

# Object-Oriented Programming



# Object-Oriented Programming

By

Amany Fawzy Elgamal

**Cambridge  
Scholars  
Publishing**



Object-Oriented Programming

By Amany Fawzy Elgamal

This book first published 2024

Cambridge Scholars Publishing

Lady Stephenson Library, Newcastle upon Tyne, NE6 2PA, UK

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

Copyright © 2024 by Amany Fawzy Elgamal

All rights for this book reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owner.

ISBN (10): 1-5275-6425-8

ISBN (13): 978-1-5275-6425-1

# TABLE OF CONTENTS

Chapter One.....	1
Introduction	
Chapter Two .....	6
UML	
Chapter Three .....	43
Object-Oriented Programming Concepts	
Chapter Four.....	92
Inheritance	
Chapter Five .....	109
Encapsulation	
Chapter Six .....	124
Polymorphism	
Bibliography .....	150



# CHAPTER ONE

## INTRODUCTION

In Structured Programming, a program is comprised of a set of instructions. Small programs are generally easy to understand, but larger ones, containing hundreds of statements, can be difficult to comprehend unless they are divided into smaller parts. Consequently, functions and procedures are utilized to make these programs easier to read and understand.

The standard approach to designing software systems involves identifying the problem, and then developing a set of functions that can accomplish the required tasks. If these functions are too complex, they are further broken down until they become simple enough to understand. This process is known as functional decomposition.

Data is usually stored in a database of some sort, or it might be held in memory as global variables.

Consider a simple example: a training center system that stores data about students and teachers, and can record information about available courses. This system also tracks each student and the courses in which they are enrolled. A potential functional design for this system would involve writing the following functions:

**add\_student**

**enter\_for\_exam**

**check\_exam\_marks**

**issue\_certificate**

**expel\_student**

Furthermore, a data model is needed to store information about students, trainers, exams, and courses, requiring the design of a database schema to hold this data, as shown in Figure 1.1.

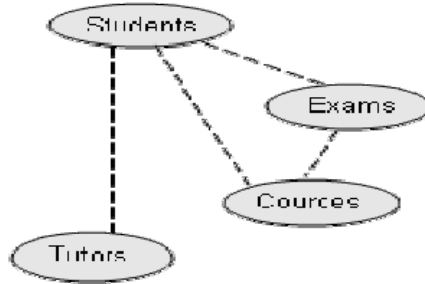


Figure 1.1: A database schema for a training center system

These functions depend on this data. For instance, the “add\_student” function will modify the “students” data, and the “issue\_certificate” function needs to access the student's data to know the details of the student who needs the certificate, as well as the exam data. Figure 1.2 illustrates this data and the functions associated with it, showing the dependency lines.

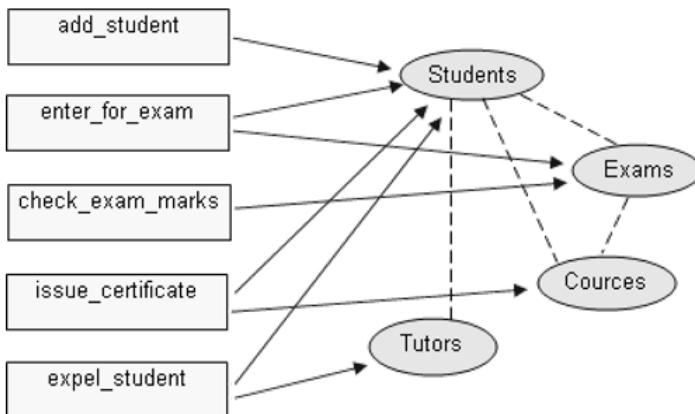


Figure 1.2: Dependency lines between functions and data.



Problems arise when things become more complex, as the difficulty of maintaining the system increases.

For example, if we find that storing a student's date of birth as a two-digit number to represent the year was a bad idea, the simple solution is to change the birthdate field in the Students' table from two to four digits for the year number. However, this change may cause unforeseen side effects. The data for exams, courses, and tutors all depend on the data in the Students' table, so we may inadvertently disable the `add_student`, `enter_for_exams`, `issue_certificate`, and `expel_student` functions. The "add\_student" function will certainly not work as it expects the year of birth in the form of a two-digit number.

To solve problems facing programmers, such as the one mentioned above, the concept of Object-Oriented Programming (OOP) is adopted. Two related issues are the absence of any restrictions on functions accessing global data in the program, and the lack of a complete connection between the functions and data, which is at odds with practical reality.

Consider a store management program. One piece of global data is the variable used to store the set of items in the store. Different functions in the program access this variable to insert a new item, display existing items, or modify an item. In a procedural program, there are usually two basic types of data or variables: local and global. Local data is specific to each function, accessible and used solely within its function, making it hidden inside this function.

