



ATL – ATLAS transformāciju valoda. Pamata konstrukcijas.

5. lekcija

Ērika Asņina, DITF LDI LDK

Kas ir ATL?

- ATL (Atlas Transformation Language) ir pētnieciskās grupas ATLAS INRIA & LINA (Francija) atbilde uz OMG grupas pieprasījumu QVT valodai
- ATL ir hibrīda modeļu transformācijas valoda, kura ir aprakstīta kā metamodelis un tekstuāla konkrēta sintakse.
 - Transformāciju rakstīšanas stils pamatā ir deklaratīvs (kartēšanas vienkāršošanai)
 - Imperatīvas konstrukcijas ir nodrošinātas specifiskajām kartēšanām
- MDE jomā ATL piedāvā līdzekļus mērķa modeļu veidošanai no ieejas modeļu kopas

ATL transformācijas programma

- Sastāv no likumiem, kuri definē kā atrast un vadīt avota modeļu elementus, lai izveidotu un inicializētu mērķa modeļu elementus.
- ATL definē:
 - Modeļu bāzes transformāciju
 - Modeļu vaicāšanas papildus ierīci, kura ļauj specificēt vaicājumus modeļiem
 - Koda faktORIZāciju, izmantojot ATL bibliotēku definīciju
- ATL IDE
 - Izstrādāts uz Eclipse platformas,
 - Standarta izstrādes rīki (sintakses izcelšana, atklūdotājs utt.)
 - Papildus ierīces modeļu un metamodeļu apstrādei

Ieskats ATL valodā: vienības

- Ļauj uzprojektēt dažādu veidu ATL vienības (*units*), katra no kurām ir definēta atsevišķajā failā ar paplašinājumu .atl.
 - Modeļu transformācijas tiek aprakstītas ATL *moduļos* (*modules*)
 - Transformācijas no modeļiem uz primitīvajiem datu tipiem tiek aprakstītas ATL *vaicājumos* (*queries*). Vaicājuma mērķis ir aprēķināt primitīvo vērtību (kā string vai integer) no avotmodeļiem.
 - Ļauj izstrādāt neatkarīgas ATL *bibliotēkas* (*libraries*), kuras var importēt no jebkura tipa ATL vienības, tai skaitā pašām bibliotēkām
- Paredzētas koda faktORIZĀcijai

ATL modulis (*module*)

- Atbilst transformācijām no avota modeļiem uz mērķa modeļiem
 - Avota un mērķa modeļu skaits **ir fiksēts**
- Avota un mērķa modeļi ir “tipizēti” atbilstoši to metamodeliem
 - UML metamodelis – tips UML,
 - Ecore metamodelis – tips Ecore,
 - utt.

ATL moduļa struktūra

- *Header* (galvene) definē transformācijas moduļa atribūtus
- *Import section* (importa sadaļa) ir opcijas daļa, kura ļauj importēt eksistējošās ATL bibliotēkas
- *Helper* (palīgs) ir ATL valodas ekvivalents Java metodēm, var būt vesela kopa palīgu moduļī.
- *Rule* (likums) definē veidu kā mērķa modeļiem jābūt ģenerētiem no avota modeļiem, var būt vesela kopa likumu moduļī.
- ***Helpers* un *rules* var būt deklarēti jebkādā kārtībā, ņemot vērā dažus nosacījumus.**

Galvene (*header*)

- Nosaka transformācijas moduļa un atbilstošo avota un mērķa modeļiem mainīgo nosaukumus

```
module moduļa_nosaukums;  
create izejas_modeli [from | refining] ieejas_modeli;
```

- Atslēgvārds **module** ievieš moduļa vārdu
 - faila nosaukumam jāsakrīt ar tajā izvietotā moduļa vārdu!
- Atslēgvārds **create** deklarē mērķa modeļus
- Atslēgvārdi **from** (normālā režīmā) un **refining** (precizējošās transformācijas gadījumā) deklarē avota modeļus
- Modeļu deklarācijas forma
modeļa_nosaukums : metamodela_nosaukums
- Vairāk par vienu modeli modeļu nosaukumi tiek atdalīti ar **komātu**.

Importa sadaļa

- Opcijas importa sekcija ļauj deklarēt kādas ATL bibliotēkas jāimportē.
- ATL bibliotēkas deklarācija ir realizēta šādi:
 - **uses bibliotēkas_faila_nosaukums_bez_paplašinājuma;**
 - Piemēram, lai importētu simbolu virkņu apstrādes bibliotēku, jādeklarē **uses strings;**
- **Lai deklarētu vairākas bibliotēkas jāizmanto dažas sekojošas viena aiz otrās *uses* instrukcijas.**

Helpers (palīgi jeb līdzētāji) [1]

- Ekvivalents Java metodēm
- Atvērās iespēja definēt faktorizētu ATL kodu, kuru ir iespējams izsaukt no ATL transformācijas dažādām vietām
 - Nosaukums (kurš atbilst metodes nosaukumam);
 - Konteksta tips, kas definē kontekstu, kurā ir definēts šis atribūts (tādā pašā veidā kā metode tiek definēta dotās klases kontekstā OOP);
 - Atgriežamas vērtības tips. **Katram līdzētājam jābūt atgriežamas vērtības tips;**
 - ATL izteiksme, kas attēlo ATL līdzētāja kodu;
 - Opciju parametru kopa, kurā parametrs ir identificēts ar pāri <parametra_nosaukums, parametra_tips)

Helpers (palīgi jeb līdzētāji) [2]

- Līdzētājs ar parametriem un kontekstu
 - **helper context** Integer **def** : max(x : Integer) : Integer = ...;
- Līdzētājs bez parametriem un ar kontekstu
 - **helper context** Integer **def** : double() : Integer = self * 2;
- Līdzētājs ar parametriem un ATL moduļa kontekstā
 - **helper def** : max(x1 : Integer, x2 : Integer) : Integer = ...;
 - Šajā gadījumā mainīgais **self** attiecās uz pašu izpildāmo moduli /vaicājumu.
- Atļauts vienā modulī izvietot līdzētājus ar vienādiem nosaukumiem, bet ar dažādām signatūrām

Helpers (palīgi jeb līdzētāji) [3]

■ Pastāv divu veidu līdzētāji

- Funkcionālie, ar vai bez parametru pieņemšanas
- Atribūtu līdzētājs, obligāti bez parametriem
 - ATL valodā to tā arī sauc par “atribūtu”
 - Definēts ATL moduļa vai modeļa elementa kontekstā
 - **Nedrīkts realizēt kolekcijas elementa kontekstā !**
- Abiem jābūt definētiem datu tipa kontekstā

