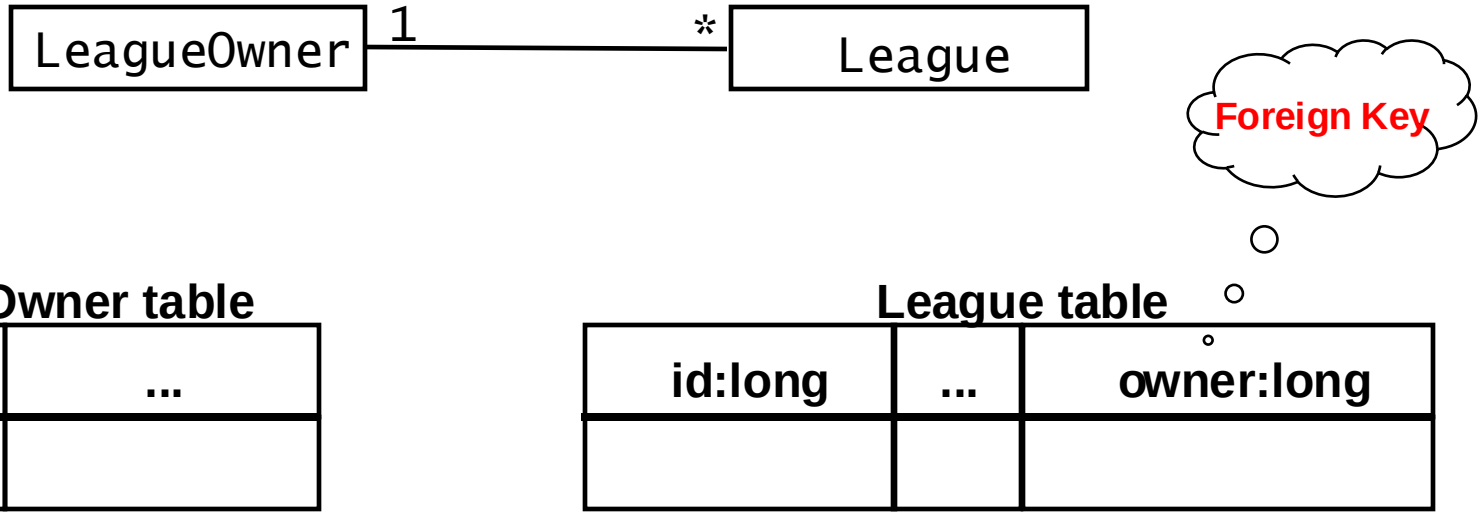
League table

name	login
“tictactoeNovice”	“am384”
“tictactoeExpert”	“am384”
“chessNovice”	“js289”

Foreign key referencing **User table**

Mappare le associazioni (1)

- Le associazioni **One-to-one e one-to-many** sono implementate usando una chiave esterna (**buried association**).
- Le **associazioni one-to-one** sono mappate inserendo una chiave esterna in una delle due tabelle rappresentanti le classi.
- Le **associazioni one-to-many** sono mappate usando la chiave esterna sul lato many.

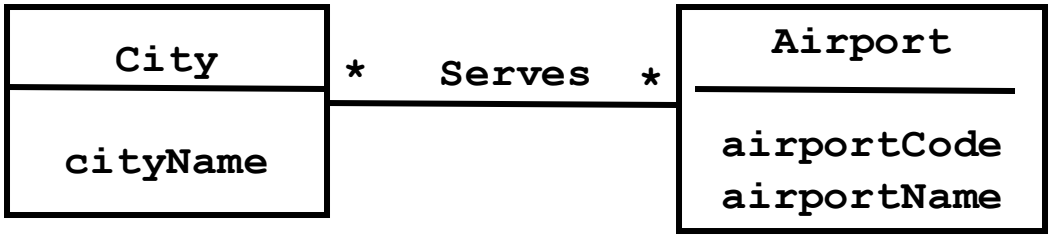


Mappare le associazioni (2)

- Le **associazioni many-to-many** sono implementate usando una tabella separata costituita di due colonne che contengono la chiave esterna di ciascuna classe coinvolta nell'associazione.

