Podmienky a ohraničenia v modeloch: OCL

Poznámky k prednáškam z predmetu Modelovanie softvéru

Valentino Vranić
http://fiit.sk/~vranic/, vranic@stuba.sk
Ústav informatiky a softvérového inžinierstva
Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

25. október 2016

Obsah

1 Vyjadrenie ohraničení v UML **1**

2 Základy jazyka OCL **1**

3 Dopyty v jazyku OCL **3**

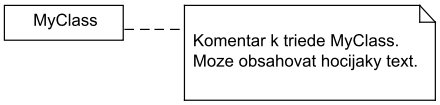
4 Invarianty, predpoklady a dôsledky v jazyku OCL **6**

5 Sumarizácia **11**

1 Vyjadrenie ohraničení v UML

Poznamky v UML modeloch

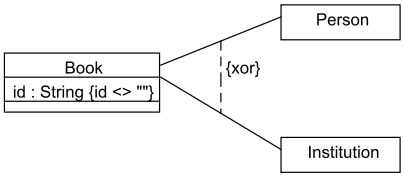
- UML má prvok na grafické vyjadrenie poznamky – note
- Poznámka sa prerušovanou čiarou priradí k prvku, na ktorý sa vzťahuje



- Poznámky možno využiť na vyjadrenie ohraničení – formálne alebo nefromálne

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

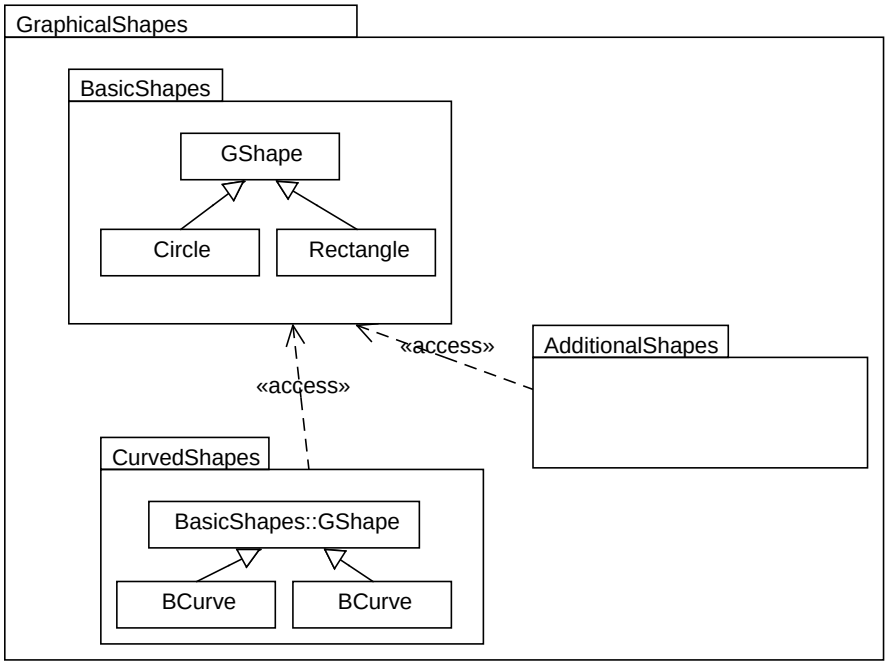
2 Základy jazyka OCL

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

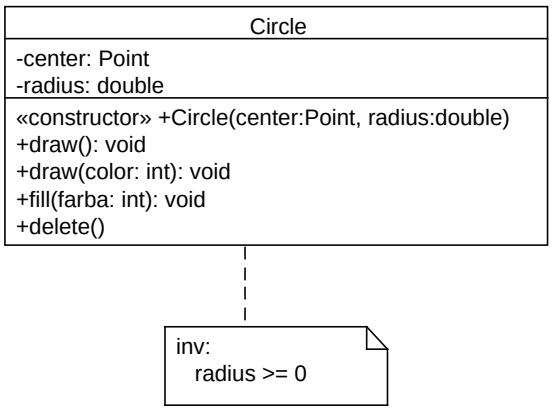
- Kontext závisí od priradenia OCL výrazu
- Vždy je to však inštancia klasifikátora, ku ktorému alebo ku ktorého prvku je OCL výraz priradený
- Ku kontextu sa pristupuje prostredníctvom priradeného identifikátora alebo kľúčovým slovom `self` (analógia `this` v Jave a C++)




```
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 Jave)



Typy OCL výrazov

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

3 Dopyty v jazyku OCL

Dopyty v jazyku OCL

- V jazyku OCL možno formulovať dopyty nad modelmi v UML
- Dôležité pre formulovanie zložitejších podmienok
- Možno využiť na špecifikáciu operácií dopytového typu
- Príprava pre formuláciu transformácie – jazyk QVT nadväzuje na OCL a používa sa v prístupe MDA (Model Driven Architecture) na definíciu transformácií medzi modelmi

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

Systém typov v OCL

- Silne typový jazyk
- Primitívne typy: `Boolean, Integer, Real, String`

- Štrukturovaný typ: `Tuple`
- Vstavané 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`)
  - `OclMesssage` – správa
- Typy z pridruženého UML modelu sa tiež stávajú OCL typmi

