An Insight to Object Analysis and Design

An Insight to Object Analysis and Design

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

Edition 2 - 2014

Dear Learner,

We congratulate you on your decision to pursue an Aptech course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc*. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- Strategies for delivery of instruction

  Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

  The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

- Assessment of learning

  The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

- Evaluation of instructional process and instructional materials

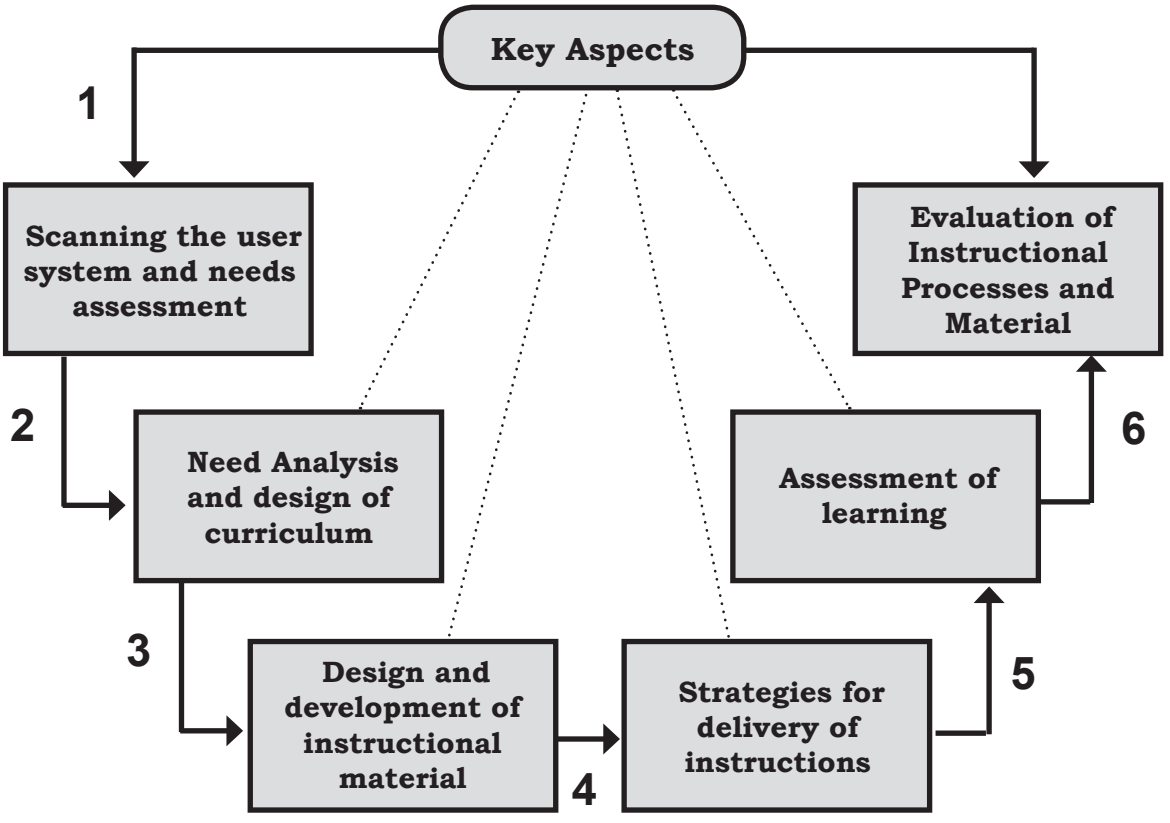
  The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Industry Recruitment Profile Survey - The Industry Recruitment Profile Survey was conducted across 1581 companies in August/September 2000, representing the Software, Manufacturing, Process Industry, Insurance, Finance & Service Sectors.

Aptech New Products Design Model



Onlinevarsity
your e-way to learning

WRITE-UPS BY

EXPERTS AND **LEARNERS**

TO PROMOTE NEW AVENUES AND
ENHANCE THE LEARNING EXPERIENCE

FOR FURTHER READING, LOGIN TO

www.onlinevarsity.com

Preface

Computer programmers strive to build systems that work and are useful whereas software engineers are faced with the task of creating complex systems with limited computing and human resources.

Object-oriented technology has evolved for managing the complexity present in the different kinds of systems. The concept of object model has proved to be very powerful.

This book covers an introduction to the standard notation used in system and software development, the Unified Modeling Language. This book also focuses on the different modeling techniques, use case diagrams, UML package diagrams, state machine diagrams and the design patterns.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

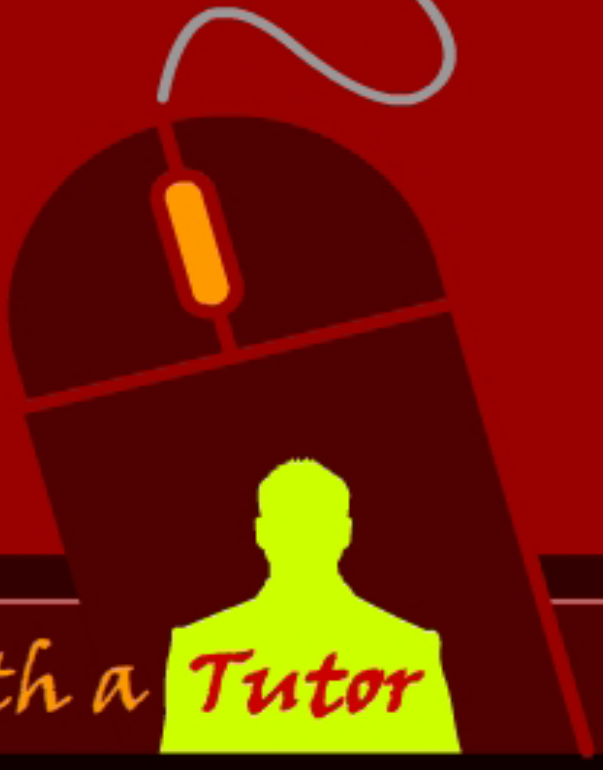
We will be glad to receive your suggestions.

Design Team

Tutor Chat

Onlinevarsity
your e-way to learning

are just a click away



To chat with a **Tutor**

login to **www.onlinevarsity.com**

Table of Contents

Modules

MODULE 1

OOAD with UML

Welcome to the module, **OOAD with UML**. This module introduces Object Oriented Analysis & Design and explains about the concepts of Object Orientation.

In this module, you will learn about:

- Object Oriented Analysis and Design
- The need for Modeling
- Unified Modeling Language
- Modeling Techniques

1.1 Understanding Object Oriented Analysis

In the first lesson, **Understanding Object Oriented Analysis**, you will learn to:

- Define and describe the concept of Analysis
- Define and describe the concept of Design
- Define and describe the concept of Object-Oriented Analysis and Design
- State the need for OOAD
- List the advantages and disadvantages of OOAD

1.1.1 Introduction

Any project irrespective of its field, civil, mechanical or software involves many activities starting from its inception to delivery. Software projects also involve various stages of development before being successfully released. Projects have to go through different stages of:

- Planning
- Design
- Implementation
- Rigorous testing before completion

The main reason for going through various stages, maintainable, reusable software that is implemented within the specified time without overshooting the budget.

The main phases of building a software system are as follows:
- Requirements gathering
- Analysis
- Design
- Development and Testing

These steps are done iteratively to produce incremental versions of software.

1.1.2 Concept of Analysis

Analysis is the decomposition of an application into its constituent parts. It can also be described as separation or breaking up of a whole into its fundamental elements or component parts.

This is accomplished by beginning with a set of requirements, and resulting in a set of already established software components and structures. Analysis focuses on translating the functional requirements into software concepts.

Analysis phase takes few pieces of requirement as an input and decomposes them into simple abstractions.

An Object oriented developer identifies a set of objects with few characteristics and behaviors.

For example a C programmer takes a portion of a problem analyzes it and comes up with a set of functions and structures.

1.1.3 Concept of Design

The goal of design is to refine a model with the intention of developing a design model that will allow a seamless transition to the coding phase.

In design, we adapt to:

- The implementation
- The deployment environment

The implementation is carried out by a team of developers mostly involved i writing the code and the deployment involves the hardware support team.

A Design phase consists of the set of decisions that determine how the abstractions will be implemented.

Typically, an Object Oriented developer designs relationships like:

- Inheritance
- Association between the interacting objects during a design phase

Design phase involves application of well-defined principles and patterns that are followed in implementation.

The real power of software design is that it can create more powerful metaphors for the real world, which change the nature of the problem, making it easier to solve.

1.1.4 Concept of OOAD

Object-Oriented Analysis, starts from the requirements and results in the abstractions that underlie those requirements.
It attempts to define the objects that represent those abstractions, and the messages that those objects pass between each other in order to implement the required behaviors.

Requirements are examined, one by one, and result in dynamic scenarios composed of objects and messages that address those requirements.

In the process, OOAD involves the use of various notations that represent the artifacts of the system.
Real world entities like Car or Person can be programmatically represented as a Car Object or Person Object respectively. Object Oriented Programming has revolutionized the way of building software projects with a number of features that make the software applications extensible, and easily maintainable.

Figure 1.1 depicts some real world entities.

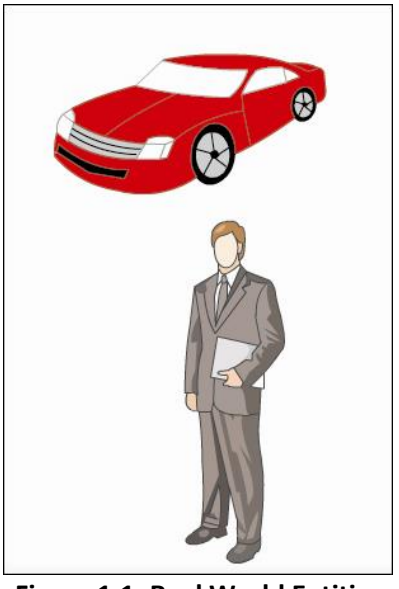


Figure 1.1: Real World Entities

Typically, an analysis and design phase involves the following activities:

- Finding the objects
- Organizing the objects
- Describing interaction between the objects
- Defining the internals of the objects
- Obtaining an understanding of the system based on functional requirements

OO Analysis exposes the components of the required behaviors, in order that they may be implemented.

OO Design results in decisions like how the objects will be implemented without exposing all the details of implementation. The differences between analysis and design are ones of focus and emphasis.

1.1.5 Need for OOAD

In the seventies and eighties Structured Analysis and Design was described by few pioneers of the industry. These practices became very popular, and by the early eighties had a profound effect upon the definition of analysis and Design.

Figure 1.2 depicts the phases of OOAD.

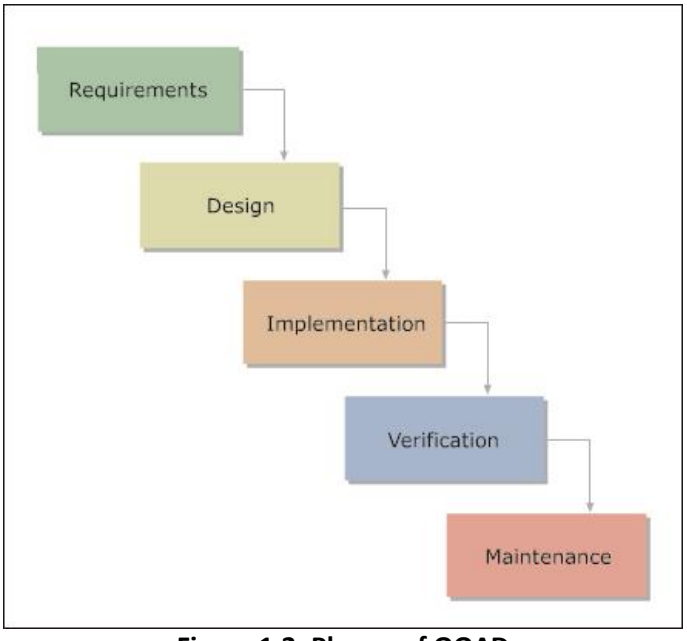


Figure 1.2: Phases of OOAD

Data flow diagrams were widely used during this period. However, functional decomposition of a system had inherent problems like difficulty in maintenance and incorporating changes.

Structured Analysis was not well suited for complex projects. Structured Analysis was a technique in which the requirements of the customer were broken down into a hierarchy of functions. This breakdown was known as functional decomposition.

1.1.6 Advantages and Disadvantages of OOAD

Object Oriented Analysis and Design phases are usually done concurrently. It is difficult to have a design model without an analysis model and vice versa. This serves as an advantage as well as a disadvantage.

Irrespective of two distinct phases there are no separate documents developed in each phase. There will not be an Analysis document and a separate Design document. At the end of both Analysis and Design phases there is normally only one set of documents created and maintained thereby making the job easier for the team.

The disadvantage lies in drawing a line between the two phases. There is no clear dividing line between analysis and design phase. They are very closely related.

Knowledge Check 1

1. Match the following statements about the concept of Analysis against their corresponding descriptions.

	DESCRIPTION	CONCEPT	
(A)	Requirements gathering, Analysis, Design, Development and testing are phases of	(1)	Analysis & Design
(B)	Important part of creating an application is	(2)	Software concepts
(C)	Breaking up of a whole into its fundamental elements or components is called	(3)	Simple abstractions
(D)	Focus of Analysis is to translate the functional requirements into	(4)	Analysis
(E)	Analysis takes few pieces of requirement as an input & decomposes them into	(5)	Building a software system

(A)	A-4, B-5, C-1, D-2, E-3	(C)	A-5, B-1, C-4, D-2, E-3
(B)	A-2, B-3, C-4, D-1, E-2	(D)	A-3, B-4, C-5, D-1, E-2

2. Match the following statements about the concept of Design against their corresponding descriptions.

	DESCRIPTION	Element	
(A)	To refine a model is the goal of	(1)	Object-Oriented designer
(B)	In designs, we adapt to	(2)	Design
(C)	Implementation environment is the environment of	(3)	Implementation & deployment
(D)	Designing relationships between objects is the job of	(4)	Developer

(A)	A-4, B-1, C-2, D-3	(C)	A-4, B-3, C-2, D-1
(B)	A-2, B-3, C-4, D-1	(D)	A-3, B-4, C-1, D-2

3. Which of these statements about OOAD are true?

(A)	Object oriented languages represent real world entities in terms of objects.
(B)	Object oriented analysis starts from requirements.
(C)	OOAD does not involve organizing the objects.
(D)	OOAD represents the artifacts of the system.
(E)	The differences between analysis & design are ones of focus and emphasis.

(A)	A, B, C	(C)	A, D, E
(B)	B, C, D	(D)	C, D, E

4. Which of these statements about the need for OOAD are true?

(A)	The practice of structural analysis and design did not have any effect upon definition of analysis and design.
(B)	Structured analysis was a technique in which the requirements of the customer were broken down into a hierarchy of functions.
(C)	Structured analysis fell short for big and complex programs.

(A)	A, B	(C)	A, C
(B)	B, C	(D)	A, B, C

5. Which of these statements about the advantages and disadvantages of OOAD are true?

(A)	OO Analysis models the problem Domain by developing an OO Domain.
(B)	Only one set of documents is maintained in OOAD.
(C)	There is clear a dividing line between analysis and design phase.
(D)	The Analysis and design phase are closely related.

(A)	A, B	(C)	A, C
(B)	**B, C**	(D)	B, D

1.2 Modeling

In this second lesson, **Modeling**, you will learn to:

- State the need of Modeling
- List and describe the Key Modeling Principles
- Describe the Object Oriented Modeling Techniques

1.2.1 Need for Modeling

Modeling is a central part of all the activities that lead up to the deployment of good software. It requires building a model, which communicates the desired structure and behavior of the system, visualizes and controls the system's architecture to understand the system better, and is often exposed to opportunities for simplification and reuse.

- If we want to build a small toy house, we can pretty much start with a pile of wood or plywood, some nails, and a few basic tools, such as a hammer, saw and a measuring tape
- If we want to build a house for our family, we can start with a pile of wood or plywood, some nails, and a few basic tools, but it will take a lot longer. Unless we have already done it a few dozen times before, it will require planning in detail before the first nail is pounded or the foundation laid
- If we want to build a high-rise office building, it would not be a good idea to start with a pile of wood or plywood, some nails, and a few basic tools. As we are probably using other people's money, they will demand to have input into the size, shape, and style of the building

Figure 1.3 depicts the scenarios.

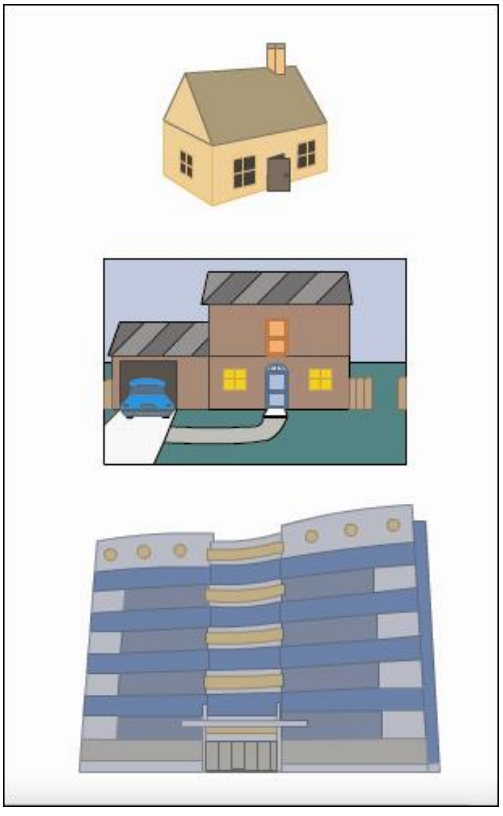


Figure 1.3: Need for Modelling

Curiously, many software development organizations start out wanting to build high rises but approach the problem as if they were knocking out a toy house.

Modeling is not just a part of the building industry. It would be inconceivable to deploy a new aircraft or an automobile without first building models—from computer models to physical wind tunnel models to full-scale prototypes.

New electrical devices, from microprocessors to telephone switching systems require some degree of modeling to understand the system and to communicate those ideas to others.
In the motion picture industry, storyboarding, which is a form of modeling, is central to any production. In the fields of sociology, economics, and business management, people build models so that they can validate their theories or try out new ones with minimal risk and cost.

A model is a simplification of reality. A model provides the blueprints of a system and may encompass detailed plans of the system under consideration.
A good model includes those elements that have broad abstraction. Every system may be described from different aspects using different models and each model is therefore, a semantically closed abstraction of the system.

Modeling helps to achieve four important aims:

- Helps to visualize a system as it is or according to need

- Permits specifying the structure or behavior of a system
- Gives a template that guides in constructing a system
- Documents the decisions that have been made

A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system. The fundamental reason for building models is to facilitate a better understanding of the system that is going to be developed.

Modeling is not just for big systems. Even the software equivalent of a toy house can benefit from some modeling. It is definitely true that the larger and more complex the system, the more important modeling becomes for one very simple reason, that is, *models of complex systems should be built because people cannot comprehend such a system in its entirety.*

There are limits to the human ability to understand complexity. Modeling helps by the principle: "Attack a hard problem by dividing it into a series of smaller problems that you can solve". Furthermore, through modeling, the human intellect is amplified. A model properly chosen can enable the modeler to work at higher levels of abstraction.

Divide and Conquer

Through modeling, the problem being studied is narrowed by focusing on only one aspect at a time. This is essentially the approach of "divide-and-conquer".

Figure 1.4 depicts the Divide and Conquer approach.

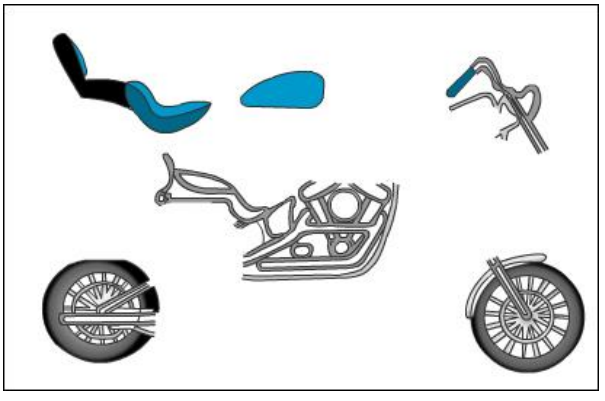


Figure 1.4: Divide and Conquer

1.2.2 Need for OOAD

Saying that one ought to model does not necessarily make it so. In fact, a number of studies suggest that most software organizations do little, if any, formal modeling.

Plotting the use of modeling against the complexity of a project reveals that the simpler the project, the less likely it is that formal modeling will be used. The operative word here is "formal."

In reality, in even the simplest project, developers perform some amount of modeling, albeit very informally. A developer may sketch out an idea on a blackboard or a scrap of paper in order to visualize a part of a system. There is nothing wrong with any of these models.
However, these informal models are often unplanned and do not provide a common language that can easily be shared with others.

1.2.3 Key Modeling Principles

The use of modeling has a rich history in all the engineering disciplines. The experience thus gained suggests four basic principles of modeling:

- The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped. If a system is built through the eyes of a database developer, the focus is likely to be on entity-relationship models that push behavior into triggers and stored procedures. If a system is built through the eyes of a structured analyst, the system, in all likelihood, will be one where the architecture is centered around a sea of classes and patterns of interaction that direct how those classes work together. Any of these approaches may be right for a given application and development culture, although experience suggests that object-oriented approaches have a large database or computational element. That fact notwithstanding, the point is that each view leads to a different kind of a system, with different costs and benefits.

- Every model may be expressed at different levels of precision. If a high rise is being built, sometimes a 30,000-foot view is needed—for instance, to help the investors visualize its look and feel. At other times, there is a need to get down to the level of the studs—for instance, when there is a tricky pipe run or an unusual structural element. The same is true with software models. Sometimes, a quick and simple executable model of the user interface is exactly what is needed; at other times, getting down and dirty with the bits is what is required, such as when cross-system interfaces are specified or when networking bottlenecks are tackled.

- The best models are connected to reality. In software, the Achilles heel of structured analysis techniques is the fact that there is a basic disconnect between its analysis model and the system's design model. Failing to bridge this chasm causes the system as conceived and the system as built to diverge over time. In object-oriented systems, it is possible to connect all the nearly independent views of a system into one semantic whole.

- No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models. Depending on the nature of the system, some models may be more important than others. For example, in data-intensive systems, models addressing static design views will dominate. In GUI-intensive systems, static and dynamic use-case views are quite important. In hard real time systems, dynamic process views tend to be more important. Finally, in distributed systems, such as those found in Web-intensive applications, implementation and deployment models are the most important.

1.2.4 Modeling Techniques

Traditionally, software engineers and systems analysts have used quite a number of modeling techniques. Some of the commonly used ones are as listed below:

- Data Modeling
- Information Modeling
- E-R Modeling
- State Transition Modeling
- Data Flow Models
- Process Flow Models

Many O-O analysis and design models have evolved from the mentioned models. Different O-O methodologies configure the models together in different ways. Using different kinds of models is generally

a good practice to represent different views of a problem or a solution. Different models provide different views of a problem while focusing on particular perspective. Broadly these models can be classified into two; Static and Dynamic models. Static models represent various objects (or "entities" or "things") in a problem, their characteristics, commonalities, and the structural relationships among them. Dynamic models represent the events and their sequence when the system is running, the collaborations, and the behavior of objects. Static and dynamic models are captured in many different kinds of work products done by different authors and vendors.

Figure 1.5 depicts the modeling techniques.

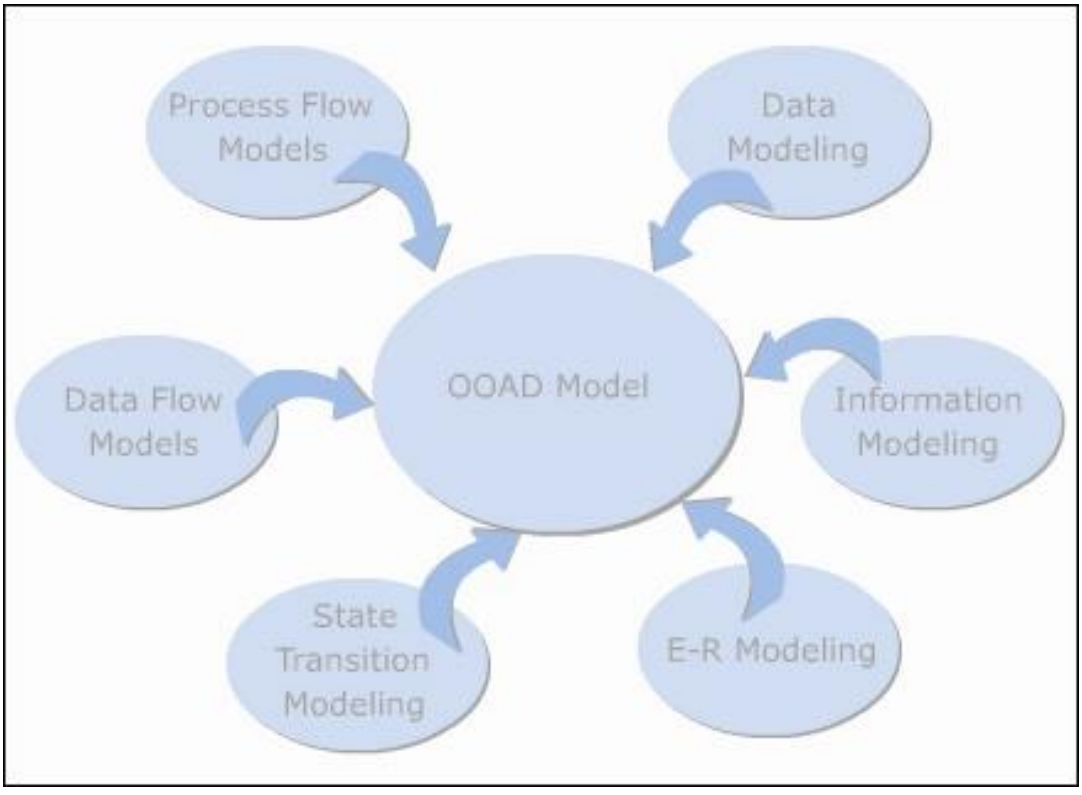


Figure 1.5: Modeling Techniques

- Static Models:
  - i. E-R (or E-R-A) diagrams
  - ii. Information models (Shlaer/Meltor)
  - iii. Objects models (OMT/Rumbaugh, Fusion, WSDDM-OT)
  - iv. Class diagrams (Booch, UML)
- Dynamic Models:
  - i. Use cases (Jacobson, WSDDM-OT, others)
  - ii. Scenarios (WSDDM-OT, others)
  - iii. Textual Scripts (OBA/Rubin & Goldberg)
  - iv. Event traces (Rumbaugh)
  - v. Object interaction diagrams (Booch, Jacobson, WSDDM-OT)
  - vi. Object Communication diagrams (Shlaer/Mellor)
  - vii. State transition diagrams (WSDDM-OT, lots -- different versions)
  - viii. Collaboration diagrams (UML)
  - xi. Sequence Diagrams (UML)

Knowledge Check 2

1. Match the following statements about the need for modeling against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	A model is a simplification of	(1)	Divide & Rule
(B)	Model facilitates better understanding of	(2)	Visualize a system
(C)	In modeling the approach used is	(3)	Reality
(D)	Model helps to	(4)	System to be developed

(A)	A-4, B-1, C-2, D-3	(C)	A-4, B-3, C-2, D-1
(B)	A-2, B-3, C-4, D-1	(D)	A-3, B-4, C-1, D-2

2. Match the statements about key modeling principles against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	The choice of what modules to create has a profound influence on	(1)	Precision
(B)	Good models bring to light even the most difficult	(2)	Reality
(C)	Every model may be expressed at different levels of	(3)	How problem is attacked & how solution is shaped
(D	Best models are connected to	(4)	Independent models
(E)	Every nontrivial system is best approached through a small set of	(5)	Development problems

(A)	A-4, B-5, C-1, D-2, E-3	(C)	A-5, B-1, C-2, D-3, E-4
(B)	A-3, B-5, C-1, D-2, E-4	(D)	A-2, B-3, C-4, D-5, E-1

3. Match the elements of about object oriented modeling techniques against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Every object has	(1)	Procedure or function
(B)	Structural models help people	(2)	Identity
(C)	Most common ways to approach a model are from	(3)	Object or class

(D)	Main building block of all software in an algorithmic perspective is	(4)	Algorithmic and object oriented perspective
(E)	Main building block in object-oriented perspective is	(5)	Object or class

(A)	A-4, B-5, C-1, D-2, E-3	(C)	A-5, B-1, C-2, D-3, E-4
(B)	A-3, B-5, C-1, D-2, E-4	(D)	A-2, B-5, C-4, D-1, E-3

1.3 Unified Modeling Language

In this third lesson, **Unified Modeling Language**, you will learn to:

- Define and describe the purpose of UML
- List the advantages of UML
- Describe the application of UML in Software Development Life Cycle

1.3.1 Purpose and Definition of UML

The software systems being developed today are much more complex than the human mind can comprehend in their entirety. This is why systems are modeled. The choice of what models to create has a profound influence upon how a problem is attacked and how a solution is shaped. No single model is sufficient; every complex system is best approached through a small set of nearly independent models. To increase comprehension, a common language like the Unified Modeling Language is used to express models. UML is a language that helps to visualize, specify, construct, and document models. The Unified Modeling Language (UML) provides a standard for the artifacts of development (semantic models, syntactic notation, and diagrams). However, UML is not a standard for the development process. Despite all of the value that a common modeling language brings, successful development of today's complex systems cannot be achieved solely by the use of the UML.

1.3.2 Advantages of UML

The UML provides a rich notation for visualizing the models. This includes the following key diagrams:

- Use-Case diagrams to illustrate user interactions with the system
- Class diagrams to illustrate logical structure
- Object diagrams to illustrate objects and links
- State diagrams to illustrate behavior
- Component diagrams to illustrate physical structure of the software
- Deployment diagrams to show the mapping of software to hardware configuration
- Interaction Overview diagram (mix of Sequence and Activity diagrams) to illustrate behavior
- Activity diagrams to illustrate the flow of events in a Use-Case
- Communication diagrams to illustrate interaction with objects
- Timing diagrams with emphasis on timing
- Composite Structure diagrams to illustrate runtime decomposition of class

1.3.3 Application of UML in SDLC

A software project life cycle includes processes that are Use case driven, Architecture centric and Iterative and Incremental. These processes are the way for analyzing, designing, and developing a software system.

Visual Modeling is used for generating graphic notations to represent various parts of a system. Visual modeling creates Object-Oriented-Models for a system under construction for its analysis, designing, & implementation in an object-oriented language. UML is the widely accepted standard language for modeling of the systems. UML gives a set of graphical notations prescribed for different entities & terms used in Object Oriented Modeling of any system.

Knowledge Check 3

1. Match the following statements about UML against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	UML provides a rich notation for	(1)	How a problem is attacked and how solution is shaped
(B)	UML provides a standard for	(2)	Visualizing the models
(C)	The choice of what models to create has an influence upon	(3)	Different views of the system
(D)	In building a visual model for a system, many different diagrams are needed to	(4)	The artefacts of the development

(A)	A-4, B-1, C-2, D-3	(C)	A-4, B-3, C-2, D-1
(B)	A-2, B-4, C-1, D-3	(D)	A-3, B-4, C-1, D-2

2. Match the following statements about the advantages of UML against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Use case diagrams illustrate	(1)	Behavior
(B)	Class diagrams illustrate	(2)	Runtime decomposition of class
(C)	State diagrams illustrate	(3)	User interactions with the system
(D)	Component diagrams illustrate	(4)	Logical structure
(E)	Composite structure diagram illustrate	(5)	Physical structure of the system

(A)	A-4, B-1, C-2, D-5, E-3	(C)	A-4, B-3, C-2, D-5, E-1
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-3, B-4, C-1, D-5, E-2

3. Which of these statements about the application of UML in SDLC are true?

(A)	UML is a generic modeling language
(B)	One can produce blueprints for any kind of software system with UML
(C)	RUP does not use UML as modeling language
(D)	RUP instructs the developer on how to implement the activities using the tools the developer is using

(A)	A, B, C	(C)	A, C, D
(B)	A, B, D	(D)	B, C, D

1.4 Modeling Techniques

In this fourth lesson, **Modeling Techniques**, you will learn to:

- Define and describe Static Models
- Define and describe Analysis Model
- Define and describe Dynamic Modeling
- Define and describe Design Model

1.4.1 Static Modeling

Using different kinds of models is generally a good practice to represent different views of a problem or a solution. Different models provide different views of a problem while focusing on particular perspective.

Static modeling is used to represent various objects (or “entities” or “things”) in a problem, their characteristics, commonalities, and the structural relationships among them.

- E-R (or E-R-A) diagrams
- Information models (Shlaer/Meltor)
- Objects models (OMT/Rumbaugh, Fusion, WSDDM-OT)
- Class diagrams (Booch, UML)

Some of the commonly used modeling techniques are as follows:

- Data Modeling
- Information Modeling
- E-R Modeling
- State Transition Modeling
- Data Flow Models
- Process Flow Models

1.4.2 Analysis Models

Analysis Model normally consists of class diagrams and sequence diagrams that describe the logical implementation of the functional requirements that you identified in the use case model.
The analysis model identifies the main classes in the system and contains a set of use case realizations that describe how the system will be built.

Class diagrams describe the static structure of the system by using stereotypes to model the functional parts of the system. Sequence diagrams realize the use cases by describing the flow of events in the use cases when they are executed. These use case realizations model how the parts of the system interact within the context of a specific use case.

Figure 1.6 depicts the diagrams of Analysis Model.

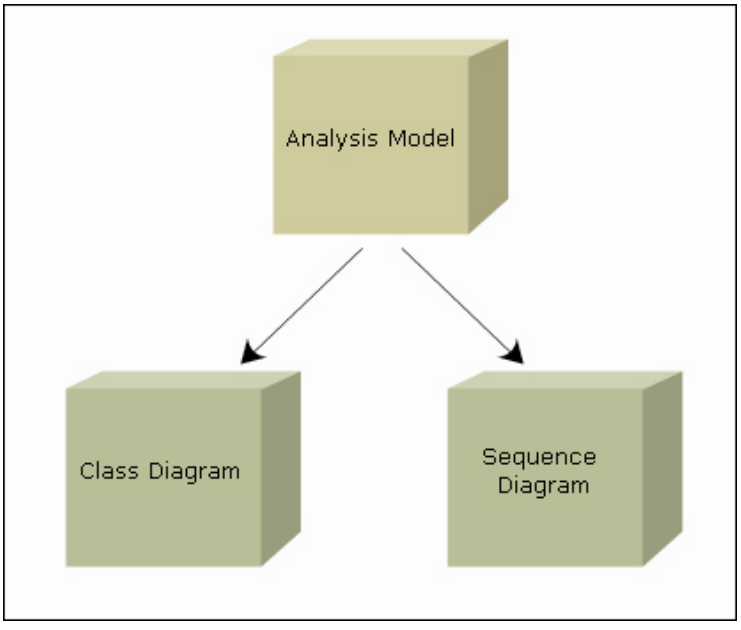


Figure 1.6: Analysis Model

- The analysis model describes the structure of the system or application that we are modeling.

1.4.3 Dynamic Models

Dynamic models represent the events and their sequence when the system is running, the collaborations, and the behavior of objects.

- Use cases (Jacobson, WSDDM-OT, others)
- Scenarios (WSDDM-OT, others)
- Textual Scripts (OBA/Rubin & Goldberg)
- Event traces (Rumbaugh)
- Object interaction diagrams (Booch, Jacobson, WSDDM-OT)
- Object Communication diagrams (Shlaer/Mellor)
- State transition diagrams (WSDDM-OT, lots -- different versions)
- Collaboration diagrams (UML)
- Sequence Diagrams (UML)

WSDDM-OT (World Wide Software Design and Delivery – Object Technology) is an IBM Global Services methodology and process model.

1.4.4 Design Models

Design models show the objects, classes, and relationships between these entities. Few examples of design models are Sub-system models that show logical groupings of objects into coherent subsystems, Sequence models that show the sequence of object interactions and State machine models that show how individual objects change their state in response to events. Design model potentially simplifies system evolution.

