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 al richiesta dei requisiti e la progettazione del sistema.
Tuttavia quando gli sviluppatori iniziano a integrare i vari sottosistemi, possono incontrare delle varie difficolta come:

- Parametri non documentati potrebbero essere stati aggiunti all'API per far fronte a una modifica dei requisiti.
- Eventuali attributi sono stati aggiunti al modello a oggetti ma non sono gestiti dal sistema di gestione persistente, forse a causa di una comunicazione errata.

Si potrebbero risolvere tali problemi ma ciò comporta ulteriori modifiche improvvise del codice e soluzioni alternative che alla fine cedono al degrado del sistema. Di conseguenza il codice finale avrebbe poca somiglianza con il design originale ed sarà sicuramente difficile da capire.
Quindi per evitare tale problematiche, si incorre all'uso di un approccio disciplinato per evitare la detrazione del sistema:

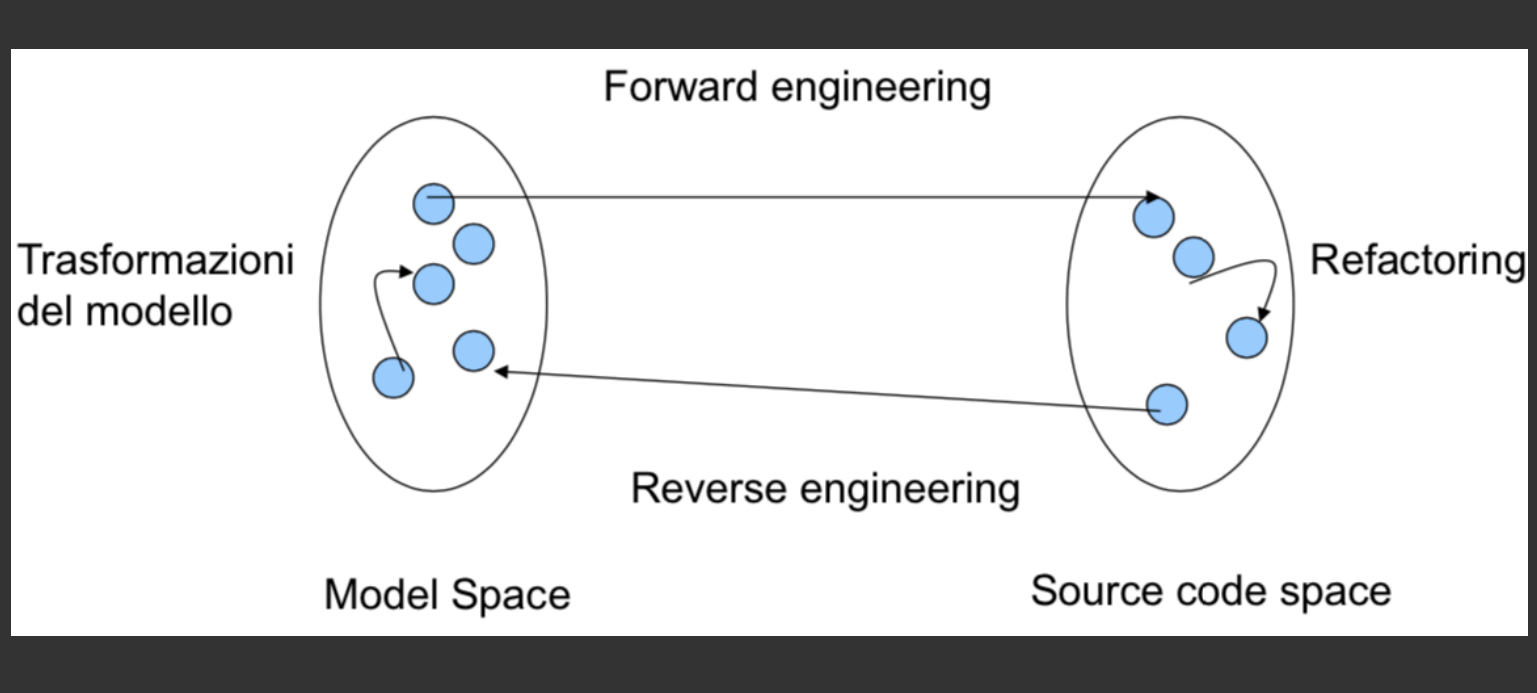
- Ottimizzazione del modello di classe.
- Mappare le associazioni in collezioni.
- Mappare i contratti delle operazioni in eccezioni.
- Mappare il modello delle classi in uno schema di memorizzazione persistente.

Vi è un insieme di tecniche che consente di ridurre gli errori che si introducono durante le varie "trasformazioni":

- Trasformazioni del modello a oggetti.
- Refactoring.
- Forward Enginering.
- Reverse Enginering.

