

Curs 10

Transformarea modelelor în cod

*Suport de curs bazat pe **B. Bruegge and A.H. Dutoit***

"Object-Oriented Software Engineering using UML, Patterns, and Java"

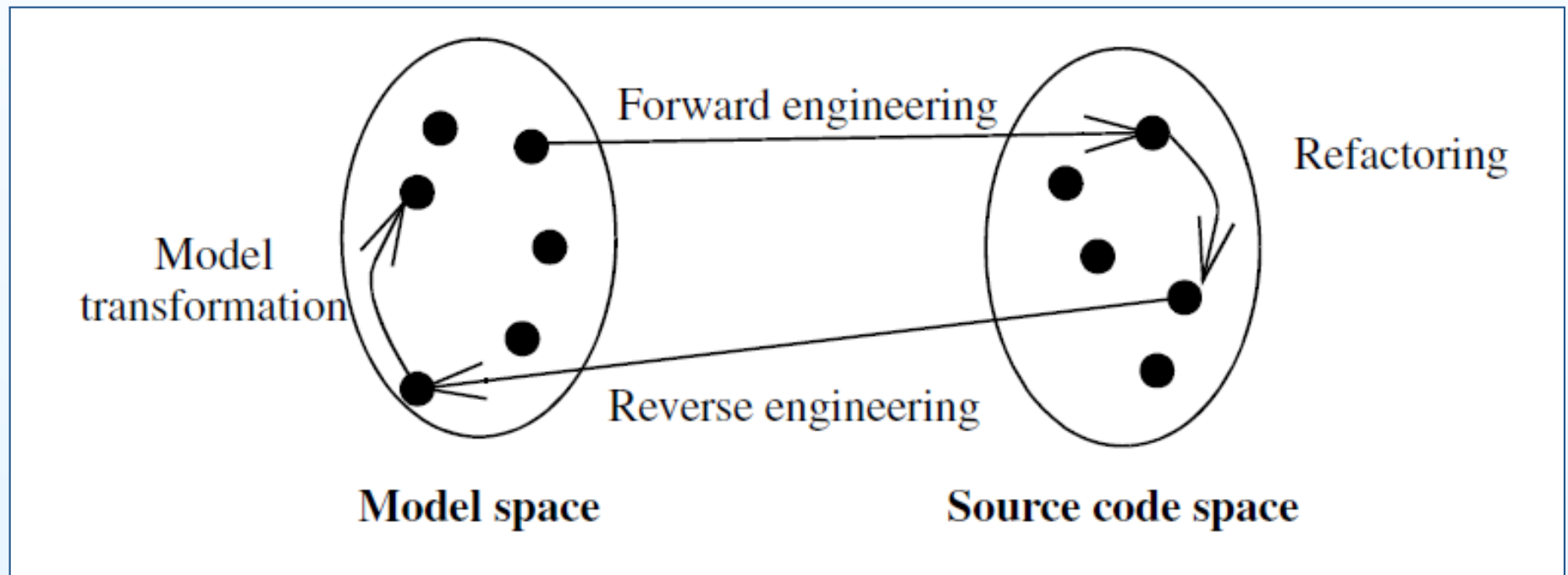
Modele și transformări

- O *transformare* are drept scop îmbunătățirea unei caracteristici a unui model (ex.: modularitatea), cu păstrarea celorlalte proprietăți ale acestuia (ex.: funcționalitatea)
 - O transformare este, de obicei, localizată, afectează un număr relativ mic de clase/atribute/operații și se execută într-o succesiune de pași mărunți
- Astfel de transformări caracterizează preponderent activitățile legate de proiectarea obiectuală și implementarea sistemului
 - *Optimizare* - îndeplinirea cerințelor legate de performanța sistemului, prin
 - reducerea multiplicității asocierilor, pentru a crește viteza interogărilor
 - adăugarea unor asocieri redundante, pentru eficiență
 - introducerea unor atribute derivate, pentru a îmbunătăți timpul de acces la obiecte
 - *Reprezentarea asocierilor* - implementarea asocierilor în cod folosind (colecții de) referințe
 - *Reprezentarea contractelor* - descrierea comportamentului sistemului în cazul violării contractelor, folosind excepții
 - *Reprezentarea entităților persistente* - maparea claselor la nivelul depozitelor de date (baze de date, fișiere text, etc.)

Tipuri de transformări

- *Transformări la nivelul modelului (eng. model transformations)*
 - Operează pe un model, au ca și rezultat un model
 - Ex.: transformarea unui atribut (atribut *adresă*, reprezentată ca și string) într-o clasă (clasa *Adresă*, cu attribute *stradă*, *număr*, *oraș*, *cod poștal* etc.)
- *Refactorizări (eng. refactorings)*
 - Operează pe cod sursă, au ca și rezultat cod sursă
 - Similar transformărilor la nivelul modelului, îmbunătățesc un aspect al sistemului, fără a-i afecta funcționalitatea
- *Inginerie directă (eng. forward engineering)*
 - Produce un fragment de cod aferent unui model obiectual
 - Multe dintre conceptele de modelare (ex.: attribute, asocieri, semnături de operații) pot fi transformate automat în cod sursă; corpul metodelor, precum și metodele adiționale (private) sunt inserate manual de către dezvoltatori
- *Inginerie inversă (eng. reverse engineering)*
 - Produce un model, pe baza unui fragment de cod sursă
 - Utilă atunci când modelul de proiectare nu (mai) există sau atunci când modelul și codul au evoluat desincronizat

Tipuri de transformări (cont.)