Knowledge Check 4

1. Which of these statements about the Analysis Model are true?

(A)	Event traces are part of an analysis model
(B)	Analysis model describes the structure of the system
(C)	Analysis model includes object interaction diagrams

(A)	A	(C)	B
(B)	A, B	(D)	B, C

2. Which of these statements about the Dynamic Model are true?

(A)	WISDDM-OT is an IBM Global Services Methodology
(B)	Textual scripts are part of design models
(C)	Design models include sequence diagrams

(A)	A, B	(C)	B
(B)	A, C	(D)	B, C

3. Which of these statements about the Design Model are true?

(A)	Design models show the objects & Classes and relationships between these entities
(B)	Design models amplify system evolution
(C)	Sub-system models logically group objects into coherent sub systems

(A)	A, B	(C)	B
(B)	B, C	(D)	A, C

Module Summary

In this module, **OOAD with UML**, you learnt about:

- **Object Oriented Analysis and Design**
  OOAD involves the use of various notations that represent the artifacts of the system. Real world entities like Car or Person can be programmatically represented as a Car Object or Person Object respectively. Object Oriented Programming has revolutionized the way of building software projects with a number of features that make the software applications extensible, and easily maintainable.

- **The need for Modeling**
  Modeling is a central part of all the activities that lead up to the deployment of good software. It requires building a model, which communicates the desired structure and behavior of the system, visualizes and controls the system's architecture to understand the system better, and is often exposed to opportunities for simplification and reuse.

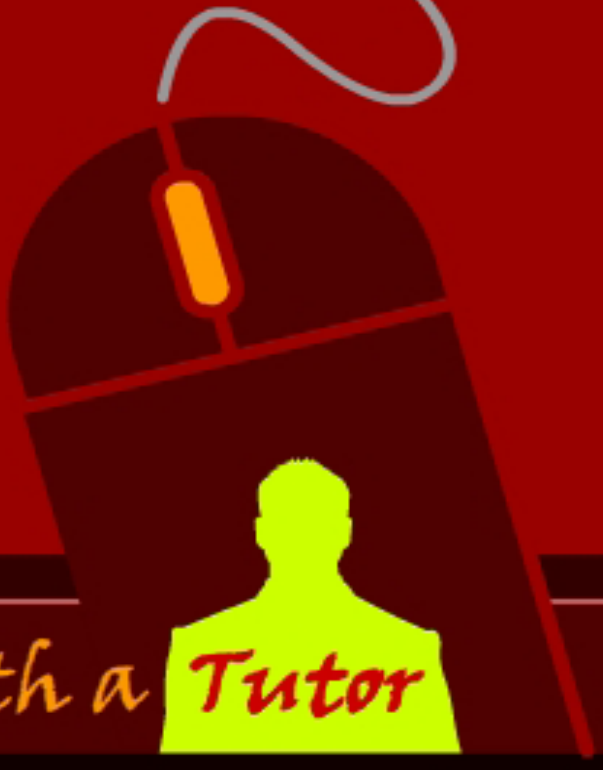
- **Unified Modeling Language**
  The Unified Modeling Language (UML) is a language that helps to visualize, specify, construct, and document models. UML provides a standard for the artifacts of development like semantic models, syntactic notation, and diagrams.

- **Modeling Techniques**
  Different models provide different views of a problem while focusing on particular perspective.
  Some of the commonly used modeling techniques are as follows:
  a. Data Modeling
  b. Information Modeling
  c. E-R Modeling
  d. State Transition Modeling
  e. Data Flow Models
  f. Process Flow Models

Tutor Chat

Onlinevarsity
your e-way to learning

are just a click away



To chat with a **Tutor**

login to **www.onlinevarsity.com**

MODULE 2

UML Overview

Welcome to the module, **UML Overview**. This module introduces the concept of a Modelling Language and then gives an insight into Unified Modelling Language. It goes on to explain the various UML Diagrams and briefly explains UML Elements.

In this module, you will learn about:

- Basic purpose of the following UML Diagrams
  - Use-Case diagram
  - Object diagram
  - Class diagram
  - Sequence diagram
  - Collaboration diagram
  - State Chart diagram
  - Activity diagram
  - Component diagram
  - Deployment diagram
- Context of usage of UML Elements

2.1 UML Diagrams

In this first lesson, **UML Diagrams,** you will learn to:

- Define and describe Use Case Diagrams
- Define and describe Object Diagrams
- Define and describe Class Diagrams
- Define and describe Sequence Diagrams

2.1.1 Unified Modelling Language

A system design is expressed by 'Methods' using a graphical notation.
This graphic notation is called a 'Modelling Language'.

The Unified Modelling Language provides a standard for the artifacts of system development like:

- Semantic modelling
- Syntactic notation
- Diagrams

UML took shape from OOAD methods of the late 80s and early 90s. It is a combination of the methods of Booch, Rumbaugh, and Jacobson.

UML helps to visualize, specify, construct, and document models. It supports a variety of diagrams.

2.1.2 Use Case Diagram

In any system, the user will have a specific goal. In order to reach this goal he would have to go through a set of scenarios. This set is a Use Case.

Figure 2.1 depicts a Use Case diagram.

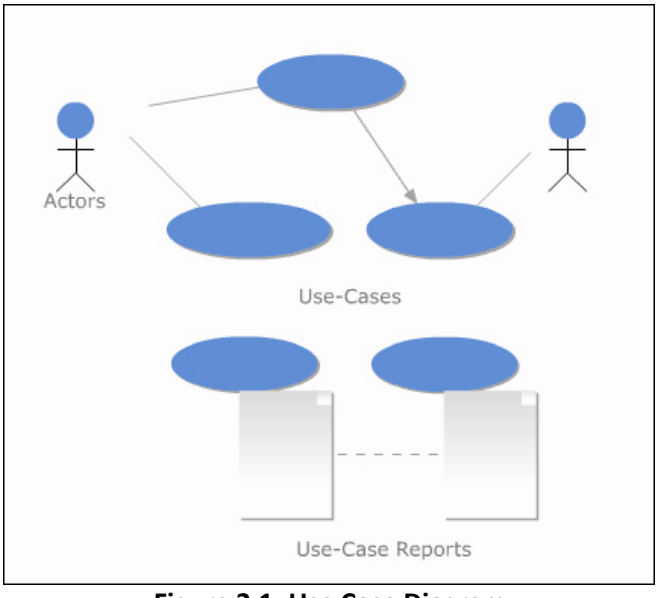


Figure 2.1: Use Case Diagram

The diagram that visualizes the Use Case is a Use Case diagram.

The **Use-Case Model** is a model of the system's intended functionalities and its environment. It bridges connectivity between the customers and the developers as a contract.

The main elements of Use-Case Diagrams are as follows:

- **Actor**: It is the role that a user plays in the system. An actor interacts with or performs a Use Case. There may be many actors in a Use case. Example: A Sales Manager in an Accounting System

- **Use-Cases**: This is the relationship between the different Use Case packages. Example: In an Accounting system, Calculating Sale value may be used by many Use cases.

- **Generalization**: This is done when a Use case is needed to add a little more activity or a variation in an existing Use Case. Example: In any business, Customer Treatment will be similar in Individual and Group customers, but they also have some variations and special clauses.

2.1.3 Object Diagram

Object diagrams are useful for depicting the real world examples of objects and the relationships between them in a system.

Figure 2.2 depicts an Object diagram.

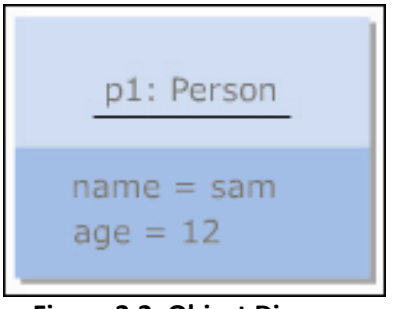


Figure 2.2: Object Diagram

UML object diagrams show objects and the connections between them.

2.1.4 Class Diagram

Classes are templates for objects. Class diagrams are widely used in Analysis and Design. They form the base of almost all OO methods.

They describe the classes in a system and their attributes and behavior. They show the inter relationships like inheritance, aggregation and composition.

Figure 2.3 depicts a Class diagram.

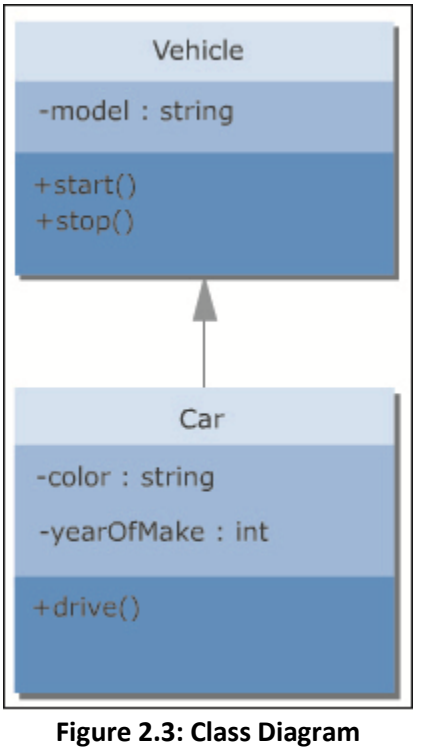


Figure 2.3: Class Diagram

2.1.5 Sequence Diagram

A sequence diagram visualizes:

- The pattern of interaction
- The messages the objects send

Sequence diagrams have the following notations:

- A vertical dashed line called the **Lifeline** shows the object's **existence** during the interaction
- An **object** is drawn at the **head** of the lifeline, and shows the name of the object and its class separated by a colon and underlined
- A **message** is shown as **a horizontal solid arrow** from the lifeline of one object to the lifeline of another object

Figure 2.4 depicts a Sequence diagram.

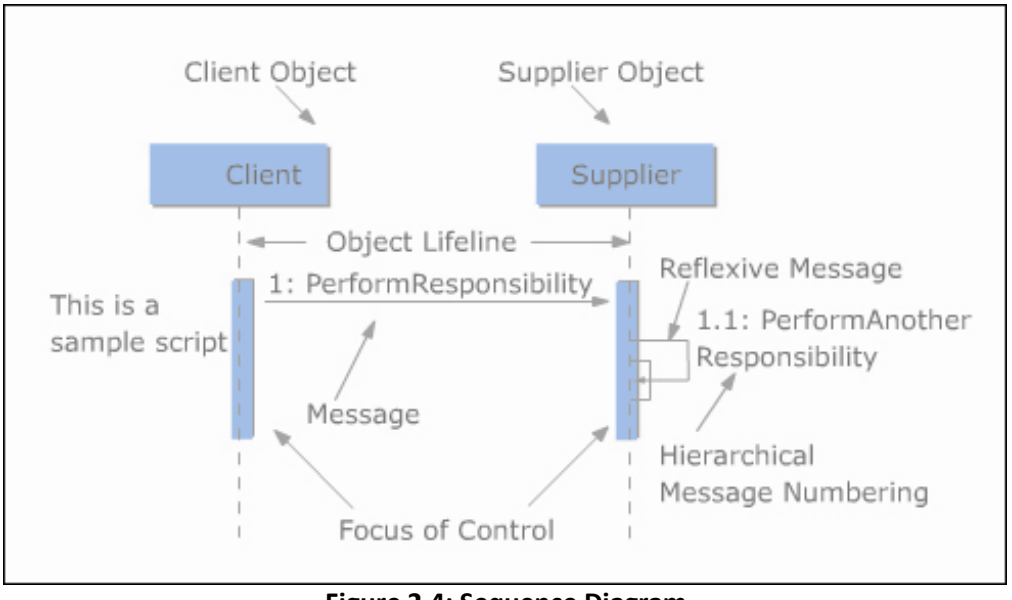


Figure 2.4: Sequence Diagram

Knowledge Check 1

1. Which of the following statements are true?

(A)	Class diagrams are redundant when Object diagrams are already there for a system
(B)	Use case diagrams are used for depicting the overall functionality of the system
(C)	An object's existence is shown by a solid horizontal line between 2 objects
(D)	Class diagrams illustrate the attributes and behavior of a system

(A)	A, B	(C)	A, B, D
(B)	B, C	(D)	B, D

2.2 More on UML Diagrams

In this second lesson, **More on UML Diagrams,** you will learn to:

- Define and describe Collaboration Diagrams
- Define and describe State Chart Diagrams
- Define and describe Activity Diagrams
- Define and describe Component Diagrams
- Define and describe Deployment Diagrams

2.2.1 Collaboration Diagram

A collaboration diagram is similar to a sequence diagram. However, the emphasis is on the structural links between the objects. In sequence diagrams, the emphasis is on the sequence or order of the messages.

A link is shown as a solid line between two objects. A link can be an instance of an association, or it can be anonymous, meaning that its association is unspecified.

An object can be represented in three ways:

Objectname: Classname - An instance (a specific object) of an association (class)
ObjectName - A named object
:ClassName - Anonymous or unnamed object

Figure 2.5 depicts a Collaboration diagram.

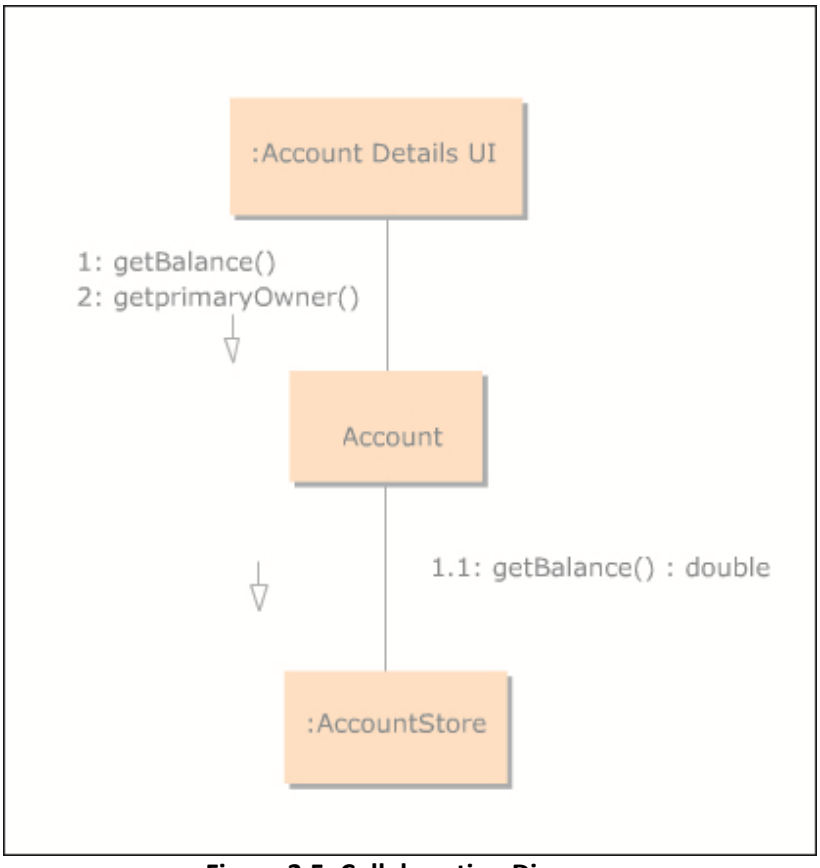


Figure 2.5: Collaboration Diagram

- **links**
  A link is a relationship between two objects that can communicate with each other using messages.

  Syntax:

```
Objectname:Classname, An instance(a specific object) of an
association(class)
ObjectName - A named object
ClassName – Anonymous or unnamed object
```

2.2.2 State Chart Diagram

A state chart diagram describes the:

- States an object can assume
- Events that trigger an object to make a transition from one state to another
- Significant activities and actions that occur as a result of the transition

- **States**
  An object's state is one of the possible conditions in which the object exists. The state can change over time.

Figure 2.6 depicts a State Chart Diagram.

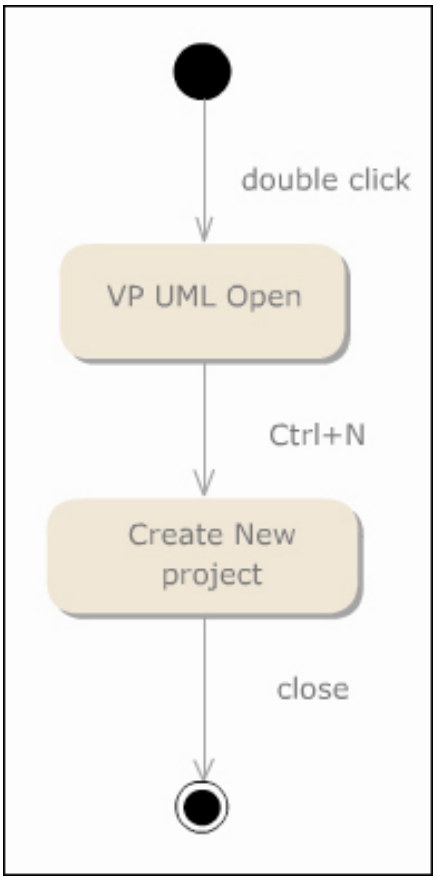


Figure 2.6: State Chart Diagram

2.2.3 Activity Diagram

An Activity Diagram describes the

- Flow of events of a Use case
- The action states

It is useful especially when describing behavior that has conditional and parallel processes. An activity diagram is a variant or a special case of a state diagram.

The main elements of the Activity diagram are as follows:

Activity: Represents a defined behavior or work of an object.

Guard Conditions: A set of conditions deciding the execution of an activity. These guard conditions control which transition follows once an activity is completed.

Fork: A fork is where a single flow of control is split into two or more concurrent flows of control.

Join: A join is where two or more concurrent flows of control are synchronized.

Merge: A merge is where there are multiple input flows and a single output. It is the end of a conditional behavior.

Figure 2.7 depicts an Activity diagram.

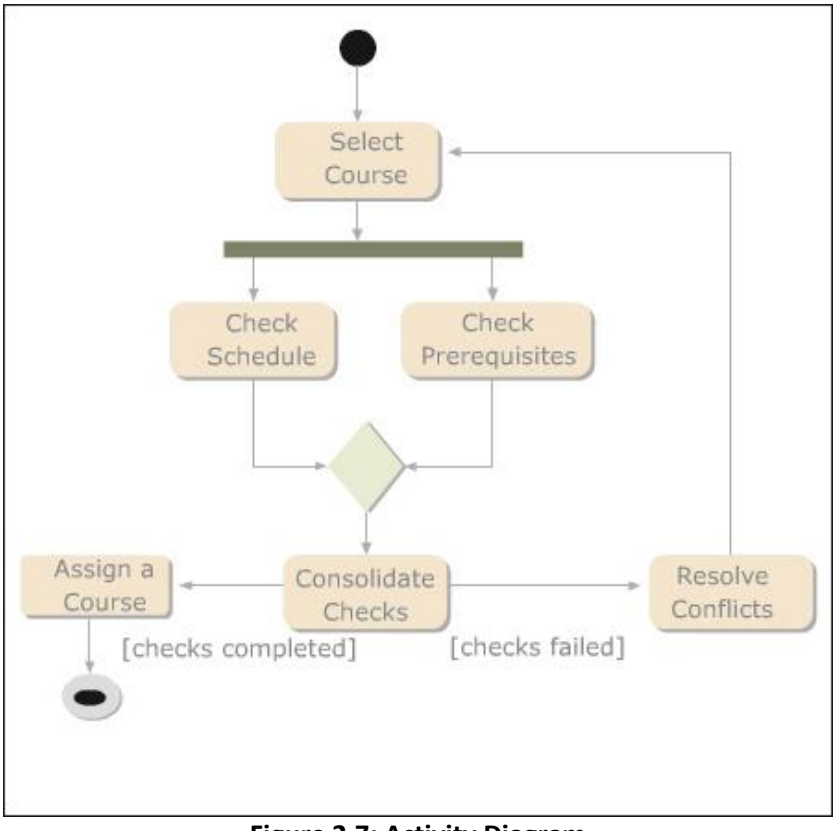


Figure 2.7: Activity Diagram

The activity diagram in figure 2.7 shows the steps of the 'Register for courses' Use-Case.

2.2.4 Component Diagram

Component diagrams have a graphical display of pieces of **software** or components in a system and how they depend on each other in the **implementation environment**.

The Software components are as follows:

- Source code components. Example: .h files, .cpp files, shell scripts, data files
- Binary code components. Example: Java Beans, ActiveX controls, COM objects (DLL's and OCX's from VB), CORBA objects
- Executable components Example: .exe files
- Stereotypes (with alternatives icons) may be used to define these specific kinds of components
- **Components**
  A component is defined as a physical and replaceable part of a system that has a real set of interfaces

Figure 2.8 depicts a Component diagram.

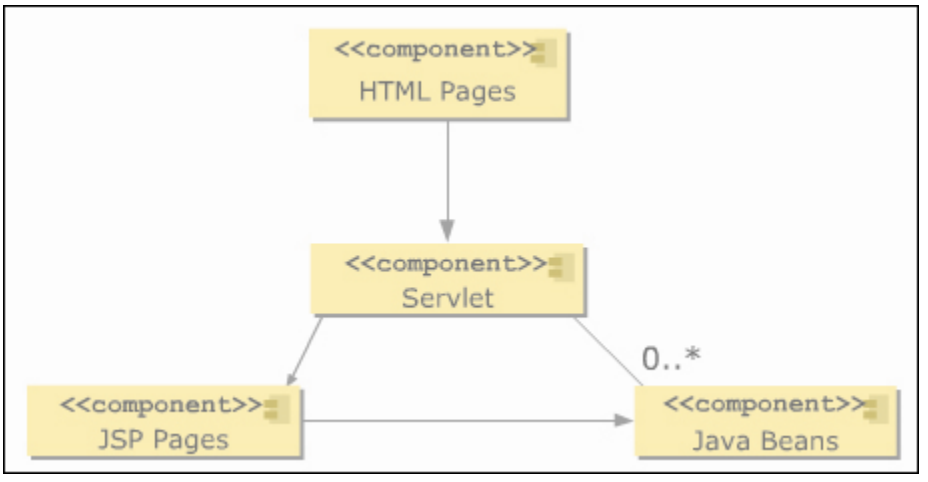


Figure 2.8: Component Diagram

2.2.5 Deployment Diagram

Deployment diagrams are a graphical display mostly of the **hardware** of the **implementation environment**.

A Deployment diagram is a collection of **nodes.**

Each node represents a type of hardware such as Web Server, Database, client PC.

A node may also be a human being or functionality or an organizational unit. Nodes can contain other nodes or software artifacts. It can contain other components as well.
The three-dimensional boxes represent nodes, either software or hardware.

Figure 2.9 depicts a Deployment diagram.

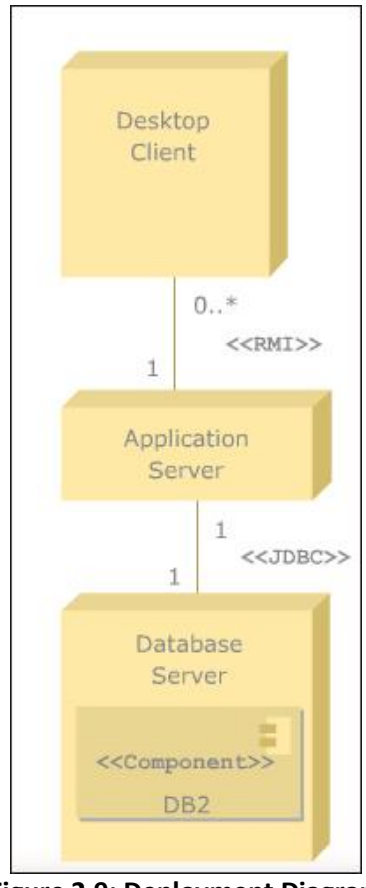


Figure 2.9: Deployment Diagram

In figure 2.9, the client node interacts with an application Server node that, in turn, connects to the Database server.

Connections between nodes are represented with **simple lines**, and can be assigned stereotypes such as JDBC to indicate the type of connection.

- **Components**
  A component is defined as a physical and replaceable part of a system that has a real set of interfaces.

Knowledge Check 2

1. Match the usage of diagrams/elements with the corresponding representations.

	REPRESENTATION		DIAGRAM
(A)	Illustrate the structural links between objects	(1)	Deployment diagrams
(B)	Show order of messages between the objects	(2)	A fork in an activity diagram
(c)	Useful for splitting the flow of control into executions of different actions	(3)	Sequence diagrams

(D)	Contain nodes representing hardware or functionalities in the system	(4)	Shown by a State Chart diagram
(E)	The transition of an object from one state to another	(5)	Collaboration diagrams

(A)	A-4, B-5, C-1, D-2, E-3	(C)	**A-5, B-3, C-2, D-1, E-4**
(B)	A-2, B-3, C-4, D-1, E-2	(D)	A-3, B-4, C-5, D-1, E-2

2.3 More on UML Elements

In this third lesson, **More on UML Elements,** you will learn to:

- Define and describe Stereotypes
- Define and describe Tagged Values
- Define and describe Constraints
- Define and describe Packages

2.3.1 UML Elements

As a designer or modeler, you sometimes find that you need a modelling element that is not in the UML but is very similar to one of the existing elements. That is when you construct a Stereotype of the existing element.

These are modelling **mechanisms** that allow you to **extend** the UML modelling elements.

In other words, Stereotypes allow you to extend vocabulary of UML to create new model elements, derived from existing ones. The new element may contain additional semantics, but still applies in all instances where the original element is used. They have specific properties that are suitable for your problem domain. Therefore, they help simplify the overall notation.

Usage of Stereotypes:

Stereotypes are enclosed by guillemets (<<...>>) and placed above the name of another element **<<**Stereotype name**>>**. Guillemets are punctuation marks marking the beginning and end of quotations.

Figure 2.10 depicts an example of gulliemets.

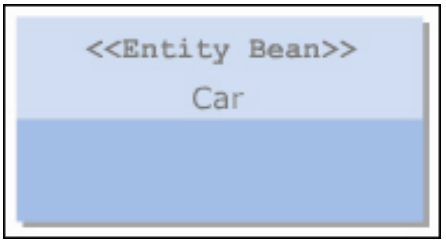


Figure 2.10: Gulliemets

A unique icon may be defined for the stereotype and the new element may be modelled using the defined icon or the original icon with the stereotype name displayed.

Stereotypes can be applied to all modelling elements like classes, relationships, and components.

Each UML element can have only one stereotype.

Stereotypes may be used in modifying code generation behavior, using a different or domain specific icon or color anywhere an extension is needed or if it would be helpful in making a model more clear or useful.

Examples:

- In a class diagram, stereotypes can be used to classify method behavior such as «constructor» and «getter»
- «interface» can be considered as a stereotype derived from class

Syntax: <<Stereotype name>>

2.3.2 Tagged Values

Tagged values also allow you to extend the properties of the basic UML building blocks like classes and packages.

They are the properties or specific attributes of a UML element.

Some properties are defined by UML. They are the predefined properties. Modellers or Designers can also create properties for any purpose.

Tagged values are placed within braces beneath the name of the element to which the property applies.

Properties are often used for third-party tools, code generators, and to express language-specific or process-specific information.

Examples:

- Persistence is a predefined UML property that can be applied to attributes, classes, and object. It denotes whether or not the items state is retained when the instance is destroyed

- Location is a predefined UML property that can be applied to attributes, classes, and objects; it specifies the component where the class is. For components, it denotes what node the component resides on

- Configuration management information like version number and status is another property

Figure 2.11 depicts the use of Tagged values.

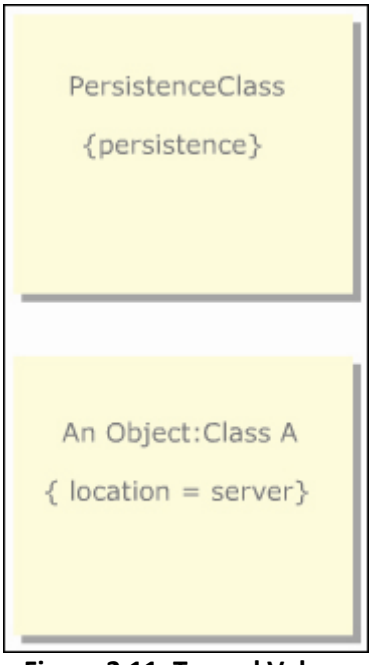


Figure 2.11: Tagged Values

2.3.3 Constraints

A Constraint can be any specific information. Example: In a sales system, if a customer has a certain code, he can avail a certain discount. Others cannot avail the discount.

Constraints can be used in class diagrams.

Constraints allow for addition of new semantics, or for changing existing semantics in a UML model. What is important is making it as simple and readable as possible.

Constraints are enclosed in braces and placed near the element the constraint applies to.

If a constraint needs to be attached to more than one element, then dependency relationships can be specified.

Figure 2.12 depicts the use of Constraints.

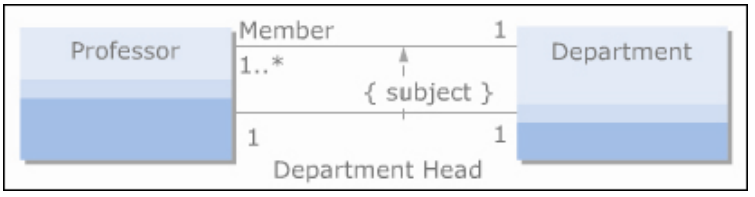


Figure 2.12: Constraints

In figure 2.12, a professor can only be the Department Head for a Department of which he is a Member.

2.3.4 Packages

Any complex system needs to be broken down into smaller systems for better understanding and making it less overwhelming. It makes the system modular, helping in managing it better.

While breaking down the bigger system, you try to group them into meaningful and related systems that are smaller. This organizing of elements into logical groups is called a Package.

A package contains classes that are needed by a number of different packages and/or subsystems, but are treated as a 'behavioral unit'.

An element in UML cannot be owned by more than one package.

Figure 2.13 depicts a Package diagram.

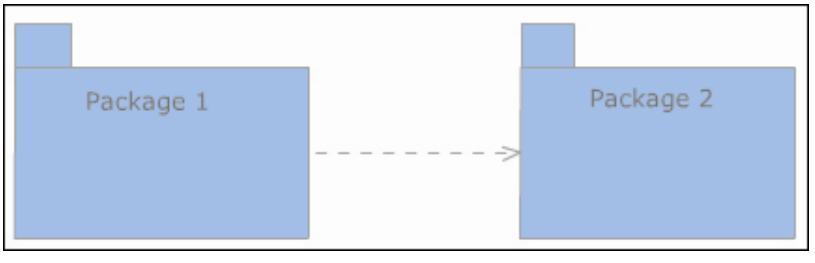


Figure 2.13: Package Diagram

Knowledge Check 3

1. Which of the statements about UML Elements are true?

(A)	Stereotypes can be applied to any modelling element (class, relationship, etc)
(B)	Tagged Values are properties that can be used to express domain specific information
(C)	Constraints should be as semantically cryptic as possible
(D)	Packages may show the hierarchy of a system
(E)	An element of UML can occur in more than one package

(A)	B, D, E	(C)	A, B, D
(B)	A, B, E	(D)	A, B, C

2. Match the description with the features.

	DESCRIPTION		FEATURE
(A)	This emphasizes on the links during interactions between objects	(1)	Model
(B)	This diagram is a general purpose entity for organizing classes	(2)	Actor
(C)	Predefined UML property that can be applied to attributes classes and objects	(3)	Class diagram
(D)	Blue print of a system	(4)	Object diagram
(E)	A part of use case diagram	(5)	Persistence

(A)	A-4, B-3, C-5, D-1, E-2	(C)	A-5, B-3, C-2, D-1, E-4
(B)	A-2, B-3, C-4, D-1, E-2	(D)	A-3, B-4, C-5, D-1, E-2

Module Summary

In this module, **UML Overview**, you learnt about:

- **UML Diagrams**
  UML Diagrams are used to visualize the system in its various aspects. A few important UML Diagrams are, Use Case diagrams, Object diagrams, Class diagrams, Sequence diagrams, Collaboration diagrams Sequence diagrams, State chart diagrams, Activity diagrams Component diagrams and Deployment diagrams

- **UML Elements**
  A few important aspects of UML elements include Stereotypes, Tagged Values, Constraints, and Packages

MODULE 3

Working with VP-UML

Welcome to the module, **Working with VP-UML**. This module introduces working with VP-UML. It describes the purpose features and advantages of VP-UML as a tool. It helps you learn about how to work in the VP UML environment.

In this module, you will learn about:

- Purpose, features, and the advantages of VP-UML as a tool
- Using the VP-UML environment to create projects and diagram elements

3.1 VP-UML as a Tool

In this first lesson, **VP-UML as a Tool**, you will learn to:

- Explain the purpose of using VP-UML as a tool
- Describe the features of VP-UML
- List the advantages of VP-UML

3.1.1 Purpose of using VP-UML as a tool

VP-UML is a powerful UML case tool. It provides an environment to carry out various activities of Object-Oriented analysis and design through easy drag and drop operations to create UML diagrams.

VP-UML can be used to perform a detailed domain modeling and analysis of the problem in hand using its powerful features.

3.1.2 Features of VP-UML.

VP-UML provides a development workspace, which allows us to create different types of diagrams in a visual environment.

It provides a collection of menus, toolbars, and windows that make up the workspace.

VP-UML is equipped with functionalities like generation of code from the UML diagrams, reverse engineering and so on.

The professional edition of VP-UML not only provides a visual modeling for both logical data design and physical data design, but also automates the mapping between object model and data model.

3.1.3 Advantages of VP-UML.

- VP-UML generates a cost-effective, reliable, scalable and high-performance object to relational mapping layer
- VP-UML increases the productivity and significantly reduces the risk of developing the artifacts of a system manually
- VP-UML is capable of generating Java and .NET persistent code
- VP-UML provides an extension for the major Integrated Development Environments (IDEs), including Eclipse, Borland JBuilder®, NetBeans/Sun. ONE etc
- It is designed for a wide range of users, including Software Engineers, System Analysts, Business Analysts and System Architects alike

Knowledge Check 1

1. Which of the following activities can be carried out using VP-UML?

(A)	Object-oriented analysis
(B)	Subject-oriented analysis
(C)	Object-oriented planning
(D)	Subject-oriented planning

2. Which of these features does VP-UML provide?

(A)	Selections, texts and phrases
(B)	Menus, toolbars and windows
(C)	Pop-ups, drop-downs and status bars
(D)	Drop-downs, selections and tool-bars

3. Which of the following is true in developing the artifacts of a system manually?

(A)	Reduces productivity and increases risk
(B)	Increases productivity and increases risk
(C)	Increases productivity and reduces risk
(D)	Reduces productivity and reduces risk

Module Summary

In this module, **Working with VP UML**, you learnt about:

- **VP-UML as a Tool**
  VP UML is used to perform a detailed domain modeling and analysis of the problem in hand. It provides a development workspace, which allows us to create different types of diagrams in a visual environment. VP-UML increases the productivity and significantly reduces the risk of developing the various artifacts of a system manually.

Onlinevarsity
your e-way to learning

Visit

Frequently Asked Questions

@

www.onlinevarsity.com

MODULE 4

Use Case Diagram

Welcome to the module, **Use Case Diagram**. This module begins with a brief explanation on use case diagram. It introduces use case diagram, explains the essential elements of Use-Case diagrams and finally, it provides an overview on developing Use-Case Model Methodology.

In this module, you will learn about:

- Use-Case diagrams
- Use-Case in action

4.1 Use Case Diagram

In this first lesson, **Use Case Diagram**, you will learn to:

- Define and describe System Boundary
- Define and describe a Use Cases
- Define and describe Actors
- Define and describe Connectors
- List and describe the Essential Elements
- List the advantages of Use Case Diagrams

Use case diagram describes what the system will do.

Use-Case model serves as a contract between:

- The customer
- The users
- The system developers

Use-Case model allows customers and users to validate that the system will become what they expect, and system developers to build what is expected.

The Use-Case model consists of:

- Use-Cases
- Actors

Figure 4.1 depicts the actors of a Use Case diagram.

