**TABLE OF CONTENT**

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Contents** | **PAGE NO** |
| 1 | Syllabus | 1 |
| 2 | Course Overview, Objective and LearningOutcomes | 3-4 |
| 4 | EvaluationCriteria | 5 |
| 5 | BooksRecommendation | 6 |
| 6 | SessionPlan | 7 |
| 7 | Brief Profile of Faculty | 11-12 |
| 8 | **Unit I:** Object oriented concepts, modeling and UML:   * Introduction to class,object,inheritance * Introduction to modeling * Object oriented modeling * Object oriented system development * Datamethods * Object oriented analysis * Object oriented construction * Object oriented Testing | 14 |
| 9 | **Unit II**: Interactive development and UML:   * Understanding requirements * Rational unified process * Introduction to UML * Conceptual model of UML * Advantages of UML * Behavioral modeling – use case diagrams, Activity diagrams | 38 |

|  |  |  |
| --- | --- | --- |
| 10. | **Unit III**: Basic and advanced structural modeling:  • Class diagrams- identifying classes, objects, relations.  • Advanced class modeling- interface, types , roles  • Object diagrams  • State chart diagrams  • Package diagrams | 58 |
| 11. | **Unit IV:** Interaction Modeling :  • Interaction diagrams and its need  • Collaboration diagrams  • Sequence diagrams | 77 |
| 12. | **Unit V** : Architectural diagram:  • Component diagrams and its need  • Deployment diagrams , its purpose, Architecture system | 106 |
| 13. | **Unit VI:** Object oriented programming styles:  • Object oriented style with reference to reusability, extensibility, robustness.  • Case studies on: library management system, hospital management system, online shopping etc. | 117 |
| 14. | Practice Questions | 122 |
| 15. | Previous year Internal Question papers | 124 |
| 16. | Previous year University question papers | 130 |
| 17. | Solved paper | 136 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Course Number** | **Course Name** | **L-T-P- Credits** | **Year of Introduction** |
| 502 | OOAD | 3L-1T-0P-=4C | 2018 |
| **Course Objectives**   * To understand the concepts of object oriented concepts, modeling and UML * To understand the concept of UML and behavioral modeling * To understand the concept of basic and advanced structural modeling * To understand Interaction modeling * To understand Architectural modeling * To understand the concept of object oriented programming styles | | | |
| **Expected Outcome :**   1. To be able to specify, analyse and design the use case driven requirements for a particular system. 2. To model the event driven state of object and transform them into implementation specific layouts. 3. To identify, analyse the subsystems, various components and collaborate them interchangeably. | | | |
| **Reference Books:**   1. “Object oriented analysis and design “by James Ramough. 2. Principles of information security by whiteman 3. The complete reference-information security-mark Rhodes ousley | | | |
| **Course Plan** | | | |
| **UNIT** | | **Contents** | |
| 1 | | **Object oriented concepts, modeling and UML:**   * Introduction to class,object,inheritance * Introduction to modeling * Object oriented modeling * Object oriented system development * Datamethods * Object oriented analysis * Object oriented construction * Object oriented Testing | |
| 2 | | **Interactive development and UML:**   * Understanding requirements * Rational unified process * Introduction to UML * Conceptual model of UML * Advantages of UML * Behavioral modeling – use case diagrams, Activity diagrams | |
| 3 | | **Basic and advanced structural modeling:**  • Class diagrams- identifying classes,objects,relations.  • Advanced class modeling- interface, types , roles  • Object diagrams  • State chart diagrams  • Package diagrams | |
| 4 | | **Interaction Modeling :**  • Interaction diagrams and its need  • Collaboration diagrams  • Sequence diagrams | |
| 5 | | **Architectural diagram:**  • Component diagrams and its need  • Deployment diagrams , its purpose, Architecture system | |
| 6 | | **Object oriented programming styles:**  • Object oriented style with refrence to reusability,extensibility,robustness.  • Case studies on : library management system, hospital management system,online shopping etc. | |

**CourseOverview:**

C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and Object oriented analysis and designs aims at developing skills to analyze and design a software system and use the skills in unified process environment. Object-oriented design concepts, Introducing OO concepts through typical OO programming languages. Features and problems of complex systems, evolution the object-oriented model, foundations and elements of the object-oriented model, classes and objects, relationships among classes, relationships among objects, interplay of classes and objects, approaches to identifying classes and objects, object-oriented design methodologies, methodology notation (elements of UML or any other selected notation, class and object diagrams, interaction diagrams, state transition diagrams, process and module diagrams, etc.), applications and case studies, CASE tools. It includes Understanding the  importance and basic concepts  of  object oriented modeling,. The students will have an understanding of the fundamental principles of the UML Diagrams. **The students are expected to review the course readings and the indicated portion of the prescribed text for class discussions prior to attending each session.**

**Learning Outcome:**

**After undergoing this course, the student will be able to:**

* Understand the concepts of object oriented concepts, modeling and UML
* Understand the concept of UML and behavioral modeling
* Understand the concept of basic and advanced structural modeling
* Understand Interaction modeling
* Understand Architectural modeling
* Understand the concept of object oriented programming styles

1. **EvaluationCriteria:**

|  |  |  |
| --- | --- | --- |
| **Component** | **Description** | **Weight age** |
| **First**  **Internal Examination** | First internal question paper will be based on  first 3 unit of syllabus. | 10marks |
| **Second**  **Internal Examination** | Second internal question paper will be based on last 3 unit of syllabus. | 10marks |
| **CES Quiz** | Class test will be conducted based on the  different aspects of C# Language Programming Constructs | 5 marks |
| **CES Quiz** | Class test will be conducted based on the  different aspects of C# Language Programming Constructs | 5 marks |
| **CES practical** | Will have multiple choice , true false and fill in  the blanks type question based on fundamental concepts of c# programming | 5 marks |
| **Attendance** | Above 75% - 10 marks  Below 75% - 0 mark | 10 marks |
| **Note :**  All three CES will be mandatory. If any student misses anyone CES in that case the weightage of each CES would be 3.33 marks and if a student attempts all three CES then his/her best two CES will be considered, in that case the weightage would be 5 marks each. | | |

**Recommended/ Reference Text Books andResources:**

|  |  |
| --- | --- |
| Text Book | “Object oriented analysis and design “by James Ramough. |
| Course Reading | * + Principles of information security by whiteman   + The complete reference-information security-mark Rhodes ousley |
| Internet Resource: | * 1. [www.tutorialspoint.com](http://www.tutorialspoint.com)   2. [www.techopedia.com](http://www.techopedia.com) |

**SessionPlan:**

|  |  |  |
| --- | --- | --- |
| **Module I : Object Oriented Concepts, Modelling and UML** | | |
| 1 | Introduction to OO Concepts | Understanding features and uses of using object oriented. LO1 |
| 2 | OOAD Concepts | Properties of object oriented.O1 |
| 3 | Introduction to modelling | Modeling Concepts and purpose of modeling.LO1,LO2 |
| 4 | Object oriented modeling | Object oriented modeling  Based on oops Concepts .LO1,LO2 |
| 5 | Object oriented system development | Different phases in Object oriented system development .LO1,LO2 |
| 6 | Datamethods, Object oriented analysis, Object oriented LOnstruction, Object oriented Testing | Construction and testing of ooad designs.Concepts.LO1,LO2 |
| **Module II: Iterative development and UML** | | |
| 7 | OOAD requirements | Understanding requirements  .LO2 |
|
| 8 | RUP | Rational unified model in detail.LO2 |
| 9 | Understanding inception and requirements | Requirement engineering Concepts.LO2 |
| 10 | Introduction to UML | UML diagram Concepts.LO2,LO3 |
| 11 | Writing use cases in UI free style | Description LO2 |
| 12 | Use case modelling | Use case Concepts and notations.LO2 |
| 13 | Activity diagrams | Activity diagrams and notations. LO2 |
| **Module III : Basic and Advanced Structural Modelling** | | |
| 14 | Class diagrams | Concepts of classes and objects LO3 |
| 15 | Class modelling | Advanced class modeling- interface, types , role. LO3 |
| 16 | Object diagrams | Relationships and how to make object diagram from class diagramLO3 |
| 17 | State chart diagrams | State chart diagrams in details.LO3 |
| 18 | Package diagrams | Package diagrams in detail. LO3 |
| **Module IV : Interaction Modelling** | | |
| 19 | Interaction diagrams | Concepts of Interaction diagrams and its need. LO4 |
| 20 | Sequence diagrams | Relationships and how to make sequence diagram from use case diagramLO4 |
| 21 | COllaboration diagrams | Concept of COllaboration diagrams  LO4 |
| **Module V: Architectural Modelling** | | |
| 22 | Component diagrams | Concept of Component diagrams  in detail.LO5 |
| 23 | Component diagrams need | Able to draw Component diagrams  . LO5 |
| 24 | Component diagrams | Realization of Component diagrams need  LO5 |
| 25 | Component diagrams | Relating Components of Component diagrams  Concepts LO5 |
| 26 | Architectural system | Architectural system of Component diagrams  LO5 |
| 27 | Deployment diagrams | Concept of Deployment diagrams LO5 |
| 28 | Deployment diagrams | Architectural system of Deployment diagrams LO5 |
| 29 | Deployment diagrams | Representation of Deployment diagrams LO5 |
| **Module VI : Object Oriented Programming Styles** | | |
| 30 | Object oriented style | Concept of ooad style.LO6, LO1 |
| 31 | Reusability.extensibility, robustness. | LOnept of Inheritance and robustness. LO1, LO6 |
| 32 | Qualified and reflexive associations | Packages notations ,Concepts ,uses.LO4 |
| 33 | Modeling large softwares | Concepts of UML diagrams.LO2 , LO3, LO5, LO6. |
| 34 | Case studies | Library management system. LO2 , LO3, LO5, LO6. |
| 35 | Case studies | Hospital management system LO2 , LO3, LO5, LO6. |
| 36 | Case studies | ATM system. LO2 , LO3, LO5, LO6. |
| 37 | Case studies | Online shopping system. LO2 , LO3, LO5, LO6. |
| 38 | Case studies | Naukari.LOm website. LO2 , LO3, LO5, LO6. |
|
| 39 | Case studies | Matrimonial site. LO2 , LO3, LO5, LO6. |
| 40 | Case studies | LOllege management system. LO2 , LO3, LO5, LO6. |

**UNIT 1**

* **Introduction to class,object,inheritance**
* **Introduction to modeling**
* **Object oriented modeling**
* **Object oriented system development**
* **Datamethods**
* **Object oriented analysis**
* **Object oriented construction**
* **Object oriented Testing**

# Object-Oriented Analysis

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system’s object model, which comprises of interacting objects.

The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

Grady Booch has defined OOA as, “Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain”.

The primary tasks in object-oriented analysis (OOA) are:

* + Identifying objects
  + Organizing the objects by creating object model diagram
  + Defining the internals of the objects, or object attributes
  + Defining the behavior of the objects, i.e., object actions
  + Describing how the objects interact

The common models used in OOA are use cases and object models.

**Object-Oriented Design**

Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

The implementation details generally include:

* Restructuring the class data (if necessary),
* Implementation of methods, i.e., internal data structures and algorithms,
* Implementation of control,and
* Implementation ofassociations.

Grady Booch has defined object-oriented design as “a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system underdesign”.

**Object-Oriented Programming**

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object–oriented programming are:

* + Bottom–up approach in program design
  + Programs organized around objects, grouped in classes
  + Focus on data with methods to operate upon object’s data
  + Interaction between objects through functions
  + Reusability of design through creation of new classes by adding features to existing classes

Some examples of object-oriented programming languages are C++, Java ,C# etc.

## Objects and Classes

The concepts of objects and classes are intrinsically linked with each other and form the foundation of object–oriented paradigm.

### Object

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has:

* Identity that distinguishes it from other objects in the system.
* State that determines the characteristic properties of an object as well as the values of the properties that the object holds.
* Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modelled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

### **Class**

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

**The constituents of a class are:**

* A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.
* A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

**Example**

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two–dimensional space. The attributes of this class can be identified as follows:

* x–coord, to denote x–coordinate of the center
* y–coord, to denote y–coordinate of the center
* a, to denote the radius of the circle

Some of its operations can be defined as follows:

* findArea(), method to calculate area
* findCircumference(), method to calculate circumference
* scale(), method to increase or decrease the radius

During instantiation, values are assigned for at least some of the attributes. If we create an object my\_circle, we can assign values like x-coord : 2, y-coord : 3, and a : 4 to depict its state. Now, if the operation scale() is performed on my\_circle with a scaling factor of 2, the value of the variable a will become 8. This operation brings a change in the state of my\_circle, i.e., the object has exhibited certain behavior.

## Encapsulation and Data Hiding

### *Encapsulation*

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

### *Data Hiding*

Typically, a class is designed such that its data (attributes) can be accessed only by its class methods and insulated from direct outside access. This process of insulating an object’s data is called data hiding or information hiding.

Example :In the class Circle, data hiding can be incorporated by making attributes invisible from outside the class and adding two more methods to the class for accessing class data, namely:

* setValues(), method to assign values to x-coord, y-coord, and a
* getValues(), method to retrieve values of x-coord, y-coord, and a

Here the private data of the object my\_circle cannot be accessed directly by any method that is not encapsulated within the class Circle. It should instead be accessed through the methods setValues() and getValues().

## Message Passing

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing. Suppose a system has two objects: obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods.

The features of message passing are:

* Message passing between two objects is generally unidirectional.
* Message passing enables all interactions between objects.
* Message passing essentially involves invoking class methods.
* Objects in different processes can be involved in message passing.

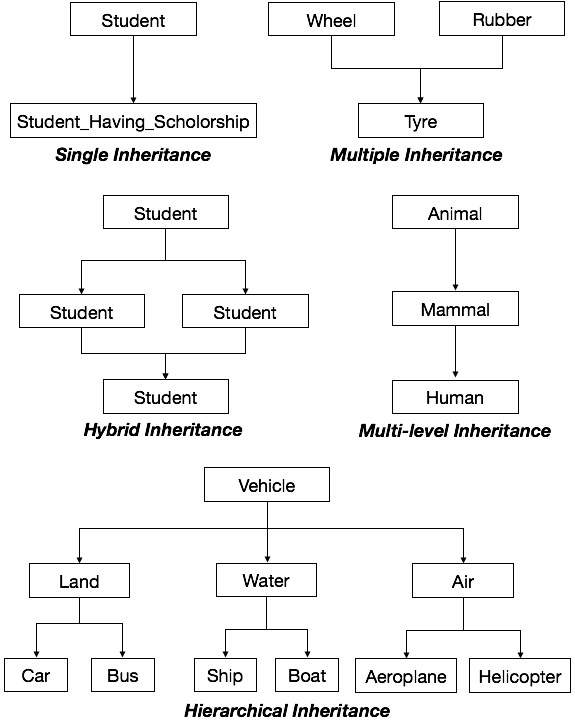
## Inheritance

Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses. The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines an “is – a” relationship.

### *Types of Inheritance:*

* Single Inheritance : A subclass derives from a single super-class.
* Multiple Inheritance : A subclass derives from more than one super-classes.
* Multilevel Inheritance : A subclass derives from a super-class which in turn is derived from another class and so on.
* Hierarchical Inheritance : A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.
* Hybrid Inheritance : A combination of multiple and multilevel inheritance so as to form a lattice structure.

The following figure depicts the examples of different types of inheritance.



## *Polymorphism*

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

Example :Let us consider two classes, Circle and Square, each with a method findArea(). Though the name and purpose of the methods in the classes are same, the internal implementation, i.e., the procedure of calculating area is different for each class. When an object of class Circle invokes its findArea() method, the operation finds the area of the circle without any conflict with the findArea() method of the Square class.

**Advantages of Object-Oriented Analysis and Design**

* It is easy to understand.
* It is easy to maintain. Due to its maintainability OOAD is becoming more popular day by day
* It provides re-usability
* It reduce the development time & cost
* It improves the quality of the system due to program reuse

# Object-Oriented Modeling (OOM)

Object-oriented modeling (OOM) is the construction of objects using a collection of objects that contain stored values of the instance variables found within an object. Unlike models that are record-oriented, object-oriented values are solely objects.

The object-oriented modeling approach creates the union of the application and database development and transforms it into a unified data model and language environment. Object-oriented modeling allows for object identification and communication while supporting data abstraction, inheritance and encapsulation.

Object-oriented modeling is the process of preparing and designing what the model’s code will actually look like. During the construction or programming phase, the modeling techniques are implemented by using a language that supports the object-oriented programming model.

OOM consists of progressively developing object representation through three phases: analysis, design, and implementation. During the initial stages of development, the model developed is abstract because the external details of the system are the central focus. The model becomes more and more detailed as it evolves, while the central focus shifts toward understanding how the system will be constructed and how it should function.

# Object Oriented Testing

Software typically undergoes many levels of testing, from unit testing to system or acceptance testing. Typically, in-unit testing, small “units”, or modules of the software, are tested separately with focus on testing the code of that module. In higher, order testing (e.g, acceptance testing), the entire system (or a subsystem) is tested with the focus on testing the functionality or external behavior of the system.

Testing classes is a fundamentally different problem than testing functions. A function (or a procedure) has a clearly defined input-output behavior, while a class does not have an input-output behavior specification. We can test a method of a class using approaches for testing functions, but we cannot test the class using these  
approaches.

Techniques of object-oriented testing are as follows:

1. **Fault Based Testing:**  
   This type of checking permits for coming up with test cases supported the consumer specification or the code or both. It tries to identify possible faults (areas of design or code that may lead to errors.). For all of these faults, a test case is developed to “flush” the errors out. These tests also force each time of code to be executed.

This method of testing does not find all types of errors. However, incorrect specification and interface errors can be missed. These types of errors can be uncovered by function testing in the traditional testing model. In the object-oriented model, interaction errors can be uncovered by scenario-based testing. This form of Object oriented-testing can only test against the client’s specifications, so interface errors are still missed.

