3. Object Constraint Language

3.1 Why Object Constraint Language (OCL)?

- Graphical model not enough for a precise and unambiguous specification; need to describe additional constraints about the objects
- Disadvantage of traditional formal languages: difficult for the average business or system modeler; OCL developed to fill this gap
- OCL has its roots in the Syntropy method
- OCL is a pure expression language; every OCL expression is side effect free

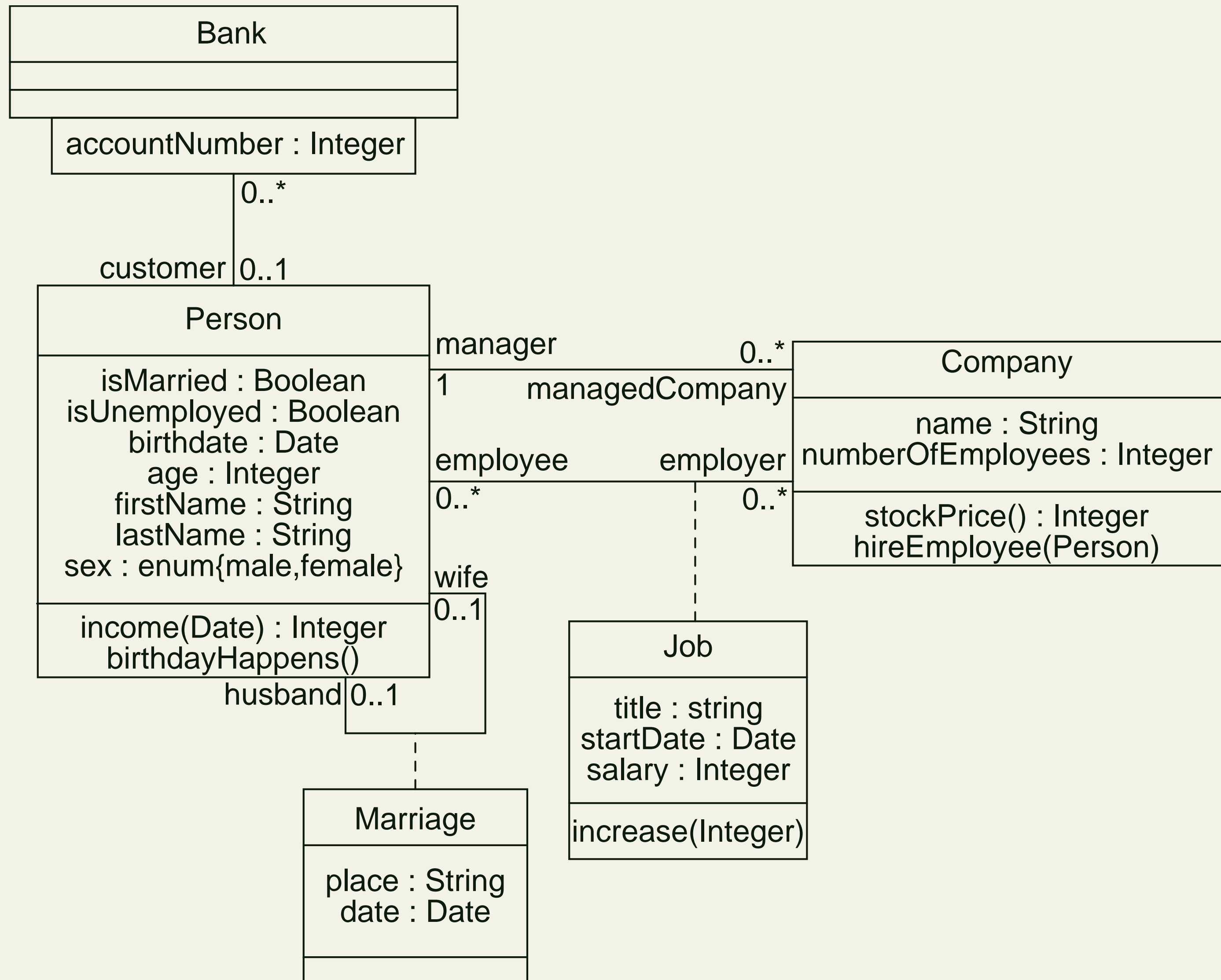
Why OCL? (cnt'd)

- OCL is not a programming language; not possible to write program logic or flow control in OCL
- OCL is a typed language, each OCL expression has a type; types within OCL can be any kind of Classifier
- Implementation issues are out of scope and cannot be expressed in OCL

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
- Within the UML Semantics chapter, OCL used in the well-formedness rules as invariants on the meta-classes in the abstract syntax; also used to define additional operations

3.2 OCL by Example



Induced Properties for Example Class Diagram (Part One)