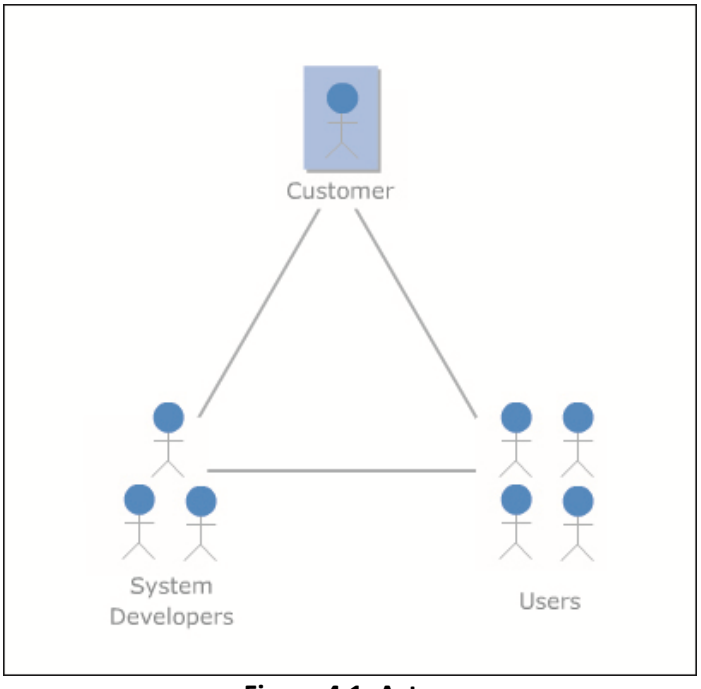


Figure 4.1: Actors

4.1.1 Defining a System Boundary

A system boundary element signifies a classifier, such as a class, component, or sub-system, to which the enclosed use cases are applied.

By depicting a boundary, its referenced classifier does not reflect ownership of the embodied use cases, but instead indicates usage.

Figure 4.2 depicts the System Boundary.

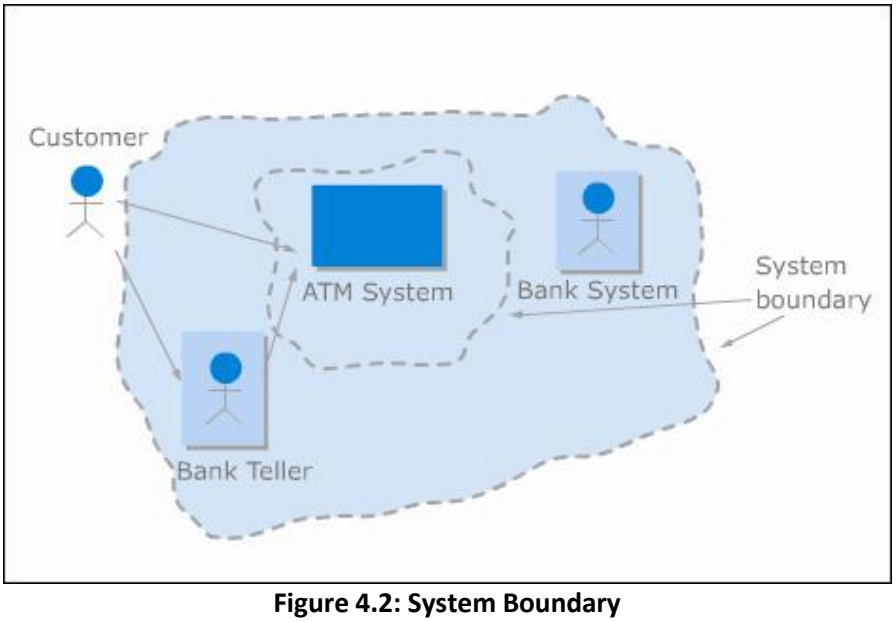


Figure 4.2: System Boundary

The choice of actors scopes the system.
If the inner circle is the system boundary, then there are three actors:

- Customer
- Bank Teller
- Bank System

If the outer circle is the system boundary, then there is only one actor,
- Customer

The system to be built depends on:

- Who the actors are
- Who will use the system
- The method for doing so

4.1.2 Defining a Use Case

A Use-Case is a sequence of actions a system performs that yield an observable result of value to a particular actor.

Use-Cases are the conduit between:

- The end users
- The developers

One of their primary purposes is to serve as a communication vehicle, so that end users and developers can clearly understand the requirements.

A Use-Case models a dialogue between:

- Actors
- The system

A Use-Case is initiated by an actor in order to invoke certain functionality in the system. A Use-Case is a complete and meaningful flow of events.

In order to “scope” the size of a Use-Case, it can be considered that a Use-Case represents a major system usage goal for one or more of the actors that interact with the Use-Case.

Taken together, all Use-Cases constitute all possible ways of using the system.

Figure 4.3 depicts the Use Cases.

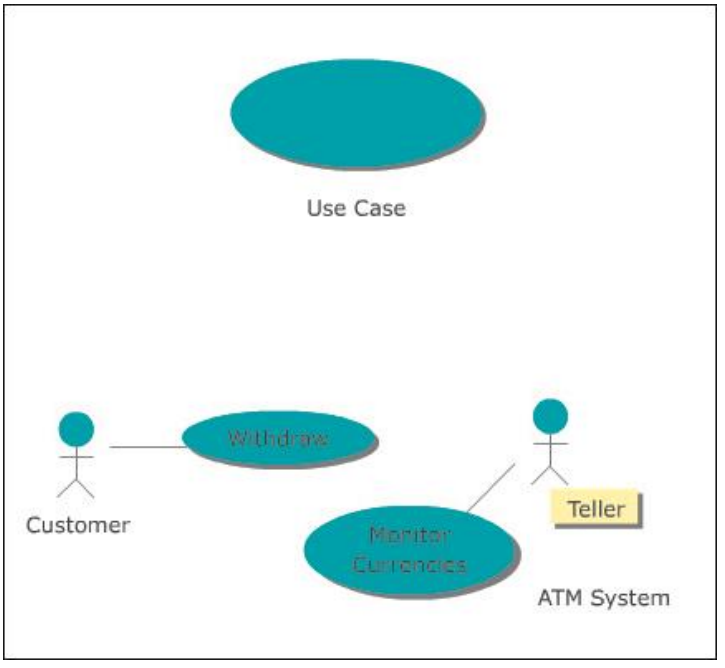


Figure 4.3: Use Cases

4.1.3 Defining an Actor

An **actor** represents anything that interacts with the system. An actor is external in that they are not part of the system.

They represent roles a user in the system can play. An actor may actively interchange information with the system or may be a passive recipient of information.

An actor can represent:

- A human
- A machine or another system

Figure 4.4 depicts the symbol for an Actor.

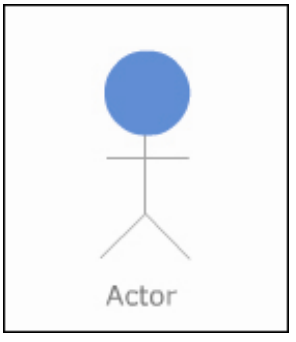


Figure 4.4: Actor

4.1.4 Connector

A connector is the line connecting two shapes on the diagram. A connector element is used to connect different elements of a use-case diagram.

VP-UML provides three styles for the connector:

- Rectilinear
- Oblique
- Curve

Figure 4.5 depicts Connectors.

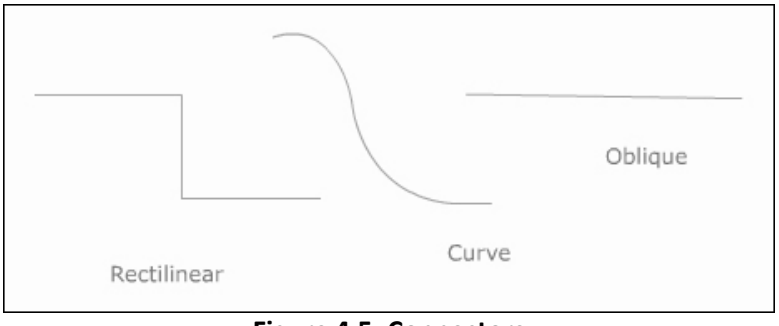


Figure 4.5: Connectors

4.1.5 Association

Association is the general relationship type between elements. Actors are associated with use-cases in use case diagrams. An interaction between an actor and a use-case is modeled as association. A use case can be carried out by many actors; an actor can carry out many use cases. Associations are indicated by solid lines with an optional arrowhead at one end of the line.

Figure 4.6 depicts Association.

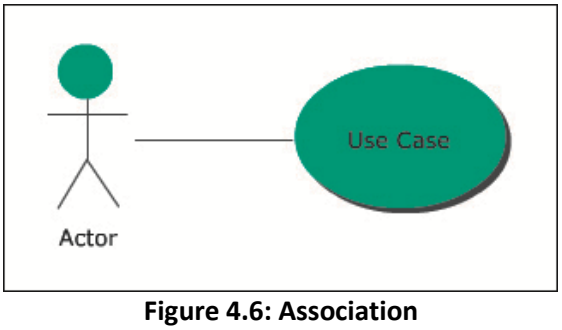


Figure 4.6: Association

4.1.6 Actor Generalization

Several actors can play the same role in a particular Use-Case.

Figure 4.7 depicts Actor Generalization.

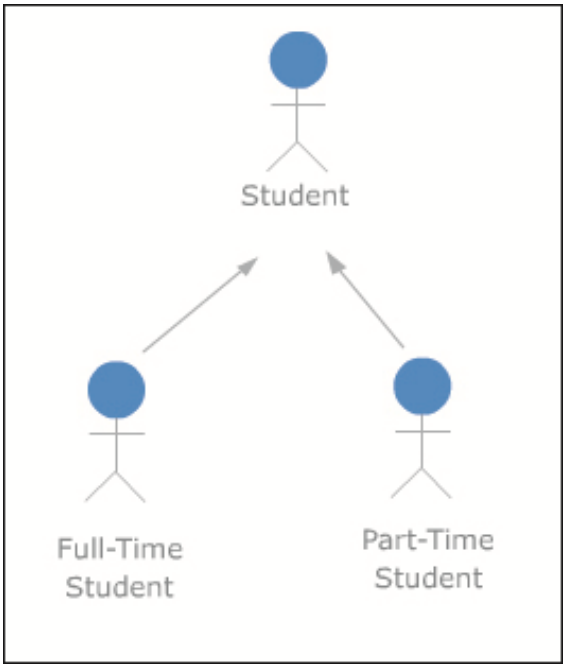


Figure 4.7: Actor Generalization

In the image, there are full-time and part-time students, both can register for courses, and are seen as the same external entity by the Use-Case that does the registering.

The shared role is modeled as an actor and student, which is inherited by the two original actors. This relationship is shown with actor-generalizations.

4.1.7 Include

An include connection indicates that the source element includes the functionality of the target element.

Include connections are used in use-case models to reflect that one use-case includes the behavior of another.

Figure 4.8 depicts Include relationship.

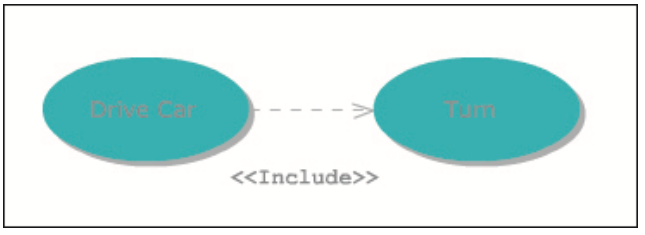


Figure 4.8: Include

4.1.8 Extend

An extend connection is used to indicate that an element extends the behavior of another.

Extensions are used in use-case models to indicate one use-case (optionally) extends the behavior of another.

Figure 4.9 depicts the Extend relationship.

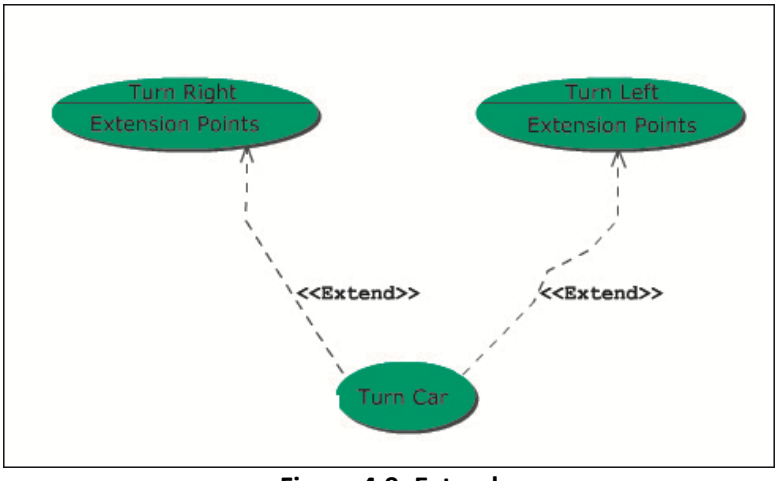


Figure 4.9: Extend

4.1.9 Dependency

Dependency is a relationship implying that a use case requires other another use case for its specification or implementation. Dependency relationships are used to model a wide range of dependent relationships between use cases. Dependency relationships occurs in different flavors such as:

- **Include** dependency: One use case includes the functionality of the other use case
- **Extend** dependency: One use case extends the functionality of the other use case

Figure 4.10 depicts dependency relationship.

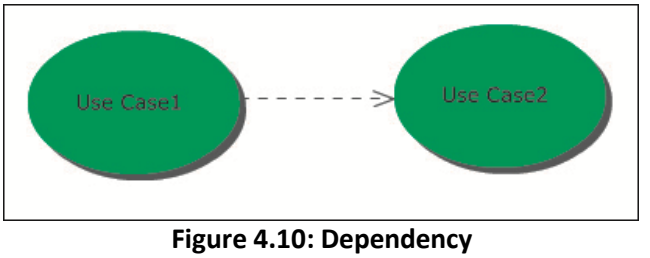


Figure 4.10: Dependency

4.1.10 Packages

Packages are just a general grouping mechanism for grouping elements into semantically related groups.

Packages can be used in the Use-Case Model to reflect order, configuration, or delivery units in the finished system.

Allocation of resources and the competence of different development teams may require that the project should be divided among groups at different sites.

One can use Use-Case packages to structure the Use-Case model in a way that reflects the user types. In UML, a package is represented as a tabbed folder.

Figure 4.11 depicts Use Case Packages.

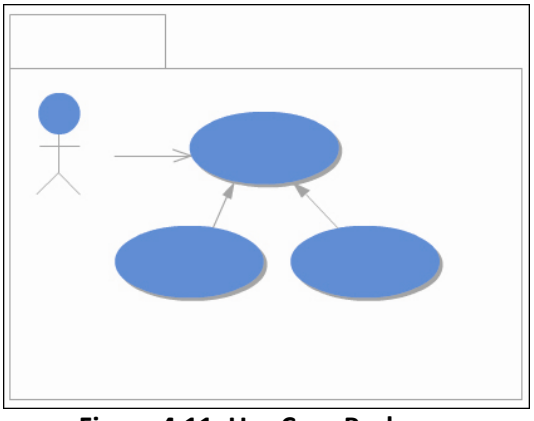


Figure 4.11: Use Case Packages

4.1.11 Note

Note denotes a simple comment or a note about a use-case or the system. It is represented by a box that has one corner turned down. It contains text that does not fit under a specific section. Note boxes can be used to add more detail to the requirements for the use case action. Note can be attached to any set of elements.

Figure 4.12 depicts the symbol of a Note.

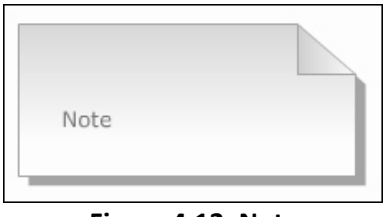


Figure 4.12: Note

4.1.12 Advantages

Advantages of the Use-Case diagrams:

- Use-cases are powerful tools for analysts to use when partitioning the functionality of a system
- Use-case relationships and the corresponding diagrams help analysts to structure use-cases such that their textual descriptions contain minimum redundant information; thus making the whole text document much easier to maintain
- Use-cases are not design tools. They do not specify the structure of the eventual software, nor do they imply the existence of any classes or objects. They are purely functional descriptions written in a formalism that is completely separate from software design

Knowledge Check 1

1. Match the following statements against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	The enclosed Use-Cases are applied to	(1)	Usage
(B)	Depicting a boundary indicates	(2)	Actors
(C)	Referenced classifier in a boundary does not reflect	(3)	System Boundary
(D)	System to be built depends on	(4)	Ownership of the embodied Use-Cases

(A)	A-3, B-4, C-1, D-2	(C)	A-4, B-1, C-2, D-3
(B)	A-2, B-3, C-4, D-1	(D)	A-3, B-1, C-4, D-2

2. Match the following statements against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Sequence of actions a system performs to a particular actor is a	(1)	Actors & the System
(B)	Primary purpose of use-case is to serve as a	(2)	Use-Case
(C)	Use-case models a dialogue between	(3)	The system
(D)	Use-cases constitute all possible ways of using	(4)	Communication Vehicle

(A)	A-3, B-4, C-1, D-2	(C)	A-4, B-1, C-2, D-3
(B)	A-2, B-4, C-1, D-3	(D)	A-3, B-1, C-4, D-2

3. Which of these statements about Actors are true?

(A)	Anything that interacts with the system represents an actor
(B)	An actor may actively interchange information with system
(C)	An actor does not represent roles a user in the system can play
(D)	Actor can be a passive recipient of information
(E)	An actor represents only humans

(A)	A, B, C	(C)	A, B, D
(B)	B, D, E	(D)	A, C, E

4. Match the following statements against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	The line connecting two shapes on the diagram is a	(1)	Use-Case Diagrams
(B)	Connector element connects different elements of a	(2)	VP-UML
(C)	Rectilinear, Oblique & Curve are styles provided by	(3)	Connector

(A)	A-3, B-2, C-1	(C)	A-1, B-2, C-3
(B)	A-2, B-3, C-1	(D)	A-3, B-1, C-2

5. Match the following statements against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Actors are associated with use-cases in	(1)	Dependent Relations
(B)	To indicate that one use-case extends the behavior of another is shown by	(2)	Tabbed folder
(C)	Dependency relationships are used to model which relationships between the cases	(3)	Use-Case Diagram
(D)	In UML, a package is represented as a	(4)	Extend Connection

(A)	A-3, B-4, C-1, D-2	(C)	A-4, B-1, C-2, D-3
(B)	A-2, B-4, C-1, D-3	(D)	A-3, B-4, C-1, D-2

6. Which of these statements about advantages of use-case diagrams are true?

(A)	Use-Cases are powerful tools to use when partitioning the functionality of the system
(B)	Use-Case diagrams do not specify the structure of the eventual software
(C)	Use–cases are design tools
(D)	Use-cases are not purely functional descriptions separate from software design

(A)	A, B, C	(C)	A, B, D
(B)	B, D, E	(D)	A, B

4.2 Use Case in Action

In this second lesson, **Use Case in Action**, you will learn to:

- Explain the use of System Boundary
- Explain the use of Use Cases
- Explain the use of Actors
- Explain the use of Stereotypes

- Explain the use of Connectors

4.2.1 Concept

Use-Cases defined for a system are the basis for the entire development process. They provide the unifying thread through the system and define the behavior of a system. Use-Cases play a role in each of the core process workflow:

- The Use-Case model is a result of the requirements workflow. In this early process, the Use-Cases are needed to model what the system should do from the user's point of view

- In Analysis and Design, Use-Cases are realized in a design model. One should create Use-Case realizations, which describe how the Use-Case is performed in terms of interacting objects in the design model. This model describes, in terms of design objects, the different parts of the implemented system, and how the parts should interact to perform the Use-Cases

- During implementation, the design model is in the implementation specification. Because Use-Cases are the basis for the design model, they are implemented in terms of design classes

- During test, the Use-Cases constitute the basis for identifying test cases and test procedures. The system is verified by performing each Use-Case

- Use-Cases have other roles as well. They could be the:

  - Basis for planning an iterative development

  - Foundation for what is described in user manuals

  - Definition of ordering units. For example, a customer can get a system configured with a particular mix of Use-Cases

Use-Case diagram is drawn to illustrate that Use-Cases and actors interact sending stimuli to one another.

Figure 4.13 depicts a Use Case Diagram.

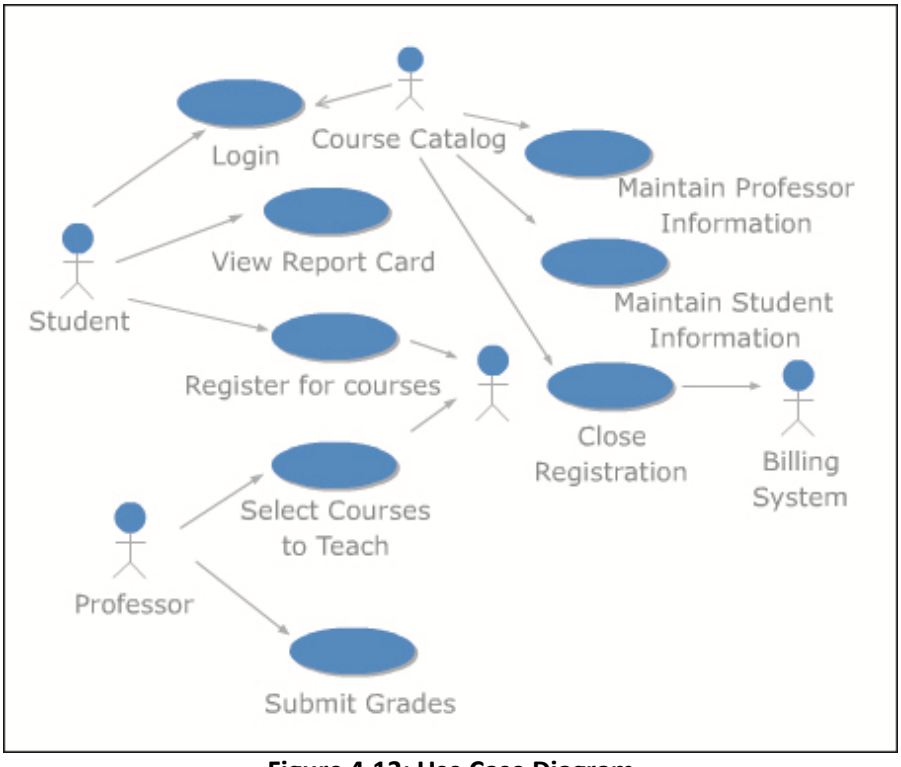


Figure 4.13: Use Case Diagram

4.2.2 System Boundary

A system boundary element signifies a classifier, such as a class, component, or sub-system, to which the enclosed use cases are applied. By depicting a boundary, its referenced classifier does not reflect ownership of the embodied use cases, but instead indicates usage. The choice of actors scopes the system. If the inner circle is the system boundary, then there are three actors: Customer, Bank Teller, and Bank System. If the outer circle is the system boundary, then there is only one actor, the Customer. The system to be built depends on the actors who will use the system, and the method for doing so.

4.3 Use case

A strict approach to design might begin just with the system object and work from there by identifying related responsibilities and creating classes with those responsibilities. This approach could then be continued carefully, distributing responsibilities and eventually determining a design.
A complete system may have many use cases and responsibilities, making a strict decomposition very difficult. Finally, a strict approach would make it difficult to allow consideration of design structures that arise from elsewhere.

Use-Case Analysis is an important state in the OOAD. It describes how a particular Use-Case is realized within the design model, in terms of collaborating objects. A Use-Case realization specifies what classes must be built to implement each Use-Case.

Use-Case Analysis is performed by the Designer, one per iteration per use-case realization. The flows of events, and therefore the use-case realizations to be worked on during the current iteration are defined

prior to the start of a Use-Case Analysis in Architectural Analysis.

4.3.1 Use case Analysis

- **Purpose**

The purposes of a Use-case analysis are to:

- Identify the classes which perform a use case's flow of events
- Distribute the use-case behavior to those classes, using the use-case realizations
- Identify the responsibilities, attributes and associations of the classes
- Note the usage of architectural mechanisms

The resulting artifacts from a Use-case analysis are as listed below:

- Analysis classes
- Analysis model and/or Design model

4.3.2 Actor

Identifying the actors is an easy task.

The most obvious candidates for actors are the humans in the system; except in rare cases when the system we are describing is actually a human process (such as a specific method of dealing with customers that employees should follow) the humans that we must interact with will all be actors.
If the system interacts with other systems (databases, servers maintained by other people, legacy systems) it is better to treat these as actors, also.

If there are interactions between the actors in your system, we cannot represent those interactions on the same diagram as our system.

It is better to draw a separate use-case diagram, treating one of the actors itself as a system, and our original system (along with the other actors) as actors on this new diagram.

- **Example:** For example, browser software has the user as an actor as well as the server at the other end as an actor.

4.3.3 Stereotype

Stereotypes allow one to extend the basic UML notation by allowing to define a new modeling element based on an existing modeling element.

The new element may contain additional semantics, but still applies in all instances where the original element is used.

In this way, the number of unique UML symbols is reduced, simplifying the overall notation.

Use case is provided by default with the UML standard stereotypes (metaclass, `powertype`, `process`, and `thread`, `utility`) for classifiers.

Stereotyping can be useful when creating use cases in the problem domain (requirements capture) and solution domain (analysis), but none of the predefined stereotypes is well suited to this.

The name of a stereotype is shown in guillemots, for example, <<stereotype name>>.

4.3.4 Connector

A connector can be used to connect diagram elements. There are 3 connector styles that are provided in VP-UML. Any connector style that is suitable for our created diagram can be chosen.

Figure 4.14 shows the different connector styles.

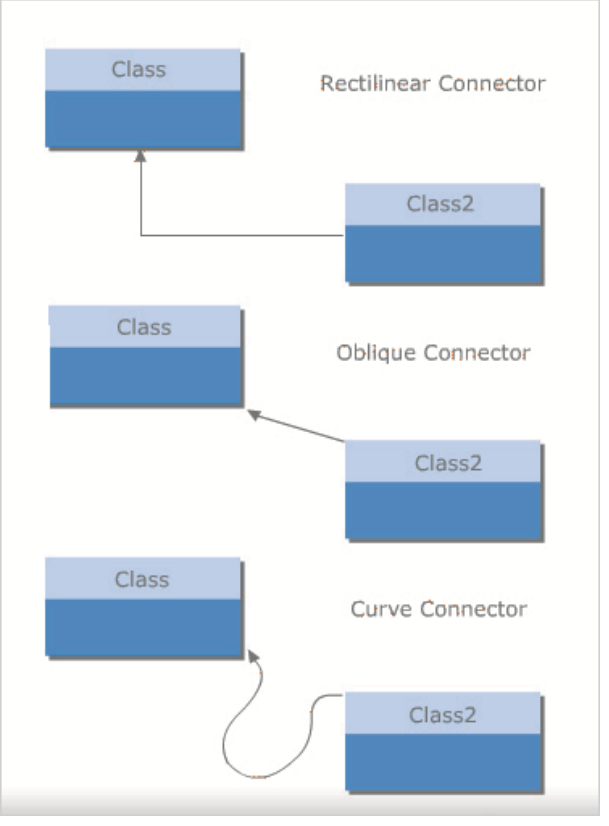


Figure 4.14: Connector Styles

Knowledge Check 2

1. Match the following statements against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	The familiar approach in design is to present the system as a	(1)	System boundary
(B)	The actor's intention from the system's responsibility is separated by	(2)	Boundary of the system
(C)	With essential use-cases, responsibilities can be used to determine	(3)	Method parameter & return value
(D)	The interaction in a use-case resembles	(4)	Black-box

(A)	A-3, B-4, C-1, D-2	(C)	A-4, B-1, C-2, D-3
(B)	A-2, B-4, C-1, D-3	(D)	A-3, B-4, C-1, D-2

2. Match the following statements against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Use-case analysis is an important state in	(1)	Designer
(B)	Use-case analysis is performed by	(2)	Analysis classes & Analysis / Design Model
(C)	Purpose of Use-case analysis is to note the usage of	(3)	OOAD
(D)	To identify the classes which perform a use-cases flow of events is the purpose of	(4)	Architectural mechanisms
(E)	The resulting artifacts from a Use-case analysis are	(5)	Use-case analysis

(A)	A-3, B-1, C-4, D-5, E-2	(C)	A-4, B-1, C-5, D-3, E-2
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-3, B-4, C-1, D-5, E-2

3. Which of these statements about use of Actors are true?

(A)	To identify the actors is a complex task
(B)	If the system interacts with another system, they cannot be treated as actors
(C)	The interactions between actors in a system cannot be represented on the same diagram

(A)	A, B	(C)	C
(B)	B, C	(D)	B

4. Match the following statements against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Stereotypes allow one to extend	(1)	Guillemots
(B)	The name of a stereotype is shown in	(2)	Problem domain & Solution domain
(C)	Use case is provided by default with the	(3)	Basic UML notation
(D)	Stereotyping can be useful when creating use cases in the	(4)	UML standard stereotypes

(A)	A-3, B-4, C-1, D-2	(C)	A-4, B-1, C-2, D-3
(B)	A-2, B-4, C-1, D-3	(D)	A-3, B-1, C-4, D-2

5. Which of these statements about use of Connectors are true?

(A)	Connector is used to connect diagram elements
(B)	VP-UML provides 2 connector styles
(C)	Any connector style suitable for the created diagram can be chosen

(A)	A, B	(C)	A, C
(B)	B, C	(D)	B

Module Summary

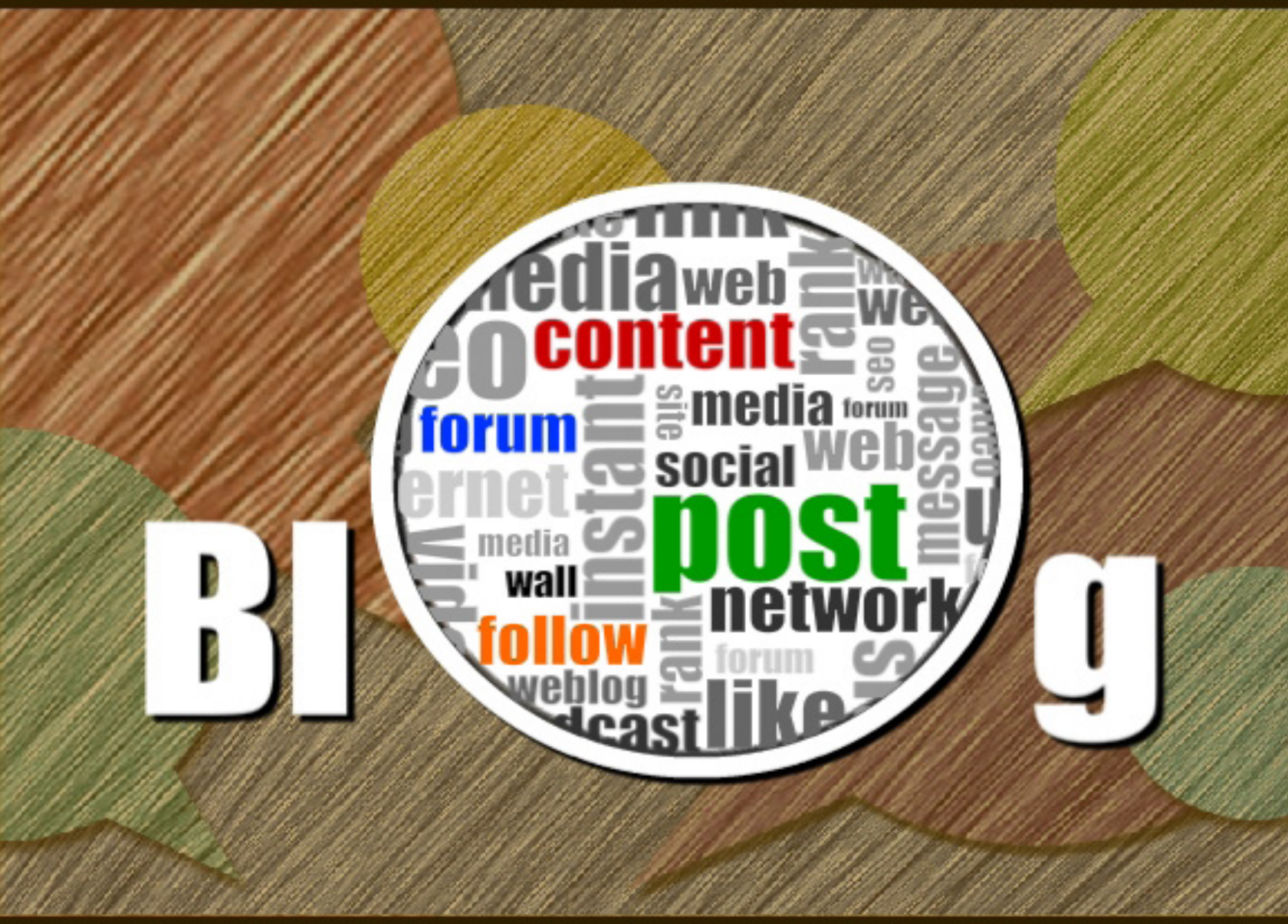
In this module, **Use-Case Diagram**, you learnt about:

- **Use-Case Diagram**
  The Use-Case Diagram describes what the system will do. It serves as a contract between the customer, the users, and the system developers. It allows customers and users to validate that the system developed will become what they expect. It helps the system developers to build what is expected.

- **Use-Case in action**
  Use-Cases defined in the system are the basis for the entire development process. In Analysis and Design phase, Use cases are realized in design-model that is implemented in terms of design classes during implementation phase. During test phase, the Use-Cases constitute the basis for identifying test cases and test procedures. Use-Case diagram are drawn to illustrate that Use-Cases and actors interact by sending stimuli to one another.

Onlinevarsity

your e-way to learning



Balanced Learner-Oriented Guide

for enriched learning available @

www.onlinevarsity.com

MODULE 5

Static Modeling

Welcome to the module, **Static Modeling**. The module begins by defining Class Diagrams and explains their purpose. It describes relationships and terminology in Class Diagrams and various other Modeling Elements.

In this module, you will learn about:

- Class Diagrams
- Other Modeling Elements

5.1 Class Diagrams

In this first lesson, **Class Diagrams**, you will learn to:

- Define and describe the purpose of Class Diagrams
- List and describe the elements of Class Diagrams
- Explain relationships in Class Diagrams
- Describe terminologies used in Class Diagrams

5.1.1 Purpose and Definition

The Class Diagram is the most essential part of UML modeling as it represents the static structure of our solution, modeled at the most detailed level.

When the developers develop the solution, their main source of information is the class diagram.

The Class Diagram provides developers with detailed information about what operation code to develop, along with information about data types, parameters, and namespaces.

The most important part of our UML modeling is that we can generate code only from our static structure diagrams, and class diagrams are the only diagram type from which we'll have a code skeleton for ready use.

5.1.2 Elements

The different elements of Class Diagram are as follows:

- Class
- Representation
- Attributes
- Operation
- Class Visibility

Figure 5.1 depicts the elements of a Class diagram.

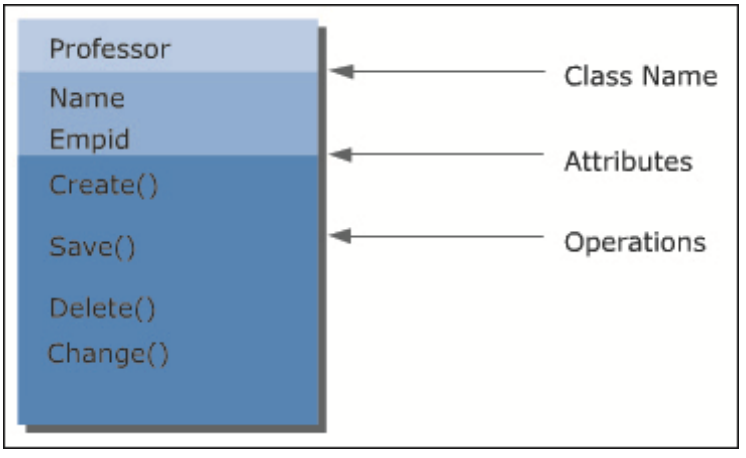


Figure 5.1: Elements of a Class DiagramClass

A class is a description of a group of objects with common properties (attributes), behavior (Operations), relationships, and semantic.