- Tale tabella è detta **tabella associativa**:
  - ogni riga di tale tabella corrisponde ad un collegamento tra due istanze dell'associazione molti-a-molti.

Esempio di associazioni molti-a-molti



Primary Key

Separate Table

City Table

cityName
Houston
Albany
Munich
Hamburg

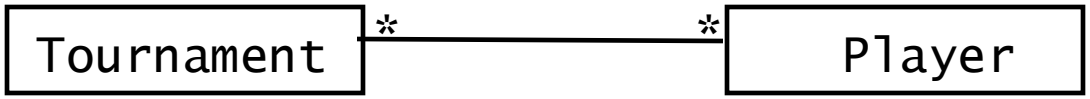
Airport Table

airportCode	airportName
IAH	Intercontinental
HOU	Hobby
ALB	Albany County
MUC	Munich Airport
HAM	Hamburg Airport

Serves Table

cityName	airportCode
Houston	IAH
Houston	HOU
Albany	ALB
Munich	MUC
Hamburg	HAM

Mapping the Tournament/Player association as a separate table



Tournament table

id	name	...
23	novice	
24	expert	

TournamentPlayerAssociation table

tournament	player
23	56
23	79

Player table

id	name	...
56	alice	
79	john	

Cambiamenti nelle associazioni

- Anche le **associazioni one-to-one e one-to-many** possono essere realizzate con una tabella di associazione invece che con chiavi esterne.
- L'utilizzo di tabelle separate:
  - rende lo schema facilmente modificabile (ad es. se cambia la molteplicità dell'associazione non dobbiamo cambiare lo schema);
  - accresce il numero delle tabelle ed il tempo per attraversare l'associazione.
- Per scegliere dobbiamo rispondere alle seguenti domande:
  - Il tempo di risposta è un fattore critico per la nostra applicazione?
  - Quanto è probabile che l'associazione cambi?