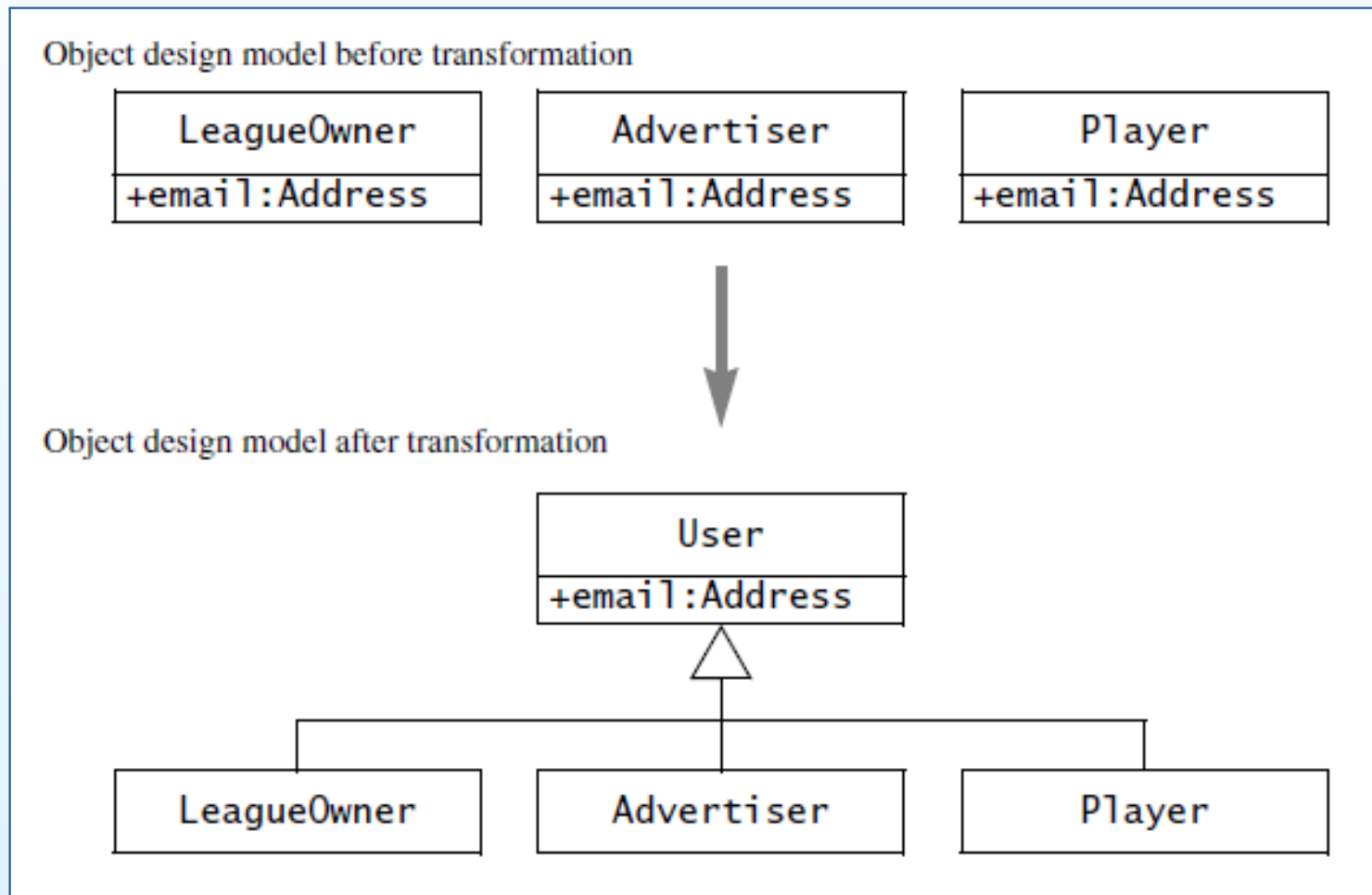


Transformări la nivelul modelului

- O astfel de transformare este aplicată unui model obiectual și rezultă într-un nou model obiectual
- Obiectivul este simplificarea, detalierea sau optimizarea modelului inițial, în conformitate cu cerințele din specificație
- O transformare la nivelul modelului poate să adauge, să elimine sau să redenumască clase, operații, asocieri sau attribute
- Întreg procesul de dezvoltare poate fi considerat ca și o succesiune de transformări de modele, începând cu modelul de analiză și terminând cu cel obiectual de proiectare, fiecare astfel de transformare adăugând detalii care țin de domeniul soluției
- Deși aplicarea unei astfel de transformări poate fi, de cele mai multe ori, automatizată, identificarea tipului de transformare de aplicat, precum și a claselor concrete implicate necesită raționament și experiență

Transformări la nivelul modelului (cont.)

- *Ex. 10.1:* utilizarea unei transformări pentru introducerea unei ierarhii de clase și eliminarea redundanței din modelul obiectual de analiză



Refactorizări

- O *refactorizare* reprezintă o transformare a codului sursă, care crește inteligibilitatea sau modificabilitatea acestuia, fără a-i schimba comportamentul [Fowler, 2000]
 - O refactorizare are drept scop îmbunătățirea design-ului unui sistem funcțional, focusându-se pe o anumită metodă sau pe un anumit câmp al unei clase
 - Pentru a asigura păstrarea neschimbată a comportamentului sistemului, o refactorizare se realizează incremental, pașii de refactorizare fiind intercalați cu teste
- *Ex. 10.2*: Transformarea de model din *Ex. 10.1* corespunde unei serii de 3 refactorizări
 1. Refactorizarea *Pull Up Field*
 - Transferă câmpul *email* din subclase în superclasa *User*
 2. Refactorizarea *Pull Up Constructor Body*
 - Transferă codul de inițializare din subclase în superclasă
 3. Refactorizarea *Pull Up Method*
 - Transferă metodele care utilizează câmpul *email* din subclase în superclasă

Pașii refactorizării *Pull Up Field*

1. Inspectează clasele *Player*, *LeagueOwner* și *Advertiser*, pentru a certifica echivalența semantică a atributelor de tip *e-mail*. Redenumeste attributele echivalente la *email*, dacă este necesar
2. Creează clasa publică *User*
3. Asignează clasa *User* ca și superclasă pentru *Player*, *LeagueOwner* și *Advertiser*
4. Adaugă câmpul protected *email* clasei *User*
5. Șterge câmpul *email* din clasele *Player*, *LeagueOwner* și *Advertiser*
6. Compilează și testează

Before refactoring

```
public class Player {
    private String email;
    //...
}
public class LeagueOwner {
    private String eMail;
    //...
}
public class Advertiser {
    private String email_address;
    //...
}
```

After refactoring

```
public class User {
    protected String email;
}
public class Player extends User {
    //...
}
public class LeagueOwner extends User
{
    //...
}
public class Advertiser extends User {
    //...
}
```


Pașii refactorizării *Pull Up Constructor Body*

1. Adaugă clasei *User* constructorul *User(String email)*
2. În constructor, asignează câmpului *email* valoarea transmisă ca și parametru
3. Înlocuiește corpul constructorului clasei *Player* cu apelul *super(email)*
4. Compilează și testează
5. Repetă pașii 1-4 pentru *LeagueOwner* și *Advertiser*

