Introduction to the Object Constraint Language (OCL)

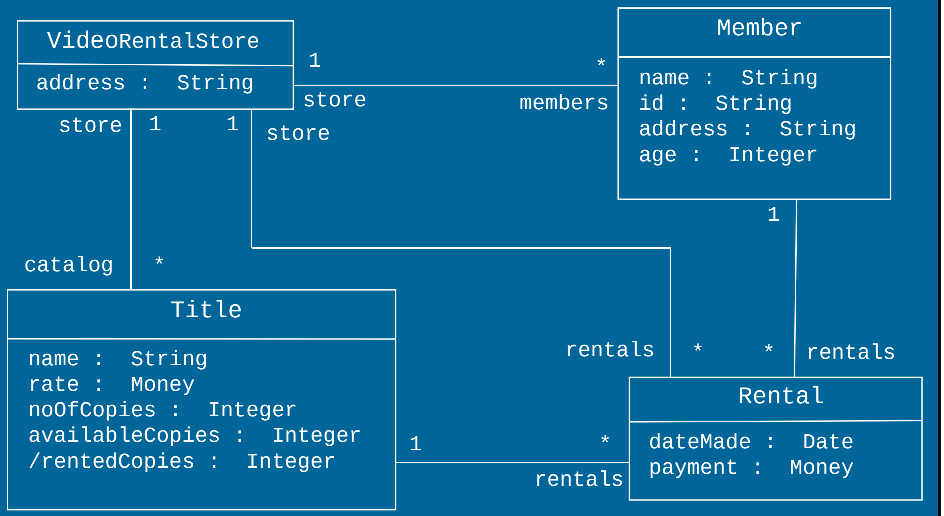
Motivation for Formal Model-Based Specification

- UML (Unified Modeling Language) 2.5 is a (semi-formal) modeling language proposed by the OMG (Object Management Group)[1].
- UML is the de facto *industry standard* notation to model software analysis and design artifacts.
- UML Superstructure specification 2.5[2] describes 14 (semi-) formal diagram types, e.g., class and use-case diagrams.
- Limits:
  - *not precise* and automatic verification hardly possible
  - *weak code generation capabilities* (usually only code skeletons, not fully functional code)

---

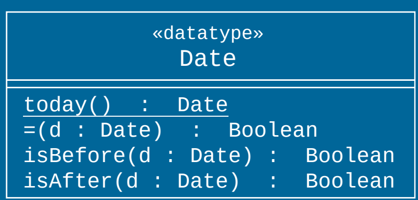
[1]http://www.omg.org/
[2]http://www.omg.org/spec/UML/2.5/

Example: Video Store Class Diagram



3

Auxiliary Types

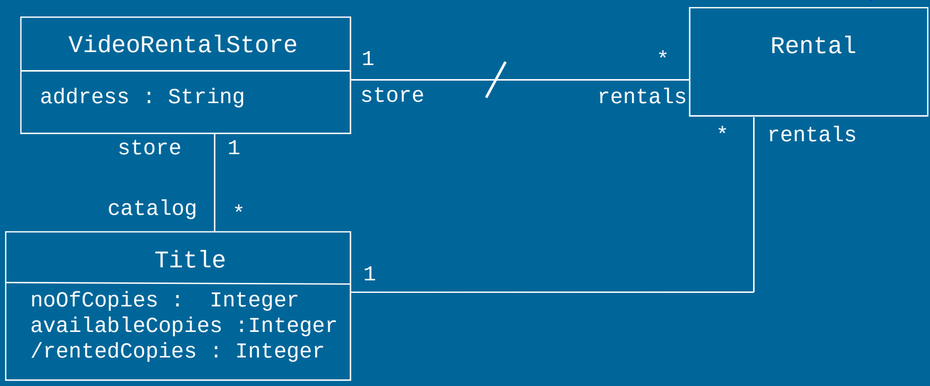


Date is a utility class for representing date values

Money is a class for representing currency values

4

Derived Attributes/Associations

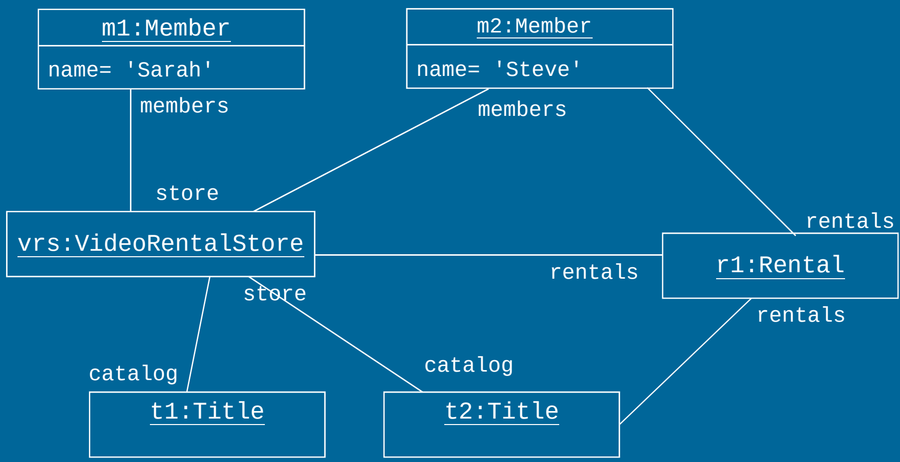


- Derived attributes and associations are sometimes added to help readers understand the diagram
- Derived attributes and associations are marked with "/"

Instance Diagram (snapshot)

- An instance diagram is used to show the state of a system at a point in time
- An object is represented as a rectangle
- If two objects are associated, a line is drawn between the objects, which is called a *link*
- Interesting properties of objects are described inside the rectangle and called attributes

Instance Diagram from Video Store



Consistency between Class and Instance Diagrams

- The class diagram and instance diagrams must be consistent
- Some simple tests can be performed to verify consistency:
  - Every link in the instance diagram must correspond to an association in the type diagram, e.g.:

There is no association between Title and Member

  - An object in the instance diagram can not be linked up to more or fewer objects of another type than specified by the multiplicity of association in the type diagram. E.g.:

The rental must be linked to exactly one member

Expressions in the Class Diagram

- The vocabulary defined in the class diagram provides the means for forming precise expressions (formal queries)
  - Example: *what are all the rentals a title has*?
- To check that a query is supported by the diagram:
  - The diagram can be informally inspected
  - or a formal query can be written
- A formal definition of "*what are all the rentals a title has*?" would be written like this (OCL notation):

```
context VideoRentalStore::allRentals(t:Title):Set(Rental)
def: t.rentals
```

- Expressions are essential when high confidence in the model is required
- The OCL will be used for writing precise expressions