Niektoré operácie nad typom OclAny

- 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.oclIsTypeOf(b : OclType) : Boolean`
  - porovnanie typov objektov, ale vrátane podtypov `a.oclIsKindOf(b : OclType) : Boolean`
  - porovnanie stavov objektov
    `a.oclInState(b : OclType) : Boolean`
  - či je objekt nedefinovaného typu
    `a.oclIsUndefined() : Boolean`
- Dopyty:
  - množina všetkých inštancií typu A
    `A::allInstances() : Set(A)`
  - či objekt a vznikol ako výsledok danej operácie
    `a.oclIsNew() : 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 (templates)
- Inštancia kolekcie sa vytvára pre požadovaný typ: `Set(Circle)`
- OCL obsahuje veľa preddefinovaných operácií nad kolekciami – vyvolávajú sa pomocou operátora `->`:
  - `aCollection->collectionOperation(parameters)`
- Operácie nad kolekciami je však možné formulovať aj ako všeobecnú iteráciu s uplatnením zodpovedajúcej operácie nad jednotlivými prvkami kolekcie

Iterácie nad kolekciami

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

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

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

- Príklad všeobecnej iterácie:

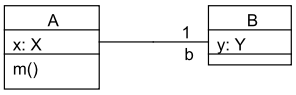
```
Bag{1, 2, 3, 4, 5}->iterate(number : Integer;
   sum : Integer = 0| sum = sum + number)
```

- To je ekvivalentné použitiu operácie `sum()`:

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

Navigácia

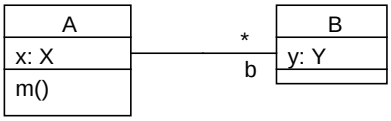
- Navigácia v rámci kontextovej inštancie: `self`, atribúty a operácie bez vedľajších účinkov (isQuery() = true)
- Navigácia cez asociácie
- Príklad:



- 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)

4 Invarianty, predpoklady a dôsledky v jazyku OCL

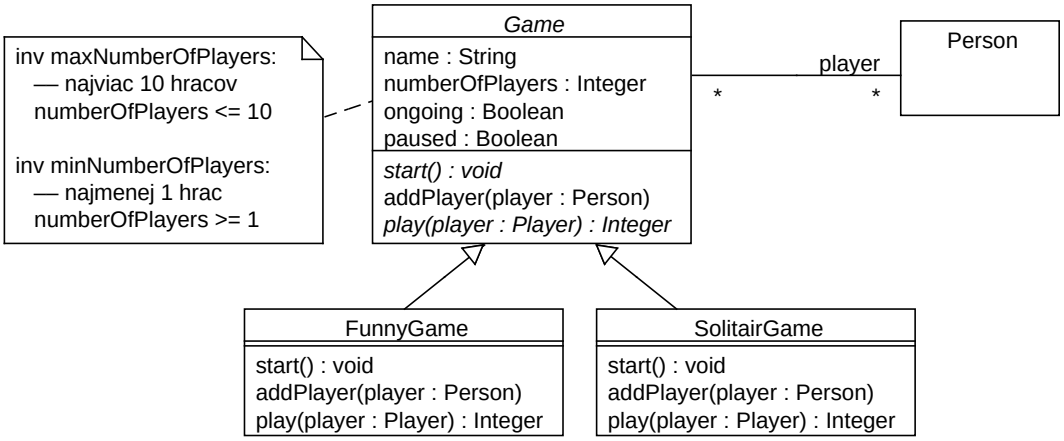
Invarianty, predpoklady a dôsledky

- 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

- Príklad: model hry



- 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

- 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 FunnyGame::addPlayer(player : Person) : void
   pre addingAPlayerToGame:
      true
```