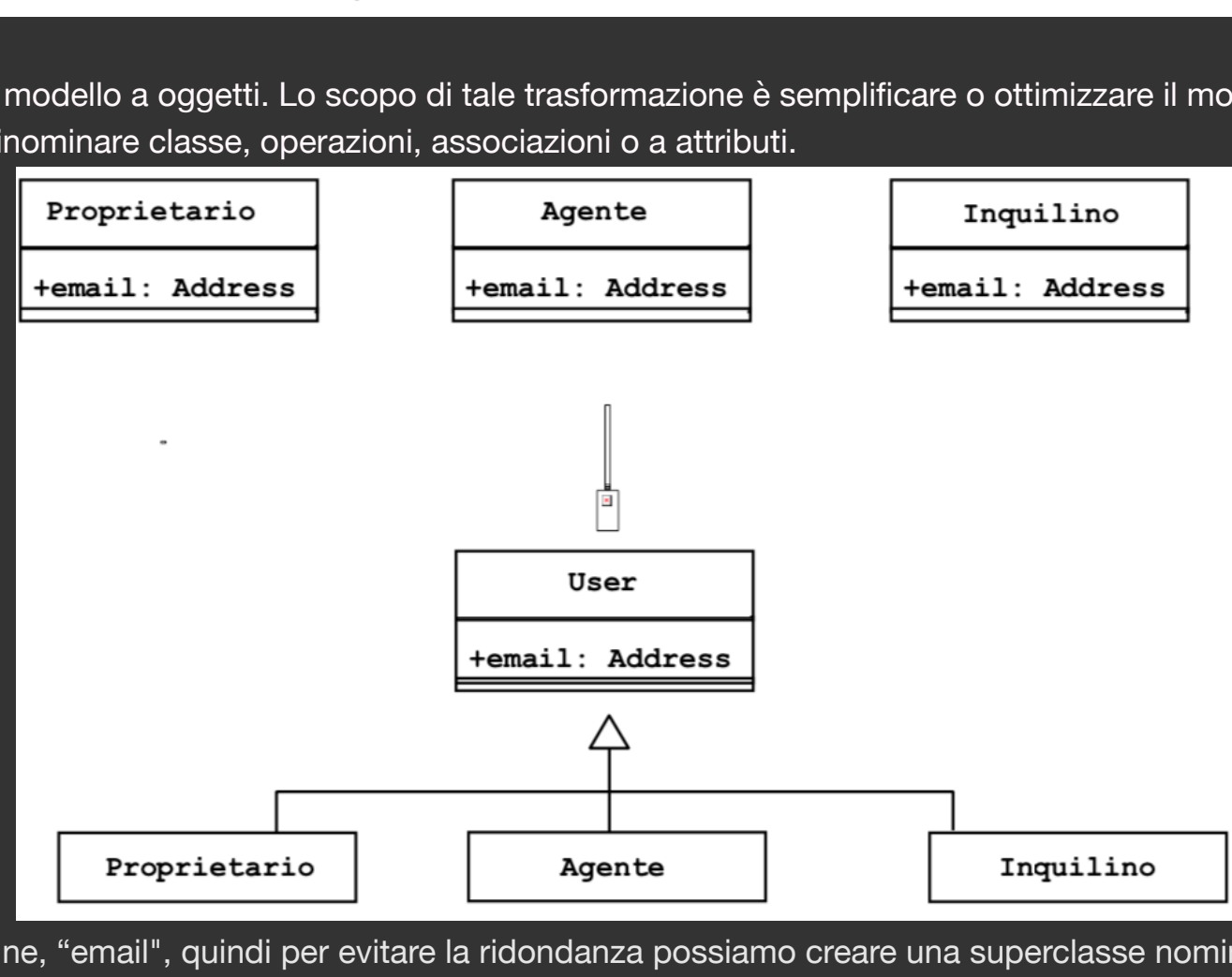
Trasformazione

Una "trasformazione" mira a migliorare un aspetto del modello preservando tutte le sue altre proprietà. Di solito, una trasformazione è localizzata, influisce su un piccolo numero di classi, attributi e operazioni ed è eseguita in una serie di piccoli passati.
Trasformazioni del modello a oggetti: conversione di un attributo semplice (ad esempio un indirizzo rappresentato come stringa) in una classe (ad esempio, una classe con indirizzo, codice postale, città, stato e attributi del paese).
Refactoring: trasformazioni che operano sul codice sorgente. Simili alle trasformazioni dei modelli di oggetti in quanto migliorano un singolo aspetto del sistema senza modificarne la funzionalità. Si differenziano per il fatto che manipolano il codice sorgente.
Forward Engineeeing: produce un modello di codice sorgente che corrisponde a un modello a oggetti.
Reverse Engineering: produce un modello che corrisponde al codice sorgente. Questa trasformazione viene utilizzata quando la progettazione del sistema è andata persa e deve essere recuperata dal codice sorgente.



Trasformazioni del modello a oggetti

Viene applicata a un modello di oggetti e risulta un altro modello a oggetti. Lo scopo di tale trasformazione è semplificare o ottimizzare il modello originale.
Una trasformazione potrebbe aggiungere, rimuovere o rinominare classe, operazioni, associazioni o a attributi.



Notiamo che queste 3 classi hanno un attributo in comune, "email", quindi per evitare la ridondanza possiamo creare una superclasse nominata con user aggiungendo cosi l'attributo in comune.
Applicare un modello di trasformazione è attività meccanica, identificare quale trasformazione applicare a quale gruppo di classi richiede esperienza e giudizio.