Formal Models

- (Semi-formal) visual models can be enriched with formal specification of
  - *state constraints* (with invariants)
  - *operation semantics* (with pre- and post-conditions)
- UML defines a language that can be used with this goal: Object Constraint Language (OCL)
- Advantages:
  - UML diagrams enriched with OCL expressions lead to precise specifications that can be verified automatically
  - formal specifications remove the ambiguity that characterizes informal specifications
  - formal specifications can be automatically verified
  - tools exist that generate code and assertions in Java from OCL specifications of state invariants and operations' pre and post conditions

The Object Constraint Language (OCL)

- A formal specification language for specifying additional constraints on UML models
- A precise and unambiguous language that can be read and understood by object modelers and software engineers
- OCL is not a programming language; therefore, it is not possible to write program logic or flow control in OCL.
- A *purely* declarative language – i.e. it has *no side-effects* (in other words its expressions do not change the state of the system; it describes *what* rather than *how*)
- A language that is based on predicate calculus and on textual syntax rather symbolic syntax (easy to read!)
- A strongly typed language – every expression must have a type

OCL: Basic Concepts

- OCL is a rich language that offers a predefined mechanism for:
  - Retrieving the values of an object
  - Navigating through a set of related objects
  - Iterating over a collection of objects (e.g. `select, forAll, exists`)
- OCL includes a predefined standard library: set of types + operations on them
  - Primitive types: `Boolean, Integer, Real, String`
  - Collection types: `Set, Bag, OrderedSet and Sequence`
  - Tuple types
- Classes from UML model are also types in OCL, e.g. `Member, Title, enumeration` classes, etc

Primitive Types

Type	Values	Operators and operations
`Boolean`	`true, false`	`=, <>, and, or, xor, not, implies, if-then-else-endif`
`Integer`	`-3,0,6, ...`	`=,<>,>,<, >=, <=, *,+,-(unary), -(binary), /(real), abs(), max(b),min(b), mod(b),div(b)`
`Real`	`-1.2,0.0,...`	`=,<>,>,<,>=,<=,*,+,-(unary), - (binary),/, abs(), max(b), min(b),round(),floor()`
`String`	`'hello world'`	`=, <>,size(), concat(s2), substring(lower, upper) (1<=lower<=upper<=size), toReal(), toInteger()`

Collection and Tuple Types

Description	Syntax	Examples
Abstract collection of elements of type T	`Collection(T)`	
Unordered collection, no duplicates	`Set(T)`	`Set{1,2}`
Ordered collection, duplicates allowed	`Sequence(T)`	`Sequence{1,2,1}` `Sequence{1..4} (same as Sequence{1,2,3,4})`
Ordered collection, no duplicates	`OrderedSet(T)`	`OrderedSet {2, 1}`
Unordered collection, duplicates allowed	`Bag(T)`	`Bag{1, 1, 2}`
Tuple (with named parts)	`Tuple(field1:T1, ...,fieldn: Tn)`	`Tuple{age:Integer = 5, name: String = 'Joe‘}` `Tuple{name='Joe',age=5}`

Note 1: They are value types: “=” and “<>” compare values and not references.

Note 2: Tuple components can be accessed with “.” as in “`t1.name`”

Operations on Collection(T)

Operation	Description
`size():Integer`	the number of elements in the collection
`count(o:T) : Integer`	the number of occurrences of object `o` in the collection
`includes(o:T): Boolean`	true if object `o` is an element of the collection
`includesAll(c :Collection(T)) : Boolean`	true if all the element in collection `c` are present in the current collection
`excludes(o : T) : Boolean`	true if object `o` is not an element of the collection
`excludesAll(c:Collection(T)): Boolean`	true if all the element in collection `c` are not present in the current collection
`isEmpty() : Boolean`	true if the collection contains no elements
`notEmpty(): Boolean`	true if the collection contains one or more elements
`sum() : T`	the addition of all the elements in the collection

Note: Operations on collections are applied with "`->`" and not "`.`"

Operations on Set(T)

Operation	Description
`=(s:Set(T)):Boolean`	Do `self` and `s` contain the same elements?
`union(s: Set(T)): Set(T)`	The union of `self` and `s`.
`union(b: Bag(T)): Bag(T)`	The union of `self` and bag `b`.
`intersection(s:Set(T)):Set(T)`	The intersection of `self` and `s`.
`intersection(b:Bag(T)):Set(T)`	The intersection of `self` and `b`.
`-(s: Set(T)) : Set(T)`	The elements of `self`, which are not in `s`.
`including(object: T): Set(T)`	The set containing all elements of `self` plus `object`.
`excluding(object: T): Set(T)`	The set containing all elements of `self` minus `object`.
`symmetricDifference(s:Set(T)) :Set(T)`	The set containing all the elements that are in `self` or `s`, but not in both.

Operations on Set(T)

Operation	Description
`flatten() : Set(T2)`	If `T` is a collection type, the result is the set with all the elements of all the elements of `self`; otherwise, the result is `self`.
`asOrderedSet():OrderedSet(T)`	`OrderedSet` with elements from `self` in undefined order.
`asSequence(): Sequence(T)`	Sequence with elements from `self` in undefined order.
`asBag(): Bag(T)`	Bag with all the elements from `self`.

Operations on Bag(T)

Operation	Description
`=(b: Bag(T)) : Boolean`	True if `self` and `bag` contain the same elements, the same number of times.
`union(b: Bag(T)): Bag(T)`	The union of `self` and `b`.
`union(s: Set(T)): Bag(T)`	The union of `self` and set `s`.
`intersection(b:Bag(T)):Bag(T)`	The intersection of `self` and `b`.
`intersection(s: Set(T)):Set(T)`	The intersection of `self` and `s`.
`including(object:T):Bag(T)`	The bag with all elements of `self` plus `object`.
`excluding(object:T):Bag(T)`	The bag with all elements of `self` minus `object`.

Operations on Bag(T)

Operation	Description
`flatten(): Bag(T2)`	If `T` is a collection type: bag with all the elements of all the elements of `self`; otherwise: `self`.
`asOrderedSet():OrderedSet(T)`	`OrderedSet` with elements from self in undefined order, without duplicates.
`asSequence():Sequence(T)`	Seq. with elements from `self` in undefined order.
`asSet(): Set(T)`	`Set` with elements from `self`, without duplicates.

