Curs 9

Proiectarea obiectuală: Specificarea interfeţelor

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

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

Specificarea interfeţelor

- Scopul proiectării obiectuale este reprezentat de identificarea şi rafinarea obiectelor din domeniul soluţiei necesare realizării comportamentului subsistemelor identificate în etapa proiectării de sistem
- Produse existente până în momentul etapei de specificare a interfeţelor
  - *modelul obiectual de analiză*: entităţi din domeniul problemei (cu atribute, relaţii, unele operaţii) + obiecte boundary şi control inteligibile utilizatorului
  - *descompunerea sistemului*: subsisteme, servicii oferite, dependenţe între subsisteme, obiecte noi din domeniul soluţiei
  - *mapare hardware/software*: maşina virtuală = componente reutilizate pentru serviciile standard
  - *şabloane de proiectare reutilizate*, *componente de bibliotecă reutilizate pentru structuri de date şi servicii de bază*
- Subactivităţi în specificarea interfeţelor
  - Identificarea atributelor şi operaţiilor lipsă
  - Specificarea vizibilităţii şi signaturii operaţiilor
  - **Specificare pre/post-condiţiilor pentru operaţii şi a invarianţilor de tip**

Object Constraint Language (OCL)

- OCL [Warmer, 1999] - limbaj formal, utilizat pentru definirea de expresii pe modelele UML
  - introdus iniţial ca şi 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 cărui natură diagramatică nu permite formularea multora dintre constrângerile caracteristice sistemelor nontriviale
    - Pentru dezvoltatori, specificaţiile 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 şi face obiectul verificărilor privind conformanţa tipurilor

Object Constraint Language (OCL) (cont.)

- *Limbaj bazat pe logica de ordinul întâi*
- *Limbaj care suportă principalele caracteristici OOP*
  - Specificaţiile OCL sunt moştenite în descendenţi, unde pot fi supradefinite
  - Redefinirea constrângerilor se conformează regulilor DbC
  - Limbajul suporta up/down-casting şi verificări de tip

- Utilitate
  - *navigarea modelului*
    - interogarea informaţiei din model, prin navigări repetate ale asocierilor, folosind nume de roluri
  - *specificarea aserţiunilor*
    - definirea explicită a pre/post-condiţiilor şi invarianţilor, conform principiilor DbC
  - *specificare comportamentală*
    - specificarea comportamentului observatorilor (operaţiilor de interogare) din model, specificarea regulior de derivare pentru atributele/referinţele derivate, definirea de noi atribute sau operaţii
  - *specificarea gărzilor*, *specificarea invarianţilor 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 ataşate 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 colecţie: `Collection`, `Set`, `OrderedSet`, `Sequence`, `Bag`
  - `Enumeration`, `TupleType`
- *Tipuri specifice OCL*
  - Tipul `OclAny`
    - Supertipul tuturor tipurilor OCL (=> în particular, fiecare clasă din modelul UML moşteneşte toate operaţiile definite în `OclAny`)
    - Operaţii
      `=(object2:OclAny):Boolean` - egalitatea a două obiecte
      `<>(object2:OclAny):Boolean` - egalitatea a două obiecte
      `oclIsTypeOf(type:Classifier):Boolean` - conformanţa tipurilor

Sistemul de tipuri OCL (cont.)

- Operaţii
  `oclIsKindOf(type:Classifier):Boolean` - conformanţa tipurilor
  `oclType():Classifier` - inferă tipul
  `oclAsType(type:Classifier):T` - conversie
  `oclIsNew():Boolean` - utilizat în postcondiţia unei operaţii, verifică dacă obiectul a fost creat în timpul execuţiei operaţiei 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 puţin `oclInvalid`
  - Denotă absenţa 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 excepţii privind împărţirea la zero, accesarea unei valori de pe un index nepermis, etc.

Sistemul de tipuri OCL (cont.)

