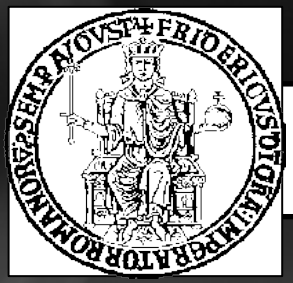


Università degli Studi di Napoli "Federico II"

Ingegneria del Software

a.a. 2013/14

Object Constraint Language Specification

Motivazioni per OCL

•Un modello grafico (ad esempio un class diagram) non è sufficiente per dare una specifica non ambigua

• Servono vincoli aggiuntivi per gli oggetti nel modello
ad esempio
  - invarianti di classe,
  - pre e post condizioni sui metodi

•Vincoli scritti in linguaggio naturale non risolvono l'ambiguità

•Dall'altro lato, i linguaggi formali per la scrittura di vincoli sono spesso non comprensibili a persone non specialiste.

•OCL viene presentato con l'intenzione di essere un linguaggio formale di facile uso e comprensione.

-

Caratteristiche generali di OCL

•**OCL è un linguaggio per scrivere espressioni sullo stato di un diagramma di oggetti**:

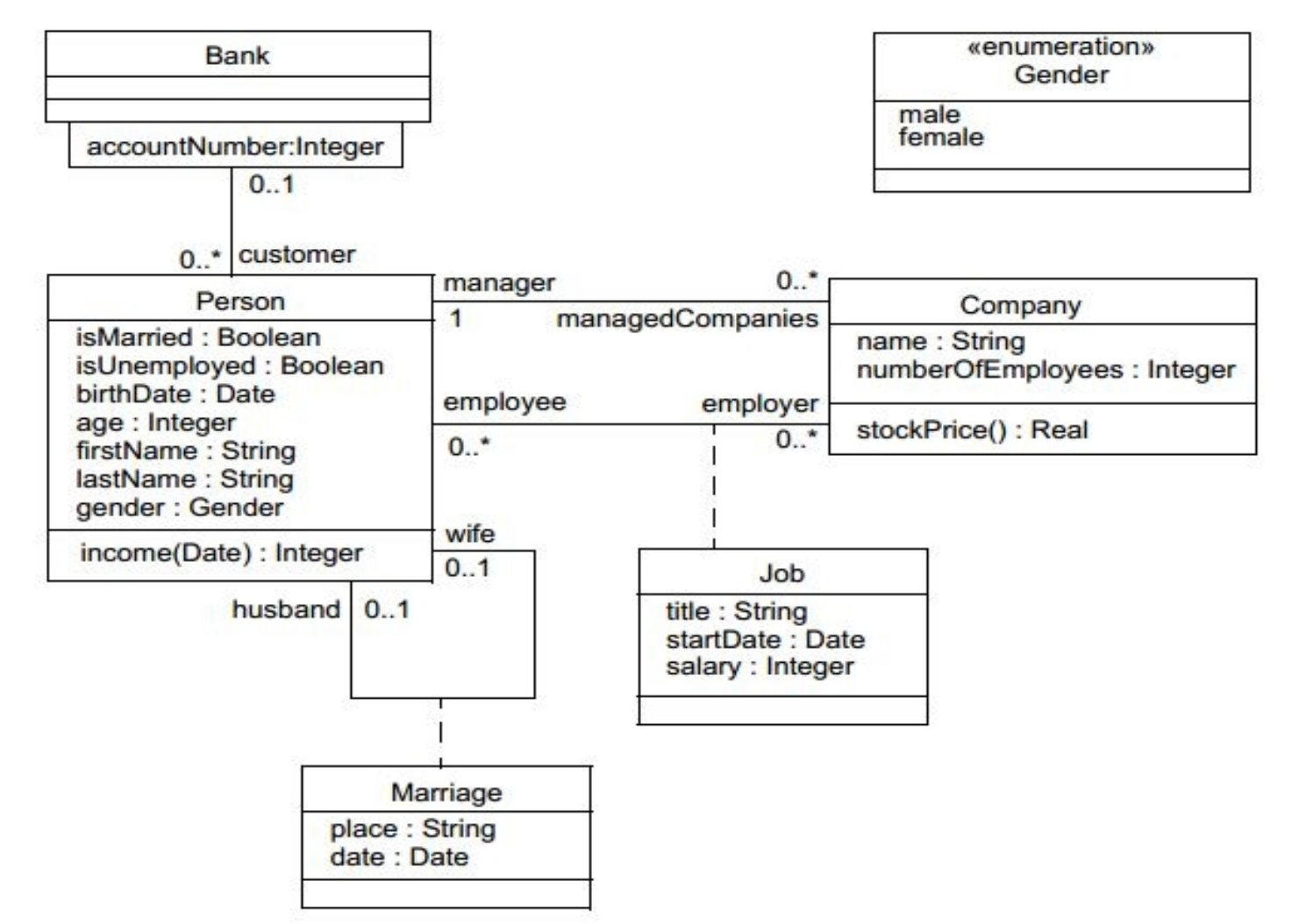
• La valutazione di una espressione restituisce un valore;
• La valutazione di una espressione non altera lo stato degli oggetti (**no side effect**);
• Ciascuna espressione OCL è **concettualmente atomica** e si intende che lo stato degli oggetti non cambia durante la valutazione.
• **OCL non è un linguaggio di programmazione** e non permette di descrivere il flusso di controllo in un programma, permette solo di parlare delle proprietà di una collezione di oggetti.
 OCL è un linguaggio tipato: ogni espressione ha un tipo e le operazione applicate a una espressione devono conformarsi al suo tipo.