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

Zosilnenie predpokladu – porušenie LSP

- Odvodená hra však nesmie predpoklad zosilniť:

```
context FunnyGame::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)
- 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):

```
for (Game game : allGames)
   if (!game.ongoing || game.paused) // ongoing => paused
      game.addPlayer(player)
```

- Týmto pridáme aj hráča do hry typu `FunnyGame`, 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 jestvuje vzťah typ-podtyp[1]
- O tom je Liskovej princíp substitúcie (Liskov substitution principle):[2]

  > Ak pre každý objekt o_1 typu S jestvuje objekt o_2 typu T taký, že pre všetky programy P definované v zmysle T správanie P je nezmenené, keď o_1 nahradí o_2, potom je S podtypom T.

Kruh a elipsa

- 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());
   }
   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 odvodený od elipsy – trochu údajov bude naviac, 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());
   }
```

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

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

[3] R. C. Martin. The Liskov Substitution Principle. C++ Report, 1996. http://www.objectmentor.com/resources/articles/lsp.pdf

```
    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 bude s klientskym kódom?
- 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 posunie o dvojnásobnú vzdialenosť!
- To, že sa nemyslelo na správanie odvodeného typu na mieste pôvodného, predstavuje porušenie Liskovej princípu substitúcie

Design by Contract

- 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) – podmienka, ktorá musia platiť pred operáciou
  - dôsledok (postcondition) – 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
- 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

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

  - dôsledok sa môže len zachovať alebo zosilniť

- Za týmto účelom sme mohli k operácii `setF1()` definovať nasledujúci dôsledok (v OCL):

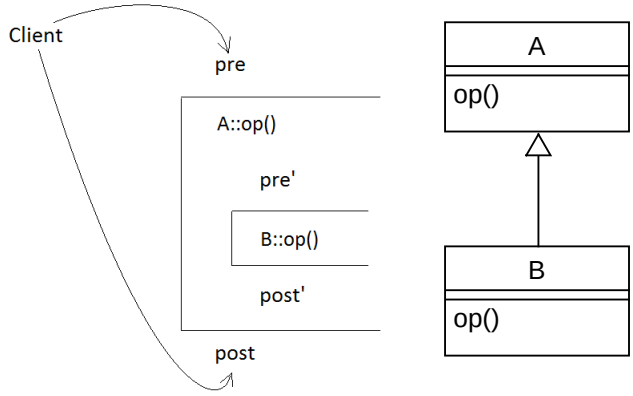
```
f2 = f2@pre
```

- Toto ohraničenie by potom musela dodržať (alebo zosilniť) aj prekonávajúca metóda a problém by nenastal

- Prekonávajúca operácia nemôže požadovať nič nad rámec toho, čo požaduje prekonaná operácia (predpoklad sa môže len zachovať alebo zoslabiť)
  - K prekonávajúcej operácií sa dá dostať cez prekonanú operáciu, lebo sa objekt podtypu môže vyskytnúť na mieste objektu nadtypu
  - Klientsky kód neoverí podmienky, ktoré musia platiť pred vyvolaním prekonanej operácie (ani ich nemusí poznať), a vyvolá ju

- Prekonávajúca operácia musí dodržať všetko, čo sa prekonaná operácia zaviazala splniť (dôsledok sa môže len zachovať alebo zosilniť)
  - Na mieste objektu nadtypu sa objaví objekt podtypu
  - Klientsky kód očakáva, že po vyvolaní prekonávajúcej operácie budú platiť podmienky, ktoré sa zaviazala splniť prekonaná operácia, a vyvolá ju

- Predpoklady a dôsledky pri dedení



$pre' \subseteq pre$
$post' \supseteq post$

5 Sumarizácia

- Operácia je ako služba: jej uskutočnenie je regulované zmluvou

- Na zápis podmienok je potrebný formálny jazyk: pri modelovaní v UML sa používa OCL

- Predpoklady špecifickejšej (prekonávajúcej) operácie možno nanajvýš zachovať, kým jej dôsledky musia byť aspoň zachované
- Niektoré podmienky platia vždy – invarianty; pri dedení musia byť aspoň zachované
- Podmienky v OCL sa vždy vyjadrujú v kontexte určitého objektu, pričom je niekedy potrebná zložitejšia navigácia
- OCL možno použiť aj pri strážcoch
- Pokročilé použitia OCL zahŕňajú definovanie pravidiel modelovania a transformácií