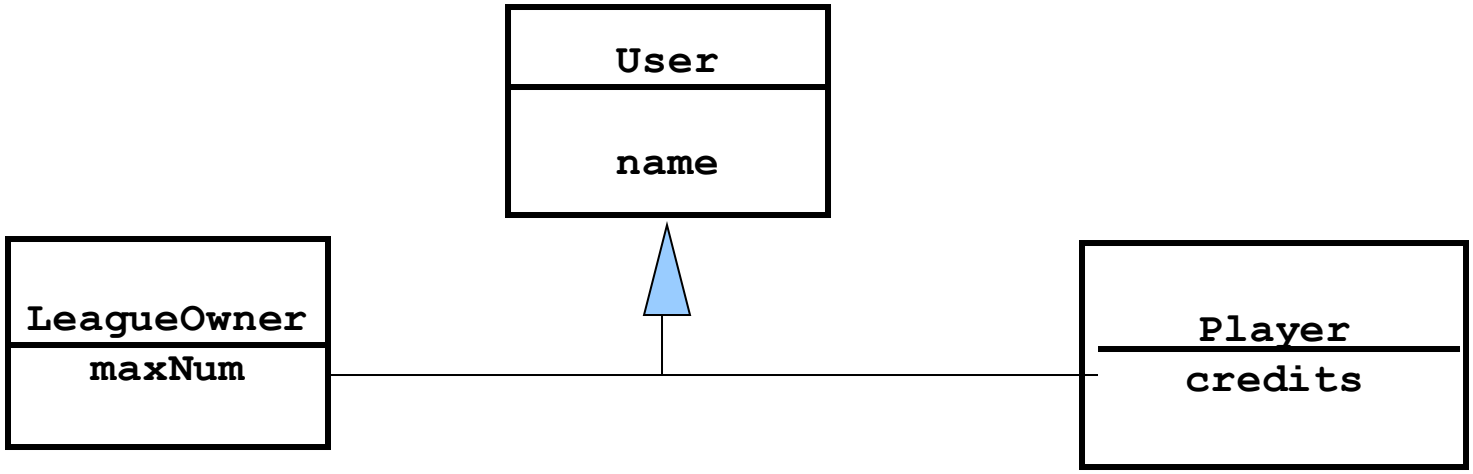
Mappare le relazioni di ereditarietà

- **I database relazionali non supportano l'ereditarietà.**
- Esistono **due opzioni per mappare l'ereditarietà** in uno schema di un database:
  - **Mapping verticale:** simile al mapping di associazioni uno-a-uno, ogni classe è rappresentata da una tabella e utilizza una chiave esterna per collegare la tabella corrispondente ad una sottoclasse con quella corrispondente alla superclasse.
  - **Mapping orizzontale:** gli attributi della superclasse sono ricopiati in tutte le sottoclassi e la superclasse viene eliminata.

Mapping Verticale

- Data una relazione di ereditarietà, mappiamo la super-classe e la sottoclasse in tabelle individuali.
- La tabella relativa alla superclasse include:
  - una colonna per ogni attributo definito nella superclasse;
  - una colonna addizionale denotante la sottoclasse che corrisponde al data record.
- La tabella relativa alla sottoclasse include:
  - una colonna per ogni attributo della sottoclasse.
- Entrambe le tabelle hanno la stessa chiave primaria.

Mapping Verticale



User table

id	name	.....	ruolo
56	Rossi		League Owner
79	bianchi		player

LeagueOwner table

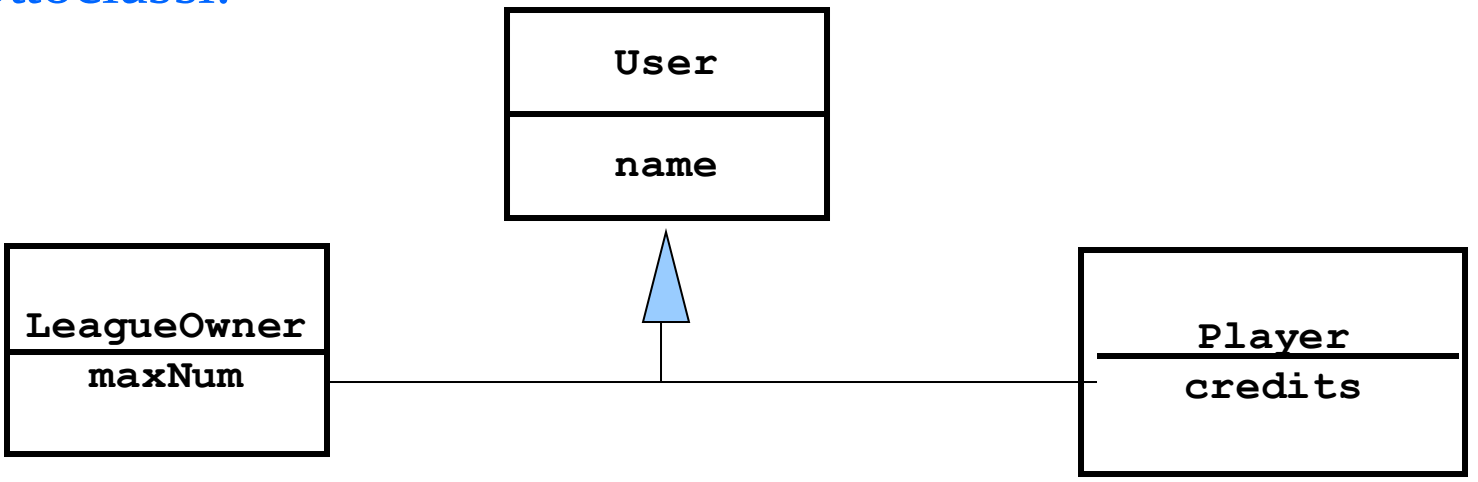
id	maxNum	.....
56	12	

Player table

id	Credits	.....
79	126	

Mapping Orizzontale

- Consiste nel mettere tutti gli attributi della superclasse nelle sottoclassi.