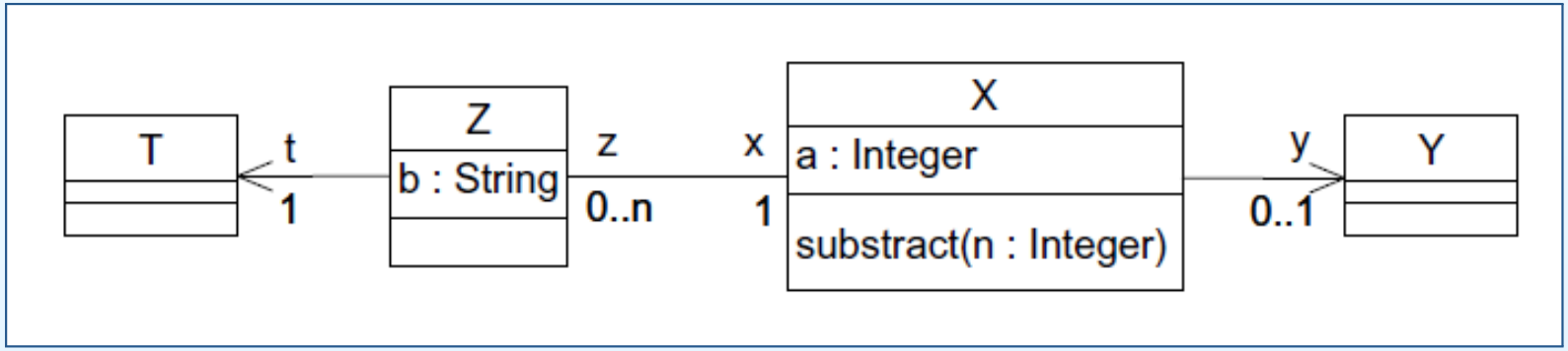
- Tpurile colecţie (tipuri template)
  - Operaţiile pe colecţii sunt apelate folosind notaţia $\rightarrow$
  - Modalităţi de a obţine o colecţie: prin navigare, ca rezultat al unei operaţii pe colecţii, folosind specificaţii cu literali (`Set{}`, `Bag{1,2,1}`, `Sequence{1..10}`)
  - Tipul `Collection`
    - Supertipul abstract al celorlalte tipuri colecţie din biblioteca standard OCL (Set, OrderedSet, Bag, Sequence)
    - Defineşte operaţii cu semantică comună tuturor subtipurilor
    - Unele operaţii sunt redefinite în subtipuri, având o postcondiţie mai puternică sau o valoare de retur mai specializată
    - Operaţii 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 colecţii
  - **select**: `source->select(iterator | body)` - returnează subcolecţia colecţiei `source` pentru care `body` se evaluează la `true`
  - **reject**: `source->reject(iterator | body)` - returnează subcolecţia colecţiei `source` pentru care `body` se evaluează la `false`
  - **forAll**: `source->forAll(iterator | body)` - returnează `true` dacă pentru toate elementele din colecţia `source` `body` se evaluează la `true`
  - **exists**: `source->exists(iterator | body)` - returnează `true` dacă există cel puţin un element în colecţia `source` pentru care `body` se evaluează la `true`
  - **one**: `source->one(iterator | body)` - returnează `true` dacă există exact un element al colecţiei `source` pentru cate `body` se evaluează la `true`
  - **any**: `source->any(iterator | body)` - returnează un element arbitrar din colecţia `source` pentru care `body` se evaluează la `true`
  - **isUnique**: `source->isUnique(iterator | body)` - returnează `true` dacă evaluările lui `body` conduc la elemente distincte
  - **colect**: `source->collect(iterator | body)` - returnează colecţia rezultată prin aplicarea lui `body` pe fiecare element al colecţiei sursă

Proprietăţi şi navigare

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



  - În contextul clasei `X`, `self.a` şi `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)

Proprietăţi şi 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 multiplicităţii capătului opus de asociere este cel puţin 1, tipul expresiei rezultate prin navigare într-un singur pas este `Set` sau `OrderedSet`, funcţie de prezenţa sau absenţa constrângeii `{ordered}` pe capătul opus
    - În contextul `X`, tipul expresiei `self.z` este `Set(Z)`
  - Atunci când navigarea presupune mai mulţi paşi, tipul expresiei rezultat este `Bag`
    - În contextul `X`, tipul expresiei `self.z.t` este `Bag(T)`
- În afară de accesarea proprietăţilor instanţei contextuale, este posibilă utilizarea operaţiei `allInstances` pe un anumit clasificator => mulţimea tuturor obiectelor existente, având acel clasificator ca şi 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 instanţei contextuale
  - Un invariant este introdus de cuvântul cheie `inv`, urmat de un identificator opţional şi de expresia OCL a invariantului

- Constrângeri de tip `precondition/postcondition`

```
context X::substract(n:Integer)
 pre substractPre:   self.a >= n
 post substractPost: self.a = self.a@pre - n
```