1. **Class Testing Based on Method Testing:**  
   This approach is the simplest approach to test classes. Each method of the class performs a well defined cohesive function and can, therefore, be related to unit testing of the traditional testing techniques. Therefore all the methods of a class can be involved at least once to test the class.
2. **Random Testing:**  
   It is supported by developing a random test sequence that tries the minimum variety of operations typical to the behavior of the categories
3. **Partition Testing:**  
   This methodology categorizes the inputs and outputs of a category so as to check them severely. This minimizes the number of cases that have to be designed.
4. **Scenario-based Testing:**  
   It primarily involves capturing the user actions then stimulating them to similar actions throughout the test.  
   These tests tend to search out interaction form of error.

**Object Oriented Testing methods:**

Testing is a continuous activity during software development. In object-oriented systems, testing encompasses three levels, namely, unit testing, subsystem testing, and system testing.

**Unit Testing:**

* In unit testing, the individual classes are tested. It is seen whether the class attributes are implemented as per design and whether the methods and the interfaces are error-free.
* Unit testing is the responsibility of the application engineer who implements the structure.

**Subsystem Testing:**

* This involves testing a particular module or a subsystem and is the responsibility of the subsystem lead. It involves testing the associations within the subsystem as well as the interaction of the subsystem with the outside.
* Subsystem tests can be used as regression tests for each newly released version of the subsystem.

**System Testing:**

* System testing involves testing the system as a whole and is the responsibility of the quality-assurance team. The team often uses system tests as regression tests when assembling new releases.

**Object-Oriented Testing Techniques:**

**Grey Box Testing:**

The different types of test cases that can be designed for testing object-oriented programs are called grey box test cases. Some of the important types of grey box testing are:

* **State model based testing:** This encompasses state coverage, state transition coverage, and state transition path coverage.
* **Use case based testing:** Each scenario in each use case is tested.
* **Class diagram based testing:** Each class, derived class, associations, and aggregations are tested.
* **Sequence diagram based testing:** The methods in the messages in the sequence diagrams are tested.

**Techniques for Subsystem Testing:**

The two main approaches of subsystem testing are:

* **Thread based testing:** All classes that are needed to realize a single use case in a subsystem are integrated and tested.
* **Use based testing:** The interfaces and services of the modules at each level of hierarchy are tested. Testing starts from the individual classes to the small modules comprising of classes, gradually to larger modules, and finally all the major subsystems.

**Categories of System Testing:**

* **Alpha testing:** This is carried out by the testing team within the organization that develops software.
* **Beta testing:** This is carried out by select group of co-operating customers.
* **Acceptance testing:** This is carried out by the customer before accepting the deliverables.

**Black-box testing:**  
Testing that verifies the item being tested when given the appropriate input provides the expected results.

**Boundary-value testing:**

Testing of unusual or extreme situations that an item should be able to handle.

**Class testing:**  
The act of ensuring that a class and its instances (objects) perform as defined.

**Component testing:**  
The act of validating that a component works as defined.

**Inheritance-regression testing:**

The act of running the test cases of the super classes, both direct and indirect, on a given subclass.

**Integration testing:**  
Testing to verify several portions of software work together.

**Model review:**  
An inspection, ranging anywhere from a formal technical review to an informal walkthrough, by others who were not directly involved with the development of the model.

**Path testing:**  
The act of ensuring that all logic paths within your code are exercised at least once.

**Regression testing:**

The acts of ensuring that previously tested behaviors still work as expected after changes have been made to an application.

**Stress testing:**

The act of ensuring that the system performs as expected under high volumes of transactions, users, load, and so on.

**Technical review:**

A quality assurance technique in which the design of your application is examined critically by a group of your peers. A review typically focuses on accuracy, quality, usability, and completeness. This process is often referred to as a walkthrough, an inspection, or a peer review.

**User interface testing:**

The testing of the user interface (UI) to ensure that it follows accepted UI standards and meets the requirements defined for it. Often referred to as graphical user interface (GUI) testing.

**White-box testing:**  
Testing to verify that specific lines of code work as defined. Also referred to as clear-box testing.

**UNIT 2 :**

* **Understanding requirements**
* **Rational unified process**
* **Introduction to UML**
* **Conceptual model of UML**
* **Advantages of UML**
* **Behavioral modeling – use case diagrams, Activity diagrams**

[**Requirement Types**](http://acqnotes.com/acqnote/tasks/requirement-types)

There are a number of different type of requirement that system engineers will have to develop on a acquisition program through it life-cycle. These requirements range from very high level concept focused to very specific for a part. The main types of requirements are:

* Functional Requirements
* Performance Requirements
* System Technical Requirements
* Specifications

**Functional Requirements**

A functional requirement is simply a task (sometimes called an action or activity) that must be accomplished to provide an operational capability (or satisfy an operational requirement). Some functional requirements that are associated with operations and support can be discerned from the needed operational capability (see [Operational Requirements](http://acqnotes.com/acqNote/operational-requirements)). Others often result only from diligent systems engineering. Experience in systems engineering has identified eight generic functions that most systems must complete over their life cycle: development, manufacturing, verification, deployment, training, operations, support, and disposal. These are known as the eight primary system functions. Each must usually be considered to identify all the functional requirements for a system.

**Performance Requirements**

A performance requirement is a statement of the extent to which a function must be executed, generally measured in terms such as quantity, accuracy, coverage, timeliness, or readiness

**System Technical Requirements**

Result in both allocated and derived requirements.

* **Allocated Requirements**: flow directly from the system requirements down to the elements of the system.
* **Derived Requirements:** dependent on the design solution (and so are sometimes called design requirements). They include internal interface constraints between the elements of the system.

**Specifications**

A specification is a detailed, exact statement of particulars, especially a statement prescribing materials, dimensions, and quality of work for something to be built, installed, or manufactured. The overall purpose of a specification is to provide a basis for obtaining a product or service that will satisfy a particular need at an economical cost and to invite maximum reasonable competition. By definition, a specification sets limits and thereby eliminates, or potentially eliminates, items that are outside the boundaries drawn. A good specification should do four (4) things:

1. Identify minimum requirements
2. List reproducible test methods to be used in testing for compliance with specifications
3. Allow for a competitive bid
4. Provide for an equitable award at the lowest possible cost.

## Introduction to requirement engineering

In requirements engineering, requirements elicitation is the practice of collecting the requirements of a system from users, customers and other stakeholders. The practice is also sometimes referred to as "requirement gathering". ... Commonly used elicitation processes are the stakeholder meetings or interviews.

* The process of collecting the software requirement from the client then understand, evaluate and document it is called as requirement engineering.
* Requirement engineering constructs a bridge for design and construction.

**Requirement engineering consists of seven different tasks as follow: *1. Inception/start***

* Inception is a task where the requirement engineering asks a set of questions to establish a software process.
* In this task, it understands the problem and evaluates with the proper solution.
* It collaborates with the relationship between the customer and the developer.
* The developer and customer decide the overall scope and the nature of the question.

***2.Elicitation/find***  
Elicitation means to find the requirements from anybody.  
The requirements are difficult because the following problems occur in elicitation.  
***3. Elaboration***

* In this task, the information taken from user during inception and elaboration and are expanded and refined in elaboration.
* Its main task is developing pure model of software using functions, feature and constraints of a software.

***4. Negotiation***

* In negotiation task, a software engineer decides the how will the  project be achieved with limited business resources.
* To create rough guesses of development and access the impact of the requirement on the project cost and delivery time.

***5. Specification***

* In this task, the requirement engineer constructs a final work product.
* The work product is in the form of software requirement specification.
* In this task, formalize the requirement of the proposed software such as informative, functional and behavioral.
* The requirement are formalize in both graphical and textual formats.

***6. Validation***

* The work product is built as an output of the requirement engineering and that is accessed for the quality through a validation step.
* The formal technical reviews from the software engineer, customer and other stakeholders helps for the primary requirements validation mechanism.

***7. Requirement management***

* It is a set of activities that help the project team to identify, control and track the requirements and changes can be made to the requirements at any time of the ongoing project.
* These tasks start with the identification and assign a unique identifier to each of the requirement.
* After finalizing the requirement traceability table is developed.
* The examples of traceability table are the features, sources, dependencies, subsystems and interface of the requirement.

## Eliciting Requirements

Eliciting requirement helps the user for collecting the requirement.

Eliciting requirement steps are asfollows:  
***1. Collaborative requirements gathering***

* Gathering the requirements by conducting the meetings between developer and customer.
* Fix the rules for preparation and participation.
* The main motive is to identify the problem, give the solutions for the elements, negotiate the different approaches and specify the primary set of solution requirements in an environment which is valuable for achieving goal.

***2. Quality Function Deployment (QFD)***

* In this technique, translate the customer need into the technical requirement for the software.
* QFD system designs a software according to the demands of the customer.
* QFD consist of three types of requirement:
  + Normal requirements
* The objective and goal are stated for the system through the meetings with the customer.
* For the customer satisfaction these requirements should be there.

Expected requirement

* These requirements are implicit.
* These are the basic requirement that not be clearly told by the customer, but also the customer expect that requirement.

Exciting requirements

* These features are beyond the expectation of the customer.
* The developer adds some additional features or unexpected feature into the software to make the customer more satisfied.  
  For example, the mobile phone with standard features, but the developer adds few additional functionalities like voice searching, multi-touch screen etc. then the customer more excited about that feature.

***3. Usage scenarios***

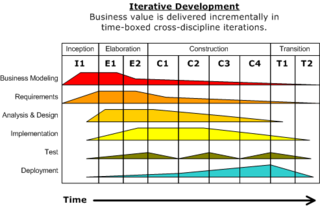
* Till the software team does not understand how the features and function are used by the end users it is difficult to move technical activities.
* To achieve above problem the software team produces a set of structure that identify the usage for the software.
* This structure is called as 'Use Cases'.

***4. Elicitation work product***

* The work product created as a result of requirement elicitation that is depending on the size of the system or product to be  built.
* The work product consists of a statement need, feasibility, statement scope for the system.
* It also consists of a list of users participate in the requirement elicitation
* **RUP**

The Rational Unified Process (RUP) is an [iterative](https://en.wikipedia.org/wiki/Iterative_and_incremental_development)[software development process](https://en.wikipedia.org/wiki/Software_development_process) framework created by the [Rational Software](https://en.wikipedia.org/wiki/Rational_Software) Corporation, a division of [IBM](https://en.wikipedia.org/wiki/IBM) since 2003.[[1]](https://en.wikipedia.org/wiki/Rational_Unified_Process#cite_note-1) RUP is not a single concrete prescriptive process, but rather an adaptable process [framework](https://en.wikipedia.org/wiki/Software_framework), intended to be tailored by the development organizations and software project teams that will select the elements of the process that are appropriate for their needs. RUP is a specific implementation of the [Unified Process](https://en.wikipedia.org/wiki/Unified_Process).

### Four project life-cycle phases



**RUP phases and disciplines**

The RUP has determined a project life-cycle consisting of four phases. These phases allow the process to be presented at a high level in a similar way to how a 'waterfall'-styled project might be presented, although in essence the key to the process lies in the iterations of development that lie within all of the phases. Also, each phase has one key objective and milestone at the end that denotes the objective being accomplished. The visualization of RUP phases and disciplines over time is referred to as the [RUP hump](https://en.wikipedia.org/wiki/RUP_hump) chart.

#### Inception phase

The primary objective is to scope the system adequately as a basis for validating initial costing and budgets. In this phase the business case which includes business context, success factors (expected revenue, market recognition, etc.), and financial forecast is established. To complement the business case, a basic use case model, project plan, initial risk assessment and project description (the core project requirements, constraints and key features) are generated. After these are completed, the project is checked against the following criteria:

* [Stakeholder](https://en.wikipedia.org/wiki/Stakeholder_%28corporate%29#In_management) concurrence on scope definition and cost/schedule estimates.
* Requirements understanding as evidenced by the fidelity of the primary use cases.
* Credibility of the cost/schedule estimates, priorities, risks, and development process.
* Depth and breadth of any architectural prototype that was developed.
* Establishing a baseline by which to compare actual expenditures versus planned expenditures.

If the project does not pass this milestone, called the life cycle objective milestone, it either can be cancelled or repeated after being redesigned to better meet the criteria.

#### Elaboration phase

The primary objective is to mitigate the key risk items identified by analysis up to the end of this phase. The elaboration phase is where the project starts to take shape. In this phase the problem domain analysis is made and the architecture of the project gets its basic form.

The outcome of the elaboration phase is:

* A use-case model in which the use-cases and the actors have been identified and most of the use-case descriptions are developed. The use-case model should be 80% complete.
* A description of the software architecture in a software system development process.
* An [executable architecture](https://en.wikipedia.org/wiki/Executable_Architecture) that realizes architecturally significant use cases.
* Business case and risk list which are revised.
* A development plan for the overall project.
* Prototypes that demonstrably mitigate each identified technical risk.
* A preliminary user manual (optional)

This phase must pass the lifecycle architecture milestone criteria answering the following questions:

* Is the vision of the product stable?
* Is the architecture stable?
* Does the executable demonstration indicate that major risk elements are addressed and resolved?
* Is the construction phase plan sufficiently detailed and accurate?
* Do all stakeholders agree that the current vision can be achieved using current plan in the context of the current architecture?
* Is the actual vs. planned resource expenditure acceptable?

If the project cannot pass this milestone, there is still time for it to be canceled or redesigned. However, after leaving this phase, the project transitions into a high-risk operation where changes are much more difficult and detrimental when made.

**The key domain analysis for the elaboration is the system architecture.**

#### Construction phase

The primary objective is to build the software system. In this phase, the main focus is on the development of components and other features of the system. This is the phase when the bulk of the coding takes place. In larger projects, several construction iterations may be developed in an effort to divide the use cases into manageable segments that produce demonstrable prototypes.

This phase produces the first external release of the software. Its conclusion is marked by the initial operational capability milestone.

#### Transition phase

The primary objective is to 'transit' the system from development into production, making it available to and understood by the end user. The activities of this phase include training the end users and maintainers and beta testing the system to validate it against the end users' expectations. The system also goes through an evaluation phase, any developer which is not producing the required work is replaced or removed. The product is also checked against the quality level set in the Inception phase.

If all objectives are met, the product release milestone is reached and the development cycle is finished

### Note : Introduction to UML, UML diagrams, purpose of UML ,use caseetc. refer PPT

### Use case diagrams

To model a system, the most important aspect is to capture the dynamic behavior. Dynamic behavior means the behavior of the system when it is running/operating.

Only static behavior is not sufficient to model a system rather dynamic behavior is more important than static behavior. In UML, there are five diagrams available to model the dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature, there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. Use case diagrams consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

Hence to model the entire system, a number of use case diagrams are used.

## Purpose of Use Case Diagrams

The purpose of use case diagram is to capture the dynamic aspect of a system. However, this definition is too generic to describe the purpose, as other four diagrams (activity, sequence, collaboration, and Statechart) also have the same purpose. We will look into some specific purpose, which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified.

When the initial task is complete, use case diagrams are modelled to present the outside view.

In brief, the ***purposes of use case diagrams*** can be said to be as follows −

* Used to gather the requirements of a system.
* Used to get an outside view of a system.
* Identify the external and internal factors influencing the system.
* Show the interaction among the requirements and actors.

## *How to Draw a Use Case Diagram?*

Use case diagrams are considered for high level requirement analysis of a system. When the requirements of a system are analyzed, the functionalities are captured in use cases.

We can say that use cases are nothing but the system functionalities written in an organized manner. The second thing which is relevant to use cases are the actors. Actors can be defined as something that interacts with the system.

Actors can be a human user, some internal applications, or may be some external applications. When we are planning to draw a use case diagram, we should have the following items identified.

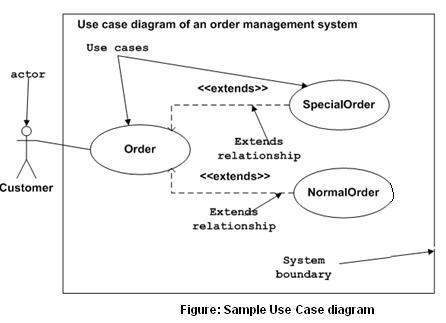
* Functionalities to be represented as use case
* Actors
* Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. After identifying the above items, we have to use the following guidelines to draw an efficient use case diagram

* The name of a use case is very important. The name should be chosen in such a way so that it can identify the functionalities performed.
* Give a suitable name for actors.
* Show relationships and dependencies clearly in the diagram.
* Do not try to include all types of relationships, as the main purpose of the diagram is to identify the requirements.
* Use notes whenever required to clarify some important points.

Following is a sample use case diagram representing the order management system. Hence, if we look into the diagram then we will find three use cases (Order, SpecialOrder, and NormalOrder) and one actor which is the customer.

The SpecialOrder and NormalOrder use cases are extended from *Order* use case. Hence, they have extended relationship. Another important point is to identify the system boundary, which is shown in the picture. The actor Customer lies outside the system as it is an external user of the system.



***Types of actors in use case diagrams:***

Actors can be primary or secondary actors.  Primary actors initiate a use case, while secondary actors support a use case or receive something of value from the use case.  While this answer might score you some points in the interview, there is another way to classify actors that is important to know and can show that you understand some of the finer points of use case diagramming.

Actors can be:

1. Human
2. Systems/Software
3. Hardware
4. Timer/Clock

Many analysts miss key actors during the use case diagramming process because they only identify human actors.  Categorizing use case actors in this ways helps the analyst ensure they haven’t overlooked any critical actors within the use case diagram.

* **use-case realization**

A use-case realization represents how a use case will be implemented in terms of collaborating objects. The realizations reside within the design. By walking through a design exercise of showing how the design elements will perform the use case, the team gets confirmation that the design is robust enough to perform the required behavior.

The realization can take various forms. It may include, for example, a textual description (a document), class diagrams of participating classes and subsystems, and interaction diagrams (communication and sequence diagrams) that illustrate the flow of interactions between class and subsystem instances.

