

Prednáška 5: Jazyk OCL

Metódy a prostriedky špecifikácie 2013/14

Valentino Vranić

Ústav informatiky a softvérového inžinierstva
Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

22. október 2013

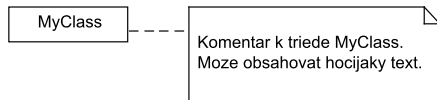
Obsah prednášky

- 1 Ohraničenia v UML
- 2 Základy jazyka OCL
- 3 Invarianty, predpoklady a dôsledky v jazyku OCL

Ohraničenia v UML

Poznámky v UML modeloch

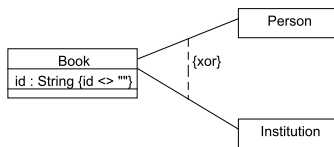
- UML má prvok na grafické vyjadrenie poznámky – note
- Poznámka sa prerušovanou čiarou priradí k prvku, na ktorý sa vzťahuje



- Poznámky možno využiť na vyjadrenie ohraničení – formálne alebo neformálne

Ohraničenia k prvkom

- Ohraničenia sa v UML uvádzajú v množinových zátvorkách
- Ohraničenie možno priradiť k atribútu
- Ohraničenia možno využiť na vyjadrenie ohraničení medzi vzťahmi



- Ohraničenie xor je jedným z preddefinovaných ohraničení v UML
- Na presnejšie vyjadrenie ohraničení sa používa jazyk OCL

Základy jazyka OCL

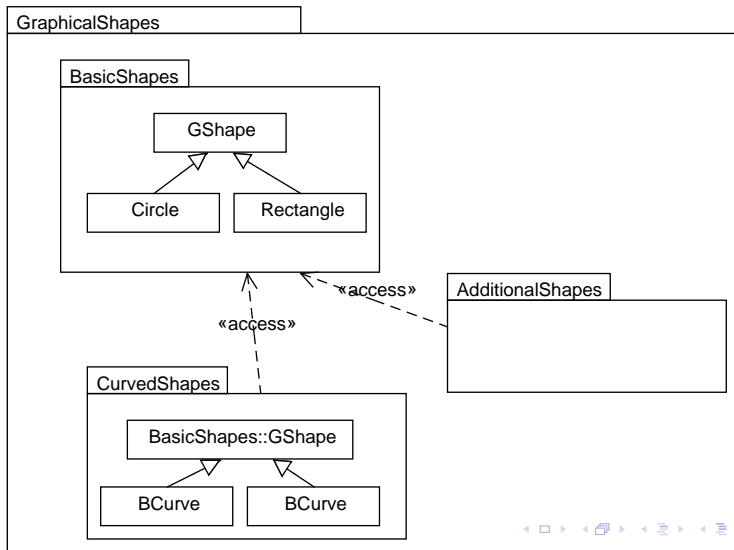
OCL

- Object Constraint Language
- Jazyk na vyjadrenie ohraničení medzi objektmi
- Textový spôsob vyjadrenia
- Deklaratívny jazyk
- Dopĺňa UML vo veciach, ktoré sa nedajú vyjadriť graficky
- Ohraničenia v OCL sa vyjadrujú výrazmi v kontexte určitého prvku modelu v UML

Kontext výrazu v OCL (1)

- Kontext závisí od priradenia OCL výrazu
- Vždy je to však inštancia klasifikátora, ku ktorému alebo ku ktorého prvku je OCL výraz priradený
- Ku kontextu sa pristupuje prostredníctvom priradeného identifikátora alebo kľúčovým slovom `self` (analógia `this` v Jave a C++)

Kontext výrazu v OCL (2)