■ Definēšanas shēma

helper [context konteksta_tips]? **def** :

līdzētāja_vārds(parametri) : atgriežamais_tips =
izteiksme;

Helpers (palīgi jeb līdzētāji) [4]

- Signatūra ietver līdzētāja vārdu ar atslēgvārdu *def*, to kontekstu, parametrus un atgriežamo vērtību.
 - **Tāču realizēta tikai vārda un konteksta kombinācija!**
- Līdzētāja ķermenis ir aprakstīts kā OCL izteiksme
 - OCL ir Object Constraint Language (objektu ierobežojumu valoda)
 - Palīginfo
 - prezentācija ORTUS-ā par OCL valodas pamatiem un
 - OCL Types and OCL Expressions in ATL (pp. 6-12)
<http://www.scribd.com/doc/50487315/3/OCL-Types-and-OCL-Expressions-in-ATL>

Helper (palīgi jeb līdzētāji) [5]

- Funkcionālie līdzētāji un atribūti izpildās dažādi
 - Funkcionālais līdzētāja rezultāts tiek aprēķināts **katru reizi**, kad tas tiek izsaukts
 - ATL atribūta rezultāts tiek aprēķināts **vienīgo reizi**, kad vērtība ir pieprasīta **pirmajā reizē**
 - ATL atribūti, kuri ir definēti ATL moduļa kontekstā, ir inicializēti (inicializēšanas posma gaitā) **to deklarēšanas secībā**.

Rules (likumi)

- Matched rules - sakrītošie likumi (deklaratīvā programmēšana)
- Lazy rules - “slinkie” likumi
- Called rules - izsaucamie likumi (imperatīva programmēšana)

Matched rules (1)

- Ļauj specificēt
 - Kādiem avota elementu tipiem jābūt uzģenerētiem mērķa elementiem
 - Veidu, ka uzģenerētiem mērķa elementiem jābūt inicializētiem.
- Likums tiek definēts pēc tā vārda.
- Darbības princips
 - Sakrīt ar avota modeļu elementa uzdoto tipu un ģenerē mērķa modeļu elementu vienu vai vairākus tipus
- Struktūra ar ievadvārdiem
 - Ievadvārds **rule**
 - Divas obligātas daļas (avota **from** un mērķa **to** paraugi)
 - Divas opciju daļas (lokālie mainīgie **using** un imperatīvs **do**)

Matched rules (2)

```
rule Author {  
  from  
    a : MMAuthor!Author  
  to  
    p : MMPerson!Person (  
      name <- a.name,  
      surname <- a.surname  
    )  
}
```


Matched rules (3)

■ Ierobežojumi!!!

- ☐ Katram avota modeļa elementam jābūt tikai vienam ATL sakrītošam likumam
- ☐ ATL sakrītošais likums nevar uzģenerēt ATL primitīvo tipu vērtības

Lazy un izsaucamie likumi

■ Lazy likumi

- Tādi paši kā sakrītošie likumi, tikai tiek izmantoti pēc tiešās izsaušanas citā likumā

■ Izsaucamie likumi

- Imperatīvas programmēšanas ierīce
- Līdzīgi līdzētājiem tiem jābūt tiešā veidā izsauktajiem, lai izpildītos, un tie pieņem parametrus
- Producē mērķa modeļu elementus kā sakrītošie likumi
- Var būt izsaukts no imperatīva koda daļas, sakrītošā likuma vai cita izsaucama likuma

Izsaucamais likums

- Opcijas lokālo mainīgo daļa
- NAV avota modeļa elementu paraugu daļas **from**
- Opcijas mērķa modeļu elementu paraugu daļa **to**
 - Mērķa modeļu elementu, kuri ir uzģenerēti izmantojot mērķa paraugu, inicializācijai jāizmanto lokālo mainīgo, parametru un moduļa atribūtu kombinācija
- Opcijas imperatīva daļa **do**

Izsaucama likuma piemērs

```
rule NewPerson (na: String, s_na: String) {  
  to  
    p : MMPerson!Person (  
      name <- na  
    )  
  do {  
    p.surname <- s_na  
  }  
}
```

ATL moduļa izpildes režīmi

■ Režīms pēc noklusēšanas

- Izstrādātājiem tiešā veidā jāspecificē veids kā mērķa modeļu elementiem jābūt ģenerētiem no avota modeļa elementiem.
- Ieejas modeļu nelielu izmaiņu gadījumā tas var būt ļoti nogurusi – jo būs jādefinē arī likumi, kas kopēs ieejas elementus izejas elementos

■ Precizēšanas izpildes režīms (refining)

- Ļauj izstrādātājam specificēt tikai tās modifikācijas, kurām jābūt izpildītām starp ieejas un izejas modeļiem transformācijās

Normālais izpildes režīms

- Režīms pēc noklusēšanas
- Saistīts ar atslēgvārdu *from* moduļa galvenē
 - Režīmā pēc noklusēšanas, ATL izstrādātājam ir jāapraksta (ar sakrītošiem vai nu izsaukamiem likumiem) veids kā ģenerēt katru no gaidāmo mērķa modeļu elementiem.
- Piemērots transformācijām, kurās mērķa modeļi atšķiras no avota modeļiem

Precizējošais izpildes režīms (refining)

- Atvieglo transformāciju starp līdzīgiem avota un mērķa modeļiem, jo galvenais uzsvars ir uzlikts uz modificēto mērķa elementu ģenerēšanu.
 - Nemaināmi elementi netieši kopējas uz mērķa modeļiem
- ATL moduļa galvenē uz šo režīmu norādā atslēgvārds ***refining***.
- Granularitāte ir definēta modeļa elementu līmenī
 - Tiklīdz elementa īpašība (atribūts vai atsauce) ir izmainīta, nepieciešams to aprakstīt ATL kodā
- Pašlaik tas ir realizēts 1:1 modeļiem, pie tam vienam un tam pašam ieejas un izejas modeļu metamodelim

Moduļu izpildes semantika

■ Refining režīms

- “in-place” versija, kad visas izmaiņas tiek tieši pielietotas **avota modeļu elementiem**

■ Izpildes režīms pēc noklusēšanas

- Sastāv no trīs secīgiem posmiem, proti
 - Moduļa inicializācijas posms
 - Avota modeļu elementu sakrišanas posms
 - Mērķa modeļu elementu inicializācijas posms

Moduļa inicializācijas posms

- Tiek inicializēti atribūti, kuri ir definēti transformācijas moduļa kontekstā
 - Tajā pašā laikā šo atribūtu inicializācija var izmantot atribūtus, kuri ir definēti avota modeļa elementu kontekstā, t.i. šie atribūti arī var būt inicializēti
- Ja bija definēts ieejas punkta izsaucamais likums ATL moduļa ietvaros, šā likuma kods (tai skaitā mērķa modeļu elementu ģenerācija) tiek izpildīta **pēc** ATL moduļa atribūtu inicializācijas.