Refactoring

E' una trasformazione di un blocco di codice che migliora la leggibilità o la modificabilità senza cambiare il comportamento del sistema. Mirano ad un campo o ad un metodo specifico di una classe e per assicurarsi che tale modifiche non cambiano il comportamento, vengono fatti dei test ad ogni modifica incrementale. Tali test consente quindi di modificare il codice con la sicurezza di non combinare dei disastri.
Tale esempio mostrato in seguito corrisponde a un sequenza di tre refactoring:

1. Spostare l'email dalle sottoclassi alla superclasse.

```
Prima del refactoring
public class Player{
    private String email;
}

public class LeagueOwen{
    private String email;
}

public class Advertiser{
    private String email_address;
}
```

```
Dopo il refactoring
public class User{
    protected String email;
}

public class Player extends User{
    //
}


public class LeagueOwen extends User{
    //
}


public class Advertiser extends User{
    //
}
```

2. Spostare il codice di inizializzazione dalle sottoclassi alla superclasse.

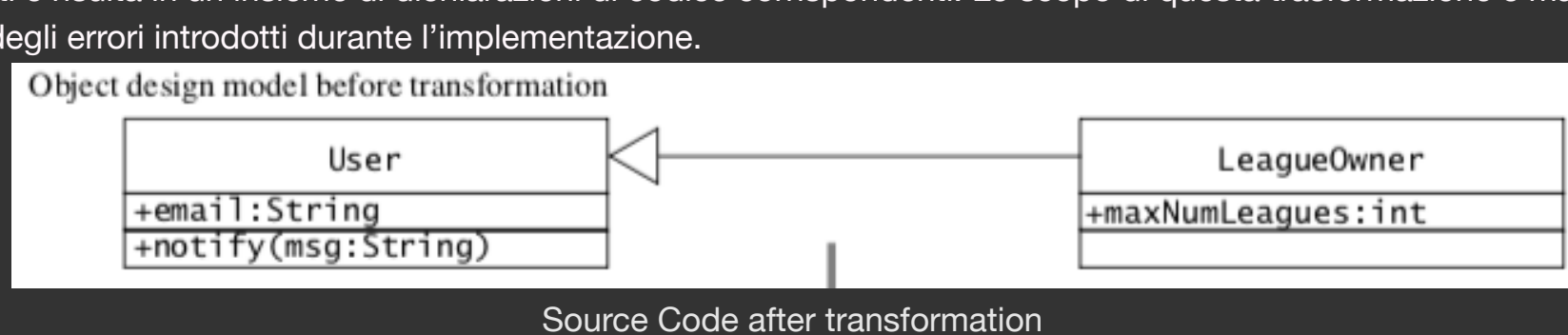
```
Prima refactoring
public class User{
   private String email;
}


public class Player extends User{
    public Player(String email){
        this.email=email;
    }
}

public class LeagueOwen extends User{
    public LeagueOwner(String email){
        this.email=email;
    }
}

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

```
Dopo refactoring
public class User{
    private String email;

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


public class Player extends User{
    public Player(String email){
        super(email);
    }
}

public class LeagueOwen extends User{
    public LeagueOwner(String email){
        super(email);
    }
}

public class Advertiser extends User{
    public Advertiser(String email){
        super(email);
    }
}
```

3. Spostare i metodi che manipolano tale campo dalle sottoclassi alla superassi.

Forward Engineering

E'applicato a un insieme di modelli di elementi e risulta in un insieme di dichiarazioni di codice corrispondenti. Lo scopo di questa trasformazione è mantenere una forte corrispondenza tra object design model e il codice, e ridurre il numero degli errori introdotti durante l'implementazione.



Source Code after transformation

```
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 LeagueOwner extends User{
    private int maxNumLeagues;
    public int getMaxNumLeagues(){
        return maxNumLeagues;
    }

