

UNIVERSITY OF PETROLEUM & ENERGY STUDIES

Department of Systemics , School of Computer Science

OBJECT ORIENTED ANALYSIS AND DESIGN

LAB FILE

SUBMITTED BY : SUBMITTED TO:

Vanshika Garg Mrs VedPrakash

ROLL NO. : 059

SAP ID : 500061511

BATCH : Devops –b2

**EXPERIMENT-1**

|  |
| --- |
| ***TITLE:*** *OOAD Overview, UML OVERVIEW, Goals of UML, A conceptual model of UML, Object oriented concepts, Role of UML in OO design*. |

**OOAD OVERVIEW:**

Object-oriented analysis and design (OOAD) is a technical approach used in the analysis and design of an application or system through the application of the object-oriented paradigm and concepts including visual modelling. This is applied throughout the development life cycle of the application or system, fostering better product quality and even encouraging stakeholder participation and communication.

Software and computer application systems are incredibly complex concepts since there are few material restrictions and a lot of possible arbitrary reconstructions. Contrast that to things like bridge or building design, where the concept of a bridge or building is defined by the materials to be used and the nature of the environment it is built upon, which results in few options. Software does not enjoy the same restrictions, and the room for complexity to grow is very large. This is where object-oriented analysis and design comes into play. It uses abstraction as a tool to encapsulate complexity, and the more abstractions are introduced, the greater is the reduction in complexity. These acts of abstraction and encapsulation allow for certain problems to be highlighted and subsequently suppressed.  
  
OOAD is best applied iteratively since there is no clear process involved, but each aspect where OOAD is applied is refined as it is reused. This is because major portions of the designs are based on the entire aspects of the system and on the entities rather than on individual functions and code. This enforces the modular approach of OOAD whose goal is to break down the problem or the system into smaller units, called objects, that can stand on their own and be changed without affecting the ones around them too much. This makes it easy to add functionality and behaviour and allow the system to gracefully accept change.

**UML OVERVIEW:**

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.OMG is continuously making efforts to create a truly industry standard.

UML stands for Unified Modeling Language.

UML is different from the other common programming languages such as C++, Java, COBOL, etc.

UML is a pictorial language used to make software blueprints.

UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.

Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.

UML is not a programming language but tools can be used to generate c-\*ode in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design. After some standardization, UML has become an OMG standard.

**Goals of UML:**

There are a number of goals for developing UML but the most important is to define some general purpose modeling language, which all modelers can use and it also needs to be made 4.UML diagrams are not only made for developers but also for business users, common people, and anybody interested to understand the system. The system can be a software or non-software system. Thus it must be clear that UML is not a development method rather it accompanies with processes to make it a successful system.

In conclusion, the goal of UML can be defined as a simple modeling mechanism to model all possible practical systems in today’s complex environment.

**Conceptual Model of UML:**

A conceptual model can be defined as a model which is made of concepts and their relationships.A conceptual model is the first step before drawing a UML diagram. It helps to understand the entities in the real world and how they interact with each other.

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −

* UML building blocks
* Rules to connect the building blocks
* Common mechanisms of UML

*UML building blocks :*

The building blocks of UML can be defined as −

* Things
* Relationships
* Diagrams

## (a)Things: Things are the most important building blocks of UML. Things can be –

|  |
| --- |
| * **Structural :**   Structural things define the static part of the model. They represent the physical and conceptual elements. Following are the brief descriptions of the structural things.  *classCLASS-* Class represents a set of objects having similar responsibilities  *INTERFACE-* defines a set of operations, which specify the responsibility of a class.  Interface  *COLLABORATION−*Collaboration defines an interaction between elements.  Collaboration  *USE CASE***−**Use case represents a set of actions performed by a system for a specific goal.  Use case  *COMPONENT***−**Component describes the physical part of a system.  Component  *NODE −* A node can be defined as a physical element that exists at run time.  **Node**   * **Behavioral**   *A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things –*  *INTERACTION***−** Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.  Interaction  *STATE MACHINE***−** State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change  State machine   * **Grouping-**   ***Grouping things*** *can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available –*  *PACKAGE−* Package is the only one grouping thing available for gathering structural and behavioral things.  Package   * **Annotational**   ***Annotational things*** *can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements.. A note is used to render comments, constraints, etc. of an UML element.*  Note |

