OCL Notes

Object Constraint Language (OCL)

- A UML diagram (e.g., a class diagram) does not provide all relevants aspects of a specification
- It is necessary to describe additional constraints about the objects in the model
- Constraints specify invariant conditions that must hold for the system being modeled
- Constraints are often described in natural language and this always result in ambiguities
- Traditional formal languages allow to write unambiguous constraints, but they are difficult for the average system modeler
- OCL: Formal language used to express constraints, that remains easy to read and write

Object Constraint Language (OCL)

- **Pure expression language**: expressions **do not have side effet**
  - when an OCL expression is evaluated, it returns a value
  - its evaluation cannot alter the state of the corresponding executing system
  - an OCL expression can be used to specify a state change (e.g., in a post-condition)
- **Not a programming language**
  - it is not possible to write program logic or flow of control in OCL
  - cannot be used to invoke processes or activate non-query operations
- **Typed language**: each expression has a type
  - well-formed expressions must obey the type conformance rules of OCL
  - each classifier defined in a UML model represents a distinct OCL type
  - OCL includes a set of supplementary predefined types
- The evaluation of an OCL expression is **instantaneous**
  - the state of objects in a model cannot change during evaluation

Where to Use OCL

- To specify invariants on classes and types in the class model
- To specify type invariants for stereotypes
- To describe pre- and post-conditions on operations and methods
- To describe guards
- As a navigation language
- To specify constraints on operations
- OCL is used to specify the well-formedness rules of the UML metamodel

Comments and Infix Operators

Comments

- Denoted by `--`

```
-- this is a comment
```

Infix Operators

- Use of infix operators (e.g., `+`, `-`, `=`, `<`, ...) is allowed
- Expression `a + b` is conceptually equivalent to `a.+(b)`, i.e., invoking the `+` operation on `a` with `b` as parameter
- Infix operators defined for a type must have exactly one parameter

Context and Self

- All classifiers (types, classes, interfaces, associations, datatypes, ...) from an UML model are types in the OCL expressions that are attached to the model
- Each OCL expression is written in the context of an instance of a specific type

```
context Person
  ...
```

- Reserved word `self` is used to refer to the contextual instance
- If the context is `Person`, `self` referes to an instance of `Person`

Object and Properties

- All **properties** (attributes, association ends, methods and operations without side effects) defined on the types of a UML model can be used in OCL expressions
- The value of a property of an object defined in a class diagram is specified by a dot followed by the name of the property
- If the context is `Person`, `self.age` denotes the value of attribute `age` on the instance of `Person` identified by `self`
- The type of the expression is the type of attribute `age`, i.e., `Integer`
- If the context is `Company`, `self.stockPrice()` denotes the value of operation `stockPrice` on the instance identified by `self`
- Parentheses are mandatory for operations or methods, even if they do not have parameters

Invariants

- Determine a constraint that must be true for all instances of a type
- Value of attribute `noEmployees` in instances of `Company` must be less than or equal to 50

```
context Company inv:
  self.noEmployees <= 50
```

- Equivalent formulation with a `c` playing the role of `self`, and a name for the constraint

```
context c: Company inv SME:
  c.noEmployees <= 50
```

- The stock price of companies is greater than 0

```
context Company inv:
  self.stockPrice() > 0
```

Pre and Post conditions

- Constraints associated with an operation or other behavioral feature
- **Pre-condition**: Constraint assumed to be true **before** the operation is executed
- **Post-condition**: Constraint satisfied **after** the operation is executed
- Pre- and post-conditions associated to operation `income` in `Person`

```
context Person::income(): Integer
  pre:  self.age >= 18
  post: result < 5000
```

- `self` is an instance of the type which owns the operation or method
- `result` denotes the result of the operation, if any
- Type of `result` is the result type of the operation (`Integer` in the example)
- A name can be given to the pre- and post-conditions

```
context Person::income(): Integer
  pre adult:  self.age >= 18
  post resultOK: result < 5000
```

Previous values in Post conditions

- In a postcondition, the value of a property `p` is the value upon completion of the operation
- The value of `p` at the start of the operation is referred to as `p@pre`

```
context Person::birthDayHappens()
  post: age = age@pre + 1
```

- For operations, '`@pre`' is postfixed to the name, before the parameters

```
context Company::hireEmployee(p: Person)
post: employee = employee@pre->including(p) and
      stockPrice() = stockPrice@pre() + 10
```

- The '`@pre`' postfix is allowed only in postconditions
- Accessing properties of previous object values
  - `a.b@pre.c`: the new value of `c` of the old value of `b` of `a`
  - `a.b@pre.c@pre`: the old value of `c` of the old value of `b` of `a`

