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"

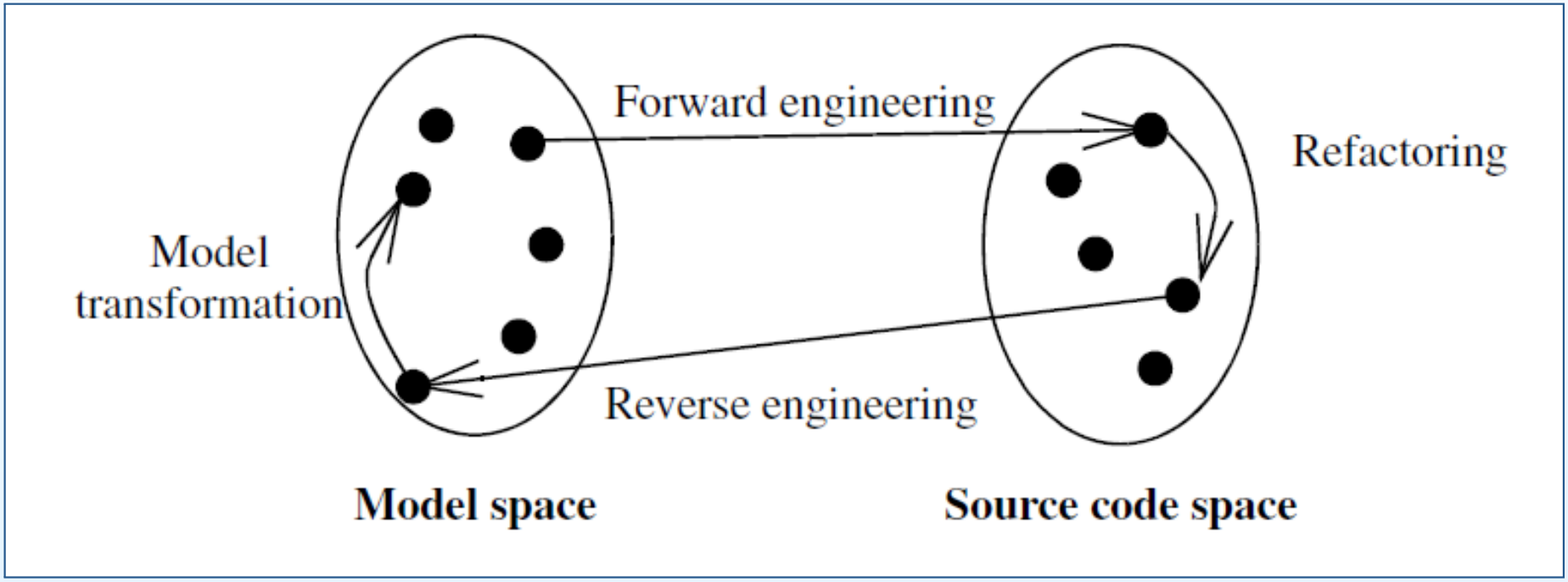
Modele şi transformări

- O *transformare* are drept scop îmbunătăţirea unei caracteristici a unui model (ex.: modularitatea), cu păstrarea celorlalte proprietăţi ale acestuia (ex.: funcţionalitatea)
  - O transformare este, de obicei, localizată, afectează un număr relativ mic de clase/atribute/operaţii şi se execută într-o succesiune de paşi mărunţi
- Astfel de transformări caracterizează preponderent activităţile legate de proiectarea obiectuală şi implementarea sistemului
  - *Optimizare* - îndeplinirea cerinţelor legate de performanţa sistemului, prin
    - reducerea multiplicităţii asocierilor, pentru a creşte viteza interogărilor
    - adăugarea unor asocieri redundante, pentru eficienţă
    - introducerea unor atribute derivate, pentru a îmbunătăţi timpul de acces la obiecte
  - *Reprezentarea asocierilor* - implementarea asocierilor în cod folosind (colecţii de) referinţe
  - *Reprezentarea contractelor* - descrierea comportamentului sistemului în cazul violării contractelor, folosind excepţii
  - *Reprezentarea entităţilor persistente* - maparea claselor la nivelul depozitelor de date (baze de date, fisiere text, etc.)

Tipuri de transformări

- *Transformări la nivelul modelului* (eng. *model transformations*)
  - Operează pe un model, au ca şi rezultat un model
  - Ex.: transformarea unui atribut (atribut *adresă*, reprezentată ca şi string) într-o clasă (clasa *Adresă*, cu atribute *stradă*, *număr*, *oraş*, *cod poştal* etc.)
- *Refactorizări* (eng. *refactorings*)
  - Operează pe cod sursă, au ca şi rezultat cod sursă
  - Similar transformărilor la nivelul modelului, îmbunătăţesc un aspect al sistemului, fără a-i afecta funcţionalitatea
- *Inginerie directă* (eng. *forward engineering*)
  - Produce un fragment de cod aferent unui model obiectual
  - Multe dintre conceptele de modelare (ex.: atribute, asocieri, signaturi de operaţii) pot fi transformate automat în cod sursă; corpul metodelor, precum şi metodele adiţionale (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 şi codul au evoluat desincronizat

Tipuri de transformări (cont.)