## (b)Relationship-Relationship is another most important building block of UML. It shows how the elements are associated with each other and this association describes the functionality of an application.

|  |
| --- |
| *DEPENDENCY-*Dependency is a relationship between two things in which change in one element also affects the other. Dependency *ASSOCIATION-*Association is basically a set of links that connects the elements of a UML model. It also describes how many objects are taking part in that relationship. Association *GENERALIZATION-*Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes the inheritance relationship in the world of objects. Generalization *REALIZATION-* Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility, which is not implemented and the other one implements them. This relationship exists in case of interfaces. Realization |

**UML Diagrams** -UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system. The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it complete.

|  |
| --- |
| UML includes the following nine diagrams, the details of which are described in the subsequent chapters.   * Class diagram * Object diagram * Use case diagram * Sequence diagram * Collaboration diagram * Activity diagram * Statechart diagram * Deployment diagram * Component diagram |

#### 2. Rules of the UML

The UML's building blocks can't simply be thrown together in a random fashion. Like any language, the UML has a number of rules that specify what a well-formed model should look like. A well-formed model is one that is semantically self-consistent and in harmony with all its related models. The UML has syntactic and semantic rules for-

| Names | What you can call things, relationships, and diagrams |
| --- | --- |
| Scope | The context that gives specific meaning to a name |
| Visibility | How those names can be seen and used by others |
| Integrity | How things properly and consistently relate to one another |
| Execution | What it means to run or simulate a dynamic model |

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are

| Elided | Certain elements are hidden to simplify the view |
| --- | --- |
| Incomplete | Certain elements may be missing |
| Inconsistent | The integrity of the model is not guaranteed |

#### 3.Common Mechanisms in the UML

A building is made simpler and more harmonious by the conformance to a pattern of common features. A house may be built in the Victorian or French country style largely by using certain architectural patterns that define those styles. The same is true of the UML. It is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

|  |
| --- |
| **Specifications** The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies; visually, that class icon might only show a small part of this specification. Furthermore, there might be another view of that class that presents a completely different set of parts yet is still consistent with the class's underlying specification. You use the UML's graphical notation to visualize a system; you use the UML's specification to state the system's details. Given this split, it's possible to build up a model incrementally by drawing diagrams and then adding semantics to the model's specifications, or directly by creating a specification, perhaps by reverse engineering an existing system, and then creating diagrams that are projections into those specifications.  The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion. The UML's diagrams are thus simply visual projections into that backplane, each diagram revealing a specific interesting aspect of the system **(b)Adornments** Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented systems. The class notation also exposes the most important aspects of a class, namely its name, attributes, and operations.A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation. For example, [Figure](http://umlguide2.uw.hu/ch02lev1sec2.html#ch02fig18)  shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.    **(c)Common Divisions** In modeling object-oriented systems, the world often gets divided in several ways.  First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in [Figure.](http://umlguide2.uw.hu/ch02lev1sec2.html#ch02fig19)Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlying the object's name.    In this figure, there is one class, named Customer, together with three objects: Jan (which is marked explicitly as being a Customer object), :Customer (an anonymous Customer object), and Elyse (which in its specification is marked as being a kind of Customer object, although it's not shown explicitly here).  Almost every building block in the UML has this same kind of class/object dichotomy. For example, you can have use cases and use case executions, components and component instances, nodes and node instances, and so on.Second, there is the separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown in [Figure](http://umlguide2.uw.hu/ch02lev1sec2.html#ch02fig20).      In this figure, there is one component named SpellingWizard.dll that provides (implements) two interfaces, IUnknown and ISpelling. It also requires an interface, IDictionary, that must be provided by another component.Almost every building block in the UML has this same kind of interface/implementation dichotomy. For example, you can have use cases and the collaborations that realize them, as well as operations and the methods that implement them.  Third, there is the separation of type and role. The type declares the class of an entity, such as an object, an attribute, or a parameter. A role describes the meaning of an entity within its context, such as a class, component, or collaboration. Any entity that forms part of the structure of another entity, such as an attribute, has both characteristics: It derives some of its meaning from its inherent type and some of its meaning from its role within its context ([Figure](http://umlguide2.uw.hu/ch02lev1sec2.html#ch02fig21)).     **Extensibility Mechanisms** The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. For this reason, the UML is opened-ended, making it possible for you to extend the language in controlled ways. The UML's extensibility mechanisms include   * Stereotypes * Tagged values * Constraints   A [**stereotype**](http://umlguide2.uw.hu/gloss01.html#gloss01entry161) extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, you only want to allow them to be thrown and caught, nothing else. You can make exceptions first-class citizens in your modelsmeaning that they are treated like basic building blocksby marking them with an appropriate stereotype, as for the class Overflow in [Figure](http://umlguide2.uw.hu/ch02lev1sec2.html#ch02fig19).  A [t**agged valu**e](http://umlguide2.uw.hu/gloss01.html#gloss01entry173) extends the properties of a UML stereotype, allowing you to create new information in the stereotype's specification. For example, if you are working on a shrink-wrapped product that undergoes many releases over time, you often want to track the version and author of certain critical abstractions. Version and author are not primitive UML concepts. They can be added to any building block, such as a class, by introducing new tagged values to that building block. In [Figure](http://umlguide2.uw.hu/ch02lev1sec2.html#ch02fig19), for example, the class EventQueue is extended by marking its version and author explicitly.  A [**constraint**](http://umlguide2.uw.hu/gloss01.html#gloss01entry44) extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. For example, you might want to constrain the EventQueue class so that all additions are done in order. As [Figure](http://umlguide2.uw.hu/ch02lev1sec2.html#ch02fig22)shows, you can add a constraint that explicitly marks these for the operation add.    Collectively, these three extensibility mechanisms allow you to shape and grow the UML to your project's needs. These mechanisms also let the UML adapt to new software technology, such as the likely emergence of more powerful distributed programming languages. You can add new building blocks, modify the specification of existing ones, and even change their semantics. Naturally, it's important that you do so in controlled ways so that through these extensions, you remain true to the UML's purposethe communication of information. |