Operations on Sequence(T)

Operation	Description
`=(s:Sequence(T)):Boolean`	True if `self` contains the same elements as `s`, in the same order.
`union(s:Sequence(T)): Sequence(T)`	The sequence consisting of all elements in `self`, followed by all elements in `s`.
`flatten():Sequence(T2)`	If `T` is a collection type, the result is the set with all the elements of all the elements of self; otherwise, it's `self`.
`append(object:T): Sequence(T)`	The sequence with all elements of self, followed by object.
`prepend(obj:T): Sequence(T)`	The sequence with object, followed by all elements in `self`.
`insertAt(index:Integer, object:T): Sequence(T)`	The sequence consisting of `self` with object inserted at position `index` (`1<=index<=size+1`)
`subSequence(lower:Integer ,upper:Integer): Sequence(T)`	The sub-sequence of `self` starting at index `lower`, up to and including index `upper` (`1<=lower<=upper<=size`)

Operations on Sequence(T)

Operation	Description
at(i:Integer): T	The i-th element of self (1<=i<=size)
indexOf(object:T): Integer	The index of object in self.
first(): T	The first element in self.
last(): T	The last element in self.
including(object:T): Sequence(T)	The sequence containing all elements of self plus object added as last element
excluding(object: T): Sequence(T)	The sequence containing all elements of self apart from all occurrences of object.
asBag(): Bag(T)	The Bag containing all the elements from self, including duplicates.
asSet(): Set(T)	The Set containing all the elements from self, with duplicates removed.
asOrderedSet(): OrderedSet(T)	An OrderedSet that contains all the elements from self, in the same order, with duplicates removed.

Operations on OrderedSet(T)

Operation	Description
append(object:T): OrderedSet(T)	The set of elements, consisting of all elements of self, followed by object.
prepend(object:T): OrderedSet(T)	The sequence consisting of object, followed by all elements in self.
insertAt(index:Integer, object:T): OrderedSet(T)	The set consisting of self with object inserted at position index.
subOrderedSet(lower: Integer, upper:Integer): OrderedSet(T)	The sub-set of self starting at number lower, up to and including element number upper (1<=lower<=upper<=size).
at(i:Integer): T	The i-th element of self (1<=i<=size).
indexOf(object:T): Integer	The index of object in the sequence.
first(): T	The first element in self.
last(): T	The last element in self.

Operations Defined in OclAny

Operation	Description
=(object2:OclAny):Boolean	True if self is the same object as object2.
<>(object2:OclAny):Boolean	True if self is a different object from object2.
oclIsNew():Boolean	Only used in a postcondition. True if self was created during the operation execution.
oclAsType(t:OclType):T	Cast (type conversion) operation. Useful for downcast.
oclIsTypeOf(t:OclType):Boolean	True if self is of type t.
oclIsKindOf(t:OclType):Boolean	True if self is of type t or a subtype of t.
oclIsInState(s:OclState): Boolean	True if self is in state s.
oclIsUndefined(): Boolean	True if self is equal to null or invalid.
oclIsInvalid(): Boolean	True if self is equal to invalid.
allInstances(): Set(T)	Static operation that returns all instances of a classifier.

Operations Defined in OclMessage

Operation	Description
hasReturned(): Boolean	True if type of template parameter is an operation call, and the called operation has returned a value.
result()	Returns the result of the called operation, if type of template parameter is an operation call, and the called operation has returned a value.
isSignalSent():Boolean	Returns true if the OclMessage represents the sending of a UML Signal.
isOperationCall():Boolean	Returns true if the OclMessage represents the sending of a UML Operation call.

OCL Type Hierarchy

- Type conformance relation ≤
  - `OclVoid, OclInvalid ≤ T` for all types `T`
  - `Integer ≤ Real`
  - `T ≤ T'  ⇒  C(T)≤ C(T')` for collection type `C`
  - `C(T) ≤ Collection(T)` for collection type `C`
  - `B ≤ OclAny` for all primitives and classifiers `B`
  - Generalization hierarchy from UML model
- Examples
  - `String ≤ OclAny`
  - `Set(Integer) ≤ Set(Real)`

Quantification Expressions: forAll

- The `forAll` operation can be used to specify that a certain condition must hold for all elements of a collection.
- The `forAll` operation takes an OCL expression as parameter.
- This operation is used when there already is a (sub)set of all instances of a class, and the elements of that (sub)set should be checked.
- The result of the operation is a boolean value:
  - true if the expression evaluates to true for all elements in the collection
  - otherwise false
- Syntax: `c->forAll(e : T | exp)`
- Example:
  `Set{1,2,3}->forAll(n:Integer | n > 3)`
  evaluates to false

Quantification Expressions: exists

- The `exists` operation can be used to specify that a certain condition must hold for at least one element of a collection.
- The `exists` operation takes an OCL expression as parameter.
- This operation is used when there already is a (sub)set of all instances of a class, and some elements of that (sub)set should be checked.
- The result of the operation is a boolean value:
  - true if the expression evaluates to true for some elements in the collection
  - otherwise false
- Syntax: `c->exists(e : T | exp)`
- Example:

  `Set{-1,5,8}->exists(n:Integer | n+1>10)`

  evaluates to false

The select Operation

- The `select` operation takes an OCL expression as parameter.
- The result of select is a subcollection of the collection on which it is applied.
- `select` selects all elements from the collection for which the expression evaluates to *true*.
- Syntax: `c->select(e : T | exp)`
- Example

  `Set{1,2,3}->select(n:Integer |n >= 2)`

  evaluates to `Set{2, 3}`

The `reject` Operation

- The `reject` operation is analogous to `select`.
- `reject` selects all elements from the collection for which the expression evaluates to *false*.
- Syntax: `c->reject(e : T | exp)`
- Example

  `Set{1,2,3}->reject(n:Integer | n >= 2)`
  evaluates to `Set{1}`

The `collect` Operation

- The operation can be used to collect attribute values
- The operation also can be used to build a new collection from the objects held by association ends.
- Syntax: `c->collect(e : T |  exp)`

  collection of elements with `exp` applied to each element of `c`
- Example

```
Set{1,2,3}->collect(n:Integer |  n + 1)
members->collect(name)
```

The collect Operation

- The resulting collection contains different objects from the original collection.
- When the source collection is a `Set` the resulting collection is not a `Set` but a `Bag`.
- If the source collection is a `Sequence` or an `OrderedSet`, the resulting collection is a `Sequence`.
- The dot notation is an abbreviation for applying the `collect` operation:
  - `members.name`
  - `catalog.rentals`

Other Iterator Expressions

- Uniqueness

  `c->isUnique(e : T | exp)`

  true if `exp` has a unique value for all elements in the collection

- Sorting

  `c->sortedBy(e : T | exp)`

  Sequence of all the elements in the collection in the order specified (< must be defined on `exp`)

- Iteration

  `c->iterate( e : T1; result : T2 = exp1 | exp2)`

  Iterates over all the elements of the collection and accumulates the result in the variable `result`.

  - Example:

    `Set{1,2,3}->iterate(i:Integer;sum:Integer=0 | sum+i)` evaluates to `6`.