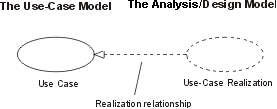
The reason for separating the use-case realization from its use case is that doing so allows the requirements, in the form of use cases, to be managed separately from the design, in the form of realizations. This decoupling will prove invaluable if the architecture is changed enough that the realization needs to be reworked while the requirements remain unaffected. Even without such a circumstance, the clear separation of concerns between requirements and design is valuable.

In a model, a use-case realization is represented as a UML (Unified Modeling Language) collaboration that groups the diagrams and other information (such as textual descriptions) that form the use-case realization.

Like other aspects of the design, the UML diagrams that support use-case realizations can be produced at various levels of abstraction. A first pass in creating the realization might produce a diagram at an analysis level of abstraction where the participants will be high-level elements that are expected to be revisited and detailed down to the design level in a second pass. If the architecture and design idioms are well-understood, the realization could immediately be created at a low level of abstraction that specifies more detail on the elements and how they will collaborate to realize the behavior of the use case. In the latter case, it is valuable to model patterns and architectural mechanisms to reduce the amount of low-level detail in each realization.

For each use case in the requirements, there can be a use-case realization in the design with a realization relationship to the use case, as the following figure shows. In UML, this is shown as a dashed arrow with an arrowhead, like a generalization relationship, indicating that a realization is a kind of inheritance, as well as a dependency (see the figure that follows).

**The UML notation for use-case realization**



# Activity Diagrams

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent.

## Purpose of Activity Diagrams

It captures the dynamic behaviour of the system. Activity is a particular operation of the system. Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in the activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is sometimes considered as the flowchart. Although the diagrams look like a flowchart, they are not. It shows different flows such as parallel, branched, concurrent, and single.

The purpose of an activity diagram can be described as −

* Draw the activity flow of a system.
* Describe the sequence from one activity to another.
* Describe the parallel, branched and concurrent flow of the system.

## How to Draw an Activity Diagram?

Activity diagrams are mainly used as a flowchart that consists of activities performed by the system. Activity diagrams are not exactly flowcharts as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane, etc.

Before drawing an activity diagram, we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities, we need to understand how they are associated with constraints and conditions.

Before drawing an activity diagram, we should identify the following elements −

* Activities
* Association
* Conditions
* Constraints

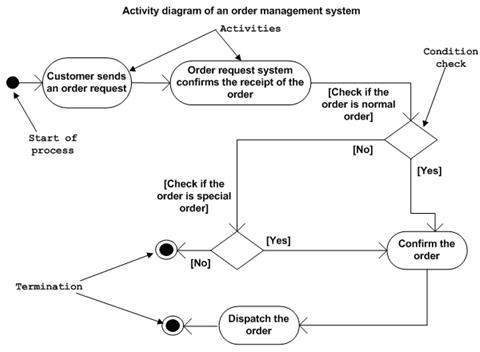
Once the above-mentioned parameters are identified, we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

Following is an example of an activity diagram for order management system. In the diagram, four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and is mainly used by the business users

Following diagram is drawn with the four main activities −

* Send order by the customer
* Receipt of the order
* Confirm the order
* Dispatch the order

After receiving the order request, condition checks are performed to check if it is normal or special order. After the type of order is identified, dispatch activity is performed and that is marked as the termination of the process.



## Where to Use Activity Diagrams?

The basic usage of activity diagram is similar to other four UML diagrams. The specific usage is to model the control flow from one activity to another. This control flow does not include messages.

Activity diagram is suitable for modeling the activity flow of the system. An application can have multiple systems. Activity diagram also captures these systems and describes the flow from one system to another. This specific usage is not available in other diagrams. These systems can be database, external queues, or any other system.

**Activity diagram can be used for −**

* Modeling work flow by using activities.
* Modeling business requirements.
* High level understanding of the system's functionalities.
* Investigating business requirements at a later stage.

### Activity Diagram Notations –

1. **Initial State –** The starting state before an activity takes place is depicted using the initial state.

UML-State-Diagram  
**Figure –** notation for initial state or start state

A process can have only one initial state unless we are depicting nested activities. We use a black filled circle to depict the initial state of a system. For objects, this is the state when they are instantiated. The Initial State from the UML Activity Diagram marks the entry point and the initial Activity State.

For example – Here the initial state is the state of the system before the application is opened.

  
**Figure –** initial state symbol being used

1. **Action or Activity State –** An activity represents execution of an action on objects or by objects. We represent an activity using a rectangle with rounded corners. Basically any action or event that takes place is represented using an activity.

UML-Activity-Diagram  
**Figure –** notation for an activity state

For example – Consider the previous example of opening an application opening the application is an activity state in the activity diagram.

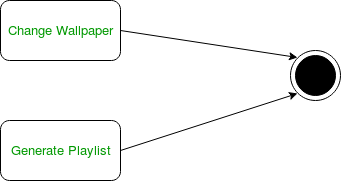
  
**Figure –** activity state symbol being used

1. **Action Flow or Control flows –** Action flows or Control flows are also referred to as paths and edges. They are used to show the transition from one activity state to another.

UML-Object-Diagram  
**Figure –** notation for control Flow

An activity state can have multiple incoming and outgoing action flows. We use a line with an arrow head to depict a Control Flow. If there is a constraint to be adhered to while making the transition it is mentioned on the arrow.

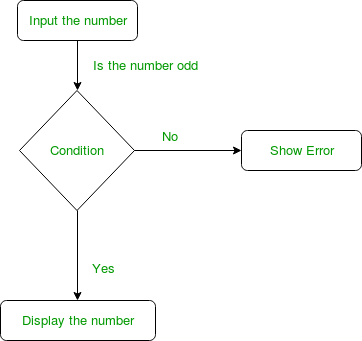
Consider the example – Here both the states transit into one final state using action flow symbols i.e. arrows.

  
**Figure –** using action flows for transitions

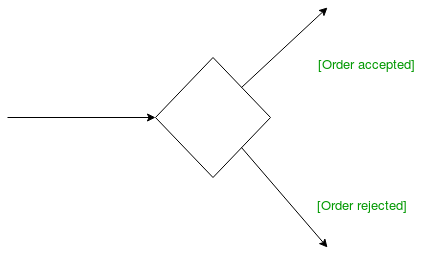
1. **Decision node and Branching –** When we need to make a decision before deciding the flow of control, we use the decision node.

  
**Figure –** notation for decision node

The outgoing arrows from the decision node can be labelled with conditions or guard expressions.It always includes two or more output arrows.

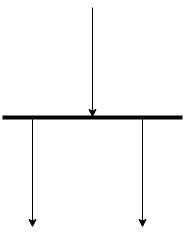
  
**Figure –** an activity diagram using decision node

1. **Guards –** A Guard refers to a statement written next to a decision node on an arrow sometimes within square brackets.

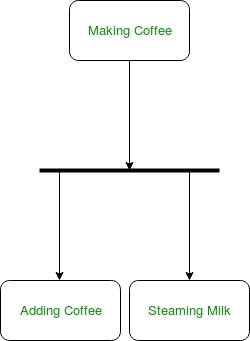
  
**Figure –** guards being used next to a decision node

The statement must be true for the control to shift along a particular direction. Guards help us know the constraints and conditions which determine the flow of a process.

1. **Fork –** Fork nodes are used to support concurrent activities.

  
**Figure –** fork notation

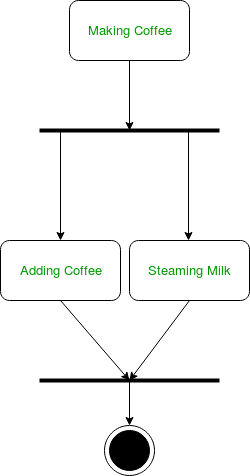
When we use a fork node when both the activities get executed concurrently i.e. no decision is made before splitting the activity into two parts. Both parts need to be executed in case of a fork statement.  
We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent activity state and outgoing arrows towards the newly created activities.  
For example: In the example below, the activity of making coffee can be split into two concurrent activities and hence we use the fork notation.

  
**Figure –** a diagram using fork

1. **Join –** Join nodes are used to support concurrent activities converging into one. For join notations we have two or more incoming edges and one outgoing edge.

  
**Figure –** join notation

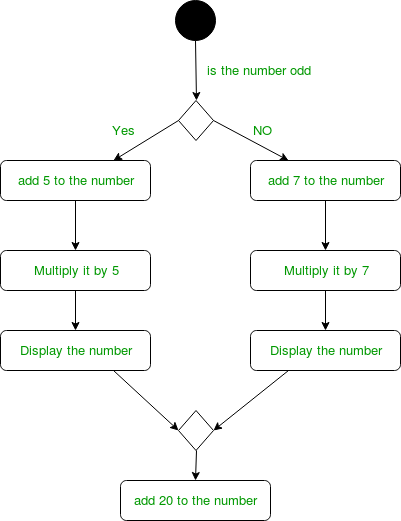
For example – When both activities i.e. steaming the milk and adding coffee get completed, we converge them into one final activity.

  
**Figure –** a diagram using join notation

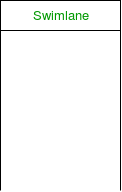
1. **Merge or Merge Event –**Scenarios arise when activities which are not being executed concurrently have to be merged. We use the merge notation for such scenarios. We can merge two or more activities into one if the control proceeds onto the next activity irrespective of the path chosen.

  
**Figure –** merge notation

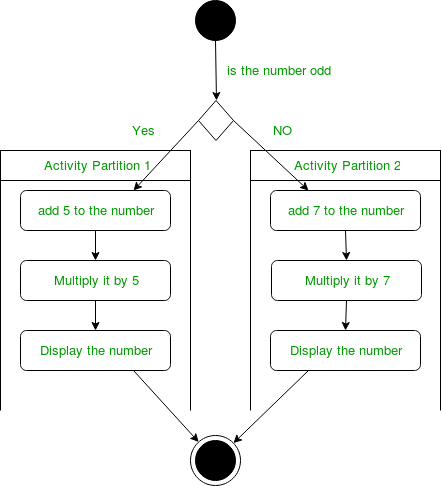
For example – In the diagram below: we can’t have both sides executing concurrently, but they finally merge into one. A number can’t be both odd and even at the same time.

  
**Figure –** an activity diagram using merge notation

1. **Swimlanes –** We use swimlanes for grouping related activities in one column. Swimlanes group related activities into one column or one row. Swimlanes can be vertical and horizontal. Swimlanes are used to add modularity to the activity diagram. It is not mandatory to use swimlanes. They usually give more clarity to the activity diagram. It’s similar to creating a function in a program. It’s not mandatory to do so, but, it is a recommended practice.

  
**Figure –** swimlanes notation  
We use a rectangular column to represent a swimlane as shown in the figure above.

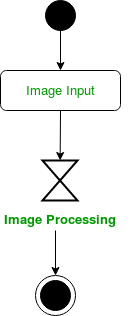
For example – Here different set of activities are executed based on if the number is odd or even. These activities are grouped into a swimlane.

  
**Figure –** an activity diagram making use of swimlanes

1. **Time Event –**

  
**Figure –** time event notation

We can have a scenario where an event takes some time to complete. We use an hourglass to represent a time event.  
For example – Let us assume that the processing of an image takes takes a lot of time. Then it can be represented as shown below.

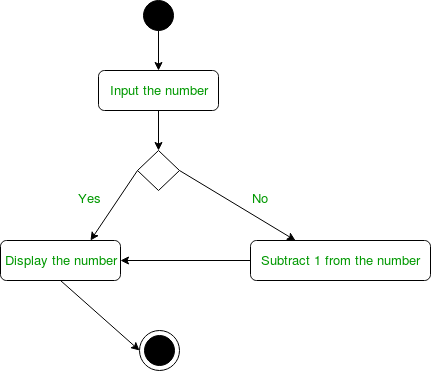
  
**Figure –** an activity diagram using time event

1. **Final State or End State –** The state which the system reaches when a particular process or activity ends is known as a Final State or End State. We use a filled circle within a circle notation to represent the final state in a state machine diagram. A system or a process can have multiple final states.

UML-State-Diagram  
**Figure –** notation for final state

### How to Draw an activity diagram –

1. Identify the initial state and the final states.
2. Identify the intermediate activities needed to reach the final state from he initial state.
3. Identify the conditions or constraints which cause the system to change control flow.
4. Draw the diagram with appropriate notations.

  
**Figure –** an activity diagram

The above diagram prints the number if it is odd otherwise it subtracts one from the number and displays it.

### Uses of an Activity Diagram –

* Dynamic modelling of the system or a process.
* Illustrate the various steps involved in a UML use case.
* Model software elements like methods,operations and functions.
* We can use Activity diagrams to depict concurrent activities easily.
* Show the constraints, conditions and logic behind algorithms.

**UNIT 3 :**

* **Class diagrams- identifying classes,objects,relations.**
* **Advanced class modeling- interface, types , roles**
* **Object diagrams**
* **State chart diagrams**
* **Package diagrams**

## *Classes*

In the hr package we might find the Position and Person classes. Typical attributes of persons and positions can be listed in the attribute compartments of the respective class icons. Typical operations performed by persons and positions can be listed in their operation compartments:



In this example, an object representing a position, vice president for example, has a title attribute of type String and an execute operation that takes a task, t, as an input, and outputs a result.

An object representing a person has four attributes: last name, first name, and Social Security number, all of type String, and a salary of type Money and with initial value $0.0. The person class has no operations specified at this point.

It is often the case that operations correspond to responsibilities and are not assigned to classes until the design phase. For this reason operation compartments are often suppressed in domain models. Attribute compartments can also be suppressed if they are not needed.

Note that the "-" sign indicates that the scope of an attribute or association endpoint is private, visible only inside of the class. Other possible scopes are:

+ = public = system-wide scope  
~ = package scope  
# = protected = subclass scope

Generally speaking, the default for an attribute or association endpoint is private, while the default for an operation is public.

## *Associations*

The relationship "Position X is filled by person Y" can be represented by an association connecting Position and Person:



Note that the name of an association can be decorated by a tiny arrow showing which way the association is read. This helps distinguish a relationship from its inverse.

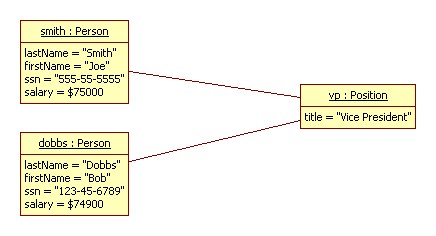
Note that the endpoints of an association can be decorated by:

name = the name used by objects on one end to refer to objects on the other end. For example, an employee refers to his position as his "role" while a position refers to the employees filling it as its "actors."

multiplicity = the number of objects that can be related to a given object. For example, a given position, vice president, say, can be filled by by 0 or more persons (indicated by "\*"), while a given person, Joe Smith, say, can fill one position.

## Objects

Specific persons and positions can be represented using objects. Specific instances of the filledBy relationship can be represented by links connecting the participating objects:



**State chart diagram**

The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a component in a system. The states are specific to a component/object of a system.

A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.

Activity diagram explained in the next chapter, is a special kind of a Statechart diagram. As Statechart diagram defines the states, it is used to model the lifetime of an object.

## Purpose of Statechart Diagrams

Statechart diagram is one of the five UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events. Statechart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.

Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

**Following are the main purposes of using Statechart diagrams −**

* To model the dynamic aspect of a system.
* To model the life time of a reactive system.
* To describe different states of an object during its life time.
* Define a state machine to model the states of an object.

## How to Draw a Statechart Diagram?

Statechart diagram is used to describe the states of different objects in its life cycle. Emphasis is placed on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

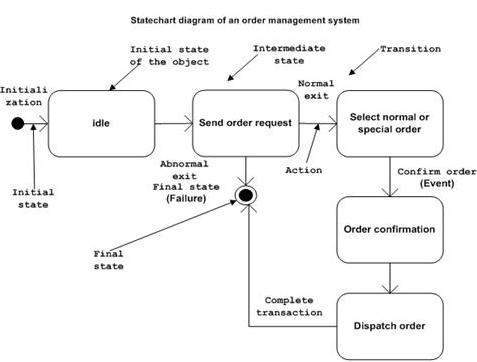
Before drawing a Statechart diagram we should clarify the following points −

* Identify the important objects to be analyzed.
* Identify the states.
* Identify the events.

***Following is an example of a Statechart diagram where the state of Order object is analyzed***

The first state is an idle state from where the process starts. The next states are arrived for events like send request, confirm request, and dispatch order. These events are responsible for the state changes of order object.

During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exits. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete, it is considered as a complete transaction as shown in the following figure. The initial and final state of an object is also shown in the following figure.

****

## Where to Use Statechart Diagrams?

From the above discussion, we can define the practical applications of a Statechart diagram. Statechart diagrams are used to model the dynamic aspect of a system like other four diagrams discussed in this tutorial. However, it has some distinguishing characteristics for modeling the dynamic nature.

Statechart diagram defines the states of a component and these state changes are dynamic in nature. Its specific purpose is to define the state changes triggered by events. Events are internal or external factors influencing the system.

Statechart diagrams are used to model the states and also the events operating on the system. When implementing a system, it is very important to clarify different states of an object during its life time and Statechart diagrams are used for this purpose. When these states and events are identified, they are used to model it and these models are used during the implementation of the system.

If we look into the practical implementation of Statechart diagram, then it is mainly used to analyze the object states influenced by events. This analysis is helpful to understand the system behavior during its execution.

The main usage can be described as −

* To model the object states of a system.
* To model the reactive system. Reactive system consists of reactive objects.
* To identify the events responsible for state changes.
* Forward and reverse engineering.