Properties induced by role names

```
employee: Company -> Set(Person)
employer: Person -> Set(Company)
manager : Company -> Set(Person)
manager : Company -> Person  -- multipl. 1 on manager side
managedCompany : Person -> Set(Company)

wife : Person -> Set(Person)
wife : Person -> Person -- multipl. 0..1 on wife side
husband : Person -> Set(Person)
husband : Person -> Person -- multipl. 0..1 on husband side

customer : Bank -> Set(Person)
-- multipl. 0..1 on customer side
-- qualifier attribute accountNumber
customer : Bank x Integer -> Person
bank : Person -> Set(Bank)
```

Induced Properties for Example Class Diagram (Part Two)

Properties induced by association classes

```
job : Person -> Set(Job)
job : Company -> Set(Job)
employee : Job -> Person
employer : Job -> Company

marriage[wife] : Person -> Set(Marriage)
-- multipl. 0..1 on wife side
marriage[wife] : Person -> Marriage
marriage[husband] : Person -> Set(Marriage)
-- multipl. 0..1 on husband side
marriage[husband] : Person -> Marriage
wife : Marriage -> Person
husband : Marriage -> Person
```

Attributes and Operations

Attributes, pre- and postconditions, operations, navigation

```
context Person inv: -- attribute
  self.age>0

context Job::increase( perCent : Integer )
  pre:  0<perCent and perCent<=100 -- precondition
  post: salary=salary@pre*(1+perCent/100) -- postcondition

context Company inv: -- operation
  self.stockPrice()>0

context Company
  inv: self.manager.isUnemployed = false
  -- navigation on single object: 'object'.'object-property'
  inv: self.employee->notEmpty
  -- navigation on object set: 'object-set' -> 'set-property'
```

Set-Valued Terms

Set-valued terms

```
context Person inv: -- size property
  self.employer->size < 3

context Person inv: -- isEmpty property
  self.employer->isEmpty = false

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

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

Association Classes and Qualifier

Connectives, association classes, qualifier

```
context Person inv: -- logical connective *implies*
  self.wife->notEmpty implies self.wife.sex = #female

context Person inv: -- logical connective *and*
  self.wife->notEmpty implies self.wife.age >= 18 and
  self.husband->notEmpty implies self.husband.age >= 18

context Job -- properties of association classes
  inv: self.employer.numberOfEmployees >= 1
  inv: self.employee.age > 21

context Bank inv: -- unqualified use of customer
  self.customer->notEmpty

context Bank inv: -- qualified use of customer
  self.customer[8764423].age >= 18
```

oclType and oclIsTypeOf

```
context Person inv: -- oclType
  self.oclType = Person

context Person -- oclIsTypeOf
  inv: self.oclIsTypeOf(Person)
  inv: not(self.oclIsTypeOf(Company))
```

Collections

```
Set{ 1, 2, 5, 88 }
Set{ 'apple', 'orange', 'strawberry' }
Sequence{ 1, 3, 45, 2, 3 }
Sequence{ 'ape', 'nut' }
Bag{1, 3, 4, 3, 5 }
Sequence{ 1..(5+4) } =
Sequence{ 1.. 9 } =
Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }
Set{ Set{1, 2}, Set{3, 4}, Set{5, 6} }
= Set{ 1, 2, 3, 4, 5, 6 } -- flattening
```

Pre- and Postconditions, and Queries

Pre- and Postconditions

```
-- @pre, postconditions
context Person::birthdayHappens()
  post: age = age@pre+1

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

Select and reject expressions

```
context Company inv:
  self.employee->select(age > 50)->notEmpty
context Company inv:
  self.employee->select(p | p.age > 50)->notEmpty
context Company inv:
  self.employee.select(p:Person |
    p.age > 50)->notEmpty
context Company inv:
  self.employee->reject( isMarried )->isEmpty
```

Queries

Collect expressions

```
context Company inv:
  self.employee->collect(birthDate)->notEmpty
context Company inv:
  self.employee->collect( p | p.birthDate )->notEmpty
context Company inv:
  self.employee->collect( p:Person | p.birthDate)->notEmpty
context Company inv:
  self.employee->collect(birthDate)->asSet->notEmpty
context Company inv:
  self.employee->collect(birthDate)->notEmpty
context Company inv:
  self.employee.birthdate->notEmpty -- shorthand
context Person inv:
  self.job->collect(salary)->sum > 12000
context Person inv:
  self.job.salary->sum > 12000 -- shorthand
```