Before refactoring

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

After refactoring

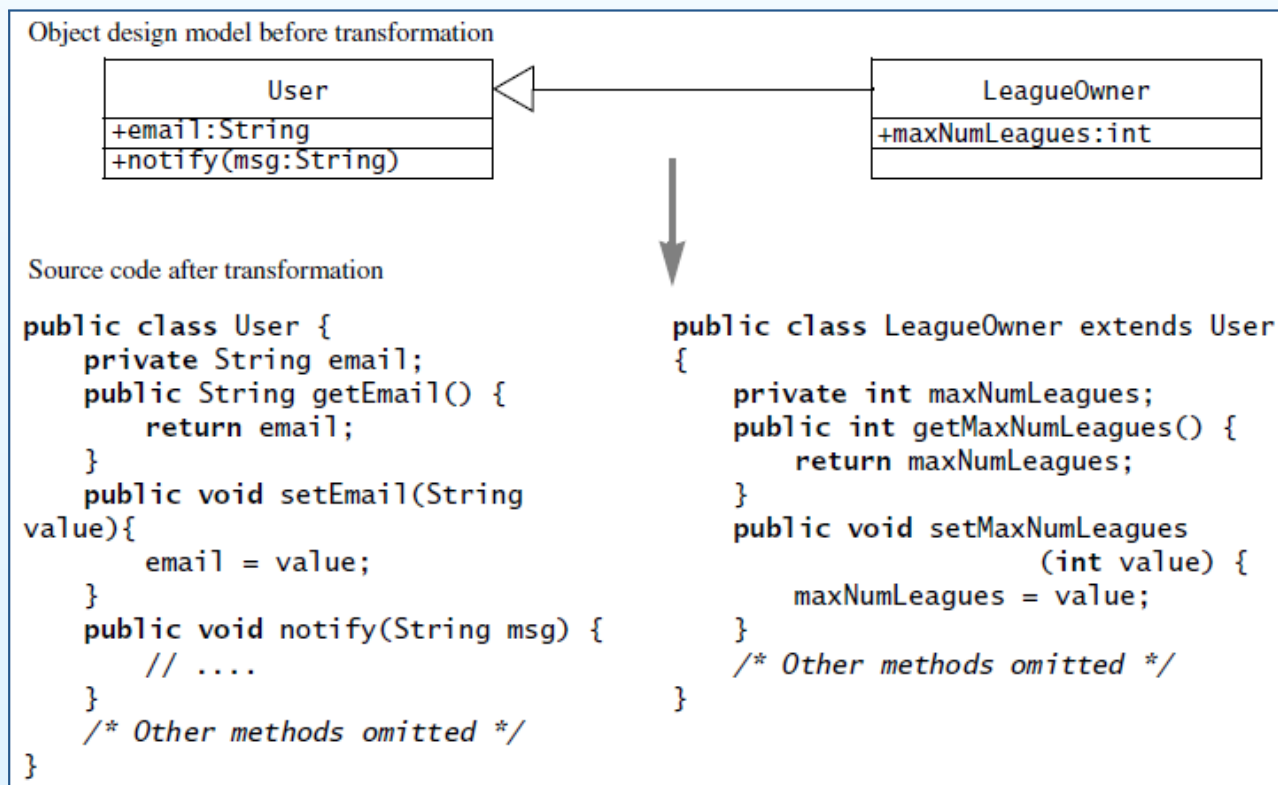
```
public class User {  
    public User(String email) {  
        this.email = email;  
    }  
}  
  
public class Player extends User {  
    public Player(String email) {  
        super(email);  
        //...  
    }  
}  
  
public class LeagueOwner extends User  
{  
    public LeagueOwner(String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

Pașii refactorizării *Pull Up Method*

1. Examinează metodele din *Player* care utilizează câmpul *email*. Presupunem că *Player.notify()* utilizează acest câmp, însă nu folosește nici un alt câmp și nici o altă operație specifice lui *Player*
2. Copiază metoda *notify()* în clasa *User* și recompilează
3. Șterge metoda *Player.notify()*
4. Compilează și testează
5. Repetă pașii 1-4 pentru *LeagueOwner* și *Advertiser*

Inginerie directă

- *Ingineria directă* se aplică unei mulțimi de elemente din model și rezultă într-o mulțime de instrucțiuni într-un limbaj de programare (cod sursă)
 - Scopul ingineriei directe este acela de a întreține o corespondență între modelul obiectual de proiectare și cod și de a reduce numărul de erori introduse la implementare (diminuând astfel efortul de implementare)



Inginerie inversă

- *Ingineria inversă* se aplică unei mulțimi de elemente din codul sursă, rezultând într-o mulțime de elemente de model
 - Scopul ingineriei inverse este acela de a recrea modelul aferent unui sistem, ca urmare a inexistenței/pierderii sale sau a lipsei de sincronizare a acestuia cu codul sursă
 - Este transformarea opusă ingineriei directe (creează o clasă UML pentru fiecare declarație de clasă din codul sursă, adaugă un atribut pentru fiecare câmp al clasei, o operație pentru fiecare metodă)
 - Dat fiind că, prin inginerie directă, se pierde informație din model (ex. asocierile sunt convertite în referințe sau colecții de referințe), ingineria inversă nu va produce, de regulă, același model
 - Majoritatea instrumentelor CASE existente cu suport integrat pentru ingineria inversă oferă cel mult o aproximare care permite dezvoltatorului reconstituirea modelului inițial

Principii de transformare

- 1. Fiecare transformare trebuie să vizeze optimizări din perspectiva unui singur criteriu
 - Ex.: o aceeași transformare nu poate avea drept scop diminuarea timpului de răspuns al sistemului și creșterea inteligibilității codului
 - Încercarea de a adresa mai multe criterii printr-o aceeași transformare crește complexitatea transformării și oferă condiții pentru introducerea unor erori
- 2. Fiecare transformare trebuie să fie locală
 - O transformare trebuie să afecteze doar un număr mic de metode/clase la un moment dat
 - O modificare la nivelul implementării unei metode nu va afecta clienții acesteia
 - Dacă transformarea vizează o interfață, clienții trebuie modificați pe rând
- 3. Fiecare transformare trebuie aplicată izolat de alte schimbări
 - Ex.: Adăugarea unei noi funcționalități și optimizarea codului existent nu se vor opera simultan

Principii de transformare (cont.)

- 4. Fiecare transformare trebuie urmată de validări aferente
 - O transformare care operează doar asupra modelului trebuie urmată de modificarea diagramelor de interacțiune afectate de schimbarea de model efectuată și de revizuirea cazurilor de utilizare aferente, pentru a certifica oferirea funcționalității dorite
 - O refactorizare trebuie urmată de execuția cazurilor de test aferente claselor afectate de schimbările efectuate
 - Introducerea unor noi funcționalități trebuie urmată de proiectarea unor cazuri de test aferente

Optimizarea modelului obiectual de proiectare

- Are drept scop îndeplinirea criteriilor de performanță ale sistemului (legate de timp de răspuns/execuție sau spațiu de memorare)
- Tipuri comune de optimizări
 - Optimizarea căilor de acces
 - Transformarea unor clase în attribute
 - Amânarea operațiilor costisitoare
 - Memorarea (eng. *caching*) rezultatelor operațiilor costisitoare
- Trebuie menținut un echilibru între eficiență și claritate, întrucât transformările care vizează eficientizarea codului, au, de obicei, efecte negative asupra inteligibilității sistemului

Optimizarea căilor de acces

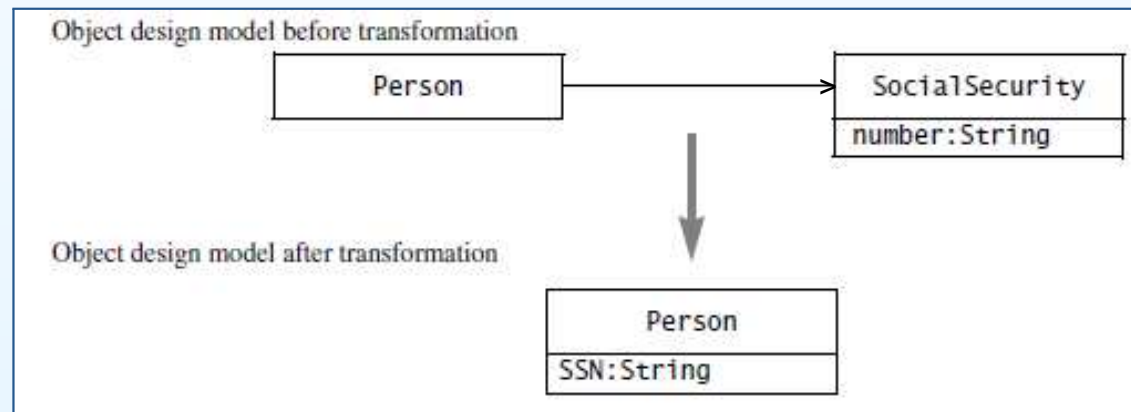
- Surse comune de ineficiență la nivelul unui model obiectual
 - Traversarea repetată a unui număr mare de asocieri