In the store program, for example, the `Display()` function uses local data that is completely linked to its function and cannot be modified through any other function. If two or more functions share the same data, then this data must be global, where any function within the program can access it.

Managing many functions and much public data (variables) can lead to an increase in errors resulting from accessing those variables, where one of the functions may accidentally modify one of the global variables.

Another issue is the difficulty of understanding the overall structure of the program, as well as the challenge of modifying the program. Any change in a global variable may require modifying all functions that use this variable. In the store program, for example, if we want to change the product code from a 5-digit to a 12-digit number, which requires changing the data type from short to long, we must modify all functions that use this

variable to handle data of type long instead of short. Thus, separating data from functions is problematic, which is inconsistent with practical reality.

In our daily life, we deal with objects such as people, cars, and so on, which are not merely functions or data, but entities that contain attributes and behaviors.

Attributes are the various elements associated with an object that uniquely define it. For example, in the case of people, eye color, height, and job, and in the case of cars, horsepower, and the number of doors, are all attributes equivalent to data within the program.

Behavior refers to the operations that an entity performs in real life in response to certain events. This behavior is equivalent to the function inside the program.

Neither data alone nor functions alone are sufficient to represent objects in real life as efficiently as desired. The fundamental idea of OOP languages is to group the data and functions into a single unit called an object. This way, we can effectively model real-world objects in our programs. An object-oriented design for the training center system mentioned earlier might involve objects for Students, Courses, and Exams, each with its own data and functions. The Student object would have data like name and birthdate and functions like `enroll_in_course()`. This encapsulation of data and functions into objects leads to more intuitive, maintainable, and flexible programs.

In the context of our previous store management program example, each item in the store could be represented as an object, with attributes such as name, price, and quantity, and methods such as `restock()` or `sell()`. The item's code could be changed from a 5-digit to a 12-digit number simply by modifying the Item class, without the need to change every function that uses this item code.

Thus, in OOP, the data and the operations that can be performed on it are combined into a single entity, making it easier to understand how a program works. Additionally, OOP provides several benefits such as encapsulation, inheritance, and polymorphism that make it easier to design, implement, and manage software systems.

A program written in the OOP style differs from a program written in a structured style in two ways.

### 1- Program building unit:

The program written in the structural style has a main programming unit and a group of sub-units. The sub-units are called from the main unit. As for the program written in the OOP style, the program unit becomes the class, which is a template for defining objects.

### 2- Dealing with data:

Where in the structural program, the programmer's effort is focused on the program code, that is, the lines of the program that handle the flow of operations, while the view of the data is a secondary view. OOP considers the data an important part of the program. Thus, we have a library not only of functions, but a library of classes that contain data and functions.

In summary, while structured programming is a powerful tool for creating simple to moderately complex programs, object-oriented programming offers a more flexible and intuitive approach for developing complex software systems. However, the best approach depends on the specific needs and constraints of the project at hand.

In the rest of this book, we will cover the details of OOP concepts.

# CHAPTER TWO

## UML

Unified Modeling Language (UML) is a visual modeling language that provides developers (working in the field of analysis and design of object-oriented systems) with diagrams and methods to visualize, create, and document software systems, and to model the organizations that use these systems. Note that it is not a way to build or develop systems; it will not direct them on what to do first and what to do next. It will not tell you how to design the system but it will help you with modeling.

Several participants participate in the development of software systems, each with a role, for example:

- Analysts
- Designers
- Programmers
- Testers

Each of them is interested in different aspects of the system, and needs a different level of detail. For example, the programmer needs to understand the design of the system in order to convert it into code. The system analyst is concerned with the behavior of the system as a whole.

UML attempts to provide a highly expressive language so that participants can take advantage of system diagrams. It has become the approved language for modeling software operations after it has been widely accepted by those interested in building software. This language provides a simplified symbolic means to express various models of software through which it is easy for analysts, designers, programmers, and clients to communicate with each other and pass information in a concise, unified format. This aspect is similar to building schemes in architecture or electrical circuit diagrams that can be understood and dealt with among workers in the field.

Between 1989 and 1994, there were a large number of software modeling languages, each with its own symbols and rules, and each language had

elements similar to the elements of other languages without an integrated language that satisfies the software needs. Until the mid-nineties, three methodologies appeared. Each methodology has its own strengths and these methodologies are:

1. Booch's methodology is characterized by quality in terms of design and implementation. Grady Booch worked extensively on the Ada language, and had a major role in the development of Object Oriented techniques for the language. Despite the strength of Booch's methodology, the symbols in it were not well accepted.
2. Object Modeling Technique (OMT), which is the best technique in analyzing information systems with big data.
3. OOSE (Object Oriented Software Engineering) methodology is a powerful tool as a good way to understand the behavior of an entire system.

