

Curs 9

Proiectarea obiectuală: Specificarea interfețelor

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit
"Object-Oriented Software Engineering using UML, Patterns, and Java"*

Specificarea interfețelor

- Scopul proiectării obiectuale este reprezentat de identificarea și rafinarea obiectelor din domeniul soluției necesare realizării comportamentului subsistemelor identificate în etapa proiectării de sistem
- Produse existente până în momentul etapei de specificare a interfețelor
 - *modelul obiectual de analiză*: entități din domeniul problemei (cu atribute, relații, unele operații) + obiecte boundary și control inteligibile utilizatorului
 - *descompunerea sistemului*: subsisteme, servicii oferite, dependențe între subsisteme, obiecte noi din domeniul soluției
 - *mapare hardware/software*: mașina virtuală = componente reutilizate pentru serviciile standard
 - *șabloane de proiectare reutilizate, componente de bibliotecă reutilizate pentru structuri de date și servicii de bază*
- Subactivități în specificarea interfețelor
 - Identificarea atributelor și operațiilor lipsă
 - Specificarea vizibilității și semnăturii operațiilor
 - **Specificare pre/post-condițiilor pentru operații și a invarianților de tip**

Object Constraint Language (OCL)

- OCL [Warmer, 1999] - limbaj formal, utilizat pentru definirea de expresii pe modelele UML
 - introdus inițial ca și limbaj de modelare la IBM, astăzi standard OMG [OMG, 2014]
- Caracteristici OCL
 - *Limbaj complementar* (UML-ului)
 - OCL nu e un limbaj de sine stătător; a apărut din necesitatea de a acoperi problemele de expresivitate ale UML, a cărei natură diagramatică nu permite formularea multora dintre constrângerile caracteristice sistemelor nontriviale
 - Pentru dezvoltatori, specificațiile OCL practice sunt doar cele formulate în contextul tipurilor de date utilizator introduse în modelul UML
 - *Limbaj declarativ* (limbaj de specificare pur, fără efecte secundare)
 - Evaluarea expresiilor OCL nu modifică starea modelului UML asociat
 - *Limbaj puternic tipizat*
 - Fiecare (sub)expresie OCL are un tip și face obiectul verificărilor privind conformance tipurilor

Object Constraint Language (OCL) (cont.)

- *Limbaj bazat pe logica de ordinul întâi*
- *Limbaj care suportă principalele caracteristici OOP*
 - Specificațiile OCL sunt moștenite în descendenți, unde pot fi supradefinite
 - Redefinirea constrângerilor se conformează regulilor DbC
 - Limbajul suporta up/down-casting și verificări de tip
- **Utilitate**
 - *navigarea modelului*
 - interogarea informației din model, prin navigări repetate ale asocierilor, folosind nume de roluri
 - *specificarea aserțiunilor*
 - definirea explicită a pre/post-condițiilor și invarianților, conform principiilor DbC
 - *specificare comportamentală*
 - specificarea comportamentului observatorilor (operațiilor de interogare) din model, specificarea regulilor de derivare pentru attribute/referințele derivate, definirea de noi attribute sau operații
 - *specificarea gărzilor, specificarea invarianților de tip pentru stereotipuri*

Sistemul de tipuri OCL

- OCL fiind complementar UML-ului, orice clasificator dintr-un model UML este un tip OCL valid in cadrul oricărei expresii atașate modelului în cauză
- Biblioteca standard OCL - tipuri predefinite, independente de model
 - Tipuri primitive: Integer, UnlimitedNatural, Real, Boolean, String
 - Tipuri specifice OCL: OclAny, OclVoid, OclInvalid, OclMessage
 - Tipuri colecție: Collection, Set, OrderedSet, Sequence, Bag
 - Enumeration, TupleType
- *Tipuri specifice OCL*
 - Tipul OclAny
 - Supertipul tuturor tipurilor OCL (=> în particular, fiecare clasă din modelul UML moștenește toate operațiile definite în OclAny)
 - Operații
 - = (object2:OclAny):Boolean - egalitatea a două obiecte
 - <> (object2:OclAny):Boolean - egalitatea a două obiecte
 - oclIsTypeOf (type:Classifier):Boolean - conformanța tipurilor

Sistemul de tipuri OCL (cont.)