 - Traversarea asocierilor cu multiplicitate *many*
 - Plasarea eronată a unor attribute
- Rezolvarea acestor probleme conduce la un model cu asocieri redundante intenționate, un număr mai mic de relații cu multiplicități *many* și un număr mai mic de clase
- *Traversarea repetată a unui număr mare de asocieri*
 - Operațiile care trebuie executate frecvent și presupun traversarea unui număr mare de asocieri introduc probleme de eficiență
 - Identificarea acestora se realizează urmărind diagramele de interacțiune aferente cazurilor de utilizare
 - Soluția: introducerea unor asocieri directe, redundante, între entitățile interogate și cele care interoghează
 - De cele mai multe ori, aceste transformări se aplică doar în urma testării sistemului, după confirmarea, la execuție, a problemelor de eficiență anticipate

Optimizarea căilor de acces (cont.)

- *Traversarea asocierilor cu multiplicitate many*
 - Soluții: calificarea asocierilor în scopul reducerii multiplicității; ordonarea sau indexarea obiectelor de la capătul aferent multiplicității *many*
- *Plasarea greșită a unor attribute*
 - Apare ca și rezultat al modelării excesive/exagerate în etapa de analiză
 - Soluție: attribute ale unor clase fără comportament relevant (doar metode get/set) pot fi relocate în clasa apelantă
 - Astfel de relocări pot conduce la eliminare din model a unor clase
- **ToDo :)** Imaginați-vă câte o situație de fiecare dintre cele trei tipuri enumerate și soluția de modelare aferentă (model inițial vs. model după transformare). Pentru obținerea unui bonus la curs, trimiteți răspunsul pe adresa vladi@cs.ubbcluj.ro până la finalul zilei în care a fost postat materialul de curs pe pagină.

Transformarea unor clase în atribute

- După restructurări/optimizări repetate ale modelului obiectual, unele clase vor rămâne cu un număr mic de atribute/operații
- Astfel de clase, atunci când sunt asociate cu o singură altă clasă, pot fi contopite cu aceasta, reducând astfel complexitatea modelului
- Ex.:



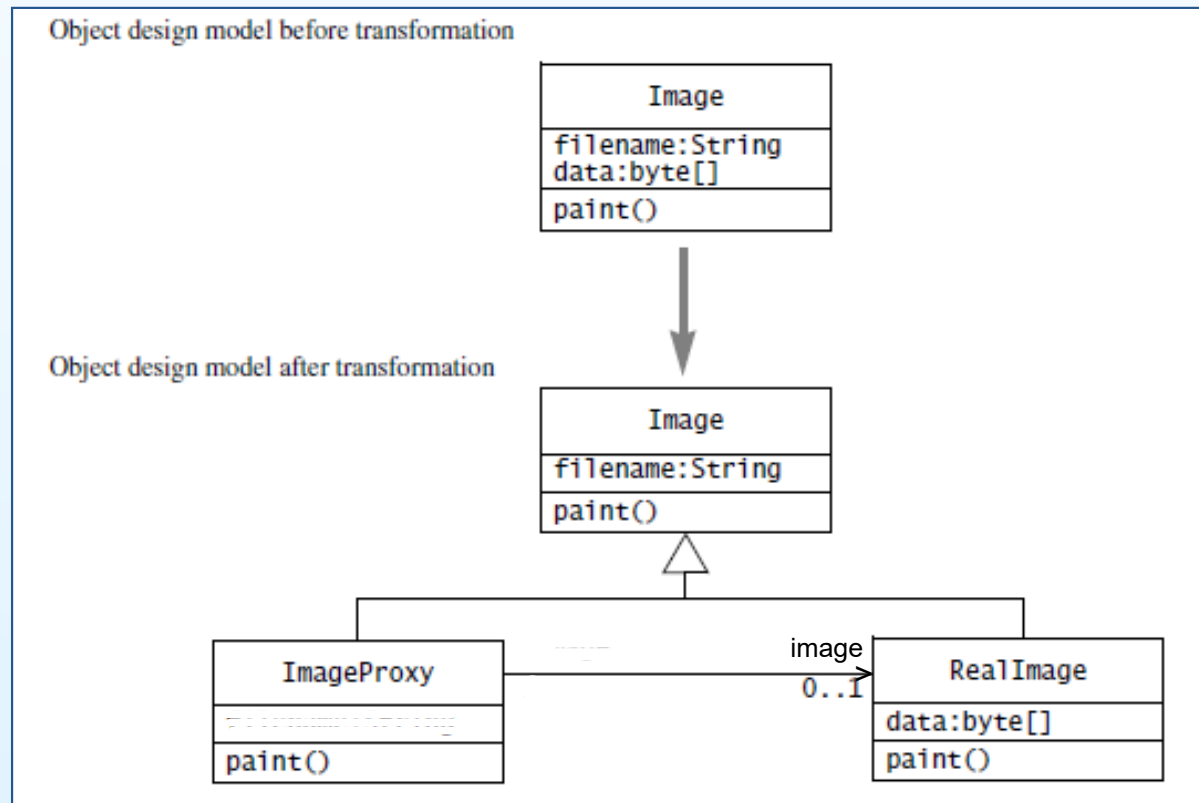
- Clasa *SocialSecurity* nu are comportament propriu netrivial și nici asocieri cu alte clase, exceptând *Person*
- Astfel de transformări trebuie amânate până spre finalul fazei de proiectare/începutul fazei de implementare, atunci când responsabilitățile claselor sunt clare

Transformarea unor clase în attribute (cont.)

- Acestei transformări îi corespunde un caz particular al refactorizării *Inline Class* [Fowler, 2000]
- Pași refactorizării *Inline Class*
 - Declară toate câmpurile și metodele publice ale clasei sursă în clasa destinație
 - Schimbă toate referințele spre clasa sursă către clasa destinație
 - Schimbă numele clasei sursă, pentru a identifica eventualele referințe nemodificate
 - Compilează și testează
 - Șterge clasa sursă

Gestionare operațiilor costisitoare

- Operațiile costisitoare, de tipul încărcării unor obiecte grafice, pot fi amânate până în momentul în care este necesară vizualizarea acestora
 - Soluție: aplicarea șablonului *Proxy*
 - Ex.:



Gestionare operațiilor costisitoare (cont.)

- Un obiect de tip *ImageProxy* (image surogat) ia locul obiectului *RealImage* (image reală), oferind aceeași interfață cu acesta
- Obiectul surogat răspunde la solicitări simple și încarcă obiectul real doar în momentul în care i se apelează operația `paint()` (mesajul de desenare va fi apoi delegat obiectului image real)
- În cazul în care clienții nu apelează `paint()`, obiectul image real nu va fi creat niciodată
- Rezultatele unor operații complexe, apelate frecvent, însă bazate pe valori care nu se schimbă sau se schimbă destul de rar, pot fi memorate la nivelul unor attribute private
 - Soluția optimizează timpul de răspuns la apelul operațiilor, însă consumă spațiu de memorie suplimentar pentru stocarea unor informații redundante

Reprezentarea asocierilor

- UML: *asocieri* = mulțimi de legături între obiecte
- Limbaje de programare: *referințe* / *colecții de referințe*
- Implementarea asocierilor în cod ține cont de *navigabilitate*, *multiplicități*, *nume de roluri* și de semantica domeniului
 - Bidirecționalitatea introduce dependențe mutuale între clase (se traduce prin perechi de referințe ce trebuie sincronizate)
 - multiplicitatea *one* necesită o referință, cea *many* - o colecție de referințe
 - numele de roluri corespund numelor de câmpuri adăugate în clase

Asocieri unidirecționale *one-to-one*

- Model



- Cod sursă după transformare