In 1994, Jim Rumbaugh, founder of OMT, left General Electric and joined Booch to work for Rational Corp. The purpose of the collaboration was to integrate their ideas and pour them into a unified methodology. Then the creator of OOSE, Ivar Jacobson, also joined Rational, and his ideas were included in the unified methodology called the Unified Modeling Language.

**The following table shows the different versions of the language:**

**Table 2.1**

Version	Date	Description
1.1	11-1997	The OMG Object Management Group (OMG™) adopts UML 1.1 proposal.
1.3	03-2000	Contains a number of changes to the UML metamodeling, semantics, and notation, but should be considered a minor upgrade to the original proposal.
1.4	09-2001	Mostly "tuning" release but not completely upward compatible with the UML 1.3. Addition profiles as UML extensions grouped together. Updated visibility of features. Stick arrowhead in Interaction diagrams now denote asynchronous call. Model element may now have multiple stereotypes. Clarified collaborations.
		Refined definitions of components and related concepts. <u>Artifacts</u> were added to represent physical representations of components.

1.5	03-2003	Added <u>actions</u> – executable actions and procedures, including their run-time semantics, defined the concept of a data flow to carry data between actions, etc.
1.4.2	01-2005	This version was accepted as ISO specification (standard) ISO/IEC International Organization for Standardization (ISO) and the International Electro technical Commission (IEC).
2.0	08-2005	New diagrams: object diagrams, <u>package diagrams</u> , <u>composite structure diagrams</u> , interaction overview diagrams, timing diagrams, <u>profile diagrams</u> . Collaboration diagrams were renamed to <u>communication diagrams</u> . <u>Activity diagrams</u> and <u>sequence diagrams</u> were enhanced. Activities were redesigned to use a Petri-like semantics. Edges can now be contained in partitions. Partitions can be hierarchical and multidimensional. Explicitly modeled <u>object flows</u> are new. Classes have been extended with internal structures and <u>ports</u> (composite structures). Information flows were added. New notation for concurrency and branching using combined fragments. Notation and/or semantics were updated for components, realization, deployments of artifacts. Components can no longer be directly deployed to <u>nodes</u> . <u>Artifacts</u> should be deployed instead. Implementation has been replaced by « <u>manifest</u> ». Artifacts can now manifest any package able element (not just components, as before). It is now possible to deploy to nodes with an internal structure. New meta classes were added: connector, collaboration use, connector end, <u>device</u> , deployment specification, <u>execution environment</u> , accept event action, send object action, structural feature action, value pin, activity final, central buffer node, data stores, flow final, interruptible regions, loop nodes, parameter, <u>port</u> , behavior, behavioral classifier, duration, interval, time constraint, combined fragment, creation event, destruction event, execution event, interaction fragment, interaction use, receive signal event, send signal event, extension, etc. Integration between structural and behavioral models was improved with better support for executable models.

2.1	04-2006	Minor revision to UML 2.0 – corrections and consistency improvements.
2.1.1	02-2007	Minor revision to the UML 2.1
2.1.2	11-2007	Minor revision to the UML 2.1.1
2.2	02-2009	Fixed numerous minor consistency problems and added clarifications to UML 2.1.2
2.3	05-2010	Minor revision to the UML 2.2, clarified <u>associations</u> and association classes, added <u>final classifier</u> , updated <u>component diagrams</u> , composite structures, actions, etc.
2.4.1	08-2011	The current version of UML, a minor revision to UML 2.3, with a few fixes and updates to classes; packages - added <u>URI package attribute</u> ; updated actions; removed creation event, execution event, send and receive operation events, send and receive signal events, renamed destruction event to <u>destruction occurrence specification</u> ; <u>profiles</u> – changed stereotypes and applied stereotypes to have upper-case first letter – «Metaclass» and stereotype application.
<u>2.5 FTF - Beta 1</u>	11-2012	From the UML language perspective, 2.5 will be a minor revision to UML 2.4.1, while they spent many effort simplifying and reorganizing specification documents. It seems that this time "professors" took over researchers and industry practitioners and instead of fixing actual UML issues waiting to be resolved for several years, they re-wrote UML specifications "to make them easier to read". For example, they tried "to reduce forward references as much as possible", which is an obvious requirement for textbooks but not for specifications. There will be no separate UML 2.5 Infrastructure document, so that the UML 2.5 specification will be a single document. <u>Package merge</u> will no longer be used within the specification itself. Four UML compliance levels (L0, L1, L2, and L3) are to be eliminated, as they were not useful in practice. UML 2.5 tools will have to support complete UML specifications. <u>Information flows</u> , <u>models</u> , and <u>templates</u> will no longer be auxiliary UML constructs.