  - All other collection operations can be defined in terms of `iterate`

Constraints in OCL

- Constraints that can be expressed include:
  - *invariants* on classes,
  - *preconditions* and *postconditions* of operations/methods
- An *invariant* on class is an assertion about the class which must *always* be true for all instances of the class in any public and visible state.
- A *precondition* of an operation/method is an assertion that states what must be true in order to meaningfully invoke the operation/method.
- A *postcondition* of an operation/method is an assertion that states what must hold after execution of the operation/method.

Constraints in OCL

- An *invariant* in OCL is expressed as follows:

```
context ClassName
inv: OCL-invariant
```

  - `context`: keyword indicating the context of the invariant `ClassName`
  - `inv`: keyword indicating an invariant
  - `OCL-invariant`: boolean expression describing the invariant

- An operation specification in OCL is expressed as follows:

```
context ClassName::operationName(arg:Type):Type
pre:  OCL-precondition
post: OCL-postcondition
```

  - `pre`: keyword indicating a precondition
  - `OCL-precondition`: boolean expression describing the precondition
  - `post`: keyword indicating a postcondition
  - `OCL-postcondition`: boolean expression describing the postcondition

Other Constraints in OCL

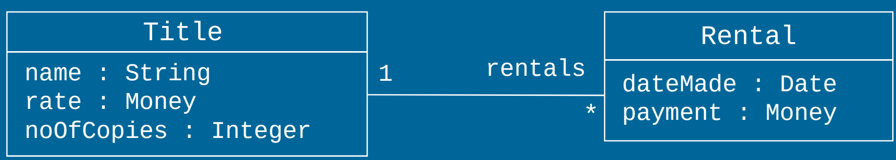
- An *initial value constraint* is a rule that states the initial value for an attribute or association end. It is expressed as:

```
context className::propertyName
init: initialValue
```

- A *derivation rule* states how a derived attribute or association end is calculated from other properties. It is expressed as:

```
context className::propertyName
derive: derivedValue
```

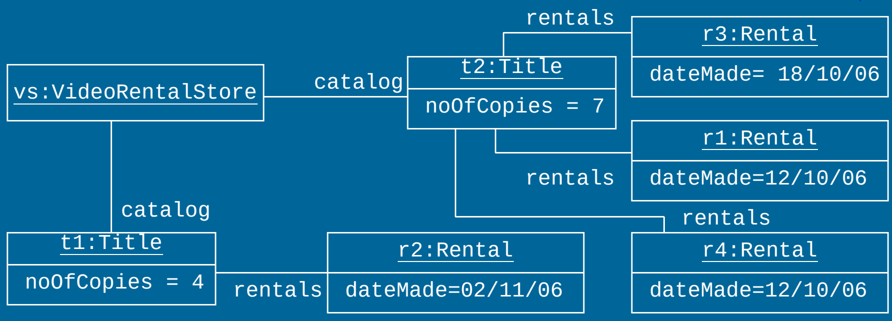

Expressions from the Video Store



`t : Title, r : Rental`

- `t.name : String` -- name of title `t`
- `t.rate : Money` -- rate of title `t`
- `t.noOfCopies : Integer` -- number of copies for `t`
- `t.rentals : Set(Rental)` -- all rentals for title `t`
- `r.Title : Title` -- title for rental `r`
- `t.rentals->size(): Integer` -- number of rentals for `t`
- `t.rentals.dateMade:Bag(Date)` -- dates for rentals of `t`

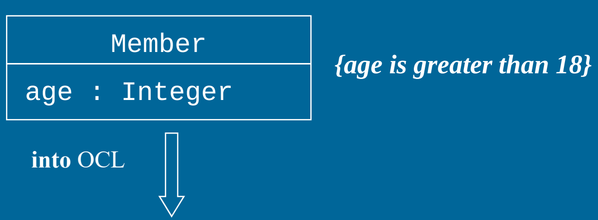
Evaluating Expressions




```
t1.noOfCopies  =  4
r2.Title  =  t1
t2.rentals  =  Set{r1, r3, r4}
t2.rentals->size()  =  3
t2.rentals.dateMade = Bag{18/10/06, 12/10/06, 12/10/06}
```


Invariants on Attributes

- The class to which the invariant refers is the context of the invariant.
- It is followed by a boolean expression that states the invariant.
- All attributes of the context class may be used in this invariant.



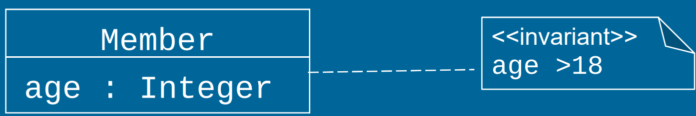

```
context   Member
inv :  self.age  > 18
```

Usage of `self`

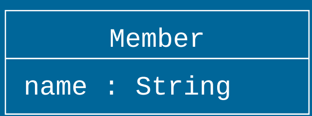
- The following invariant notations are equivalent:

```
context  Member
inv: self.age > 18
```

```
context Member
inv: age > 18
```



Invariants on Attributes



{name is not the empty string}

into OCL

```
context   Member
inv :  self.name <> ''
```

Invariants on Attributes

```
Rental
-------------------
dateMade : Date
dueDate : Date
```

{date made is before due date}

into OCL

```
context  Rental
inv:  dateMade.isBefore(dueDate)
```

- If the type of the attribute is a class, the attributes or query operations defined on that class can be used to write the invariant (using a dot notation).
- Query operation:
  An operation that does not change the value of any attributes.

Enumeration Types

- Enumeration uses datatype followed by :: and the value

```
Passenger
-------------------------------
age : Integer
needsAssistance : Assistance
```

```
«enumeration»
Assistance
-------------------
wheelChair
fullAssistance
noAssistance
```

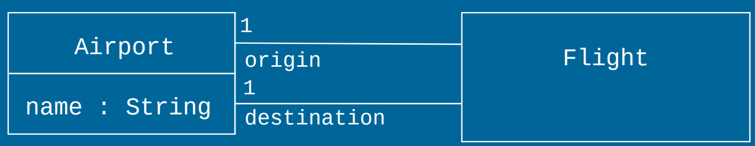
- The constraint that each passenger with an age above 95 needs assistance by a wheelchair, can be expressed as follows.

```
context Passenger
inv : self.age > 95 implies
        self.needsAssistance = Assistance::wheelChair
```

Associations and Navigation

- Every association is a navigation path.
- The context of the expression is the starting point.
- Role names (or association ends) are used to identify the navigated associations.

Associations and Navigation: Example




```
context Flight
inv:  origin <> destination
```

- The origin of each flight is unequal to the destination.

```
context Flight
inv:  origin.name  =  'Gatwick'
```

- The origin of each flight is Gatwick.

Associations and Navigation

- Often associations are one-to-many or many-to-many, which means that constraints on a collection of objects are necessary.
- OCL expressions either state a fact about all objects in the collection or states facts about the collection itself.