## Object-Oriented Concepts:

UML can be described as the successor of object-oriented (OO) analysis and design.

An object contains both data and methods that control the data. The data represents the state of the object. A class describes an object and they also form a hierarchy to model the real-world system. The hierarchy is represented as inheritance and the classes can also be associated in different ways as per the requirement.

Objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML.

UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design. UML diagrams are representation of object-oriented concepts only. Thus, before learning UML, it becomes important to understand OO concept in detail.

Following are some fundamental concepts of the object-oriented world −

* **Objects** − Objects represent an entity and the basic building block.
* **Class** − Class is the blue print of an object.
* **Abstraction** − Abstraction represents the behavior of an real world entity.
* **Encapsulation** − Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
* **Inheritance** − Inheritance is the mechanism of making new classes from existing ones.
* **Polymorphism** − It defines the mechanism to exists in different forms.

## Role of UML in OO Design:

UML is a modeling language used to model software and non-software systems. Although UML is used for non-software systems, the emphasis is on modeling OO software applications. Most of the UML diagrams discussed so far are used to model different aspects such as static, dynamic, etc. Now whatever be the aspect, the artifacts are nothing but objects.

If we look into class diagram, object diagram, collaboration diagram, interaction diagrams all would basically be designed based on the objects.

Hence, the relation between OO design and UML is very important to understand. The OO design is transformed into UML diagrams according to the requirement.Once the OO analysis and design is done, the next step is very easy. The input from OO analysis and design is the input to UML diagrams.

\

**EXPERIMENT 2**

|  |
| --- |
| *TITLE: Give overview of different UML tools available. Introduce Star UML* |

**Tools available for UML diagrams-**

|  |
| --- |
| 1. Eddraw 2. Star UML 3. Umbrello 4. Altova 5. Umple 6. Visual paradigm 7. White star UML 8. Diagramo 9. Astah 10. BOUML 11. Dia 12. Concept draw 13. Giffy 14. Lucidchart 15. Magic draw 16. Visio 17. Modelio 18. Nclass 19. Open model spere 20. Reactive blocks |