2.5 RTF - Beta 2	09-2013	Work in progress – some fixes, clarifications, and explanations added to UML 2.5 FTF – Beta 1. Updated description for multiplicity and multiplicity elements. Clarified definitions of aggregation and composition. Finally fixed a wrong «instantiate» dependency example for Car Factory. Introduced notation for inherited members with a caret “^” symbol. Clarified feature redefinition and overloading. Moved and rephrased definition of qualifiers. Few clarifications and fixes for stereotypes, state machines, activities. Protocol state machines are now denoted using «protocol» instead of {protocol}.
		Use <u>cases</u> are no longer required to express some needs of actors and to be initiated by an actor.
2.5	08-2015	UML was made simple than it was before. Rapid functioning and the generation of more effective models were introduced. Outdated features were eliminated. Models, templates were eliminated as auxiliary constructs.
2.5.1	12-2017	The current UML standards call for 13 different types of diagrams: class, activity, object, use case, sequence, package, state, component, communication, composite structure, interaction overview, timing, and deployment.

## 2.1 UML Diagrams

UML consists of a set of diagrams representing the parts of a system. These diagrams are divided into two main categories:

1- **Structure Diagrams, which** represent the architectural components of the system such as objects, relationships and the dependence of system elements on each other.

They include Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.

**Class Diagram:** A diagram shows the classes within the system and the relationships between them.

**Package Diagram:** It breaks the system down into smaller and easier-to-understand parts, and this diagram allows us to model these parts.



**Component Diagrams:** to encode how the system is separated or partitioned and how each model depends on the other. It focuses on the actual components of the program (files, link libraries, executables, and packages) and not on the logical or intellectual separation as in the Package Diagrams.

2- **Behavior Diagrams**, which represent the dynamics of the system and the interaction between the system's objects and the changes that occur in the system. They include: Use Case diagrams, Activity diagrams, State Machine, Communication diagrams, Sequence diagrams, Timing diagrams, and Interaction Overview diagrams.

**Use Case Diagram:** A description of the behavior of the system from the user's point of view. It describes interaction with the outside world; it is useful during the analysis and development stages, and helps in understanding the requirements.

**Collaborative diagrams:** to describe how the objects we build collaborate and use numbers to show the sequence of messages. Collaborative diagrams are also referred to as Communication diagrams in UML.

**Sequence Diagram:** describes how objects in the system interact over time. It displays the time sequence of the objects participating in the interaction. It consists of a vertical dimension (time) and a horizontal dimension (different object).

**State Diagrams:** to show the succession of transitions between different states of an organism during its life cycle and the events which force it to change its state. **Activity Diagram:** shows a special case where most of the states are states and actions and most transitions are triggered by the completion of the verbs of the source states. This diagram focuses on internal processing-driven flows.

There are different programs to create, document and track these diagrams and generate source code for files for several languages, which facilitates the programming process, and these programs include: Edraw UML Diagram, With Class2000, Rational Rose, Model Maker, and you can draw the diagrams through several web sites. You can rely on these diagrams in Object Oriented systems, but they are not suitable for some applications, such as Networking Projects, Web applications, and database design.

## 2.2 System modeling

**When modeling any Object Oriented system, we deal with three main parts:**

- **Functional Model:** It is a form that depends on the user's point of view and includes use case diagrams.
- **Object Model:** It is the model that is concerned with the architecture of the system and sub-systems using objects and attributes, operations and associations. This model includes a class diagram.
- **Dynamic Model:** It is the model that is concerned with the internal behavior of a system and includes Sequence Diagrams, Activity Diagram, and State Diagram.

## 2.3 Use Case Diagram

A powerful UML tool, it is a description of a set of interactions between a user and a system. By building a set of use case diagrams, we can describe the system clearly and concisely. The two main components of a use case diagram are the use case and the actor. Use case describes the interaction between the user and the system. Use cases are usually described using combinations of (verb / noun) – e.g.: ‘pay bills’, ‘update salary’ or ‘create an account’. The actor is the role that the user plays in the system, whether that user is a human or even another system that the system being modeled will interact with.

**UML provides simple notation to represent the use case and the actors as shown in figure 2.1.**

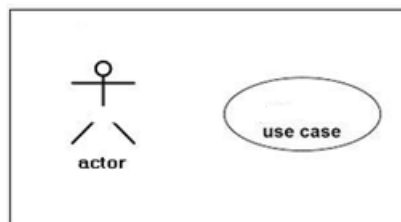


Fig. 2.1 Actor and use case

The player will activate or initiate a particular use case. For example, if we are developing a banking system, and we have a use case called “withdraw cash”, we need customers to be able to withdraw that cash as shown in figure 2.2.