Avota modeļu elementu sakrišanas posms

- Deklarēto sakrišanas likumu sakrišanas nosacījumi tiek pārbaudīti moduļa avota modeļu elementos.
- Kad sakrišanas nosacījums tiek apmierināts (izpildīts), ATL dzinējs iedala mērķa modeļu elementu kopu, kura atbilst likuma mērķa paraugam.
 - Elementi ir tikai iedalīti, inicializācija notiek nākamajā posmā!

Mērķa modeļu elementu inicializācijas posms

- Šajā posmā katrs iedalītais mērķa modeļu elements tiek inicializēts, izpildot piešķiršanas kodu, kurš ir saistīts ar atbilstošo mērķa parauga elementu.
 - Šajā posmā ir atļauta resolveTemp() operācijas izsaukšana ATL moduļa kontekstā. Tā ir specifiskā operācija, kurā iespējams norādīt uz jebkuru ģenerēšanai paredzēto mērķa modeļa elementu no ATL likuma.
- Imperatīva koda daļa tiek izpildīta tiklīdz ir pabeigts likuma inicializācijas solis.
 - Imperatīvs kods var izraisīt kāda ATL moduļa ietvaros definētā izsaucamā likuma izpildi

ATL vaicājums (query)

- Transformācija no modeļa uz primitīva tipa vērtību
 - Primitīvi tipi ir *Boolean*, *Integer*, *Real* un *String*, kuriem ATL dzinējs atbalstīta OCL specifiskās operācijas
- Visplašāk tiek pielietots tekstuāla izeja ģenerācijai no avota modeļu kopas.

ATL vaicājuma struktūra

- ATL vaicājumam jādefinē vaicājuma apstrāde **pēc importa daļas (opcijas)**
- Tiek izmantots atslēgvārds *query* un veids kā tā rezultātam jābūt aprēķinātam ar ATL izteiksmēm
 - **query** vaicājuma_vārds = izteiksme;
- Vaicājums var iekļaut līdzētāju vai atribūtu definīcijas
 - iespējams deklarēt moduļa kontekstā definētus līdzētājus un atribūtus ATL vaicājuma ietvaros.

Vaicājuma izpildes semantika

■ Posmu secība

- Inicializācijas posms – atbilst ATL moduļa inicializācijas posmam un paredzēts ATL moduļa kontekstā definēto atribūtu inicializācijai
- Aprēķina posms – tiek aprēķināta vaicājuma atgriežama vērtība, izpildot ATL vaicājuma elementa deklaratīvo kodu

■ Vaicājuma failā definētie līdzētāji arī var būt izsaukti šajā un inicializācijas posmos

ATL bibliotēkas

- Ļauj definēt ATL **funkcionālu** (ne atribūtu) **līdzētāju kopu**, kuri var būt izsaukti no dažādām ATL vienībām (moduļiem, vaicājumiem un bibliotēkām)
- Bibliotēka var importēt citu bibliotēku
- Bibliotēkās nav iespējams definēt līdzētājus moduļa kontekstā pēc noklusēšanas, kontekstam jābūt **tiešā veidā norādītam**
- Bibliotēka nevar būt izpildīta neatkarīgi no moduļiem un vaicājumiem



Object Constraint Language

Īss ievads

Kas ir OCL valoda?

- Valoda, kas ļauj papildināt UML modeli ar papildus informāciju, proti
 - Rakstīt vaicājumus, lai piekļūtu modeļa elementiem un to vērtībām
 - Uzstādīt modeļa elementu ierobežojumus
 - Definēt vaicājuma operācijas
- **OCL ir deklaratīva valoda. Tā nav darbību (action) valoda priekš UML!**

Ko nav iespējams darīt ar OCL?

- OCL nevar izmainīt modeļa elementa vērtību
- OCL nevar definēt citas operācijas, tikai vaicājuma operācijas
- OCL spēj tikai izpildīt tādas vaicājuma operācijas, kuri nemaina vērtības
- OCL nevar būt izmantots biznesa likumu specificēšanai izpildlaikā

Kāpēc izmanto OCL valodu?

- Tā ļauj spriest par UML modeļiem
- Tā ļauj ģenerēt kodu, kura pamatā ir OCL izteiksmes
- Tā ļauj samazināt pārpratumus
- Tā ir OCUP Advanced sertifikācijas daļa