There are many objects identified for any system or domain. Recognizing the commonalities amongst the objects and defining classes help us deal with the potential complexity. The key OO principal, abstraction, helps one deal with complexity.
A class is an abstraction in that it:

- Emphasizes relevant characteristics
- Suppresses other characteristics

➢ **Representation**
A class is represented using a compartmentalized rectangle.
As seen in the figure, class representation comprises of three sections:
- The first section contains the class name
- The second section shows the structure (attributes)
- The third section shows the behavior (operations)

➢ **Attributes**
Attributes represent essential characteristics of the class. Attributes provide information storage for the class instance, and are often used to represent the state of the class instance. Additional attributes may need to be added to support the implementation, and establish any new relationships that the attributes require. Any information the class itself maintains is done through its attributes.
For each attribute, the following should be defined:

- **Name**
  Its **name**, which should follow naming conventions of both the implementation language and the project.

- **Type**
  Its **type**, which will be an elementary data type supported by the implementation language.

- **Default or Initial value**
  Its **default or initial value**, to which it is initialized when new instances of the class are created.

- **Visibility**
  Its **visibility**, which will take one of the following values.

  - **Public**:
    The attribute is visible both inside and outside the package containing the class.

  - **Protected**: the attribute is visible only to the class itself, to its subclasses, or to friends of the class (language dependent).

  - **Private**:
    The attribute is only visible to the class itself and to friends of the class.

  - **Package Friendly**:
    The attribute is visible only within the package.

5.1.3 Operation

"An operation is the implementation of a service that can be requested from any object of the class to affect behavior".

An operation can either be a command or a question. A question should never change the state of the object. Only a command can change the state of the object. The outcome of the operation depends on the current state of the object.

Every class should have the following types of operations:
- Manager functions
- Implementer functions
- Access functions
- Helping functions
- Operations should be named to indicate their outcome;
- The operations should be named from the perspective of the client asking for a service to be performed by the class

Figure 5.2 depicts the operations of a class.

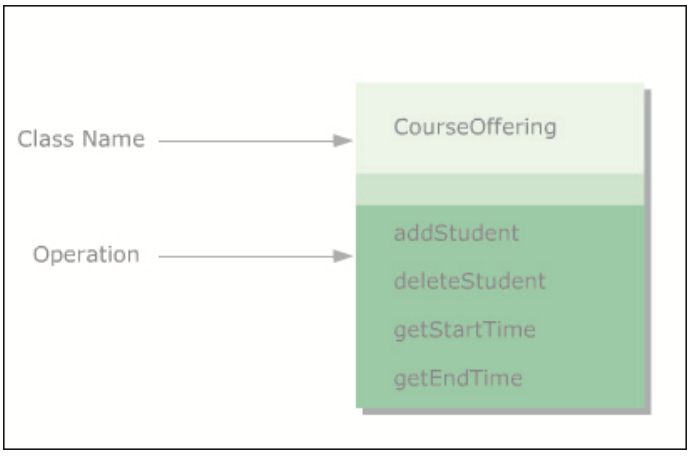


Figure 5.2: Operations of a Class

It is best to specify the operations and their parameters using implementation language syntax and semantics. In this way, the interfaces will already be specified in terms of the implementation language when coding starts.

- For example, **`getBalance()`** against **`calculateBalance()`**. One approach to naming operations that get and set properties is to name the operation the same as the property name. If there is a parameter, it sets the property; if not, it returns the current value.

- For example, **`getBalance()`** against **`receiveBalance()`**. The same applies to the operation descriptions. Descriptions should always be written from the operation user's perspective.

5.1.4 Class Visibility

In UML, we can specify the access clients that have attributes and operations.
Export control is specified for attributes and operations by preceding the name of the member with the following symbols:

- + Public
- # Protected
- - Private
- ~ package friendly

Figure 5.3 depicts the Visibility modifiers.

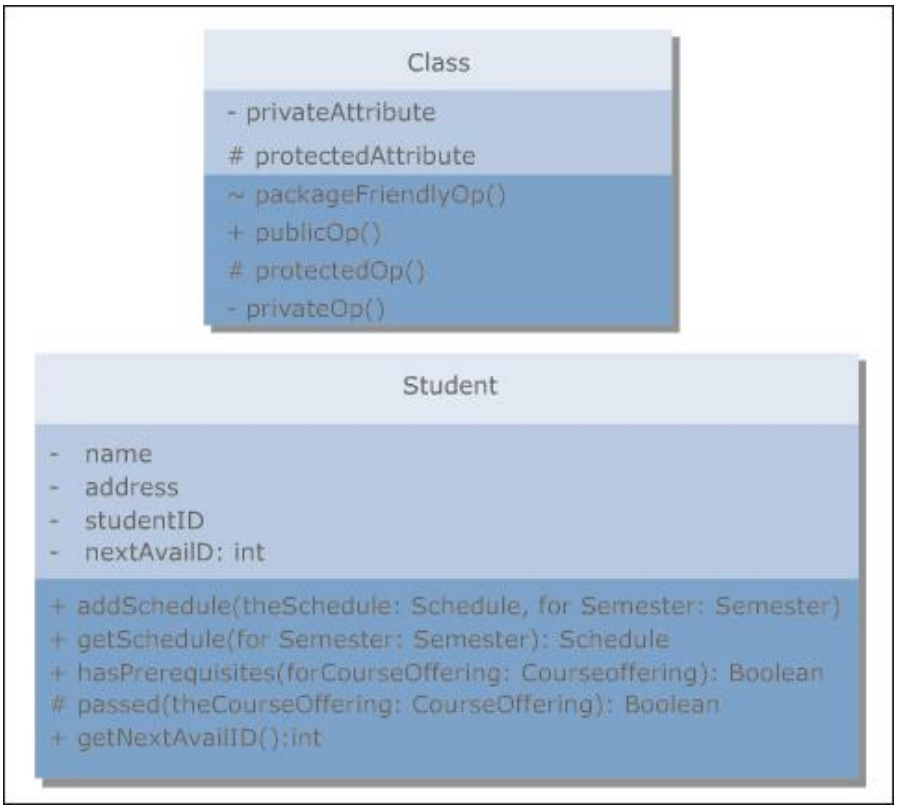


Figure 5.3: Class Visibility Modifiers

5.1.5 Relationships

Relationships provide a pathway for communication between objects. In UML, relationships are modelled as lines. Different kinds of lines used to represent the different kinds of relationships.
Here are the different types of relationships represented in UML.

- Association
- Aggregation
- Composition
- Generalization
- Realization

Figure 5.4 depicts the relationships among objects.

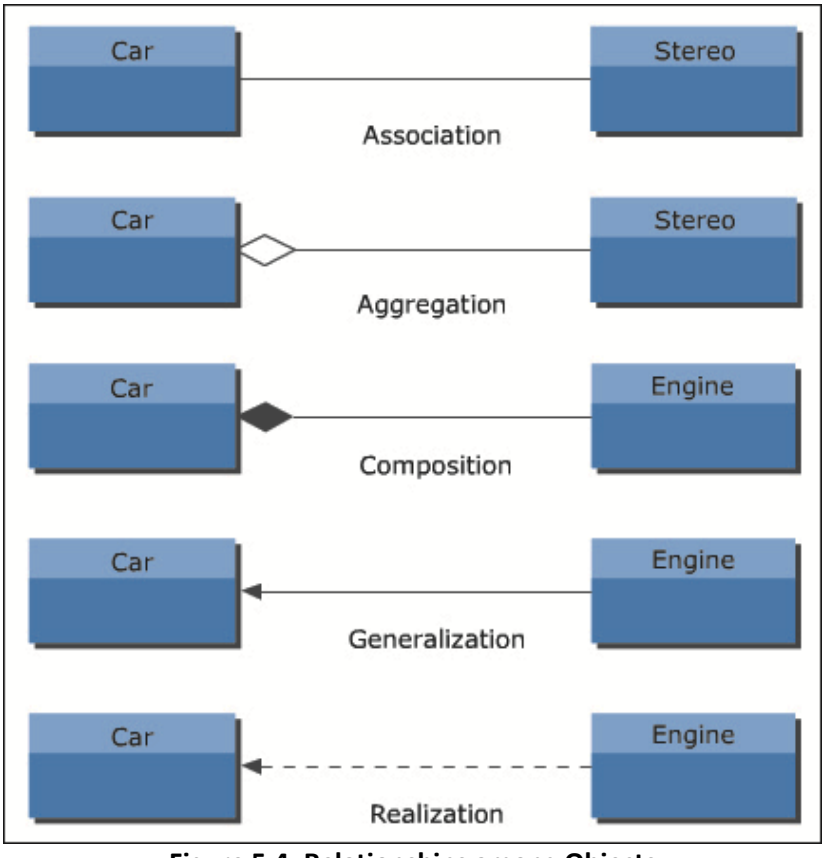


Figure 5.4: Relationships among Objects

5.1.6 Associations

An association is a bi-directional connection between classes. Associations represent structural relationships between objects of different classes, information that must be preserved for some duration and not simply procedural dependency relationships; for example, one object has a permanent association to another object. Each end of an association is a role specifying the face that a class plays in the association. The role must have name, and the role names opposite a class must be unique. The role name should be a noun indicating the associated object's role in relation to the associating object. The role name is placed next to the end of the association line. The use of association names and role names are mutually exclusive: one should not use both

an association name and role name. For each association, it should be decided as to which conveys more information.
Figure 5.5 depicts Association.

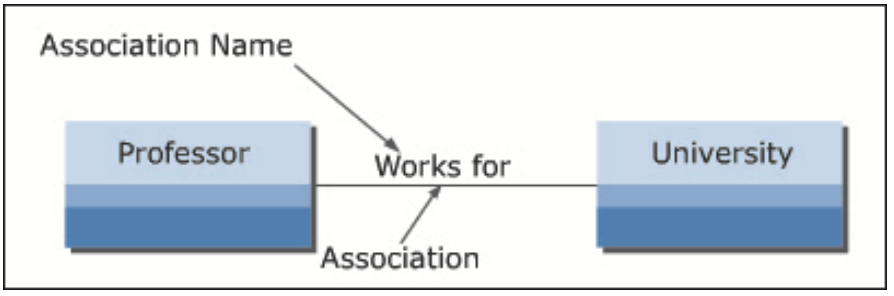


Figure 5.5: Association

- An association may have a name that is placed on or adjacent to the association path. The name of the association should reflect the purpose of the relationship and be a verb phrase; the name of an association can be omitted, particularly if roles names are used. Names like “has” and “contains” should be avoided, as they add no information about what the relationships are between the classes.

5.1.7 Association-Multiplicity & Navigation

Multiplicity defines how many objects participate in a relationship. It is the number of instances of one class related to ONE instance of the other class, which is specified for each end of the association.
Associations and aggregations are bi-directional by default, but it is often desirable to restrict navigation to one direction. If navigation is restricted, an arrowhead is added to indicate the direction of the navigation. For each role, one can specify the multiplicity of its class; how many objects of the class can be associated with one object of the other class. Multiplicity is indicated by a text expression on the role. The expression is a comma-separated list of integer ranges.

Figure 5.6 depicts Multiplicity and Navigation in an Association.

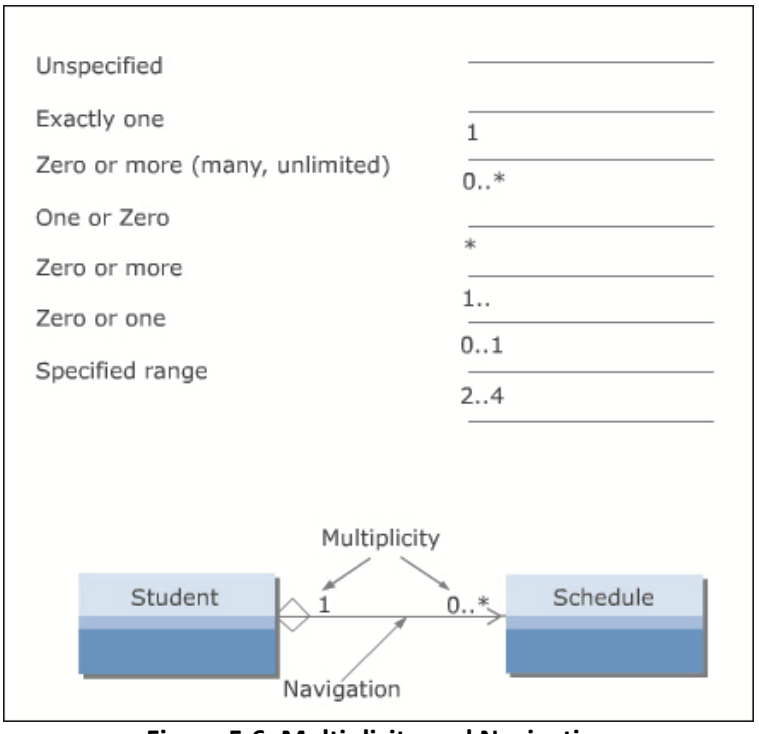


Figure 5.6: Multiplicity and Navigation

- A range is indicated by an integer (the lower value), two dots, and an integer (the upper value). A single integer is a valid range. During analysis, assume a multiplicity of 0..* (Zero to many) unless there is clear evidence of something else. A multiplicity of zero implies that the association is optional; if an object might not be there, operations, which use the association, will have to adjust accordingly. Narrower limits for multiplicity may be specified (such as 2..4). Within multiplicity ranges, probabilities may be specified. Thus, if the multiplicity is 0..*, it (the multiplicity) is expected to be between 10 and 20 in 85% of the cases. (This information will be of great importance during design). For example, if persistent storage is to be implemented using a relational database, narrower limits will help organize the database tables better.

5.1.8 Association-Multiplicity Design

Multiplicity is the number of instances that participate in an association. Initial estimates of multiplicity made during analysis must be updated and refined during design. All association and aggregation relationships must have multiplicity specified. For associations with a multiplicity of 1 or 0.1, further design is not usually required, as the relationship can be implemented as a simple value or a pointer.

For associations with a multiplicity upper bound that is greater than 1, additional decisions need to be made. This is usually designed using **"container"** classes. A container class is a class whose instances are collections of other objects. Common container classes include: sets, lists, dictionaries, stacks, and queues.

Figure 5.7 depicts Multiplicity.

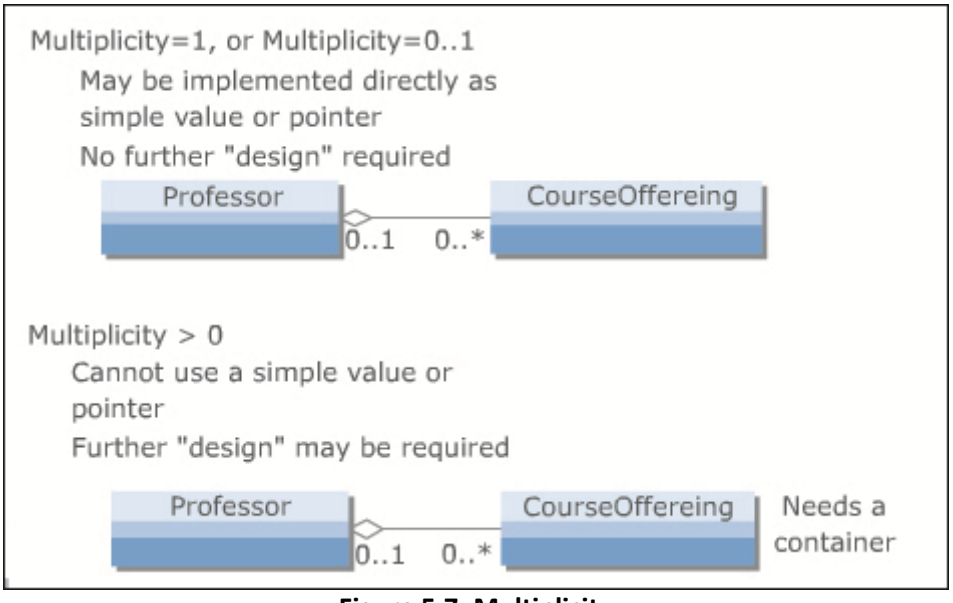


Figure 5.7: Multiplicity

- **Multiplicity Design: Optionality**
  If a link is optional, an operation to test for the existence of the link should be added to the class. For example, if a professor can be on sabbatical, a suitable operation should be included in the professor class to test for the existence of the **CourseOffering** link.

5.1.9 Aggregation

An aggregation is a stronger form of relationship where the relationship is between a whole and its parts. The aggregate has an aggregation association to the constituent parts. A hollow diamond is attached to the end of an association path on the side of the aggregate (the whole) to indicate aggregation. Since aggregation is a special form of association, the use of multiplicity, roles, and navigation is the same as for association.

Sometimes, a class may be aggregated with itself. This does not mean that an instance of that class is composed of itself; it means that one instance if the class is an aggregate composed of other instances of the same class.

Figure 5.8 depicts Aggregation.

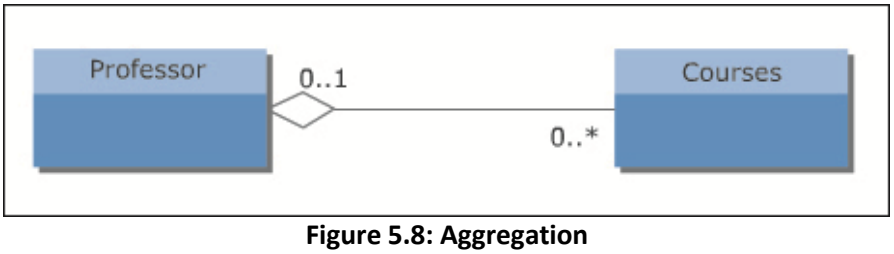


Figure 5.8: Aggregation

The following are some situations where aggregation may be appropriate:

i. An object is physically composed of other objects, for example, car being physically composed of an engine and four wheels
ii. An object is logical collection of other objects, for example, a family is a collection of parents and children
iii. An object can physically contain other objects, for example, an airplane physically contains a pilot

5.1.10 Composition

Composition is a form of aggregation with strong ownership and coincident lifetimes of the part with the aggregate. The whole "owns" the part and is responsible for the creation and destruction of the part. The part is removed when the whole is removed. The part may be removed (by the whole) before the whole is removed. A solid filled diamond is attached to the end of an association path (on the "whole side") to indicate composition.

Figure 5.9 depicts Composition.

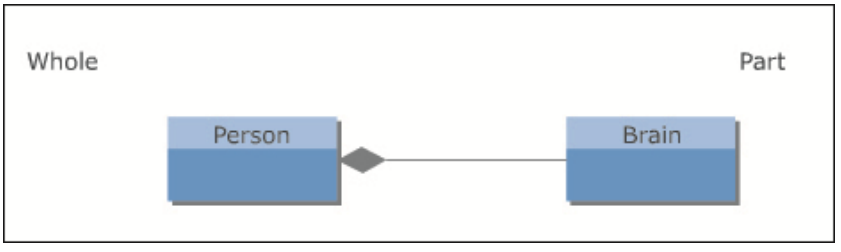


Figure 5.9: Composition

5.1.11 Aggregation or Composition

Composition should be used over "plain" aggregation when there is a strong interdependency relationship between the aggregate and the parts; where the definition of the aggregate is incomplete without the parts.

For example, it does not make sense to even have an order if there is nothing being ordered (that is Line Items). Composition should be used when the whole and part must have coincident lifetimes. Selection of aggregation or composition will determine how object creation and deletion are designed.

Figure 5.10 depicts Aggregation and Composition.

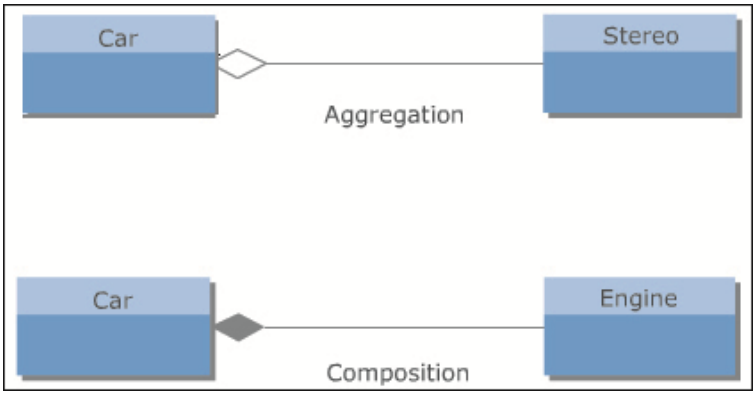


Figure 5.10: Aggregation and Composition

5.1.12 Dependency

A dependency is a **using** relationship that represents a relationship between a client and a supplier where a change in the specification of the supplier may affect the client.

A dependency relationship is a weaker form of relationship showing a relationship between a client and a supplier where the client does not have semantic knowledge of the supplier.

A dependency relationship denotes a semantic relationship between model elements, where a change in the supplier may cause a change in the client, or a relationship between two model elements, where a change in one may cause a change in the other.

Figure 5.11 depicts Dependency.

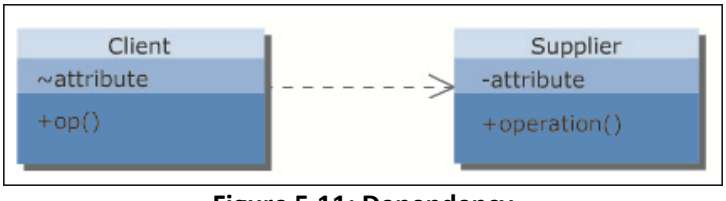


Figure 5.11: Dependency

5.1.13 Generalization

Generalization is a relationship among classes where one class shares the structure and/or behavior of one or more classes.
Generalization refines a hierarchy of abstractions in which a subclass inherits from one or more superclasses. Generalization is an “is-a-kind of” relationship. One should always be able to say that generalized class is a kind of the parent class.

The terms “ancestor” and “descendant” may be used instead of “superclass” and “subclass”.

Figure 5.12 depicts Generalization.

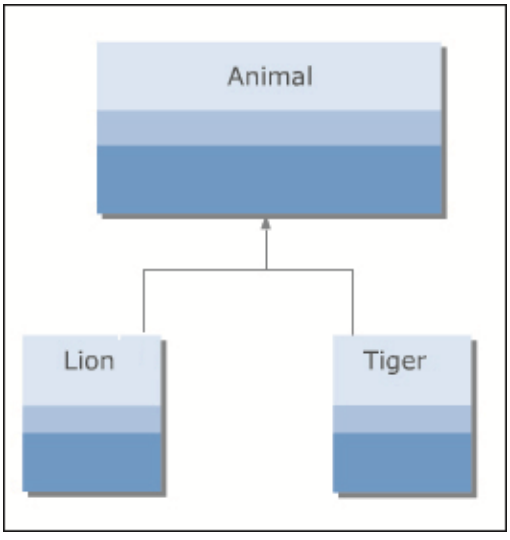


Figure 5.12: Generalization

5.1.14 Generalization-Abstract and Concrete classes

A class that exists only for other classes to inherit from it is an abstract class. Classes that are to be used to instantiate objects are concrete classes. An operation can also be tagged as abstract. This means that no implementation exists for the operation in the class where it is specified.

A class that contains at least one abstract operation must be an abstract class. Classes that inherit from an abstract class must provide implementations for the abstract operations, or the operations are considered abstract within the subclass, and the subclass is considered abstract, as well.

Concrete classes have implementations for all operations. In UML, a class is designated as abstract by putting the tagged value “{abstract}” within the name compartment of the class, under the class name. For abstract operations, “{abstract}” is placed after the operation signature. The name of the abstract item can also be shown in italics.

A discriminator can be used to indicate the basis on which the generalization/specialization occurred. A discriminator describes a characteristic that differs in each of the subclasses.

Figure 5.13 depicts Abstract and Concrete classes.

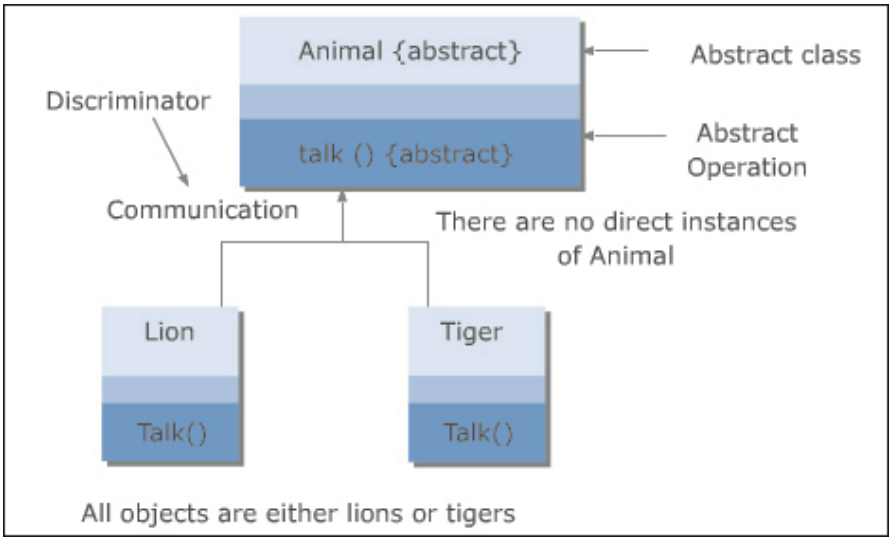


Figure 5.13: Abstract and Concrete Classes

5.1.15 Multiple Inheritance

In practice, multiple inheritance is a complex design problem and may lead to implementation difficulties. In general, multiple inheritance causes problems if any of the multiple parents has structure or behavior that overlaps. If the class inherits from several classes, one must check how the relationships, operations, and attributes are named in the ancestors.

Figure 5.14 depicts Multiple Inheritance.

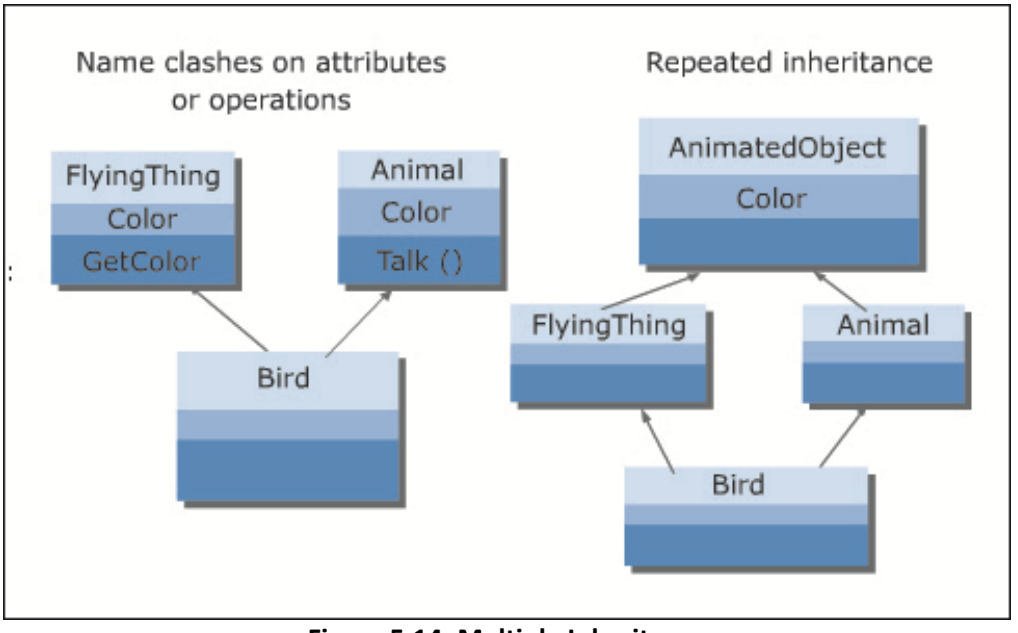


Figure 5.14: Multiple Inheritance

Specifically, there are two issues associated with multiple inheritance:

- Name collisions
- Repeated inheritance

In general, the programming language rules governing multiple inheritance are complex, and often difficult to use correctly. Therefore using multiple inheritance is recommended only when needed, and that too with caution.

- **Name collisions**
  Both ancestors have attributes and/or operations with the same name. If the same name appears in several ancestors, it must be described what this means to the specific inheriting class, for instance, by qualifying the name to indicate its source of declaration.

- **Repeated inheritance**
  The same ancestor is being inherited by a descendant more than once. When this occurs, the inheritance hierarchy will have a "diamond shape" as shown above. The descendants end up with multiple copies of an ancestor. If repeated inheritance is being used, a clear definition of its semantics is necessary; in most cases, the programming language supporting the multiple inheritance defines this.

5.1.16 Terminology

The following is the terminology used in Class Diagrams:
- Association
- Multiplicity and Navigation
- Multiplicity Design
- Aggregation
- Composition
- Dependency
- Generalization

Knowledge Check 1

1. Which of these statements about the Purpose of Class Diagram are true?

(A)	Class diagram represents static structure of the solution
(B)	Class diagram is not essential in UML modeling
(C)	Class diagram provides a code skeleton for ready use

(A)	A, B	(C)	A,C
(B)	B, C	(D)	B

2. Match the following statements about the elements of class Diagrams against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Class emphasizes relevant	(1)	3-sections
(B)	A class is represented using	(2)	Attributes
(C)	Class representation comprises of	(3)	Character
(D)	Any information the class itself maintains is done through its	(4)	The object

(E)	The outcome of the operation depends on the current state of	(5)	Compartmentalized rectangle

(A)	A-3, B-4, C-1, D-5, E-2	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-3, B-5, C-1, D-2, E-4

3. Match the following statements about the Relationships in class Diagrams against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	In UML, relationships are modeled as	(1)	Two classifiers
(B)	Association represents structural relationships between objects of	(2)	Lines
(C)	Realization is a semantic relationship between	(3)	Association line
(D)	An association is a bi-directional connection between	(4)	Different classes
(E)	The role name is placed next to the end of the	(5)	Classes

(A)	A-3, B-4, C-1, D-5, E-2	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-3, B-5, C-1, D-2, E-4

4. Match the following statements about the Relationships in class Diagrams against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Associations & Aggregations are	(1)	Role
(B)	Multiplicity is indicated by a text expression on the	(2)	An association
(C)	Multiplicity is the number of instances that participate in	(3)	Bi-directional
(D)	An arrowhead with a hallow diamond indicates	(4)	“is-a”
(E)	Phrase that best represents a generalization relationship is	(5)	Aggregation relationship

(A)	A-3, B-1, C-2, D-5, E-4	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-3, B-5, C-1, D-2, E-4

5.2 Other Modeling Elements

In this second lesson, **Other Modeling Elements**, you will learn to:

- Define and describe Interfaces in Class Diagrams
- Define and describe Stereotypes in Class Diagrams

- Define and describe Constraints in Class Diagrams

5.2.1 Interfaces

An interface is "a collection of operations that are used to specify a service of a class or component". Interfaces formalize polymorphism. The Greek term polymorphous means "having many forms". Every implementation of the interface must implement at least the interface. The implementation can, in some cases, implement more than the interface.

Interfaces allow people to define polymorphism in a declarative way, unrelated to implementation. Two elements are polymorphic with respect to a set of behavior if they realize the same interface, for example, any two "things" that realize the same interface can be said to be polymorphic.

Polymorphism is a big benefit from the object orientation, but without interfaces, there is no way to enforce it, verify it, or even express it except in informal ways, or language-specific ways. Formalization of interfaces strip away the mystery, and provide a good way to describe the interface in testable, verifiable, and precise way.

Interfaces are the key to the "Plug-and-play-ability" of architecture: any classifiers, for example, classes, subsystems, components, which realize the same interfaces, may be substituted for one another in the system, thereby supporting the changing of implementations without affecting clients.

Figure 5.15 depicts the use of Interfaces.

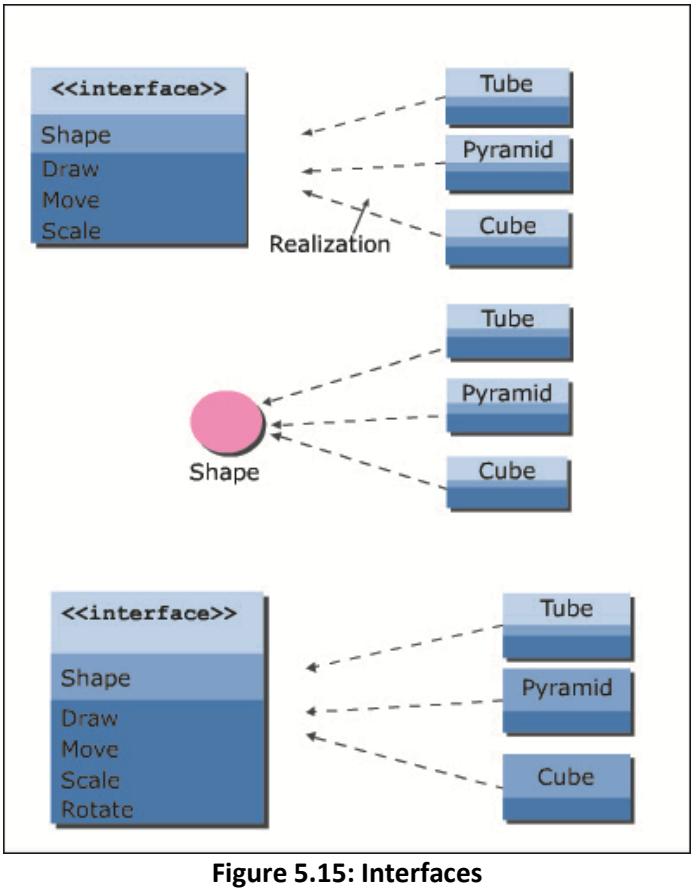


Figure 5.15: Interfaces

The lollipop notation is best used when we only need to denote the existence of an interface. If it is necessary to see the details of the interface, for example, the operations, then the class/stereotype representation is more appropriate.

- **Example**
  For example, the same remote can be used to control any type of television that supports a specific interface (the interface the remote was designed to be used with).

5.2.2 Stereotypes

Stereotypes allow one to extend the basic UML notation by allowing to define a new modeling element based on an existing modeling element. The new element may contain additional semantics, but still applies in all instances where the original element is used. In this way, the number of unique UML symbols is reduced, simplifying the overall notation.

Figure 5.16 depicts Stereotypes.

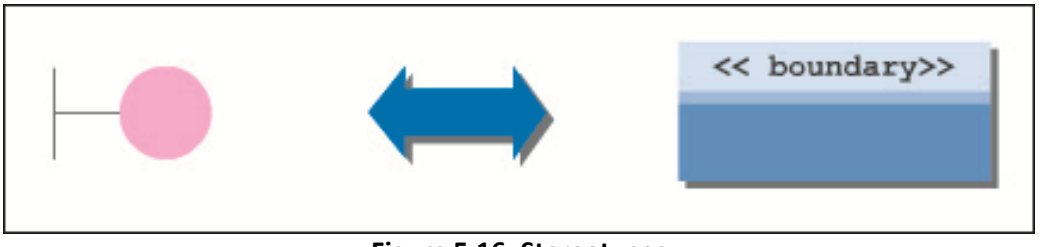


Figure 5.16: Stereotypes

Stereotypes can be applied to all modeling elements: classes, relationships, components, and so forth. Each UML element can have at the most one stereotype.

Uses of stereotypes include: modifying code generation behavior, using a different or domain specific icon or color anywhere an extension is needed or would be helpful in making a model more clear or useful.

- The name of a stereotype is shown in guillemots, for example, <<stereotype name>>. A unique icon may be defined for the stereotype and the new element may be modeled using the defined icon or the original icon with the stereotype name displayed, or both.

5.2.3 Constraints

Constraints allow for addition of new semantics, or for changing existing semantics to a UML model. Constraints are represented as strings enclosed in braces and placed near the element the constraint applies to. A dependency relationship can be used if a constraint must be attached to more than one element.

Figure 5.17 depicts Constraints.

