Assignment - 5

DESIGN AND IMPLEMENTATION DESIGN MODEL FROM ANALYSIS MODEL

Aim: Study in detail working of system/Project. Identify Design classes/ Evolve Analysis Model. Use advanced relationships. Draw Design class Model using OCL and UML2.0 Notations. Implement the design model with a suitable object-oriented language

Problem Statement:

- Prepare a Design Model from Analysis Model
- Study in detail work of the NBA Attainment System.
- Identify Design classes/ Evolve Analysis Model. Use advanced relationships. Draw Design class Model usingOCL and UML2.0 Notations. Implement the design model with a suitable object-oriented language.

Objective: To Identify Design level Classes. To Draw Design level class Model using the analysis model. To Implement Design Model-class diagram

Theory :

- Class Diagram:
  The Class diagram shows the building blocks of any object-orientated system. Class diagrams depict the static view of the model or part of the model, describing what attributes and behaviors it has rather than detailing the methods for achieving operations. Class diagrams are most useful to illustrate relationships between classes and interfaces. Generalizations, aggregations, and associations are all valuable in reflecting inheritance, composition or usage, and connections, respectively.

- Associations
  An association implies two model elements have a relationship - usually implemented as an instance variable in one class.
  Manage Test, Manage Co-Po, Manage Marks, Report Generation, Manage target Attainment class are associated with User class with relationships such as one to many and many to many.

- Generalizations
  A generalization is used to indicate inheritance. Drawn from the specific classifier to a general classifier, the generalized implication is that the source inherits the target's characteristics.
  For Teacher, Subject-Coordinator and HOD class user class is the superclass. All of these, share a relationship and these relationships are known as generalized relationships.

- Aggregations

  Aggregations are used to depict elements that are made up of smaller components. Aggregation relationships are shown by a white diamond-shaped arrowhead pointing towards the target or parent class. A stronger form of aggregation - a composite aggregation - is shown by a black diamond-shaped arrowhead and is used where components can be included in a maximum of one composition at a time.
  User class, Manage Test class, Manage Marks class, Manage target Attainment class, Manage CO PO class and Report Generation class, all are aggregated to NBA class

- Composition:
  The composition is a special type of aggregation which denotes strong ownership between two classes when one class is a part of another class.
  In the NBA system, marks cannot be entered if a test is not set. Hence, there is a composition between Test class and Marks class

Classes

A class is an element that defines the attributes and behaviors that an object can generate.