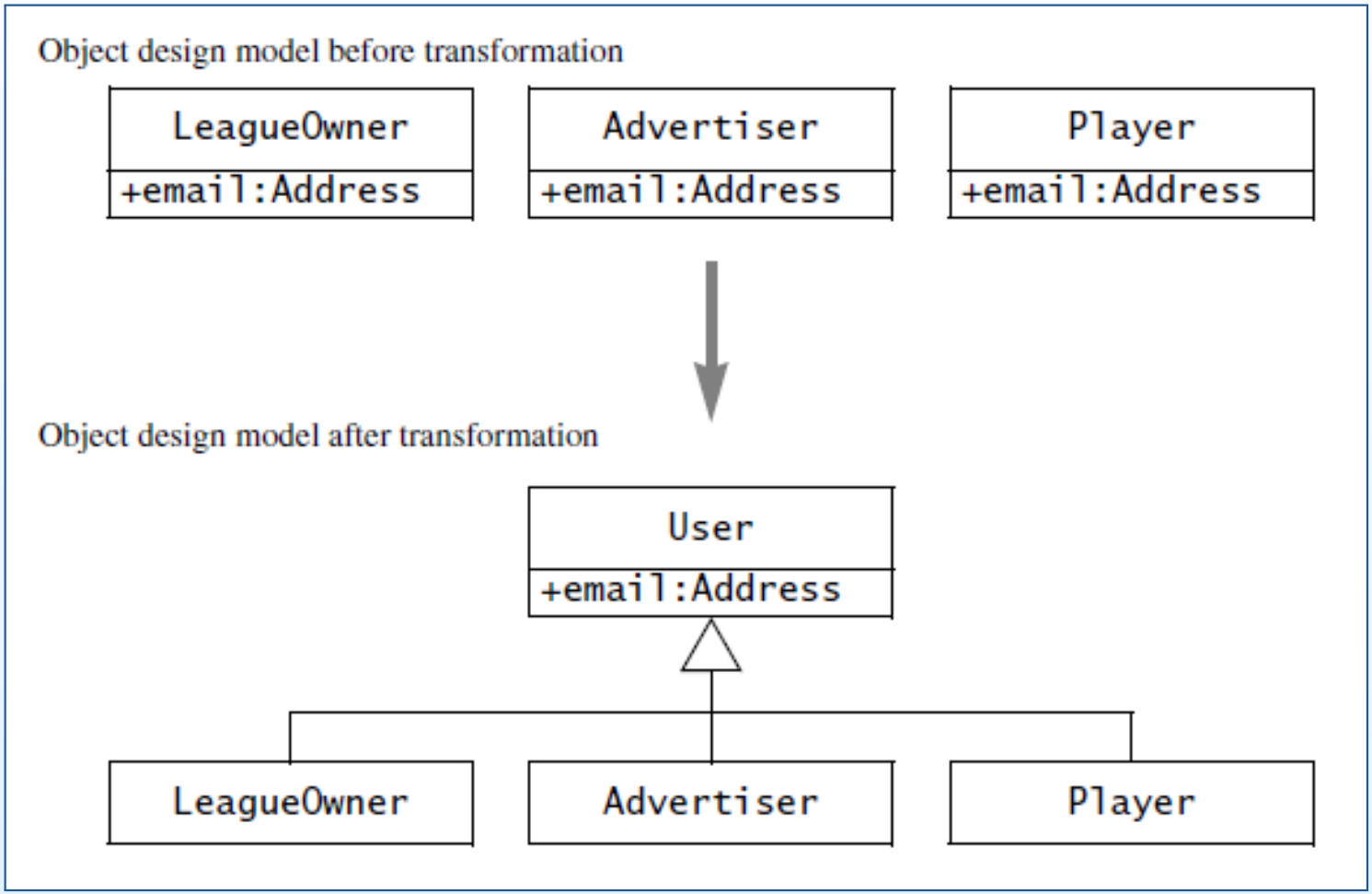


Transformări la nivelul modelului

- O astfel de transformare este aplicată unui model obiectual şi rezultă într-un nou model obiectual
- Obiectivul este simplificarea, detalierea sau optimizarea modelului iniţial, în conformitate cu cerinţele din specificaţie
- O transformare la nivelul modelului poate să adauge, să elimine sau să redenumească clase, operaţii, asocieri sau atribute
- Întreg procesul de dezvoltare poate fi considerat ca şi o succesiune de transformări de modele, începând cu modelul de analiză şi terminând cu cel obiectual de proiectare, fiecare astfel de transformare adăugând detalii care ţin de domeniul soluţiei
- Deşi aplicarea unei astfel de transformări poate fi, de cele mai multe ori, automatizată, identificarea tipului de transformare de aplicat, precum şi a claselor concrete implicate necesită raţionament şi experienţă

Transformări la nivelul modelului (cont.)

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



Refactorizări

- O *refactorizare* reprezintă o transformare a codului sursă, care creşte inteligibilitatea sau modificabilitatea acestuia, fără a-i schimba comportamentul [Fowler, 2000]
  - O refactorizare are drept scop îmbunătăţirea design-ului unui sistem funcţional, 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, paşii de refactorizare fiind intercalaţi 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 iniţializare din subclase în superclasă
  3. Refactorizarea *Pull Up Method*
     - Transferă metodele care utilizează câmpul *email* din subclase în superclasă