  - Clauza `context` menţionează signatura operaţiei aferente (`self` va fi o instanţă a tipului care deţine acea operaţie)
  - Într-o postcondiţie, notatia `@pre` referă valoare unui obiect/unei proprietăţi înainte de execuţia operaţiei în cauză

Structurarea specificaţiilor OCL

- Mecanismul `let`
  - Permite extragerea unei subexpresii OCL redundante într-o variabilă
  - Creşte inteligibilitatea constrângerii şi eficienţa 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()
```

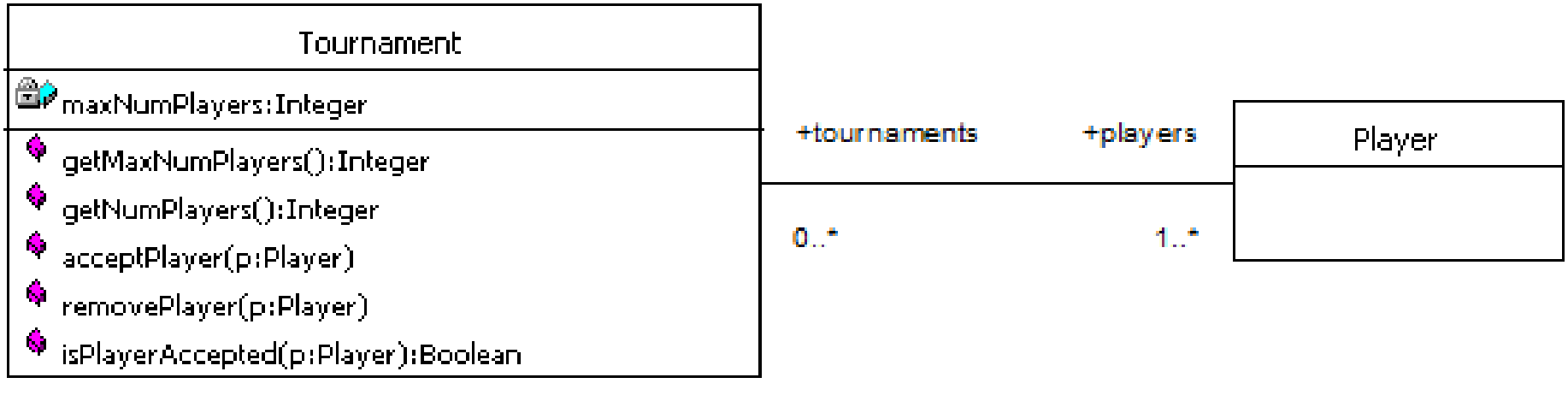
- Constrângeri de tip `definition`
  - Permit reutilizarea unei expresii OCL la nivelul mai multor constrângeri
  - Introduse prin cuvântul cheie `def`, permit definirea unor atribute/operaţii auxiliare la nivelul unui clasificator

```
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, ceilalţi pot fi exprimaţi 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))`