- Operații

`oclIsKindOf(type:Classifier):Boolean` - conformanța tipurilor

`oclType():Classifier` - inferă tipul

`oclAsType(type:Classifier):T` - conversie

`oclIsNew():Boolean` - utilizat în postcondiția unei operații, verifică dacă obiectul a fost creat în timpul execuției operației respective

`oclIsUndefined():Boolean` - verifică dacă obiectul există/e definit

`oclIsValid():Boolean` - verifică dacă obiectul este valid

- Tipul `OclVoid`

- Tip care se conformează tuturor tipurilor OCL, mai puțin `oclInvalid`

- Denotă absența unei valori (sau o valoare necunoscută la momentul respectiv), singura valoare a tipului e literalul `null`

- Tipul `OclInvalid`

- Tip care se conformează tuturor tipurilor OCL, inclusiv `OclVoid`

- Singura valoare este `invalid`, ce poate rezulta din excepții privind împărțirea la zero, accesarea unei valori de pe un index nepermis, etc.

Sistemul de tipuri OCL (cont.)

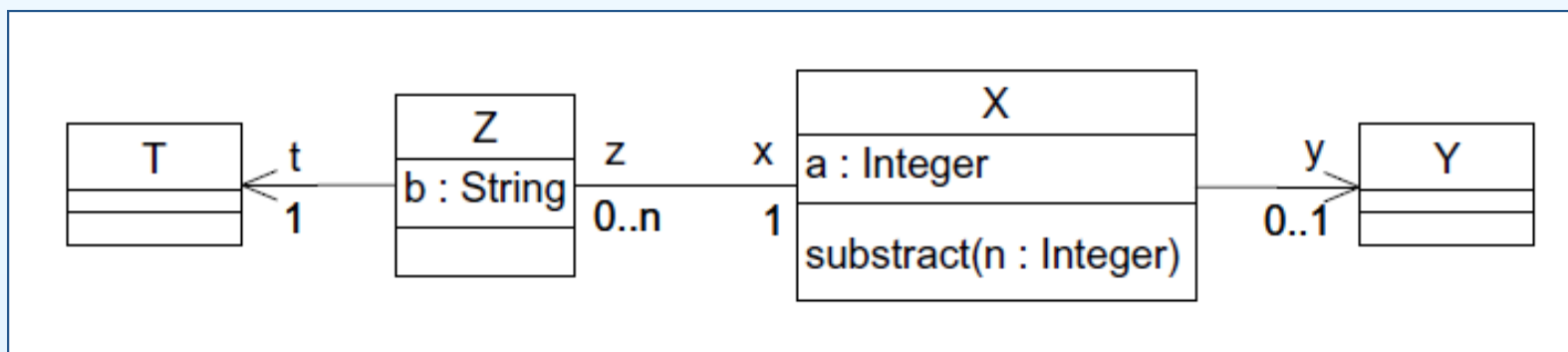
- Tipurile colecție (tipuri template)
 - Operațiile pe colecții sunt apelate folosind notația \rightarrow
 - Modalități de a obține o colecție: prin navigare, ca rezultat al unei operații pe colecții, folosind specificații cu literalii (`Set { }`, `Bag { 1 , 2 , 1 }`, `Sequence { 1 .. 10 }`)
 - Tipul `Collection`
 - Supertipul abstract al celorlalte tipuri colecție din biblioteca standard OCL (`Set`, `OrderedSet`, `Bag`, `Sequence`)
 - Definește operații cu semantică comună tuturor subtipurilor
 - Unele operații sunt redefinite în subtipuri, având o postcondiție mai puternică sau o valoare de retur mai specializată
 - Operații uzuale
`size() : Integer`, `isEmpty() : Boolean`, `notEmpty() : Boolean`
`count(object : T) : Integer`
`includes(object : T) : Boolean`, `excludes(object : T) : Boolean`
`includesAll(c2 : Collection(T)) : Boolean`
`excludesAll(c2 : Collection(T)) : Boolean`
`asSet() : Set(T)`, `asOrderedSet() : OrderedSet(T)`
`asBag() : Bag(T)`, `asSequence() : Sequence(T)`

Sistemul de tipuri OCL (cont.)