Body Expression

- Used to indicate the result of a query operation
- Income of a person is the sum of the salaries of her jobs

```
context Person::income(): Integer
body: self.job.salary->sum()
```

- Expression must conform to the result type of the operation
- Definition may be **recursive**: The right-hand side of the definition may refer to the operation being defined
- A method that obtains the direct and indirect descendants of a person

```
context Person::descendants(): Set
  body: result = self.children->union(
        self.children->collect(c | c.descendants()) )
```

- Pre-, and postconditions, and body expressions may be mixed together after one operation context

```
context Person::income(): Integer
pre: self.age >= 18
body: self.job.salary->sum()
post: result < 5000
```

Let Expression

- Allows to define a variable that can be used in a constraint

```
context Person inv:
  let numberJobs: Integer = self.job->count() in
  if isUnemployed then
    numberJobs = 0
  else
    numberJobs > 0
  endif
```

- A `let` expression is only known within its specific expression

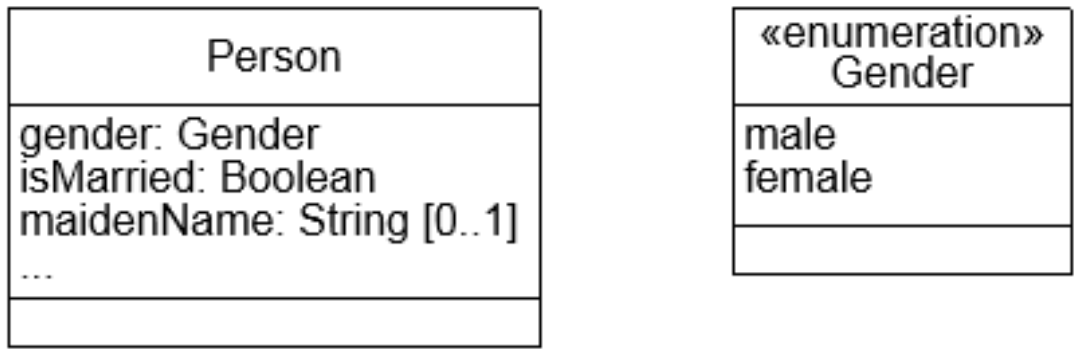
Definition Expression

- Enable to reuse variables or operations over multiple expressions
- Must be attached to a classifier and may only contain variable and/or operation definitions

```
context Person
def: name: String = self.firstName.concat(' ').concat(lastName)
def: hasTitle(t: String): Boolean = self.job->exists(title = t)
```

- Names of the attributes/operations in a `def` expression must not conflict with the names of attributes/association ends/operations of the classifier

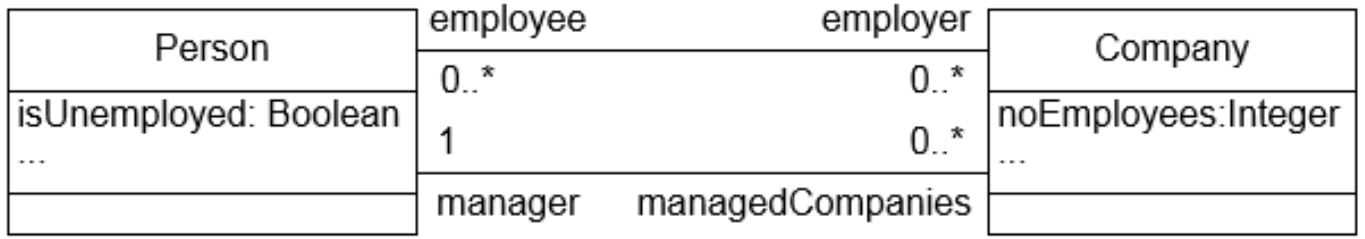
Enumeration Types



- Define a number of literals that are the possible values of the enumeration
- An enumeration value is referred as in `Gender::female`
- Only married women can have a maiden name

```
context Person inv:
  self.maidenName <> '' implies
  self.gender = Gender::female and self.isMarried = true
```

Navigating Associations



- From an object, an association is navigated using the opposite role name

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

- Value of expression depends on maximal multiplicity of the association end
  - 1: value is an object
  - *: value is a `Set` of objects (an `OrderedSet` if association is `{ordered}`)
- If role name is missing, the name of the type at the association end starting with a lowercase character is used (provided it is not ambiguous)

```
context Person
  inv: self.bank.balance >= 0
```

Navigating Associations

- When multiplicity is at most one, association can be used as a single object or as a set containing a single object
- `self.manager` is an object of type `Person`

```
context Company inv:
  self.manager.age > 40
```