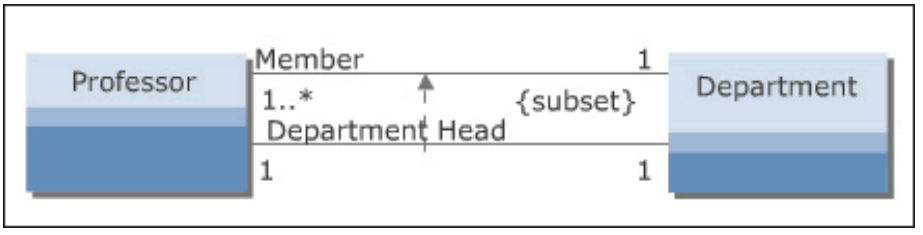


Figure 5.17: Constraints

In the example, a professor can only be the Department Head for a Department of which he is a Member.

Knowledge Check 2

1. Match the following statements about Interfaces in class Diagrams against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Interface formalizes	(1)	“plug-and-play” ability
(B)	The notation best used only to denote the existence of an interface is the	(2)	Polymorphism
(C)	Interfaces are the key to the	(3)	An interface
(D)	Polymorphism is the big benefit from the	(4)	The lollipop notation
(E)	Collection of operations that are used to specify a service of a class or component is	(5)	Object orientation

(A)	A-3, B-1, C-2, D-5, E-4	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-3, B-5, C-1, D-2, E-4

2. Match the following statements about Stereotypes in class Diagrams against their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Stereotypes allow one to extend	(1)	Guillemots
(B)	The new element may contain	(2)	Overall notation
(C)	Stereotype is shown in	(3)	Basic UML notation
(D)	Stereotypes can be applied to	(4)	Additional semantics
(E)	Reduced unique UML symbols simplifies	(5)	All modeling elements

(A)	A-3, B-1, C-2, D-5, E-4	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-3, B-4, C-1, D-5, E-2

3. Which of these statements about Constraints in Class Diagram are true?

(A)	Constraints allow for addition of new semantics
(B)	Constraints are represented as hollow diamonds
(C)	A dependency relationship can be used if a constraint must be attached to more than one elements

(A)	A, C	(C)	A, B
(B)	B, C	(D)	C

Module Summary

In this module, **Static Modeling**, you learnt about:

- **Class Diagrams**
  Class Diagrams are the most essential part of UML modeling. They represent the static structure of the solution modeled at the most detailed level. The Class Diagram provides developers with detailed information about what operation code to develop, along with information about data types, parameters, and namespaces.

- **Other Modeling Elements**
  Other Modeling Elements include Interfaces, Stereotypes, and Constraints.

MODULE 6

UML Package Diagrams

Welcome to the module, **UML Package Diagrams**. This module introduces UML Package Diagrams. It explains about the concepts of Package Diagrams, Packages and Elements and packaging strategies.

In this module, you will learn about:

- Package diagrams
- Types of Package Diagrams

6.1 Package Diagrams

In this first lesson, **Package Diagrams**, you will learn to:

- Describe the purpose and concept of package diagrams
- List and describe packaging elements
- List and describe symbols and notations
- Description of visibility and dependencies of package diagrams

Package diagrams display the organization of packages and their elements, as well as corresponding namespaces. They are typically used to depict the high-level organization of a software project.

A Package Diagram is a collection of links to other diagrams, whether it is a Class Diagram, Use Case Diagram, Sequence Diagram, or even another Package Diagram.

A package diagram can be used to

- Represent an overview of the requirements, like grouping of a collection of use-case
- Depict a high-level overview of a collection of UML Class diagrams
- Logically modularize a complex diagram

6.1.1 Packaging Elements

Package Diagrams are used to reflect the organization of packages and their elements. They can be used to represent class elements and use case elements. Elements contained in a package share the same namespace and so the elements should have unique names.

6.1.2 Symbols and Notations

- **Package**
  A package provides the same functionality of a folder in Windows operating system. They are depicted as file folders and can be applied on any UML diagram.

- **Elements and Contents**
  Elements of a package can be represented inside the rectangular portion of the package box. Contents of a package can also be represented outside the package box and linked using a (+) symbol.

Figure 6.1 depicts a Package diagram.

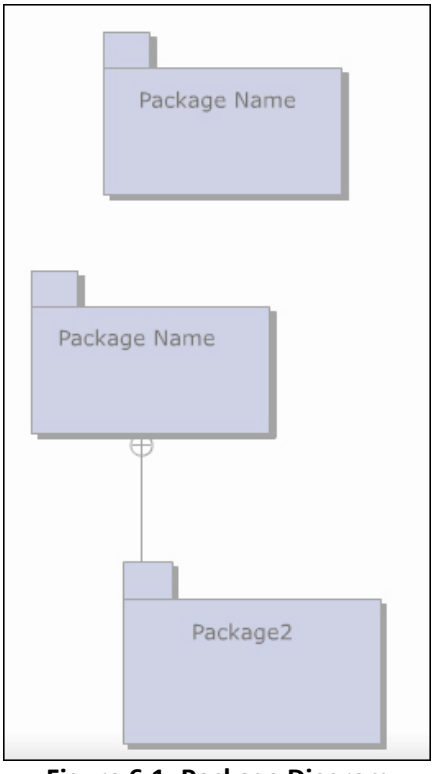


Figure 6.1: Package Diagram

6.1.3 Visibility and Dependency

Visibility may be defined for package elements in the same way it is defined for class attributes and operations. Visibility of a package element can be expressed by including a visibility symbol as a prefix to the package element name. Three types of visibilities are defined in UML:

- **Public**: Public classes can be accessed outside of the owning package and any packages that inherit from the owning package. The visibility symbol is '+'
- **Protected**: Protected classes can only be accessed by the owning package and any packages that inherit from the owning package. The visibility symbol is '#'
- **Private**: Private classes can only be accessed by classes within the owning package. The visibility symbol is '-'

Figure 6.2 depicts Visibility of Packages.

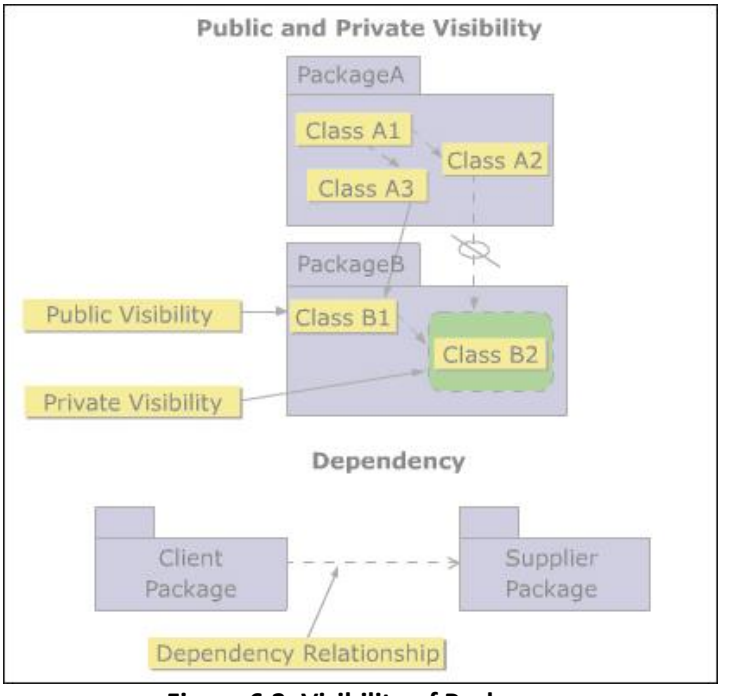


Figure 6.2: Visibility of Packages

Elements in one package can import elements in another package. In UML, this is represented as a **dependency** relationship. The relationships of the packages reflect the allowable relationships between the contained classes.

Knowledge Check 1

1. Which of the following statements about Package diagrams are true?

(A)	They depict an overview of the system requirements
(B)	They are used for grouping a collection of Use-cases
(C)	They represent the lower detailed levels of Class diagrams
(D)	They help in modularizing and visualizing a complex system
(E)	They reflect the organization of the elements within them

(A)	A, B	(C)	A,C, E
(B)	B, C	(D)	A, B, D, E

2. Match the elements with their corresponding descriptions.

	DESCRIPTION	ELEMENT	
(A)	Elements in a package	(1)	Symbol '-'
(B)	Packages and its elements are similar to	(2)	Symbol '+'

(C)	Importing elements from other packages	(3)	Unique names
(D)	Public visibility	(4)	Folders and files in an OS
(E)	Private visibility	(5)	Dependency relationship

(A)	A-3, B-4, C-1, D-5, E-2	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-3, B-4, C-5, D-2, E-1

6.2 Types of Package Diagrams

In this second lesson, **Types of Package Diagrams**, you will learn to:

- Define and describe Class Package Diagrams
- Define and describe Data Package Diagrams
- Define and describe Use Case Package Diagrams
- Explain Packaging Strategies

6.2.1 Class Package Diagrams

Classes in the same inheritance hierarchy typically belong in the same package. Also classes related to one another via composition often belong in the same package. Figure 6.3 depicts a class package diagram. It shows several packages and the dependencies between them.

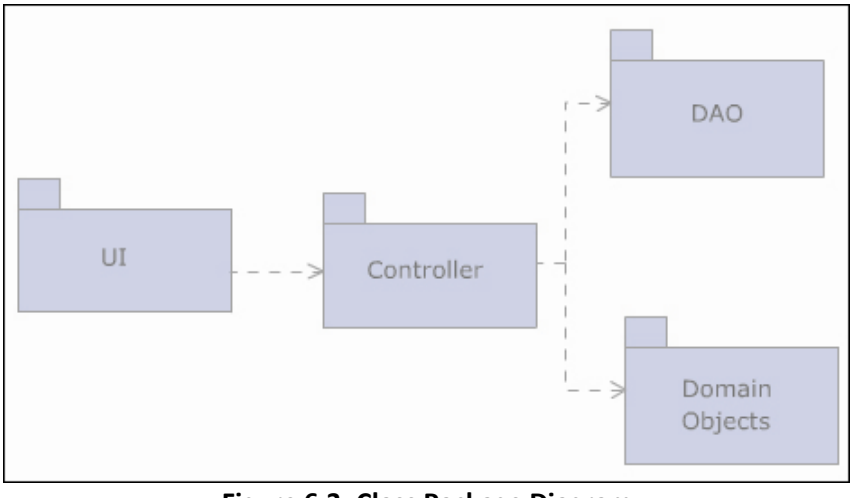


Figure 6.3: Class Package Diagram

6.2.2 Data Package Diagrams

Data Package Diagrams are used to organize data entities into large scale business domains. For example in the figure**,** if all the Packages except Domain Objects Package are removed it will lead to UML data models instead of UML class models. Tables, entities, and views are all modeled using rectangular class boxes with the appropriate stereotypes.

6.2.3 Use Case Package Diagrams

Packages can be used in the Use-Case Model to reflect order, configuration, or delivery units in the finished system. One can use Use-Case packages to structure the Use-Case model in a way that reflects the user types. In UML, a package is represented as a tabbed folder.

Figure 6.4 depicts a Use Case Package diagram.

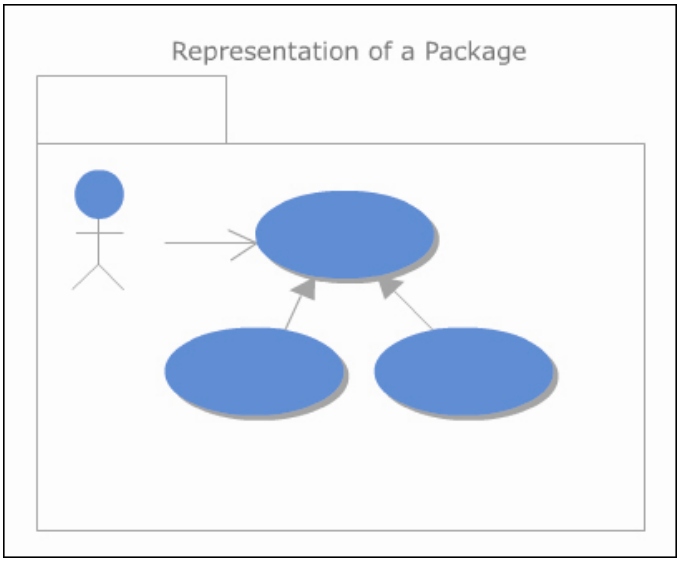


Figure 6.4: Use Case Package Diagram

6.2.4 Packaging Strategies

A well-structured architecture generally has the following characteristics:

- It encompasses a set of classes, typically organized into multiple hierarchies
- It provides a set of collaborations that specify how these classes cooperate to provide various system functions

A good packaging mechanism needs to implement the following mechanisms -

- Packages should be cohesive
- Names of the packages should be short and descriptive
- Package description must match with the responsibilities of contained classes
- Package dependencies have to correspond to the relationships between the contained classes
- Classes contained in a package must be there according to the criteria for the package division
- The ratio between the number of packages and the number of classes should be appropriate

Knowledge Check 2

1. Which of the following statements about package diagrams are true?

(A)	Classes of the same inheritance hierarchy are grouped in the same package
(B)	The Use-Case model makes use of packages to show the configuration and delivery units in the system
(C)	Use-Case packages cannot be used to categorize user types
(D)	Tables and Views are modelled in Data Package Diagrams using rectangular boxes
(E)	Packages should be as modular as possible

(A)	A, B, C	(C)	A,C, D
(B)	A, B, D, E	(D)	B, D, E

Module Summary

In this module, **UML Package Diagrams**, you learnt about:

- **Package Diagrams**
  A package is a general-purpose mechanism for organizing elements into groups. A package diagram is a UML diagram composed of packages. It contains the dependencies between them.

- **Types of Package Diagrams**
  Types of Package Diagrams include Class Package Diagrams, Data Package Diagrams, and Use-Case Package Diagrams.

ASK to LEARN

Onlinevarsity
your e-way to learning

Questions *in your* mind?

are here to HELP

Post your queries in ASK to LEARN @

www.onlinevarsity.com

MODULE 7

Dynamic Modeling

Welcome to the module, **Dynamic Modeling**. This module introduces Object Oriented Analysis & Design and explains about the concepts of Object Orientation.

In this module, you will learn about:

- Interaction Diagrams
- Sequence Diagrams
- Collaboration Diagrams

7.1 Interaction Diagrams

In this first lesson, **Interaction Diagrams**, you will learn to:

- Understand the purpose and concept of interaction diagrams
- List and describe types of interaction diagrams

UML interaction diagrams are a good way to depict dynamic models and compare them to the static models that must support them.

Interaction diagrams describe the communication between objects and how a group of objects collaborates in some behavior.

In UML 2.0, interaction diagrams comprise the following types of diagrams.

- Sequence Diagrams
- Collaboration Diagrams (or) Communication diagrams
- Interaction Overview diagrams
- Timing diagrams

We will now look into the first two types in detail.

Knowledge Check 1

1. Which of the following statements regarding components of Interaction diagrams are true?

(A)	Activity diagram is an interaction diagram
(B)	Each dynamic model brings out a different facet of an use case
(C)	Interaction diagrams describe the behavior and communication between objects
(D)	Interaction diagrams are used in dynamic modeling

(A)	A, B	(C)	A,C
(B)	B, C	(D)	A, D

2. Which of the following are dynamic modeling techniques?

(A)	Activity Diagrams
(B)	Communication Diagrams
(C)	Timing Diagrams
(D)	Interaction Diagrams

(A)	A, B	(C)	A,C
(B)	B, C	(D)	A, B, C, D

7.2 Sequence Diagrams

In this second lesson, **Sequence Diagrams**, you will learn to:

- Describe the purpose and concept of sequence diagrams
- List and describe the elements of sequence diagrams
- Define and describe conditions and controls
- Define and describe interpretation

UML sequence diagrams are one of the popular UML artifacts for dynamic modeling, which focus on identifying the behavior within your system.

Figure 7.1 depicts a Sequence diagram.

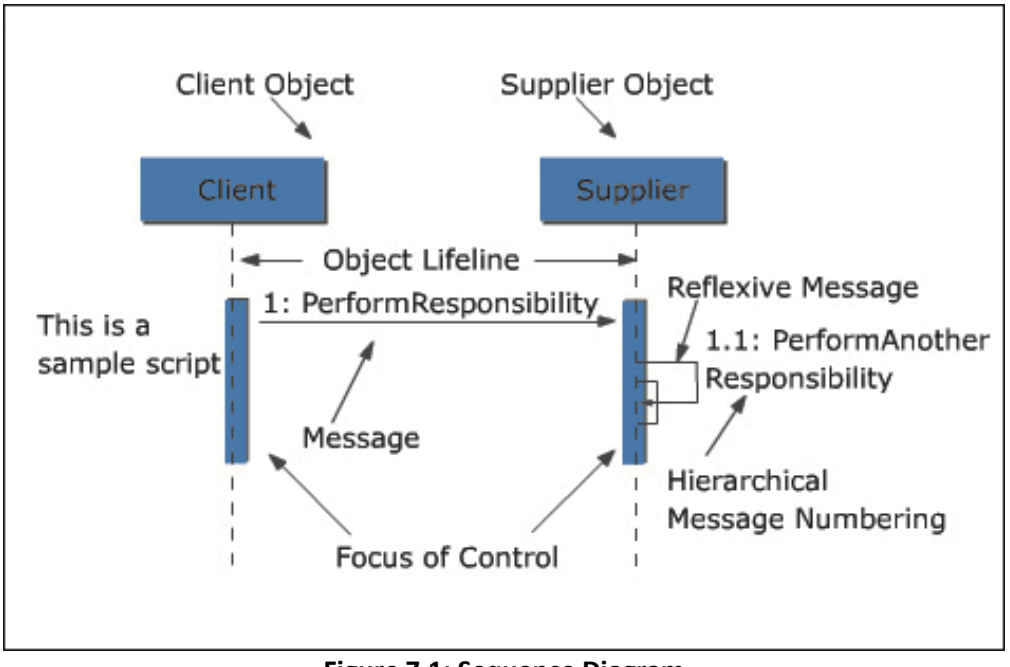


Figure 7.1: Sequence Diagram

Sequence diagrams model the flow of logic within the system in a visual manner. They describe a pattern of interaction and the messages objects send. Sequence diagrams are commonly used for both analysis and design purposes.

Sequence diagrams are drawn and read from left-to-right and return values from right-to-left, although that does not always work with complex objects/classes.

7.2.1 Elements of sequence diagrams

- **Object**
  An object symbol is drawn at the head of the lifeline, and shows the name of the object and its class underlined and separated by a colon.

- **Message**
  A message is a communication between objects that conveys information with the expectation that activity will ensue. A message is shown as a horizontal solid arrow from the lifeline of one object to the lifeline of another object. For a reflexive message, the arrow starts and finishes on the same lifeline. The arrow is labeled with the name of the message and its parameters. The arrow may be labeled with a sequence number.

- **Lifeline**
  A vertical dotted line identifies the existence of the object over time. The lifeline represents the existence of the object at a particular time.

- **Activation**
  Rectangular blocks of time that indicate when a certain object is active. It can be important to consider the length of time it takes to perform actions. It is also called **Focus of Control.** It represents the relative time that the flow of control is focused on an object, thereby representing the time an object is directing messages. The focus of control is shown as narrow rectangle on object lifelines. A self-message can represent a recursive call of an operation, or one method calling another method belonging to the same object. It is shown as creating a nested focus of control in the lifeline's execution occurrence.

- **Actor**
  Same as a use-case actor.

- **Hierarchical numbering**
  This bases all messages on a dependent message. The **dependent message** is the message whose focus of control is the other messages that originate within. For example, Message 1.1 is dependent on Message 1, and the **Scripts** describe the flow of events textually.

Figure 7.2 depicts elements of a Sequence diagram.

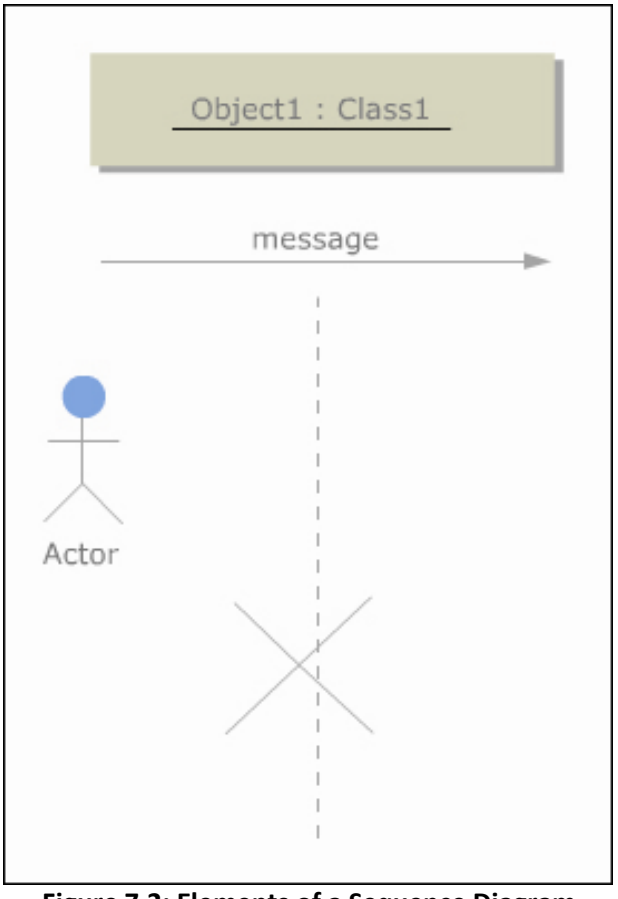


Figure 7.2: Elements of a Sequence Diagram

Sequence diagrams provide ways to show conditions and looping control mechanisms but it is not a very easy task as it makes visualization of objects complex. The notations for conditionals and control structures are known as interaction frames.

There are a number of mechanisms that do allow for adding a degree of procedural logic to diagrams and which come under the heading of combined fragments. A combined fragment is one or more processing sequence enclosed in a frame and executed under specific named circumstances.

Some of the fragments available are as follows:

- Alternative fragment (denoted “alt”) models if…then…else constructs
- Option fragment (denoted “opt”) models switch constructs
- Loop fragment (denoted “loop”) models execution multiple times and the “guard”(denoted inside […]) indicates the basis for looping

Figure 7.3 depicts loops in a Sequence diagram.

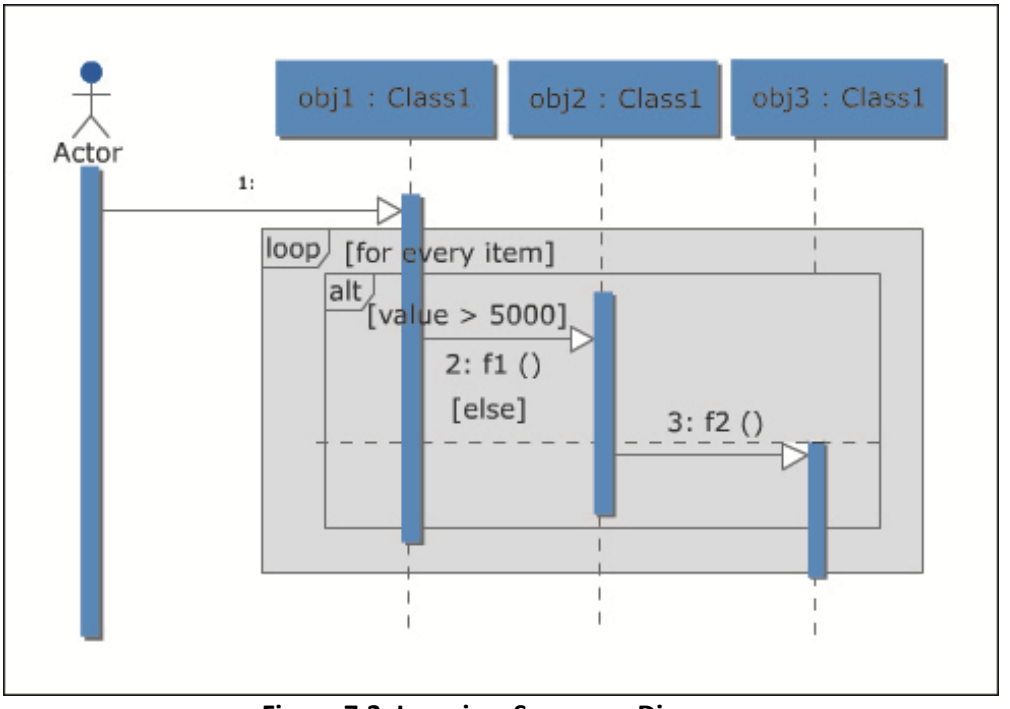


Figure 7.3: Loop in a Sequence Diagram

In it a loop is opened for every item and if the value is greater than 5000 operation f1 is called on obj2, else f2 is called on obj3.

7.2.2 Interpretation

Developers who need to interpret this diagram and use it to construct code must have the information provided in the notes to do so effectively. Notes are a great aid in helping us understand more fully the details associated with this sequence of events.

Many of the elements of the UML have a precise mapping to the Java programming language. As developers, we must interpret each of these elements in a fashion that is faithful to this claim. Different interpretations of these elements can result in miscommunication, which is the very challenge the UML attempts to resolve.

Knowledge Check 2

1. Match the following descriptions with the corresponding elements.

Description		Element	
(A)	Life line	(1)	Narrow rectangle
(B)	Focus of Control	(2)	Vertical dotted line
(C)	Object	(3)	Horizontal solid arrow
(D)	Message	(4)	Rectangular box with Object name : Class name

(A)	A-3, B-4, C-1, D-2	(C)	A-4, B-1, C-2, D-3
(B)	A-2, B-1, C-4, D-3	(D)	A-3, B-4, C-1, D-2

2. Match the following descriptions with the corresponding elements.

Description		Element	
(A)	Sequence Diagrams	(1)	Show 'switch' constructs
(B)	Notation for conditional constructs	(2)	Show 'if … else' constructs
(C)	'Alt' Fragments	(3)	Show looping controls
(D)	'Opt' Fragments	(4)	Use square brackets to indicate 'guards'
(E)	'Loop' Fragments	(5)	Interaction frames

(A)	A-3, B-4, C-1, D-5, E-2	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-5, B-4, C-2, D-1, E-3

7.3 Collaboration Diagrams

In this third lesson, **Collaboration Diagrams,** you will learn to:

- Describe the purpose and concept of collaboration diagrams
- List and describe elements of collaboration diagrams

7.3.1 Purpose and definition of Collaboration Diagrams

A collaboration diagram (**renamed as Communication diagram in UML 2.0)** describes a pattern of interaction among objects; it shows the objects participating in the interaction and the messages that they send to each other. It shows an interaction organized around the objects and their links to each other.

A collaboration diagram is a graph of references to objects and links with message flows attached to its links. The diagram shows the objects relevant to the performance of an operation, including objects indirectly affected or accessed during the operation.

Figure 7.4 depicts a Collaboration diagram.

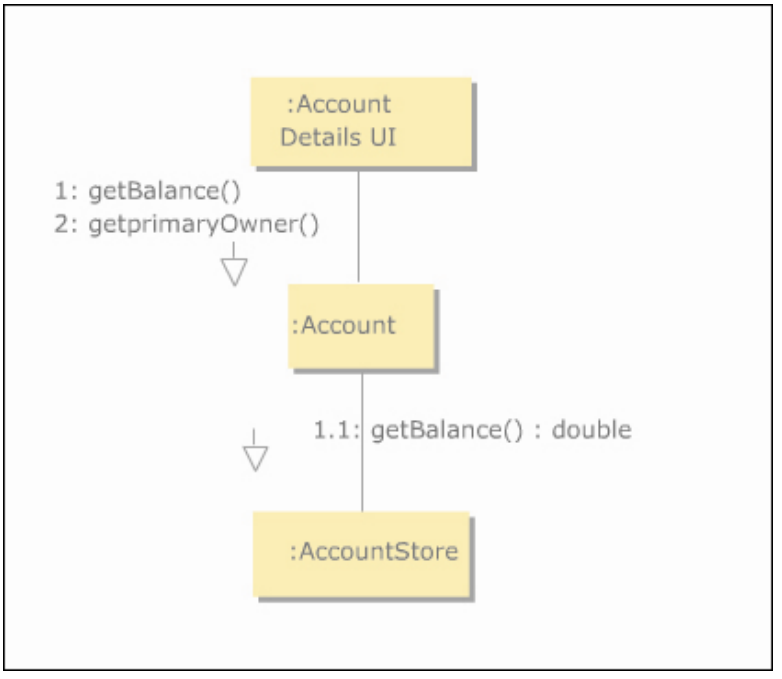


Figure 7.4: Collaboration Diagram

7.3.2 Elements of Collaboration Diagrams

The following are the elements of sequence diagrams.

- **Object**
  An object represented in a normal fashion like sequence diagram.

- **Link**
  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. A link can be an instance of an association, or it can be anonymous, meaning that its association is unspecified.

- **Message**
  A **message** is a communication between objects that conveys information with the expectation that activity will ensue in collaboration diagrams, because collaboration diagrams are the only way of describing the relative sequencing of messages. A message can be unassigned, meaning that its name is a temporary string that describes the overall meaning of the message. The message can later be assigned by specifying the operation of the message's destination object. The specified operation will then replace the name of the message.

- **Hierarchical numbering**
  The numbering style in collaboration diagram is fairly straightforward.

  The internal messages that implement a method are numbered starting with number 1. For a procedural flow of control, the subsequent message numbers are nested in accordance with call nesting. For a

nonprocedural sequence of messages exchanged among concurrent objects all the sequence numbers are at the same level (that is, they are not nested).

Figure 7.5 depicts elements of a Collaboration diagram.

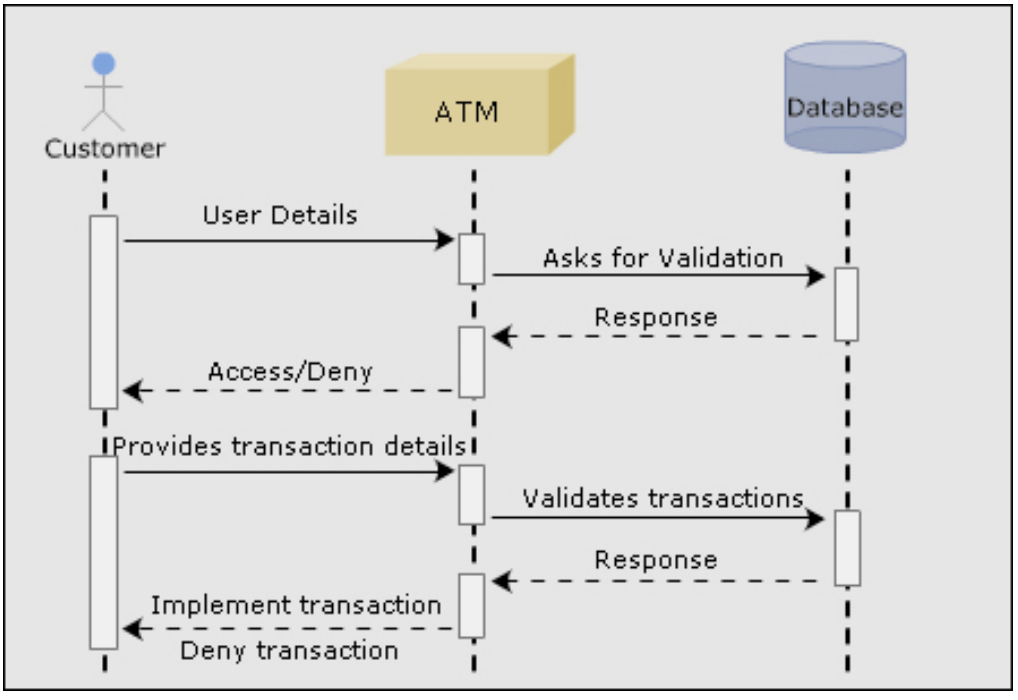


Figure 7.5: Elements of a Collaboration Diagram

Knowledge Check 3

1. Which of the following is NOT true about Collaboration Diagrams?

(A)	It describes a pattern of interaction among objects
(B)	It shows the sequence of messages using the time dimension
(C)	It shows the relationships of objects
(D)	It emphasises on the structural collaboration of the system

Module Summary

In this module, **Dynamic Modeling**, you learnt about:

- **Interaction Diagrams**
  Interaction diagrams are made of sequence diagrams and communication or collaboration diagrams. They are used to verify whether there is an existing object that can perform the behavior or not. Only when there is no existing object that can perform the behavior, should the new classes be created.

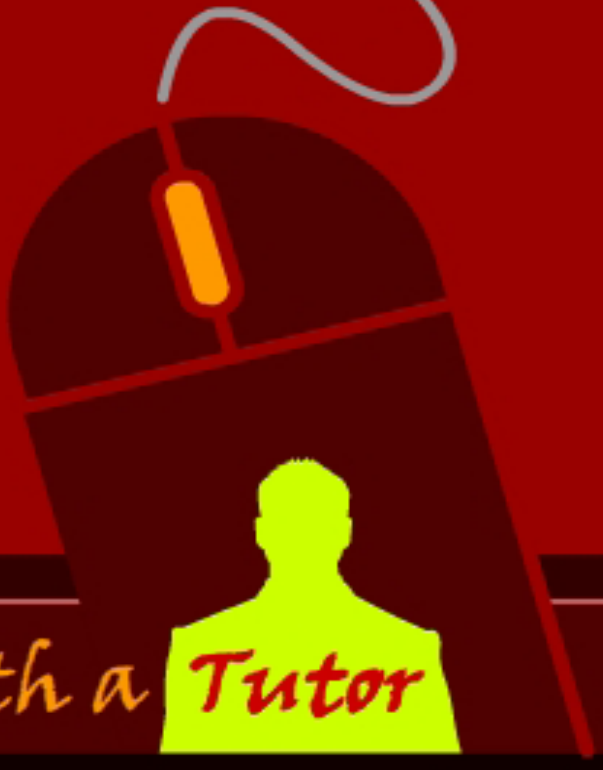
- **Sequence Diagrams**
  Sequence diagrams describe a pattern of interaction and the messages objects send. They specify the activation time and lifeline of the objects. Looping mechanisms and other fragments like alternate behavior can be represented using sequence diagrams.

- **Collaboration Diagrams**
  Collaboration diagrams show the objects participating in the interaction and the messages that they send to each other.

Tutor Chat

Onlinevarsity
your e-way to learning

are just a click away



To chat with a **Tutor**

login to **www.onlinevarsity.com**

MODULE 8

State Machine Diagrams

Welcome to the module, **State Machine Diagrams**. This module introduces State Machine Diagrams, Notations used, and advanced concepts in State Machine Diagrams.

In this module, you will learn about:

- State Machine Diagrams
- More on State Machine Diagrams

8.1 State Machine Diagrams

In this first lesson, **State Machine Diagrams,** you will learn to:

- Describe the purpose and concept of State Machine Diagrams
- List and describe notations for states

The behavior of an entity is not only a direct consequence of its input, but it also depends on its preceding state. The history of an entity can best be modelled by a finite state diagram. A State Machine diagram can show the different states of an entity and also how an entity responds to various events by changing from one state to another.

Figure 8.1 shows an example of a state machine diagram.

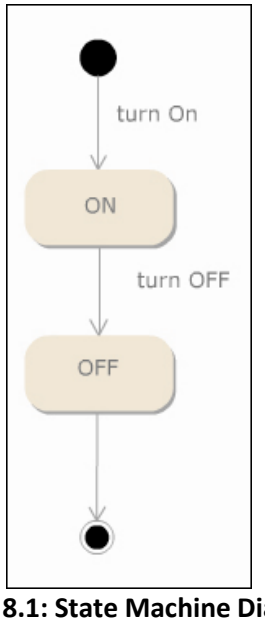


Figure 8.1: State Machine Diagram

8.1.1 Notations for States

A state is a stage in the behavior pattern of an entity. States are represented by the values of the attributes of an entity.

Figure 8.2 depicts notations for states.