OCL valodas sintakse

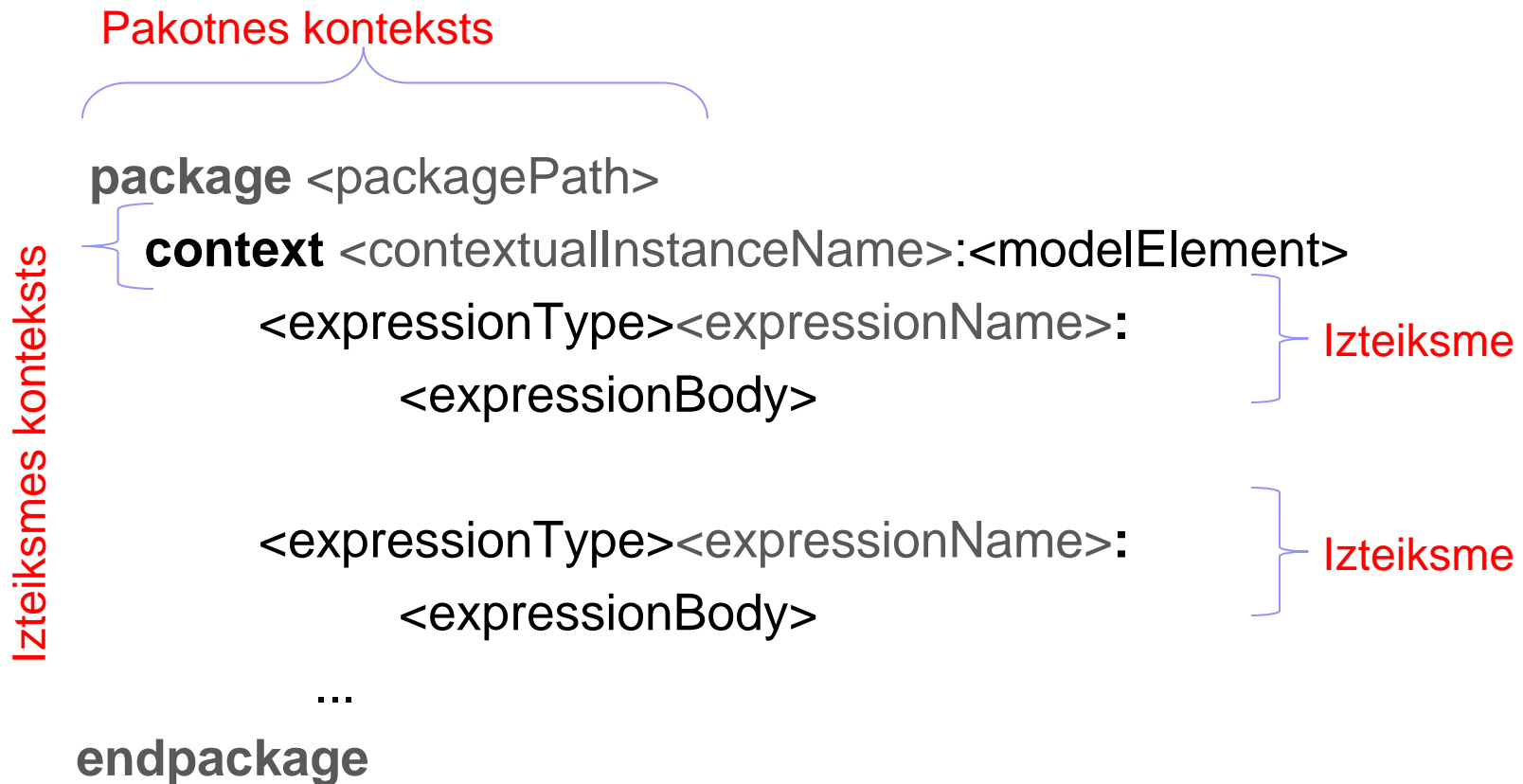
- Sintakse ir konstrukcijas un to formēšanas likumi kādā valodā
 - Līdzīga C/C++/Java stilam ar dažiem SmallTalk valodai līdzīgiem elementiem
- Semantika ir formāli definēta un neatkarīga no sintakses
- OCL valoda ir deklaratīva valoda
 - Drīzāk apraksta rezultātu, kurš ir jāsasniedz, nekā šā rezultāta sasniegšanu (atšķirībā no procedurālām valodām kā Java, C# un C++)

OCL valodas sintakse (turp.)

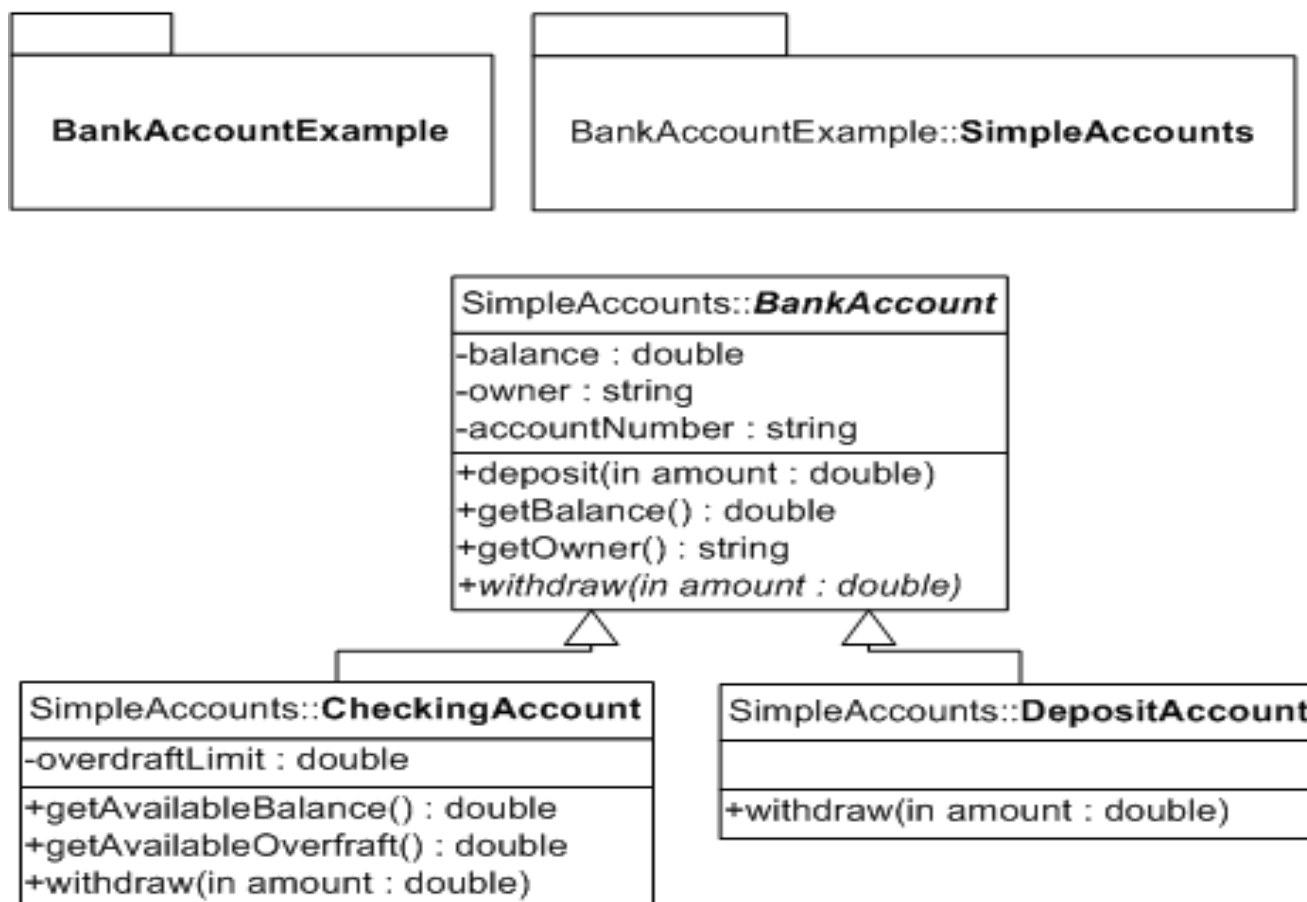
- OCL izteiksmes ir piesaistītas modeļa elementiem, lai kaut kādā veidā ierobežot modeli
- ***OCL nav programmēšanas valoda, tā ir ierobežošanas valoda***
- **OCL valodā specifiskā vaicājumus un nosacījumus, *nevis* uzvedību**

OCL valodas sintakse (turp.)