Invariant with the size operation

{A member cannot have more than 10 rentals }

into OCL ⇩

```
context Member
inv : self.rentals->size() <= 10
```

Invariant with the select operation

```
VideoRentalStore ---------- catalog * ---------- Title
                                                 noOfCopies : Integer
```

{number of titles with no copies is less than 5 }

into OCL ⇩

```
context VideoRentalStore
inv : self.catalog->select(t:Title|t.noOfCopies=0)
                                          -> size() < 5
```

This can also be written as:

```
context VideoRentalStore
inv : self.catalog->select(noOfCopies =0)-> size()< 5
```


Invariant with forAll Operation

```
VideoRentalStore ---------- members * ---------- Member
                                                 id : String
```

{each member of the video store has a unique id}

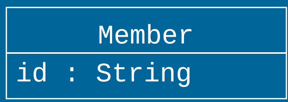
into OCL ⇩

```
context VideoRentalStore
inv : self.members->forAll(m1,m2:Member |
                         m1.id=m2.id implies m1=m2)
```

Alternatively:

```
context VideoRentalStore
inv :  self.members->isunique(id)
```

Using alIInstances Operation

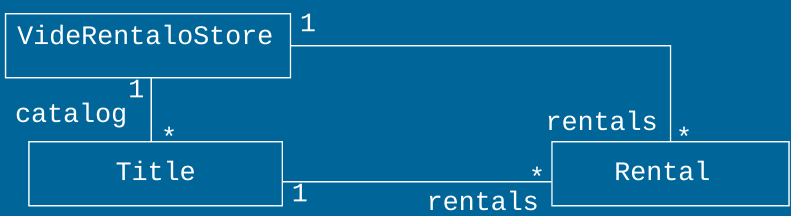


{each member of the video store has a unique id}

into OCL

```
context VideoRentalStore
inv : Member.allInstances->isUnique(id)
```


Invariant with collect Operation

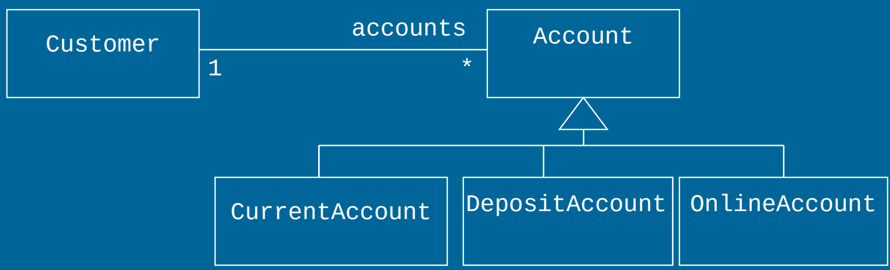


{the rentals of the video store are the rentals of all titles in the store}

into OCL

```
context  VideoRentalStore
inv:self.rentals=self.catalog.rentals->asSet()
```

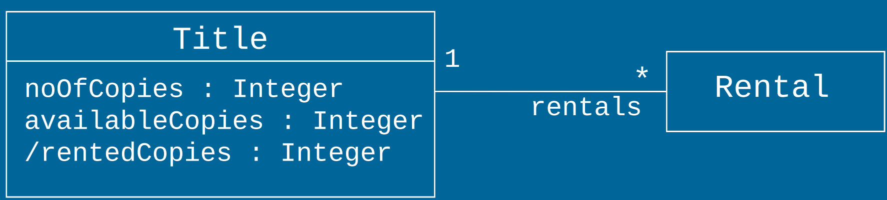
Invariant with oclIsKindOf Operation




```
context Customer
inv:  accounts->select(a  |
         a.oclIsKindOf(CurrentAccount))->size() >= 1
```

Meaning: Every customer has a least one current account

Initial values and Derivation Rules



-- initially the number of copies is zero

```
context Title::noOfCopies
init:  0
```

-- initially the set of rentals is empty

```
context Title::rentals
init:  set{}
```

-- derivation rule for the number of rented copies

```
context Title::rentedCopies
derive:  noOfCopies  -  availableCopies
```

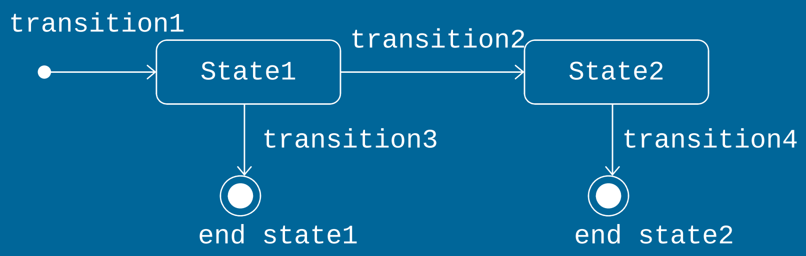
Context of an Invariant

- If the invariant restricts the value of attribute of a type, then that type is a candidate.
- If the invariant restricts the value of attributes of more than one type, the types containing any of the attributes are candidates.
- If a class is responsible for maintaining the constraint that class should be the context.
- Any invariant should navigate through the smallest possible number of associations.
- Choose an invariant which simple to read and write.

State Models

- Objects have states:
  - `Title:`
    - `AvaialbleForRent`
    - `UnavailableForRent`
  - Member:
    - `Active`
    - `Banned`
- A state model specifies:
  - the states an object may have
  - the order in which objects change states
  - events that can cause the object to change state
    e.g. a return event changes state from `UnavaialbleForRent` to `availableForRent`

State Models Notation

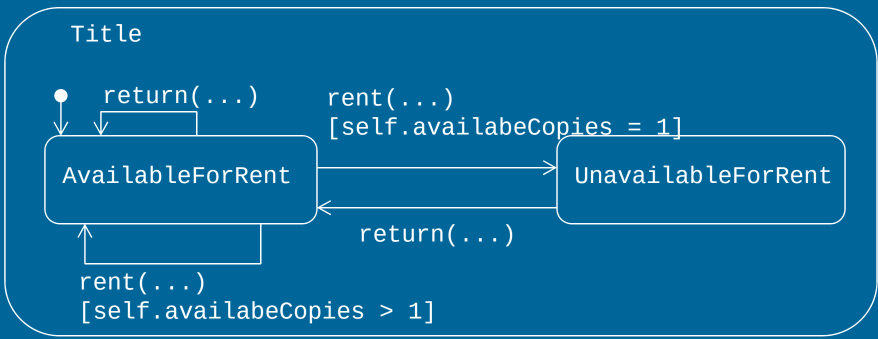


- Notation:
  - The states are shown as
  - The possible transition path are indicated as
  - End states are shown as
  - The initial state is indicated with

Rules for State Diagrams

- The states in a simple state diagram are mutually exclusive - an object can be in only one state at a time
- Every state must be related to attributes and associations on a type diagram
- We should be able to define the relationship between states, attributes and associations using invariants
- Every transition should map to a system operation

State Model for Title



Dictionary

Type	States	Description
Title	AvailableForRent UnavailableForRent	Title has copies available for rent All copies of title are rented

Connecting State models to Type Model

```
context Title
inv: self.oclInState(AvailableForRent)  =
                self.availableCopies > 0

inv: self.oclInState(UnavailableForRent)  =
                self.availableCopies = 0
```

Operation Specifications

- An operation can be specified using a *precondition* and *postcondition* pair. Such pair is called a *contract*.
- A *precondition* states the condition under which it is appropriate to invoke the operation.
- A *postcondition* states the conditions that must hold after the operation is complete, under the assumption that the precondition holds.
- OCL will be used for specifying operations.

