ID2207 - Modern Methods in Software Engineering

Tutorial 4

September 2025

Object Constraint Language OCL

Remember:
A constraint is a rule that allows you to specify some limits on model elements. It can also be defined as a restriction on one or more values of (part of) an object-oriented model or system. We express them using OCL language. Types of constraints are: invariants, pre-conditions and post-conditions

Invariants:
The invariant is a property that must remain true throughout the life of an object. The constraint applies to all instances of this class. The invariants should be true at the start of the task, and remain true at the end of the task.

Example of invariants from our "CarSharer" business that we analyzed in tutorial 2 and 1:

- The age of a car sharer to be over 21
- The car sharer must have a valid driving licence.

How to use OCL to express them:

context CarSharer **inv**:
age >21

The keyword **context** introduces a model element to which the invariant applies (the class name in this example), and **inv** introduces the constraint as an invariant.

Pre-Conditions:
A *pre-condition* is something that must be true before a particular part of the system is executed. Pre-conditions can be used to perform checks before an operation is executed. This is a way of preventing the system from entering illegal states. If a pre-condition is violated, then an exception should be raised and the execution of the operation should be stopped.

Example: If the user enters an illegal ZIP or post code while registering the car, an error message should be shown on the screen to ask the user to re-enter the information.
Another example would be making sure that enough money is in a savings account before making a withdrawal, this would be written in OCL as:

context Account::withdraw(amount:Integer)
pre: balance >= amount

Post-Conditions:
A *post-condition* is something that must be true after a particular part of the system is executed. For example, after registering a car sharer, the car sharer's details must be recorded, and a request for payment must be sent to the car sharer to his or her credit card or bank account.
Example: an account's balance must be reduced by the amount withdrawn after a withdrawal. It can be written in OCL as:

context Account::withdraw(amount:Integer)
post: balance = balance@pre - amount

It is sometimes necessary to refer to the initial value of a property that is being changed by an operation as post condition. This is done using @pre on the property.

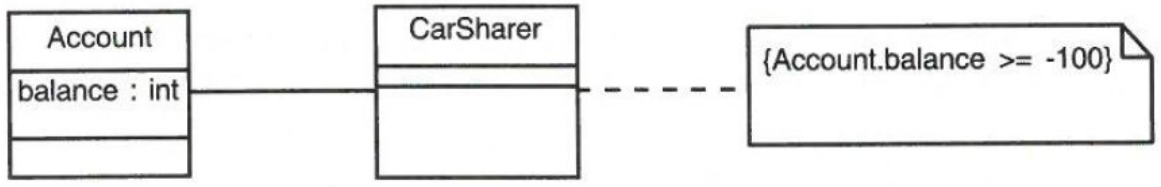
So, OCL is an expression language that provides a clear and unambiguous description of constraints (invariants, pre-conditions and post-conditions). It is the developer's job to interpret OCL expressions and convert them into meaningful actions in a programming language. All OCL statements must take place within a context. The context must be a class for invariants, or an operation for pre-conditions and post-conditions.

context ClassName **inv**:

context ClassName::operationName(param1: Type1, ….):ReturnType
Pre: constraint
Post: constraint

Navigation:
Within a context, the navigation to an element of a model is by the left-to-right traversal of a path separated by full-stops. For example, to refer to a car sharer's account balance, with an association, and placing an invariant on the car sharer that the balance should be no more than 100$ overdrawn, we could write the following:

context CarSharer **inv**:
Account.balance >= -100



The role name on the association is used to navigate to the class. When the role name is missing, then the name of the type, which is Account in the previous example is the default role name. If the expression is complex, and you want to avoid ambiguity, the word **self** can be used to refer to the instance of the class that is named as the context. So the following expression is equivalent to the previous one:

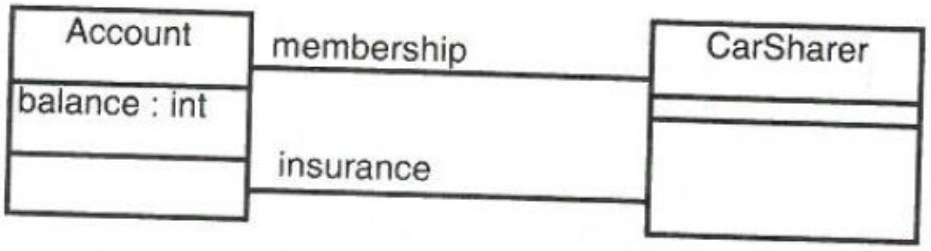
context CarSharer **inv**:
self.Account.balance >= -100

This would be used in cases where another instance of the same class is involved in a different role, and it is necessary to distinguish between the two. Alternatively, it is possible to refer to a named instance of the class as in the following:

context c.CarSharer **inv**:
C.Account.balance >= -100

For associations, the role name can be part of the path. Suppose there are two types of accounts: membership and insurance and you want to indicate that the insurance account must not be overdrawn by more than 500$ we would write the following:

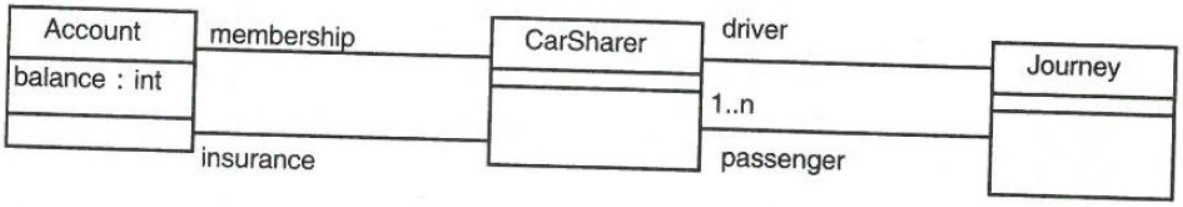
context CarSharer **inv**:
Insurance.balance >= -500



It is not necessary here to indicate the type name (Account) as it is unambiguous. The role name that is used in navigation is the one at the far end of the association, that is at the account end for paths running from CarSharer

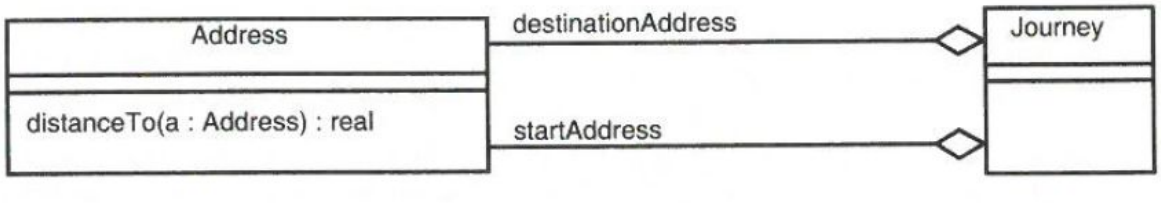
It is possible to navigate across a number of associations. So, if there is a condition that the driver in a journey has paid his or her insurance, we can express it as the following:

context Journey **inv**:
Driver.insurance.balance >= 0



Another example: if we want to express that the distance of any journey is two miles or more, we construct an operation on an address that allows us to calculate the distance to any other address, and we express it as the following:

context Journey **inv**:
startAddress.distanceTo(self.destinationAddress) >= 2.0



Types of collections used in OCL:

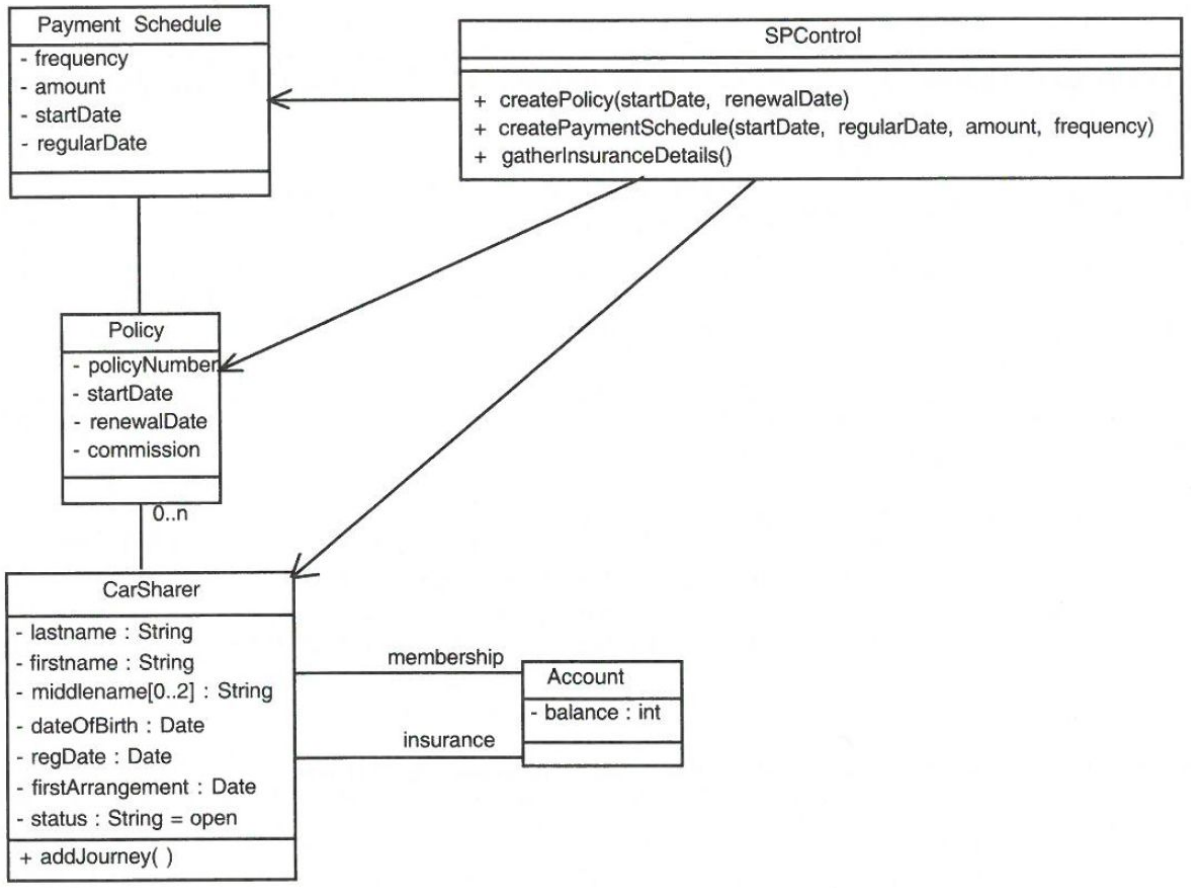
- A **Set** consists of distinct instances, there are no repetitions in the set. Sets have no ordering.
- A **bag** is an unordered collection that allows repetitions.
- A **Sequence** is an ordered collection that allows repetition - the order is the order written or collected rather than the order of the values.