**Star UML**

|  |
| --- |
| StarUML is an open source software modeling tool that supports UML (Unified Modeling Language). It is based on UML version 1.4, provides eleven different types of diagram and it accepts UML 2.0 notation. It actively supports the MDA (Model Driven Architecture) approach by supporting the UML profile concept and allowing to generate code for multiple languages.  StarUML makes a clear conceptual distinction between models, views and diagrams. A Model is an element that contains information for a software model. A View is a visual expression of the information contained in a model, and a Diagram is a collection of view elements that represent the user’s specific design thoughts.  StarUML supports the following diagram types   * Use Case Diagram * Class Diagram * Sequence Diagram * Collaboration Diagram * Statechart Diagram * Activity Diagram * Component Diagram * Deployment Diagram * Composite Structure Diagram   The user interface is intuitive. On the upper right side, a window allows to rapidly navigate between all the content of a project, adopting either a model or a diagram view. Multiple diagrams can be open at the same time and tabs allow switching rapidly between views. The lower right window allows to document the current diagram, either with plain text or attaching an external document. During diagram editing, "wizards" are located around the object that give you the quick shortcuts to main associated tasks with your current operation, like adding an attribute when you create a class for instance. A right-click on the mouse brings the full set of operations at your disposal.StarUML has also a model verification feature. You can export diagram in different formats (jpg, bmp, wmf). It also supports a patterns approach and import of Rational Rose files.  StarUML is mostly written in Delphi. However, StarUML is *multi-lingual project* and not tied to specific programming language, so any programming languages can be used to develop StarUML. (for example, C/C++, Java, Visual Basic, Delphi, JScript, VBScript, C#, VB.NET, ...) |

**EXPERIMENT-3**

|  |
| --- |
| ***TITLE:***  *Explain Use Case Diagram.(Use case ,Actors, Relationships)*  *Draw Use Case Diagram* |

USE CASE DIAGRAM

A use case diagram at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different [use cases](https://en.wikipedia.org/wiki/Use_case) in which the user is involved. A use case diagram can identify the different types of users of a system and the different use cases and will often be accompanied by other types of diagrams as well. The use cases are represented by either circles or ellipses.

Use case diagrams are valuable for visualizing the functional requirements of a system that will translate into design choices and development priorities.

They also help identify any internal or external factors that may influence the system and should be taken into consideration.

They provide a good high level analysis from outside the system. Use case diagrams specify how the system interacts with actors without worrying about the details of how that functionality is implemented.

|  |  |
| --- | --- |
| **Notation Description** | **Visual Representation** |
| **Actor**   * Someone interacts with use case (system function). * Named by noun. * Actor plays a role in the business * Similar to the concept of user, but a user can play different roles * For example:   + A prof. can be instructor and also researcher   + plays 2 roles with two systems * Actor triggers use case(s). * Actor has a responsibility toward the system (inputs), and Actor has expectations from the system (outputs). | Use Case Diagram Notation - Actor |
| **Use Case**   * System function (process - automated or manual) * Named by verb + Noun (or Noun Phrase). * i.e. Do something * Each Actor must be linked to a use case, while some use cases may not be linked to actors. | Use Case Diagram Notation - Use Case |
| **Communication Link**   * The participation of an actor in a use case is shown by connecting an actor to a use case by a solid link. * Actors may be connected to use cases by associations, indicating that the actor and the use case communicate with one another using messages. |  |
| **Boundary of system**   * The system boundary is potentially the entire system as defined in the requirements document. * For large and complex systems, each module may be the system boundary. * For example, for an ERP system for an organization, each of the modules such as personnel, payroll, accounting, etc. * can form a system boundary for use cases specific to each of these business functions. * The entire system can span all of these modules depicting the overall system boundary | Use Case Diagram Notation - System Boundary |