Paşii refactorizării *Pull Up Field*

1. Inspectează clasele *Player*, *LeagueOwner* şi *Advertiser*, pentru a certifica echivalenţa semantică a atributelor de tip *e-mail*. Redenumeşte atributele echivalente la *email*, dacă este necesar
2. Creează clasa publică *User*
3. Asignează clasa *User* ca şi superclasă pentru *Player*, *LeagueOwner* şi *Advertiser*
4. Adaugă câmpul protected *email* clasei *User*
5. Şterge câmpul *email* din clasele *Player*, *LeagueOwner* şi *Advertiser*
6. Compilează şi 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 {
    //...
}
```

Paşii refactorizării *Pull Up Constructor Body*

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

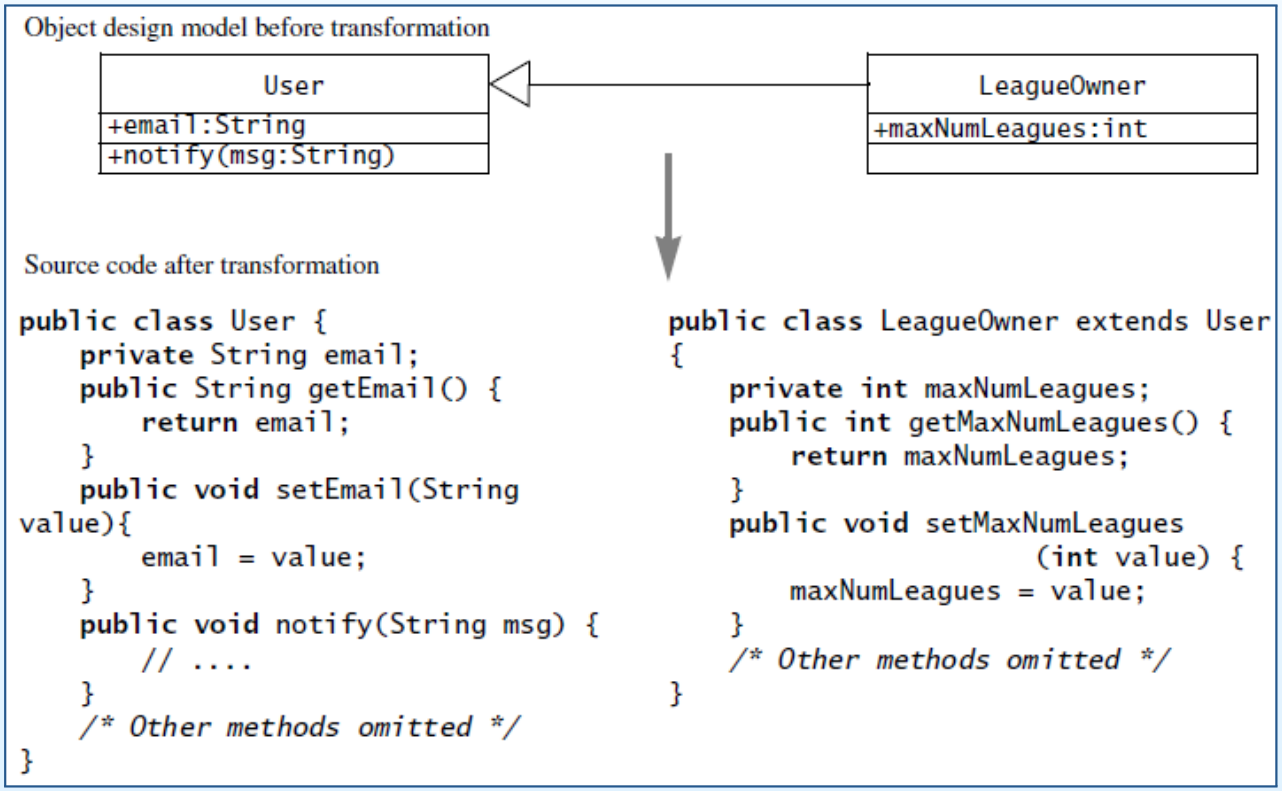
```
public class User {
    public User(String email) {
        this.email = email;
    }
}
public class Player extends User {
    public Player(String email) {
        super(email);
    //...
    }
}
public class LeagueOwner extends User
{
    public LeagueOwner(String email) {
        super(email);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

Paşii refactorizării *Pull Up Method*

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

Inginerie directă

- *Ingineria directă* se aplică unei mulţimi de elemente din model şi rezultă într-o mulţime de instrucţiuni într-un limbaj de programare (cod sursă)
  - Scopul ingineriei directe este acela de a întreţine o corespondenţă între modelul obiectual de proiectare şi cod şi de a reduce numărul de erori introduse la implementare (diminuând astfel efortul de implementare)



Inginerie inversă

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

Principii de transformare

- 1. Fiecare transformare trebuie să vizeze optimizări din perspectiva unui singur criteriu
  - Ex.: o aceeaşi transformare nu poate avea drept scop diminuarea timpului de răspuns al sistemului şi creşterea inteligibilităţii codului
  - Încercarea de a adresa mai multe criterii printr-o aceeaşi transformare creşte complexitatea transformării şi oferă condiţii 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 clienţii acesteia
  - Dacă transformarea vizează o interfaţă, clienţii trebuie modificaţi pe rând
- 3. Fiecare transformare trebuie aplicată izolat de alte schimbări
  - Ex.: Adăugarea unei noi funcţionalităţi şi 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 interacţiune afectate de schimbarea de model efectuată şi de revizuirea cazurilor de utilizare aferente, pentru a certifica oferirea funcţionalităţii dorite
  - O refactorizare trebuie urmată de execuţia cazurilor de test aferente claselor afectate de schimbările efectuate
  - Introducerea unor noi funcţionalităţi trebuie urmată de proiectarea unor cazuri de test aferente

Optimizarea modelului obiectual de proiectare

- Are drept scop îndeplinirea criteriilor de performanţă ale sistemului (legate de timp de răspuns/execuţie sau spaţiu de memorare)
- Tipuri comune de optimizări
  - Optimizarea căilor de acces
  - Transformarea unor clase în atribute
  - Amânarea operaţiilor costisitoare
  - Memorarea (eng. *caching*) rezultatelor operaţiilor costisitoare
- Trebuie menţinut un echilibru între eficienţă şi claritate, întrucât transformările care vizează eficientizarea codului, au, de obicei, efecte negative asupra inteligibilităţii sistemului