## What are the elements in State Chart diagrams?

- Initial State: This state shows the first activity of the flow.  
- State: A state represents the state of an object at a particular given point of time.  
- Transition: The transition from one state to another state of objects is represented by an arrow.  
- Event and Action: A trigger that causes a transition to occur.  
- Signal: When a message or a trigger caused by an event to a state, which causes a transition, this message is called as a signal.  
- Final State: The state diagram ends with a diagram that depicts a bull’s eye is known as Final State

**Difference between activity diagram and state chart diagram**

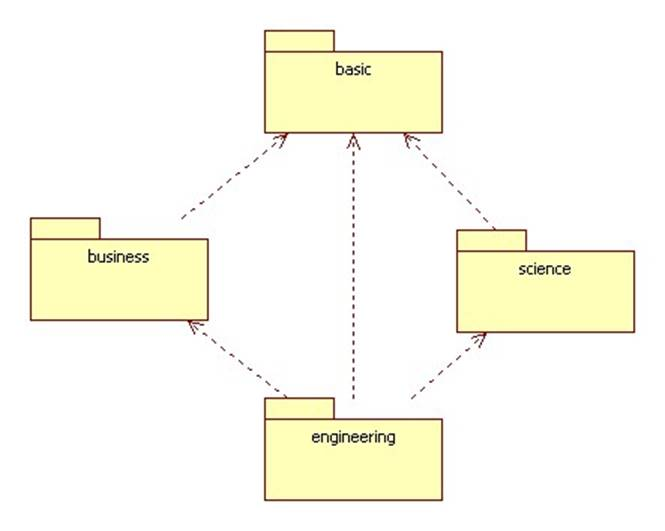
 But the difference is, activity diagrams describe activities and state chartsdescribe states. So those models are orthogonal - you might imagine there is an activity between two states (somethign that occours during the transition) and you might say there is a state between two activities. This is rather a simplification, but I think it might suffice to explain the difference, which is primarily focus, both diagrams are for describing behaviours.

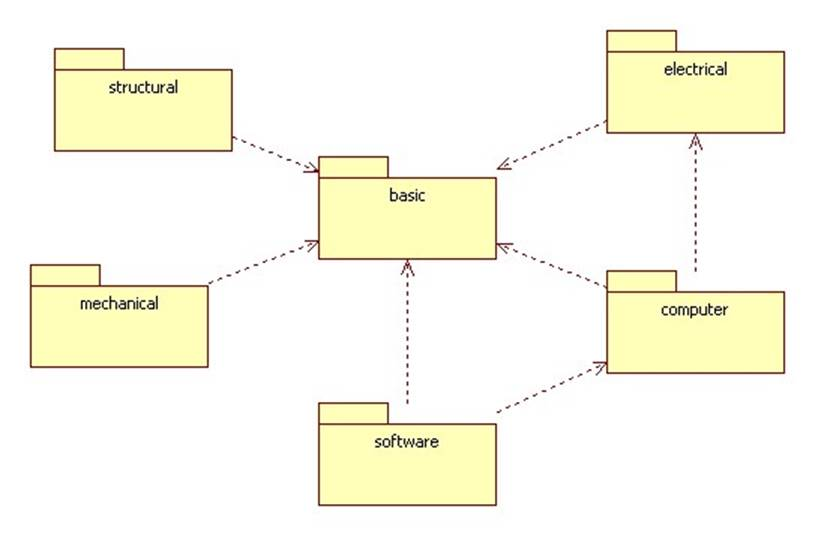
**Package diagrams**

A package diagram is a class diagram that shows packages and the dependencies between them.

A dependency between two packages is represented by a dashed arrow. More specifically, dashed arrow from package A to package B indicates that package A uses some of the items contained in package B. We say that A imports from B or that B exports to A.

Here are a few examples:





**UNIT - 4:**

* **Interaction diagrams and its need**
* **Collaboration diagrams**
* **Sequence diagrams**

## Elaboration Iteration 1 Requirements and Emphasis: Core OOA/D Skills

In these case studies, iteration - 1 of the elaboration phase emphasizes a range of fundamental and common OOA / D skills used in building object systems. Many other skills and £teps - such as database design, usability engineering, and UI design - are of course needed to build software, but they are out of scope in this introduction focusing on OOA / D and applying the UML.

**NextGen POS**

The requirements for the first iteration of the NextGen POS application follow:

Implement a basic, key scenario of the Process Sale use case: entering items and receiving a cash payment.Implement a Start Up use case as necessary to support the initialization needs of the iteration.Nothing fancy or complex is handled, just a simple happy path scenario, and the design and implementation to support it.There is no collaboration with external services, such as a tax calculator or product database.No complex pricing rules are applied.

The design and implementation of the supporting UI, database, and so forth, would also be done, but is not covered in any detail.

**Monopoly**

The requirements for the first iteration of the Monopoly application follow:

Implement a basic, key scenario of the Play Monopoly Game use case: players moving around the squares of the board.Implement a Start Up use case as necessary to support the initialization needs of the iteration.Two to eight players can play.A game is played as a series of rounds. During a round, each player takes one turn. In each turn, a player advances his piece clockwise around the board a number of squares equal to the sum of the number rolled on two six - sided dice.Play the game for only 20 rounds.After the dice are rolled, the name of the player and the roll are displayed. When the player moves and lands on a square, the name of the player and the name of the square that the player landed on are displayed.In iteration - 1 there is no money, no winner or loser, no properties to buy or rent to pay, and no special squares of any kind.Each square has a name. Every player begins the game with their piece located on the square named "Go." The square names will be Go, Square 1, Square 2, ... Square 39Run the game as a simulation requiring no user input, other than the number of players.

Subsequent iterations will grow on these foundations.

**In Iterative Development We Don't Implement All the Requirements at Once**

Note that these requirements for iteration - 1 are subsets of the complete requirements or use cases. For example, the NextGen POS iteration - 1 requirements are a simplified version of the complete Process Sale use case; they describe one simple cash - only scenario.

Note also that we haven't done all the requirements analysis for the NextGen POS system, we've only analyzed the process Sale use case in detail; many others are not yet analyzed.

This is a key understanding in iterative lifecycle methods (such as the UP, XP, Scrum, and so forth): We start production - quality programming and testing for a subset of the requirements, and we start that development before all the requirements analysis is complete - in contrast to a waterfall process.

Incremental Development for the Same Use Case Across Iterations

Notice that not all requirements in the Process Sale use case are being implemented in iteration - 1. It is common to work on varying scenarios of the same use case over several iterations and gradually extend the system to ultimately handle all the functionality required. On the other hand, short, simple use cases may be completed within one iteration.

Use case implementation may be spread across iterations

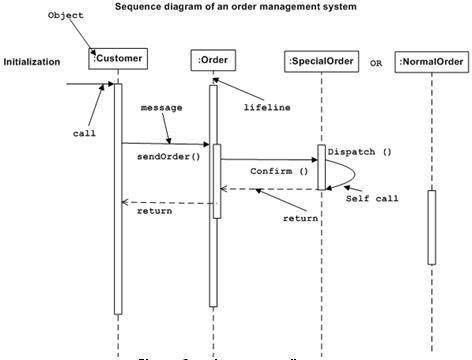
State sequence diagrams

### The Sequence Diagram

The sequence diagram has four objects (Customer, Order, SpecialOrder and NormalOrder).

The following diagram shows the message sequence for SpecialOrder object and the same can be used in case of NormalOrder object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

The first call is sendOrder () which is a method of Order object. The next call is confirm () which is a method of SpecialOrder object and the last call is Dispatch () which is a method of SpecialOrder object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.



### What is a Sequence Diagram?

Sequence diagrams, commonly used by developers, model the interactions between objects in a single use case. They illustrate how the different parts of a system interact with each other to carry out a function, and the order in which the interactions occur when a particular use case is executed.

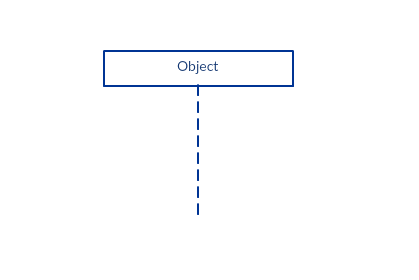
In simpler words, a sequence diagram shows different parts of a system work in a ‘sequence’ to get something done.

### Sequence Diagram Notations

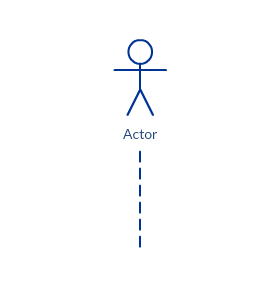
A sequence diagram is structured in such a way that it represents a timeline which begins at the top and descends gradually to mark the sequence of interactions. Each object has a column and the messages exchanged between them are represented by arrows.

A Quick Overview of the Various Parts of a Sequence Diagram

Lifeline Notation

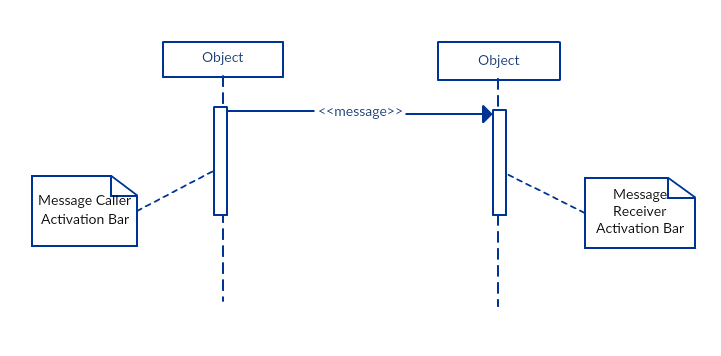
A sequence diagram is made up of several of these lifeline notations that should be arranged horizontally across the top of the diagram. No two lifeline notations should overlap each other. They represent the different objects or parts that interact with each other in the system during the sequence.

A lifeline notation with an actor element symbol is used when the particular sequence diagram is owned by a use case.

Activation Bars

Activation bar is the box placed on the lifeline.  It is used to indicate that an object is active (or instantiated) during an interaction between two objects. The length of the rectangle indicates the duration of the objects staying active.

In a sequence diagram, an interaction between two objects occurs when one object sends a message to another. The use of the activation bar on the lifelines of the Message Caller (the object that sends the message) and the Message Receiver (the object that receives the message) indicates that both are active/is instantiated during the exchange of the message.



Message Arrows

An arrow from the Message Caller to the Message Receiver specifies a message in a sequence diagram.   A message can flow in any direction; from left to right, right to left or back to the Message Caller itself. While you can describe the message being sent from one object to the other on the arrow, with different arrowheads you can indicate the type of message being sent or received.

The message arrow comes with a description, which is known as a message signature, on it. The format for this message signature is below. All parts except the message\_name is optional.

attribute = message\_name (arguments): return\_type

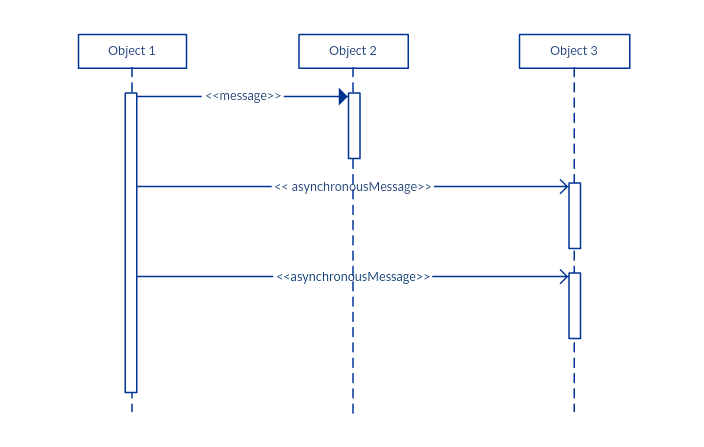
* Synchronous message

As shown in the activation bars example, a synchronous message is used when the sender waits for the receiver to process the message and return before carrying on with another message.  The arrow head used to indicate this type of message is a solid one, like the one below.



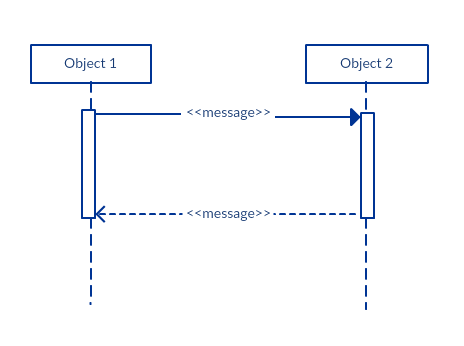
* Asynchronous message

An asynchronous message is used when the message caller does not wait for the receiver to process the message and return before sending other messages to other objects within the system. The arrow head used to show this type of message is a line arrow like shown in the example below.



* Return message

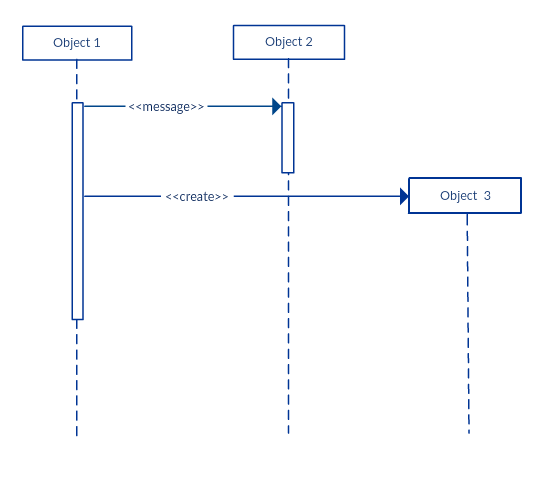
A return message is used to indicate that the message receiver is done processing the message and is returning control over to the message caller. Return messages are optional notation pieces, for an activation bar that is triggered by a synchronous message always implies a return message.

Tip: You can avoid cluttering up your diagrams by minimizing the use of return messages since the return value can be specified in the initial message arrow itself.  


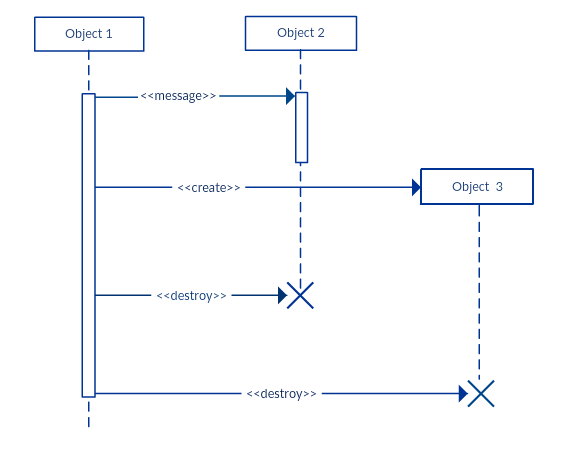
* Participant  creation message

Objects do not necessarily live for the entire duration of the sequence of events. Objects or participants can be created according to the message that is being sent.

The dropped participant box notation can be used when you need to show that the particular participant did not exist until the create call was sent.  If the created participant does something immediately after its creation, you should add an activation box right below the participant box.

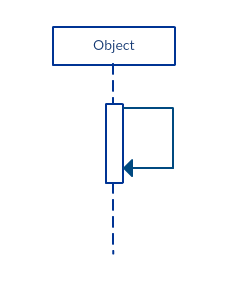


* Participant destruction message

Likewise, participants when no longer needed can also be deleted from a sequence diagram. This is done by adding an ‘X’ at the end of the lifeline of the said participant. 

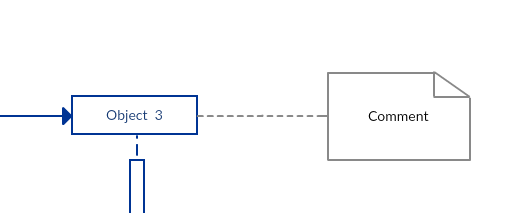
* Reflexive message

When an object sends a message to itself, it is called a reflexive message. It is indicated with a message arrow that starts and ends at the same lifeline like shown in the example below.



Comment

[UML diagrams](https://creately.com/Draw-UML-and-Class-Diagrams-Online) generally permit the annotation of comments in all [UML diagram types](http://creately.com/blog/diagrams/uml-diagram-types-examples/). The comment object is a rectangle with a folded-over corner as shown below. The comment can be linked to the related object with a dashed line.



Note:  View Sequence Diagram Best Practices to learn about sequence fragments.

### Sequence Diagram Best Practices

* Manage complex interactions with sequence fragments

A sequence fragment is represented as a box that frames a section of interactions between objects (as shown in the examples below) in a sequence diagram.

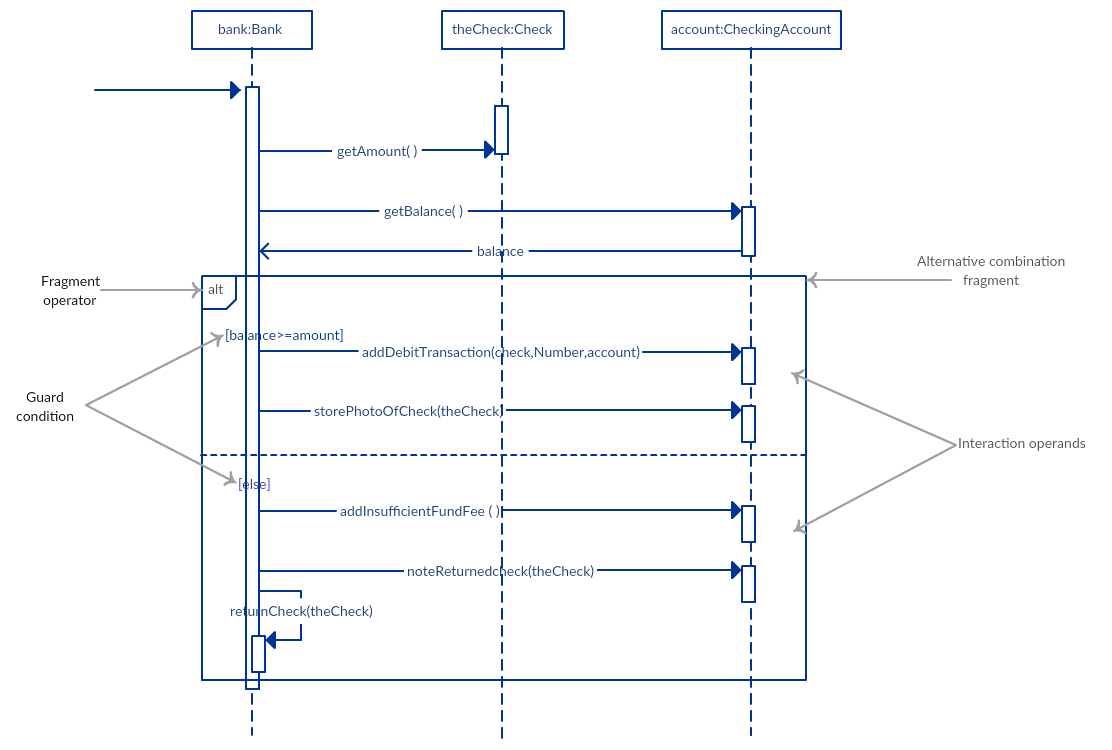
It is used to show complex interactions such as alternative flows and loops in a more structured way. On the top left corner of the fragment sits an operator. This – the fragment operator – specifies what sort of a fragment it is.