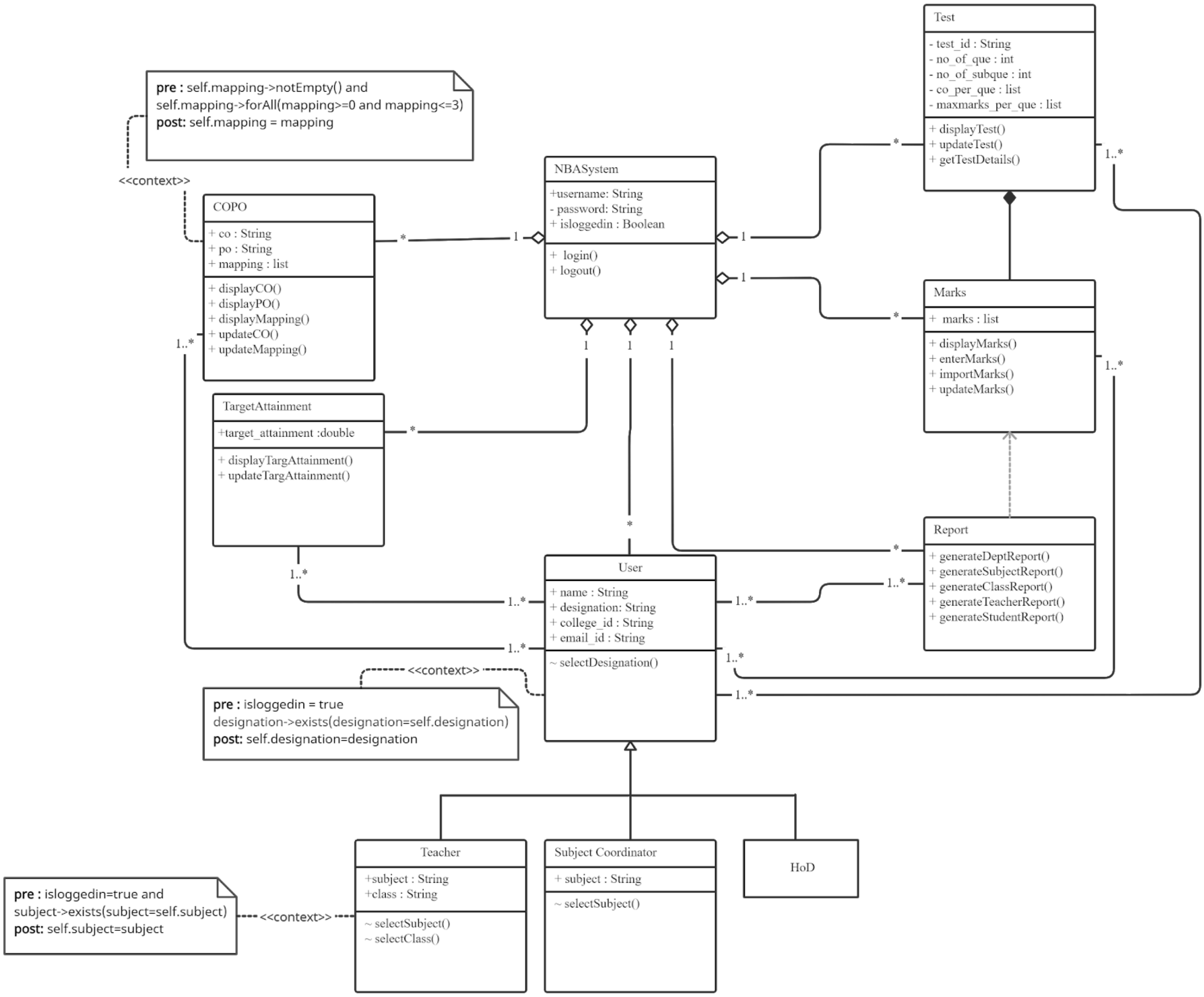
Classes of NBA Attainment System are :

- NBASystem
  Manages the complete system
- User
  It manages all operations of the user.
- COPO
  It manages CO’s, PO’s and its mapping
- Marks
  It manages marks
- Test
  It manages displaying and updation of tests
- Report
  It manages report generation
- TargetAttainment
  Manages target attainment
- Teacher
  It manages all operations of the teacher.
- SubjectCoordinator
  It manages all operations of the subject-coordinator.
- HoD
  It manages all operations of HOD.

Class Notation :

Classes are represented by rectangles which show the name of the class and optionally the name of the operations and attributes. Classes are composed of three things: a name, attributes, and operations.

Class Diagram for NBA Attainment System



2. Object Constraint Language The Object Constraint Language (OCL) is a declarative language describing rules applying to Unified Modeling Language (UML) models developed at IBM and is now part of the UML standard.

Initially, OCL was merely a formal specification language extension for UML. OCL supplements UML by providing expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. OCL is also a navigation language for graph-based models.

Object Constraint Language (OCL), is a formal language to express side-effect-free constraints. Users of the Unified Modeling Language and other languages can use OCL to specify constraints and other expressions attached to their models.

The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use. OCL has been developed to fill this gap. It is a formal language that remains easy to read and write. OCL is a pure expression language. Therefore, an OCL expression is guaranteed to be without side effects it cannot change anything in the model. This means that the state of the system will never change because of an OCL expression, even though an OCL expression can be used to specify a state change, e.g. in a post-condition. All values for all objects, including all links, will not change.