**RELATIONSHIPS**

Use cases share different kinds of relationships. Defining the relationship between two use cases is the decision of the software analysts of the use case diagram. A relationship between two use cases is basically modeling the dependency between the two use cases. The reuse of an existing use case by using different types of relationships reduces the overall effort required in developing a system

|  |  |
| --- | --- |
| **Use Case Relationship** | **Visual Representation** |
| **Extends**   * Indicates that an **"Invalid Password"** use case may include (subject to specified in the extension) the behavior specified by base use case **"Login Account"**. * Depict with a directed arrow having a dotted line. The tip of arrowhead points to the base use case and the child use case is connected at the base of the arrow. * The stereotype "<<extends>>" identifies as an extend relationship | Use Case Diagram Notation - Extend |
| **Include**   * When a use case is depicted as using the functionality of another use case, the relationship between the use cases is named as include or uses relationship. * A use case includes the functionality described in another use case as a part of its business process flow. * A uses relationship from base use case to child use case indicates that an instance of the base use case will include the behavior as specified in the child use case. * An include relationship is depicted with a directed arrow having a dotted line. The tip of arrowhead points to the child use case and the parent use case connected at the base of the arrow. * The stereotype "<<include>>" identifies the relationship as an include relationship. | Use Case Diagram Notation - Include |
| **Generalization**   * A generalization relationship is a parent-child relationship between use cases. * The child use case is an enhancement of the parent use case. * Generalization is shown as a directed arrow with a triangle arrowhead. * The child use case is connected at the base of the arrow. The tip of the arrow is connected to the parent use case. | Use Case Diagram Notation - Generalization |

***Use Case Diagram for An organization wants to develop the system based on following Requirements***:

1. Students have to register first through fill up the registration form.

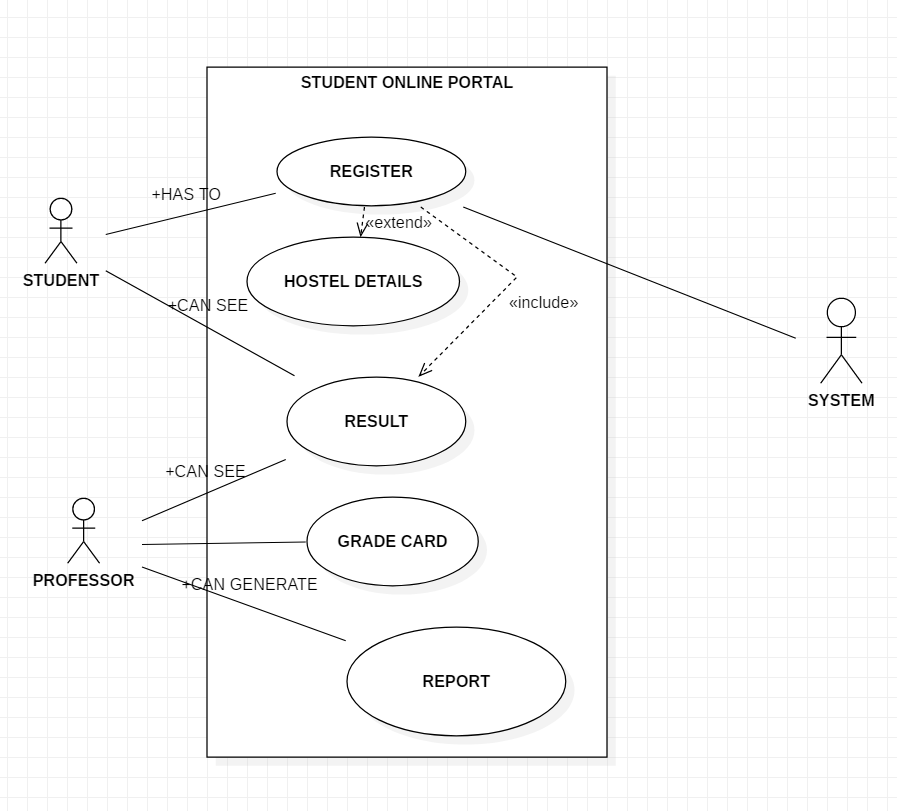
2. Faculty can prepare the grade sheet of concern subjects.

3. Student and faculty can see the result of each student.

4. Faculty can generate the report about registered students.

5. System can verify the students Registration Details First.

6. Student may have to fill up the hostel details during registration process.



***Use case diagram for a book store***

Consider a book store in a shopping mall. The customer selects the books from racks to purchase. The customer brings selected books to cashier. The cashier scans each item with checkout system to prepare an order. The cashier requests to customer for payment. The customer gives credit card to cashier. The verifier and checkout system scans the card. The verifier accepts the card and payment is accepted. Customer signs the credit card slip. The purchased books are handed over to customer.