Quando usare OCL

• **I vincoli espressi in OCL possono essere usati nelle seguenti occasioni**:

• Per esprimere **invarianti** su classi e tipi nel class diagram;
• Per esprimere **pre – condizioni** di una operazione o metodo in un class diagram.
• Per esprimere guardie
•Per esprimere vincoli sulle operazioni
•Per espremire l'insieme di destinatori di messaggi ed azioni

Class diagram esemplificativo



Invarianti in OCL

- **Un invariante** è un vincolo sullo stato che deve essere soddisfatto da ogni istanza della classe quando non sia in atto una variazione ad opera di un metodo o operazione
- Es.

  context Company inv:
  self.numberOfEmployees > 50
- Variante equivalente

  context c : Company inv:
  c.numberOfEmployees > 50
- Invariante con nome

  context c : Company inv enoughEmployees:
  c.numberOfEmployees > 50

Pre e Post condizioni in OCL

• Una **pre-condizione** è un vincolo (sullo stato e sui parametri attuali) che deve essere soddisfatto perché il metodo possa essere invocato

•Una **post-condizione** è un vincolo
- sullo stato precedente l'invocazione del metodo,
- sullo stato successivo
-eventualmente sul risultato restituito
che deve essere soddisfatto dopo che il metodo è stato invocato.

Pre e Post condizioni in OCL

- FORMATO:

  context Typename::operationName(param1 : Type1, …):
  ReturnType
  pre : param1 > …
  post: result = …

  context Person::income(d : Date) : Integer
  post: result = 5000

  - **Con nome**
    context Typename::operationName(param1 : Type1, …):
    ReturnType
    pre parameterOk: param1 > …
    post resultOk : result = …

Quando usare OCL

- OCL può essere usato per esprimere interrogazioni su un class diagram

- **Body expression: Formato**

context Typename::operationName(param1 : Type1, …): ReturnType
body: -- some expression

- Esempio

```
context Person::getCurrentSpouse() : Person
pre:   self.isMarried = true
body:  self.mariages->select( m | m.ended = false ).spouse
```