Optimizarea căilor de acces

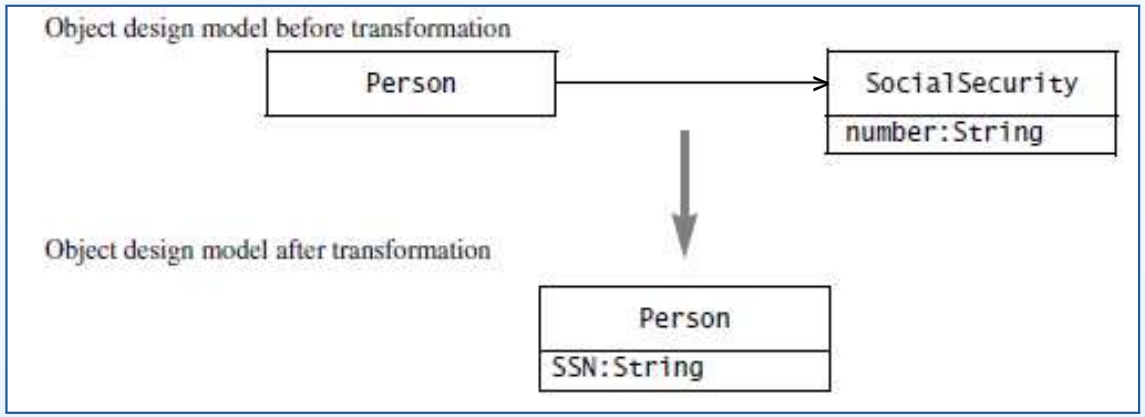
- Surse comune de ineficienţă la nivelul unui model obiectual
  - Traversarea repetată a unui număr mare de asocieri
  - Traversarea asocierilor cu multiplicitate *many*
  - Plasarea eronată a unor atribute
- Rezolvarea acestor probleme conduce la un model cu asocieri redundante intenţionate, un număr mai mic de relaţii cu multiplicităţi *many* şi un număr mai mic de clase
- *Traversarea repetată a unui număr mare de asocieri*
  - Operaţiile care trebuie executate frecvent şi presupun traversarea unui număr mare de asocieri introduc probleme de eficienţă
  - Identificarea acestora se realizează urmărind diagramele de interacţiune aferente cazurilor de utilizare
  - Soluţia: introducerea unor asocieri directe, redundante, între entităţile interogate şi cele care interoghează
  - De cele mai multe ori, aceste transformări se aplică doar în urma testării sistemului, după confirmarea, la execuţie, a problemelor de eficienţă anticipate

Optimizarea căilor de acces (cont.)

- *Traversarea asocierilor cu multiplicitate many*
  - Soluţii: calificarea asocierilor în scopul reducerii multiplicităţii; ordonarea sau indexarea obiectelor de la capătul aferent multiplicităţii *many*
- *Plasarea greşită a unor atribute*
  - Apare ca şi rezultat al modelării excesive/exagerate în etapa de analiză
  - Soluţie: atribute 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 :)** Imaginaţi-vă câte o situaţie de fiecare dintre cele trei tipuri enumerate şi soluţia de modelare aferentă (model iniţial vs. model după transformare). Pentru obţinerea unui bonus la curs, trimiteţi răspunsul pe adresa `vladi@cs.ubbcluj.ro` până la finalul zilei in 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/operaţii
- 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 şi nici asocieri cu alte clase, exceptând *Person*
  - Astfel de transformări trebuie amânate până spre finalul fazei de proiectare/începutul fazei de implementare, atunci când responsabilităţile claselor sunt clare

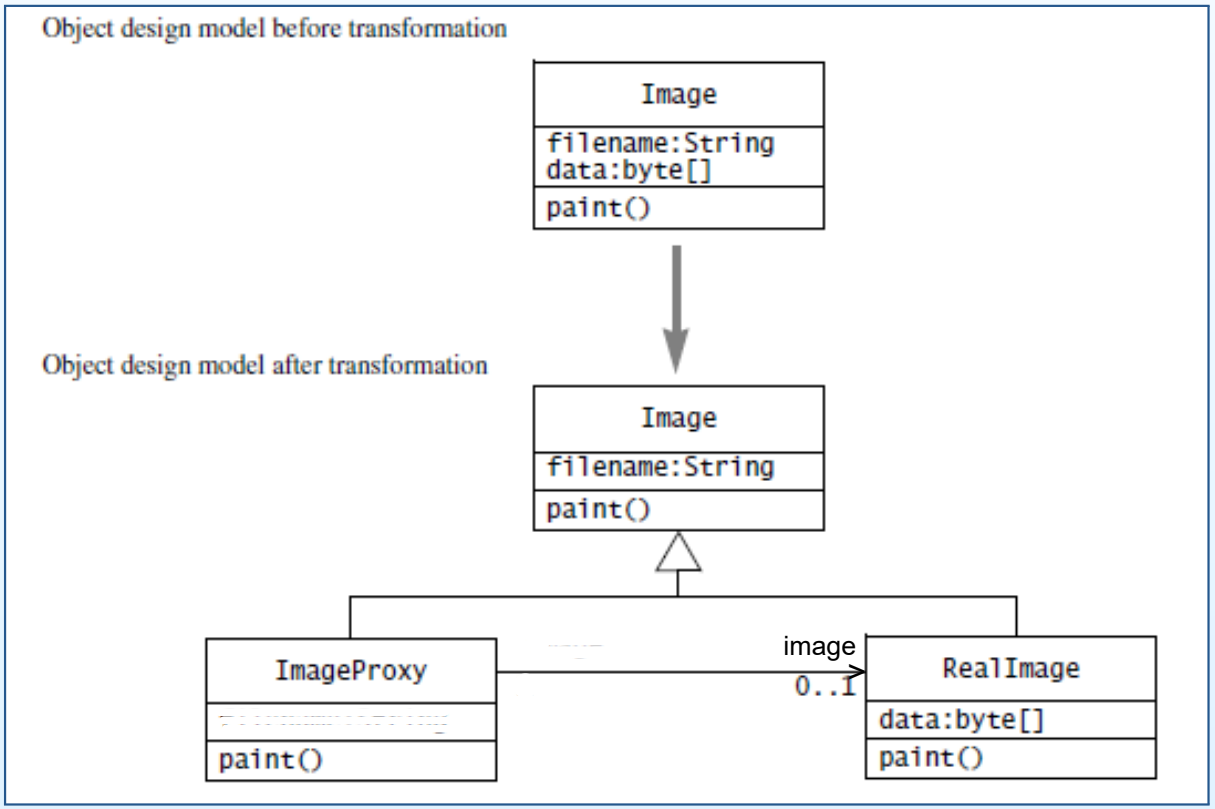
Transformarea unor clase în atribute (cont.)