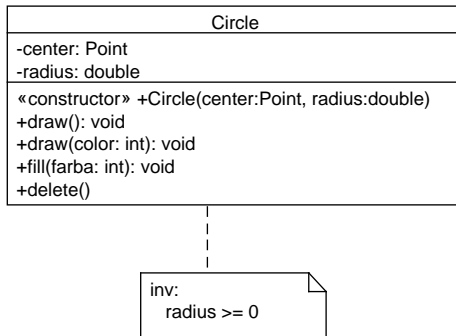


Kontext výrazu v OCL (3)

```
package GraphicalShapes::BasicShapes
  context circle : Circle
    inv: radius >= 0
endpackage
```

Implicitný kontext výrazu v OCL

- Pri priradení poznámkou kontext je implicitný (ako v tele nestatickej metódy v Java)



Typy OCL výrazov

- Špecifikácia ohraničení (invariantov, predpokladov a dôsledkov): `inv`, `pre` a `post`
- Špecifikácia definícií (atribútov, tiel operácií a lokálnych premenných): `init`, `body`, `def`, `let` a `derive`

Syntax OCL výrazov

- Komentár: `--` alebo `/* ... */`
- Kľúčové slová: `and`, `attr`, `context`, `def`, `else`, `endif`, `endpackage`, `if`, `implies`, `in`, `inv`, `let`, `not`, `oper`, `or`, `package`, `post`, `pre`, `then`, `xor`, `body`, `init`, `derive`
- Definovaná je implicitná priorita operácií, ale lepšie je používať zátvorky

System typov v OCL

- Silne typový jazyk
- Primitívne typy: Boolean, Integer, Real, String
- Štrukturovaný typ: Tuple
- Vstavane typy:
 - OclAny – nadtyp všetkých typov v OCL a pridruženom UML modeli
 - OclType – všetky typy v pridruženom UML modeli
 - OclState – všetky stavy v pridruženom UML modeli
 - OclVoid – void (má jedinu inštanciu: OclUndefined)
 - OclMessage – správa
- Typy z pridruženého UML modelu sa tiež stávajú OCL typmi

Niektoré operácie nad typom OclAny (1)

- V nasledujúcich výrazoch `a` a `b` sú referencie objektov
- Porovnanie:
 - porovnanie inštancií na rovnosť
`a = b`
 - porovnanie inštancií na nerovnosť
`a <> b`
 - porovnanie typov objektov
`a.ocllsTypeOf(b : OclType) : Boolean`
 - porovnanie typov objektov, ale vrátane podtypov
`a.ocllsKindOf(b : OclType) : Boolean`
 - porovnanie stavov objektov
`a.ocllnState(b : OclType) : Boolean`
 - či je objekt nedefinovaného typu
`a.ocllsUndefined() : Boolean`

Niektoré operácie nad typom OclAny (2)

- Dopyty:
 - množina všetkých inštancií typu A
`A::allInstances() : Set(A)`
 - či objekt a vznikol ako výsledok danej operácie
`a.ocllsNew() : Boolean`
- Konverzia typov (casting):
 - a bude pretypované na SubType
`a.oclAsType(SubType) : SubType`
- Ostatné typy dedia tieto operácie od OclAny

Operácie nad primitívnymi typmi

- Boolean, Integer, Real a String
- Obvyklé infixové operátory – ale niektoré realizované ako operácie
- Ak predstavujú logické operácie, vracajú logickú hodnotu (true/false)
- Napr. pre typ Boolean: `a.and(b)`, `a.or(b)`, `a.implies(b)`...
- Integer a Real: `a.mod(b)`, `a.max(b)`, `a.round(b)`...
- String: `a.size()`, `a.concat(b)`, `a.toInteger()`...