■ OCL izteiksmes vispārīgā forma



Izskatāmas klašu diagrammas piemērs*



Pakotnes konteksts un ceļa vārdi

- Specificē pakotni, kura definē vārdu telpu (namespace) OCL izteismēm
- Likumi
 - Ja pakotnes konteksts *nav specificēts*, vārdu telpa izteiksmei ir vesels modelis
 - Ja OCL izteiksme ir piesaistīta konkrētam modeļa elementam, tad izteiksmes vārdu telpa ir elementa piederības pakotne

package BankAccountExample::SimpleAccounts

...

endpackage

Pakotnes konteksts un ceļa vārdi (turp.)

- Ja elementu nosaukumi ir unikāli
 - Drīkst atsaukties uz elementa vārdu
- Ja elementiem *dažādās pakotnēs* ir viens un tas pats vārds, tad var
 - Definēt pakotnes kontekstu katrai OCL izteiksmei, kas attiecas uz elementu
 - Atsaukties uz elementiem, izmantojot pilnu ceļa vārdu
BankAccountExample::SimpleAccounts::BankAccount

OCL izteiksme ceļa vārdam atsaucei uz *elementu*:
Pakotne1::Pakotne2::.....::PakotneN::ElementaVārds

OCL: Izteiksmes konteksts

Izteiksmes konteksts

context <contextualInstanceName>:<modelElement>

<expressionType><expressionName>:

<expressionBody>

Izteiksme

<expressionType><expressionName>:

<expressionBody>

Izteiksme

...

Izteiksmes konteksts

- Izteiksmes konteksts norāda uz modeļa elementu, kuram ir piesaistīta OCL izteiksme

```
package BankAccountExample :: SimpleAccounts
```

```
context account : CheckingAccount
```

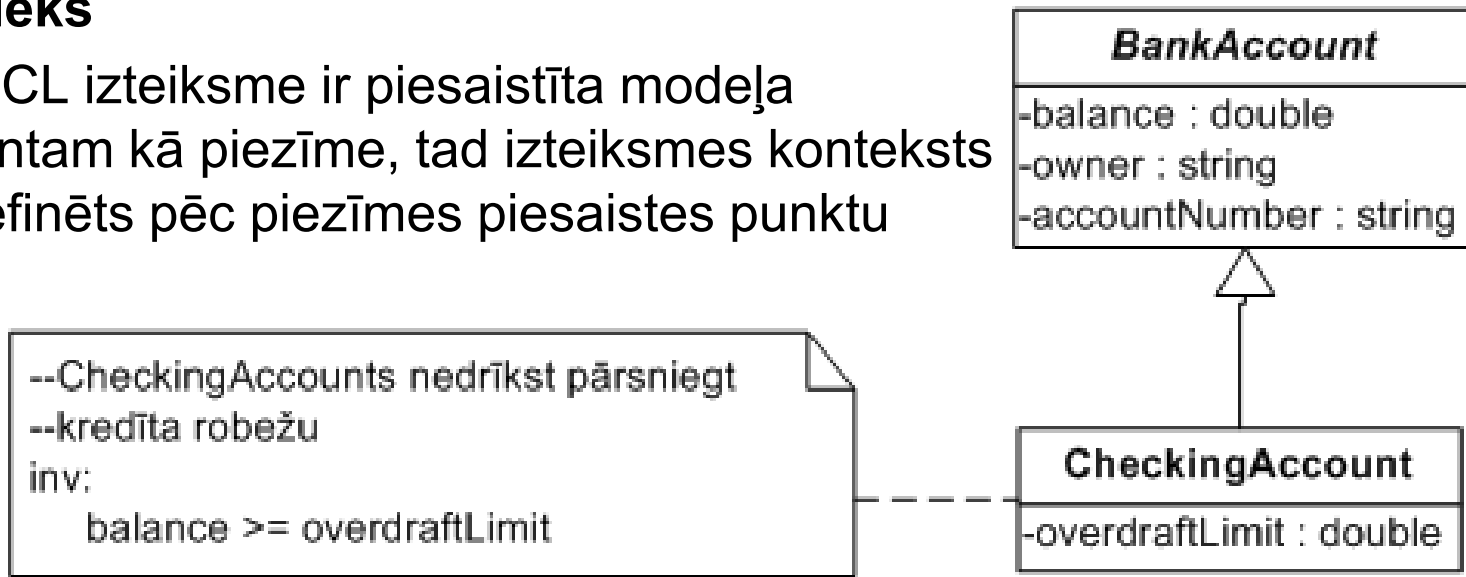
```
...
```

```
endpackage
```

- Izteiksme definē *kontekstuālo instanci*
 - *account* - opcijas nosaukums (vārds)
 - Izteiksmes ķermenī drīkst atsaukties ar atslēgvārdu *self*
 - *CheckingAccount* – obligāts tips

Izteiksmes konteksts (turp.)

- Ja izteiksmes konteksts ir klasifikators (klase, interfeiss, pakotne), tad kontekstuāla instance vienmēr ir **šī klasifikatora** instance
- Ja izteiksmes konteksts ir operācija vai atribūts, tad kontekstuāla instance ir tā klasifikatora eksemplārs, kas ir šīs operācijas vai atribūta **īpašnieks**
- Kad OCL izteiksme ir piesaistīta modeļa elementam kā piezīme, tad izteiksmes konteksts tiek definēts pēc piezīmes piesaistes punktu



Izteiksmes ķermenis

■ Komentāri

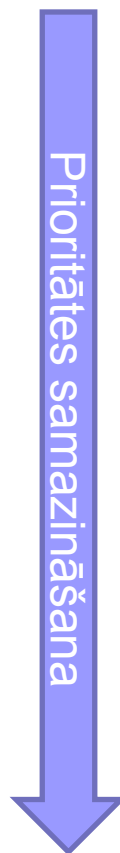
- -- Tas ir komentārs.