Quando usare OCL

- OCL può essere usato per esprimere **inizializzazione** di attributi e **attributi derivati**

```
context Typename::attributeName: Type
init:  -- some expression representing the initial value
context Typename::assocRoleName: Type
derive:  -- some expression representing the derivation rule

example:
context Person::income : Integer
init:   parents.income->sum() * 1% -- pocket allowance
derive:  if underAge
        then parents.income->sum() * 1% -- pocket allowance
        else job.salary              -- income from regular job
        endif
```

Tipi per le espressioni OCL: Tipi base

- Tipi di base sempre disponibili

type	values
Boolean	true, false
Integer	1, -5, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...'

type	operations
Boolean	and, or, xor, not, implies, if-then-else
String	concat(), size(), substring()

type	operations
Integer	*, +, -, /, abs()
Real	*, +, -, /, floor()

Tipi per espressioni OCL

- Enumerazioni
  - Es. Gender enum{'male','female'}
  - context Person inv: gender = Gender::male
- Ogni classe presente nel contesto considerato (Class Diagram) è un tipo OCL
- Tipi Collezione
- Collection (T)
  - Set(T) collezione di elementi di tipo T, senza molteplicità, senza ordinamento
  - Bag(T) collezione di elementi di tipo T, con molteplicità, senza ordinamento
  - Sequence(T) collezione ordinata di elementi di tipo T.

Tipi per espressioni OCL, Collezioni di letterali

- **Set di letterali**
  - Set { 1 , 2 , 5 , 88 }
  - Set { 'apple' , 'orange', 'strawberry' }
- **Sequenze di letterali**
  - Sequence { 1, 3, 45, 2, 3 }
  - Sequence { 'ape', 'nut' }
- Bag di letterali
  - Bag {1 , 3 , 4, 3, 5 }

Tipi per espressioni OCL, Collezioni di letterali (2)

- Definizione di sequenze di interi mediante espressioni di tipo intero
- Sequenze definite da due espressioni intere, i cui valori rappresentano gli estremi dell'intervallo

Sequence {int_expr1 .. int_expr2}

- Le seguenti definizioni danno luogo alla stessa sequenza
  - Sequence{ 1..(6 + 4) }
  - Sequence{ 1..10 }
  - Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

Come ottenere istanze di collezioni

- **Scrivere una collezione di letterali**:
  - Set {2 , 4, 1 , 5 , 7 , 13, 11, 17 }
  - OrderedSet {1 , 2, 3 , 5 , 7 , 11, 13, 17 }
  - Sequence {1 , 2, 3 , 5 , 7 , 11, 13, 17 }
  - Bag {1, 2, 3, 2, 1}
- **Mediante navigazione delle proprietà a partire da una istanza di oggetto o da una classe**
  - **context** Company **inv**:
  - self.employee
- **Usando operazioni sulle collezioni a partire da collezioni**
  - collection1->union(collection2)
  - **Esiste una ampia gamma di operazioni predefinite sulle collezioni**

Operazioni predefinite sul tipo Collection

- **size() : Integer**
  - Il numero di elementi nella collezione self.
- **isEmpty() : Boolean**
  - true se self è la collezione vuota, false altrimenti
- **notEmpty() : Boolean**
  - true se self non è la collezione vuota, false altrimenti
- **includes(object : T) : Boolean**
  - True se object è un elemento di self, false altrimenti
- **count(object : T) : Integer**
  - Il numero di occorrenze di object in self.

…….

Operazioni predefinite sul tipo Collection (2)

- **sum() : Integer**
  - La somma di tutti gli elementi in self (T tipo compatibile).
- **product(c2: Collection(T2)) : Set(Tuple(first: T, second: T2))**
- Il prodotto cartesiano di self and c2.

Esempi operazioni predefinite sul tipo Set

- **Tutte le operazioni definite su Collection**
- **union(s : Set(T)) : Set(T)**
  - L'unione di self and s.
- **intersection(s : Set(T)) : Set(T)**
  - Intersezione di self and s
- **– (s : Set(T)) : Set(T)**
  - Gli elementi di self, che non sono in s.
- **= (s : Set(T)) : Boolean**
  - true se self s s contengono gli stessi elementi.