Kolekcie

- Set, OrderedSet, Bag a Sequence
- Sú to šablóny
- Inštancia pre požadovaný typ: Set(Circle)
- Veľa operácií nad kolekciami – vyvolávajú sa pomocou operátora ->:
 - `aCollection->collectionOperation(parameters)`
- Operácie konverzie, porovnania, dopytu, prístupu a výberu

Iterácie nad kolekciami (1)

```
aCollection->iteratorOperation(iteratorVariable : Type |  
    iteratorExpression)
```

- Boolovské iteračné operácie:
exists, forall, isUnique, one
- Výberové iteračné operácie:
any, collect, collectNested, select, reject, sortBy
- Všeobecná iterácia:

```
aCollection->iterate(iteratorVariable : Type;  
    result : ResultType = initializationExpression |  
    iteratorExpression)
```

Iterácie nad kolekciami (2)

- Príklad všeobecnej iterácie:

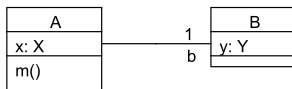
```
Bag{1, 2, 3, 4, 5}->iterate(number : Integer;  
    sum : Integer = 0 | sum = sum + number)
```

- To je ekvivalentné použitiu operácie sum():

```
Bag{1, 2, 3, 4, 5}->sum()
```

Navigácia

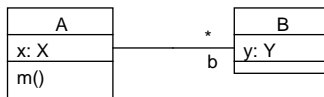
- Navigácia v rámci kontextovej inštancie: `self`, atribúty a operácie bez vedľajších účinkov (`isQuery() = true`)
- Navigácia cez asociácie
- Príklad:



- Ak je kontext trieda A
 - `self` – aktuálny objekt triedy A
 - `x` alebo `self.x` – hodnota atribútu `x`
 - `m()` alebo `self.m()` – výsledok operácie `m()`
 - `b` alebo `self.b` – objekt typu B, s ktorým je spojený `self`
 - `b.y` alebo `self.b.y` – hodnota atribútu `y`

Navigácia pri násobnosti väčšej ako 1

- Operácie vracajú množiny a multimnožiny (bag – vrece)



- Ak je kontext trieda A
 - b alebo `self.b` – množina všetkých objektov typu B spojených s objektom `self` – `Set(B)`
 - `b.y` alebo `self.b.y` – multimnožina hodnôt atribútu y – `Bag(Y)`
- Aký typ kolekcie operácia presne vráti závisí od vlastnosti príslušného konca asociácie (ordered/unordered, unique/nonunique)

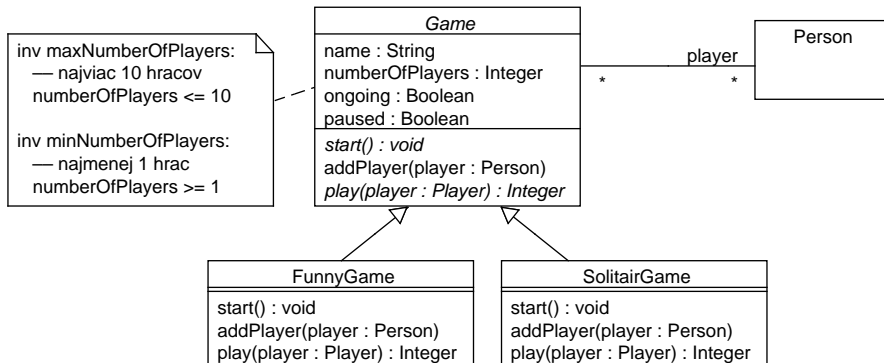
Invarianty, predpoklady a dôsledky v jazyku OCL

Invarianty, predpoklady a dôsledky

- Invariant musí platiť počas existencie každého objektu daného typu
- Predpoklad – precondition – má plátiť pred vykonaním operácie, inak nie sú zaručené správne výsledky
- Dôsledok – postcondition – to, čo operácia zaručuje, že bude platiť, ak platí predpoklad
- Invarianty sa definujú pre typy ako také a týkajú sa atribútov
- Predpoklady a dôsledky sa definujú pre operácie
- Pri dedení sa dajú prekonať a musí platiť:
 - invariant sa môže len zachovať alebo zosilniť
 - predpoklad sa môže len zachovať alebo zoslabiť
 - dôsledok sa môže len zachovať alebo zosilniť

Invariant (1)

- Príklad: model hry



Invariant (2)