- `self.manager` as a set

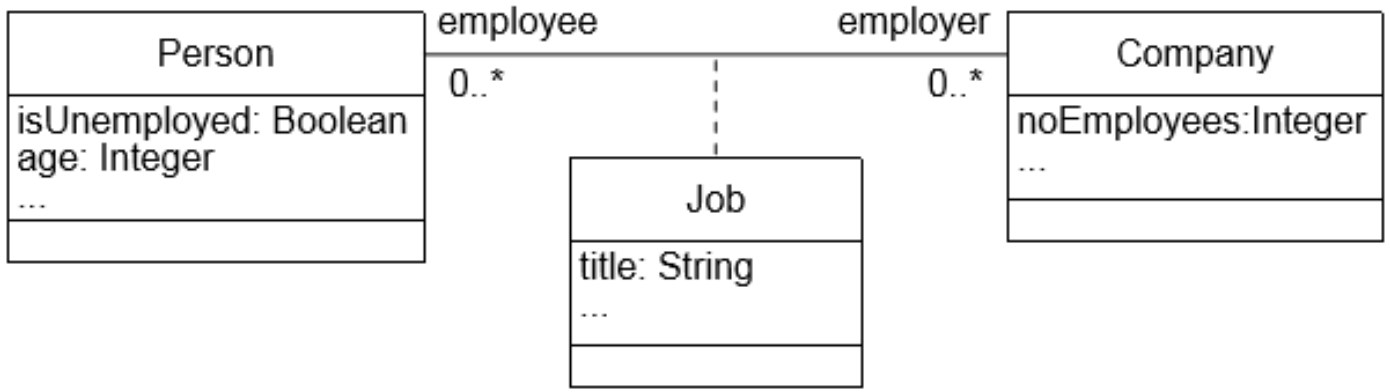
```
context Company inv:
  self.manager->size() = 1
```

- For optional associations, it is useful to check whether there is an object or not when navigating the association

```
context Person inv:
  self.wife->notEmpty() implies self.gender = Gender::male and
  self.husband->notEmpty() implies self.gender = Gender::female
```

- OCL expressions are read and evaluated from left to right

Association Class



- For navigating **to** an association class: a dot and the name of the association class starting with a lowercase character is used

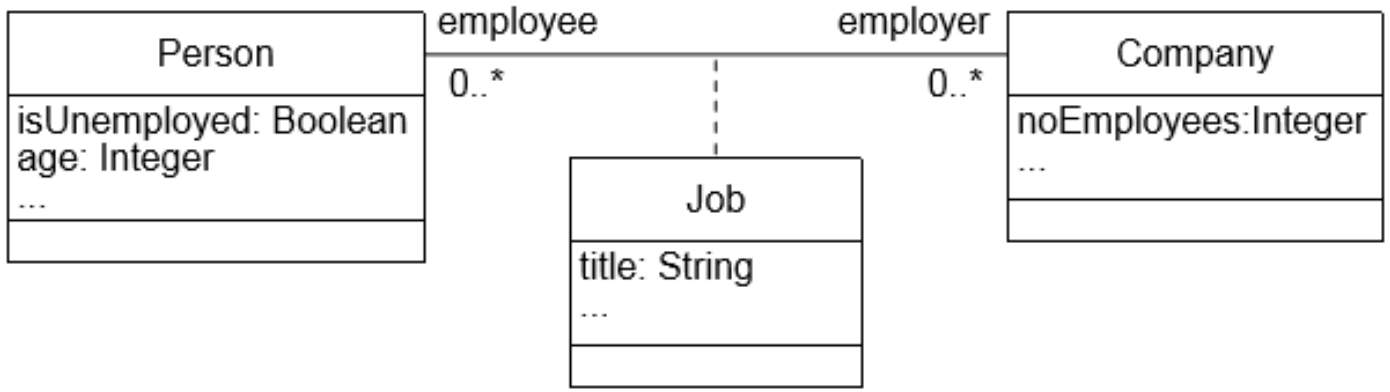
```
context Person
  inv: self.isUnemployed = false implies self.job->size() >= 1
```

- For navigating **from** an association class to the related objects: a dot and the role names at the association ends is used

```
context Job
  inv: self.employer.noEmployees >= 1
  inv: self.employee.age >= 18
```

- This always results in exactly **one** object

Association Class



- For navigating **to** an association class: a dot and the name of the association class starting with a lowercase character is used

```
context Person
  inv: self.isUnemployed = false implies self.job->size() >= 1
```

- For navigating **from** an association class to the related objects: a dot and the role names at the association ends is used

```
context Job
  inv: self.employer.noEmployees >= 1
  inv: self.employee.age >= 18
```

- This always results in exactly **one** object