LeagueOwner table

id	name	maxNum	...
56	Rossi	12	

Player table

id	name	credits	...
79	Bianchi	126	

Mapping Orizzontale vs Verticale

- **Mapping Verticale**
  - Utilizzando una tabella separata:
    - ☺ possiamo facilmente aggiungere un attributo alla superclasse aggiungendo una colonna alla tabella superclasse;
    - ☺ aggiungere una sottoclasse significa aggiungere una tabella per la sottoclasse con un attributo per ogni colonna della sottoclasse;
    - ☹ ricercare tutti gli attributi di un oggetto richiede una operazione di Join.
- **Mapping Orizzontale**
  - Duplicando le colonne:
    - ☺ gli oggetti non sono frammentati fra più tabelle e le query sono più veloci;
    - ☹ modificare lo schema è più complesso.

Mapping Orizzontale vs Verticale: trade-off

- Il trade-off riguarda la modificabilità ed il tempo di risposta:
  - Quanto è probabile che la superclasse sia modificata?
  - Quali sono le prestazioni richieste per le query?

- **Mapping Verticale:** *modificabile ma meno efficiente*
- **Mapping Orizzontale:** *poco modificabile ma più efficiente*

Per scegliere quali tra i due utilizzare va esaminata la probabilità di avere cambiamenti VS i requisiti di prestazione

Euristiche per effettuare trasformazoni

- Utilizzare sempre il medesimo strumento per effettuare le trasformazioni:
  - ad esempio, se è stato utilizzato un particolare strumento CASE per trasformare le associazioni in codice, bisogna utilizzare lo stesso tool per modificare la molteplicità delle associazioni.
- Mantenere traccia dei contratti nel codice sorgente e non solo nel modello di object design:
  - se si utilizza la specifica dei contratti come commento del codice sorgente è più probabile che tale specifica venga modificata se viene modificato il codice sorgente.
- Utilizzare gli stessi nomi per gli stessi oggetti:
  - se un nome viene modificato nel modello bisogna modificarlo anche nel codice sorgente e/o nello schema della base di dati;
  - ciò permette di mantenere la tracciabiità tra i diversi modelli.
- Utilizzare delle linee guida per le trasformazioni:
  - se si riportano in un manuale le convenzioni che si vogliono utilizzare per le trasformazioni tutti gli sviluppatori potranno apliccarle allo stesso modo.

Conclusioni: Gestire l'implementazione

- Le **trasformazioni** ci consentono di
  - migliorare aspetti specifici del modello e di convertirlo in codice sorgente;
  - ridurre lo sforzo complessivo ed il numero totale di errori nel codice sorgente.
- È necessario documentare le trasformazioni così che possano essere riapplicate in caso di cambiamenti nel modello di object design o nel codice sorgente.

Conclusioni: Responsabilità

- Diversi ruoli cooperano per selezionare, documentare e applicare le trasformazioni:
  - Il **"core architect"** seleziona le trasformazioni che devono essere applicate sistematicamente.
    - Es: database deve essere modificabile (le associazioni vanno mappate in tabelle separate)
  - L'**"architecture liaison"** è responsabile della documentazione dei contratti associati alle interfacce dei sottosistemi.
    - Quando un contratto cambia, egli ha la responsabilità di darne notifica a tutti gli utenti delle classi.
  - Lo **"sviluppatore"** ha la responsabilità di seguire l'insieme delle convenzioni stabilite dal "core architect", applica le trasformazioni e converte il modello di object design in codice sorgente.
    - È responsabile dell'aggiornamento dei commenti del codice sorgente.

Riassumendo…

- **Modifiche non disciplinate => degradazione del modello del sistema**
- Abbiamo introdotto quattro **tipi di trasformazioni:**
  - la trasformazione di modello migliora l'aderenza del modello di object design con gli obiettivi di design;
  - il forward engineering migliora la consistenza del codice rispetto al modello di object design
  - il refactoring migliora la leggibilità e la modificabilità del codice sorgente.
  - il reverse engineering tenta di ricavare il modello design a partire dal codice sorgente.
- Abbiamo descritto alcune **tecniche di trasformazione di modello e di forward engineering**:
  - Ottimizzazione del modello delle classi.
  - Mapping di associazioni in collezioni.
  - Mapping di contratti in eccezioni.
  - Mapping del modello delle classi in schemi di memorizzazione.