Specification of rent: Precondition

The precondition of operation `rent` is informally specified as follows:

```
context
  VideoRentalStore::rent(m:Member,t:Title,
                                    p:Money,d : Date)
  -- rents a copy of title t to member m with payment p and date d
pre:      -- m is member of the store
          -- and t is a title of the store
          -- and t has copies available for rent
          -- and m is not currently renting a copy of t
          -- and the payment p is sufficient
```

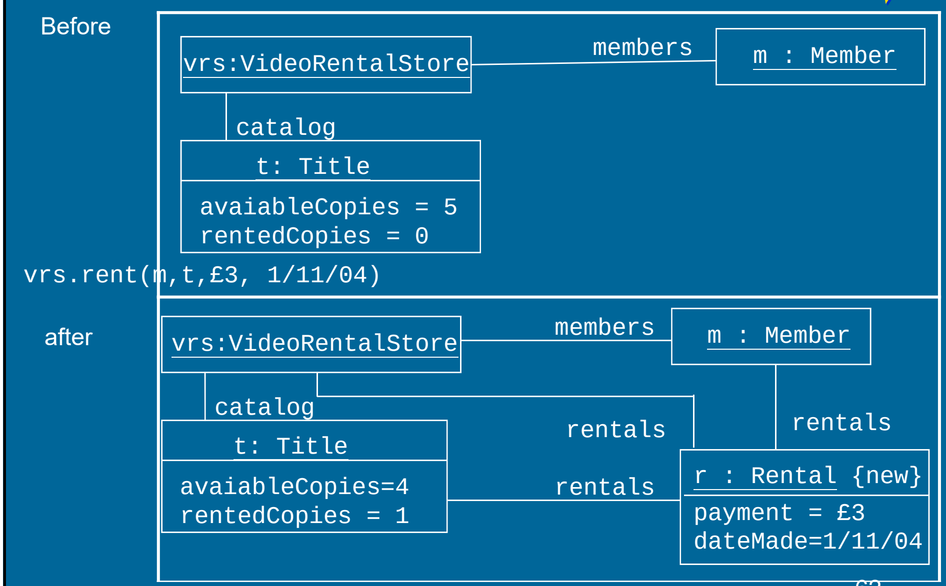
Specification of rent: Postcondition

The postcondition of operation rent is informally specified as follows:

```
VideoRentalStore::rent(m:Member,t:Title,
                                        p:Money,d : Date)
```

post: -- a new rental object was created
-- and is associated with the current video rental store
-- and is linked to member m
-- and is linked to title t
-- and the payment is equal to p
-- and its made at date is equal to d
-- and the number of available copies of t is reduced by 1
-- and the number of rented copies of t is increased by 1

Illustration of rent



Specification of rent: Precondition

The formal precondition of rent is given as follows:

```
VideoRentalStore::rent(m:Member,t:Title,
                          p:Money,d:Date)
```

-- rents a copy of title t to member m with payment p and date d

```
pre:   self.members->includes(m)
       and self.catalog->includes(t)
       and t.availableCopies > 0
       and m.rentals.title->excludes(t)
       and t.rate = p
```


Specification of rent: Postcondition

The formal postcondition of `rent` is given as follows:

```
VideoRentalStore::rent(m:Member,t:Title,
                          p:Money,d:Date)

post: self.rentals->exists(r:Rental|
                         r.oclIsNew()
                         and  r.member = m
                         and   r.title = t
                         and  r.dateMade = d
                         and  r.payment = p)
    and t.availableCopies =t.availableCopies@pre-1
    and self.rentals->size()=self.rentals@pre->size()+1
```

Query Operation Specification: Example

- The operation `getMembers` returns all the members of a video rental store. The following is a formal specification:

```
context VideoRentalStore::getMembers():Set(Member)
-- returns all the members of the current video rental store
pre: true
post: result  =  self.members
```

- If the precondition is `true` the operation can be invoked in any state. In such case the precondition can be omitted
- The variable `result` is used to hold the result of the operation

Query Operation Specification: Example

- The operation `membersOver50` returns those members over the age of 50. The following is a formal specification:

```
context VideoRentalStore::
            membersOver50():Set(Member)
pre:  true
post: result = self.members->select(age>50)
```

- With omitted precondition:

```
context VideoRentalStore::
                 membersOver50():Set(Member)

post: result = self.members->select(age>50)
```

Query Operation Specification: Example

- The operation `getMember` returns the member given the member's id. The following is a formal specification:

```
context VideoRentalStore::
                getMember(i:String):Member
-- returns the member with the id equal to i
pre: members->exists(m:Member | m.id=i)
post: result.id = i
```


Specification of **getNames()**

- The operation `getNames` returns a list of members names of a video rental store. The following is a formal specification:

```
context VideoRentalStore::
                getNames(): Sequence(String)
-- returns all the members of the current video rental store
pre:  true
post: result = self.members.name->asSequence()
```

Operation Specifications

- Steps for specifying an operation:
  - provide an informal description of the operation together with its signature and a precondition and postcondition
  - use instance diagrams to illustrate the behaviour of the operation
  - translate the precondition and postcondition to OCL using the language provided by the type diagram and OCL operations