    public void setMaxNumLeagues(int value){
        maxNumLeagues=value;
    }
}
```

- Ogni classe del diagramma UML è mappata in una classe JAVA.
- La relazione di generalizzazione UML è mappata in una istruzione extends.
- Ogni attributo del modello UML è mappato in un campo privato della classe Java e due metodi pubblici per settore e visualizzare i valori del campo.

Reverse Enginneering

E' applicato a un siete di elementi del codice sorgente e risulta in un serie di elementi del modello. Lo scopo di tale trasformazione è ricreare il modello partendo da un sistema esistente, o perché il modello si è perso o mai creato o perché è diventato non sincronizzato con il codice. Logicamente si tratta di una trasformazione inversa del Forward Engineering.

- Crea una classe UML per ciascun classe.
- Aggiunge un attributo per ciascun campo.
- Aggiunge un operazione per ciascun metodo.

Tale trasformazione non crea il modello originale in quanto il forward enginnergin può far perdere informazioni come le associazioni).

Per evitare che le trasformazioni possano introdurre errori difficili da rilevare e da correggere è necessario che ogni trasformazioni:

- migliori il sistema rispetto ad un singolo obiettivo di design.
- cambi pochi metodi o pochi classi alla volta.
- venga applicata in modo isolato rispetto agli atri cambiamenti.
- venga seguita da un passo di validazione: dopo aver effettuato una trasformazione e prima di svolgere un'altra, bisogna validare i cambiamenti.

Attività che coinvolgono trasformazioni

Ottimizzazione del modello di Object Design: attività intrapresa al fine di soddisfare i requisiti di prestazione del modello del sistema.
Realizzare associazioni: attività necessaria per mappare associazioni in costrutti di codice sorgente.
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.

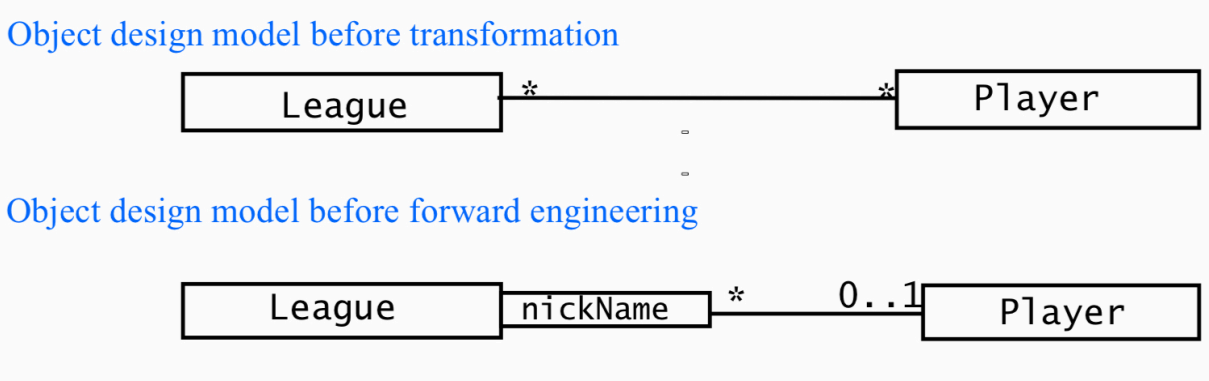
Ottimizzazione del modello di Object Design

La traduzione diretta del modello di analisi in codice sorgente è spesso inefficiente. Il modello di analisi si concentra sulle funzionalità del sistema e non tiene conto degli obiettivi di design identificati durante il system design. Quindi il compito dell'object design è trasformare il modello ad oggetti per soddisfare gli obiettivi di design identificati durante il system design(tempo di risposta, tempo di esecuzione, risorse di memoria).
Le attività coinvolte in tale fase sono:

- Ottimizzare i cammini di accesso.
- Collassare gli oggetti in attributi.
- Ritardare le elaborazioni costose.
- Mantenere in una struttura dati temporanea il risultato di elaborazioni costose.

Cammino di accesso ottimale

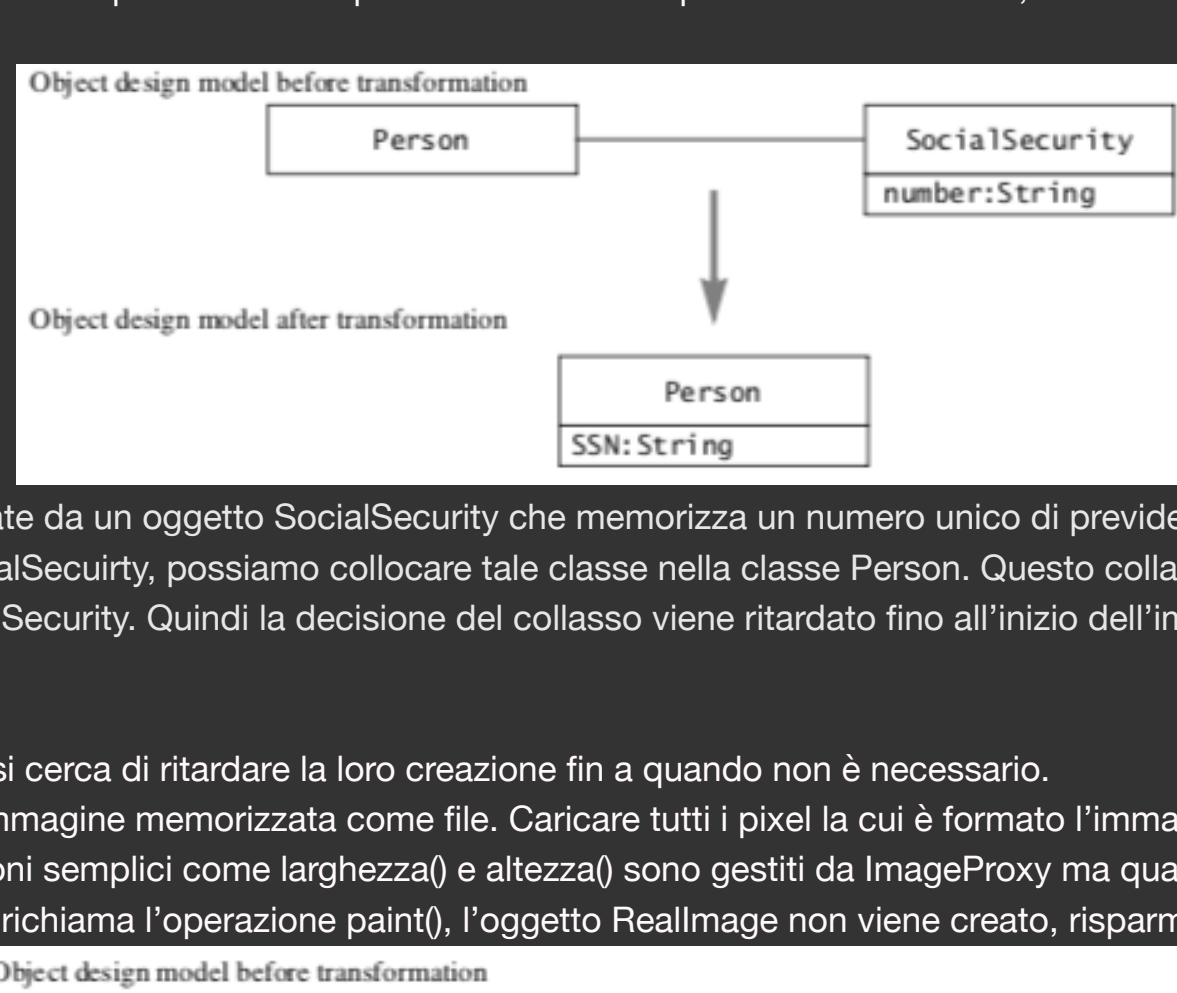
Al fine di ottimizzare i cammini di accesso, dobbiamo eliminare il ritardo ottenuto a causa:

- dell'attraversamento ripetuto di associazioni multiple;
  - Le operazioni frequenti non dovrebbero richiedere molti attraversamenti, ma dovrebbero avere una connessione diretta tra l'oggetto che interroga e l'oggetto interrogato. Se tale associazioni non è presente, è necessario aggiungere un'associazione tra questi due oggetti. Le stime per la frequenza dei percorsi di accesso possono essere derivate dal sistema legacy nel caso di progetti di interface engineering e re-engineering. Ma nel caso di greenfield engineering, può essere determinare durante il testing.
- dell'attraversamento di associazioni di tipo "molti";
  - Si dovrebbe provare a ridurre i tempi di ricerca riducendo i "molti" a "uno" utilizzando un'associazione qualificata.Se ciò non è possibile, bisogna prendere in considerazione l'ordine o l'indicizzazione degli oggetti sul lato molti per ridurre i tempi di accesso.
  - 

- della presenza di attributi mal collocati.
  - Durante l'analisi vengono identificate molte classi che risultano prive di comportamento interessante. Se la maggior parte degli attributi sono coinvolti sono nelle operazioni set() e get(), è possibile spostare spostare tali attributi nella classe chiamante. Di conseguenza, alcune classi potrebbero non servire più ed essere rimosse dal modello.

Collassare gli oggetti in attributi

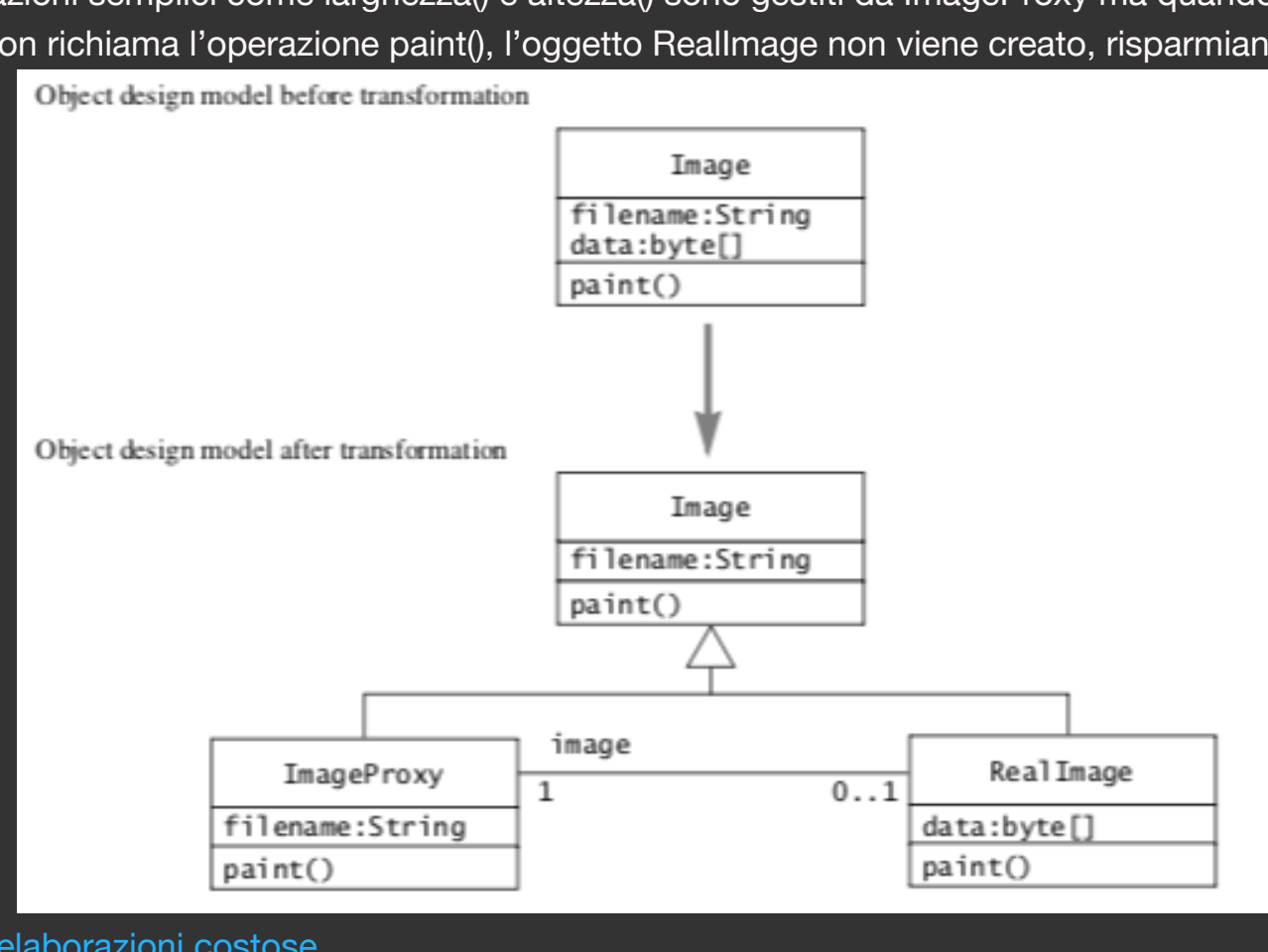
Dopo che il modello è stato ottimizzato diverse volte, alcune classi possono avere pochi attributi o comportamenti. Tali classi, se sono associate ad una sola altra classe, possono essere collassate in attributi.



Qui sopra si tratta un modello che include persone identificate da un oggetto SocialSecurity che memorizza un numero unico di previdenza sociale che identifica la persona. Se non vi cono casi d'uso che richiede alcun comportamento per l'oggetto SocialSecuirty, possiamo collocare tale classe nella classe Person. Questo collassare non è sempre cosi ovvia poichè vi possono essere dei casi in cui un comportamento particolare della classe SocialSecurity. Quindi la decisione del collasso viene ritardato fino all'inizio dell'implementazione quando sono chiari le varie responsabilità assegnati alle varie classi.