- Dá sa zapísať aj mimo diagramu, ale potom treba uviesť kontext:

```
context Game
```

```
  inv maxNumberOfPlayers:
```

```
    -- najviac 10 hracov
```

```
    numberOfPlayers <= 10
```

```
  inv minNumberOfPlayers:
```

```
    -- najmenej 1 hrac
```

```
    numberOfPlayers >= 1
```

Vyjadrenie predpokladu (1)

- Nech pre základnú hru platí, že ak hra prebieha, hráčov možno pridať iba ak je pozastavená
- V OCL:

```
context Game::addPlayer(player : Person) : void
  pre addingAPlayerToGame:
    -- ak hra prebieha, musí byť pozastavená
    ongoing implies paused
```

Zoslabenie predpokladu

- Ale odvodená hra môže zoslabiť predpoklad:

```
context FunGame::addPlayer(player : Person) : void  
  pre addingAPlayerToGame:
```

- Predpoklad `addingAPlayerToGame` bol zoslabený (nahradený prázdny predpokladom) – hráčov možno pridávať za behu bez pozastavenia

Zosilnenie predpokladu – porušenie LSP (1)

- Odvodená hra však nesmie predpoklad zosilniť:

```
context FunGame::addPlayer(player : Person) : void
  pre addingAPlayerToGame:
    -- hra nesmie prebiehať
    not ongoing
```

- Predpoklad addingAPlayerToGame bol zosilnený: hráčov nemožno pridávať za behu bez ohľadu na to, či je hra pozastavená
- Týmto však porušíme Liskovej princíp substitúcie (podrobnejšie vysvetlený ďalej)

Zosilnenie predpokladu – porušenie LSP (2)

- Príklad: majme správcu hier, ktorého jednou z operácií je pridanie hráča do všetkých hier, do ktorých je to v danom okamihu možné, čo je dané predpokladom (v slučke cez všetky hry v správe):

```
if (ongoing <> true && paused <> true)
    game.addPlayer(player)
```

- Týmto pridáme aj hráča do hry typu FunGame, pričom predpoklad nebude splnený a operácia pridávania nemusí prebehnúť korektne

Vyjadrenie dôsledku

- Pridaním hráča sa zaznamenaný počet hráčov zvýši o 1:

```
context Game::addPlayer(player : Person) : void
  post addingAPlayerRaisesNumber:
    -- zaznamenaný počet hráčov sa zvýši o 1
    numberOfPlayers = numberOfPlayers@pre + 1
```

Liskovej princíp substitúcie

- O invarianty, predpoklady a dôsledky sa musíme starať aj keď ich neuvádzame explicitne
- Zvlášť je to dôležité pri použití dedenia
- Najdôležitejšie kritérium pre použitie dedenia je, či existuje vzťah typ-podtyp¹
- O tom je Liskovej princíp substitúcie (Liskov substitution principle)²:
 - Ak pre každý objekt o_1 typu S existuje objekt o_2 typu T taký, že pre všetky programy P definované v zmysle T správanie P je nezmenené, keď sa o_1 nahradí o_2 , potom je S podtypom T .

¹J. Coplien. Advanced C++ Programming Styles and Idioms. Addison-Wesley, 1992.

²B. Liskov. Data abstraction and hierarchy, ACM SIGPLAN Notices 23(5), 1987. 

Kruh a elipsa (1)

- Kruh je špeciálny prípad elipsy – matematicky
- Naozaj je vhodné použiť dedenie pre kruh a elipsu?³

```
public class Elipsa extends Utvar {
    private Bod f1;
    private Bod f2;
    private int a;
    private int b;

    public Bod getF1() { return f1; }
    public Bod getF2() { return f2; }
    public void setF1(Bod f1) {
        this.f1.setX(f1.getX());
        this.f1.setY(f1.getY());
    }
    public void setF2(Bod f2) {
        this.f2.setX(f2.getX());
        this.f2.setY(f2.getY());
    }
    ...
}
```

³R. C. Martin. The Liskov Substitution Principle. C++ Report, 1996.