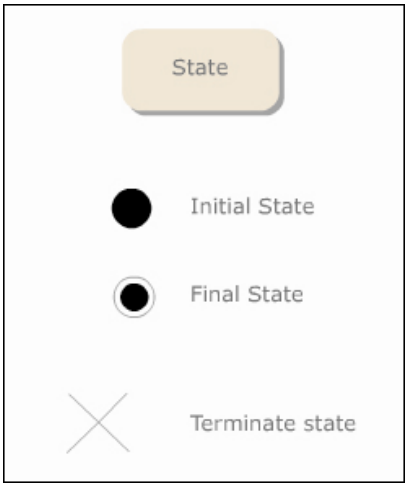


Figure 8.2: Notations for States

A State is displayed as a rounded rectangle and is usually named according to its condition.
Start and end nodes are represented as solid and empty circles and are used to represent the beginning and end of all transitions.

An abrupt termination state can be represented using terminate state.

Knowledge Check 1

1. Match the following functions with the corresponding menu.

Functions		Menu	
(A)	Close:[if not available]/error	(1)	Rounded Rectangle
(B)	States are represented by	(2)	Attribute values
(C)	State machine diagrams show	(3)	Transition and guard
(D)	Notation for state	(4)	Guard
(E)	Traversing condition for transition is represented by	(5)	States of entities

(A)	A-3, B-2, C-5, D-1, E-4	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-5, B-3, C-1, D-2, E-4

8.2 More on State Machine

In this second lesson, **More on State Machine,** you will learn to:

- Define and describe transitions and guards
- Define and describe registers and actions
- Define and describe internal transitions
- Define and describe superstates, substates, and concurrent states

8.2.1 Transitions and Guards

A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the entity being modeled.

Figure 8.3 depicts the use of Guards.

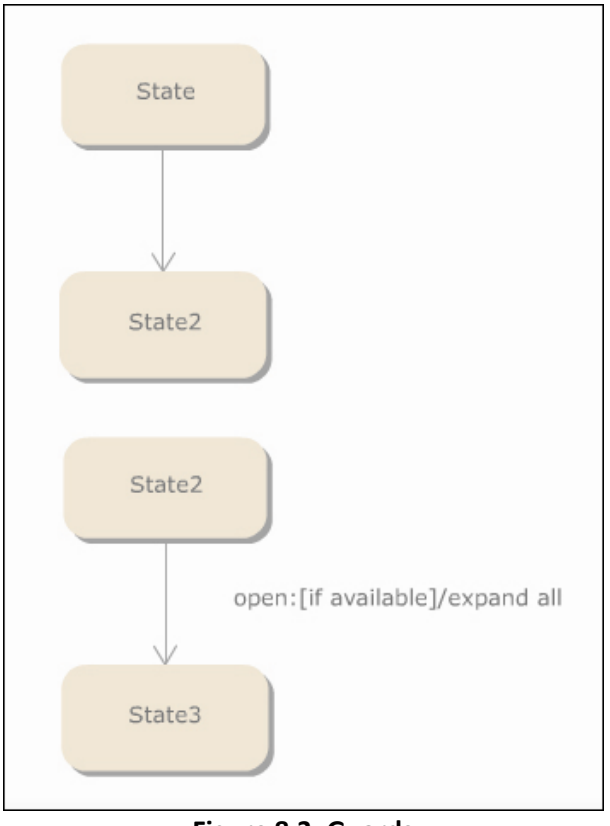


Figure 8.3: Guards

A guard is a condition that must be true in order to traverse a transition. The notation for the labels on transitions is in the format **event:[guard]/activity**.

8.2.2 Registers and Actions

An action is an operation that is invoked by/on the entity being modeled. Registers and actions are not available in State Machine Diagrams.

8.2.3 Internal Transitions

Internal transitions also known as internal activities show the case where states react to events without transition.

They are represented by putting the event and guard inside the state box itself.
They are different from the self-transition in that the internal activities do not trigger the entry and exit activities.

Figure 8.4 depicts Internal Transitions.

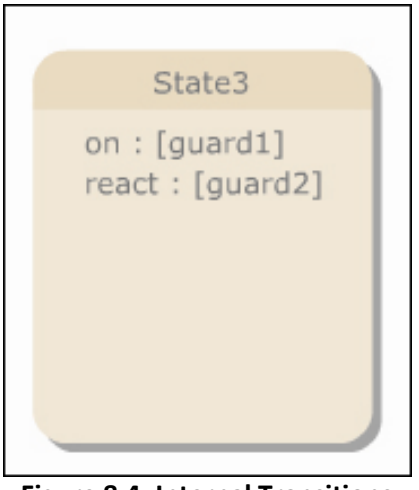


Figure 8.4: Internal Transitions

8.2.4 Superstates, Substates, and Concurrent State Diagrams

A complicated state can be decomposed into several substates for better understanding of its behavior. A substate is a state that is nested in another state.
A state that has substates is called a Superstate.

States are also broken into several (sub-) state diagrams that run concurrently, which are called concurrent states.

Figure 8.5 depicts different types of State Machine diagrams.

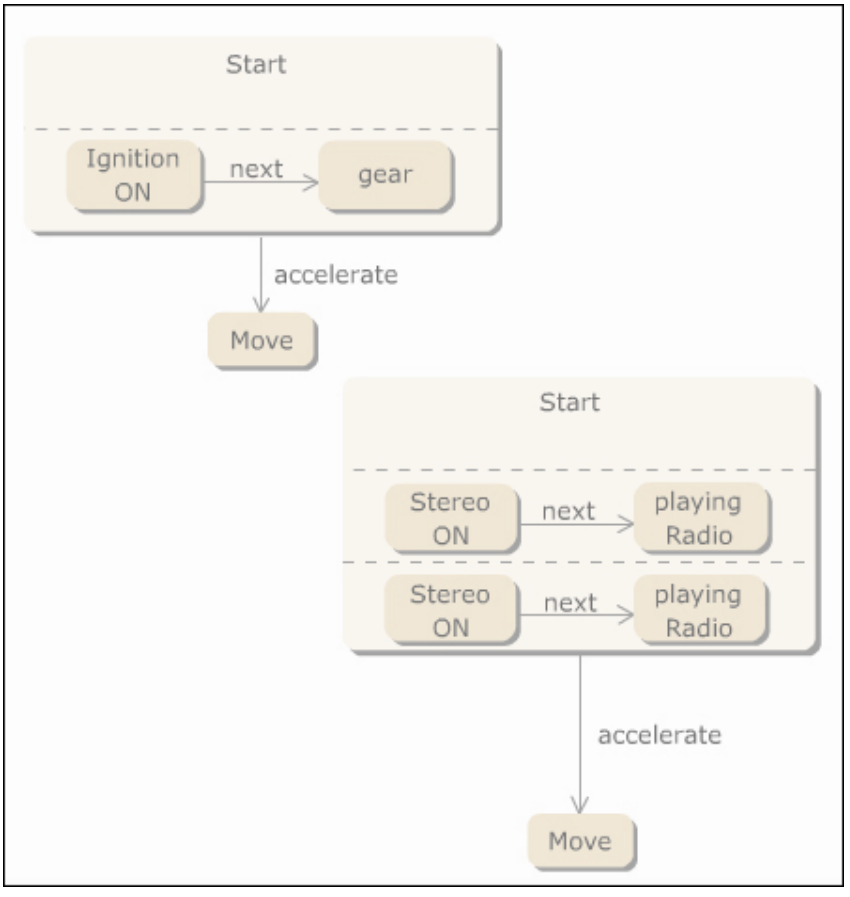


Figure 8.5: Different Types of State Machine Diagrams

Knowledge Check 2

1. Match the elements on right side with the descriptions on the left side.

DESCRIPTION		ELEMENT	
(A)	Internal Transitions	(1)	Decomposed state
(B)	Superstate of an object	(2)	Parallel states
(C)	Substate of an object	(3)	Aggregated states of an object
(D)	Concurrent states of an entity	(4)	Elements of State Machine diagrams
(E)	States and Transitions	(5)	Activities within a State

(A)	A-3, B-4, C-1, D-5, E-2	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-5, B-3, C-1, D-2, E-4

Module Summary

In this module, **State Machine Diagrams**, you learnt about:

- **State Machine Diagrams**
  State Machine Diagrams are used to understand complex classes better, particularly those that act in different manners depending on their state. State machine diagrams show states that react to events with transitions.

- **More On State Machine Diagrams**
  Transitions for a class are typically the result of the invocation of an operation that causes an important change in state. Registers and actions are not available in state machine diagrams. Internal transitions show the case where states react to events without transition. Superstates show aggregated states whereas substates are used for state decomposition.

MODULE 9

Object Design

Welcome to the module, **Object Design**. This module introduces Object Diagrams and explains about the representation of Class Diagrams.

In this module, you will learn about:

- Object diagrams
- Representation of Class Diagrams

9.1 Object Diagram

In this first lesson, **Object Diagram**, you will learn to:

- Describe the purpose and concept of object diagrams
- List and describe usage scenarios of Object diagrams
- List and describe elements of object diagrams
- List and describe usage guidelines

9.1.1 Purpose and Definition

Object diagrams, also known as instance diagrams, are useful for representing “real world” objects and the relationships between them. They use a subset of the elements of a class diagram in order to emphasize the relationship between instances of classes.

9.1.2 Usage Scenarios of Object Diagrams

Typically, object diagrams are drawn to explore the relationships between various objects. It is used to describe the system at a particular point in time.

Figure 9.1 depicts usage scenarios of Object diagrams.

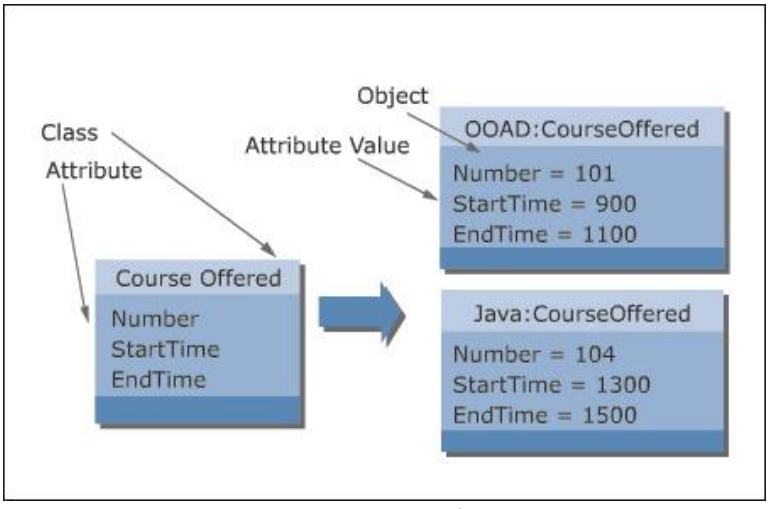


Figure 9.1: Usage Scenarios of Object Diagrams

9.1.3 Elements of an Object Diagram

An object is a concept, abstraction, or thing with sharp boundaries and meaning for an application. An object is something that possesses State and Behavior.

- **Representation of an Object:**
  Objects are identified by placing the instance name followed by a colon (:) in front of the class name. Object names are underlined and may show the name of the classifier from which the object is instantiated. The icon for an object is a rectangle divided into sections. Instance names are underlined in UML diagrams.

  Figure 9.2 shows representation of an Object.

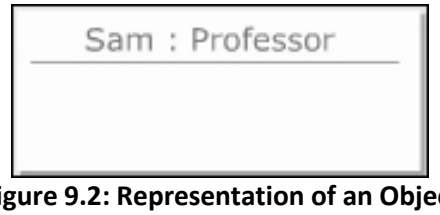


Figure 9.2: Representation of an Object

- **Attributes:**
  The state of an object is one of the possible conditions in which an object may exist, and it normally changes over time. The state of an object is usually implemented by a set of properties (called attributes), with the values of the properties, plus the links the object may have with other objects.

Figure 9.3 shows representation of attributes.

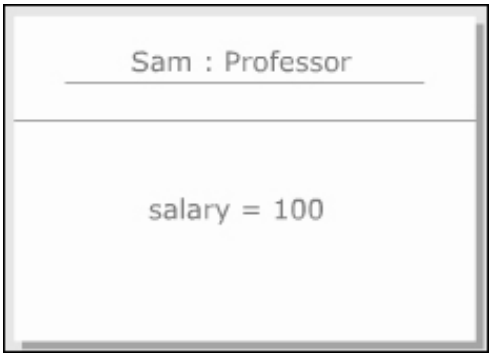


Figure 9.3: Attributes

9.1.4 Guidelines for usage of Object Diagram

There are few points to be remembered while using object diagrams in UML.

- Object diagrams can be used to check whether the system has been designed as per the requirements
- Complexity should be reduced by avoiding representation of all the objects in the system
- Always represent the state of objects in certain important flows in our application using an object diagram
- Object diagrams can be used to show how the system behaves for the business functionality needs
- Object diagram can be used as a means of debugging the functionality of the system
- Object diagrams show relationships between objects as normal associations. Relationship details and constraints are detailed in Class diagrams

Knowledge Check 1

1. Which of the following statements about Object Diagrams are true?

(A)	Object Diagrams are also known as Instance Diagrams
(B)	Object diagrams describe the system at many points in time
(C)	An object is something that does not possesses State and Behavior
(D)	Complexity should be reduced by avoiding representation of all the objects in the system

(A)	A, B	(C)	A, C
(B)	B, C	(D)	A, D

2. Which of the following characteristics of object diagram are true?

(A)	An Object diagram is an instance of a class diagram
(B)	Object diagrams do not reflect the roles played by objects in the system
(C)	Object diagrams explore relationships between objects
(D)	Object diagrams show the operations for all objects of a class

(A)	A, B	(C)	A, C
(B)	B, C	(D)	A, D

9.2 Representation of Class Diagrams

In this second lesson, **Representation of Class Diagrams**, you will learn to:

- Explain the purpose and concept of instantiations
- Explain the purpose and concept of relationships
- Explain the purpose and concept of recursive relationships

9.2.1 Class Diagrams

A class represents a prototype and an object indicates an instance of the class. Instantiation is the process of creating objects from a class. Object diagrams are also called instance diagrams for this purpose.

Object diagrams show the instances of the classes and various relationships between them. A relationship between objects is termed as linkage.

Figure 9.4 depicts a Class diagram and an Object diagram.

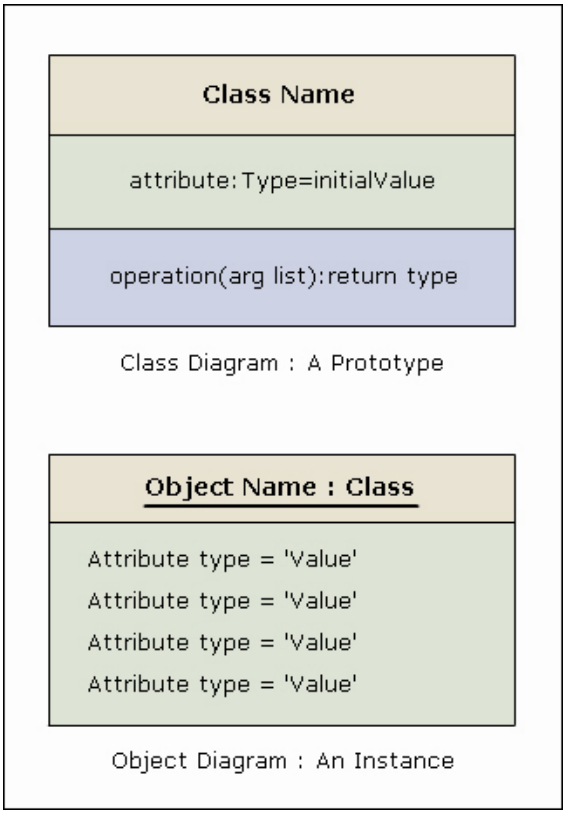


Figure 9.4: Class Diagram and Object Diagram

9.2.2 Relationships of Class Diagrams

Objects participate in relationships with other objects. Class diagrams represent different types of associations on classes. These relationships are represented as links between objects in object diagrams. Links are said to be instances of associations like objects that are instances of classes.
Associations between Objects are simply diagrammed using a line joining the two.

Figures 9.5 and 9.6 depict the relationships and associations between objects.

Figure 9.5: Associations between Objects

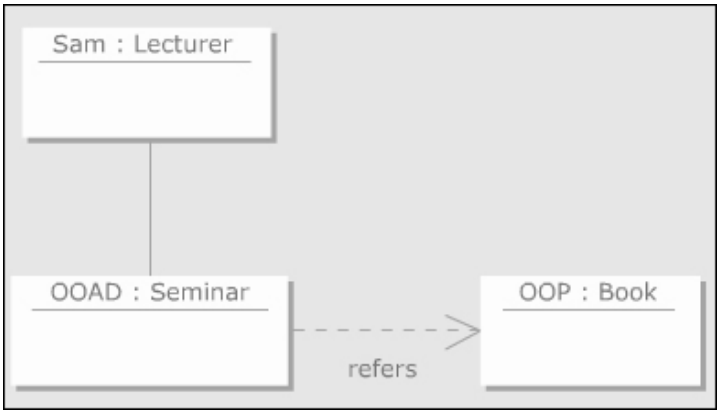


Figure 9.6: Relationships between Objects

9.2.3 Recursive Relationships of Class Diagrams

Object diagrams are useful in explaining complicated relationships like recursive relationships. Recursive relationships indicate objects with relationships on themselves.

Figure 9.7 depicts recursive relationships.

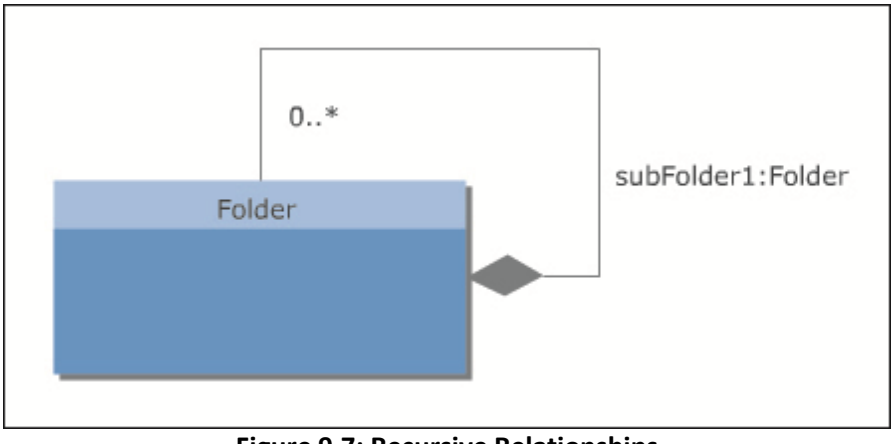


Figure 9.7: Recursive Relationships

Knowledge Check 2

1. Which of the following statements about the representation of class diagrams are true?

(A)	Recursive relationships are not depicted using object diagrams
(B)	Object diagrams can represent only 2 instances of a class at the most
(C)	Relationships between objects are called linkage in object diagrams
(D)	Object diagrams are also known as instance diagrams

(A)	A, B	(C)	A, C
(B)	B, C	(D)	C, D

2. Match the following descriptions with the corresponding elements.

	DESCRIPTION		ELEMENT
(A)	Instantiation is	(1)	A recursive Relationship
(B)	Object diagrams show relationships between objects. This is called	(2)	Links
(c)	Instances of associations between objects in an object diagram are called	(3)	Creating object
(D)	Association between objects are represented in object diagrams using	(4)	Linkage
(E)	Objects with relationships on themselves are said to have	(5)	A line

(A)	A-3, B-4, C-2, D-5, E-1	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-5, B-3, C-1, D-2, E-4

Module Summary

In this module, **Object Design**, you learnt about:

- **Object Diagrams**
  UML 2 object diagrams, also known as instance diagrams, are useful for representing "real world" objects and the relationships between them. Object Diagrams are useful in understanding Class Diagrams. They do not show anything architecturally different to class diagrams, but reflect multiplicity and roles.

- **Class Diagrams**
  Class Diagrams show various classes and associations between them. Object diagrams show the instances of the classes and various relationships between them. An Object Diagram may be considered a special case of a class diagram.

MODULE 10

System Design

Welcome to the module**, System Design**. This module defines the purpose and elements of System Design. It then moves on to explain System Design Activities

In this module, you will learn about:

- System Design
- System Design Activities

10.1 Introduction to System Design

In this first lesson**, Introduction to System Design**, you will learn to:

- Explain the purpose and concept of system design
- List and describe elements of system design

10.1.1 Purpose and Definition of System Design

System analysis and design are the most important phases in the software development life cycle. Much of the success of the project depends on how well these phases have been handled and implemented.

System design is about managing the complexity of a system efficiently and effectively using a large model with detailing in smaller models within.

It focuses on understanding the solution and performance. System design is close to the real coding of the system. It describes the object lifecycles. It can span multiple technologies and often involves multiple sub-disciplines.
Software specifications tend to be fluid, and change rapidly and often, usually while the design process is still going on.

10.1.2 Qualities of a good System Design

A good System design should
- Support a comprehensive modeling of a large and complex system
- Show comprehensive subsystems
- Result in precise specifications
- Be simple and easy to understand
- Be unambiguous

10.1.3 Elements of system design

The following are the key elements in System Design.

- Architecture
- Framework
- Patterns
- Layers
- Subsystems
- Concurrency

Knowledge Check 1

1. Which of the following statements are true?

(A)	Detailed specifications are not important for designing a system
(B)	System design should be unambiguous
(C)	Layering shows only the overall purpose of a system
(D)	Deciding the architecture will have no impact on the designing the system

(A)	A, B	(C)	A, C
(B)	B	(D)	A, D

10.2 System Design Activities

In this second lesson, **System Design Activities,** you will learn to:

- Explain the process of architectural and framework selection
- Explain the process of subsystem breakdown
- Explain the process of system design activities
- Explain the process of layering and partitioning
- Explain the use of concurrency

10.2.1 Rational Unified Process

Popularly called 'The Three Amigos' the stalwarts of OOAD, Grady Booch, Ivar Jacobson and James Rumbaugh, after creating UML as a single complete notation for describing object models, turned their efforts to the development process.

They came up with the Rational Unified Process (RUP).

RUP is a general framework used to describe specific development processes.

10.2.2 Framework

A **framework** is a skeletal solution to a particular problem devoid of many of the details. These details may be filled in by applying various analysis and design patterns.

Architectural frameworks provide the context and infrastructure in which the components exist and function.

These frameworks may provide:

- Communication mechanisms
- Distribution mechanisms
- Error processing capabilities
- Transaction support

Frameworks may be used for describing any level of solution. They could describe a fragment of an application or a complete customized solution like SAP, Peoplesoft, SanFransisco, Infinity, etc. SAP can be considered a customized framework for manufacturing and finance.

10.2.3 Software Architecture

Software architecture is the set of strategic decisions about the **organization** of the software system. It directly impacts the design and construction of the system.

It comprises of the system's structural and behavioral elements and their interfaces.

Architectural Analysis and Architectural Design are the deciding factors for the system design.

The 4+1 View Model describes software architecture using 5 concurrent views. Each view addresses a specific set of concerns.

The main views are as follows:

- Logical View
- Process View
- Physical/ Implementation View
- Development View

The fifth view is the **Use-Case View**

The architecture is evolved from these views.

The "4+1 View" Model is a good tool for various stakeholders to specifically find what they need in the software architecture.

10.2.4 Usage of the "4+1" View Model of Software Architecture

Different stakeholders use the "4+1" View for different approaches that are relevant to their role. System engineers can start from the physical view and then move on to the process view.

End users like customers and data specialists can approach it from the logical view. Project managers and software-configuration staff members can approach it from the development view.

Figure 10.1 depicts the 4+1 View Model.

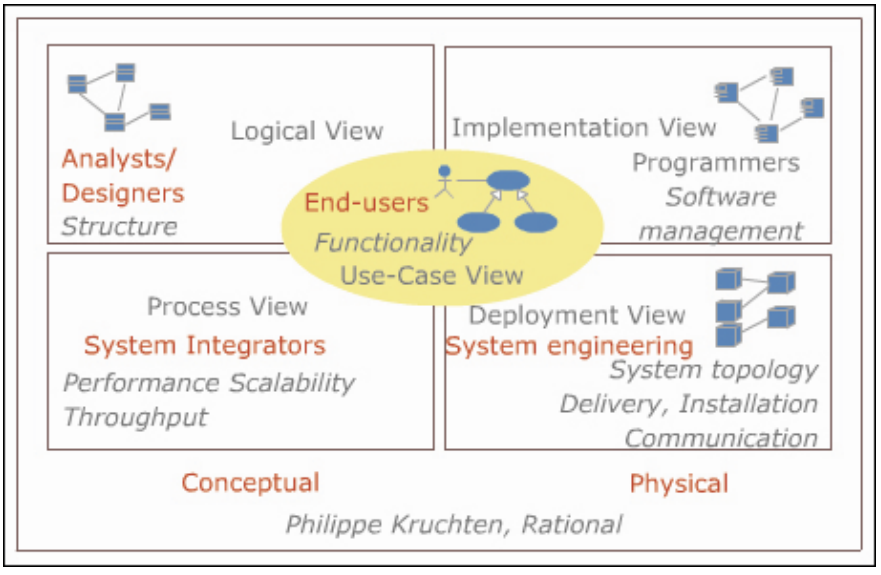


Figure 10.1: 4+1 View Model

10.2.5 Architectural Patterns

What is a pattern?

A **pattern** is a common code of specific knowledge that has been collected from experience. It is a common solution to a common problem in a specific domain. Patterns help Modeling in solving real problems efficiently. They can be reused.
Architectural patterns are schematic expressions of the basic structural organization for software systems.

They comprise predefined Subsystems, their responsibilities and guidelines for organizing the interactivity between the subsystems.

Architectural patterns imply certain characteristics of the architecture. They are as follows:

- System characteristics
- Performance characteristics
- Process characteristics
- Distribution characteristics

Each of the following commonly used patterns solves certain problems but also poses unique challenges. More than one architectural pattern can be present in a system's software architecture.

10.2.6 Common Architectural Patterns

Common Architectural patterns are as follows:

- **Layers**

  In this pattern, an application is decomposed into different levels of abstraction.

The layers range from application-specific layers at the top to implementation or technology-specific layers at the bottom.

- **Model-View-Controller (MVC)**

  In this pattern, an application is divided into three partitions:

  1. The Model - comprises the business rules and underlying data
  2. The View - shows how information is displayed to the user
  3. The Controllers - process the user input

- **Pipes and Filters**

  In this pattern, data flows in streams through pipes. The data flow line has filters. Each filter is a processing step for the data.

- **Blackboard**

  In this pattern, independent specialized applications collaborate to arrive at a solution, using a common data structure.

Each of the following commonly used patterns solves certain problems but also poses unique challenges. More than one architectural pattern can be present in a system's software architecture.

10.2.7 Subsystem

Subsystems are smaller systems within a large system. It is a model element, which has the semantics of a package. It can be a combination of a package and a class. That is, it can contain other model elements and behavior. It exhibits the object-oriented principles of encapsulation and modularity.

A subsystem realizes one or more interfaces, which define the behavior it can perform.

It may be represented as a UML package or a set of operations (behaviors). It may be used to represent the component in design.

Figure 10.2 depicts contents of a subsystem.

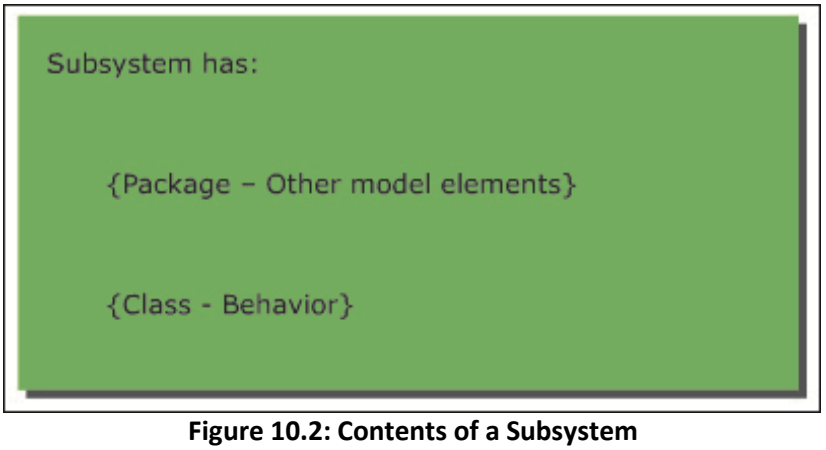


Figure 10.2: Contents of a Subsystem

10.2.8 Need for Designing Subsystems

Subsystems are designed in order to:

- Define the behaviors specified in the subsystem's interfaces in terms of collaborations of contained classes
- Document the internal structure of the subsystem
- Define realizations between the subsystems interfaces and contained classes
- Determine the dependencies upon other subsystems

10.2.9 Inputs and Outputs in Designing Subsystems

The inputs needed for Subsystem Design are as follows:

1. Use-Case Realizations
2. Design Subsystem with Interface Definitions
3. Design Guidelines, which contain detailed usage information for the architectural mechanisms

The outputs resulting from Subsystem Design are as follows:

1. Updated Use-Case Realizations
2. Updated Interface Definitions and changes in the subsystems
3. Design Classes

Figure 10.3 depicts Inputs and Outputs for a subsystem.

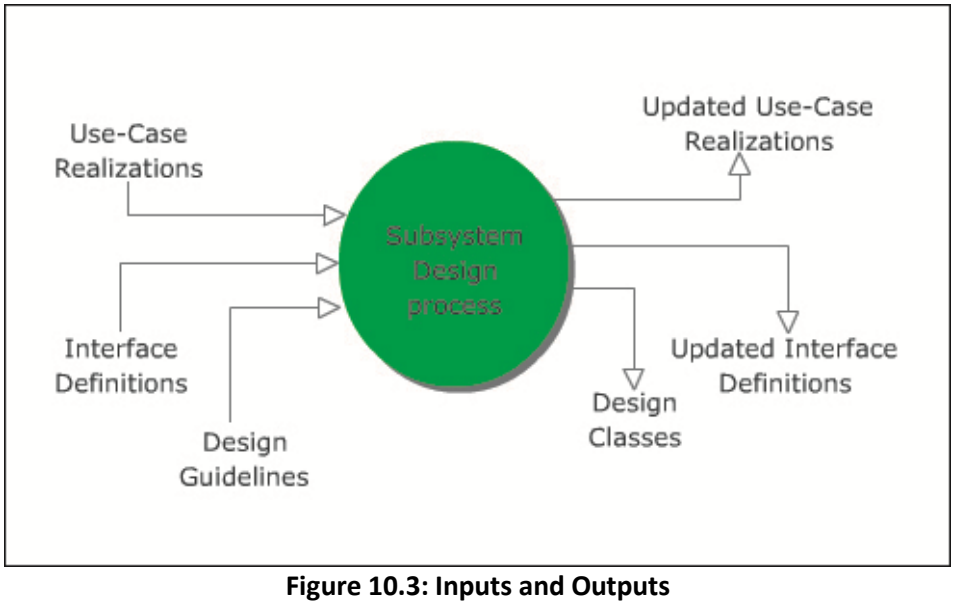


Figure 10.3: Inputs and Outputs

10.2.10 Guidelines for designing good subsystems

Guidelines for designing good subsystems:

- Each Subsystem should be as independent (modular) as possible from other parts of the system. It gives the designer total freedom in designing the internal behavior of the subsystem, as long as it provides the

correct external behavior

- Different parts of the system should be evolvable independent of other parts. This minimizes the impact of changes and saves maintenance efforts. Otherwise the system becomes brittle
- Any part of the system should be replaceable with a new part, provided the new part supports the same interfaces. In order to ensure that subsystems are replaceable elements in the model, the following are necessary

- The contents of a subsystem should not be exposed. Subsystem elements should not have 'public' visibility

- A subsystem should not directly depend on any external model elements. It should only depend on the **interfaces** of other model elements and not the **model elements** themselves.

10.2.11 Steps in designing subsystems

Steps in designing Subsystems:
The main steps in designing a Subsystem include:
1. Distributing Subsystem
2. Documenting Subsystem
3. Describing Subsystem
4. Including Checkpoints

- **Distributing Subsystem behavior to Subsystem Elements**

  First, the responsibilities allocated to the subsystems must be taken and further allocated to the subsystem elements.

- **Documenting Subsystem Elements**

  The internal structure of the subsystem needs to be documented or modeled using Class and State diagrams.

- **Describing Subsystem Dependencies**

  A subsystem's behavior can depend on the behavior of another subsystem's element. This should be recorded. Therefore, the external elements on which the subsystem depends needs to be documented.

- **Including Checkpoints**
  The purpose or behavior and key entities of a subsystem should be verified by identifying check points and validating them. Consistency with other models may also be checked at this point.

10.2.12 Distributing Subsystem behavior to Subsystem Elements

- **Why distribute?**

  This is done to:

  1. Detail the internal behavior of the subsystem
  2. Identify the need for new classes or subsystems in order to satisfy the behavioral requirements of the subsystem

- **How is distribution done?**

  Consider the Use-Case details and the Architectural Framework.

  While the interaction diagrams show the design elements, like the design classes and subsystem interfaces, this distribution of its behavior to its internal elements will describe the "local" interactions **within** a subsystem to clarify its internal design.

10.2.13 Subsystem Responsibilities

- **What are Subsystem Responsibilities?**

  The interfaces that a subsystem realizes, defines its external behavior. When a subsystem realizes an interface, it makes a commitment to support each and every operation defined by the interface. This is the responsibility of a subsystem.

  This may be done by:

  1. A class in a subsystem: An operation on a class contained by the subsystem; this operation may require collaboration with other classes or subsystems

  2. An internal interface: An operation on an interface realized by a subsystem contained within the larger subsystem

10.2.14 Distributing Subsystem Responsibilities

- **How are Subsystem Responsibilities distributed?**

  - Identify the class or a contained subsystem (a contained subsystem is a subsystem within the subsystem) that is needed to perform the operation

  - If an existing class or contained subsystem cannot perform the operation, identify and create NEW classes or contained subsystems

  - Reconsider the content and boundary of the subsystem while creating new classes or contained subsystems

  - Avoid duplication of class in two different subsystems. Duplication indicates subsystem boundaries are ambiguous

  - Revisit the architectural design to keep a check on the balance of the subsystem responsibilities

  - Document collaborations of model elements within the subsystem using one or more interaction diagrams. These diagrams are essential for subsystems with complex internal designs. These internal interaction diagrams should incorporate any applicable, mechanisms initially identified in Architectural Design, for example, persistence, distribution

- **How are Subsystem Responsibilities distributed?**

  - Subsystem interaction diagrams depict the realization of the subsystem behavior
  - They show the 'internal' interaction
  - They show exactly which classes provide the interface
  - They show what needs to happen internally to provide the subsystem's functionality
  - They show which classes are used to design the internal behavior of the subsystem

10.2.15 Documenting Subsystem Elements

These are the steps needed to document or model the internal structure of the subsystem:

- Model the subsystem element relationships

- Create one or more class diagrams (to improve readability) showing the elements contained by the subsystem, and their associations with one another

- Create a state diagram to document the possible states of the subsystem interface operations, for example, operation 1 be executed before operation 2

10.2.16 Subsystem Dependencies

A subsystem's behavior can depend on the behavior of another subsystem's element. This should be recorded. Therefore, the external elements on which the subsystem depends needs to be documented.

The Architect establishes the ground rules for the dependencies. However, the Subsystem Designer should take up identifying and using the services of other subsystems based on the architect's guidelines.

However, it is important to ensure that the dependency is on the interface with the other subsystem, and not on the element itself, which is the essence of object orientation.

Figure 10.4 depicts subsystem dependencies.