Quantification (Universal quantifier)

```
context Company
  inv: self.employee->forAll( firstName = 'Jack' )
  inv: self.employee->forAll( p | p.firstName = 'Jack' )
  inv:
    self.employee->forAll( p:Person | p.firstName = 'Jack' )

context Company
  inv:self.employee->forAll( e1,e2 |
    e1<>e2 implies e1.firstName<>e2.firstName )
  inv:self.employee->forAll( e1,e2:Person |
    e1<>e2 implies e1.firstName<>e2.firstName)
  inv:
    self.employee->forAll( e1 | self.employee->forAll ( e2 |
    e1<>e2 implies e1.firstName<>e2.firstName ) )

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

Quantification (Existential quantifier)

```
context Company inv:
  self.employee->exists( firstName = 'Jack')

context Company inv:
  self.employee->exists( p | p.firstName = 'Jack')

context Company inv:
  self.employee->exists( p:Person | p.firstName = 'Jack')
```

Iterate Expressions

```
<collec>->iterate( <elem> : <collec-elem-type>;
          <accu> : <res-type> = <init-expr> |
          <expr-with-elem-and-accu> /*:<res-type>*/)
```

Evaluation in Java-like pseudocode

```
<res-type> iterate() {
  <collec-elem-type> <elem>;
  <res-type> <accu> = <init-expr>;
  for (Iterator i = <collec>.iterator();
                    i.hasNext();) {
    <elem> = i.next();
    <accu> = <expr-with-elem-and-accu>;
  };
  return <accu>;
}
```

Example for Iterate

Collection operations collect, select, reject, forAll, and exists can be expressed in terms of iterate

```
employeeSet->collect(e : Employee | e.birthDate)
= employeeSet->iterate( e : Employee;
          birthDateBag : Bag(Date) = Bag{} |
          birthDateBag->including(e.birthDate) )

employeeSet->exists( p:Person | p.firstName='Jack' )
= employeeSet->iterate( e : Employee;
          jackExists : Bool = false |
          jackExists or e.firstName='Jack' )
```

More Examples

[1] Married people are of age $>=$ 18

```
context person inv:
  self.wife->notEmpty implies self.wife.age >= 18 and
  self.husband->notEmpty implies self.husband.age >= 18
```

[2] a company has at most 50 employees

```
context Company inv:
  self.employee->size <= 50
```

[3] A marriage is between a female (wife) and male (husband)

```
context marriage inv:
  self.wife.sex = #female and
  self.husband.sex = #male
```

[4] A person can not both have a wife and a husband

```
context person inv:
not ((self.wife->size=1) and (self.husband->size=1))
```

3.3 General Remarks on OCL
OCL Grammar(1)

(roughly one production per line)

```
<ocl-formula>                                    -- truth values
  ( <ocl-formula> and <ocl-formula> )
  ( <ocl-formula> or <ocl-formula> )
  ( <ocl-formula> implies <ocl-formula> )
  ( <ocl-formula> xor <ocl-formula> )
  ( not <ocl-formula> )
```

OCL Grammar (2)

```
<ocl-term>->forAll( <variable | <ocl-formula> )
<ocl-term>->exists( <variable | <ocl-formula> )
<ocl-term>->includes( <ocl-term> )
<ocl-term>->isEmpty
( <ocl-term> = <ocl-term> )
( <ocl-term> <> <ocl-term> )
( <ocl-term> > <ocl-term> )
( <ocl-term> >= <ocl-term> )
( <ocl-term> < <ocl-term> )
( <ocl-term> <= <ocl-term> )
??? <association>(<ocl-term>,...,<ocl-term>) ???
```

OCL Grammar (3)

```
<ocl-term>                                  -- arbitrary values
  <constant>
  <variable>
  <class-name>.allInstances
  <ocl-term>.<attribute>
  <ocl-term>.<rolename>
  <ocl-term>->size
  <ocl-term>->select( <variable> | <ocl-formula> )
```

General Remarks on OCL Syntax (1)

- `->` always applied to multi-valued terms (sets, sequences, bags)
- `<class-name>.allInstances` may be abbreviated to `<class-name>`
- if an attribute `att:s` for class `B` and variable `b` for `B` objects are given, then `b.att` is term of sort `s`

General Remarks on OCL Syntax (2)

- if an association without restricting cardinalities, role names `ra`, `rb` and variables `a`, `b` are given, then `a.rb` and `b.ra` are terms of sort `Set(B)` and `Set(a)` respectively.

  if the cardinality is `0..1` or `1`, then `a.rb` and `b.ra` are (additional) terms of sort `B` or `A`, respectively; there are also implicit role names

General Remarks on OCL Syntax (3)

- if a generalization `s1<s2` is given, a term of sort `s1` may be used where a term of `s2` is expected
- parenthesis my be omitted (as long as uniqueness is guaranteed)
- further variations (e.g. `forAll( <ocl-formula> )` instead of `forAll( <variable> | <ocl-formula> )`)
- further possibilities and functions (e.g. `reject` or `iterate` ; for details see OCL description