```
public class Advertiser {

    private Account account;

    public Advertiser()
    {
        account = new Account();
    }

    public Account getAccount()
    {
        return account;
    }

    // nu ofera setter
}
```

Asocieri bidirecționale *one-to-one*

- Model



- Cod sursă după transformare

```
public class Advertiser {

    /* The account field is initialized
       in the constructor
       and never modified */
    private Account account;

    public Advertiser()
    {
        account = new Account(this);
    }

    public Account getAccount()
    {
        return account;
    }

}
```

```
public class Account {

    /* The owner field is initialized
       in the constructor
       and never modified. */
    private Advertiser owner;

    public Account(Advertiser owner)
    {
        this.owner = owner;
    }

    public Advertiser getOwner()
    {
        return owner;
    }

}
```


Asocieri bidirecționale *one-to-many*

- Model



- Cod sursă după transformare

```
public class Advertiser {

    private Set<Account> accounts;

    public Advertiser()
    {
        accounts = new HashSet();
    }

    public void addAccount(Account account)
    {
        if(!accounts.contains(account))
        {
            accounts.add(account);
            account.internalSetOwner(this);
        }
    }
}
```

```
public class Account {

    private Advertiser owner;

    public void setOwner(Advertiser owner)
    {
        Advertiser oldOwner = this.owner;
        Advertiser newOwner = owner;
        if(oldOwner != null)
            oldOwner.internalRemoveAccount(this);
        if(newOwner != null)
            newOwner.internalAddAccount(this);
        this.owner = newOwner;
    }
}
```

Asocieri bidirecționale *one-to-many* (cont.)

```
public void removeAccount(Account account)
{
    if(accounts.contains(account))
    {
        accounts.remove(account);
        account.internalSetOwner(null);
    }
}

void internalAddAccount(Account account)
{
    if(!accounts.contains(account))
        accounts.add(account);
}

void internalRemoveAccount(Account account)
{
    if(accounts.contains(account))
        accounts.remove(account);
}

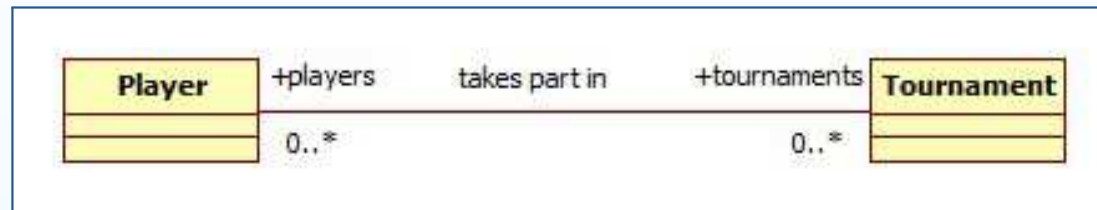
public Set<Account> getAccounts()
{
    return Collections.unmodifiableSet(accounts);
}
}
```

```
void internalSetOwner(Advertiser owner)
{
    this.owner = owner;
}

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

Asocieri bidirecționale *many-to-many*

- Model



- Cod sursă după transformare

```
public class Player {

    private Set<Tournament> tournaments;

    public Player()
    {
        tournaments = new HashSet();
    }

    public void addTournament(Tournament tournament)
    {
        //pre: tournament != null
        if(!tournaments.contains(tournament))
        {
            tournaments.add(tournament);
            tournament.internalAddPlayer(this);
        }
    }
}
```

```
public class Tournament {

    private Set<Player> players;

    public Tournament()
    {
        players = new HashSet();
    }

    public void addPlayer(Player player)
    {
        //pre: player != null
        if(!players.contains(player))
        {
            players.add(player);
            player.internalAddTournament(this);
        }
    }
}
```

Asocieri bidirecționale *many-to-many* (cont.)

```
public void removeTournament(Tournament tournament)
{
    //pre: tournament != null
    if(tournaments.contains(tournament))
    {
        tournaments.remove(tournament);
        tournament.internalRemovePlayer(this);
    }
}

void internalAddTournament(Tournament tournament)
{
    //pre: tournament != null
    if(!tournaments.contains(tournament))
        tournaments.add(tournament);
}

void internalRemoveTournament(Tournament tournament)
{
    //pre: tournament != null
    if(tournaments.contains(tournament))
        tournaments.remove(tournament);
}

public Set<Tournament> getTournaments()
{
    return Collections.unmodifiableSet(tournaments);
}
}
```

```
public void removePlayer(Player player)
{
    //pre: tournament != null
    if(players.contains(player))
    {
        players.remove(player);
        player.internalRemoveTournament(this);
    }
}

void internalAddPlayer(Player player)
{
    //pre: tournament != null
    if(!players.contains(player))
        players.add(player);
}

void internalRemovePlayer(Player player)
{
    //pre: tournament != null
    if(players.contains(player))
        players.remove(player);
}

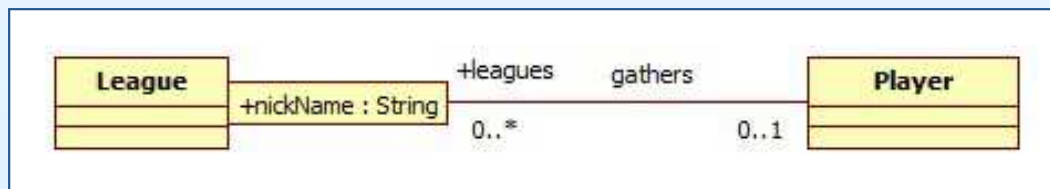
public Set<Player> getPlayers()
{
    return Collections.unmodifiableSet(players);
}
```