- /* Tas arī
ir komentārs. */

■ OCL atslēgvārdi, kurus nedrīkst izmantot kā nosaukumus

- and, attr, context, def, else,
endif, endpackage, if,
implies, in, inv, let, not,
oper, or, package, post,
pre, then, xor, body, init,
derive

Operāciju secība



Prioritātes samazināšana

::
@pre
. ->
not - ^ ^^
* /
+-
if..then..else..endif
< > <= >=
= <>
and or xor
implies

OCL tipu sistēma

■ Primitīvie tipi:

- Boolean
- Integer
- Real
- String, piem. 'Jānis'

■ Tuple (kortežs)

- Strukturēts objekts, ar vienu vai vairākām dēvētām daļām

▶ Iebūvētie tipi:

- OclAny
- OclType
- OclState
- OclVoid
- OclMessage

OCL primitīvie tipi

■ Boolean

- $a = b$, $a \neq b$, $a.\text{and}(b)$, $a.\text{or}(b)$, $a.\text{xor}(b)$, $a.\text{implies}(b)$
- if <Būlāzteikums> then
 <oclzteikums>
else
 <oclzteikums>
endif

■ Integer un Real

- $a.\text{mod}(b)$, $a.\text{div}(b)$, $a.\text{abs}(b)$, $a.\text{max}(b)$, $a.\text{min}(b)$,
 $a.\text{round}(b)$, $a.\text{floor}(b)$

OCL primitīvie tipi (turp.)

■ String

- $s1=s2$ (true/false), $s1\neq s2$ (true/false), $s1.concat(s2)$, $s1.size()$ – atgriež skaitli Integer, $s1.toLowerCase()$, $s1.toUpperCase()$, $s1.toInteger()$, $s1.toReal()$, $s1.substring(start, end)$ – apakšvirkne no start līdz end, pie kam start un end tips ir Integer, pirmā simbola pozīcija virknē ir 1, pēdējā simbola pozīcija virknē ir $s1.size()$
- Piemēram, `'Anna'.concat(' and Jon') = 'Anna and Jon'`

■ Tuple

- `Tuple{daļasVārds1:daļasTips1=vērtība1, daļasVārds2:daļasTips2=vērtība2, ...}`
- Pieeja pie daļas, izmantojot “.” operāciju

`Tuple{title:String='Aa', publisher:String='Bb'}.publisher`

OCCL iebūvētie tipi

- OclAny – visu OCCL tipu un saistītu UML modeļu supertips
- OclType – visu tipu saistītā UML modelī uzskaitījums
- OclState – visu stāvokļu saistītā UML modelī uzskaitījums
- OclVoid – OCCL valodā *null* tips. Tam ir vienīga instance OclUndefined
- OclMessage – attēlo ziņojumu

OCL iebūvētie tipi (turp.)

OclAny operācijas	Apraksts
Salīdzināšanas operācijas	
a.ocllsKindOf(b:OclType) : Boolean	Atgriež <i>true</i> , ja a ir tāds pats tips kā b vai b apakštips
a.ocllsUndefined():Boolean	Atgriež <i>true</i> , ja a = OclUndefined
a=b	Atgriež <i>true</i> , ja a ir tas pats objekts kā b
a<>b	Atgriež <i>true</i> , ja a <i>nav</i> tas pats objekts kā b
Vaicāšanas operācijas	
A::allInstances():Set(A)	Klases aptvēruma operācija, kura atgriež A tipa visu instanču kopu Set
Pārveidošanas operācijas	
a.oclAsType(SubType) : SubType	Novērtē un atgriež SubType. Ir iespējams pārveidot tipu vai nu līdz apakštipam, vai nu līdz supertipam.

OCL kolekcijas

- OCL kolekcijas var saturēt citus objektus, ieskaitot citas kolekcijas

OCL kolekcija	Sakārtota	Unikāla (bez dublikātiem)	Asociāciju galu īpašības
Set (kopa)	Nē	Jā	Pēc noklusēšanas {unordered, unique}
OrderedSet	Jā	Jā	{ordered, unique}
Bag (soma)	Nē	Nē	{unordered, nonunique}
Sequence (secība)	Jā	Nē	{ordered, nonunique}

Set(Customer)

OrderedSet{'Pirmdiena', 'Otrdiena', 'Trešdiena', 'Ceturtdiena'}

Sequence{<start> ... <end>} -> Sequence{1 ... (3+4)}

Operācijas ar kolekcijām

- ***Visas operācijas ar kolekcijām izteiksmēs tiek apzīmētas ar bultu “->”***
- Pārvēršanas (conversion) operācijas – pārveido kolekcijas vienu tipu otrā
- Salīdzināšanas operācijas – salīdzina kolekcijas
- Vaicāšanas (query) operācijas – dabū informāciju par kolekcijām
- Piekļuves (access) operācijas – piekļūst elementiem kolekcijām
- Izvēles operācijas – atgriež jaunu kolekciju, kura satur kolekcijas superkopu vai apakškopu
- Iterāciju operācijas

Konversijas operācijas

Bag{'Homer', 'Meg'} -> asOrderedSet()

Operācijas	Apraksts
X(T)::asSet() : Set(T)	Pārveido viena tipa kolekciju citā un atgriež jaunu kolekciju Kad kolekcija ir pārveidota Set tipā, dublējošie elementi tiek atmesti Kad kolekcijas ir pārveidotas OrderedSet vai Sequence tipā, tad oriģināla kārtība (ja bija) tiek saglabāta, vai nu elementi tiek patvaļīgi sakārtoti
X(T)::asOrderedSet() : OrderedSet(T)	
X(T)::asBag() : Bag(T)	
X(T)::asSequence() : Sequence(T)	

Salīdzināšanas operācijas

Operācijas	Semantika
$X(T) ::= (y: X(T))$: Boolean $Set\{1,2\} = Set\{3,4\}$	Set un Bag atgriež true, ja y satur tādus pašus elementus kā mērķa kolekcija OrderedSet un Sequence atgriež true, ja y satur tādus pašus elementus tādā pašā sakārtojumā kā mērķa kolekcija
$X(T) ::= <> (y: X(T))$: Boolean $Set\{1,2\} <> Set\{3,4\}$	Set un Bag – atgriež true, ja y <i>nesatur</i> tādus pašus elementus kā mērķa kolekcija OrderedSet un Sequence atgriež true, ja y <i>nesatur</i> tādus pašus elementus tādā pašā sakārtojumā kā mērķa kolekcija

Vaicāšanas operācijas

Operācijas	Apraksts
$X(T)::size() : \text{Integer}$ $Set\{1,2\} \rightarrow size() = 2$	Atgriež elementu skaitu mērķa kolekcijā
$X(T)::sum() : T$ $Set\{1,2\} \rightarrow sum() = 3$	Atgriež visu elementu summu mērķa kolekcijā, tipam T jāatbalsta summēšana
$X(T)::count(object:T) : \text{Integer}$	Atgriež <i>object</i> sastopamību skaitu mērķa kolekcijā
$X(T)::includes(object:T) : \text{Boolean}$	Atgriež true, ja mērķa kolekcija satur <i>object</i>
$X(T)::excludes(object:T) : \text{Boolean}$	Atgriež true, ja mērķa kolekcija nesatur <i>object</i>

Vaicāšanas operācijas (turp.)