- Iteratori pe colecții
 - **select**: `source->select(iterator | body)` - returnează subcolecția colecției `source` pentru care `body` se evaluează la `true`
 - **reject**: `source->reject(iterator | body)` - returnează subcolecția colecției `source` pentru care `body` se evaluează la `false`
 - **forAll**: `source->forAll(iterator | body)` - returnează `true` dacă pentru toate elementele din colecția `source` `body` se evaluează la `true`
 - **exists**: `source->exists(iterator | body)` - returnează `true` dacă există cel puțin un element în colecția `source` pentru care `body` se evaluează la `true`
 - **one**: `source->one(iterator | body)` - returnează `true` dacă există exact un element al colecției `source` pentru care `body` se evaluează la `true`
 - **any**: `source->any(iterator | body)` - returnează un element arbitrar din colecția `source` pentru care `body` se evaluează la `true`
 - **isUnique**: `source->isUnique(iterator | body)` - returnează `true` dacă evaluările lui `body` conduc la elemente distincte
 - **collect**: `source->collect(iterator | body)` - returnează colecția rezultată prin aplicarea lui `body` pe fiecare element al colecției sursă

Proprietăți și navigare

- O expresie OCL este formulată în contextul unui anumit tip
 - În cadrul respectivei expresii, instanța contextuală este referită de cuvântul cheie `self`
 - `self` poate fi omis, atunci când nu există risc de ambiguități
 - Pornind de la instanța contextuală, se pot accesa oricare dintre atributele, operațiile de tip interogare sau capetele opuse de asociere, în stilul orientat-obiect clasic (folosind notația ".")
- Ex.:



- În contextul clasei X, `self.a` și `self.y` sunt două expresii OCL având tipurile `Integer`, respectiv `Y` (prima accesează un atribut, a doua presupune o navigare a unei asocieri folosind numele de rol al capătului opus)

Proprietăți și navigare (cont.)

- Reguli de tipizare la navigare
 - Atunci când multiplicitatea capătului opus de asociere este cel mult 1, tipul expresiei rezultate prin navigare într-un singur pas este dat de clasificatorul de la capătul opus
 - Atunci când valoarea maximă a multiplicității capătului opus de asociere este cel puțin 1, tipul expresiei rezultate prin navigare într-un singur pas este `Set` sau `OrderedSet`, funcție de prezența sau absența constrângerii `{ordered}` pe capătul opus
 - În contextul `x`, tipul expresiei `self.z` este `Set(Z)`
 - Atunci când navigarea presupune mai mulți pași, tipul expresiei rezultat este `Bag`
 - În contextul `x`, tipul expresiei `self.z.t` este `Bag(T)`
- În afară de accesarea proprietăților instanței contextuale, este posibilă utilizarea operației `allInstances` pe un anumit clasificator => mulțimea tuturor obiectelor existente, având acel clasificator ca și tip
 - `x.allInstances()->size()` - numărul obiectelor curente de tip `x`

Design by Contract în OCL

- Constrângeri de tip `invariant`

```
context X
  inv invX1: self.a >= 0
```

- Un invariant se formulează în contextul unui clasificator, ce dă tipul instanței contextuale
- Un invariant este introdus de cuvântul cheie `inv`, urmat de un identificator opțional și de expresia OCL a invariantului

- Constrângeri de tip `precondition/postcondition`

```
context X::subtract (n:Integer)
  pre subtractPre:    self.a >= n
  post subtractPost:  self.a = self.a@pre - n
```

- Clauza `context` menționează semnătura operației aferente (`self` va fi o instanță a tipului care deține acea operație)
- Într-o postcondiție, notația `@pre` referă valoare unui obiect/unei proprietăți înainte de execuția operației în cauză

Structurarea specificațiilor OCL

- Mecanismul `let`

- Permite extragerea unei subexpresii OCL redundante într-o variabilă
- Crește inteligibilitatea constrângerii și eficiența evaluării acesteia (prin efectuarea calculului aferent o singură dată)

```
context X
  inv invX2: let allT:Bag(T) = self.z.t in
              allT->size() = allT->asSet()->size()
```