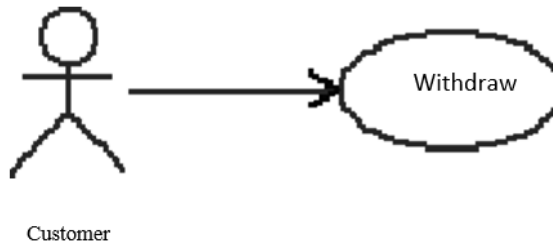


Fig. 2.2 withdraw cash use case

For most systems, a single actor can interact with a range of use cases, and more than one different actor can activate a single use case. This brings us to an integrated use case diagram. For example, a patient calls a doctor’s office to book an appointment for an examination as shown in figure 2.3.

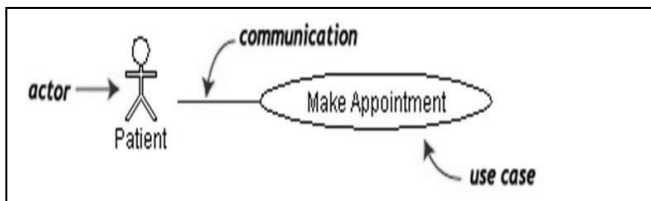


Fig. 2.3 Make appointment

Therefore, the person working in the inquiries searches the appointment book to search for an empty appointment and set the appointment.

**The full schema is shown in figure 2.4. It includes use cases for the following:**

- Dealing with the patient and making or canceling an appointment with the doctor’s schedule organizer.
- Interaction between patient and doctor requesting treatment.
- Dealing with patient to pay the account.

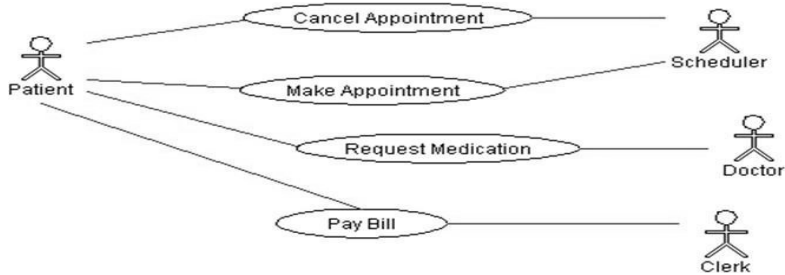


Fig. 2.4 Full Shema

**System Boundaries:** It is a square that represents the boundaries of the interaction between actors and the system, as it groups all use cases together in one place.

### 2.3.1 Relationships in Use Case

1. Generalization: means that one is a type of another. It is represented as follows in figure 2.5:

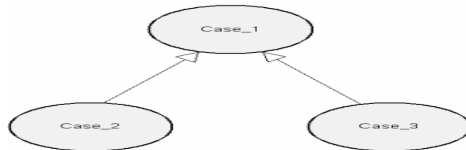


Fig. 2.5 Generalization

2. Includes: means that one is an evocation of another. It is represented as shown in figure 2.6:

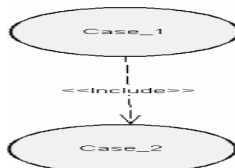


Fig. 2.6 Include

3. Extend: means that one is a variation of another. It is represented as shown in figure 2.7:

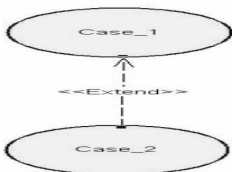


Fig. 2.7 Extend

The following figure (2.8) shows the use case diagram including these relationships.

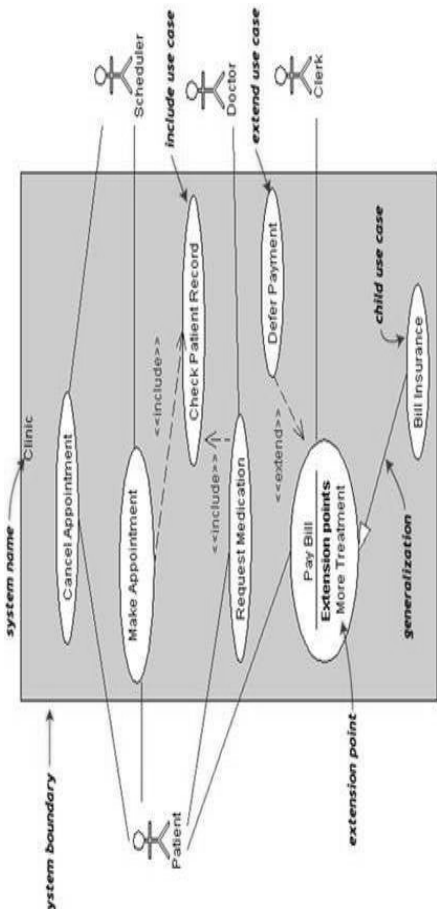


Fig. 2.8 use case with relationships

The following figure (2.9) represents the use case diagram for part of an E Learning system:

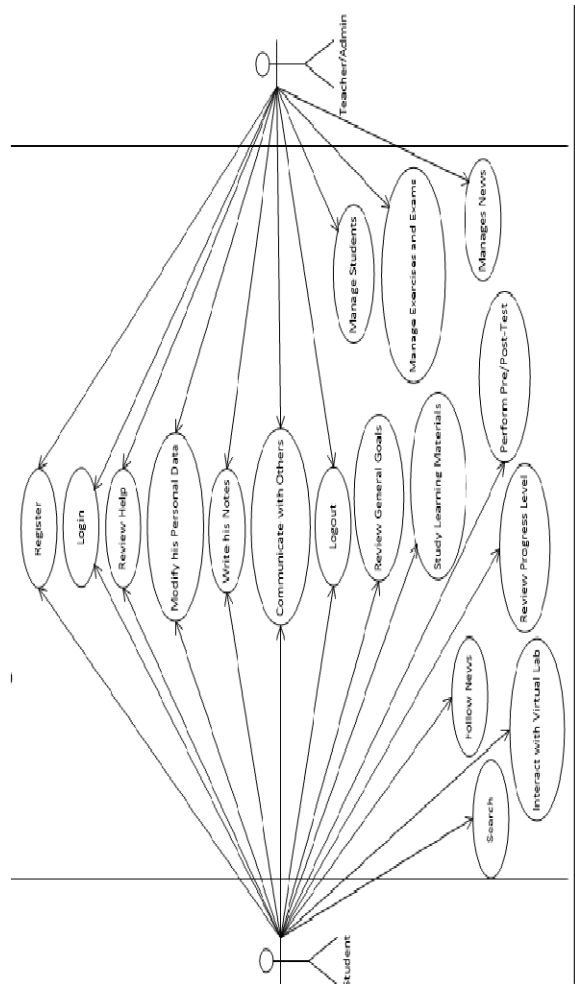


Fig 2.9 Use case for E Learning system

### 2.3.2 Purpose of use case

- Use cases can provide a structured way to describe user interactions with the system, their effectiveness in achieving this goal depend on how well they are written and how well they capture user needs.
- Use cases are a tool for describing the functionality of the system from the perspective of its users, but they do not provide a complete specification of the system. There may be other requirements or constraints that are not captured in the use cases, such as performance requirements, security requirements, or data constraints.
- They enable communication between customers and developers (as long as the scheme is that easy, everyone can understand it).
- Use cases guide members of development teams through the development processes.
- Provide a methodology for planning, and allow us to estimate the time needed to complete the work.
- Provide the basis for system design and test.
- Finally, use cases help in inform user documentation.

Use case diagrams are used in almost all projects, they help to clarify and show requirements and project planning. In the preliminary stage of the project, most use cases must be defined, and during the project's progress, there may be a need to define other use cases.

Note that each use case is associated with at least one of the actors. The actor may be a person, another computer system, or an event for whom or what will use the system. The actor does not represent these things, but represents their roles. This means that when these objects interact with the system in multiple ways, one actor does not represent them. For example, a person who works in a customer service center by telephone and receives orders from the customer must be represented twice as a “Support Staff” actor and a “Sales representative” actor.

**Figure 2.10 shows an example illustrating the use case diagrams for a library system.**

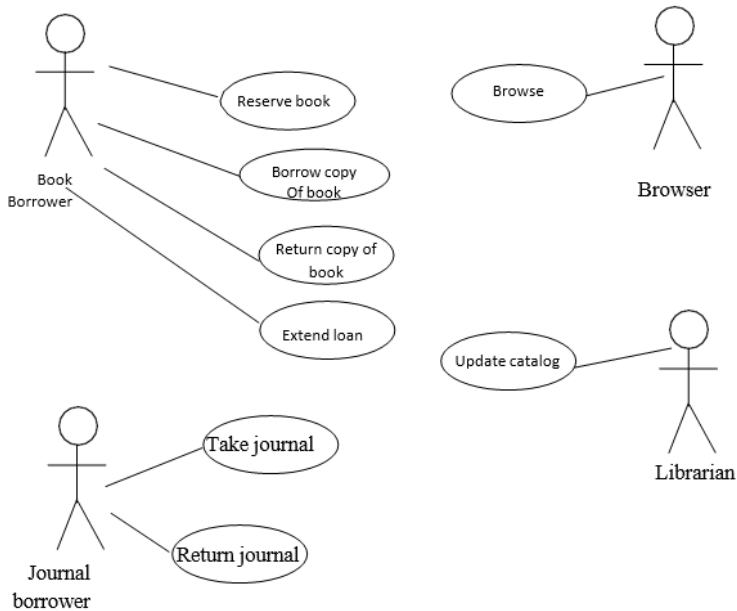


Fig. 2.10 Use cases for a library System

## 2.4 Class diagram

This diagram represents the classes included in the system and their relationships. It is referred to as a static diagram because it describes the classes with their properties, methods, and their static relationships with each other, but it does not display the interactions between them.