Alternatives

The alternative combination fragment is used when a choice needs to be made between two or more message sequences. It models the “if then else” logic.

The alternative fragment is represented by a large rectangle or a frame; it is specified by mentioning ‘alt’ inside the frame’s name box (a.k.a. fragment operator).

To show two or more alternatives, the larger rectangle is then divided into what is called interaction operands using a dashed line, like shown in the sequence diagram example above. Each operand has a guard to test against and it is placed at the top left corner of the operand.



Options

The option combination fragment is used to indicate a sequence that will only occur under a certain condition, otherwise, the sequence won’t occur. It models the “if then” statement.

Similar to the alternative fragment, the option fragment is also represented with a rectangular frame where ‘opt’ is placed inside the name box.

Unlike the alternative fragment, an option fragment is not divided into two or more operands. Option’s guard is placed at the top left corner.

(Find an example sequence diagram with an option fragment in the  Sequence Diagram Templates and Examples section).

Loops

Loop fragment is used to represent a repetitive sequence. Place the words ‘loop’ in the name box and the guard condition near the top left corner of the frame.

In addition to the Boolean test, the guard in a loop fragment can have two other special conditions tested against. These are minimum iterations (written as minint = [the number] and maximum iterations (written as maxint = [the number]).

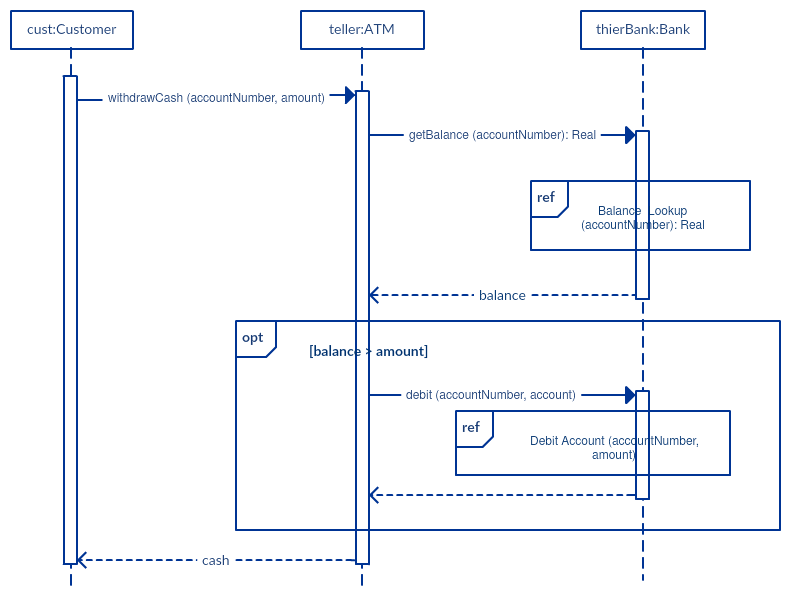
If it is a minimum iterations guard, the loop must execute not less than the number mentioned, and if it is a maximum iterations guard, the loop mustn’t execute more than the number indicated.

(Find an example of a loop fragment below in the sequence diagram templates and example section)

Reference Fragment

You can use the ref fragment to manage the size of large sequence diagrams. It allows you to reuse part of one sequence diagram in another, or in other words, you can reference part of a diagram in another diagram using the ref fragment.

To specify the reference fragment, you have to mention ‘ref’ in the name box of the frame and the name of the sequence diagram that is being referred to inside the frame.



For more sequence fragments refer to Beyond the Basics of Sequence Diagrams: [Part 1](http://creately.com/blog/diagrams/beyond-the-basics-of-sequence-diagrams-part-1/), [Part 2](http://creately.com/blog/diagrams/beyond-the-basics-of-sequence-diagrams-part-2/)and [Part 3](http://creately.com/blog/diagrams/beyond-the-basics-of-sequence-diagrams-part-3/).

* Draw smaller sequence diagrams that capture the essence of the use case

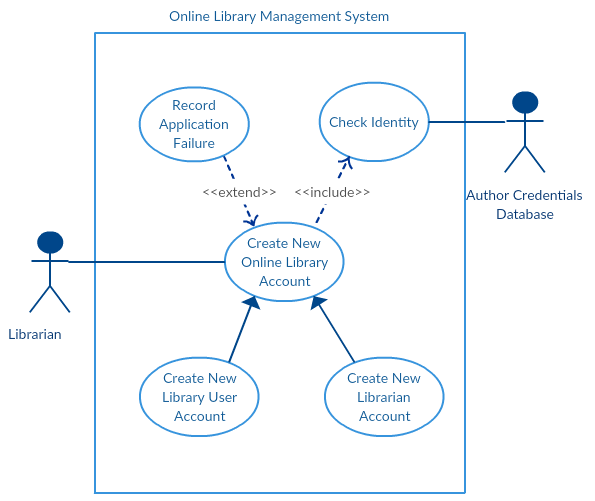
Instead of cluttering your sequence diagram with several objects and groups of messages that will confuse the reader, draw a few smaller sequence diagrams that aptly explain what your system does.  Make sure that the diagram fits on a single page and leaves space for explanatory notes too.

Also instead of drawing dozens of sequence diagrams, find out what is common among the scenarios and focus on that. And if the code is expressive and can stand on its own, there’s no need to draw a sequence diagram in the first place.

### How to Draw a Sequence Diagram

A sequence diagram represents the scenario or flow of events in one single use case. The message flow of the sequence diagram is based on the narrative of the particular use case.

Then, before you start drawing the sequence diagram or decide what interactions should be included in it, you need to [draw the use case diagram](https://creately.com/diagram-type/use-case) and ready a comprehensive description of what the particular use case does.



From the above use case diagram example of ‘Create New Online Library Account’, we will focus on the use case named ‘Create New User Account’ to draw our sequence diagram example.

Before drawing the sequence diagram, it’s necessary to identify the objects or actors that would be involved in creating a new user account. These would be;

* Librarian
* Online Library Management system
* User credentials database
* Email system

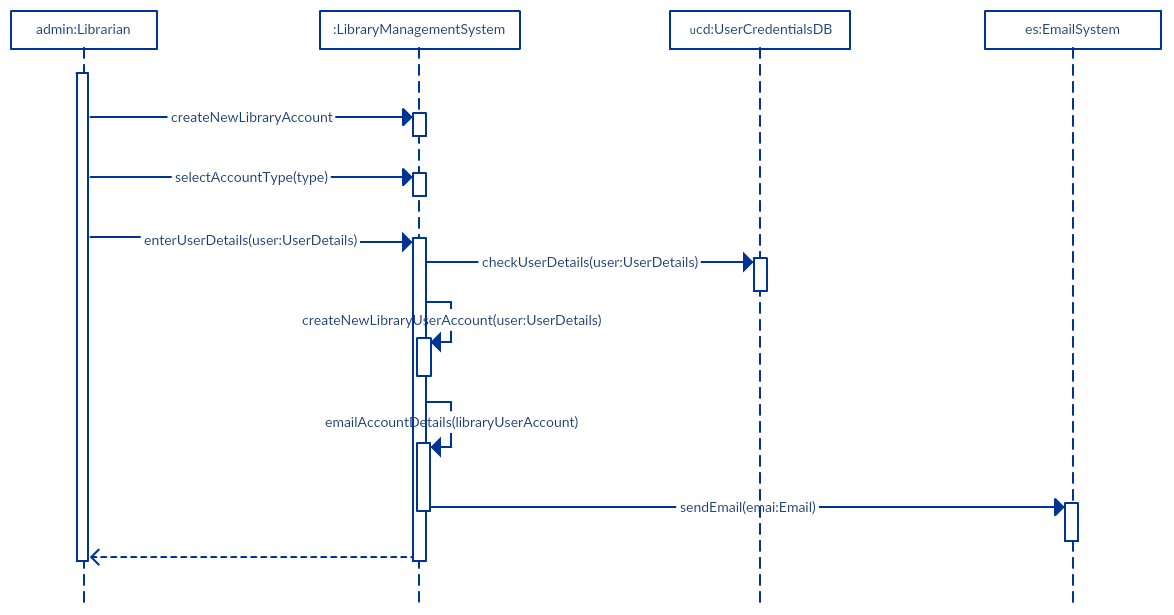
Once you identify the objects, it is then important to write a detailed description on what the use case does. From this description, you can easily figure out the interactions (that should go in the sequence diagram) that would occur between the objects above, once the use case is executed.

Here are the steps that occur in the use case named ‘Create New Library User Account’.

* The librarian request the system to create a new online library account
* The librarian then selects the library user account type
* The librarian enters the user’s details
* The user’s details are checked using the user Credentials Database
* The new library user account is created
* A summary of the of the new account’s details are then emailed to the user

From each of these steps, you can easily specify what messages should be exchanged between the objects in the sequence diagram. Once it’s clear, you can go ahead and start drawing the sequence diagram.

The sequence diagram below shows how the objects in the online library management system interact with each other to perform the function ‘Create New Library User Account’.



### Sequence Diagram Common Mistakes

When drawing sequence diagrams, designers tend to make these common mistakes. By avoiding these mistakes you can ensure the quality of your diagram.

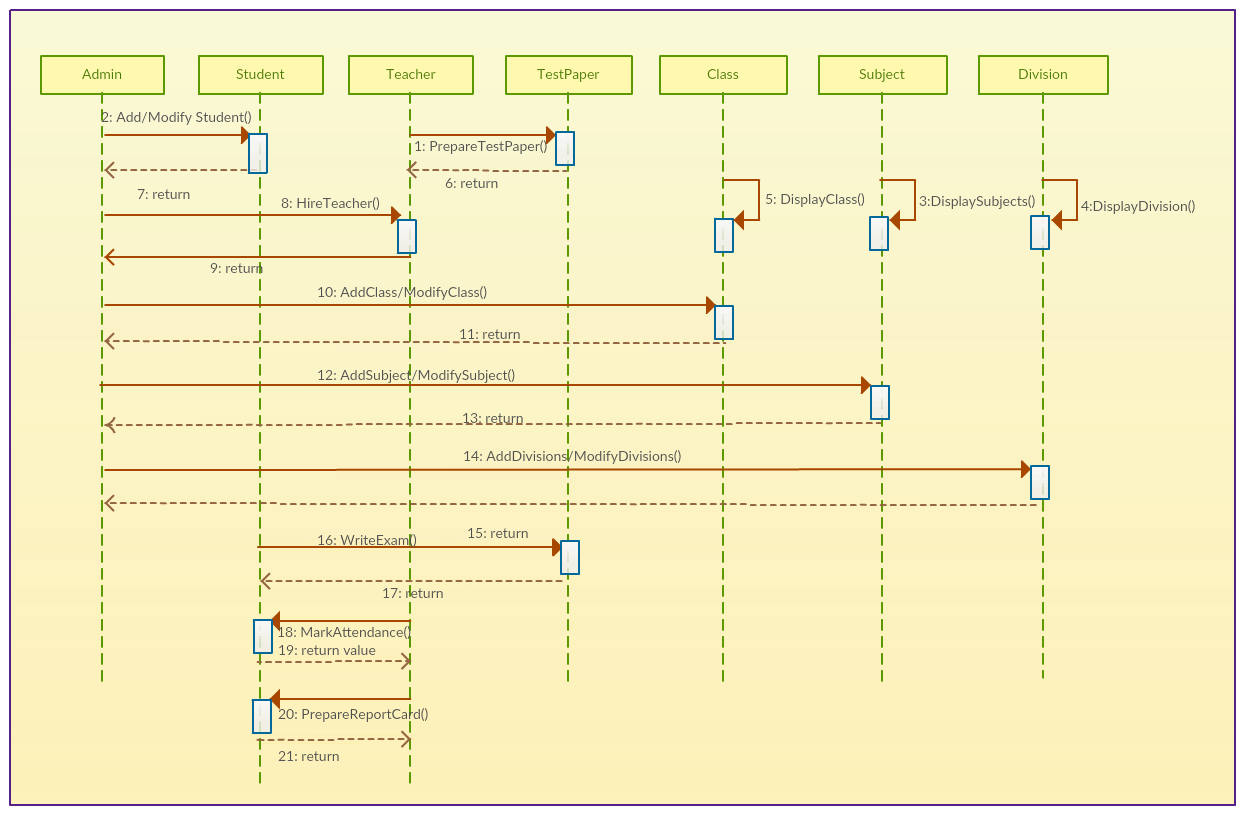
* Adding too much detail. This clutters up the diagram and makes it difficult to read.
* Obsolete and out of date sequence diagrams that are irrelevant when compared to the interfaces, actual architectures etc. of the system. Don’t forget to replace them or modify them.
* Leaving no blank space between the use case text and the message arrow; this makes it difficult for anyone to read the diagram.
* Not considering the origins of message arrows carefully.

See these common mistakes explained in detail in Sequence Diagram Guide: [Common Mistakes to Avoid When Drawing Sequence Diagrams](http://creately.com/blog/diagrams/10-common-mistakes-to-avoid-in-sequence-diagrams/).

### Sequence Diagram Templates and examples

Following are a few sequence diagram templates and examples that are drawn using Creately. [Create sequence diagrams online](https://creately.com/diagram-type/uml-sequence-diagrams) using Creately’s online tool. Click on the template to open it in the editor.

Sequence Diagram of an Online Exam System



COLLABORATION DAIGRAMS

Collaboration diagrams (**known as Communication Diagram in UML 2.x**) are used to show how objects interact to perform the behavior of a particular use case, or a part of a use case. Along with sequence diagrams, collaboration are used by designers to define and clarify the roles of the objects that perform a particular flow of events of a use case.  They are the primary source of information used to determining class responsibilities and interfaces.

## What is a Collaboration?

* A Collaboration is a collection of named objects and actors with links connecting them. They collaborate in performing some task.
* A Collaboration defines a set of participants and relationships that are meaningful for a given set of purposes
* A Collaboration between objects working together provides emergent desirable functionalities in Object-Oriented systems
* Each object (responsibility) partially supports emergent functionalities
* Objects are able to produce (usable) high-level functionalities by working together
* Objects collaborate by communicating (passing messages) with one another in order to work together

## Why Collaboration Diagram?

Unlike a sequence diagram, a collaboration diagram shows the relationships among the objects. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.

Because of the format of the collaboration diagram, they tend to better suited for analysis activities (see Activity: Use-Case Analysis).   Specifically, they tend to be better suited to depicting simpler interactions of smaller numbers of objects.  However, if the number of objects and messages grows, the diagram becomes increasingly hard to read.  In addition, it is difficult to show additional descriptive information such as timing, decision points, or other unstructured information that can be easily added to the notes in a sequence diagram. So, here are some use cases that we want to create a collaboration diagram for:

* Model collaborations between objects or roles that deliver the functionalities of use cases and operations
* Model mechanisms within the architectural design of the system
* Capture interactions that show the messages passing between objects and roles within the collaboration
* Model alternative scenarios within use cases or operations that involve the collaboration of different objects and interactions
* Support the identification of objects (hence classes) that participate in use cases
* Each message in a collaboration diagram has a sequence number.
* The top-level message is numbered 1. Messages sent during the same call have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.

## Notations of Collaboration Diagram

### Objects

An object is represented by an object symbol showing the name of the object and its class underlined, separated by a colon:

Object\_name : class\_name

You can use objects in collaboration diagrams in the following ways:

* Each object in the collaboration is named and has its class specified
* Not all classes need to appear
* There may be more than one object of a class
* An object’s class can be unspecified. Normally you create a collaboration diagram with objects first and specify their classes later.
* The objects can be unnamed, but you should name them if you want to discriminate different objects of the same class.

### Actors

Normally an actor instance occurs in the collaboration diagram, as the invoker of the interaction. If you have several actor instances in the same diagram, try keeping them in the periphery of the diagram.

* Each Actor is named and has a role
* One actor will be the initiator of the use case

### Links

Links connect objects and actors and are instances of associations and each link corresponds to an association in the class diagram

Links are defined as follows:

* A link is a relationship among objects across which messages can be sent. In collaboration diagrams, a link is shown as a solid line between two objects.
* An object interacts with, or navigates to, other objects through its links to these objects.
* A link can be an instance of an association, or it can be anonymous, meaning that its association is unspecified.
* Message flows are attached to links, see Messages.

### Messages

A message is a communication between objects that conveys information with the expectation that activity will ensue. In collaboration diagrams, a message is shown as a labeled arrow placed near a link.