Ritardare le elaborazioni costose

Spesso, gli oggetti specifici sono costosi da creare. Quindi si cerca di ritardare la loro creazione fin a quando non è necessario.
Ad esempio, consideriamo un oggetto che rappresenti un'immagine memorizzata come file. Caricare tutti i pixel la cui è formato l'immagine è costoso quindi si crea una classe proxy che prende il posto dell'immagine e fornisce la stesa interfaccia. Operazioni semplici come larghezza() e altezza() sono gestiti da ImageProxy ma quando bisogna visualizzare l'immagine, l'immageProxy provvede ad creare l'oggetto RealImage. Ma se il client non richiama l'operazione paint(), l'oggetto RealImage non viene creato, risparmiando cosi tempo di calcolo sostanziale.



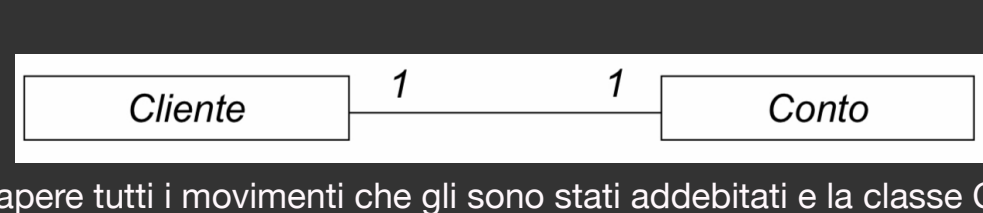
Mantenere in una struttura dati temporanea il risultato di elaborazioni costose

Alcuni metodi vengono chiamate molti volte, ma i loro risultati si basano su valori che non cambiano o cambiano solo di rado. La riduzione del numero di calcoli richiesti da questi metodi migliore i tempi di risposta complessi. In tal caso, il risultato deve essere memorizzato come attributo private.
Consideriamo l'operazione LeagueBoundary.getStatistics() che visualizza le statistiche relative a tutti i giocatori e tornei di una lega. Queste statistiche cambiando solo quando una partita è stata completata quindi non è necessario ricalcolare le statistiche ogni volta che un utente desidera vederle. Quindi tale risultato può essere memorizzato come attributo privato che rimane bloccata fino al successivo completamento di una partita.

Mappare le associazioni in collezioni

Le associazioni sono concetti UML che denotano collezioni di link bidirezionali tra due o più oggetti. Tali associazioni non esistono nei linguaggi orientati agli oggetti ma quest'ultimi si basano su "riferimento" e "collezione".
Durante l'object design trasferiamo le associazioni in termini di riferimenti considerando la molteplicità e la direzione delle associazioni.

Associazioni uno-a-uno unidirezionali e bidirezionali



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 operazione della classe Cliente, l'associazione è unidirezionale.
Si traduce inserendo un campo conto nella classe Cliente che referenzia l'oggetto Conto.