### Class representation in the Class Diagram

The classes are represented in this diagram as shown in figure 2.11.



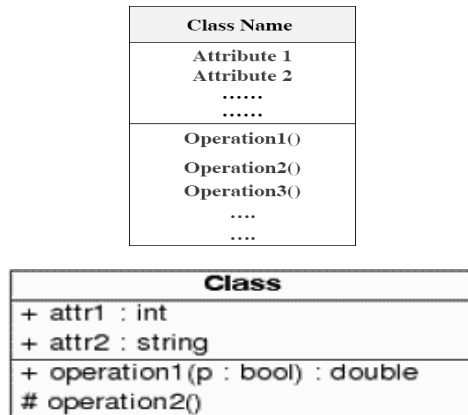


Fig. 2.11 Class diagram

A class is represented by a rectangle divided into three parts. The first part displays the class name (use the singular form for the class name, which expresses a general case such as a student or book). The class name indicates the function it performs. The second part includes all the attributes (or properties) that characterize our class. The last part contains all the methods or operations of the class. Figure 2.12 represents examples of class diagrams.

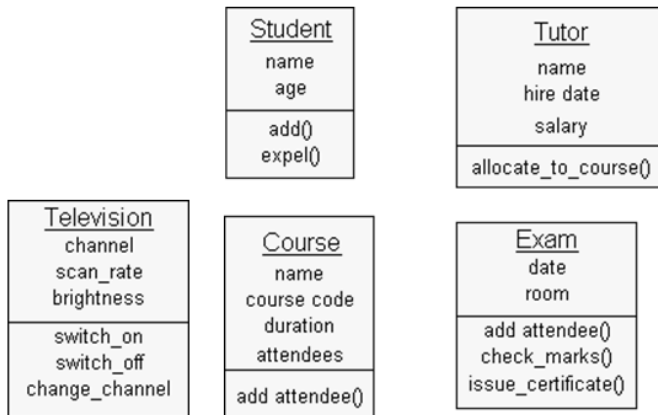


Fig. 2.12 Examples of class diagram

## Scope

The following symbols are used to define the scope of attributes or methods. These symbols show us the accessibility of the attribute or method within the class. They are known as modifiers:

- "+" stands for public.
- "#" stands for protected.
- "-" stands for private.

Public (+): Any element of the class or outside of the class can access this member (attributes or operation). Private (-): The member can only be accessed within the class itself. Protected access specifier (#): The property can only be accessed within the class or from another class that inherits this class.

The following example shows a class diagram for a class called 'Vehicle' that includes an integer property for the number of vehicles, a string property to hold the vehicle type, and two integer properties for the vehicle's maximum and minimum speed. Operations or methods of this class include a set of functions such as a process to create the vehicle, another to destroy it, a process to increase its speed, and another to move backwards as shown in Figure 2.13.

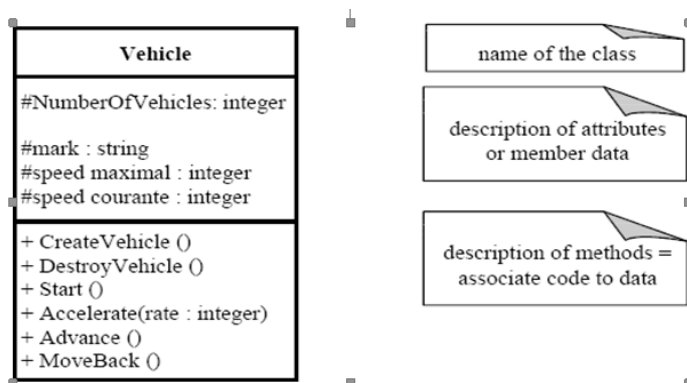


Fig. 2.13 Vehicle class

The accurate description of classes within the system is one of the most important skills in the development of Object-Oriented Systems (OOS). To perform this task, you can use a technique called Noun Identification

Technique (NIT). It uses the System Requirement Specification (SRS) document and extracts the nouns and verbs (nouns represent classes and verbs express operations or methods). We will introduce this in the following example.

In the library system, we can use the following SRS document to define the classes in the system.

### **Books and Journals**

The library contains a number of books and journals. The library can contain several copies of the same book. Some of the books are for short-term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals. The borrowing system must keep track of borrowing duration, enforcing the rules described above.

To define the classes in the borrowing system, we can omit the word “library” because it is outside the scope of the system’s classes. Also, a “short term loan” where it represents an event not a class. The word “week” is omitted because it represents a time measuring unit, also “item” because it is ambiguous, as it represents a book or newspaper, also “time” and “system” because they are outside the scope of the system, as well as the word “rules”. Thus, the classes are:

- Book
- Journal
- Copy (of book)
- Library member
- Member of staff

### **2.4.1. Relationships between classes**

Class diagrams represent classes and the relationships with each other. The relationships between classes take one of four types, which are illustrated in the following figure (2.14).