Asocieri calificate

- Asocierile calificate sunt utilizate pentru a "reduce" multiplicitatea unui capăt *many* din cadrul unei asocieri *one-to-many* sau *many-to-many*
 - Calificatorul asocierii este un atribut al clasei din capătul *many* care se dorește a fi redus, atribut care are valori unice în contextul asocierii, însă nu neapărat unice la nivel global
 - Ex.: Pentru a putea fi ușor identificați în cadrul unei ligi, jucătorii își pot alege un *nickName* care trebuie să fie unic în cadrul ligii (jucătorii pot avea *nickName*-uri diferite în ligi diferite, iar fiecare astfel de *nickName* nu trebuie să fie unic la nivel global)
 - Modelul înainte de transformare



- Modelul după transformare



Asocieri calificate (cont.)

```
public class League {

    private Map<String, Player> players;

    public League()
    {
        players = new HashMap();
    }

    public void addPlayer(String nickName, Player player)
    {
        if(!players.containsKey(nickName))
        {
            players.put(nickName, player);
            player.internalAddLeague(this);
        }
    }

    public Player getPlayer(String nickName)
    {
        return players.get(nickName);
    }

    void internalAddPlayer(String nickName, Player player)
    {
        if(!players.containsKey(nickName))
            players.put(nickName, player);
    }
}
```

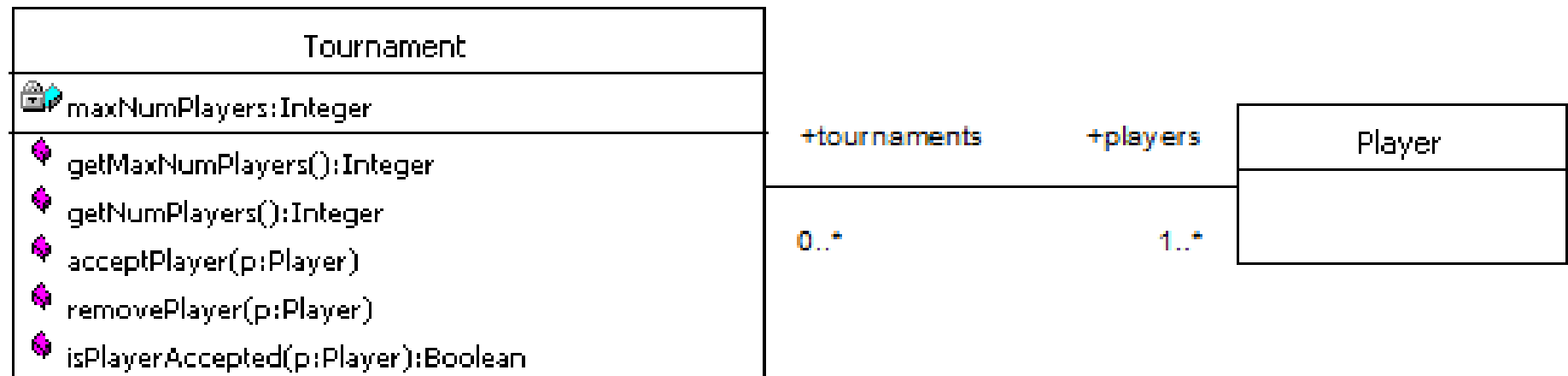
Asocieri calificate (cont.)

```
public class Player {  
  
    private Set<League> leagues;  
  
    public Player()  
    {  
        leagues = new HashSet();  
    }  
  
    public void addLeague(League league, String nickName)  
    {  
        if (league.getPlayer(nickName) == null)  
        {  
            leagues.add(league);  
            league.internalAddPlayer(nickName, this);  
        }  
    }  
  
    void internalAddLeague(League league)  
    {  
        leagues.add(league);  
    }  
  
}
```

Reprezentarea contractelor

- *Verificarea precondițiilor*
 - Precondițiile trebuie verificate la începutul fiecărei metode, înaintea efectuării procesărilor caracteristice. În cazul în care precondiția nu este adevărată, se va arunca o excepție. Se recomandă ca fiecare precondiție să corespundă unui tip particular de excepție.
- *Verificarea postcondițiilor*
 - Postcondițiile trebuie verificate la sfârșitul fiecărei metode, după terminarea tuturor procesărilor caracteristice și finalizarea schimbărilor de stare. În cazul în care contractul este violat, se va arunca o excepție specifică.
- *Verificarea invarianțelor*
 - Invarianții se vor verifica odată cu postcondițiile (la finalizarea fiecărei metode publice a clasei)
- *Moștenirea contractelor*
 - Codul de verificare al aserțiunilor trebuie încapsulat la nivelul unor metode specifice, pentru a permite apelarea acestora din subclase

Ex.: contract OCL



```
context Tournament
  inv maxNumPlayersPositive:
    self.getMaxNumPlayers() > 0

context Tournament::acceptPlayer(p: Player)
  pre: self.getNumPlayers() < self.getMaxNumPlayers() and
       not self.isPlayerAccepted(p)
  post: self.isPlayerAccepted(p) and
        self.getNumPlayers() = self@pre.getNumPlayers() + 1
```

Ex.: implementarea contractului

```
public class Tournament {  
    //...  
    private List players;  
  
    public void acceptPlayer(Player p)  
        throws KnownPlayer, TooManyPlayers, UnknownPlayer,  
            IllegalNumPlayers, IllegalMaxNumPlayers  
    {  
        // check precondition !isPlayerAccepted(p)  
        if (isPlayerAccepted(p)) {  
            throw new KnownPlayer(p);  
        }  
        // check precondition getNumPlayers() < maxNumPlayers  
        if (getNumPlayers() == getMaxNumPlayers()) {  
            throw new TooManyPlayers(getNumPlayers());  
        }  
        // save values for postconditions  
        int pre_getNumPlayers = getNumPlayers();  
    }  
}
```