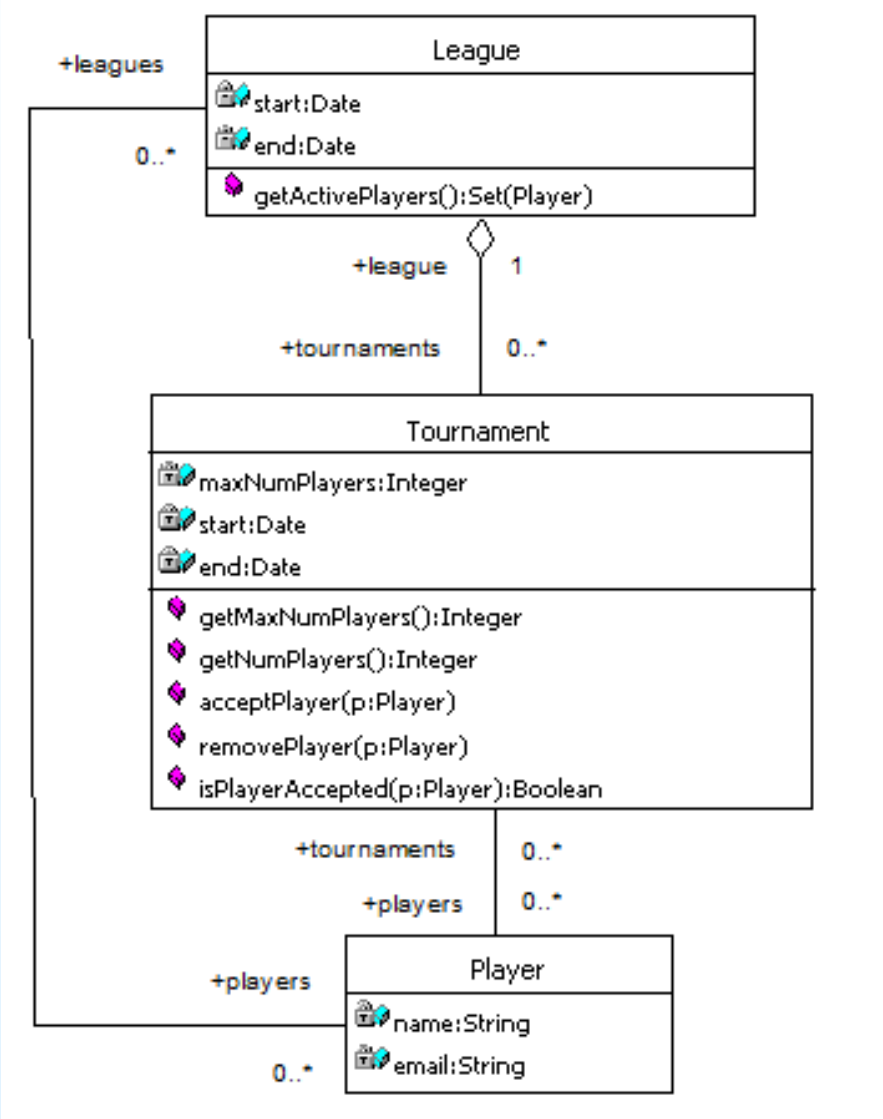
Exemple OCL



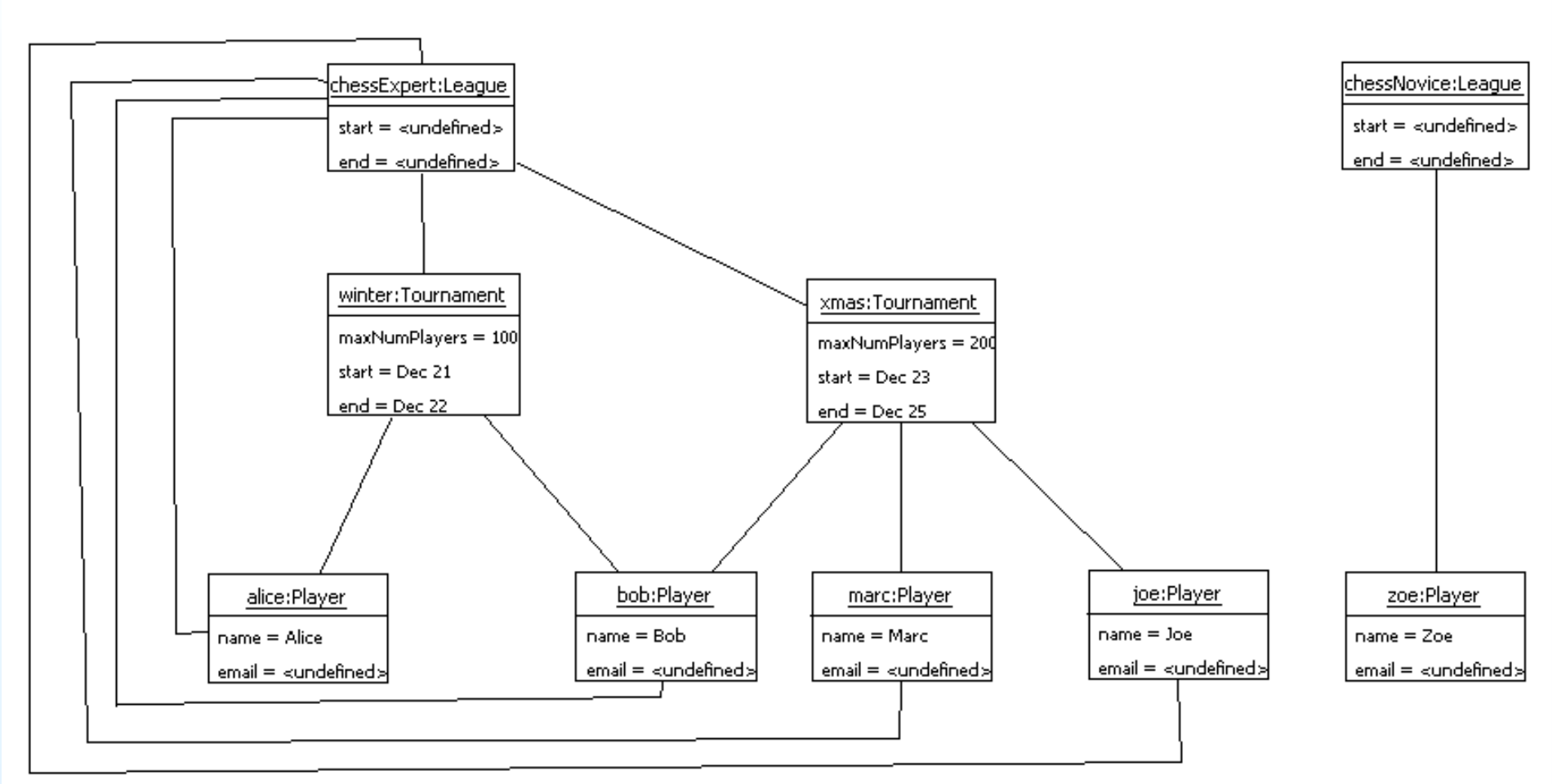

```
context Tournament
    inv maxNumPlayersPositive:
        self.getMaxNumPlayers() > 0

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
```