Kruh a elipsa (2)

```
...  
public void setA(int a) { this.a = a; }  
public void setB(int b) { this.b = b; }  
}
```

- Bod – pre úplnosť:

```
public class Bod {  
    private int x;  
    private int y;  
  
    public int getX() { return x; }  
    public int getY() { return y; }  
    public void setX(int x) { this.x = x; }  
    public void setY(int y) { this.y = y; }  
}
```

Kruh a elipsa (3)

- Kruh odvodený od elipsy – trochu údajov bude navyše, ale dá sa

```
public class Kruh extends Elipsa {
    public void setF1(Bod f1) {
        this.f1.setX(f1.getX());
        this.f1.setY(f1.getY());
        this.f2.setX(f1.getX());
        this.f2.setY(f1.getY());
    }
    public void setF2(Bod f2) {
        this.f1.setX(f2.getX());
        this.f1.setY(f2.getY());
        this.f2.setX(f2.getX());
        this.f2.setY(f2.getY());
    }
    public void setA(int a) { this.a = this.b = a; }
    public void setB(int b) { this.b = this.a = b; }
}
```

- Čo však bude s klientským kódom?

Kruh a elipsa (4)

- Vďaka polymorfizmu môžeme všade kde sa očakáva elipsa použiť kruh
- Aj tam kde sa s tým nepočítalo:

```
public class C {  
    void move(Elipsa e, Bod b) { \\ posun elipsy o (b.x, b.y)  
        e.setF1(new Bod(e.f1.getX() + b.getX(), e.f1.getY() + b.getY()));  
        e.setF2(new Bod(e.f2.getX() + b.getX(), e.f2.getY() + b.getY()));  
    }  
}
```

- Kruh sa však posunie o dvojnásobnú vzdialenosť!
- To, že sa nemyslelo na správanie odvodeného typu na mieste pôvodného, predstavuje porušenie Liskovej princípu substitúcie

Design by Contract (1)

- Liskovej princíp substitúcie súvisi s návrhom podľa zmluvy (design by contract)⁴
- V návrhu podľa zmluvy operácie vystupujú ako zmluvné strany:
- Každá operácia definuje:
 - predpoklad (precondition)⁵ – podmienka, ktoré musia platiť pred operáciou
 - dôsledok (postcondition)⁶ – podmienka, ktorej platnosť zaručuje operácia po jej vykonaní, ak pred tým platil predpoklad
- Operácia garantuje, že ak platí predpoklad, po jej vykonaní bude platiť dôsledok

⁴ B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

⁵ obvyklý preklad *vstupná podmienka*

⁶ obvyklý preklad *výstupná podmienka*

Design by Contract (2)

- Príklad: operácia `setF1()` triedy `Elipsa` by mala garantovať, že sa po jej aplikácii atribút `f2` nezmení
- Pre elipsu podmienka bude dodržaná, ale pre kruh nie
- Došlo k zoslabeniu dôsledku
- Aby bol dodržaný Liskovej princíp substitúcie, pri podtypoch:
 - predpoklad sa môže len zachovať alebo zoslabiť
 - dôsledok sa môže len zachovať alebo zosilniť
- Za týmto účelom sme mohli k operácii `setF1()` definovať nasledujúci dôsledok (v OCL):
$$f2 = f2@pre$$
- Toto ohraňenie by potom musela dodržať (alebo zosilniť) aj prekonávajúca metóda a problém by nenastal

Sumarizácia

Sumarizácia

- Ohraničenia v UML
- Základy OCL
- Kontext výrazu v OCL: explicitný/implicitný
- Kolekcie a iterácie nad kolekciami
- Navigácia a násobnosť
- Invarianty, predpoklady a dôsledky – príklad použitia: dodržanie Liskovej princípu substitúcie

Čítanie

- J. Arlow and I. Neustadt. UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design. Addison-Wesley, 2nd edition, 2005.
- Špecifikácia OCL, <http://www.omg.org/spec/OCL/2.0/>
- Špecifikácia UML, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML → Superstructure specification