- Acestei transformări îi corespunde unu caz particular al refactorizării *Inline Class* [Fowler, 2000]

- Paşi refactorizării *Inline Class*
  - Declară toate câmpurile şi metodele publice ale clasei sursă în clasa destinaţie
  - Schimbă toate referinţele spre clasa sursă către clasa destinaţie
  - Schimbă numele clasei sursă, pentru a identifica eventualele referinţe nemodificate
  - Compilează şi testează
  - Şterge clasa sursă

Gestionare operaţiilor costisitoare

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



Gestionare operaţiilor costisitoare (cont.)

- Un obiect de tip *ImageProxy* (imagine surogat) ia locul obiectului *RealImage* (imagine reală), oferind aceeaşi interfaţă cu acesta
- Obiectul surogat răspunde la solicitări simple şi încarcă obiectul real doar în momentul în care i se apelează operaţia `paint()` (mesajul de desenare va fi apoi delegat obiectului imagine real)
- În cazul în care clienţii nu apelează `paint()`, obiectul imagine real nu va fi creat niciodata

- Rezultatele unor operaţii complexe, apelate frecvent, însă bazate pe valori care nu se schimbă sau se schimbă destul de rar, pot fi memorate la nivelul unor atribute private
  - Soluţia optimizează timpul de răspuns la apelul operaţiilor, însă consumă spaţiu de memorie suplimentar pentru stocarea unor informaţii redundante

Reprezentarea asocierilor

- UML: *asocieri* = mulţimi de legături între obiecte
- Limbaje de programare: *referinţe* / *colecţii de referinţe*
- Implementarea asocierilor în cod ţine cont de *navigabilitate*, *multiplicităţi*, *nume de roluri* şi de semantica domeniului
  - Bidirecţionalitatea introduce dependenţe mutuale între clase (se traduce prin perechi de referinţe ce trebuie sincronizate)
  - multiplicitatea *one* necesită o referinţă, cea *many* - o colecţie de referinţe
  - numele de roluri corespund numelor de câmpuri adăugate în clase

Asocieri unidirecţionale *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 bidirecţionale *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 bidirecţionale *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;
   }
```

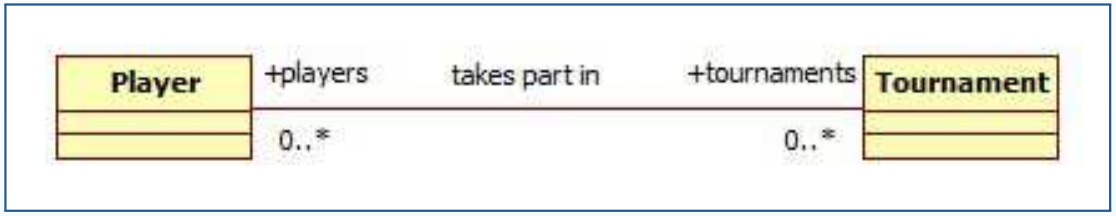
Asocieri bidirecţionale *one-to-many* (cont.)

```
    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 bidirecţionale *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);
        }
    }
```

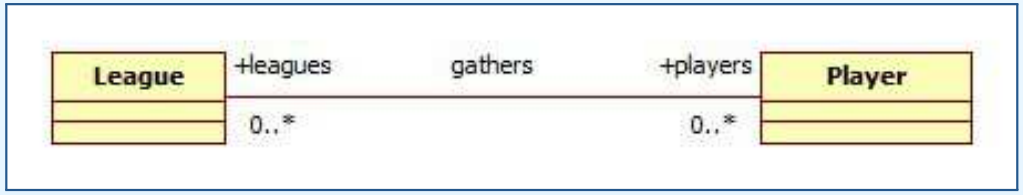
Asocieri bidirecţionale *many-to-many* (cont.)