- Toţi jucătorii care participă la un turneu trebuie să fie înregistraţi 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 puţin un turneu al ligii

```
context League::getActivePlayers():Set(Player)
    post: result = self.tournaments.players->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 puţin un turneu planificat în prima zi a ligii

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

Moştenirea contractelor

- Problema moştenirii contractelor
  - În limbajele polimorfice, o referinţă la un obiect al clasei de bază poate fi substituită de o referinţă la un obiect al unei clase derivate
  - Codul client, scris în termenii clasei de bază, poate folosi obiecte ale claselor derivate, fără a avea cunoştinţă de acest fapt
  - => Clientul se aşteaptă ca un contract formulat relativ la clasa de bază, să fie respectat şi de clasele derivate
- Reguli privind moştenirea contractelor (consecinţă a principiului Liskov al substituţiei)
  - *Precondiţii*: Unei metode dintr-o subclasă îi este permis să slăbească precondiţia metodei pe care o supradefineşte (o metodă care supradefineşte poate gestiona mai multe cazuri decât cea supradefinită)
  - *Postcondiţii*: O metodă care supradefineşte trebuie să asigure o postcondiţie cel puţin la fel de puternică precum cea supradefinită
  - *Invarianţi*: O subclasă trebuie să respecte toţi invarianţii superclaselor sale; poate, eventual, introduce invarianţi mai puternici decăt cei moşteniţi

Referinţe

- [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.

Design by Contract (DbC) [1,2]

- The Design by Contract (DbC) methodology has entered software development due to Bertrand Meyer, along with the Eiffel language
- It proposes a contractual approach to the development of object-oriented software components, based on the use of *assertions*
- The approach has been aimed at increasing the *reliability* of object-oriented software components - a critical requirement in the context of large-scale software reuse, as promoted by the object-oriented paradigm
  - *reliability* = *correctness* (software's ability to behave according to the specification) + *robustness* (the ability to properly handle situations outside of the specification)
- Expected to positively contribute to
  - the develoment of correct and robust OO systems
  - a deeper understanding of inheritance and related concepts (overriding, polymorphism, dynamic bynding), by means of the *subcontracting* concept
  - a systematic approach to *exception handling*

Software Contracts. Assertions

- A generic algorithm to solve a non-trivial task