Ex.: implementarea contractului (cont.)

```
// accomplish the real work
players.add(p);
p.addTournament(this);

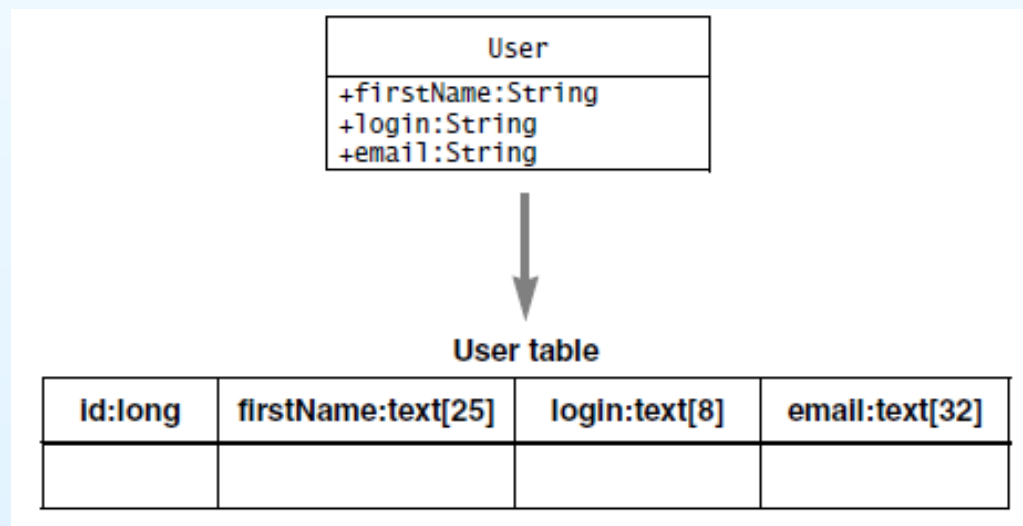
// check post condition isPlayerAccepted(p)
if (!isPlayerAccepted(p)) {
    throw new UnknownPlayer(p);
}
// check post condition getNumPlayers() = @pre.getNumPlayers() + 1
if (getNumPlayers() != pre_getNumPlayers + 1) {
    throw new IllegalNumPlayers(getNumPlayers());
}
// check invariant maxNumPlayers > 0
if (getMaxNumPlayers() <= 0) {
    throw new IllegalMaxNumPlayers(getMaxNumPlayers());
}
}
//...
}
```

Reprezentarea contractelor (cont.)

- Dezavantaje ale unei implementări/monitorizări manuale exhaustive a indeplinirii contractelor
 - *Efortul de codificare* - cod de verificare uneori mai complex decât logica operației în sine
 - *Șanse mari de introducere a unor erori*
 - *Posibilitatea de mascare a unor defecte în codul aferent funcționalității* - în cazul în care cele două sunt scrise de către același programator
 - *Dificultatea modificării codului în cazul modificării constrângerii*
 - *Probleme de performanță* la monitorizarea exhaustivă
- *Soluții*
 - Generarea automată a codului de verificare aferent contractelor folosind instrumente CASE dedicate (ex. OCLE)
 - Monitorizarea selectivă
 - la testare - toate aserțiunile
 - la exploatare - selectiv, funcție de performanțele dorite, gradul de încredere în calitatea codului și natura critică a aplicației

Reprezentarea entităților persistente

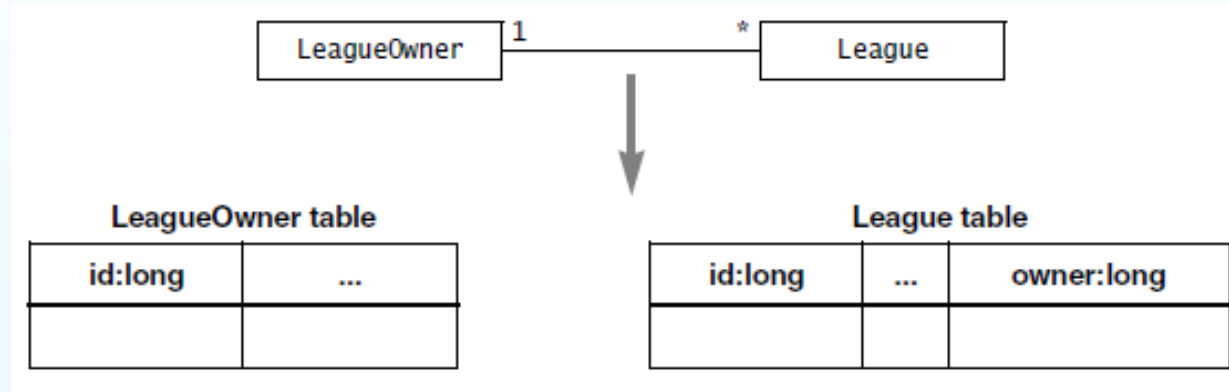
- *Reprezentarea claselor și atributelor*
 - Fiecare clasă se reprezintă folosind un tabel cu același nume
 - Pentru fiecare atribut al clasei se adaugă în tabel o coloană cu același nume
 - Fiecare linie a unui tabel va corespunde unei instanțe a clasei
 - În mod ideal, cheia primară ar trebui să fie un identificator unic (eventual autoincrement), diferit de attributele proprii ale clasei în cauză. Alegerea ca și cheie a unui atribut (grup) caracteristice tipului de entitate este problematică în momentul în care apar modificări la nivelul domeniului aplicației



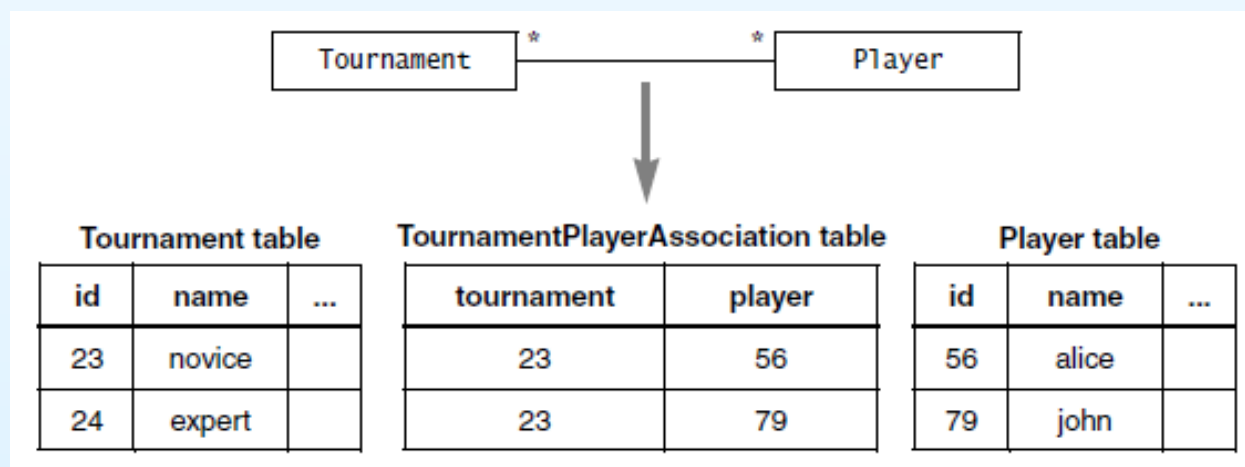
Reprezentarea entităților persistente (cont.)