Summary

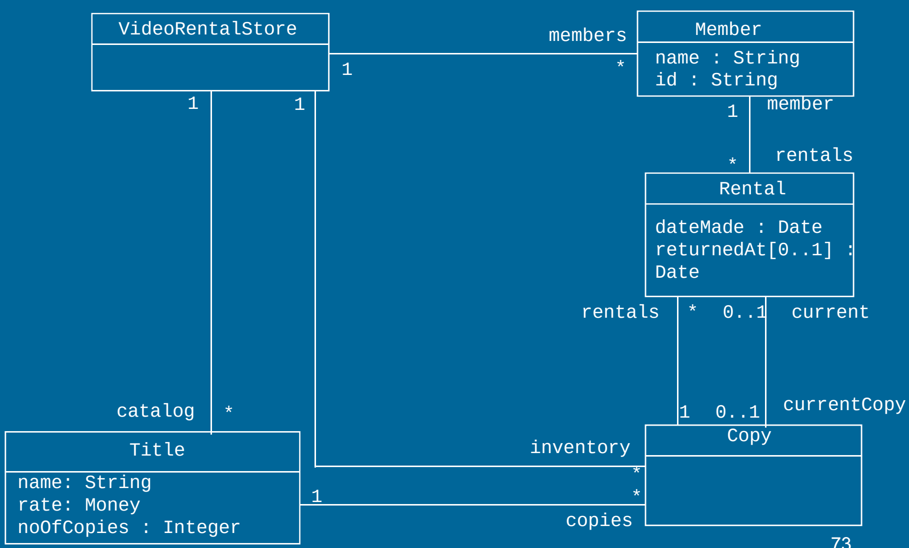
- **The OCL has been introduced as a specification language for:**
  - **Specifying additional constraints on the properties of objects in the form of invariants.**
  - **Specifying the behaviour of operations in terms of preconditions and postconditions.**
- **A declarative language**

Operation Design

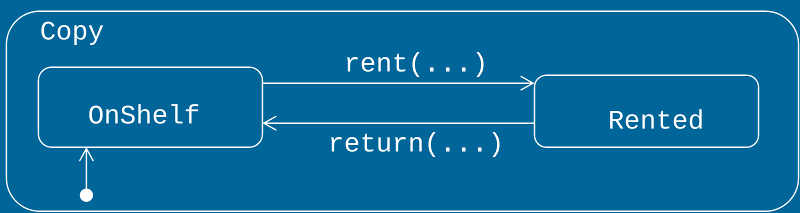
Objective

- To show how an operation specification can be refined at the design stage
- The obtained design can then be mapped into code

Class Diagram for Video Store [includes Copy class]



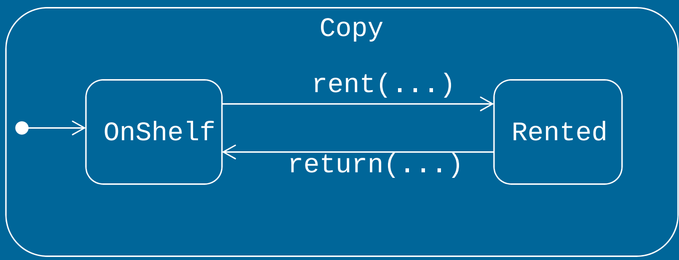
State Model for Copy



Dictionary

Type	States	Description
Copy	OnShelf	the copy is on the shelf and available for rent
	Rented	the copy is rented

States and Type Model




```
context Copy
inv: self.oclInState(OnShelf) = self.current->isEmpty
inv: self.oclInState(Rented) = self.current->notEmpty
```


Specification of rent

- The operation `rent` is informally specified as follows:

`context VideoRentalStore::`

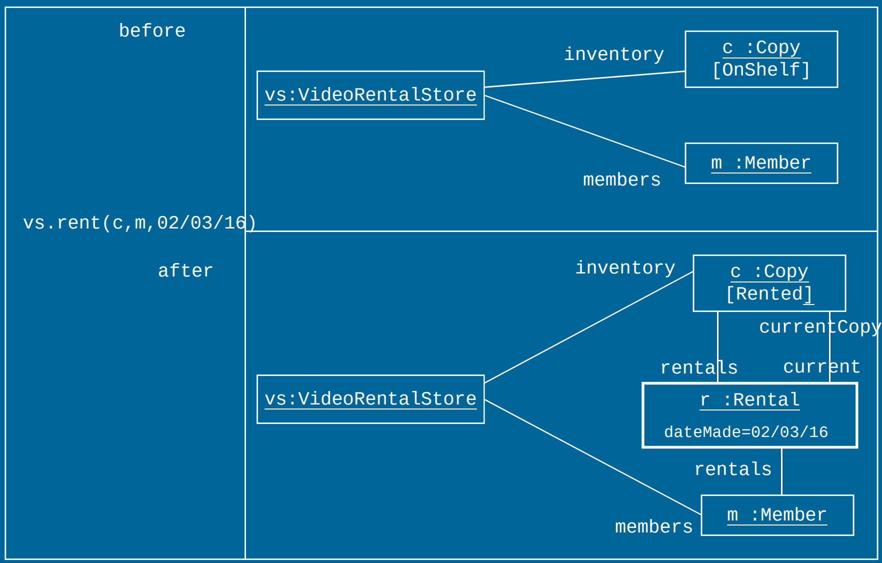
`rent(c : Copy, m : Member, out : Date)`

-- rents a copy `c` from the video store to member `m`

pre: the copy *c* is in the inventory of the store
and *m* is member of the store
and the copy *c* is on shelf

post: a new rental object is created
and is associated with member *m*
and associated with copy *c*
and the rented at date is set to *out*
and the copy is rented