…... e molte altre.

Esempi operazioni predefinite sul tipo OrderedSet

- **Tutte le operazioni definite su Collection**
- **append (object: T) : OrderedSet(T)**
  - L'insieme di elementi di self seguito da object
- **prepend(object : T) : OrderedSet(T)**
  - L'insieme di elementi di self preceduto da object
- **insertAt(index : Integer, object : T) : OrderedSet(T)**
  - L'insieme di elementi di self con l'inserimento di object nella posizione indicata

Esempi operazioni predefinite sul tipo OrderedSet(2)

- **at(i : Integer) : T**
  - i-esimo elemento dell'insieme

indexOf(obj : T) : Integer

  - L'indice dell'elemento obj nell'insieme
- **first() : T**
  - Il primo elemento dell'insieme
- **last() : T**
  - L'ultimo elemento dell'insieme
- .... altre operazioni

Uso di sottoespressioni

- E' possibile associare un nome ed un tipo ad una sottoespressione quando è utile usarla più volte.

```
context Person inv:
    let income : Integer = self.job.salary->sum() in
        if isUnemployed then
           income < 100
        else
          income >= 100
        endif
```

Conformità di tipo

- I tipi di OCL sono organizzati in una gerarchia di tipi.
- La gerachia determina la conformità di un tipo con un altro
- Una espressione Ocl è valida se i tipi coinvolti sono conformi
- Un tipo *type1* è conforme a un tipo *type2 quando una isatnza del di type1 può essere sostituita al posto di una occerrenza di di type2*
- Regole di conformità
- *Ogni tipo è conforme ad un suo supertipo*
- *La conformità è transitiva: se type1 è conforme a type2 e type2 è conforme a type3 allora type1 è conforme a type3*

Regole di conformità per tipi collezione e basici

Type	Conforms to/Is a subtype of	Condition
Set(T1)	Collection(T2)	if T1 conforms to T2
Sequence(T1)	Collection(T2)	if T1 conforms to T2
Bag(T1)	Collection(T2)	if T1 conforms to T2
Integer	Real	

Operazione di casting a un sottotipo

- L'operazione di casting a un sottotipo può servire ad accedere alla proprietà di una istanza quando sia certo che essa è associata a un sottotipo type2 del tipo corrente type1
- L'oggetto corrente può essere ri-tipato con l'operazione

oclAsType(OclType)

- Avento un oggetto object di tipo type1 e un suo sottotipo type2 si può scrivere

object.oclAsType(type2)

Valori indefiniti

- La valutazione di una espressione può portare ad un valore indefinito (ad es. estrarre il primo elemento di una sequenza vuota)
- In generale una espressione ha valore indefinito se una sua sottoespressione ha valore indefinito con le seguenti eccezioni
  - **True OR Undef valuta True**
  - **False AND Undef valuta False**
  - **False IMPLIES Undef valuta True**
  - **Undef IMPLIES True valuta True**
- Esiste una operazione sul tipo generale (OclAny) per testare l'indefinitezza del valore di una espressione

oclIsUndefined()

(valuta a True se l'argomento valuta a Undef., a False altrimenti)

Oggetti e proprietà

- Una espressione OCL può far riferimento alle seguenti **proprietà** presenti in un class diagram (classi, interfacce, associazioni):
- Attributi
- Nomi di ruolo nelle associazioni
- Un operazione avente l'attributo **isQuery** a vero (l'esecuzione dell'operazione non ha side effect sullo stato)
- Un metodo avente l'attributo isQuery a vero.