* The message is directed from sender to receiver
* The receiver must understand the message
* The association must be navigable in that direction

## Steps for Creating Collaboration Diagrams

1. Identify behavior whose realization and implementation is specified
2. Identify the structural elements (class roles, objects, subsystems) necessary to carry out the functionality of the collaboration
   * Decide on the context of interaction: system, subsystem, use case and operation
3. Model structural relationships between those elements to produce a diagram showing the context of the interaction
4. Consider the alternative scenarios that may be required
   * Draw instance level collaboration diagrams, if required.
   * Optionally draw a specification level collaboration diagram to summarize the alternative scenarios in the instance level sequence diagrams

## Benefits of a Collaboration Diagram

1. The collaboration diagram is also known as Communication Diagram.
2. It mainly puts emphasis on the structural aspect of an interaction diagram, i.e., how lifelines are connected.
3. The syntax of a collaboration diagram is similar to the sequence diagram; just the difference is that the lifeline does not consist of tails.
4. The messages transmitted over sequencing is represented by numbering each individual message.
5. The collaboration diagram is semantically weak in comparison to the sequence diagram.
6. The special case of a collaboration diagram is the object diagram.
7. It focuses on the elements and not the message flow, like sequence diagrams.
8. Since the collaboration diagrams are not that expensive, the sequence diagram can be directly converted to the collaboration diagram.
9. There may be a chance of losing some amount of information while implementing a collaboration diagram with respect to the sequence diagram.

## The drawback of a Collaboration Diagram

1. Multiple objects residing in the system can make a complex collaboration diagram, as it becomes quite hard to explore the objects.
2. It is a time-consuming diagram.
3. After the program terminates, the object is destroyed.
4. As the object state changes momentarily, it becomes difficult to keep an eye on every single that has occurred inside the object of a system.

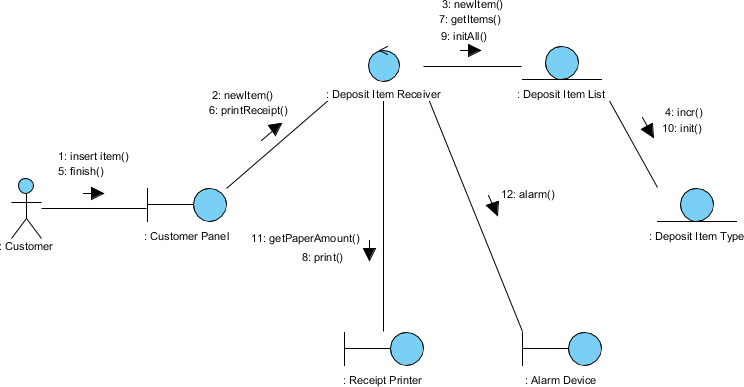
## Collaboration Diagram Example

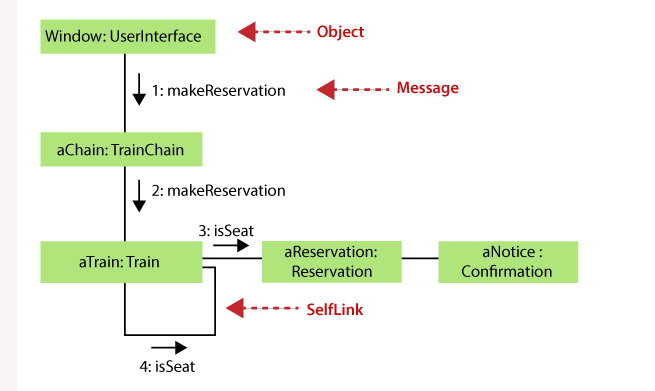
## Collaboration Diagram Example

## 

## Collaboration Diagram in Robustness Diagram Format

You can have objects and actor instances in collaboration diagrams, together with links and messages describing how they are related and how they interact. The **Receive Deposit Item** in the **Recycling-Machine System** diagram shown below describes what takes place in the participating objects, in terms of how the objects communicate by sending messages to one another. You can make a collaboration diagram for each variant of a use case’s flow of events.



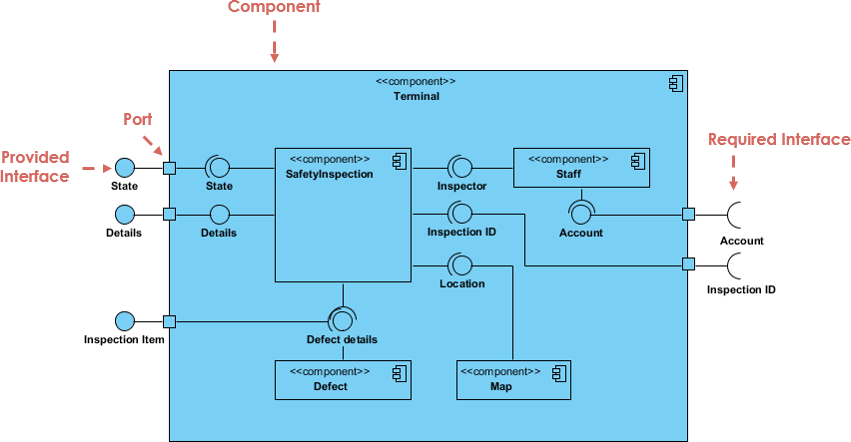


**UNIT :5**

* **Architectural diagram**
* **Component diagrams and its need**
* **Deployment diagrams , its purpose, Architecture system**

## Component Diagram at a Glance

A component diagram breaks down the actual system under development into various high levels of functionality. Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis.



The example above shows the internal components of a larger component:

* The data (account and inspection ID) flows into the component via the port on the right-hand side and is converted into a format the internal components can use. The interfaces on the right are known as required interfaces, which represents the services the component needed in order to carry out its duty.
* The data then passes to and through several other components via various connections before it is output at the ports on the left. Those interfaces on the left are known as provided interface, which represents the services to deliver by the exhibiting component.
* It is important to note that the internal components are surrounded by a large 'box' which can be the overall system itself (in which case there would not be a component symbol in the top right corner) or a subsystem or component of the overall system (in this case the 'box' is a component itself).

## Basic Concepts of Component Diagram

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. In UML 2, a component is drawn as a rectangle with optional compartments stacked vertically. A high-level, abstracted view of a component in UML 2 can be modeled as:

1. A rectangle with the component's name
2. A rectangle with the component icon
3. A rectangle with the stereotype text and/or icon

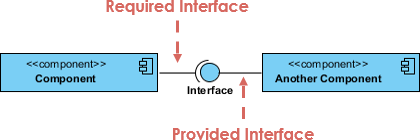
Looks of a Component

## Interface

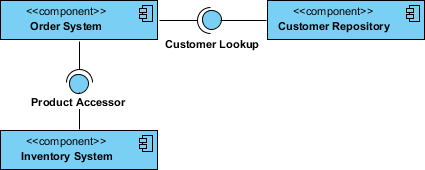
In the example below shows two type of component interfaces:

**Provided interface** symbols with a complete circle at their end represent an interface that the component provides - this "lollipop" symbol is shorthand for a realization relationship of an interface classifier.

**Required Interface** symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires (in both cases, the interface's name is placed near the interface symbol itself).

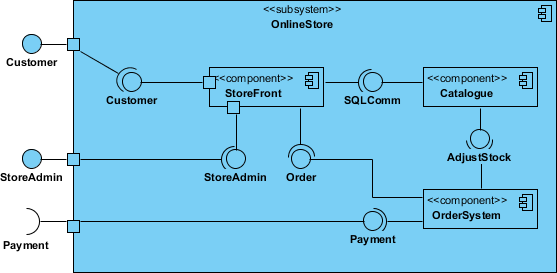


#### Component Diagram Example - Using Interface (Order System)



## Subsystems

The subsystem classifier is a specialized version of a component classifier. Because of this, the subsystem notation element inherits all the same rules as the component notation element. The only difference is that a subsystem notation element has the keyword of subsystem instead of component.



## Port

Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component.



## Relationships

Graphically, a component diagram is a collection of vertices and arcs and commonly contain components, interfaces and dependency, aggregation, constraint, generalization, association, and realization relationships. It may also contain notes and constraints.

|  |  |
| --- | --- |
| **Relationships** | **Notation** |
| **Association**:   * An association specifies a semantic relationship that can occur between typed instances. * It has at least two ends represented by properties, each of which is connected to the type of the end. More than one end of the association may have the same type. | Component Diagram Notation: Association |
| **Composition**:   * Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. * If a composite is deleted, all of its parts are normally deleted with it. | Component Diagram Notation: Composition |
| **Aggregation**   * A kind of association that has one of its end marked shared as kind of aggregation, meaning that it has a shared aggregation. | Component Diagram Notation: Aggregation |
| **Constraint**   * A condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element. | Component Diagram Notation: Constraint |
| **Dependency**   * A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. * This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s). | Component Diagram Notation: Dependency |
| **Links:**   * A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. * Each instance of the specific classifier is also an indirect instance of the general classifier. * Thus, the specific classifier inherits the features of the more general classifier. | Component Diagram Notation: Generalization |

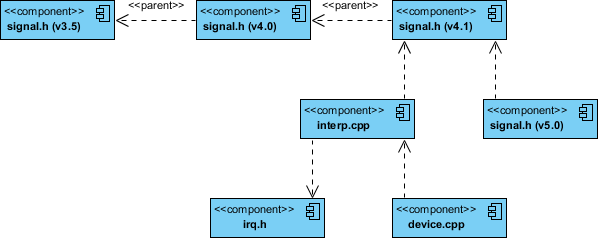
## Modeling Source Code

* Either by forward or reverse engineering, identify the set of source code files of interest and model them as components stereotyped as files.
* For larger systems, use packages to show groups of source code files.
* Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.
* Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.

Component Example - Java Source Code

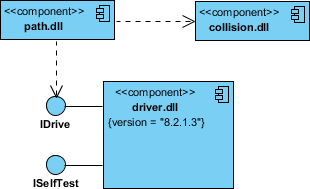


Component Diagram Example - C++ Code with versioning



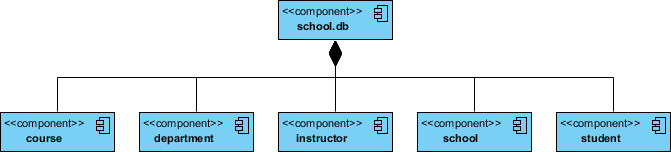
## Modeling an Executable Release

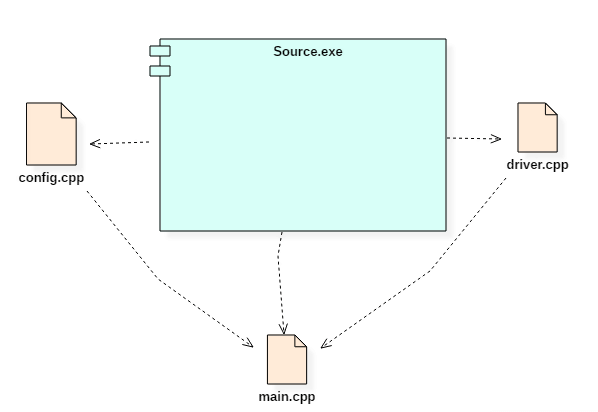
* Identify the set of components you'd like to model. Typically, this will involve some or all the components that live on one node, or the distribution of these sets of components across all the nodes in the system.
* Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). You can use the UML's extensibility mechanisms to provide visual cues(clues) for these stereotypes.
* For each component in this set, consider its relationship to its neighbors. Most often, this will involve interfaces that are exported (realized) by certain components and then imported (used) by others. If you want to expose the seams in your system, model these interfaces explicitly. If you want your model at a higher level of abstraction, elide these relationships by showing only dependencies among the components.



## Modeling a Physical Database

* Identify the classes in your model that represent your logical database schema.
* Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system.
* To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.
* Where possible, use tools to help you transform your logical design into a physical design.





### Summary

* A component is a replaceable and executable piece of a system.
* A component provides the set of required interfaces that a component realizes or implements.
* These are the static diagrams of the unified modeling language.
* It is a modular part of a system that encapsulates its contents
* Component diagrams are used to represent the working and behavior of various components of a system.
* Various components together make a single system.

**DEPLOYMENT DAIGRAM**

A [UML](https://en.wikipedia.org/wiki/Unified_Modeling_Language) deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them. Deployment diagrams is a kind of structure diagram used in modeling the physical aspects of an object-oriented system. They are often be used to model the static deployment view of a system (topology of the hardware).

## When to Use Deployment Diagram

* What existing systems will the newly added system need to interact or integrate with?
* How robust does system need to be (e.g., redundant hardware in case of a system failure)?
* What and who will connect to or interact with system, and how will they do it
* What middleware, including the operating system and communications approaches and protocols, will system use?
* What hardware and software will users directly interact with (PCs, network computers, browsers, etc.)?
* How will you monitor the system once deployed?
* How secure does the system needs to be (needs a firewall, physically secure hardware, etc.)?

## Purpose of Deployment Diagrams

* They show the structure of the run-time system
* They capture the hardware that will be used to implement the system and the links between different items of hardware.
* They model physical hardware elements and the communication paths between them
* They can be used to plan the architecture of a system.
* They are also useful for Document the deployment of software components or nodes

## Deployment Diagram at a Glance

Deployment diagrams are important for visualizing, specifying, and documenting embedded, client/server, and distributed systems and also for managing executable systems through forward and reverse engineering.

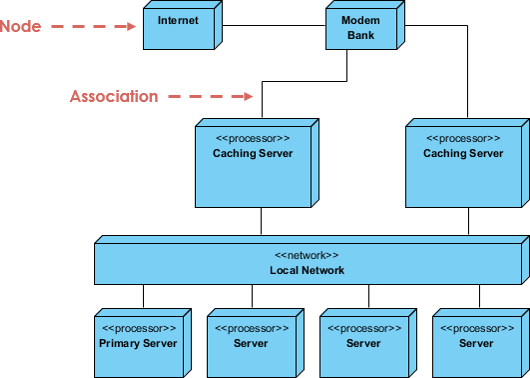
A deployment diagram is just a special kind of class diagram, which focuses on a system's nodes. Graphically, a deployment diagram is a collection of vertices and arcs. Deployment diagrams commonly contain:

### Nodes

* 3-D box represents a node, either software or hardware
* HW node can be signified with <<stereotype>>
* Connections between nodes are represented with a line, with optional <<stereotype>>
* Nodes can reside within a node

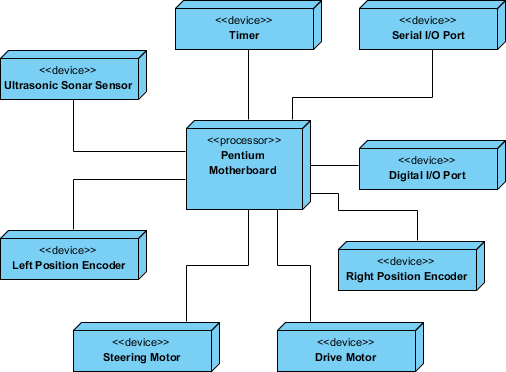
### Other Notations

* Dependency
* Association relationships.
* May also contain notes and constraints.



## Steps for Modeling an Embedded System

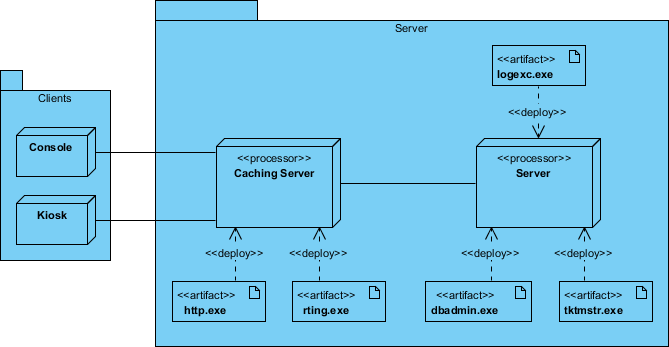
1. Identify the devices and nodes that are unique to your system.
2. Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific [stereotypes](https://en.wikipedia.org/wiki/Stereotype_(UML)) with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
3. Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.
4. As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.



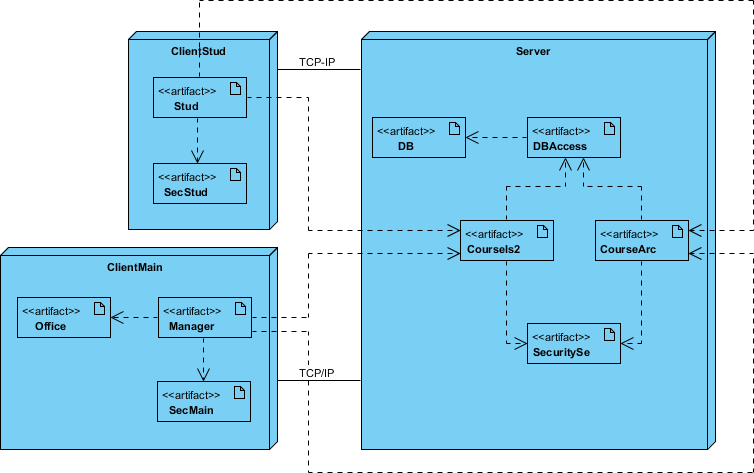
## Steps for Modeling a Client/Server System

1. Identify the nodes that represent your system's client and server processors.
2. Highlight those devices that are germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.
3. Provide visual cues for these processors and devices via stereotyping.
4. Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

The example shows the topology of a human resources system, which follows a classical client/server architecture.



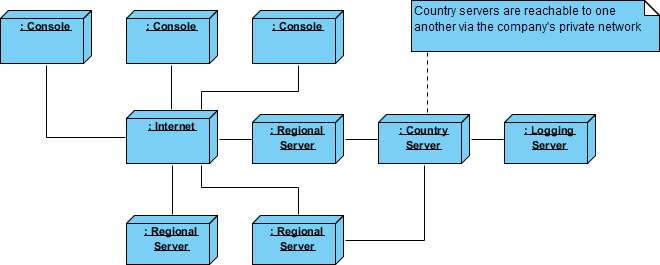
### TCP/IP Client / Server Example



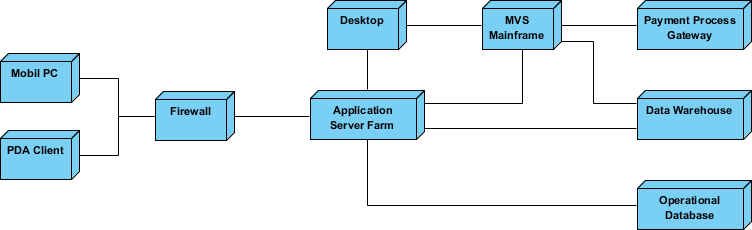
### Deployment Diagram Example - Modeling a Distributed System

1. Identify the system's devices and processors as for simpler client/server systems.
2. If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make these assessments.
3. Pay close attention to logical groupings of nodes, which you can specify by using packages.
4. Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.
5. If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.
6. When modeling a fully distributed system, it's common to reify the network itself as an node. i.e. Internet, LAN, WAN as nodes

The Example shows the topology of a fully distributed system.