Whenever an OCL expression is evaluated, it simply delivers a value. OCL statements are constructed in four parts:
1. A context that defines the limited situation in which the statement is valid
2. A property that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute)
3. An operation (e.g., arithmetic, set-oriented) that manipulates or qualifies a property, and
4. Keywords (e.g., if, then, else, and, or, not, implies) that are used to specify conditional expressions.

Object Constraint Language

Context User:: selectDesignation(designation:String): String
pre : isloggedin = true designation->exists(designation=self.designation)
post: self.designation=designation

Context Teacher:: selectSubject(subject:String): String
pre : isloggedin=true and subject->exists(subject=self.subject)
post: self.subject=subject

Context Teacher:: selectClass(subject:String,class:String): String
pre : isloggedin=true and class>exists(class=self.class)
post: self.class=class

Context SubjectCoordinator:: selectSubject(subject:String): String
pre : isloggedin=true and subject->exists(subject=self.subject)
post: self.subject=subject

Context ManageCoPo:: displayCO(subject:String,co:list): void
pre : co->notEmpty()
post:

Context ManageCoPo:: updateCO(subject:String,co:list,designation:String): list
pre : self.co->notEmpty() and self.designation = "Subject Coordinator"
post: self.co=co

Context ManageCoPo:: displayPO(po:list): void
pre : self.po->notEmpty()
post:

Context ManageCoPo:: displayMapping(subject:String,mapping:list): void
pre : self.mapping->notEmpty()
post:

Context ManageCoPo:: updateMapping(subject:String,mapping:list): list
pre : self.mapping->notEmpty() and self.mapping->forAll(mapping>=0 and mapping<=3)
post: self.mapping = mapping

Context Test:: displayTest(test_id:String,no_of_que:int,no_of_subque:int): void
pre : self.test->exists(test_id=self.test_id)
post:

Context Test::
updateTest(test_id:String,no_of_que:int,no_of_subque:int,co_per_que:list,maxmarks_per_que:list): Test
pre : self.test_id->notEmpty() and self.no_of_que->notEmpty() and self.no_of_subque->notEmpty() and self.co_per_que->notEmpty() and self.maxmarks_per_que->notEmpty()
post: self.test_id=test_id and self.no_of_que=no_of_que and self.no_of_subque=no_of_subque and self.co_per_que=co_per_que and self.maxmarks_per_que=maxmarks_per_que

Context Test::
getTestDetails(test_id:String,no_of_que:int,no_of_subque:int,co_per_que:list,maxmarks_per_que:list): Test
pre : self.test_id->exists(test_id=self.test_id)
post: self.test_id=test_id and self.no_of_que=no_of_que and self.no_of_subque=no_of_subque and self.co_per_que=co_per_que and self.maxmarks_per_que=maxmarks_per_que

Context ManageMarks:: enterMarks(test_id:String,marks:list,maxmarks_per_que:int): list
pre : self.marks->forAll(self.marks>=0 and self.marks<=maxmarks_per_que)
post: self.marks=marks

Context ManageMarks:: importMarks(test_id:String,marks:list): list
pre : self.marks->forAll(self.marks>=0 and self.marks<=maxmarks_per_que)
post: self.marks=marks

Context ManageMarks:: displayMarks(test_id:String,marks:list): void
pre : self.test->exists(test_id=self.test_id) and marks->notEmpty()
post:

Context ManageMarks:: updateMarks(test_id:String,marks:list): list
pre : self.test->exists(test_id=self.test_id) and self.marks->forAll(self.marks>=0 and self.marks<=maxmarks_per_que)
post: self.marks=marks

Context ManageTargetAttainment:: displayTargetAttainment(attainment:double,class:String,subject:String): void
pre : self.attainment->notEmpty()
post:

Context ManageTargetAttainment:: updateTargetAttainment(attainment:double,subject:String,class:String): double
pre : self.attainment>=0 and self.attainment<=1
post: self.attainment = attainment

Conclusion: Thus in this assignment, we have successfully designed a structure of the NBA Attainment System using UML 2.0 class diagram notations. Furthermore, we have used OCL to describe expressions and constraints on our object-oriented models