- Constrângeri de tip `definition`

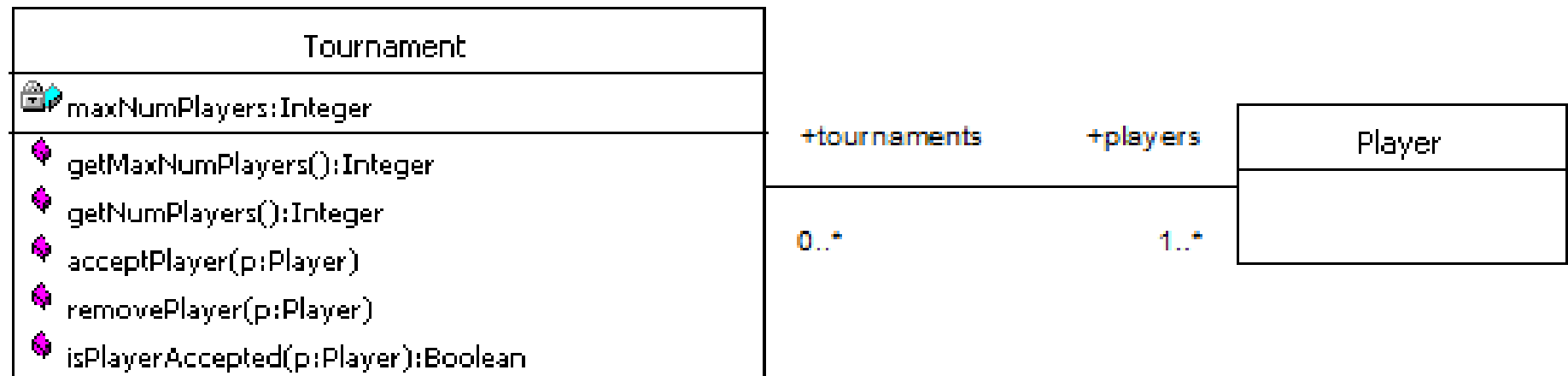
- Permit reutilizarea unei expresii OCL la nivelul mai multor constrângeri
- Introduse prin cuvântul cheie `def`, permit definirea unor attribute/operații auxiliare la nivelul unui clasificator

```
context X
  def: hasY:Boolean = not self.y.oclIsUndefined()
  def: hasZWithBValue(value:String):Boolean =
    self.z->exists(zz | zz.b = value)
```

Iteratori - variante de sintaxă

- **select** (analog reject, forAll, exists)
 - `collection->select(v:Type | boolean-expression-with-v)`
 - `collection->select(v | boolean-expression-with-v)`
 - `collection->select(boolean-expression)`
- **collect**
 - `collection->collect(v:Type | expression-with-v)`
 - `collection->collect(v | expression-with-v)`
 - `collection->collect(expression)`
- **iterate**
 - `collection->iterate(elem:Type; acc:Type = <expression> | expression-with-elem-and-acc)`
 - Cel mai generic iterator, ceilalți pot fi exprimați folosindu-l pe `iterate`
 - **Ex.:** `collection->collect(x:T | x.property)` is equivalent to `collection->iterate(x:T; acc:Bag(T2) = Bag{} | acc->including(x.property))`