Operācija	Apraksts
<code>X(T)::includesAll(c:Collection(T)) : Boolean</code>	Atgriež <i>true</i> , ja mērķa kolekcija satur visu, kas ir <i>c</i> kolekcijā
<code>X(T)::excludesAll(c:Collection(T)) : Boolean</code>	Atgriež <i>true</i> , ja mērķa kolekcija <i>nesatur</i> visu, kas ir <i>c</i> kolekcijā
<code>X(T)::isEmpty() : Boolean</code>	Atgriež <i>true</i> , ja mērķa kolekcija ir tukša, citādi atgriež <i>false</i>
<code>X(T)::notEmpty() : Boolean</code>	Atgriež <i>true</i> , ja mērķa kolekcija nav tukša, citādi atgriež <i>false</i>

Piekluves operācijas

Pieklūt elementam pēc pozīcijas drīkst tikai kolekcijās `OrderedSet` un `Sequence`, pārējās kolekcijās ir nepieciešama iterācija

Operācija	Apraksts
<code>OrderedSet(T)::first() : T</code> <code>Sequence(T)::first() : T</code>	Atgriež pirmo elementu kolekcijā
<code>OrderedSet(T)::last() : T</code> <code>Sequence(T)::last() : T</code>	Atgriež pēdējo elementu kolekcijā
<code>OrderedSet(T)::at(i) : T</code> <code>Sequence(T)::at(i) : T</code>	Atgriež elementu pozīcijā <i>i</i>
<code>OrderedSet::indexOf(T) : Integer</code>	Atgriež parametra objekta indeksu <code>OrderedSet</code> kolekcijā

Iterāciju operācijas

- Vispārīga forma ir šāda

aCollection -> <iteratoraDarbība>
(<iteratoraMainīgais>:<Tips> | <iteratoralzteiksme>)

- Iteratora darbība ir tāda

- *iteratoraDarbība* apmeklē katru elementu pēc kārtas kolekcijā *aCollection*. Tekošo elementu attēlo *iteratoraMainīgais*. *iteratoralzteiksme* tiek pielietots *iteratoraMainīgais*, lai panāktu rezultātu. Katra *iteratoraDarbība* apstrādā rezultātu savā specifiskā veidā. *Tips* ir vienmēr tāds, ka *aCollection* elementiem.

Iterāciju operācijas (turp.)

■ Operācija *iterate*

aCollection->iterate(<iteratorVariable> : <Type>

<resultVariable> : <ResultType> = <initializationExpression> |
<iteratorExpression>)

■ Piemēri:

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

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

Iterāciju operācijas (turp.)

Būla iteratora operācijas	Apraksts
$X(T)::\text{exists}(i:T \mid \text{iterator}z\text{teiksme}) : \text{Boolean}$	Ja ir vismaz viens patiess izteiksmes rezultāts
$X(T)::\text{forAll}(i:T \mid \text{iterator}z\text{teiksme}) : \text{Boolean}$	Ja visām i vērtībām ir patiess izteiksmes rezultāts
$X(T)::\text{forAll}(i:T, j:T, \dots, n:T \mid \text{iterator}z\text{teiksme}) : \text{Boolean}$	Ja ir patiess izteiksmes rezultāts katram kortežam $\{i, j, \dots, n\}$
$X(T)::\text{isUnique}(i:T \mid \text{iterator}z\text{teiksme}) : \text{Boolean}$	Ja katrai i vērtībai ir unikāla vērtība iteratoralzteiksme-ē
$X(T)::\text{one}(i:T \mid \text{iterator}z\text{teiksme}) : \text{Boolean}$	Ja iteratoralzteiksme ir pateiss <i>tieši</i> vienai i vērtībai

Iterāciju operācijas (turp.)

Iterācijas operācija	Apraksts
$X(T)::\text{any}(i:T \mid \text{iteratoralzteiksme}) : T$	Atgriež gadījuma elementu, kuram iteratoralzteiksme ir patiesa
$X(T)::\text{collect}(i:T \mid \text{iteratoralzteiksme}) : \text{Bag}(T)$	Atgriež Bag ar elementiem, kuriem iteratoralzteiksme ir patiesa
$X(T)::\text{collectNested}(i:T \mid \text{iteratoralzteiksme}) : \text{Bag}(T)$	Atgriež Bag ar kolekcijām, kurām iteratoralzteiksme ir patiesa
$X(T)::\text{select}(i:T \mid \text{iteratoralzteiksme}) : X(T)$	Atgriež kolekciju ar elementiem, kuriem iteratoralzteiksme ir patiesa
$X(T)::\text{reject}(i:T \mid \text{iteratoralzteiksme}) : X(T)$	Atgriež kolekciju ar elementiem, kuriem iteratoralzteiksme ir aplama
$X(T)::\text{sortedBy}(i:T \mid \text{iteratoralzteiksme}) : X(T)$	Atgriež kolekciju, kuras elementi ir sakārtoti atbilstoši iteratoralzteiksmei

OCL navigācija

- Navigācija ir process, kad jūs varat sekot saitei no avota objekta līdz vienam vai vairākiem mērķa objektiem
- Navigācijas izteiksmes var attiekties uz sekojošiem elementiem
 - Klasifikatori
 - Atribūti
 - Asociācijas gali
 - Vaicāšanas operācijas (operācijas, kuriem īpašība *isQuery* ir vienāda ar *true*)

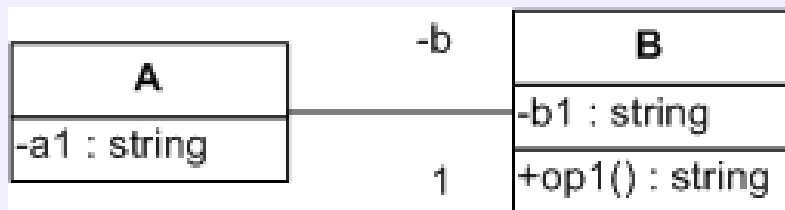
OCL navigācija (turp.)