Per riferire una proprietà si fa uso della notazione puntata (dot notation)

Es. obiect.attributo

context Person inv:

self.age > 0

Oggetti e proprietà (2)

Es. obiect.metodo()

aPerson.income(aDate).bonus = 300 and

aPerson.income(aDate).result = 5000

.bonus fa riferimento ad un parametro out del metodo

.**result** fa riferimento al risultato del metodo

Es. definizione risultato di una operazione

context Person::income (d: Date) : Integer

post: result = age * 1000

Result è parola chiave che indica il valore restituito

Oggetti e proprietà (3)

Definizione del risultato dell'operazione e del parametro out

context Person::income (d: Date, bonus: Integer) : Integer

post: result = Tuple { bonus = ...,

Result = ... }

In presenta di parametri **out o in/out** il risultato è costituito da una ennupla (formata col costruttore **Tuple{}) che comprende il risultato (result)** e i valori dei parametri **out e in/out**

Oggetti e proprietà: ruoli e navigazione (2)

- Se l'associazione ha cardinalità m .. * (molti)
  - object.role1 è una espressione di tipo Set (Class1)
  - Ha come valore una collezione di oggetti di tipo Class1
- Se l'associazione ha cardinalità m .. * (molti) ed è qualificata come {ordered}
  - object.role1 è una espressione di tipo Sequence(Class1)
  - Ha come valore una collezione ordinata di oggetti di tipo Class1
- Es. Invariante (il manager di una compagnia non è disoccupato; la compagnia non ha insieme vuoto di dipendenti)
  - **Alla proprietà di una collezione si accede mediante ->**

```
context Company
inv: self.manager.isUnemployed = false
inv: self.employee->notEmpty()
```

Oggetti e proprietà: ruoli e navigazione (3)

- In caso di ruolo con cardinalità 0..1 o 1 è possibile considerare il valore dell'espressione object.role come un insieme singoletto (contenente un unico oggetto) anziché un oggetto.
- Es. : Una compagnia ha uno ed un solo manager

  context Company **inv**:

  self.manager->size() = 1

- Tecnica usata per controllare l'esistenza di un oggetto associato in caso di partecipazione parziale all'associazione

  context Person inv:

  self.wife->notEmpty() implies self.wife.gender = Gender::female

Navigazione alle classi di associazione

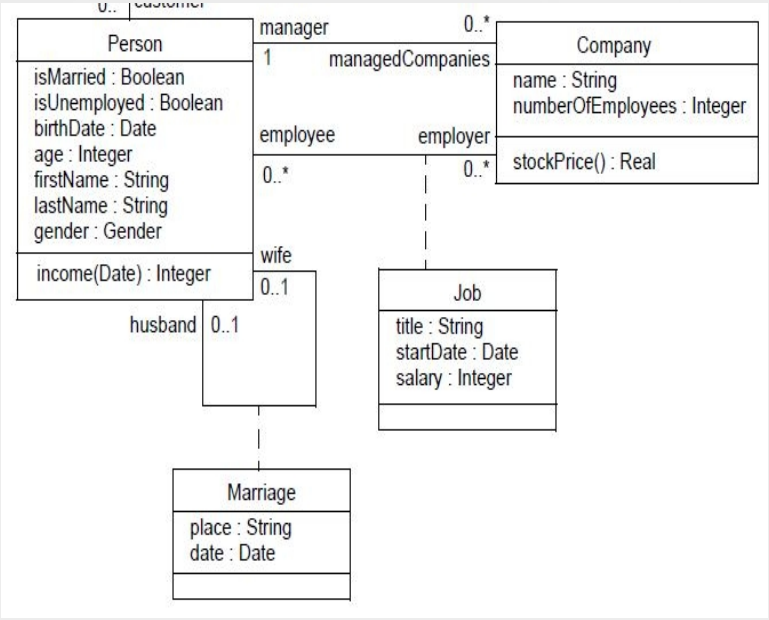
- Per la navigazione ad una classe di associazione si usa
- Il nome della classe di associazione con lettera minuscola
- In caso di ambiguità la qualificazione del ruolo di navigazione
- Es. **context Person inv:**

  self.job[employer]