### Deployment Diagram Example - Corporate Distributed System



## Deployment Planning Checklist

When you are drafting a deployment planning for your company, you may find that you do not know where to start or what you should focus on. The following checklist may give you some ideas with planning for deployment:

* **How will your system be installed?**
  1. Who will install it? How long should it take to install?
  2. Where the installation possibly fail?
  3. How do you back out if the installation fails? How long does it take to back out?
  4. What is your installation window (during what time period can you install your system)?
  5. What backups do you need before installation?
  6. Do you need to do a data conversion?
  7. How do you know that the installation was successful?
* **If different versions of the system will be in production at the same time, how will you resolve differences?**
* **What physical sites do you need to deploy to and in what order?**
  1. How will you train your support and operations staff?
  2. Do you need to deploy a production support system so that the support staff uses their own environment to simulate problems?
* **How will you train your users?**
  1. What documentation, and in what formats and languages, do your users, and support and operation staff need?
  2. How will updates to documentation be deployed?

**UNIT 6:**

* Object oriented programming styles:
* Object oriented style with refrence to reusability,extensibility,robustness.
* Case studies on : library management system, hospital management system,online shopping etc.

## What is Object Oriented Programming?

Object Oriented programming (OOP) is a programming paradigm that includes or relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes) which are used to create individual instances of objects. Because OOP is a programming paradigm, there are many object-oriented programming languages including: C++, Java, and Python.

A programmer designs a software program by organizing related pieces of information and behaviors together into a template called a class. Then individual objects are instantiated or created from the class template; these objects usually represent a real world thing. The entire software program runs by having multiple objects interact with objects to create the larger program.

## Why OOP?

OOP makes code organized, reusable, and easy to maintain; It follows the DRY method (Don’t Repeat Yourself). Benefits of OOP include security; OOP prevents unwanted access to data, or exposing proprietary code through encapsulation and abstraction - both are discussed further in the Principles of OOP section.

So how do programmers create Object Oriented programs? Well the short answer is by making classes, and creating objects from the classes. In OOP everything is an object. Classes form the blueprint for how data & behaviors are structured.

Objects are created for specific instances of a class. As a programmer, you might create a dog class (blueprint) as a standard way to organize all the important information about dogs, and then instantiate an individual dog as an object created from the dog class - like your dog Fluffy.

**Advantages of OOPs –**

* It models the real world very well.
* With OOP, programs are easy to understand and maintain.
* OOP offers code reusability. Already created classes can be reused without having to write them again.
* OOP facilitates the quick development of programs where parallel development of classes is possible.
* With OOP, programs are easier to test, manage and debug.

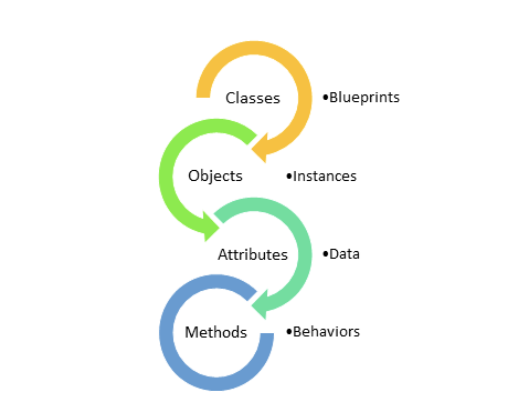
**Disadvantages of OOP –**

* With OOP, classes sometimes tend be over-generalised.
* The relations among classes become superficial at times.
* The OOP design is tricky and requires appropriate knowledge. Also, one needs to do proper planning and design for OOP programming.
* To program with OOP, the programmer needs proper skills such as that of design, programming and thinking in terms of objects and classes et

## Building blocks of OOP

For this article, we’ll discuss OOP in JavaScript. Assuming you’ve got a limited familiarity with JavaScript, the code building blocks to build an OOP program we’ll discuss are:

* classes
* objects
* methods
* attributes



**Major Elements** − By major, it is meant that if a model does not have any one of these elements, it ceases to be object oriented. The four major elements are −

* Abstraction
* Encapsulation
* Modularity
* Hierarchy

**Minor Elements** − By minor, it is meant that these elements are useful, but not indispensable part of the object model. The three minor elements are −

* Typing
* Concurrency
* Persistence

## Abstraction

Abstraction means to focus on the essential features of an element or object in OOP, ignoring its extraneous or accidental properties. The essential features are relative to the context in which the object is being used.

Grady Booch has defined abstraction as follows −

“An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.”

**Example** − When a class Student is designed, the attributes enrolment\_number, name, course, and address are included while characteristics like pulse\_rate and size\_of\_shoe are eliminated, since they are irrelevant in the perspective of the educational institution.

## Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. The class has methods that provide user interfaces by which the services provided by the class may be used.

## Modularity

Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem. Booch has defined modularity as −

“Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.”

Modularity is intrinsically linked with encapsulation. Modularity can be visualized as a way of mapping encapsulated abstractions into real, physical modules having high cohesion within the modules and their inter–module interaction or coupling is low.

## Hierarchy

In Grady Booch’s words, “Hierarchy is the ranking or ordering of abstraction”. Through hierarchy, a system can be made up of interrelated subsystems, which can have their own subsystems and so on until the smallest level components are reached. It uses the principle of “divide and conquer”. Hierarchy allows code reusability.

The two types of hierarchies in OOA are −

* **“IS–A” hierarchy** − It defines the hierarchical relationship in inheritance, whereby from a super-class, a number of subclasses may be derived which may again have subclasses and so on. For example, if we derive a class Rose from a class Flower, we can say that a rose “is–a” flower.
* **“PART–OF” hierarchy** − It defines the hierarchical relationship in aggregation by which a class may be composed of other classes. For example, a flower is composed of sepals, petals, stamens, and carpel. It can be said that a petal is a “part–of” flower.

## Typing

According to the theories of abstract data type, a type is a characterization of a set of elements. In OOP, a class is visualized as a type having properties distinct from any other types. Typing is the enforcement of the notion that an object is an instance of a single class or type. It also enforces that objects of different types may not be generally interchanged; and can be interchanged only in a very restricted manner if absolutely required to do so.

The two types of typing are −

* **Strong Typing** − Here, the operation on an object is checked at the time of compilation, as in the programming language Eiffel.
* **Weak Typing** − Here, messages may be sent to any class. The operation is checked only at the time of execution, as in the programming language Smalltalk.

## Concurrency

Concurrency in operating systems allows performing multiple tasks or processes simultaneously. When a single process exists in a system, it is said that there is a single thread of control. However, most systems have multiple threads, some active, some waiting for CPU, some suspended, and some terminated. Systems with multiple CPUs inherently permit concurrent threads of control; but systems running on a single CPU use appropriate algorithms to give equitable CPU time to the threads so as to enable concurrency.

In an object-oriented environment, there are active and inactive objects. The active objects have independent threads of control that can execute concurrently with threads of other objects. The active objects synchronize with one another as well as with purely sequential objects.

## Persistence

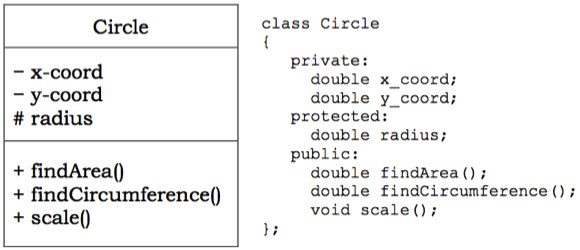
An object occupies a memory space and exists for a particular period of time. In traditional programming, the lifespan of an object was typically the lifespan of the execution of the program that created it. In files or databases, the object lifespan is longer than the duration of the process creating the object. This property by which an object continues to exist even after its creator ceases to exist is known as persistence.

Implementing an object-oriented design generally involves using a standard object oriented programming language (OOPL) or mapping object designs to databases. In most cases, it involves both.

## Implementation using Programming Languages

Usually, the task of transforming an object design into code is a straightforward process. Any object-oriented programming language like C++, Java, Smalltalk, C# and Python, includes provision for representing classes. In this chapter, we exemplify the concept using C++.

The following figure shows the representation of the class Circle using C++.



## Implementing Associations

Most programming languages do not provide constructs to implement associations directly. So the task of implementing associations needs considerable thought.

Associations may be either unidirectional or bidirectional. Besides, each association may be either one–to–one, one–to–many, or many–to–many.

### Unidirectional Associations

For implementing unidirectional associations, care should be taken so that unidirectionality is maintained. The implementations for different multiplicity are as follows −

* **Optional Associations** − Here, a link may or may not exist between the participating objects. For example, in the association between Customer and Current Account in the figure below, a customer may or may not have a current account.

Unidirectional Association

For implementation, an object of Current Account is included as an attribute in Customer that may be NULL. Implementation using C++ −

classCustomer{

private:

// attributes

Current\_Account c;//an object of Current\_Account as attribute

public:

Customer(){

c = NULL;

}// assign c as NULL

Current\_Account getCurrAc(){

return c;

}

void setCurrAc(Current\_Account myacc){

c = myacc;

}

void removeAcc(){

c = NULL;

}

};

* **One–to–one Associations** − Here, one instance of a class is related to exactly one instance of the associated class. For example, Department and Manager have one–to–one association as shown in the figure below.

One to One Unidirectional Association

This is implemented by including in Department, an object of Manager that should not be NULL. Implementation using C++ −

classDepartment{

private:

// attributes

Manager mgr;//an object of Manager as attribute

public:

Department(/\*parameters\*/,Manager m){//m is not NULL

// assign parameters to variables

mgr = m;

}

Manager getMgr(){

return mgr;

}

};

* **One–to–many Associations** − Here, one instance of a class is related to more than one instances of the associated class. For example, consider the association between Employee and Dependent in the following figure.

One to Many Unidirectional Association

This is implemented by including a list of Dependents in class Employee. Implementation using C++ STL list container −

classEmployee{

private:

char\* deptName;

list <Dependent> dep;//a list of Dependents as attribute

public:

void addDependent (Dependent d){

dep.push\_back(d);

}// adds an employee to the department

void removeDeoendent(Dependent d){

int index = find ( d, dep );

// find() function returns the index of d in list dep

dep.erase(index);

}

};

### Bi-directional Associations

To implement bi-directional association, links in both directions require to be maintained.

* **Optional or one–to–one Associations** − Consider the relationship between Project and Project Manager having one–to–one bidirectional association as shown in the figure below.

One to One Bidirectional Association

Implementation using C++ −

ClassProject{

private:

// attributes

Project\_Manager pmgr;

public:

void setManager (Project\_Manager pm);

Project\_Manager changeManager();

};

classProject\_Manager{

private:

// attributes

Project pj;

public:

void setProject(Project p);

Project removeProject();

};

* **One–to–many Associations** − Consider the relationship between Department and Employee having one–to–many association as shown in the figure below.

One to Many Bidirectional Association

### Implementation using C++ STL list container

classDepartment{

private:

char\* deptName;

list <Employee> emp;//a list of Employees as attribute

public:

void addEmployee (Employee e){

emp.push\_back(e);

}// adds an employee to the department

void removeEmployee(Employee e){

int index = find ( e, emp );

// find function returns the index of e in list emp

emp.erase(index);

}

};

classEmployee{

private:

//attributes

Department d;

public:

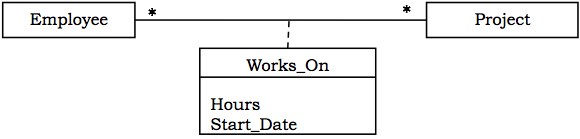
void addDept();

void removeDept();

};

### Implementing Associations as Classes

If an association has some attributes associated, it should be implemented using a separate class. For example, consider the one–to–one association between Employee and Project as shown in the figure below.



### Implementation of WorksOn using C++

classWorksOn{

private:

Employee e;

Project p;

Hours h;

char\* date;

public:

// class methods

};

## Implementing Constraints

Constraints in classes restrict the range and type of values that the attributes may take. In order to implement constraints, a valid default value is assigned to the attribute when an object is instantiated from the class. Whenever the value is changed at runtime, it is checked whether the value is valid or not. An invalid value may be handled by an exception handling routine or other methods.

**Example**

Consider an Employee class where age is an attribute that may have values in the range of 18 to 60. The following C++ code incorporates it −

classEmployee{

private:char\* name;

int age;

// other attributes

public:

Employee(){// default constructor

strcpy(name,"");

age =18;// default value

}

classAgeError{};// Exception class

void changeAge(int a){// method that changes age

if( a <18|| a >60)// check for invalid condition

throwAgeError();// throw exception

age = a;

}

};

## Implementing State Charts

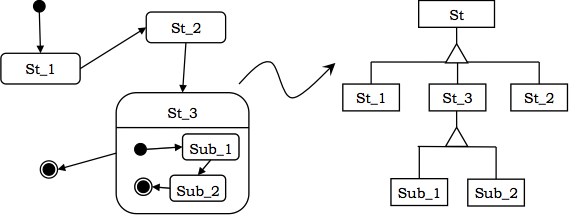
There are two alternative implementation strategies to implement states in state chart diagrams.

### Enumerations within Class

In this approach, the states are represented by different values of a data member (or set of data members). The values are explicitly defined by an enumeration within the class. The transitions are represented by member functions that change the value of the concerned data member.

### Arrangement of Classes in a Generalization Hierarchy

In this approach, the states are arranged in a generalization hierarchy in a manner that they can be referred by a common pointer variable. The following figure shows a transformation from state chart diagram to a generalization hierarchy.



## Object Mapping to Database System

### Persistency of Objects

An important aspect of developing object-oriented systems is persistency of data. Through persistency, objects have longer lifespan than the program that created it. Persistent data is saved on secondary storage medium from where it can be reloaded when required.

### Overview of RDBMS

A database is an ordered collection of related data.

A database management system (DBMS) is a collection of software that facilitates the processes of defining, creating, storing, manipulating, retrieving, sharing, and removing data in databases.

In relational database management systems (RDBMS), data is stored as relations or tables, where each column or field represents an attribute and each row or tuple represents a record of an instance.

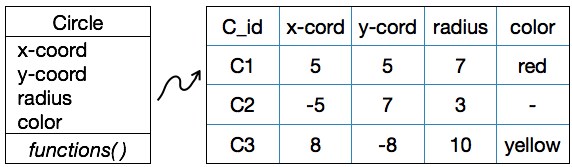
Each row is uniquely identified by a chosen set of minimal attributes called **primary key**.

A **foreign key** is an attribute that is the primary key of a related table.

### Representing Classes as Tables in RDBMS

To map a class to a database table, each attribute is represented as a field in the table. Either an existing attribute(s) is assigned as a primary key or a separate ID field is added as a primary key. The class may be partitioned horizontally or vertically as per requirement.

For example, the Circle class can be converted to table as shown in the figure below.



SchemaforCircleTable: CIRCLE(CID, X\_COORD, Y\_COORD, RADIUS, COLOR)

Creating a TableCircleusing SQL command:

CREATE TABLE CIRCLE (

CID VARCHAR2(4) PRIMARY KEY,

X\_COORD INTEGER NOT NULL,

Y\_COORD INTEGER NOT NULL,

Z\_COORD INTEGER NOT NULL,

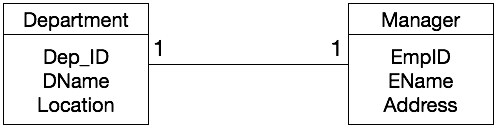
COLOR

);

## Mapping Associations to Database Tables

### One–to–One Associations

To implement 1:1 associations, the primary key of any one table is assigned as the foreign key of the other table. For example, consider the association between Department and Manager −



### SQL commands to create the tables

CREATE TABLE DEPARTMENT (

DEPT\_ID INTEGER PRIMARY KEY,

DNAME VARCHAR2(30) NOT NULL,

LOCATION VARCHAR2(20),

EMPID INTEGER REFERENCES MANAGER

);

CREATE TABLE MANAGER (

EMPID INTEGER PRIMARY KEY,

ENAME VARCHAR2(50) NOT NULL,

ADDRESS VARCHAR2(70),

);

### One–to–Many Associations

To implement 1:N associations, the primary key of the table in the 1-side of the association is assigned as the foreign key of the table at the N-side of the association. For example, consider the association between Department and Employee −



### SQL commands to create the tables

CREATE TABLE DEPARTMENT (

DEPT\_ID INTEGER PRIMARY KEY,

DNAME VARCHAR2(30) NOT NULL,

LOCATION VARCHAR2(20),

);

CREATE TABLE EMPLOYEE (

EMPID INTEGER PRIMARY KEY,

ENAME VARCHAR2(50) NOT NULL,

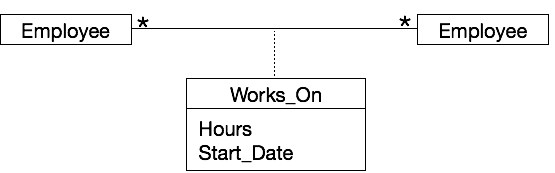
ADDRESS VARCHAR2(70),

D\_ID INTEGER REFERENCES DEPARTMENT

);

### Many–to–Many Associations

To implement M:N associations, a new relation is created that represents the association. For example, consider the following association between Employee and Project −



**Schema for Works\_On Table** − WORKS\_ON (EMPID, PID, HOURS, START\_DATE)

**SQL command to create Works\_On association** − CREATE TABLE WORKS\_ON

(

EMPID INTEGER,

PID INTEGER,

HOURS INTEGER,

START\_DATE DATE,

PRIMARY KEY (EMPID, PID),

FOREIGN KEY (EMPID) REFERENCES EMPLOYEE,

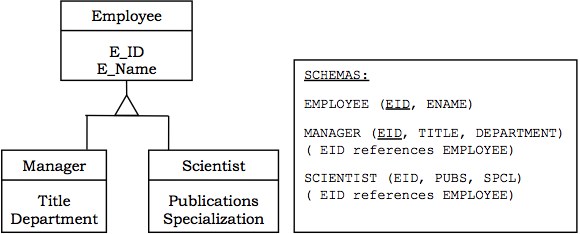
FOREIGN KEY (PID) REFERENCES PROJECT

);

## Mapping Inheritance to Tables

To map inheritance, the primary key of the base table(s) is assigned as the primary key as well as the foreign key in the derived table(s).