```
    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);
    }
}
```

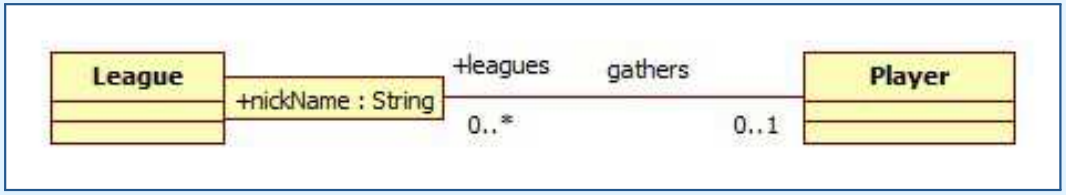
```
    public void removePlayer(Player player)
    {
        //pre: tournament != null
        if(players.contains(player))
        {
          players.remove(player);
          player.internalRemoveTournament(this);
        }
    }

    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(playe
    }

}
```

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 doreşte a fi redus, atribut care are valori unice în contextul asocierii, însă nu neapărat unice la nivel global
  - Ex.: Pentru a putea fi uşor identificaţi în cadrul unei ligi, jucătorii îşi 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);
    }
}
```

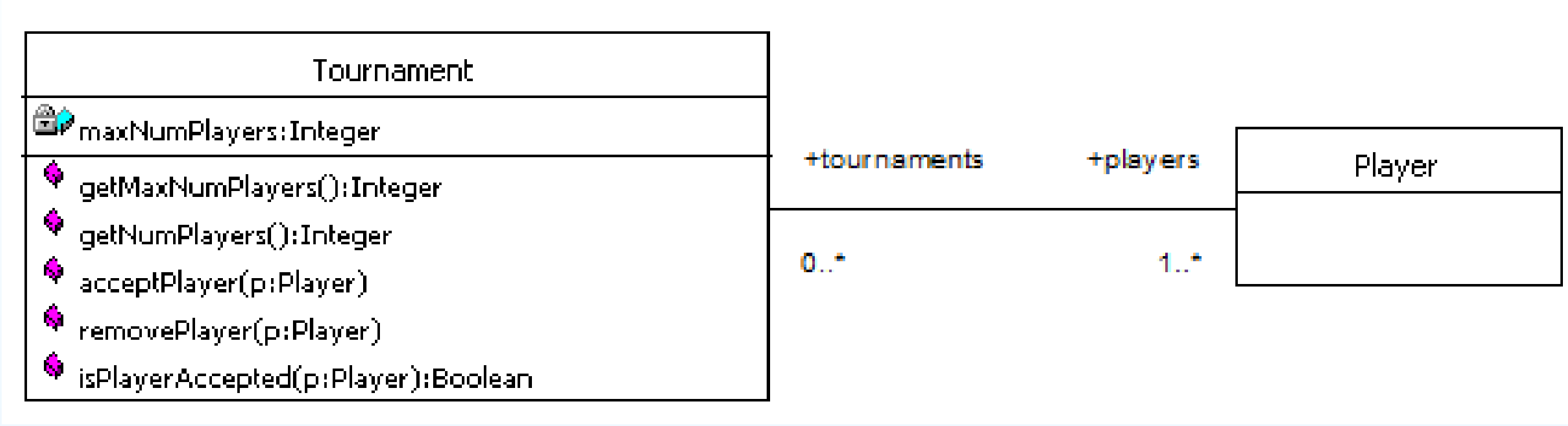
Asocieri calificate (cont.)

```
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 precondiţiilor*
  - Precondiţiile trebuie verificate la începutul fiecărei metode, înaintea efectuării procesărilor caracteristice. În cazul în care precondiţia nu este adevărată, se va arunca o excepţie. Se recomandă ca fiecare precondiţie să corespundă unui tip particular de excepţie.
- *Verificarea postcondiţiilor*
  - Postcondiţiile trebuie verificate la sfârşitul fiecărei metode, după terminarea tuturor procesărilor caracteristice şi finalizarea schimbărilor de stare. În cazul în care contractul este violat, se va arunca o excepţie specifică.
- *Verificarea invarianţilor*
  - Invarianţii se vor verifica odată cu postcondiţiile (la finalizarea fiecărei metode publice a clasei)
- *Moştenirea contractelor*
  - Codul de verificare al aserţiunilor 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.)

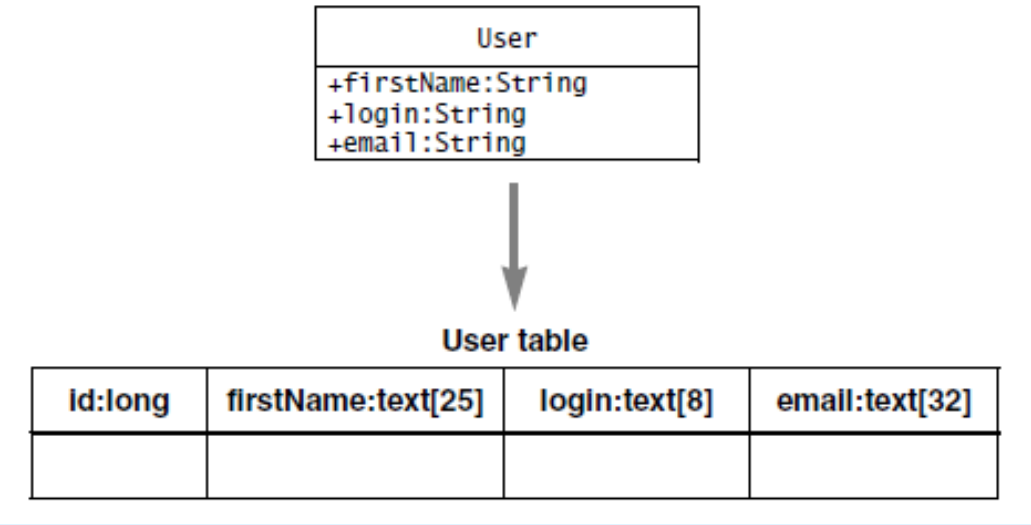
```
        // accomplish the real work
        players.add(p);
        p.addTournament(this);