In assenza di ambiguità

- **context Person inv:**

  self.job

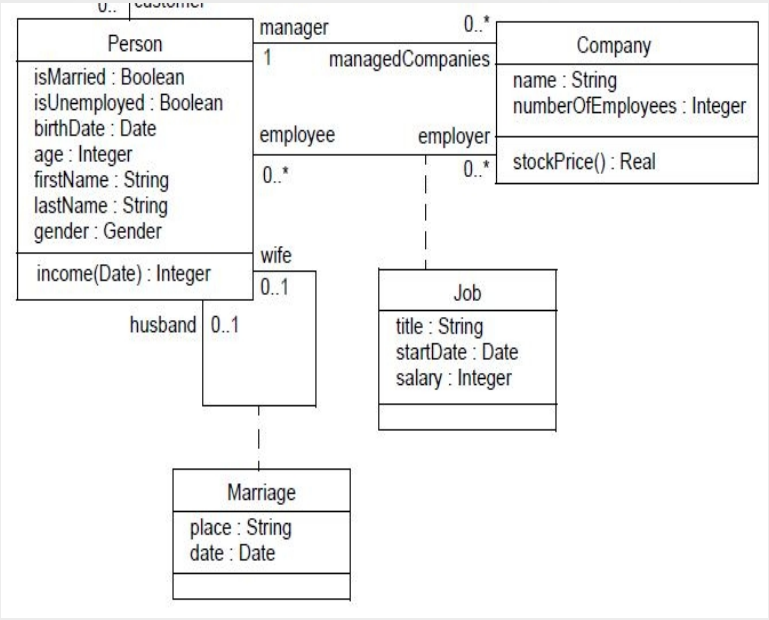


Dalle classi di associazione

- Per la navigazione da una classe di associazione si usa indicazione del ruolo.
- Ad una istanza di associazione è associato esattamente un oggetto per ogni ruolo.
- Es. **context Job**
- **Inv:**

self.employer.numberOfEmployees >= 1

inv: self.employee.age > 21



Proprietà predefinite di tutti gli oggetti

oclIsTypeOf (t : OclType) : Boolean

Vera se *type di* self e *t sono gli stessi.*

Es. context Person

inv: self.oclIsTypeOf(Person) -- è true

inv: self.oclIsTypeOf(Company) -- è false

oclIsKindOf (t : OclType) : Boolean

Vera se *type di* self e *t sono gli stessi o se t è un suo supertipo*

oclIsNew () : Boolean

Vera se self e *un nuovo oggetto creato da un metodo (usato nelle postcondizioni)*

oclAsType (t : OclType) : instance of OclType (casting)

Proprietà di classe (statiche)

- Ci sono proprietà non definite sulle istanze degli oggetti ma sulle loro classi.
- Il formato della espressione è

Class.proprietà

- Tra le proprietà predefinite più interessante quella che associa ad una classe Class la collezione delle sue istanze

Class.allInstances()

- **Esempio: Tutte le persone hanno nome diverso.**

```
context Person inv:
Person.allInstances()->forAll(p1, p2 |
p1 <> p2 implies p1.name <> p2.name)
```

- Person.allInstances() associa come valore il set delle istanze delle persone presenti al momento della valutazione.

Valori precedenti delle proprietà nelle postcondizioni

- In una postcondizione si può far riferimento ai volori di proprietà a due momenti temporali distinti
  - Il valore della proprietà all'inizio dell'operazione o metodo
  - Il valore della proprietà al completamento dell'esecuzione dell'operazione o del metodo
- Un espressione in una postcondizione fa riferimento al valore della proprietà al completamento dell'esecuzione dell'operazione o del metodo.
- Per riferirsi a al valore della proprietà all'inizio dell'operazione o metodo si deve indicare il suffisso **@pre nella proprietà.**
- Es. **context** Person::birthdayHappens()

  post: age = age@pre + 1

Valori precedenti delle proprietà nelle postcondizioni(2)