**Example**



**FORMAT OF INTERNAL QUESTION PAPER**

**1stInternal Examination (2020)**

Course: Semester:

Subject: Course Code:

Max. Marks: 40 Max. Time: 2 Hours

**Instructions (if any)**:- Use of calculator for subjects like Financial Mgt. Operation etc. allowed if required. (Scientific calculator is not allowed).

Use of unfair means will lead to cancellation of paper followed by disciplinary action.

**Question No. 1 is compulsory. Attempt any two questions from Q2 to Q5.**

**Attempt any two question from section 2.**

**Section 1**

(Theoretical Concept and Practical/Application oriented)

**Answer in 400 words. Each question carry 06 marks.**

Q. 1

Q. 2

Q.3

Q. 4

Q.5 Write Short Note on any two. Answer in 300 words. Each carry 03 marks.

**a)**

**b)**

**c)**

**Section 2**

(Analytical Question / Case Study / Essay Type Question to test analytical and Comprehensive Skills)

**Answer in 800 words. Attempt any 2 questions.Each question carry 11 marks**

Q6.

Q7.

Q8.

**PREVIOUS YEAR INTERNAL QUESTION PAPERS**

**Old syllabus Internal Question papers (for reference)**

**1st Internal Examination (2019)**

Course: BCA Semester: 5

Subject: OOAD Course Code: 506

Max. Marks: 40 Max. Time: 2 Hours

**Instructions (if any)**:- Use of calculator for subjects like Financial Mgt. Operation etc. allowed if required. (Scientific calculator is not allowed).

Use of unfair means will lead to cancellation of paper followed by disciplinary action.

**Question No.1 is Compulsory. Attempt any two questions from Q2 to Q5. Attempt any two question from section 2.**

**Section 1**

Answer in 400 words. Each Questions Carry 06 Marks

Q. 1. Draw a sequence diagram for Online Login to university Portal.

Q. 2 Explain Waterfall model.

Q. 3 Explain requirement engineering.

Q. 4 Draw Use case diagram (with extend, include, generalization) for ATM

Q. 5 write short notes on any two of the following

a)aggregation

b)generalization

c)specialization

**Section 2**

Answer in 800 words. Attempt any 2. Each Questions Carry 11 Marks

Q. 6 What is Inception? What are three kind of actor, explain with example.

Q. 7 Explain OOAD in detail.

Q. 8 write short notes on any two of the following, With Example

a) Inheritance

b) Encapsulation c) Polymorphism

**1st Internal Examination (2019)**

Course: BCA Semester: 5

Subject: OOAD Course Code: 506

Max. Marks: 40 Max. Time: 2 Hours

**Question No.1 is Compulsory. Attempt any two questions from Q2 to Q5.Attempt any two question from section 2.**

**Section 1**

Answer in 400 words. Each Questions Carry 06 Marks

Q. 1What is use case diagram? Explain with example.

Q. 2Explain RUP model.

Q. 3Explain UP artifacts.

Q. 4Draw a use collaboration diagram for online room booking (hotel management system).

Q. 5 write short notes on any two of the following

a)Inheritance

b)Encapsulation

c)Polymorphism

**Section 2**

Answer in 800 words. Attempt any 2. Each Questions Carry 11 Marks

Q. 6Explain SSD. Draw a sequence diagram for cash withdrawal from ATM.

Q. 7Draw Use case diagram (with extend, include, generalization) for university management system.

Q. 8 write short notes on any two of the following

a) class

b) object

c) Association

**1st Internal Examination (2019)**

Course: BCA Semester: 5

Subject: OOAD Course Code: 506

Max. Marks: 40 Max. Time: 2 Hours

**Question No.1 is Compulsory. Attempt any two questions from Q2 to Q5.Attempt any two question from section 2.**

**Section 1**

Answer in 400 words. Each Questions Carry 06 Marks

Q. 1What is use case diagram? Explain with example.

Q. 2Explain RUP model.

Q. 3Explain UP artifacts.

Q. 4Draw a use collaboration diagram for online room booking (hotel management system).

Q. 5 write short notes on any two of the following

a)Inheritance

b)Encapsulation

c)Polymorphism

**Section 2**

Answer in 800 words. Attempt any 2. Each Questions Carry 11 Marks

Q. 6Explain SSD. Draw a sequence diagram for cash withdrawal from ATM.

Q. 7Draw Use case diagram (with extend, include, generalization) for university management system.

Q. 8 write short notes on any two of the following

a) class

b) object

c) Association

**1st Internal Examination (2019)**

Course: BCA Semester: 5

Subject: OOAD Course Code: 506

Max. Marks: 40 Max. Time: 2 Hours

**Instructions (if any)**:- Use of calculator for subjects like Financial Mgt. Operation etc. allowed if required. (Scientific calculator is not allowed).

Use of unfair means will lead to cancellation of paper followed by disciplinary action.

**Question No.1 is Compulsory. Attempt any two questions from Q2 to Q5. Attempt any two question from section 2.**

**Section 1**

Answer in 400 words. Each Questions Carry 06 Marks

Q. 1. Draw a sequence diagram for Online Login to university Portal.

Q. 2 Explain RUP Model.

Q. 3 Explain CRC Card With example.

Q. 4 Draw Use case diagram (with extend, include, generalization) for ATM

Q. 5 write short notes on any two of the following

a)attributes and foreign key

b)generalization

c)specialization

**Section 2**

Answer in 800 words. Attempt any 2. Each Questions Carry 11 Marks

Q. 6 How are requirements organized in UP artifact.

Q. 7 Draw a sequence diagram for Hotel Management.

Q. 8 write short notes on any two of the following, With Example

a) Contract

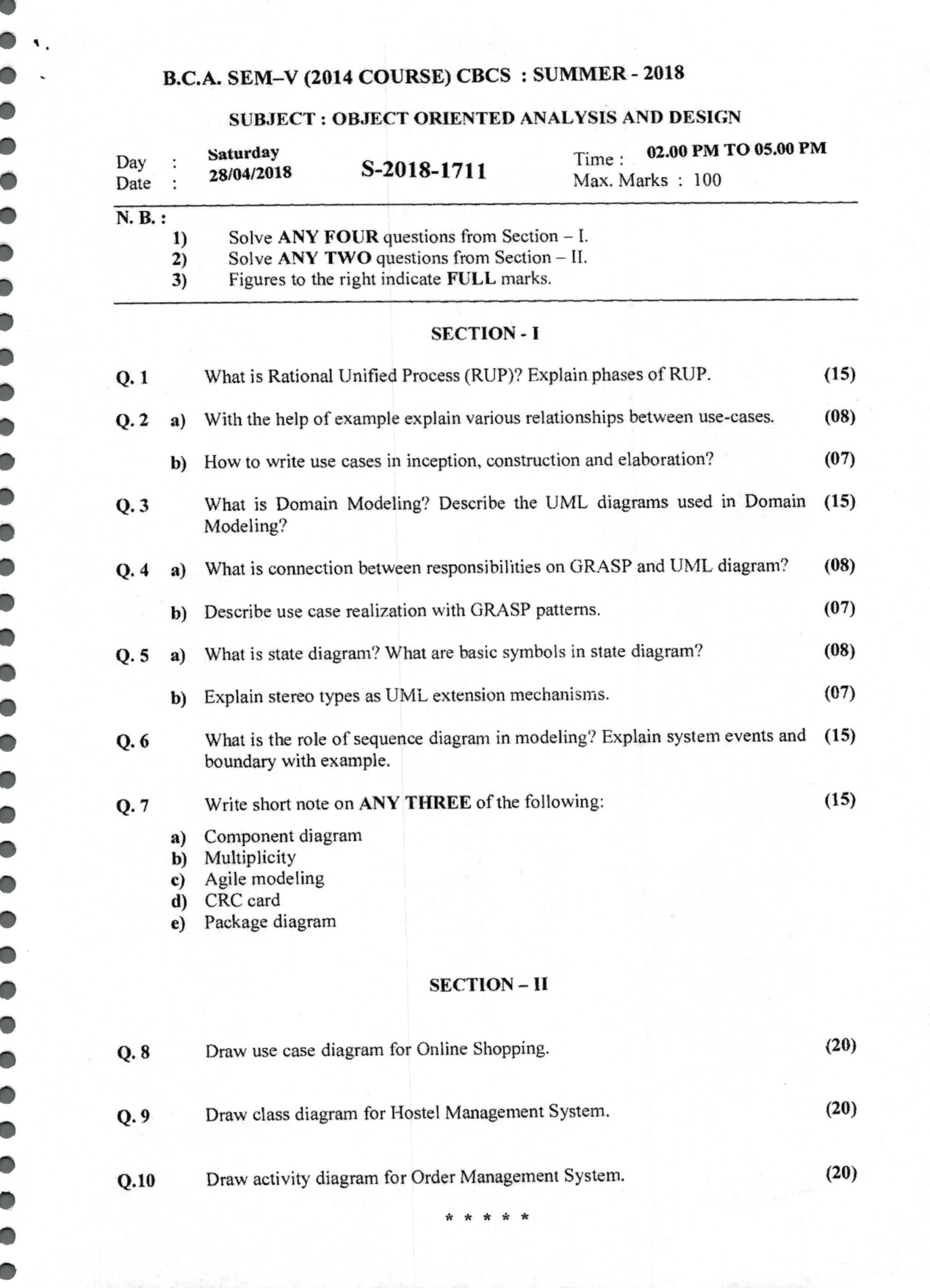
b) Multiplicity

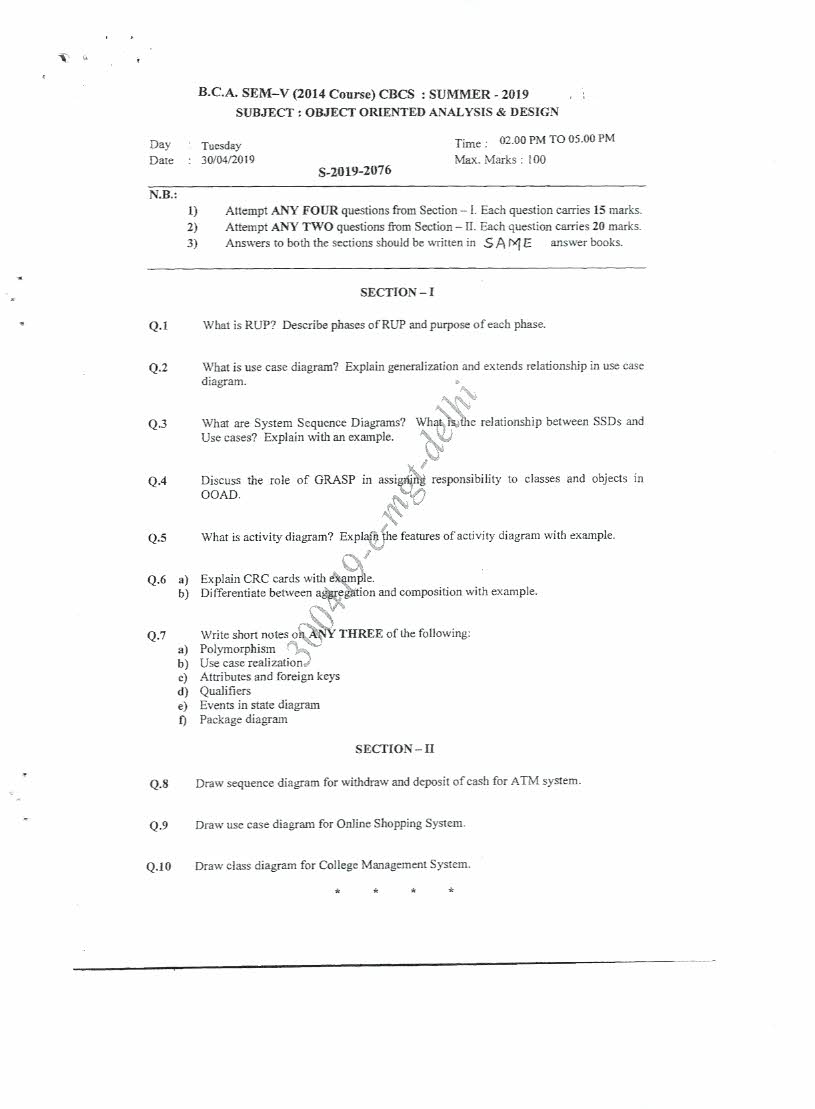
c) Interface

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Previous Year University Question**

* **Old syllabus University Question papers (for reference)**

****

****

**Practice Questions with solution**

**1. What is Object-Oriented Analysis?**

During object-oriented analysis there is an emphasis on finding and describing the objects or concepts in the problem domain. For example, in the case of the flight information system, some of the concepts include Plane, Flight, and Pilot.

**2. What is Object-Oriented Design?**

During object-oriented design (or simply, object design) there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. The combination of these two concepts shortly known as object oriented analysis and design.

**3. What is Object-Oriented Analysis and Design?**

During object-oriented analysis there is an emphasis on finding and describing the objects or concepts in the problem domain. For example, in the case of the flight information system, some of the concepts include Plane, Flight, and Pilot. During object-oriented design (or simply, object design) there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. The combination of these two concepts shortly known as object oriented analysis and design.

**4. What is Analysis and Design?**

Analysis emphasizes an investigation of the problem and requirements, rather than a solution. Design emphasizes a conceptual solution (in software and hardware) that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects.

**5. Define Design Class Diagrams**

A static view of the class definitions is usefully shown with a design class diagram. This illustrates the attributes and methods of the classes.

**6. What is the UML?**

The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems.

**7. What are the three ways and perspectives to Apply UML?**

Ways-UML as sketch, UML as blueprint, UML as programming language Perspectives-Conceptualperspective, Specification (software) perspective, Implementation (Software) perspective.

**8. What is Inception?**

Inception is the initial short step to establish a common vision and basic scope for the Project. It will include analysis of perhaps 10% of the use cases, analysis of the critical non- Functional requirement, creation of a business case, and preparation of the development Environment so that programming can start in the elaboration phase. Inception in one Sentence: Envision the product scope, vision, and business case.

**9. What Artifacts May Start in Inception?**

Some sample artifacts are Vision and Business Case, Use-Case Model, Supplementary Specification, Glossary, Risk List & Risk Management Plan, Prototypes and proof-of-concepts etc.

**10. Define Requirements and mention its types.**

Requirements are capabilities and conditions to which the system and more broadly, the project must conform.

1.       Functional

2.       Reliability

3.       Performance

4.       Supportability

**11. What are Actors?**

An actor is something with behavior, such as a person (identified by role), computer system, or organization; for example, a cashier.

**12. What is a scenario?**

A scenario is a specific sequence of actions and interactions between actors and the system; it is also called a use case instance. It is one particular story of using a system, or one path through the use case; for example, the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit payment denial.

**13. Define Use case.**

A use case is a collection of related success and failure scenarios that describe an actor using a system to support a goal. Use cases are text documents, not diagrams, and use-case modeling is primarily an act of writing text, not drawing diagrams.

**14. What are Three Kinds of Actors?**

Primary actor, Supporting actor, offstage actor.

**15. What Tests Can Help Find Useful Use Cases?**

1.       The Boss Test

2.       The EBP Test

3.       The Size Test

**16. What are Use Case Diagrams?**

A use case diagram is an excellent picture of the system context; it makes a good context diagram that is, showing the boundary of a system, what lies outside of it, and how it gets used.It serves as a communication tool that summarizes the behavior of a system and its actors.

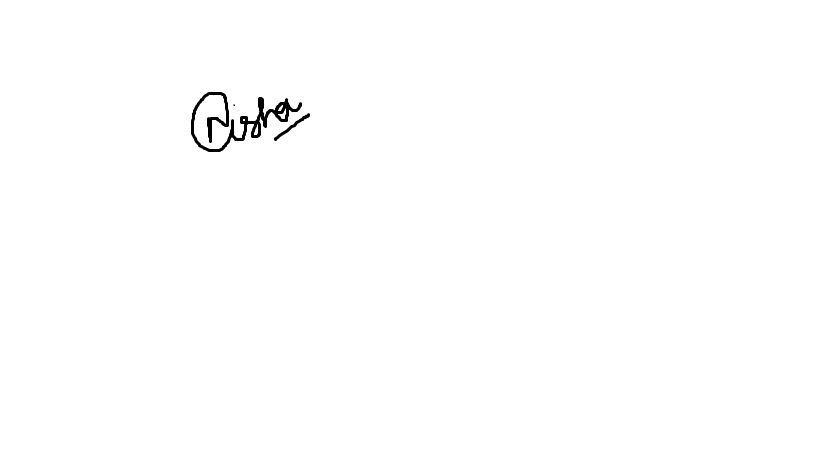
**17. What are Activity Diagrams?**

A diagram which is useful to visualize workflows and business processes. These can be a useful alternative or adjunct to writing the use case text, especially for business use cases that describe complex workflows involving many parties and concurrent actions.



**Declaration by Faculty**

I **Preeti Taneja** and **Nisha Malhotra, Designation Visiting Faculty** Teaching **Object oriented analysis and design** subject in **BCA,** course**Vth**sem have incorporated all the necessary pages section/quotations papers mentioned in this check list above.



Nisha Malhotra Preeti Taneja