Specification of rent



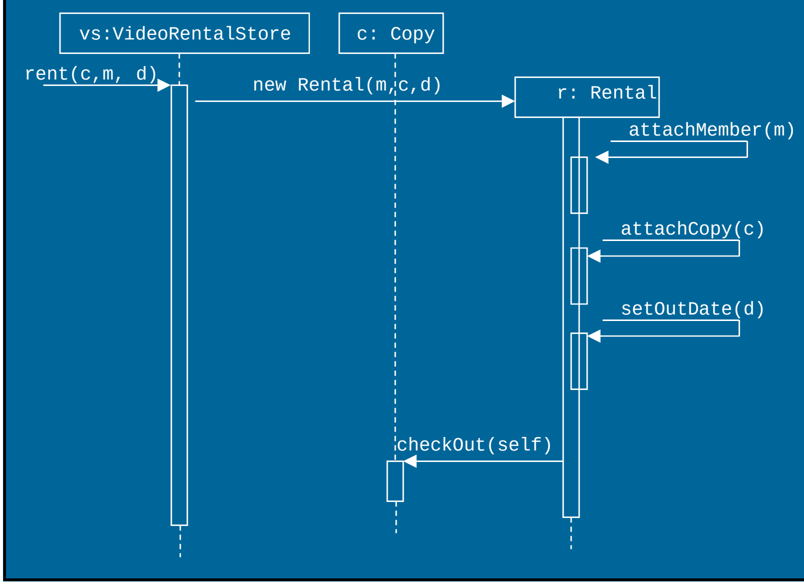
77

Specification of rent

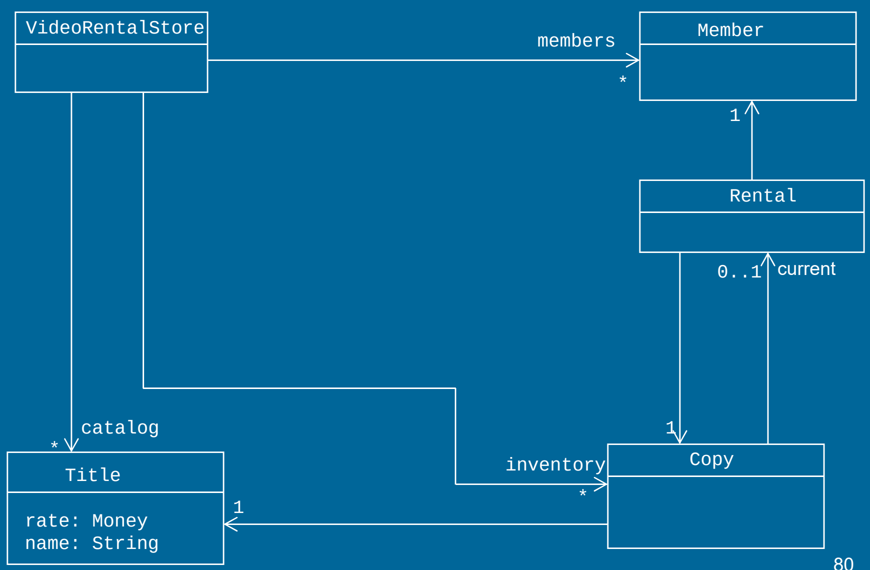
```
context VideoRentalStore::
          rent(c : Copy, m: Member, out : Date)
pre:  -- m is one of the members of the store
      self.members->includes(m)
      -- c is part of the inventory of the store
      and self.inventory->includes(c)
      -- c is on shelf
      and c.oclInState(OnShelf)
post: -- a new rental object is created
   Rental.allInstances->exists(r:Rental| r.oclIsNew()
      -- which is a rental for member m and copy c
      and r.member = m
      and r.copy = c
      and c.current = r
      -- the rented at date is set to out
      r.datemade = out
      -- and copy c is rented
      and c.oclInState(Rented)
```

78

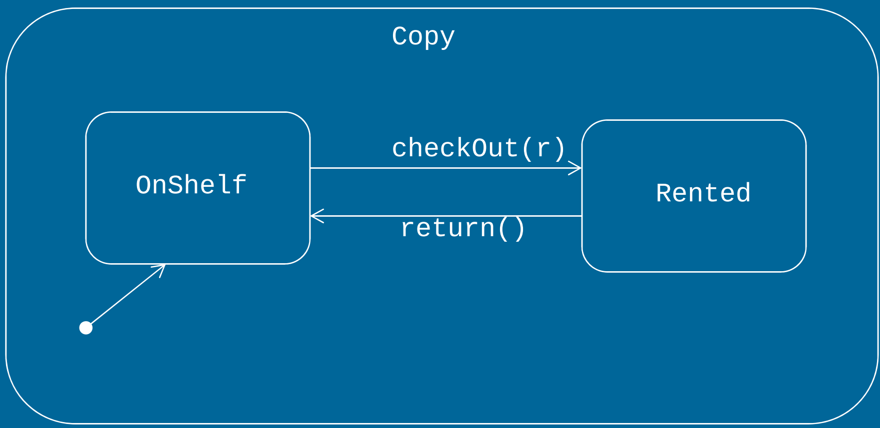
Operation Design: rent



Design Class Diagram



State Diagram for Copy



Operation Specification

- **context** Rental::attachMember(m : Member)
  -- sets the member of the rental to m
  pre: true
  post: self.member = m
- **context** Rental::attachCopy(c:Copy)
  -- sets the copy of the rental to c
  pre: true
  post: self.copy = c
- **context** Copy::checkOut(r : Rental)
  -- sets the current rental of copy to r
  pre: true
  post: self.current = r
- **context** Rental::setOutDate(d : Date)
  -- sets the out date of the rental
  pre: true
  post: self.dateMade = d

Operation Specification

- **context** `Rental::Rental(m:Member,c:Copy, d: Date)`
  -- creates a rental object and sets its member and copy to
  -- m and c respectively

  pre: `true`

  post: -- a new rental object is created

```
Rental.allInstances->exists(r:Rental| r.oclIsNew()
        -- for member m
        and r.member = m
        -- and attached to copy c
        and r.copy = c
        -- and current loan of copy c is set to r
        and r.copy.current = r
        -- and rented at date is set to d
        and dateMade = d
```


Deriving the Specification of Rental [Design]

- `Rental(m, c, d)`
  - `r.attachMember(m)`
    post: `r.member = m`
  - `r.attachCopy(c)`
    post: `r.copy = c`
  - `r.setOutDate( d : Date)`
    post: `r.dateMade = d`
  - `(r.copy).checkOut(r)`
    `post: r.copy.current = r`

- The postcondition of `Rental` is the conjunction of the postconditions of the above operations

Conformance Checking

- We need to show that the operations introduced at the design level conform to the operations at the specification level.
- For the video store system, we need to show that the postcondition of the operation `VideoRentalStore::rent(c,m,d)` at the design stage implies the postcondition of the `VideoRentalStore::rent(c,m,d)` at the specification stage.

Conformance Checking

```
post:(Design)
      Rental.allInstances->
            exists(r : Rental | r.oclIsNew()
      and r.member = m
      and r.copy = c
      and r.dateMade = d
      and r.copy.current = r
```

⇩ **implies**

```
Post:(Specification)
      Rental.allInstances->
            exists(r : Rental | r.oclIsNew()
      and r.member = m
      and r.copy = c
      and c.current = r
      and r.dateMade = d
      and c.oclInState(Rented)
```

The implication follows quite easily since `r.copy=c` implies `c.current=r`

Mapping Design to Code

```
public class Rental {
     Date dateMade;
     Member member;
     Copy copy;

     public Rental(Member m, Copy c, Date d) {
             attachMember(m);
             attachCopy(c);
             setOutDate(d);
             c.checkOut(this);
     }
     public attachMemebr(Member m) {
              member = m;
     }
     public attachCopy(Copy c) {
              copy = c;
     }
     public setOutDate(Date d){
              dateMade = d;
     }
    … }
```


Mapping Design to Code

```
public class Copy {
     Rental current;
     Title title;

     public checkOut(Rental r) {
              current = r;
     }
     …
{

public class VideoRentalStore {
     List<Member> members;

     public rent(Copy c, Member m, Date d) {
              Rental r = new Rental(c,m,d);
     }
     …
{
```

Summary

- We showed how an operation specification is refined into a design that conforms to the specification
- The design can be mapped to code by mapping the design class diagram and sequence diagrams.