Example of a collection operation: Select - picks out a subset of elements from a collection that meet a particular condition, the syntax: collection -> select(v | boolean-expression-with-v)

How do we translate constraints into Code ?
Pre-conditions on operations can be translated into simple checks at the beginning of the code that implements the operations. Invariants can be checked by introducing code before and after a change to the attributes indicated by the invariant. Post-conditions should be true after the execution of the operation. You can check them at the exit of the operation.

Relationship of OCL contracts with UML diagrams ?
We either hold the OCL expressions separately or we add them as notes attached to the diagrams.

Exercise One: Insurance policies



Express the following in OCL:

•Membership must be fully paid up before taking out insurance

context SPControl::gatherInsuranceDetails()

pre: CarSharer.membership.balance >= 0

•The first payment of the payment schedule must be made before cover is granted

context SPControl::createPaymentSchedule(startDate, regularDate, amount, frequencey)

pre: PaymentSchedule.startDate.before(policy.startDate)

•Two policies must not be in force at the same time for a given car sharer

context CarSharer **inv**:

Policy -> forall(p1,p2 | p1 <> p2 and p1.startDate.before(p2.startDate) implies p1.renewalDate.before(p2.startDate)

•The renewal day of policy is always after its start day

context Policy **inv**:

startDate.before(renewalDate)

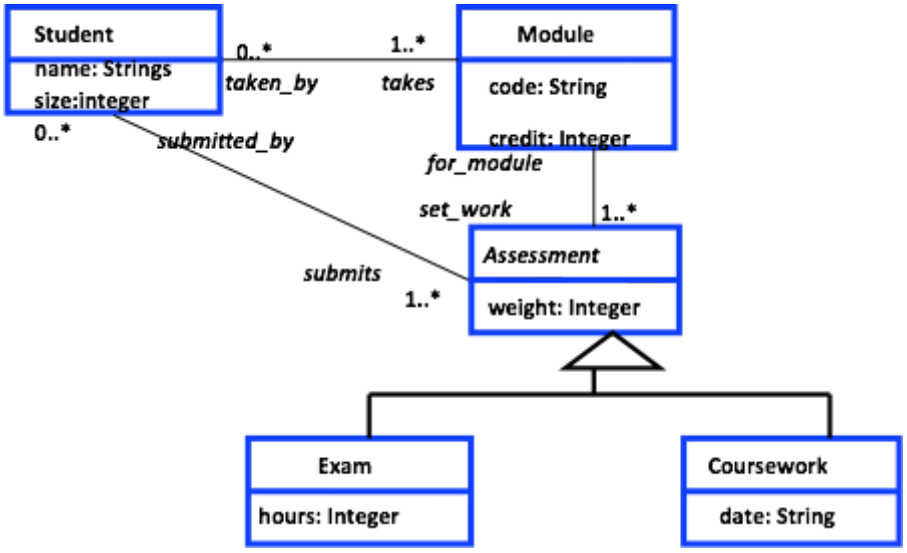
•**The renewal date of a policy must fall before start day of any later policy**

context SPControl::createPolicy(startDate, renewalDate)

pre: CarSharer.Policy -> forall(p | p.startDate.before(self.startDate) implies p.renewalDate.before(self.startDate)

****

Exercise Two: Students and Exams



Your task is to define in OCL the following constraints:

•Modules can be taken if they have more than seven students registered

context *Module*

invariant: *taken_by* **->** *size > 7*

•The assessments for a module must total 100%

context *Module* **invariant**:

Set_work.weight **->** *sum*( ) = *100*

•Students must register for 120 credits each year

context *Student*

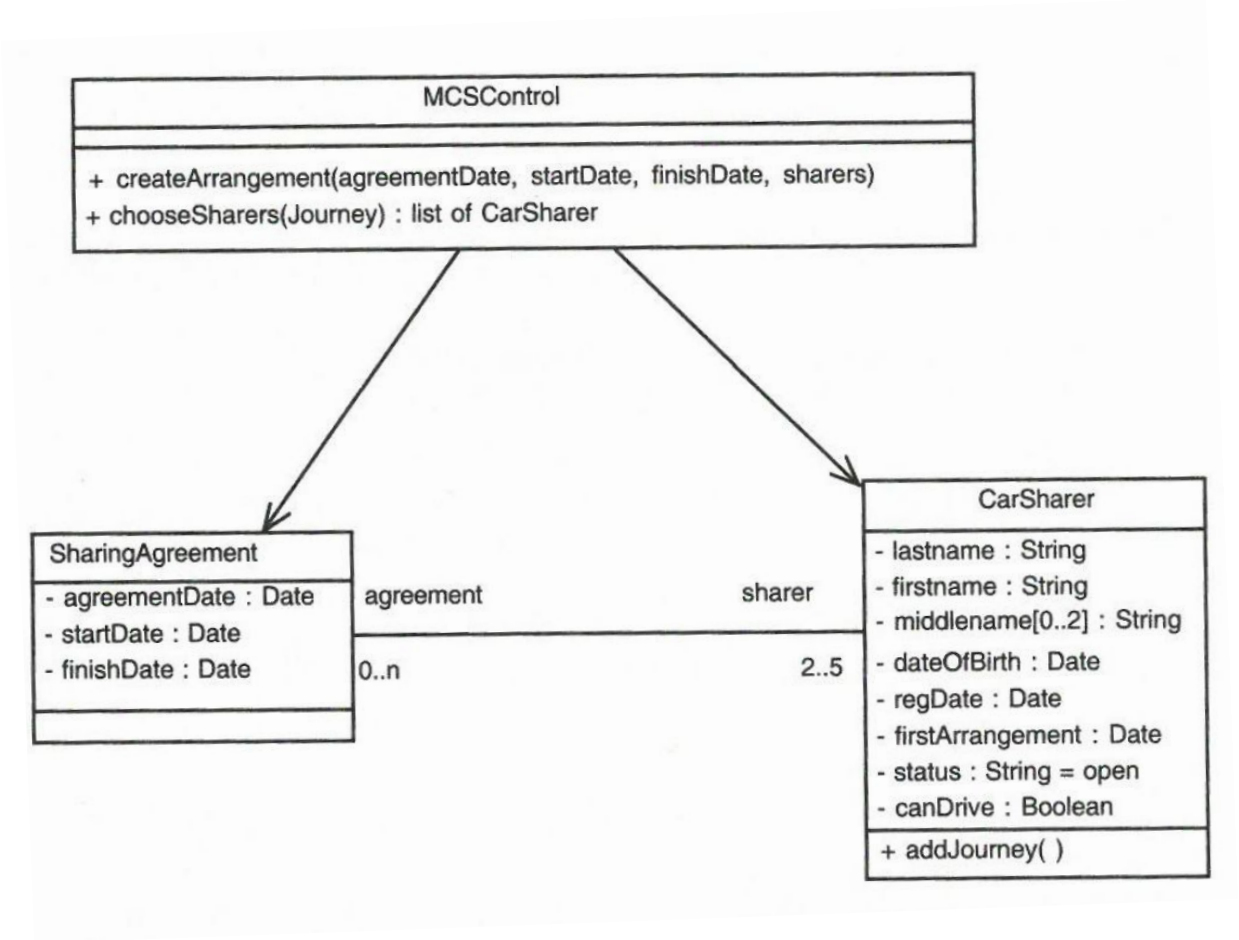
invariant: *takes.credit* **->** *sum*( ) = *120*

•In the assessments of modules that students take exams must be at least 4 hours

context Student inv: takes.set_work.hours **->** forall(h!h>=4)

****

Exercise Three: Consider the “Match Car Sharers” use case. There is the following condition of sharing: to create a sharing agreement, there must be at least one driver, and at least two people in the agreement. Write OCL constraints to handle these requirements.



The requirement means that a list of car sharers among whom is a driver should be available, because we need a means of checking if a car sharer can drive to create an arrangement, so we will have the following:

context MCSControl::createArrangment(agreementDate, startDate, finishDate, sharers)

pre: sharers->exists(x | x.canDrive) and (sharers->size) >= 2