```
public class Cliente{
    private Conto conto;

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

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

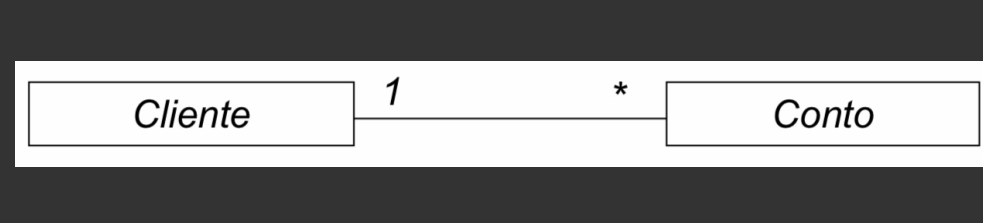
Le associazioni bidirezionali sono complesse ed introducono una dipendenza reciproca fra le classi.
Supponiamo di modificare la classe Conto in modo tale che il nome del Conto da visualizzare dipenda dal nome del Cliente.

```
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;
    }
}
```

P.S A volte le associazioni unidirezionali vengono trasformate in associazioni bidirezionali. E' consigliabile rendere sistematicamente gli attributi privati e fornire i metodi getAttribute e setAttribute. In questo modo minimizza le modifiche quando si passa da una associazione unidirezionale ad una bidirezionale e viceversa.

Associazioni uno a molti

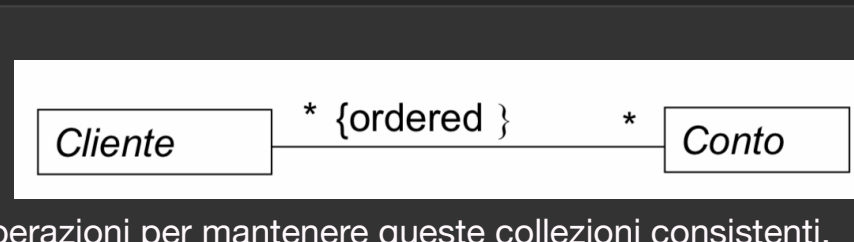


Non possono essere realizzate usando un singolo riferimento.
Esempio: supponiamo che ad un cliente possano corrispondere più conti.
Poichè i conti non hanno un ordine specifico possiamo usare un insieme di riferimenti, conti, per modellare la parte molti dell'associazione.

```
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 molti a molti



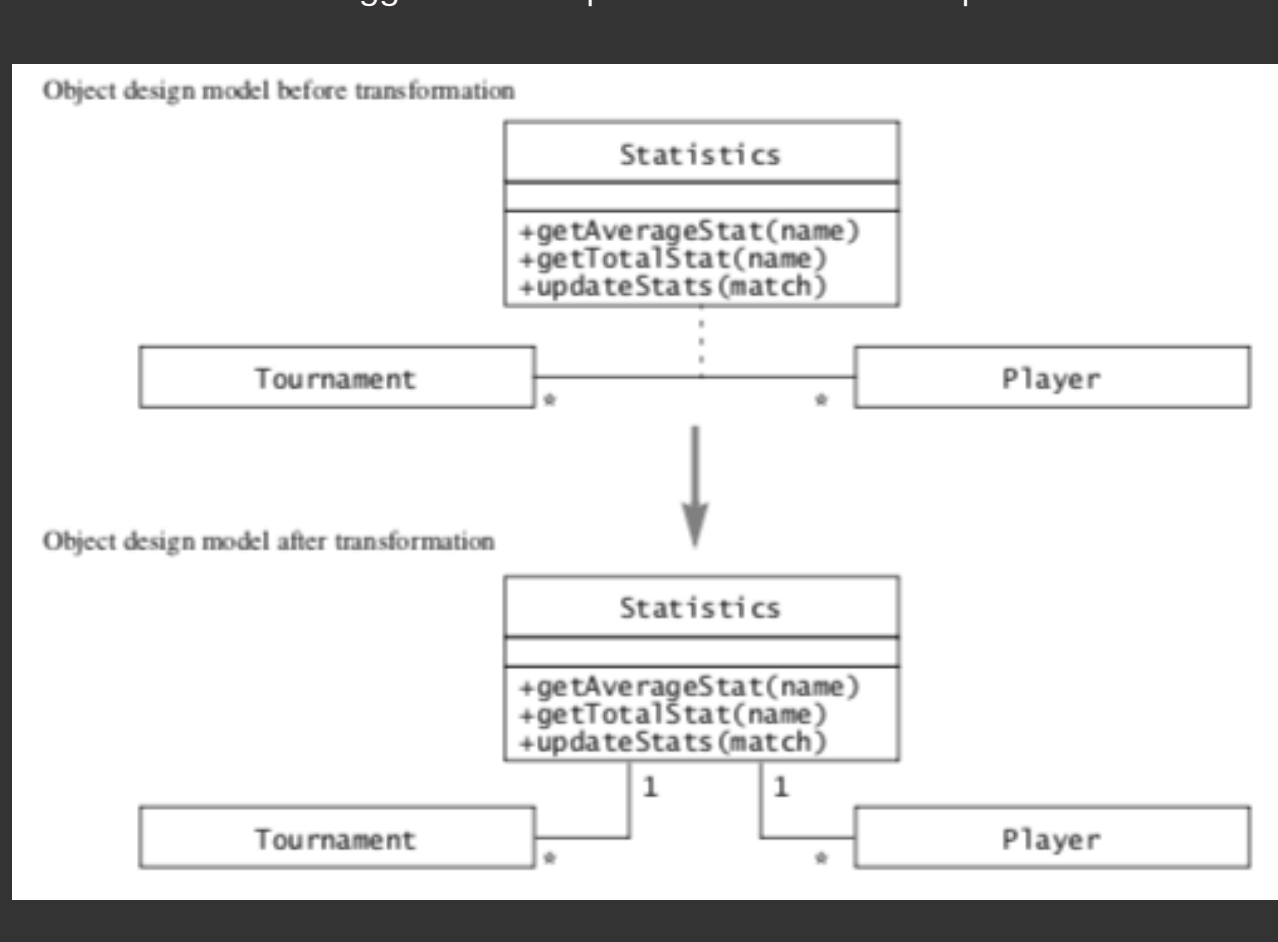
Entrambe le classi hanno campi che sono collezioni di riferimenti ed operazioni per mantenere queste collezioni consistenti.
Supponiamo che un conto possa essere intestato a più clienti.