- In un metodo con parametri **@pre viene indicato prima dei parametri.**

  Es. **context** Company::hireEmployee(p : Person)

  post: employees = employees@pre->including(p) **and**
  stockprice() = stockprice@pre() + 10

- Attenzione all'uso di **@pre nella navigazione degli oggetti**
  - **a.b@pre.c –** vecchio valore della proprietà b di a, sia x
  - - e poi il nuovo valore di c di x.
  - **a.b@pre.c@pre --** vecchio valore della proprietà b di a, sia x
  - - e poi il vecchio valore di c di x.
- Il vecchio valore di un oggetto **distrutto** nel metodo è OclUndefined
- Il vecchio valore di un oggetto **creato** nel metodo è OclUndefined

Ennuple di valori distinti (Record)

- **E' possibile comporre (anche di tipi diversi) in una ennupla .**
- Ciascuna componente della ennupla ha designato un nome ed un tipo.
- I tipi della componente di un'ennupla possono essere strutturati
- **Esempi:**
- Tuple {name: String = ‘John’, age: Integer = 10}
- Tuple {a: Collection(Integer) = Set{1, 3, 4}, b: String = ‘foo’, c: String = ‘bar’}
- **I tipi di dati sono opzionali e l'ordine non è influente**: sono equivalenti
  - Tuple {name: String = ‘John’, age: Integer = 10}
  - Tuple {name = ‘John’, age = 10}
  - Tuple {age = 10, name = ‘John’}

Ennuple di valori distinti (Record) (2)

- Alle componenti di una ennupla si fa eccesso mediante la notazione puntata
  - Es. **Tuple {x : integer = 5,y = string = 'prova'}.x =5**
- I tipi della componente di un'ennupla possono essere strutturati
- I valori delle componenti di una tupla possono essere assegnati mediante generiche espressioni OCL.
- **Tuple{Conteggio : integer = Person.allInstances->size(),**

  Collezione : Set(Person) = Person.allIstances()}
- **Tipo della tupla in esempio**
- **TupleType{Conteggio : integer,**

  Collezione : Set(Person)}

Manipolazione delle collezione

- OCL mette a disposizione oltre alla operazioni standard di manipolazione delle collezioni anche operazioni di filtraggio di collezioni.
- **Filtraggio positivo**
  - collezione->select (espressione booleana)
  - Ha come valore il sottoinsieme degli elementi della collezione rispetto ai quali l'espressione booleana valuta a true
- **Filtraggio negativo**
  - collezione->reject(espressione booleana)
  - Ha come valore il sottoinsieme degli elementi della collezione rispetto ai quali l'espressione booleana valuta a false

Filtraggio delle collezione: select

- **collezione->select (espressione booleana)**
- La compagnia deve avere almeno un impiegato di età superiore a 50 anni.
  - **context** Company **inv**:
  - self.employee->select(age > 50)->notEmpty()
- Sintassi più generale per fare riferimento esplicito agli elementi della collezione

  collezione->select (v | espressione booleana con v)

  - **v è l'iteratore nella collezione**

  - **context** Company **inv**:
  - self.employee->select(p | p.age > 50)->notEmpty()

Filtraggio delle collezione: select (2)

- Sintassi più generale per fare riferimento esplicito agli elementi della collezione e ai loro tipi

  collezione->select (v : Type | espressione booleana con v)

  - **v è l'iteratore nella collezione**
  - **Gli elementi selezionati devono essere di tipo Type**
  - **può essere usato per selezionare oggetti con sottotipo specifico rispetto al tipo della collezione**

  - **context** Company **inv**:
  - self.employee->select(p : Person | p.age > 50)->notEmpty()