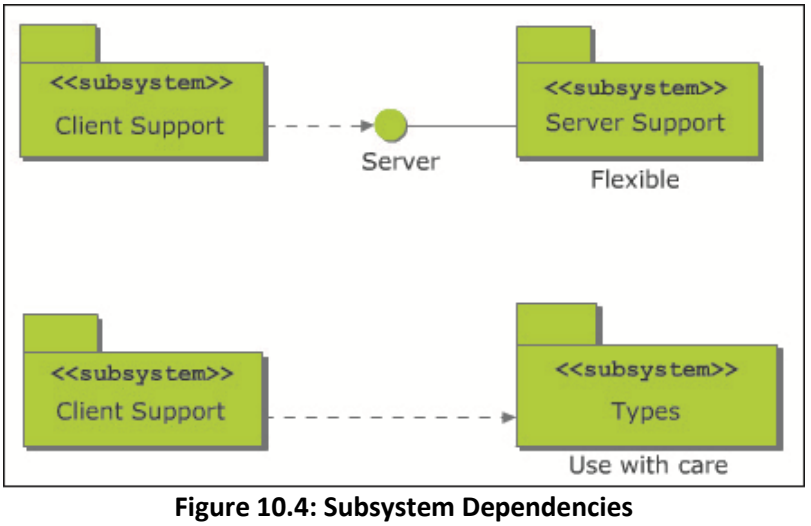


Figure 10.4: Subsystem Dependencies

10.2.17 Subsystem Interactions

Subsystem Interfaces may be represented as Classes using:

- Elided or Iconic Representation
- Canonical Representation

Iconic or Elided Representation uses a “lollipop” icon. The realization relationship may be modeled using a circle shape called “lollipop” as the contract class.

Canonical Representation uses the guillemets of Stereotype icon. The realization relationship may be modeled as a dashed line with a hollow arrowhead pointing at the contract class.

There should be at least one interaction diagram per interface operation.

Subsystem Interfaces may be represented as Classes using:

The interaction diagram illustrates how model elements in the subsystem perform the interface operations. The interaction diagram is named

“<interface name>::<operation name>”

This naming convention simplifies future tracing of interface behaviors to the classes that implement the interface operations.

Figures 10.5 and 10.6 depict subsystem interactions.

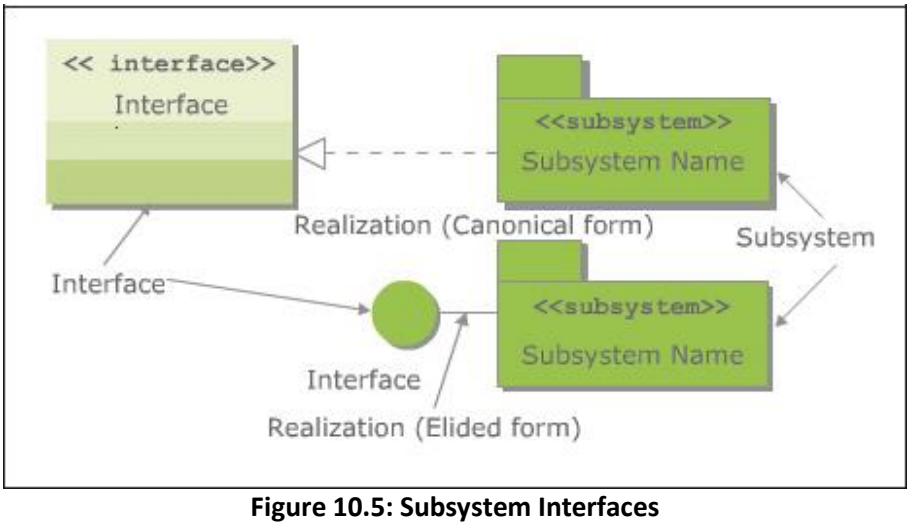


Figure 10.5: Subsystem Interfaces

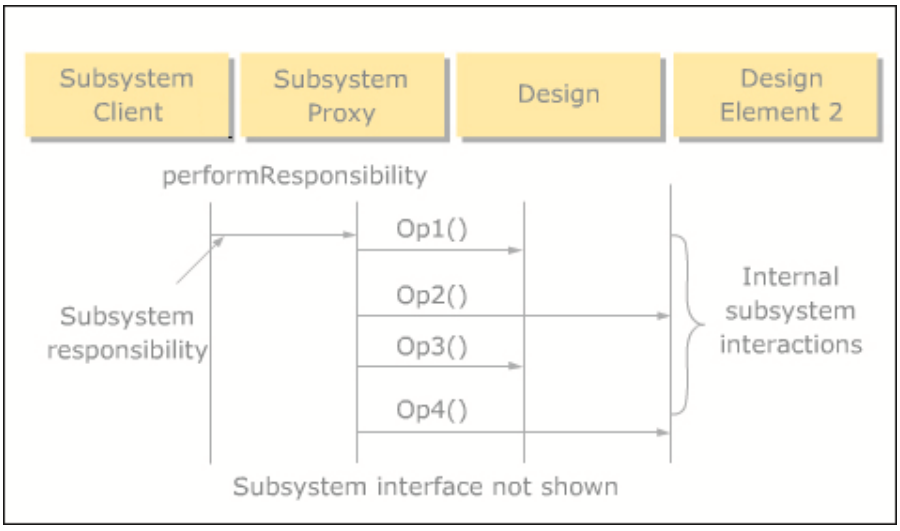


Figure 10.6: Subsystem Interactions

10.2.18 Layering

During Architectural Analysis, the initial structure for the Design Model is defined.

This constitutes

- The initial set of packages and their dependencies
- The organization of these packages into layers

What are layers?

Layers are ordered grouping of functionalities. They are used to encapsulate conceptual boundaries between different kinds of services and provide useful abstractions. This makes the design easier to understand.

The number of layers and what makes up each layer depends on the complexity of both the problem domain and the solution space.

Layering can be done for packages or subsystems.

10.2.19 The Three – Tiered Architecture: A typical layering approach

Elements prone to change with **user requirements** need to be in the highest layers. Packages or subsystems with application-specific functionalities are located in the upper layers.

Elements prone to change with **implementation platform** (hardware, language, operating system, database, etc.) need to be in the lowest layers. Packages or subsystems with operation-specific functionalities (of deployment environment) are at the lower layers.

Elements generally applicable across wide ranges of systems and implementation environments need to be sandwiched in the middle layers. Packages or subsystems with domain functionalities or general purpose services are positioned in the middle layers.

Figure 10.7 depicts the Three-Tiered Architecture.

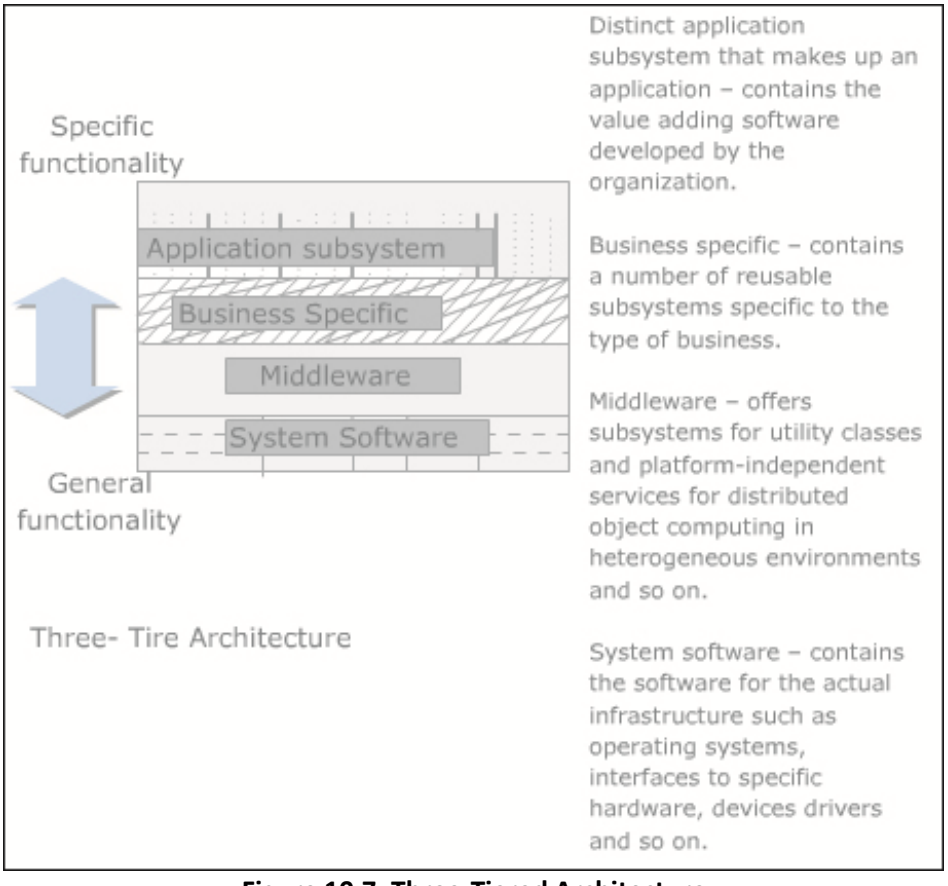


Figure 10.7: Three-Tiered Architecture

10.2.20 Concurrency

The **'Concurrency'** phase identifies the independent threads of control and maps them to the design elements (subsystems and classes).

The focus is on the Process View of the architecture.

Describe Concurrency is a separate activity and is at the same level as Architectural Design. It may also be considered a part of Architectural Design. It is performed by the **architect** as it involves establishing infrastructure applicable to the entire system.

However, if the system under development will only run on one-processor, then there is no need for a separate Process View. In such a case, we can skip describe concurrency.

Figure 10.8 depicts concurrency.

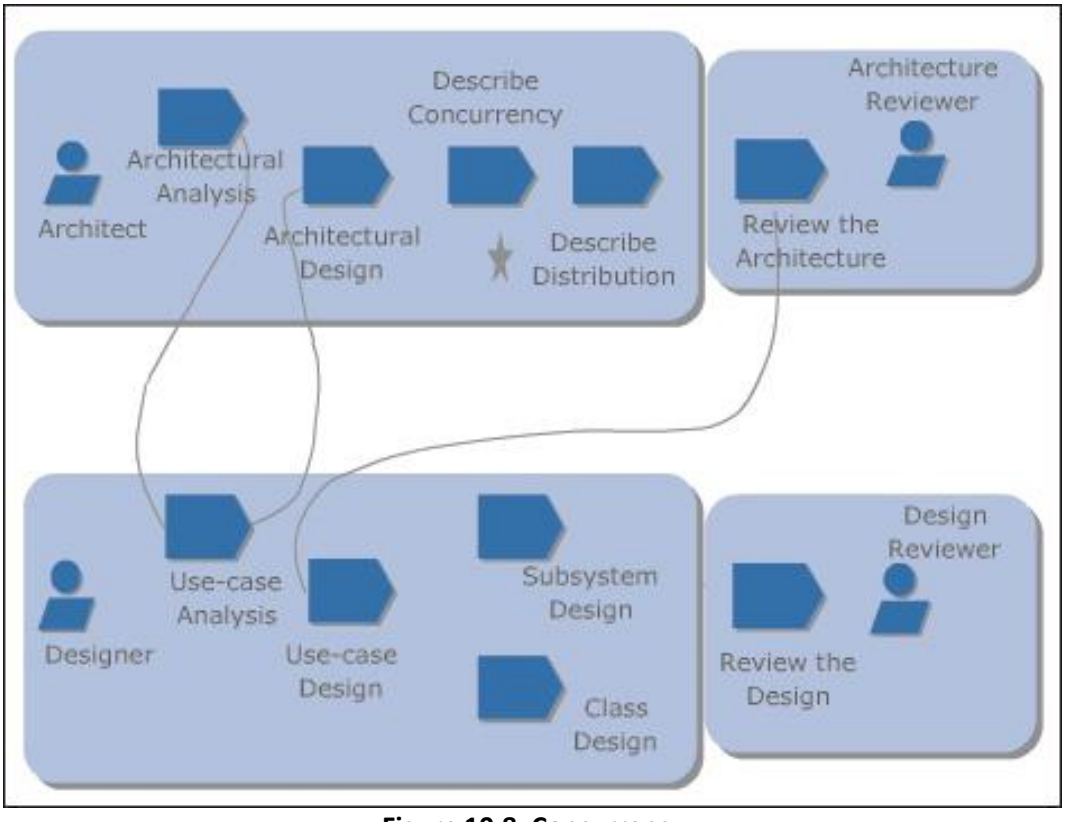


Figure 10.8: Concurrency

10.2.21 Processes and Threads in Concurrency

- **Process and Thread:**

  A **process** executes in its own memory space, encapsulates, and protects its internal structure. A process can be viewed as being a system of its own. It is initiated by an executable program. It is the execution environment in which the class or subsystem instances run. A process may contain multiple threads (for example, a number of processes may execute within a single process, sharing the same memory space).

  The execution environment of a process may be divided into one or more threads of control. A **thread** executes in a memory space that it may share with other threads.

  The difference between process and thread has to do with the memory space in which they execute.

10.2.22 Describe Concurrency Activity

- **PURPOSE OF DESCRIBE CONCURRENCY**

  The purpose of **Describe Concurrency:**

  - Express how the concurrency requirements of the system will impact the design
  - Identify the processes that should exist in the system
  - Describe the process communications

- Describe how they will be created and destroyed
- Show how they will be mapped onto the implementation environment

➢ **INPUTS FOR DESCRIBE CONCURRENCY ARE AS FOLLOWS:**

- Supplementary Specification
- Design model

➢ **OUTPUT FROM DESCRIBE CONCURRENCY**

- Process view of the software document

  The Process View describes the planned process structure of the system. It is concerned with dynamic, run-time decomposition, and takes into account some non-functional requirements, such as performance, availability, as well as some derived requirements resulting from the need to spread the system onto several computers.

  In The Process View, the system is decomposed into a set of independent tasks/threads, processes and process groups. The Process View describes process interaction, communication, and synchronization.

10.2.23 Process View

The Process View describes the planned process structure of the system. It is concerned with dynamic, run-time decomposition, and takes into account some non-functional requirements, such as performance, availability, as well as some derived requirements resulting from the need to spread the system onto several computers.

In The Process View, the system is decomposed into a set of independent tasks/threads, processes and process groups. The Process View describes process interaction, communication, and synchronization.

10.2.24 Factors to be considered in Concurrency

Shaping the architecture of a system will depend on Concurrency requirements.

Concurrency requirements define the extent to which parallel execution of tasks is required for the system.

A multi-process architecture is needed for a system whose behavior needs to be distributed across processors or nodes.
It may be necessary for the application to monopolize resources by creating a single large process, using threads to control execution within the process.
Communication between threads is generally faster and more efficient than that between processes. In many systems, there may be a maximum number of threads per process or processes per node.

Knowledge Check 2

1. Match the following descriptions with the corresponding feature.

	DESCRIPTION		FEATURE
(A)	The top layers of architecture	(1)	Elided form of Interface
(B)	The bottom layers of architecture	(2)	Application specific functionalities
(C)	Use Case realizations	(3)	Implementation specific functionalities
(D)	Lollipop representations	(4)	Helps maintaining modularity and hierarchy
(E)	Layering subsystems or packages	(5)	Inputs for designing subsystem

(A)	A-3, B-4, C-2, D-5, E-1	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-3, C-5, D-1, E-4	(D)	A-5, B-3, C-1, D-2, E-4

2. Match the following descriptions with the corresponding element.

	DESCRIPTION		ELEMENT
(A)	Architectural frameworks can show	1.	Process view
(B)	The '4+1' view illustrates parallel processes in	2.	System characteristics
(c)	Schematic patterns of a system show	3.	Common architectural patterns
(D)	Black board, Pipes and filters, MVC and Layers are	4.	All of the above

(A)	A-3, B-4, C-2, D-1	(C)	A-4, B-1, C-2, D-3
(B)	A-2, B-3, C-4, D-1	(D)	A-4, B-3, C-1, D-2

3. Which of the following statements are true?

(A)	Design guidelines are inputs to Subsystem Design and Design Classes are outputs
(B)	Subsystem elements should be exposed and have 'public' visibility
(C)	Canonical Representation uses a “lollipop” icon
(D)	Parts of subsystems should be independently evolvable
(E)	Class and State Diagrams are used to model the internal structure of subsystems

(A)	A, C	(C)	A, B, C
(B)	B, D, E	(D)	A, D, E

4. Match the following descriptions with the corresponding features.

	DESCRIPTION		FEATURE
(A)	Architectural mechanism	(1)	Focus on Process View of the architecture
(B)	Describe Concurrency	(2)	Greater detail of the system is described
(C)	Subsystem Design	(3)	Focus on Analysis efforts
(D)	Supplementary specification	(4)	Derived from Class relationships
(E)	The Process relationships	(5)	The input required for describe concurrency

(A)	A-3, B-4, C-2, D-5, E-1	(C)	A-3, B-1, C-2, D-5, E-4
(B)	A-2, B-3, C-5, D-1, E-4	(D)	A-5, B-3, C-1, D-2, E-4

Module Summary

In this module, **System Design**, you learnt about:

- **System Design**
  System Design is about managing the complexity of a system efficiently with the help of a model. Though System Design is close to the coding of the system, it focuses on understanding the solution and performance rather than coding syntax. It can span multiple technologies and often involves multiple sub-disciplines.

- **System Design Activities**
  System Design Activities includes the processes like Architectural & Framework Selection, Subsystem Breakdown, Layers and Partition and the use of Concurrency.

Onlinevarsity
your e-way to learning

WRITE-UPS BY

EXPERTS AND **LEARNERS**

TO PROMOTE NEW AVENUES AND
ENHANCE THE LEARNING EXPERIENCE

FOR FURTHER READING, LOGIN TO

www.onlinevarsity.com

MODULE 11

Design Patterns

Welcome to the module, **Design Patterns**. This module introduces different Design Patterns and explains Creational, Structural, and Behavioral Design Patterns.

In this module, you will learn about:

- Creational Design Patterns
- Structural Design Patterns
- Behavioral Design Patterns

11.1 Creational Design Patterns

In this first lesson, **Creational Design Patterns**, you will learn to:

- Define and describe the factory design pattern
- Define and describe the singleton design pattern

Creational Design Patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
All the creational patterns define the best possible way in which an object can be instantiated.
Creational Design Patterns solve this problem by controlling this object creation.
Creational Design Patterns are of two types. They are as follows:

- Factory Pattern
- Singleton Pattern

The basic form of object creation for example using “new” keyword in java, could result in design problems or added complexity to the design as there is a certain amount of hard coding in the design.

Figure 11.1 depicts the Creational Design pattern.

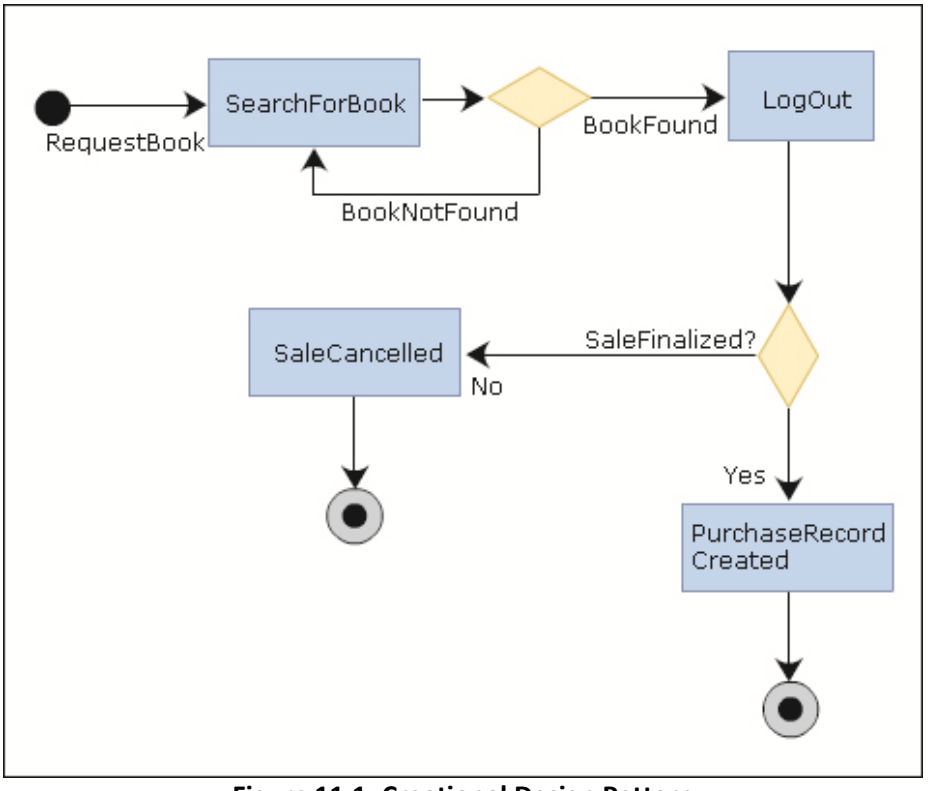


Figure 11.1: Creational Design Pattern

11.1.1 Factory

Factory pattern is used when a class cannot anticipate the class of objects it must create. If a class wants its subclasses to specify the objects it creates, this pattern is preferred.
Classes delegate responsibility to one of several helper subclasses, and we want to localize the knowledge of which helper subclass is the delegate.

- **Framework of a Computer**
  We want to develop a framework of a Computer that has memory, CPU and Modem. The actual memory, CPU, and Modem that is used depends on the actual computer model being used. We want to provide a configure function that will configure any computer with appropriate parts. This function must be written such that it does not depend on the specifics of a computer model or the components. The above problem can be implemented using factory method.

Figure 11.2 depicts the Factory pattern.

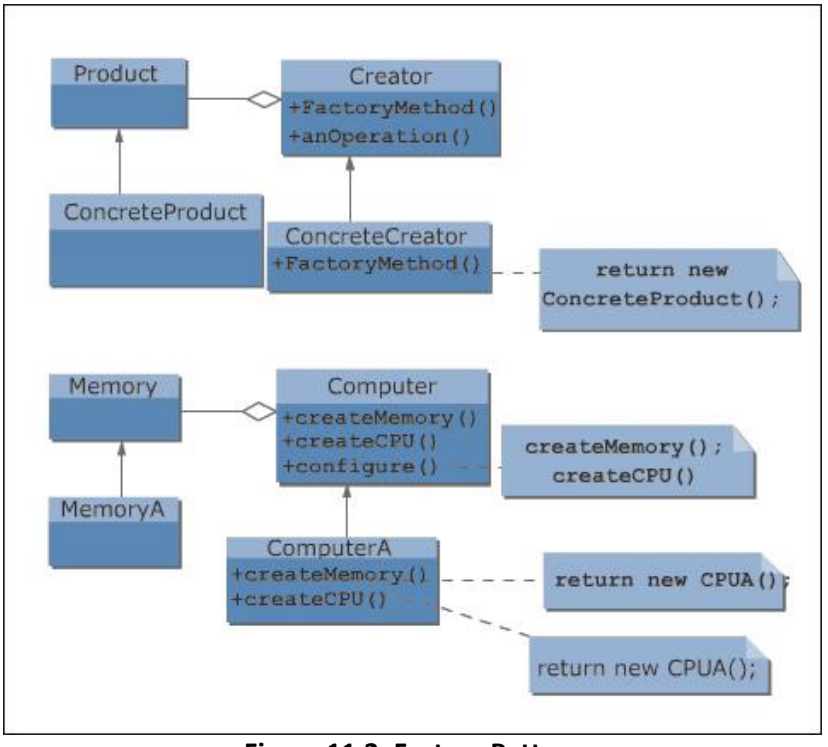


Figure 11.2: Factory Pattern

11.1.2 Singleton Design Pattern

Ensure a class only has one instance, and provide a global point of access to it.
Singleton pattern is used when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point. It provides a controlled access to sole instance. Singleton pattern permits refinement of operations & representation.
The main keyword used in implementation of singleton pattern is “static”. A static or shared keyword is used to share a data with all the objects.

An example that would benefit from the singleton pattern is an application using a File Manager. It needs only one File Manager object. It must not allow the creation of more than one object of this class. Using this object, we can perform input/output operations in our file system.

Figure 11.3 depicts the Singleton pattern.

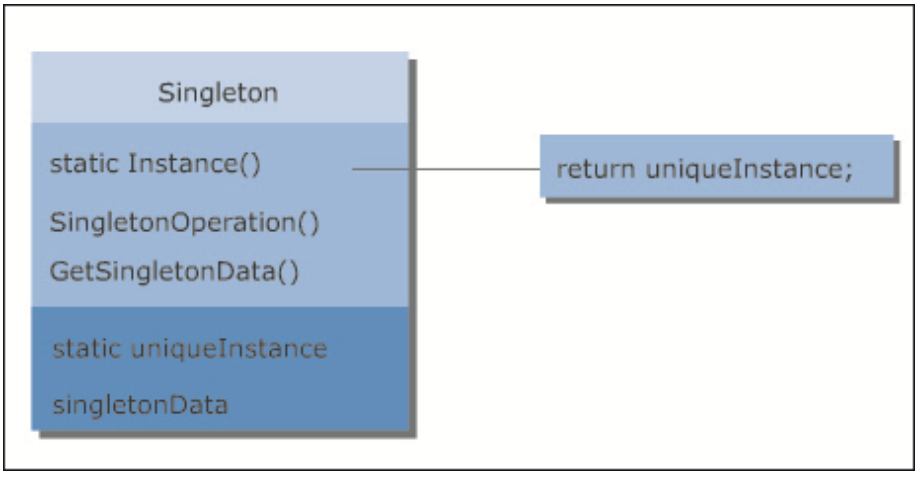


Figure 11.3: Singleton Pattern

Knowledge Check 1

1. Which of these statements about the Factory Design Pattern are true?

(A)	Factory pattern is used when a class can anticipate the class of objects it must create
(B)	Creational Design pattern deal with object creation pattern
(C)	Factory pattern results in creation of parallel hierarchy of classes

(A)	A, B	(C)	A, C
(B)	B	(D)	B, C

2. Match the following descriptions with the corresponding elements.

	DESCRIPTION	ELEMENT	
(A)	Several design patterns are implemented using	(1)	Access point
(B)	Singleton pattern is used when there must be exactly	(2)	Sole instance
(C)	In singleton pattern the class must be accessible to clients from a well known	(3)	Singleton pattern
(D)	Singleton pattern provides a controlled access to	(4)	Static
(E)	The main keyword used in implementation of singleton pattern is	(5)	One instance of a class

(A)	A-3, B-5, C-1, D-2, E-4	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-3, C-5, D-1, E-4	(D)	A-5, B-3, C-1, D-2, E-4

11.2 Structural Design Pattern

In this second lesson, **Structural Design Pattern**, you will learn to:

- Define and describe the adapter design pattern
- Define and describe the proxy design pattern

11.2.1 Concept and Definition

Structural patterns describe how classes and objects can be combined to form larger structures.
The patterns generally fall under class scope and object scope. The difference between class scope and object scope is that class patterns describe how inheritance can be used to provide more useful program interfaces.

Object patterns, on the other hand, describe how objects can be composed into larger structures using object composition, or the inclusion of objects within other objects.
Structural Design Patterns ease the design by identifying a simple way to realize relationships between entities.

Structural Design Patterns are of two types. They are as follows:
- Adapter Design Pattern
- Proxy Design Pattern

11.2.2 Adapter

Converts the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

Adapter pattern is used under the following scenarios.
- Want to use existing class, and its interface does not match the one we need
- Want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces
- (Object adapter) we need to use several existing subclasses, but it is impractical to adapt their interface by sub classing every one. An object adapter can adapt the interface of its parent class

Adapter pattern comes in two flavors:
- Class adapter
- Object adapter

Figure 11.4 depicts the Class Adapter and Object Adapter.

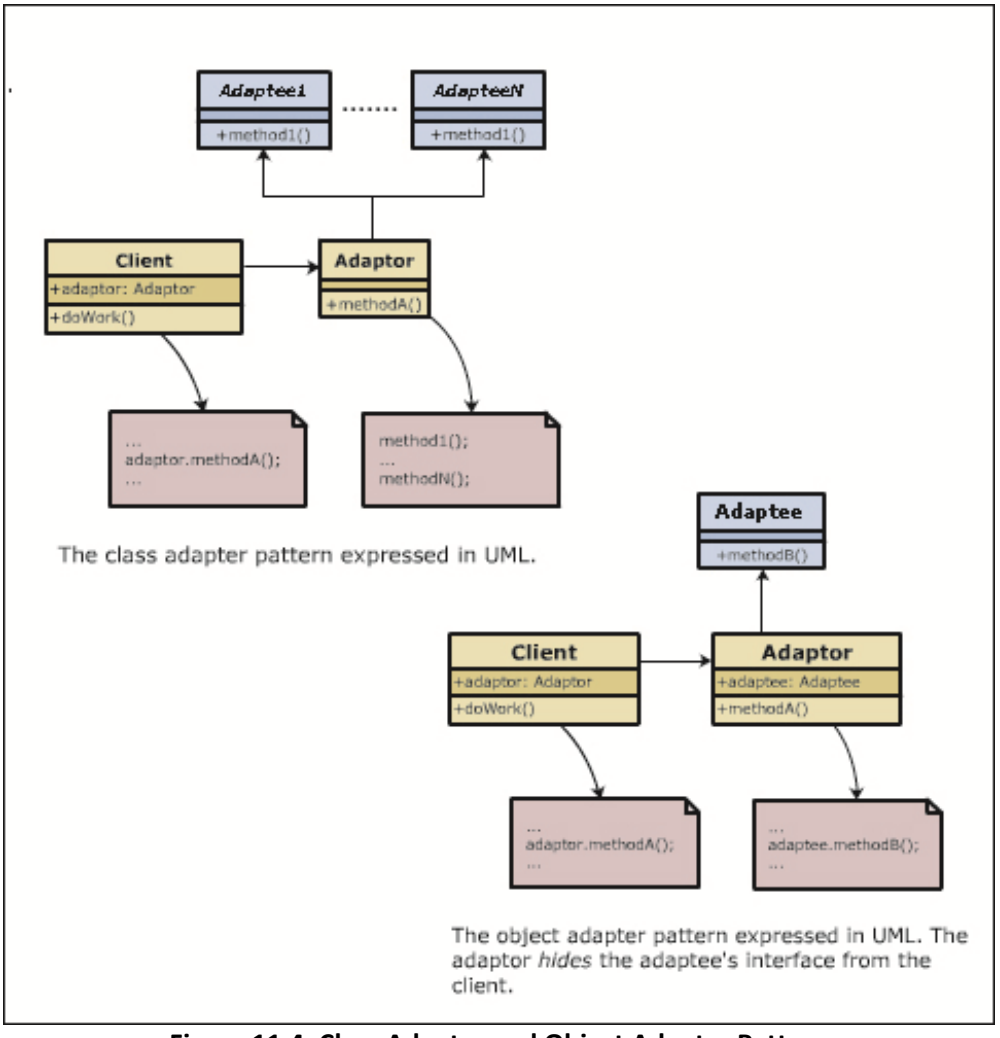


Figure 11.4: Class Adapter and Object Adapter Pattern

11.2.3 Class Adapter

- Adapts Adaptee to Target by committing to a concrete Adapter class
- Cannot adapt a class and all its subclasses
- Lets Adapter override some of Adaptee's behavior
- No extra objects due to adapter usage

Figure 11.5 depicts a Class Adapter.

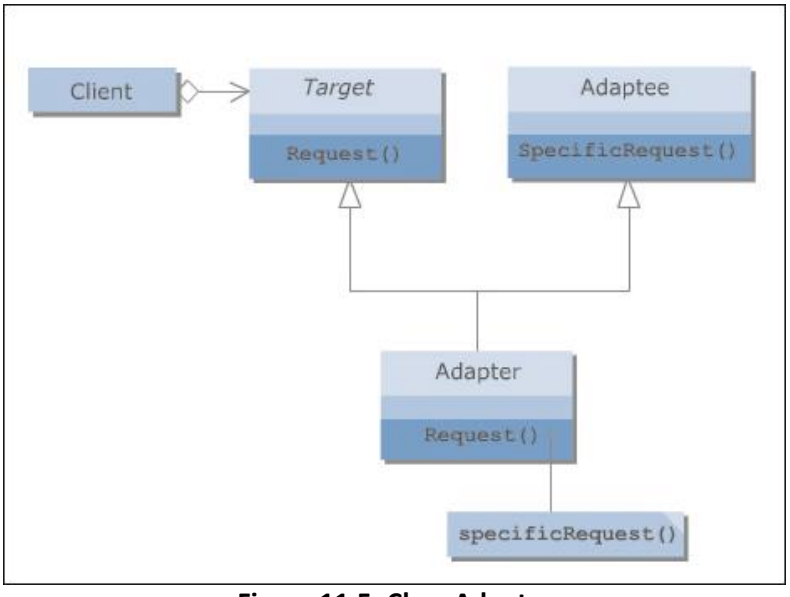


Figure 11.5: Class Adapter

11.2.4 Object Adapter

- Allows a single Adapter work with many Adaptees - a class and its subclasses
- Can add functionality to all Adaptees
- Harder to override Adaptee behavior

Figure 11.6 depicts an Object Adapter.

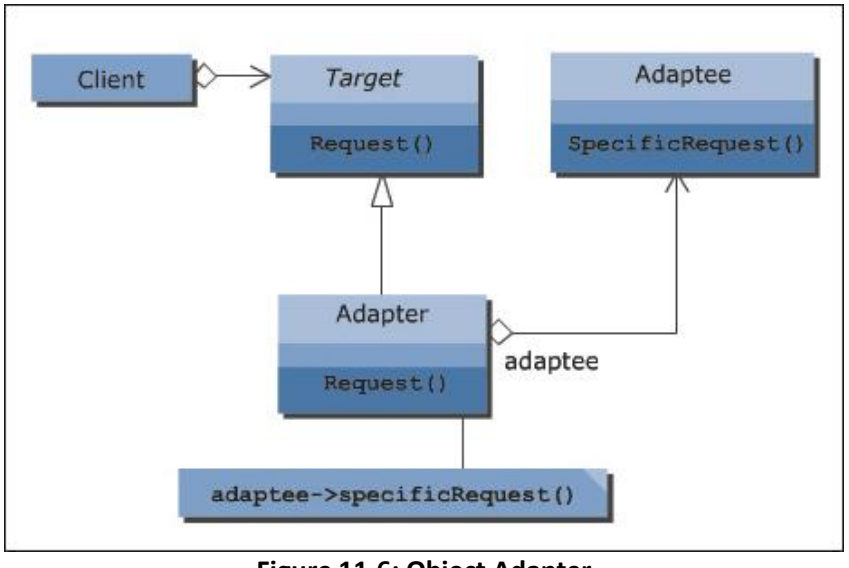


Figure 11.6: Object Adapter

11.2.5 Proxy Pattern

It provides a surrogate or placeholder for another object to control access to it. **Proxy pattern** is used in the following scenarios.

- An application needs to communicate with a remote subsystem to obtain some critical information

- The application may have code all over that deals with communication logic and data
- How to minimize the code for communication logic?
- What if not all data on the remote subsystem is changing dynamically?
- Proxy pattern is used when a more sophisticated reference than a simple pointer is needed to manage object lifetime. Proxy pattern introduces a level of indirection in accessing objects which provides flexibility.
- Proxy pattern is similar to adapter pattern in its implementation structure but while adapter changes its interface proxy provides the same interface.

Figure 11.7 depicts the proxy pattern.

Figure 11.7: Proxy Pattern

There are different types of proxies available.
1. Remote proxy: provides local representative for object in different address space
2. Virtual proxy: creates expensive objects on demand
3. Protection proxy: controls access to original object
4. Copy-on-modify proxy: guts of data not copied until if and when data modification starts

Knowledge Check 2

1. Match the following descriptions with the corresponding elements.

	DESCRIPTION	ELEMENT	
(A)	Class adapter adapts adaptee to target by committing to a concrete	(1)	Adapter
(B)	Class adapter allows adapter override some of	(2)	Adapter class
(C)	To use existing class, and its interface which does not match what we need, we use	(3)	Adaptees
(D)	Object adapter can add functionality to all	(4)	Adapter usage
(E)	In class adapter there are no extra objects due to	(5)	Adaptee's behavior

(A)	A-3, B-5, C-1, D-2, E-4	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-3, C-5, D-1, E-4	(D)	A-5, B-3, C-1, D-2, E-4

2. Match the following descriptions with the corresponding elements.

	DESCRIPTION	ELEMENT	
(A)	When an application needs to communicate with a remote subsystem to obtain some	(1)	Demand
(B)	Remote proxy provides local representative for object in different	(2)	Proxy patter is used
(C)	Virtual proxy creates expensive objects on	(3)	Data modification starts
(D)	Protection proxy controls access to	(4)	Address space
(E)	Copy-on-modify proxy guts of data not copied until if and when	(5)	Original objects

(A)	A-3, B-5, C-1, D-2, E-4	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-5, B-3, C-1, D-2, E-4

11.3 Behavioral Design Pattern

In this third lesson, **Behavioral Design Pattern**, you will learn to:

- Define and describe the command design pattern
- Define and describe the iterator design pattern

11.3.1 Purpose and Definition

Behavioral design patterns identify common communication patterns between objects and realize these patterns.