```
public classe 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);
        }
    }
}
```

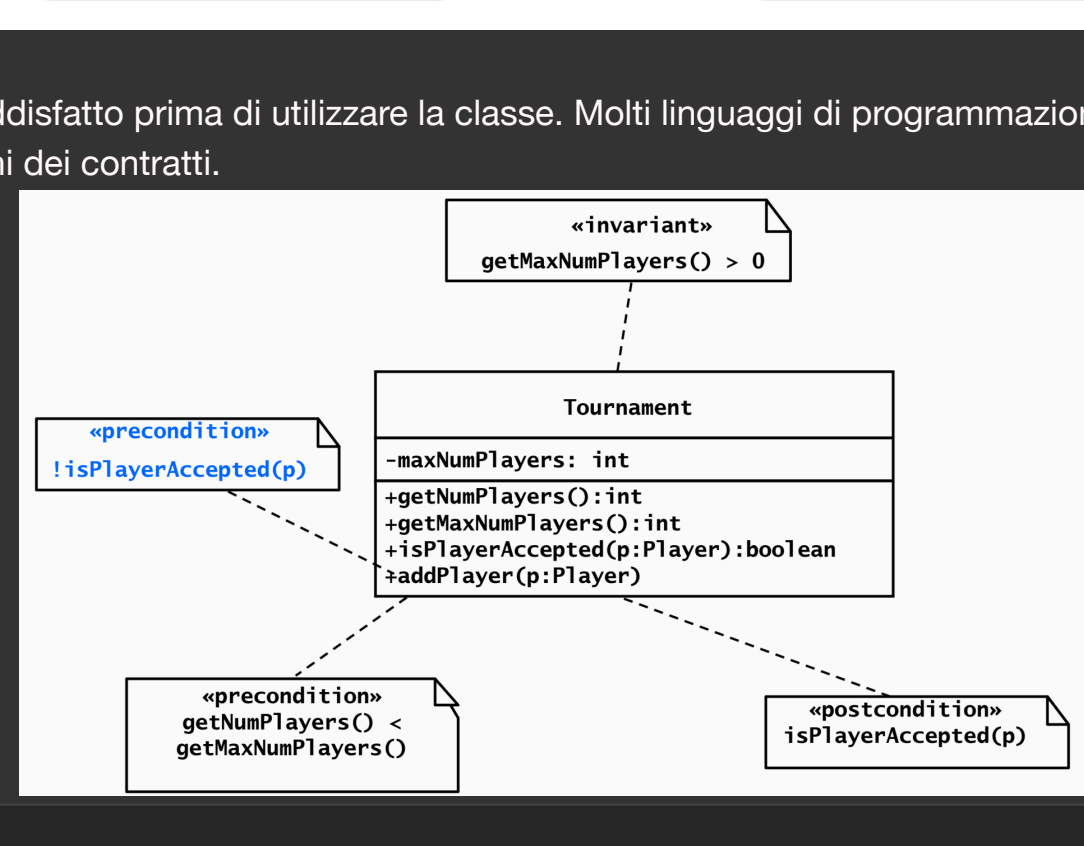
Classi associative

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.



Mappare 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 OO non forniscono supporto per i contratti quindi si utilizza il meccanismo delle eccezioni per segnalare o gestire le violazioni dei contratti.



```
public class TournamentControl {
    private Tournament tournament;

    public void addPlayer(Player p) throws KnownPlayerException {
    if (tournament.isPlayerAccepted(p)) {
        throw new KnownPlayerException(p);
    }
        //... Normal addPlayer behavior
    }
}
public class TournamentForm {
    private TournamentControl control; private List players;
    public void processPlayerApplications() {
        // Go through all the players who applied for this tournament
        for (Iterator i = players.iterator(); i.hasNext();) {
            try {
                // Delegate to the control object.
                control.acceptPlayer((Player)i.next());-->potrebbe lanciare l'eccezione.
                }
            catch (KnownPlayerException e) {
                // If an exception was caught, log it to the console, and // proceed to the next player.
                ErrorConsole.log(e.getMessage());
              }
        }
    }
}
```

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 invariati 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 contrati 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 invariati:
  - 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.
- Riusere codice per il controllo dei vincoli:
  - molte operazioni hanno precondizioni simili;
  - incapsulare il codice per il controllo degli stessi vincoli in metodi cosi possono condividere le stesse classi di eccezioni.

Mappare l'Object Design in schemi di memorizzazione persistenti

Concetto già visto con il corso Database