- *Reprezentarea asocierilor*

- Asocierile *one-to-one* și *one-to-many* se reprezintă folosind chei străine

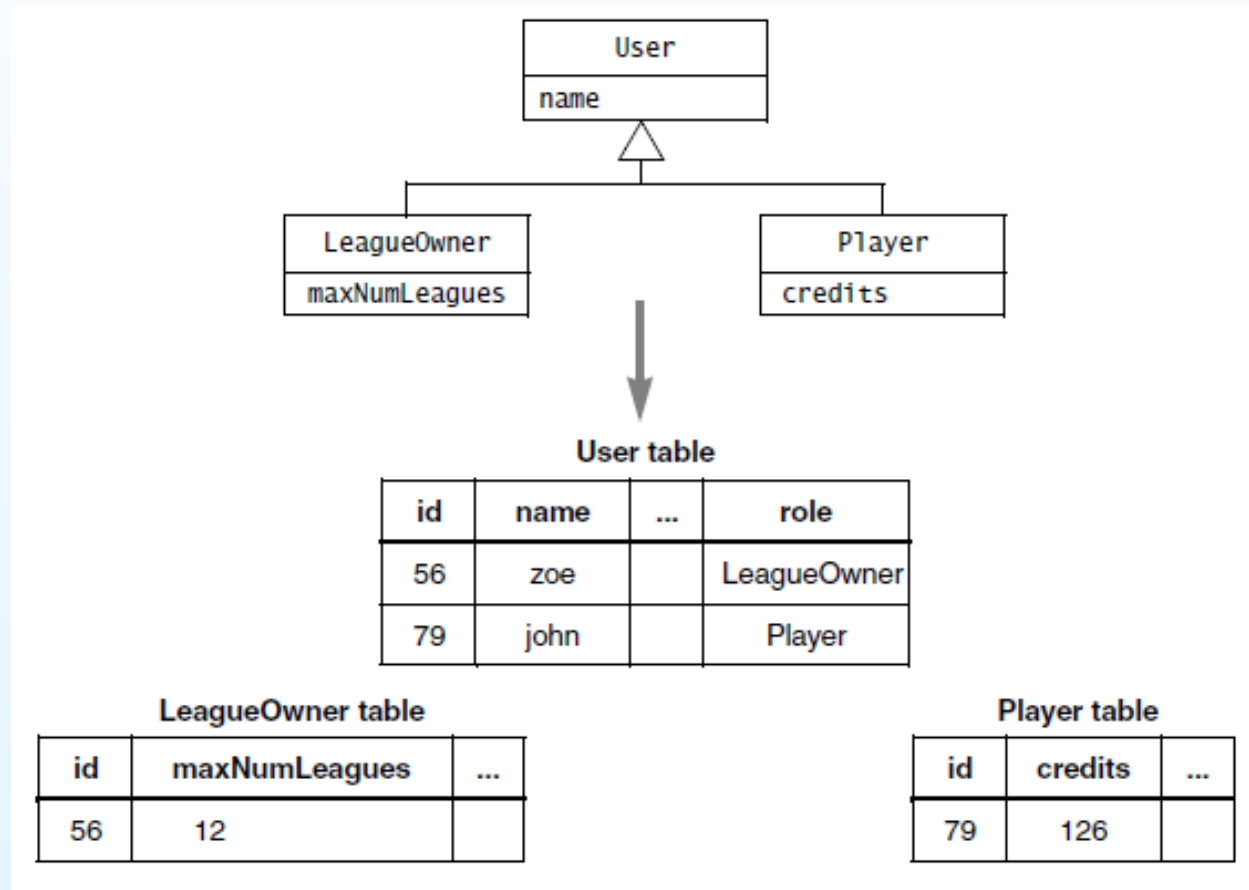


- Asocierile *many-to-many* se reprezintă folosind tabele de legătură



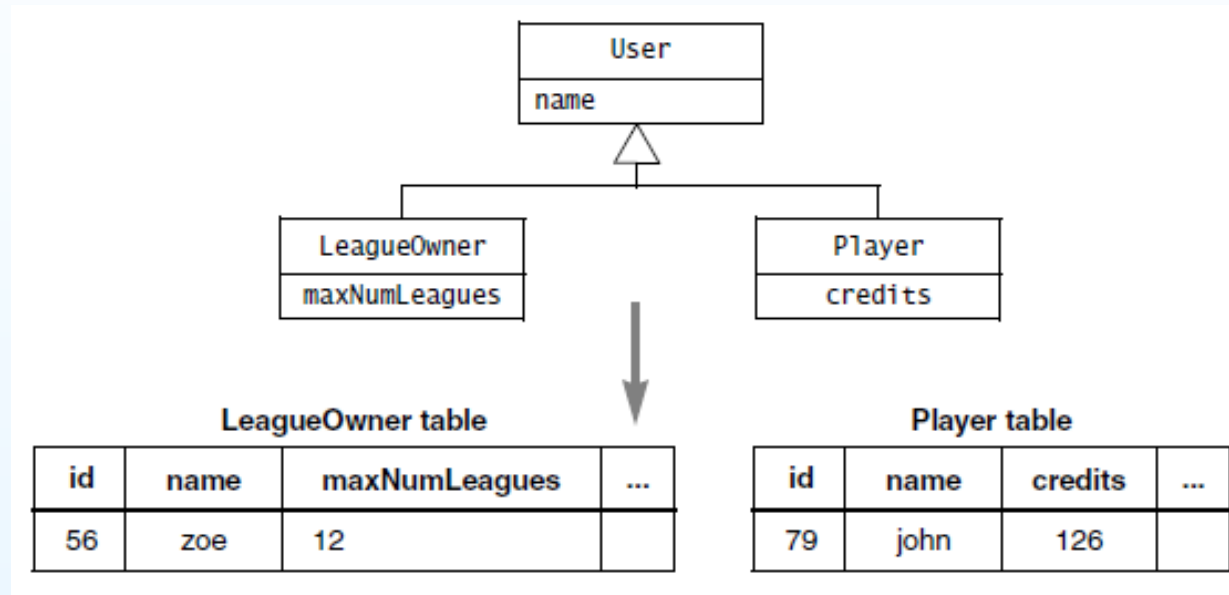
Reprezentarea entităților persistente (cont.)

- *Reprezentarea moștenirii*
 - Mapare verticală



Reprezentarea entităților persistente (cont.)

- Mapare orizontală



- Mapare verticală vs. mapare orizontală = modificabilitate vs. performanță
 - Maparea verticală: adăugarea unui atribut în clasa de bază => adăugarea unei coloane în tabelul aferent; adăugarea unei noi clase derivate => definirea unui tabel cu attributele proprii ale acesteia; fragmentarea obiectelor individuale => interogări mai lente

Reprezentarea entităților persistente (cont.)

- Maparea orizontală: adăugarea unui atribut în clasa de bază => adăugarea unei coloane în fiecare dintre tabelele aferente claselor derivate; adăugarea unei noi clase derivate => definirea unui tabel cu attributele proprii + cele moștenite; nefragmentarea obiectelor individuale => interogări rapide

Referințe

- [Fowler, 2000] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Reading, MA, 2000.