        // 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 operaţiei în sine
  - *Şanse mari de introducere a unor erori*
  - *Posibilitatea de mascare a unor defecte în codul aferent funcţionalităţii* - în cazul în care cele două sunt scrise de către acelaşi programator
  - *Dificultatea modificării codului în cazul modificării constrângerii*
  - *Probleme de performanţă* la monitorizarea exhaustivă
- *Soluţii*
  - Generarea automată a codului de verificare aferent contractelor folosind instrumente CASE dedicate (ex. OCLE)
  - Monitorizarea selectivă
    - la testare - toate aserţiunile
    - la exploatare - selectiv, funcţie de performanţele dorite, gradul de încredere în calitatea codului şi natura critică a aplicaţiei

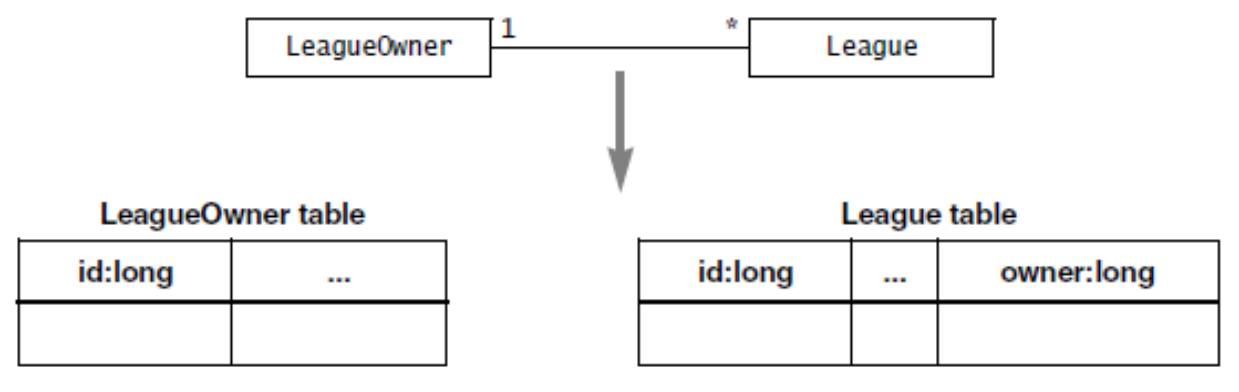
Reprezentarea entităţilor persistente

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

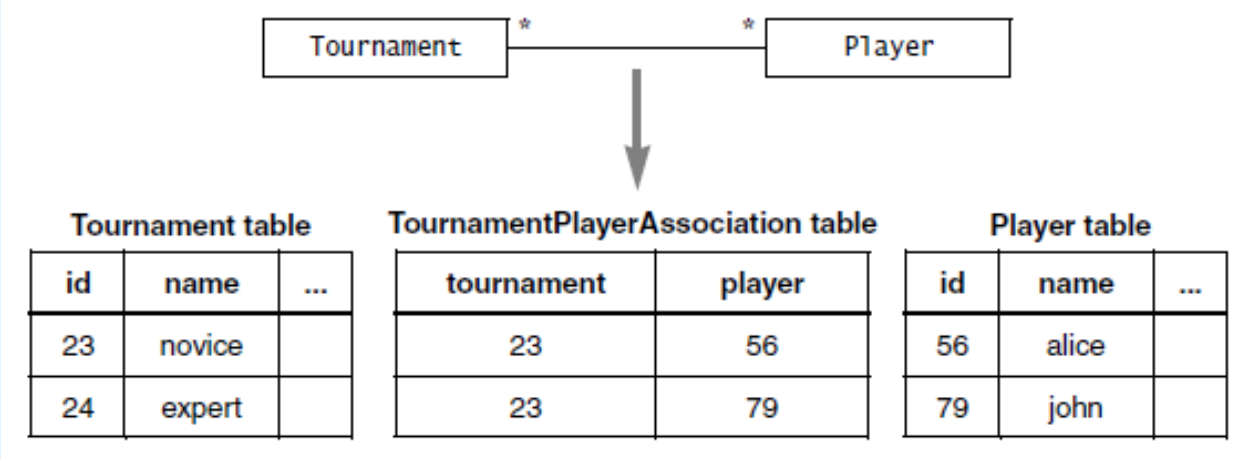


Reprezentarea entităţilor persistente (cont.)

- *Reprezentarea asocierilor*
  - Asocierile *one-to-one* şi *one-to-many* se reprezintă folosind chei străine