```
Algorithm mainTask is:
  @subTask_1;
  @subTask_2;
  ...
  @subTask_n;
End-mainTask
```

- Each subtask may be either inlined or triggering the call of a subroutine
- Analogy: calling a subroutine to solve a subtask vs. a real-life situation with a person (client) requiring the services of a third-party (provider) to accomplish a task that he cannot / would not do personally
  - e.g. contracting the services of a fast courier to deliver a package to a particular destination in a foreign city

Software Contracts. Assertions

- Characteristics of human contracts involving two parts
  - stipulate the benefits and obligations of each part; a benefit for one part is an obligation for the other
  - may also reference general laws that should be obeyed by both parts
- *Same principles should apply to software development: each time a routine depends on the call of a subroutine to accomplish a subtask, there should be a contractual specification among the client (caller) and supplier (calee)*
- The clauses of software contracts are formalized by means of *assertions*
  - *assertion = expression involving a number of software entities, which state a property that these entities should fulfill at runtime*
  - *closest concept – predicate, implementation – boolean expressions*
  - *some apply to individual routines (pre/post-conditions), other to the class as a whole (invariants)*

Pre/post-condition assertions

- A <pre, post> pair for a method expresses the software contract that the method in question (provider of some service) publishes for its callers (clients of the service)
- Characteristics of software contracts
  - the precondition defines the situation when a call is legitimate - obligation for the client (caller) and benefit for the provider (method)
  - the postcondition states which properties should be fulfilled once the execution has ended - obligation for the provider (method) and benefit for the client (caller)
- The major contribution brought by DBC in the field of software reliability: precondition = benefit of the service provider
- *DBC non-redundancy principle: The body of a mehod should never check for a precondition* (as opposed to defensive programming)
- In Eiffel, assertions are part of the language, allowing for runtime monitoring
  - precondition violation = bug in the client, postcondition violation = bug in the supplier

Invariant assertions

- In addition to pre/post-conditions, that capture the behavior of individual methods, it is possible to express global properties of a class' instances, that should be preserved by all its public methods
- An *invariant* encloses all the semantic constraints and integrity rules applying to the class in question
- Lifecycle of an object

- An assertion *I* is a correct invariant for a class *C* if and only if the following conditions hold
  - each constructor of *C*, apply to arguments that fulfill its precondition, in a state in which the class attributes have default values, leads to a state in which *I is fulfilled*
  - each public method of *C*, applied to a set of arguments and to a state satisfying both *I* and the method precondition, leads to a state satisfying *I*

Correctness of a class

- Informally, a class is said to be correct with respect to its specification if and only if its implementation, as given by the method bodies, is consistent with its preconditions, postconditions and invariant
- **Definition:** The class *C* is said to be correct with respect to its assertions (pre/post-conditions and invariant) if and only if the following conditions hold:

  (1) [*default_C* and *pre_p(x_p)*] *body_p* [*post_p(x_p)* and *INV*]

  for each class constructor *p* and each set of valid arguments of *p – x_p* and

  (2) [*pre_r(x_r)* and *INV*] *body_r* [*post_r(x_r)* and *INV*]

  for each public class method r and each set of valid arguments of *r – x_r*, where

  default_C is an assertion stating that the attributes of *C* have default values for their type, *INV* is the invariant of *C*, *pre_m*, *post_m*, *body_m* are the precondition, postcondition and body of an arbitrary method *m* of *C*.

The purpose of using assertions

- Support in writting correct software, including the means to formally define correctness
  - The writing of explicit contracts comes as a prerequisite of their enforcement in software
- Support for a better software documentation
  - Essential when it comes to reusable assets, see the case of Ariane!
- Support for testing, debugging and quality assurance
  - Levels of runtime assertion monitoring:
    - 1.preconditions only
    - 2.preconditions and postconditions
    - 3.all assertions
  - While testing, enforce level 3, in production, there is a tradeoff between trust in the code, efficiency level desired and critical nature of the application
- Support for the development of fault tolerant systems

Defensive Programming [3]

- Analogy to *defensive driving*
  - *Defensive driving*: You can never be sure what the others might do, so take responsibility of protecting yourself, such that another driver's mistake won't hurt you!
  - *Defensive programming*: If a routine is passed bad data, it should not be hurt, even if the bad data is someone else's fault (humans, software).
- The core idea of defensive programming is guarding against unexpected errors
- Acknowledges that errors happen and invites programmers to write code accordingly
- Comprises a set of techniques that make errors easier to detect, easier to repair and less damaging in production code
- Should serve as a complement to defect-prevention techniques (iterative design, pseudocode first, design inspections, etc.)
- Protecting from invalid input involves
  - Checking all data received from the outside (from users, files, network, etc.)
    - numeric values should be between tolerances, strings – short enough to handle and obeying to their intended semantics
  - Checking all input parameters
  - Deciding on how to deal with bad data

References

[1] Meyer, B., *Object-Oriented Software Construction (2nd ed.)*, Prentice-Hall, 1997. (Chapter 11 – Design by Contract: building reliable software)

[2] Meyer, B., *Applying „Design by Contract”*, IEEE Computer 25(10):40-51, 1992.

[3] McConnel, S., *Code Complete (2nd ed.)*, Microsoft Press, 2004. (Chapter 8 – Defensive Programming)