They are concerned with the assignment of responsibilities to objects, or, encapsulating behavior in an object and delegating requests to it.

The interactions between the objects are such that they are talking to each other and still are loosely coupled. These patterns increase flexibility in carrying out this communication.

11.3.2 Command Design Pattern

Using the **command pattern**, we can wrap a command as an object. It allows us to achieve complete decoupling between the sender and the receiver.

A SENDER is an object that invokes an operation, and a RECEIVER is an object that receives the request to execute a certain operation. The term REQUEST here refers to the command that is to be executed. The Command pattern also allows us to vary when and how a request is fulfilled.

The Command pattern turns the request itself into an object. This object can be stored and passed around like other objects.
Therefore, a Command pattern provides us flexibility as well as extensibility by providing an object-oriented solution to decouple a sender and receiver.

- An example of Command pattern is to use a Switch that has a common interface to flip up and flip down. The switch has to be used on different entities like Light, Fan, Air Conditioner etc. How we can use the same interface Switch to work with different objects maintaining a decoupling between them is the essence of Command pattern. Here Switch is the invoker and light, fan and Air conditioners are receiver objects. Each receiver has its own command object that conforms to a standard command interface.

Figure 11.8 depicts the Command Design pattern.

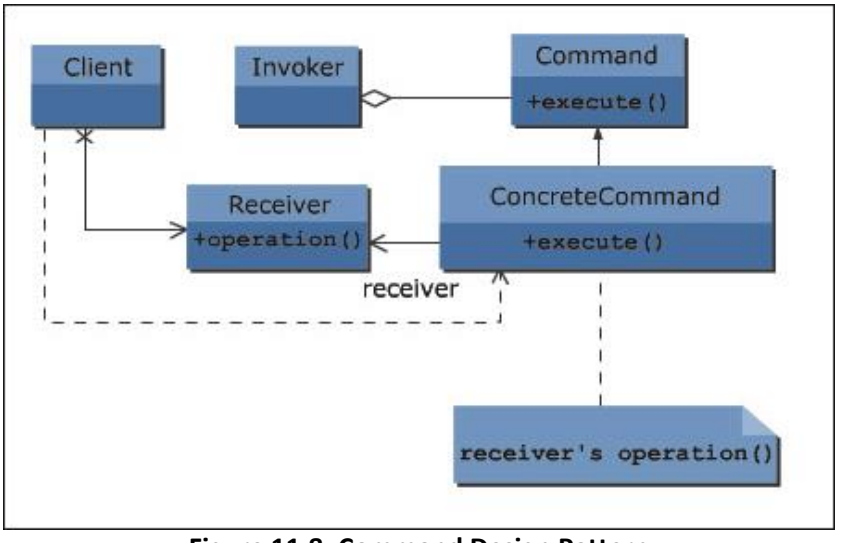


Figure 11.8: Command Design Pattern

11.3.3 Iterator Design Pattern

OO languages like Java provide several types of collections like List, Set, Queue, and Vector.

Iterator pattern is used when we want to perform some operation on each element in a collection, without regard to which collection we use. Iterator patterns are used to expose a collection without exposing its underlying implementation.

Iterator pattern is used when:

- Elements of an Aggregate needs to be accessed without exposing internal representation of the aggregate
- Multiple traversals on an aggregate
- Uniform traversal on different aggregates

Iterator pattern supports variations in the traversal of an aggregate. It is often applied to recursive structures like trees.

Figure 11.9 depicts the Iterator pattern.

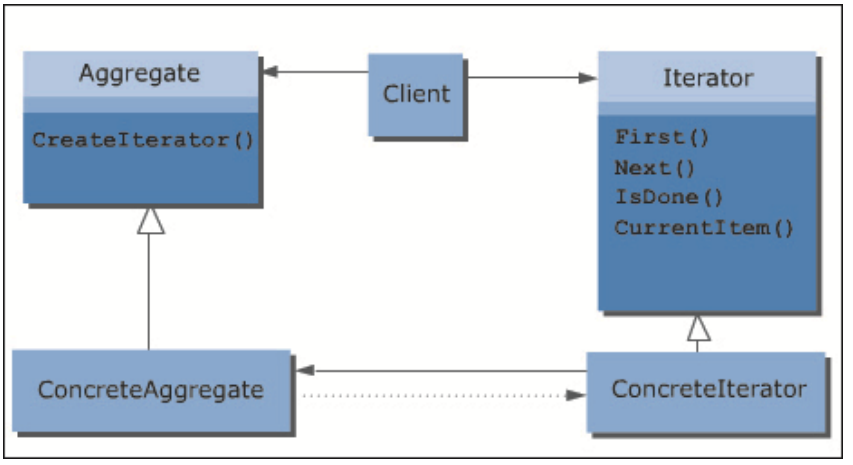


Figure 11.9: Iterator Pattern

Knowledge Check 3

1. Match the following descriptions with the corresponding elements.

	DESCRIPTION	ELEMENT	
(A)	Behavioral design patterns are concerned with assignment of	(1)	Communication
(B)	Behavioral design patterns increase flexibility in carrying out	(2)	Request
(C)	Command pattern allows to achieve complete decoupling between the	(3)	Responsibilities to objects
(D)	Invoker asks the command to carry out the	(4)	Executing an operation
(E)	Command declares an interface for	(5)	Sender and receiver

(A)	A-3, B-5, C-1, D-2, E-4	(C)	A-3, B-1, C-5, D-2, E-4
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-5, B-3, C-1, D-2, E-4

2. Match the following descriptions with the corresponding elements.

	DESCRIPTION	ELEMENT	
(A)	Collection classes like vector, stack, list can be exposed using	(1)	Different aggregates
(B)	Iterator patterns are used to expose	(2)	Trees
(C)	Iterator pattern is used when multiple traversals on	(3)	Iterator Pattern
(D)	Iterator pattern is used when uniform traversals on	(4)	An aggregate
(E)	Iterator pattern is applied to recursive structures like	(5)	Collections

(A)	A-3, B-5, C-4, D-1, E-2	(C)	A-3, B-1, C-5, D-2, E-4
(B)	A-2, B-4, C-1, D-5, E-3	(D)	A-5, B-3, C-1, D-2, E-4

Module Summary

In this module, **Design Patterns**, you learnt about:

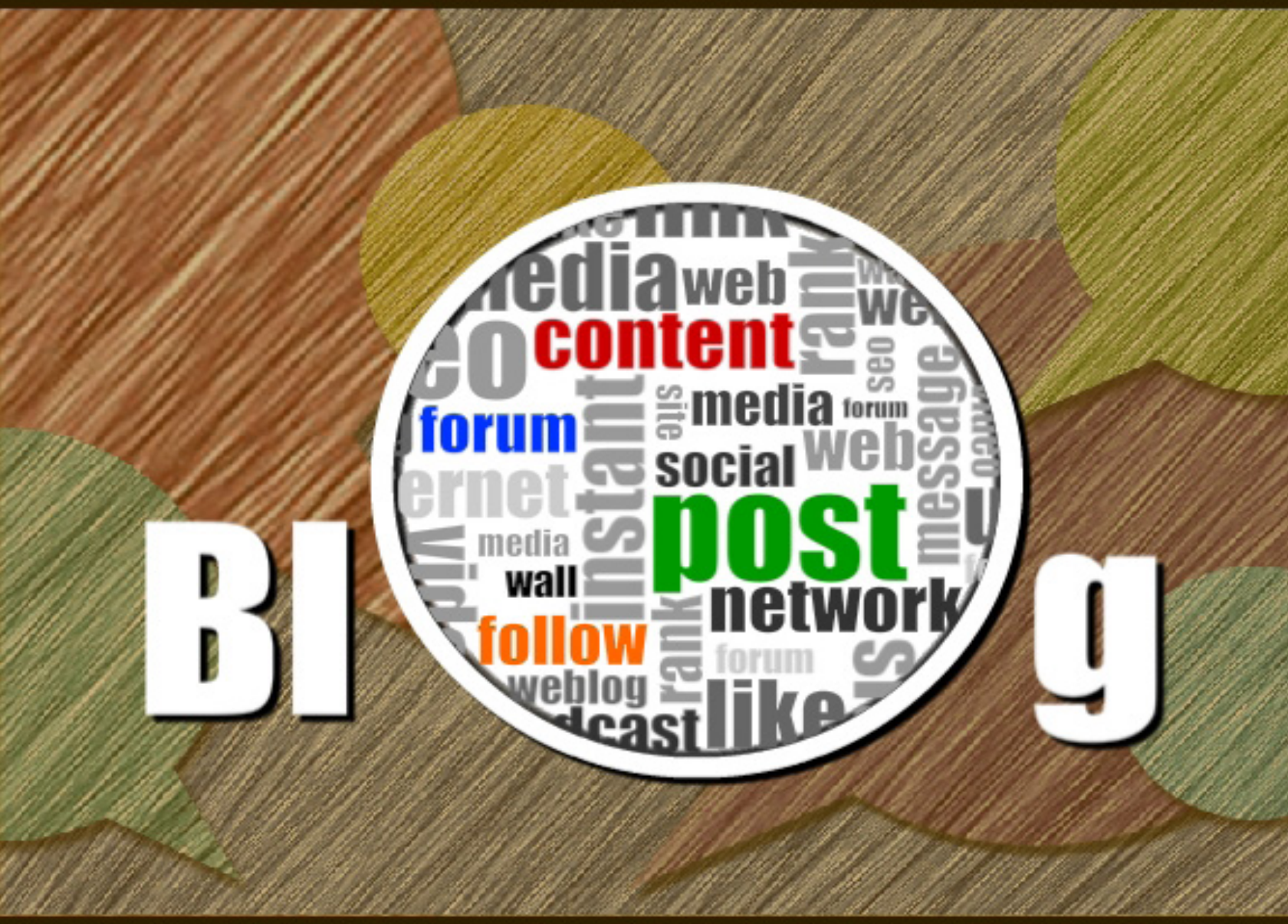
- **Creational Design Patterns**
  Creational Design Patterns deal with object creation mechanisms. The objects have to be created in a manner suitable for each unique situation. Creational Design Patterns solve this problem by controlling this object creation.

- **Structural Design Pattern**
  Structural Design Patterns describe how classes and objects can be combined to form larger structures. The patterns generally fall under class scope and object scope. The class patterns describe how inheritance can be used to provide more useful program interfaces.

- **Behavioral Design Pattern**
  Behavioral Design Patterns identify common communication patterns between objects and realize these patterns. They are concerned with the assignment of responsibilities to objects, or encapsulating behavior in an object and delegating requests to it. The interactions between the objects are such that they talk to each other and are loosely coupled.

Onlinevarsity

your e-way to learning



Balanced Learner-Oriented Guide

for enriched learning available @

www.onlinevarsity.com

MODULE 12

New Features of UML 2.0

12.1 Introduction

Welcome to the module, **New Features of UML 2.0**. This module introduces new features of UML and explains Interaction Overview Diagrams, Composite Structure Diagrams, and Timing Diagrams.

In this module, you will learn about:

- New Features
- Interaction Overview Diagrams
- Composite Structure Diagrams

12.1.1 New Features

In this first lesson, **New Features**, you will learn to:

- Define and describe interaction overview diagrams
- Define and describe composite structure diagrams
- Define and describe timing diagrams

12.1.2 Interaction Overview Diagrams

Interaction Overview diagram combines Activity diagrams with Sequence diagrams to give a big picture of the problem.
Interaction Overview Diagram focuses on the overview of the flow of control of the interactions. The Interaction Overview Diagram describes the interactions where messages and lifelines are hidden.
An interaction overview diagram is a form of activity diagram in which the nodes represent interaction diagrams.
Most of the notation for interaction overview diagrams is the same for activity diagrams.

Figure 12.1 depicts Interaction Overview diagram.

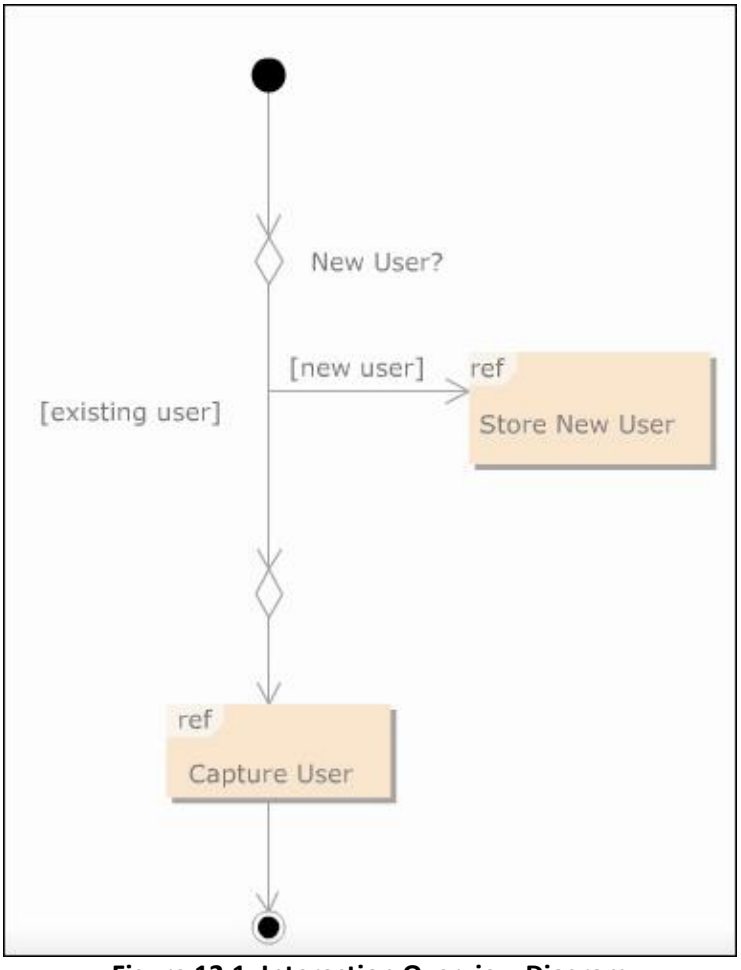


Figure 12.1: Interaction Overview Diagram

12.1.3 Composite Structure Diagrams

A composite structure diagram is a diagram that shows the internal structure of a class, component, or collaboration, including its interaction points to other parts of the system. It shows the configuration and relationship of parts that together, perform the behavior of the containing class.
An example of a composite structure that models a car built using an engine and set of components.

Class elements have been described in great detail in the section on class diagrams. This section describes the way classes can be displayed as composite elements exposing interfaces and containing ports and parts.

Figure 12.2 depicts Composite Structure diagram.

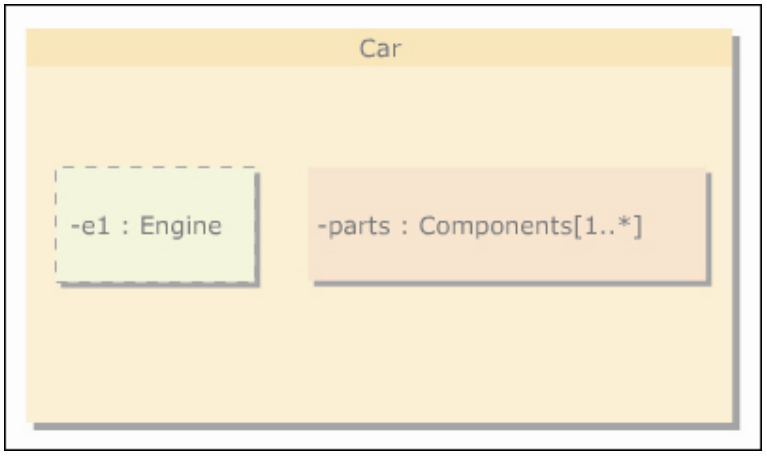


Figure 12.2: Composite Structure Diagram

12.1.4 Define and Describe Timing Diagrams

Timing diagrams are one of the new artifacts added to UML 2.0. They are used to explore the behaviors of one or more objects throughout a given period of time.

They are used to display the change in state or value of one or more elements over time. It can also show the interaction between timed events and the time and duration constraints that govern them.
It is a special purpose interaction diagram that focuses on the timing of events in a life of an object. In timing diagrams, the time axis is represented horizontally and the lifelines are shown vertically.

Figure 12.3 depicts Timing diagram.

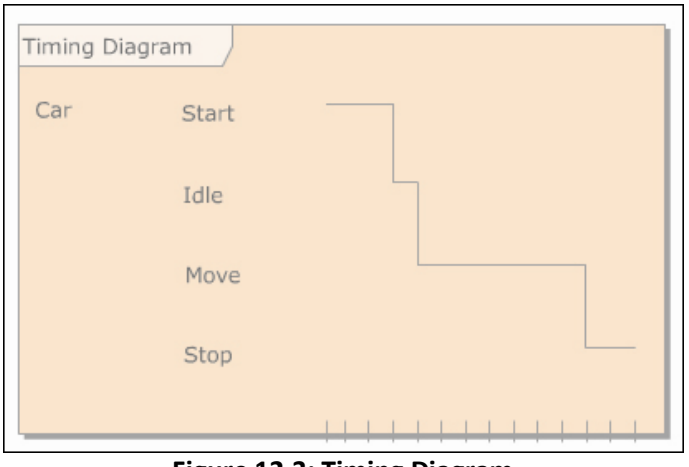


Figure 12.3: Timing Diagram

VP-UML supports both discrete timing and the general value lifeline style.

Timing diagrams are often used to design embedded software, such as control software for fuel injection system in an automobile.

Knowledge Check 1

1. Which of these statements about the Interaction Overview Diagrams are true?

(A)	The Interaction overview diagram describes the interactions where messages and lifelines are hidden
(B)	Focus of interaction overview diagram is not on the overview of flow of control of the interactions
(C)	The notation for interaction overview diagram is same for activity diagram

(A)	A, B	(C)	A, C
(B)	B	(D)	B, C

2. Which of these statements about the Composite Structure Diagrams are true?

(A)	Composite Structure Diagram shows the internal structure of class
(B)	Composite structure diagram excludes its interaction points to other points of the system
(C)	Composite structure diagram shows the configuration and relationships of parts

(A)	A, C	(C)	A, B
(B)	B	(D)	B, C

3. Match the following descriptions with the corresponding elements.

	DESCRIPTION	ELEMENT	
(A)	Timing diagrams are new artifacts added to	(1)	Behavior of one or more objects
(B)	Timing diagrams are used to explore the	(2)	Horizontally
(C)	Timing diagrams are often used to design	(3)	Vertically
(D)	In timing diagrams the time axis is represented	(4)	Embedded software
(E)	In timing diagrams the lifelines are shown	(5)	UML 2.0

(A)	A-3, B-5, C-1, D-2, E-4	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-3, C-5, D-1, E-4	(D)	A-5, B-1, C-4, D-2, E-3

12.2 Interaction Overview Diagrams

In this second lesson, **Interaction Overview Diagrams**, you will learn to:

- Describe the purpose and concept of interaction overview diagrams
- List and describe the elements of interaction overview diagrams

12.2.1 Purpose and Definition

Interaction overview diagrams show interactions and interaction occurrences.
Interactions depict any type of UML interaction diagram (sequence, timing, activity, communication diagram) or interaction occurrences, which indicate an activity or operation to invoke.
The Interaction Overview Diagram focuses on the overview of the flow of control of the interactions.
It is a variant of the Activity Diagram where the nodes are the interactions or interaction occurrences. We can link and achieve high degree of navigability between diagrams inside the Interaction Overview Diagram.

12.2.2 Elements

Interaction occurrences are references to existing interaction diagrams. An interaction occurrence is shown as a reference frame; that is, a frame with "ref" in the top-left corner.

The name of the diagram being referenced is shown in the center of the frame.

Figure 12.4 shows the ref element.

Figure 12.4: ref Element

12.2.3 Interaction

Interaction elements are similar to interaction occurrences, in that they display a representation of existing interaction diagrams within a rectangular frame.

They differ in that they display the contents of the references diagram inline.

Figure 12.5 depicts interaction.

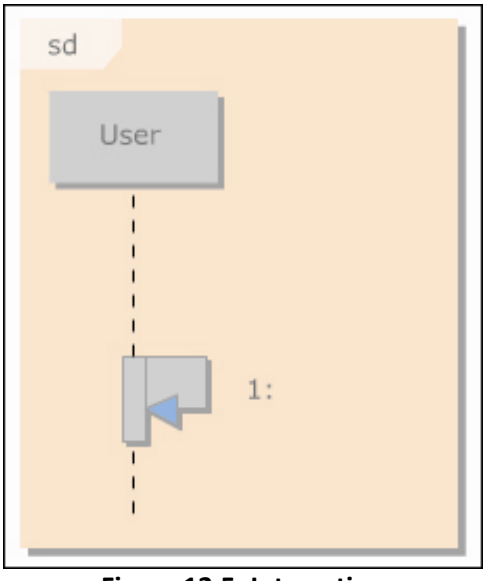


Figure 12.5: Interaction

12.2.4 Initial and Final nodes

These represent the starting point and end point nodes.

Figure 12.6 depicts Initial and Final nodes.

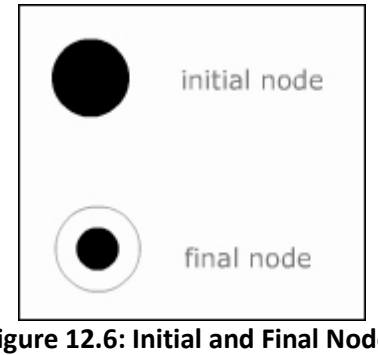


Figure 12.6: Initial and Final Nodes

Knowledge Check 2

1. Match the following descriptions with the corresponding elements.

	DESCRIPTION	ELEMENT	
(A)	Interaction Overview Diagram show	(1)	Interactions
(B)	Interaction depicts	(2)	Flow of control of interactions

(C)	Interaction overview diagram focuses on the overview of	(3)	Interactions & Interaction Occurrences
(D)	Interaction overview diagram is a variant of	(4)	Any type of UML interaction diagram
(E)	In interaction overview diagrams nodes are	(5)	Activity diagram

(A)	A-3, B-4, C-2, D-5, E-1	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-3, C-5, D-1, E-4	(D)	A-5, B-1, C-4, D-2, E-3

2. Match the following descriptions with the corresponding elements.

	DESCRIPTION	ELEMENT	
(A)	Interaction occurrences are references to existing	(1)	Interaction occurrences
(B)	An interaction occurrence is shown as a	(2)	Rectangular frame
(C)	Interaction elements are similar to	(3)	Reference frame
(D)	Interaction elements display a representation of existing interactions within a	(4)	Starting and end point
(E)	Initial and final nodes represent	(5)	Interaction diagrams

(A)	A-3, B-4, C-2, D-5, E-1	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-3, C-5, D-1, E-4	(D)	A-5, B-3, C-1, D-2, E-4

12.3 Composite Structure Diagrams

In this third lesson, **Composite Structure Diagrams**, you will learn to:

- Describe the purpose and concept of composite structure diagrams
- List and describe the elements of composite structure diagrams

12.3.1 Purpose and Definition

A composite structure diagram is an alternative for modeling aggregation and composition relationships.

The composite structure diagram allows the modeler to describe the relationships between elements that work together within a classifier. It is similar to the class diagram, but shows parts and connectors.

The parts are not necessarily classes in the model and they do not represent particular instances, but they may be roles that classes may play.

The classes can be displayed as composite elements exposing interfaces and containing ports and parts.

Figure 12.7 depicts a Composite Structure diagram.

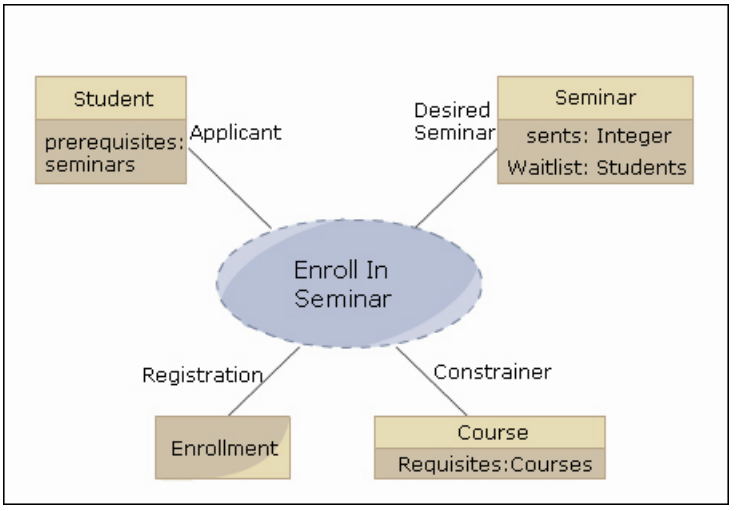


Figure 12.7: Composite Structure Diagram

12.3.2 Part

A part is an element that represents a set of one or more instances that are owned by a containing class instance.

A part is shown as a rectangle within the body of a class or component element. Parts can be connected using connectors that are modeled as straight lines.

Figure 12.8 depicts a Part.

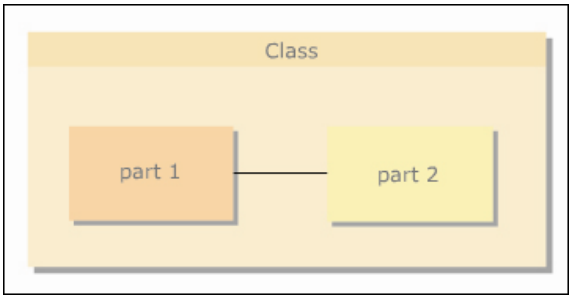


Figure 12.8: Part

12.3.3 Port

A Port defines an interaction point between the object and its environment especially between connectors and objects.

They are modeled as small squares. It appears on the boundary of an object. A port simply provides an interface encapsulating classes.

Figure 12.9 depicts a Port.

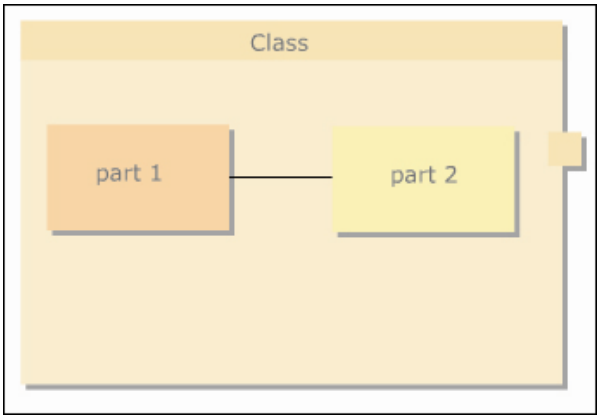


Figure 12.9: Port

12.3.4 Interface

Interfaces have been discussed in class diagrams in detail. An exposed interface of a class can be modeled as either provided or required.

A provided interface is defined by the named interface element and is defined by drawing a realization link between the class and the interface.

A required interface is a statement that the classifier is able to communicate with some other classifier that provides operations defined by the named interface element and is defined by drawing a dependency link between the class and the interface.

Figure 12.10 depicts an Interface.

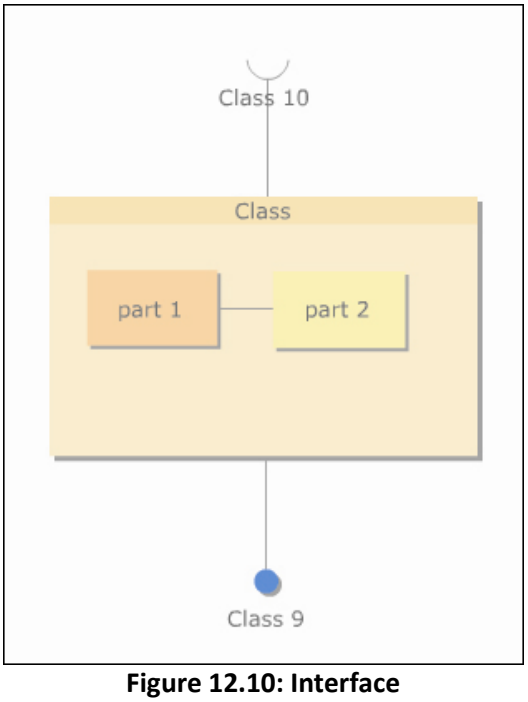


Figure 12.10: Interface

In figure 12.10, “Class 10” is shown as a provided interface and “class 9” is shown as required interface.

12.3.5 Collaboration

Collaboration models elements that represent a particular functionality. The elements may be classes, interfaces, objects, or operations.

A collaboration element is represented by an oval shape. Collaboration shows only the roles and attributes required to accomplish its function.

Figure 12.11 depicts Collaboration.

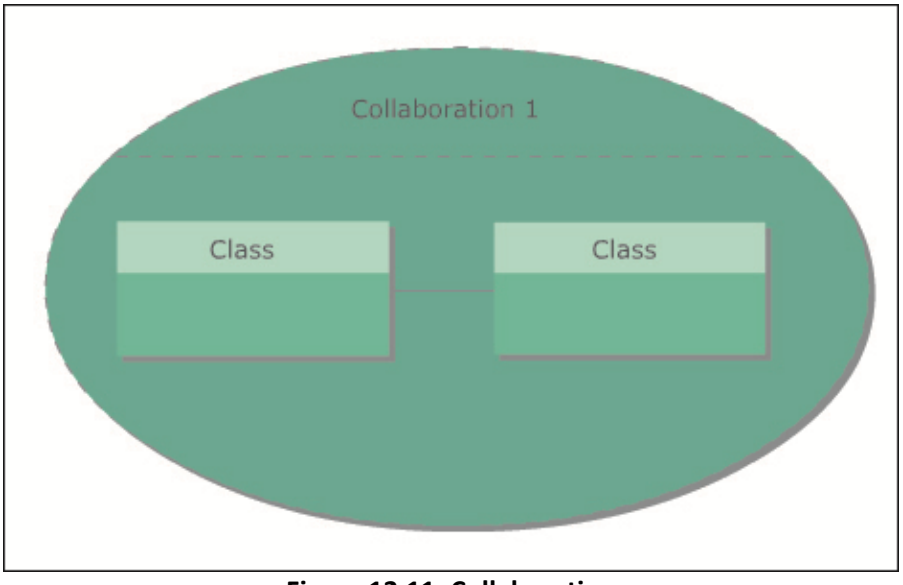


Figure 12.11: Collaboration

12.3.6 Collaboration Occurrence

Collaboration occurrences show instances of collaborations.
Connectors may be drawn from collaboration to classes to show that collaboration represents the classifier.

- It is shown as a dashed line with arrowhead and the keyword «occurrence».

Figure 12.12 depicts Collaboration Occurrences.

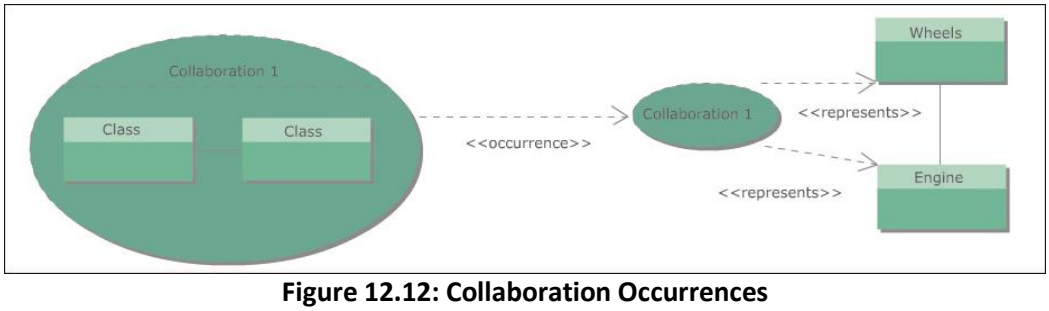


Figure 12.12: Collaboration Occurrences

Knowledge Check 3

1. Match the following descriptions with the corresponding elements.

	DESCRIPTION	ELEMENT	
(A)	A composite structure diagram is an alternative for	(1)	Parts and connectors
(B)	Composite structure diagram allows the modeler to describe	(2)	Classes in the model
(C)	Composite structure diagram shows	(3)	Modeling aggregation
(D)	In Composite structure diagram parts are not necessarily	(4)	Composite elements
(E)	In Composite structure diagram classes can be displayed as	(5)	Relationships between elements

(A)	A-3, B-5, C-1, D-2, E-4	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-3, C-5, D-1, E-4	(D)	A-5, B-3, C-1, D-2, E-4

2. Match the following descriptions with the corresponding element

	DESCRIPTION	ELEMENT	
(A)	Parts can be connected using	(1)	Collaboration
(B)	Ports are modeled as	(2)	An oval shape
(C)	Port appears on the boundary of	(3)	Connectors
(D)	Collaboration element is represented by	(4)	An object
(E)	Collaboration occurrences show instances of	(5)	Small squares

(A)	A-3, B-5, C-1, D-2, E-4	(C)	A-4, B-1, C-2, D-5, E-3
(B)	A-2, B-3, C-5, D-1, E-4	(D)	A-3, B-5, C-4, D-2, E-1

Module Summary

In this module, **New Features of UML 2.0**, you learnt about:

- **New Features**
  The different types of diagrams are Interaction Overview diagram, Composite Structure diagram and Timing diagrams. Interaction overview diagram is a type of activity diagram in which the nodes represent interaction diagram. Composite structure diagram shows the internal structure of a class, component, or collaboration. Timing diagrams are used to explore the behaviors of one or more objects throughout a given period of time.

- **Interaction Overview Diagrams**
  The Interaction Overview Diagram focuses on the flow of control of the interactions. It is a variant of the Activity Diagram where the nodes are the interactions or interaction occurrences. We can achieve a high degree of navigability between diagrams inside Interaction Overview Diagrams.

- **Composite Structure Diagrams**
  A Composite Structure Diagram shows the internal structure of a class, component, or collaboration, including their interaction points to other parts of the system. It shows the configuration and relationship of parts that together, perform the behavior of the containing class.

A Answers to Knowledge Checks

Module 1

Knowledge Check 1

1. (C)
2. (B)
3. (C)
4. (B)
5. (D)

Knowledge Check 2

1. (D)
2. (B)
3. (D)

Knowledge Check 3

1. (B)
2. (D)
3. (B)

Knowledge Check 4

1. (C)
2. (B)
3. (D)

Module 2

Knowledge Check 1

1. (D)

Knowledge Check 2

1. (C)

Knowledge Check 3

1. (C)
2. (A)

Module 3

Knowledge Check 1

1. (A)
2. (B)
3. (C)

Module 4

Knowledge Check 1

1. (D)
2. (B)
3. (C)
4. (D)
5. (D)
6. (D)

Knowledge Check 2

1. (C)
2. (A)
3. (C)
4. (D)
5. (C)

Module 5

Knowledge Check 1

1. (C)

2. (D)

3. (B)

4. (A)

Knowledge Check 2

1. (B)

2. (D)

3. (A)

Module 6

Knowledge Check 1

1. (D)

2. (D)

Knowledge Check 2

1. (B)

Module 7

Knowledge Check 1

1. (B)

2. (D)

Knowledge Check 2

1. (B)

2. (D)

Knowledge Check 3

1. (B)

Module 8

Knowledge Check 1

1. (A)

Knowledge Check 2

1. (D)

Module 9

Knowledge Check 1

1. (D)
2. (C)

Knowledge Check 2

1. (D)
2. (A)

Module 10

Knowledge Check 1

1. (B)

Knowledge Check 2

1. (B)
2. (C)
3. (D)
4. (C)

Module 11

Knowledge Check 1

1. (D)
2. (A)

Knowledge Check 2

1. (A)

2. (B)

Knowledge Check 3

1. (C)

2. (A)

Module 12

Knowledge Check 1

1. (C)

2. (A)

3. (D)

Knowledge Check 2

1. (A)

2. (D)

Knowledge Check 3

1. (A)

2. (D)