A
-a1 : string
+op1() : string

- **self** – kontekstuāla instance – A klases instance
- **self.a1** vai **a1** – kontekstuālas instances atribūta a1 vērtība
- **self.op1()** vai **op1()** – op1() izsaukšanas uz kontekstuālas instances rezultāts; operācijai op1() ir jābūt *vaicājuma* operācijai

OCL navigācija pa asociācijām

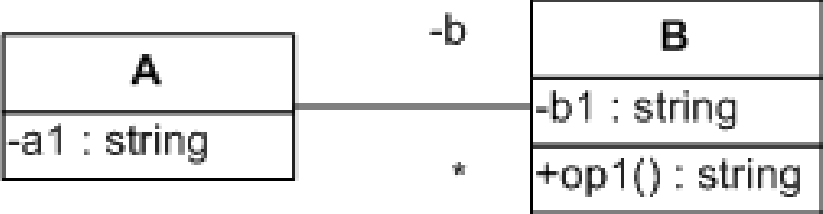
Piemēra modelis



Navigācijas izteiksme (A ir izteiksmes konteksts)

Izteiksme	Apraksts
self	Kontekstuāla instance – A klases instance
self.b	B tipa objekts
self.b.b1	B::b1 atribūta vērtība
self.b.op1()	Operācijas B::op1() rezultāts

OCL navigācija (turp.)

Piemēra modelis	Navigācijas izteiksme	
	Izteiksme	Vērtība
 <pre> classDiagram class A { -a1 : string } class B { -b1 : string +op1() : string } A --> B : -b class B "1" class A "*" </pre>	<p>self</p> <p>self.b self.b.b1</p> <p>self.b.op1()</p>	<p>Kontekstuāla instance – A klases instance B tipa objektu Set(B) B::b1 atribūta vērtību Bag(String). Īss pieraksts priekš self.b ->collect(b1) Operācijas B::op1() rezultātu Bag(String). Īss pieraksts priekš self.b ->collect(op1())</p>

Atgriežama kolekcija un asociāciju gali

OCL kolekcijas	Asociāciju galu īpašības
Set	{unordered, unique} – pēc noklusēšanas
OrderedSet	{ordered, unique}
Bag	{unordered, nonunique}
Sequence	{ordered, nonunique}

OCL izteiksmju tipi

- Ierobežojumus specificējošas izteiksmes
 - ☐ inv:
 - ☐ pre:
 - ☐ post:
- Atribūtus, operāciju ķermeņus un lokālus mainīgus specificējošas izteiksmes
 - ☐ init:
 - ☐ body:
 - ☐ def:
 - ☐ let:
 - ☐ derive:

OCL izteiksmju tipi:

inv:

- Tas, kas ir patiess visām klasifikatora instancēm
 - 1. Neviens konts nedrīkst pārsniegt kredītu vairāk par \$1000
context BankAccount
inv balanceValue :
 --BankAccount-am jābūt bilancei > - 1000.0
 self.balance >= (-1000.0)
 - 2. Katram accountNumber ir jābūt unikālam
context BankAccount
inv uniqueAccountNumber :
 -- katram BankAccount.accountNumber ir jābūt unikālam
 BankAccount::allInstances()->isUnique(account |
 account.accountNumber)

OCL izteiksmju tipi: pre:, post: un @pre

■ Operāciju pirms- un pēcnosacījumi

- 1. Noguldītam *amount* ir jābūt lielākam par 0 pirms operācijas izpildes

```
context BankAccount::deposit(amount:Double) : Double
  pre amountToDepositGreaterThanZero:
    amount > 0
```

- 2. Pēc operācijas izpildes *amount* jāpievieno *balance*

```
context BankAccount::deposit(amount:Double) : Double
  post depositSucceeded:
    self.balance = self.balance@pre + amount
```

Vārds @pre var būt izmantots tikai pēcnosacījumos!

- Pārdefinētas operācijas var tikai *pavājināt* pirms-nosacījumus un *pastiprināt* pēcnosacījumus

OCL izteiksmju tipi:

body:

■ Apraksta vaicājuma operācijas rezultātu

context CheckingAmount::getAvailableOverdraft() : Double

body:

if self.balance >= 0 then

-- ir pieejams pilns kredīts

self.overdraftLimit

else

-- kredīta daļa jau bija izmantota

self.balance + self.overdraftLimit

endif

OCL izteiksmju tipi:

init:, let:

- init: izmanto, lai uzstādītu atribūtu sākotnējas vērtības

```
context BankAccount::balance
```

```
init:
```

```
0
```

- let: izmanto, lai definētu mainīgo OCL izteiksmes robežās (kā lokālie mainīgie)

```
let <mainīgais>:<mainīgāTips> = <letIzteiksme> in  
  <izmantojošalzteiksme>
```

```
context BankAccount::withdraw(amount:Double)
```

```
post withdrawalSucceeded:
```

```
let originalBalance:Double = self.balance@pre in  
  self.balance = originalBalance - amount
```

OCL izteiksmju tipi:

derive:

- Izmanto, lai specificētu atvasināto atribūtu vērtības

```
context CheckingAccount::availableOverdraft : Double
```

```
  derive:
```

```
    if balance >= 0
```

```
      overdraftLimit
```

```
    else
```

```
      overdraftLimit + balance
```

```
    endif
```


OCL labā prakse

■ Labā prakse

- Vienmēr dot ierobežojumiem vārdus (pat ja tie ir opciju ierobežojumi)
- Izvēlēties aprakstošus vārdus, kuri atspoguļo ierobežojuma apkopotu semantiku
- Nodrošināt, ka ierobežojumu vārdi ir unikāli modeļa robežās
- Izmantot ierobežojuma nosaukumiem formātu lowerCamelCase

OCL realizācijas ATL dzinējā

■ Precizējoša informācija

- ATL/User Guide - The ATL Language
[http://wiki.eclipse.org/ATL/User_Guide -
_The_ATL_Language](http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language)
- Eclipse rīka palīgfailos