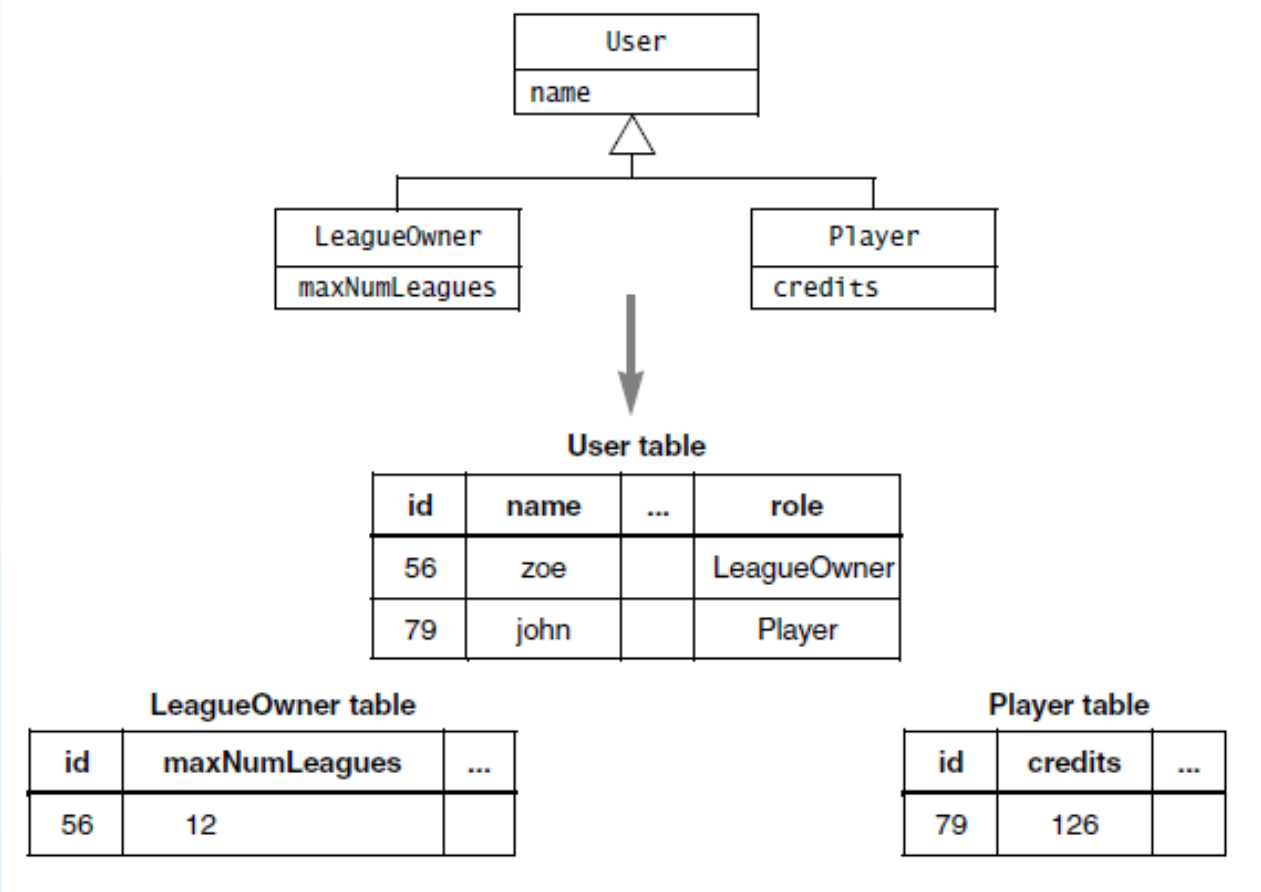


  - Asocierile *many-to-many* se reprezintă folosind tabele de legătură



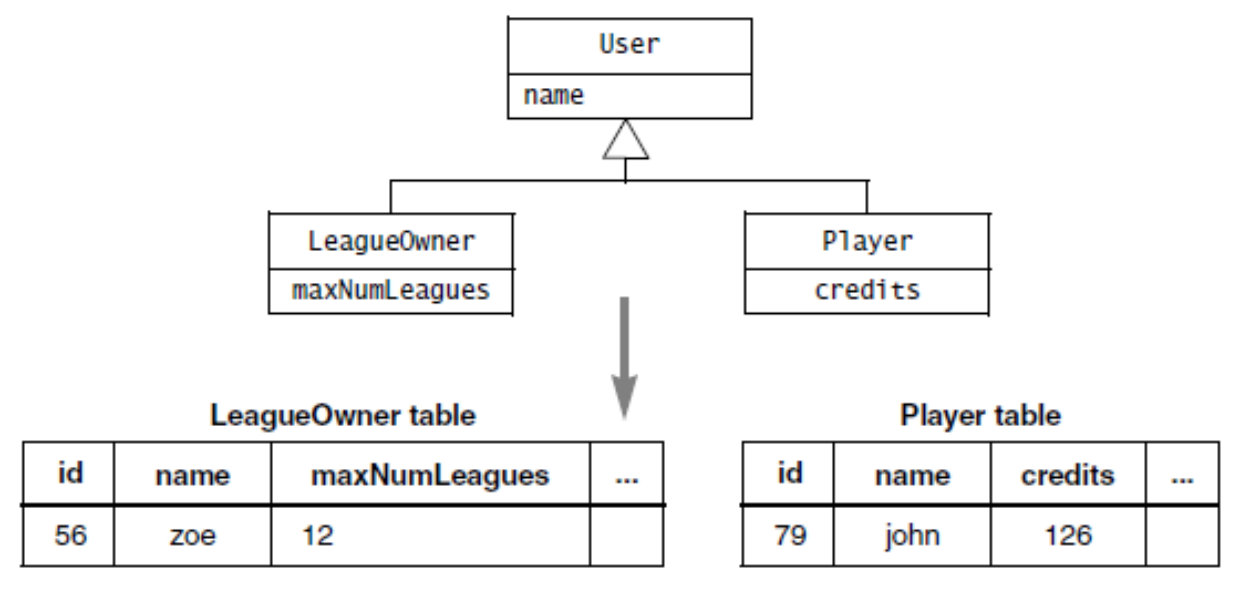
Reprezentarea entităţilor persistente (cont.)

- *Reprezentarea moştenirii*
  - Mapare verticală



Reprezentarea entităţilor persistente (cont.)

- Mapare orizontală



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

Reprezentarea entităţilor 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 atributele proprii + cele moştenite; nefragmentarea obiectelor individuale => interogări rapide

Referinţe

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