Filtraggio delle collezione: reject

- Stesse varianti sintattiche della select
- **collezione->reject (espressione booleana)**
- **collezione->reject (v : | espressione booleana con v)**
- **collezione->reject (v : Type | espressione booleana con v)**
- **Es. Tutti gli impiegati sono sposati (la collezione dei non sposati è vuota)**
  - **context** Company **inv**:
  - self.employee->reject(isMarried)->isEmpty()
- Select e reject sono definibili uno in termini dell'altro. Sono identiche le seguenti:
- **collezione->reject (v : Type | espressione booleana con v)**
- **collezione->select (v : Type | not (espressione booleana con v))**

Collezioni derivate da altre collezioni

- Select e reject servono a produrre una sottocollezione di una collezione data.
- Per creare invece una collezione derivata da una collezione data si usa l'operazione **collect**
- Può essere usata ad esempio per creare la collezione dei valori di una data proprietà di uan collezione di oggetti.
  - Es. La collezione di tutte le date di nascita degli impiegati
- **Sintassi:**
- **collezione->collect (espressione)**
- **collezione->collect (v | espressione con v)**
- **collezione->collect (v : Type | espressione con v)**
- Il tipo di valore dell'espressione è **Bag(Type')** dove Type' è il tipo dell'espressione **(possibili duplicati)**

Collezioni derivate da altre collezioni

- Es. La collezione di tutte le date di nascita degli impiegati
  - **context** Company **.....**
  - self.employee->collect(birthDate)
  - self.employee->collect(person | person.birthDate)
  - self.employee->collect(person : Person | person.birthDate)
- **Nell'esempio** self.employee->collect(birthDate) è una collezione di date di nascita (collezione Bag).
- Per la rimozione degli eventuali duplicati si può scrivere l'espressione
  - **self.employee->collect(birthDate)->asSet()**

Strutturazione del tipo di dato di una collect mediante ennuple

- Esempio.

context Person **def:**

attr statistics : Set(TupleType(company: Company, numEmployees: Integer,

wellpaidEmployees: Set(Person), totalSalary: Integer)) =

managedCompanies->collect(c |

Tuple { company: Company = c,

numEmployees: Integer = c.employee->size(),

wellpaidEmployees: Set(Person) = c.job->
select(salary>10000).employee->asSet(),

totalSalary: Integer = c.job.salary->sum()

}

)

Vincoli sulle collezioni

- A volte serve esprimere un vincolo su ogni elemento di una collezione, o che un vicolo sia soddisfatto da almeno un elemento della collezione
  - **Collezione->forAll(espressione-booleana)**
  - Valuta a true se tutti gli elementi della collezione soddisfano l'espressione boolena
  - Valuta a false altrimenti
  - **Collezione->exists (espressione-booleana)**
  - Valuta a true se almeno un elemento della collezione soddisfa l'espressione boolena
  - Valuta a false altrimenti
- **Le due espressioni hanno tipo booleano**

Vincolo **forAll** sulle collezioni

- Varianti sintattiche
  - **Collezione->forAll(espressione-booleana)**
  - **Collezione->forAll(v | espressione-booleana con v)**
  - **Collezione->forAll(v : Type | espressione-booleana con v)**
- Esempio: Tutti gli impiegati hanno età inferiore a 65 anni
  - **context** Company
  - **inv**: self.employee->forAll(p | p.age < 65)
- Si può usare più di un iteratore sulla collezione (iterazione sul prodotto cartesiano)
  - **context** Company
  - **inv**: self.employee->forAll(p1, p2 : Person | p1 <> p2 implies p1.forename <> p2.forename) (**stesso che scrivere**)
  - **inv**: self.employee->forAll(p1 : Person | forAll (p2 : Person | p1 <> p2 implies p1.forename <> p2.forename))

Vincolo **exists** sulle collezioni

- Varianti sintattiche
  - **Collezione->exists(espressione-booleana)**
  - **Collezione->exists(v | espressione-booleana con v)**
  - **Collezione->exists(v : Type | espressione-booleana con v)**
- **Esempio**: esiste un impiegato che ha nome 'Mario'
  - **context** Company
  - **inv**: self.employee->exists(p | p.forename = 'Mario')
-