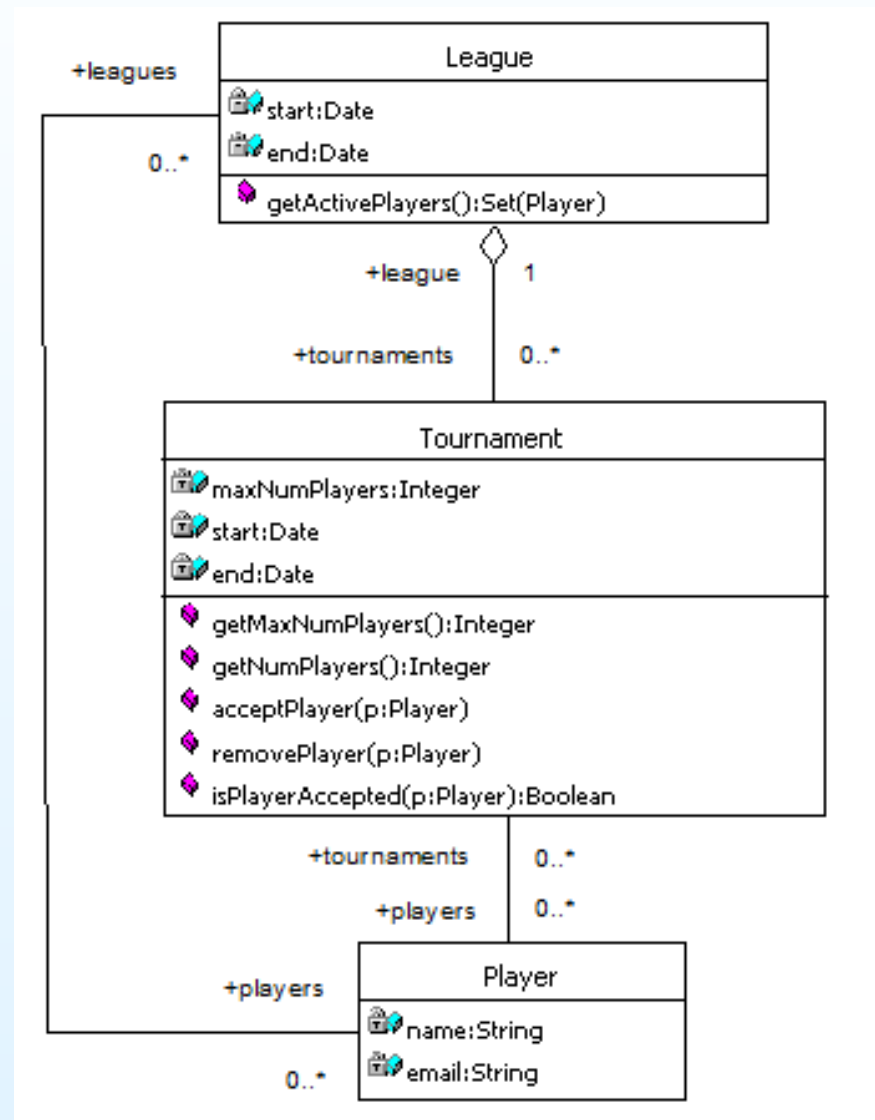
Exemple OCL



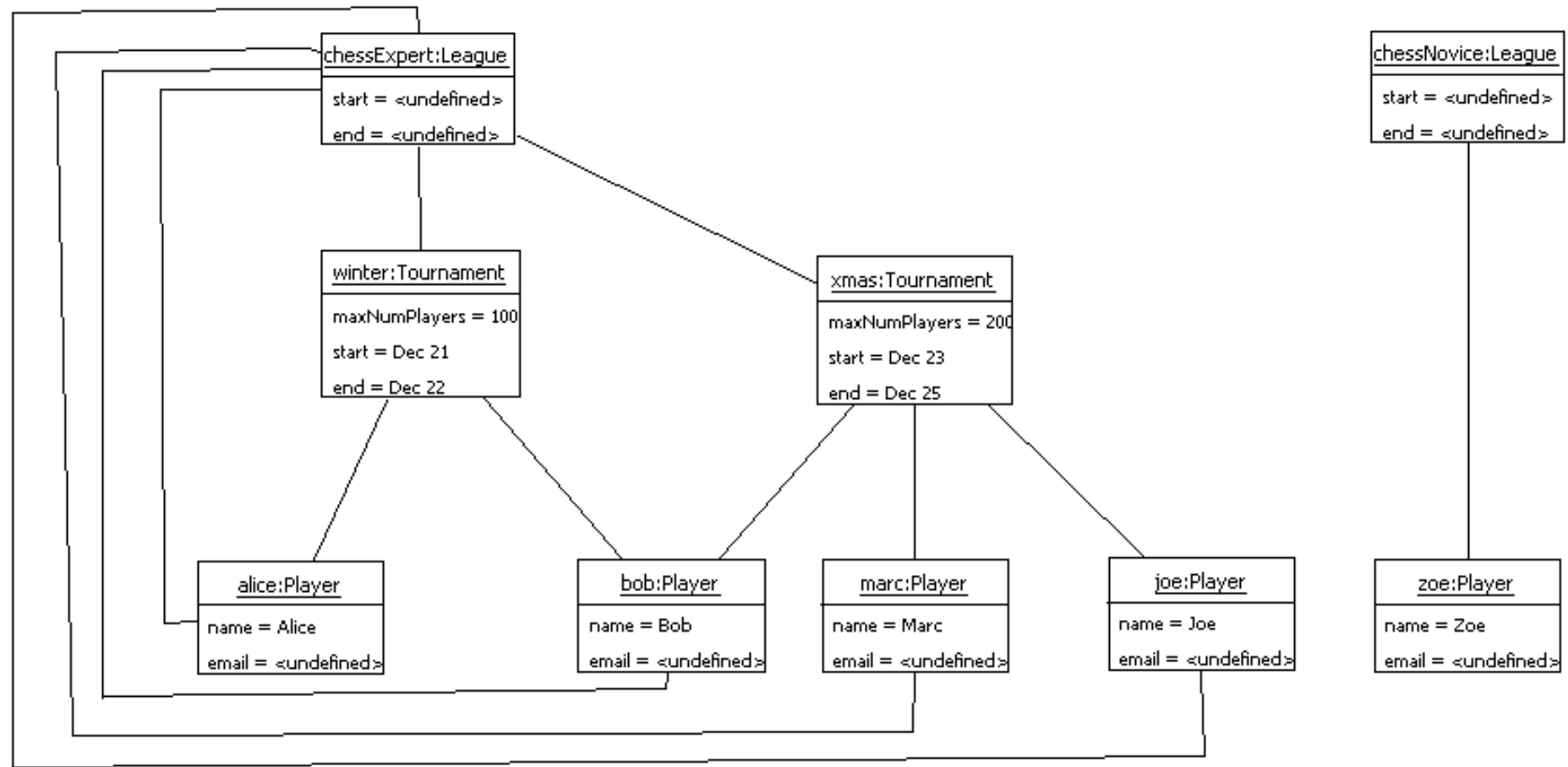
```
context Tournament
  inv maxNumPlayersPositive:
    self.getMaxNumPlayers() > 0

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
```

Exemple OCL (cont.)



Exemple OCL (cont.)



Exemple OCL (cont.)

- Durata unui turneu trebuie să fie sub o săptămână

```
context Tournament
  inv maxDuration: self.end - self.start < 7
```

- Toți jucătorii care participă la un turneu trebuie să fie înregistrați în liga aferentă acestuia

```
context Tournament
  inv allPlayersRegisteredWithLeague:
    self.league.players->includesAll(self.players)
```

```
context Tournament::acceptPlayer(p:Player)
  pre playerIsInLeague: self.league.players->includes(p)
```

Exemple OCL (cont.)

- Jucătorii activi dintr-o ligă sunt cei care au participat la cel puțin un turneu al ligii

```
context League::getActivePlayers() : Set(Player)
    post: result = self.tournaments->asSet()
```

- Toate turneele unei ligi au loc în intervalul de timp aferent ligii

```
context League
    inv: self.tournaments->forall(t:Tournament |
        t.start.after(self.start) and t.end.before(self.end))
```

- În orice ligă există cel puțin un turneu planificat în prima zi a ligii

```
context League
    inv: self.tournaments->exists(t:Tournament |
        t.start = self.start)
```

Moștenirea contractelor

- Problema moștenirii contractelor
 - În limbajele polimorfe, o referință la un obiect al clasei de bază poate fi substituită de o referință la un obiect al unei clase derivate
 - Codul client, scris în termenii clasei de bază, poate folosi obiecte ale claselor derivate, fără a avea cunoștință de acest fapt
 - => Clientul se așteaptă ca un contract formulat relativ la clasa de bază, să fie respectat și de clasele derivate
- Reguli privind moștenirea contractelor (consecință a principiului Liskov al substituției)
 - *Precondiții*: Unei metode dintr-o subclasă îi este permis să slăbească precondiția metodei pe care o supradefinește (o metodă care supradefinește poate gestiona mai multe cazuri decât cea supradefinită)
 - *Postcondiții*: O metodă care supradefinește trebuie să asigure o postcondiție cel puțin la fel de puternică precum cea supradefinită
 - *Invarianți*: O subclasă trebuie să respecte toți invarianții superclaselor sale; poate, eventual, introduce invarianți mai puternici decât cei moșteniți

Referințe

- [OMG, 2014] Object Management Group, *Object Constraint Language - version 2.4*, February 2014.
- [Warmer, 1999] J. Warmer, A. Kleppe, *Object constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.