Relationship	
Generalization (inheritance)	 "is a" "is a kind of"
Association (dependency)	 "Who does What" "uses"
Aggregation	 "has" "composed of"
Composition: Strong aggregation	

Fig. 2.14 Type of relationship

The following figure (2.15) illustrates examples of these relationships.

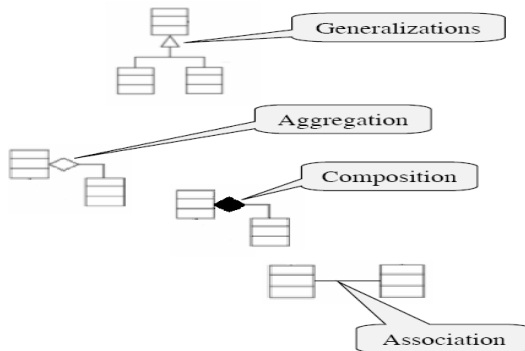


Fig. 2.15 Examples of relationships

#### 2.4.1.1 Generalization/ inheritance

This relationship is used when two classes are similar with some differences. For example, when we have a general class such as employee and another class is a special case of this class such as engineer, this means that the class of engineer has all the characteristics and operations of the employee class and acquires the properties and operations that belong to it. In this case, the employee class is the base class, while the engineer class is the derived class. This relationship is represented in the diagram by an arrow with a triangle head indicating the inherited class and its beginning from the inherited class, as shown in figure 2.16.

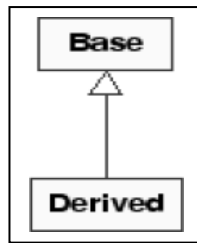


Fig. 2.16 Generalization

**Note: The derived class contains all the properties of the base class except the private members.**

The following figure (2.17) shows a diagram of a base (main) class representing the employee, including two derived classes representing managers and programmers. In addition, a manager class is a main class from which two classes are derived, namely, the project manager and department manager.

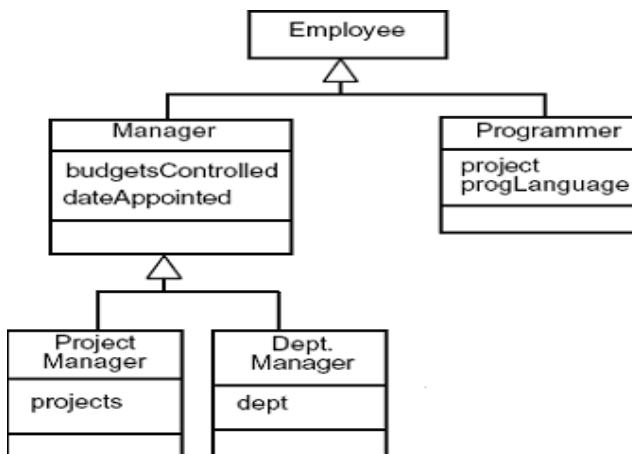


Fig. 2.17 Employee base class with two levels of inheritance

The following example shows a main class for graphic objects and then deriving two classes for line and circle, as shown in figure 2.18.

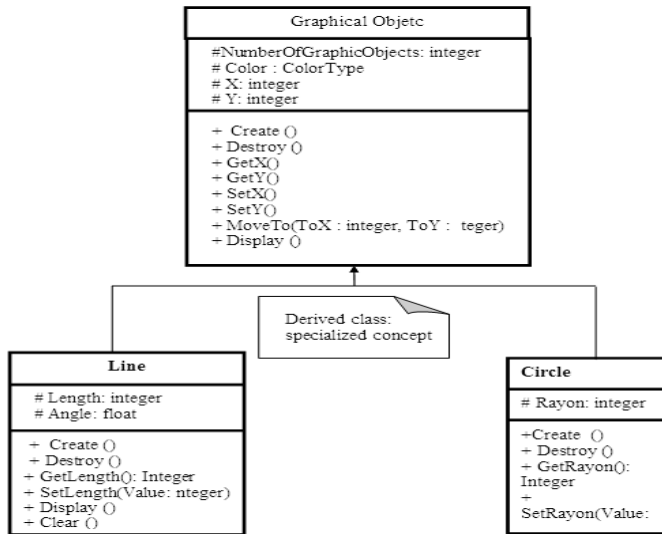


Fig. 2.18 graphical class with two sub classes

The following example represents a main class of vehicles, including derived classes representing helicopters, wagons, and ships. Note writing the name of the main class in italics as shown in figure 2.19, which expresses that it is an abstract class, meaning that there are no physical objects of this class.

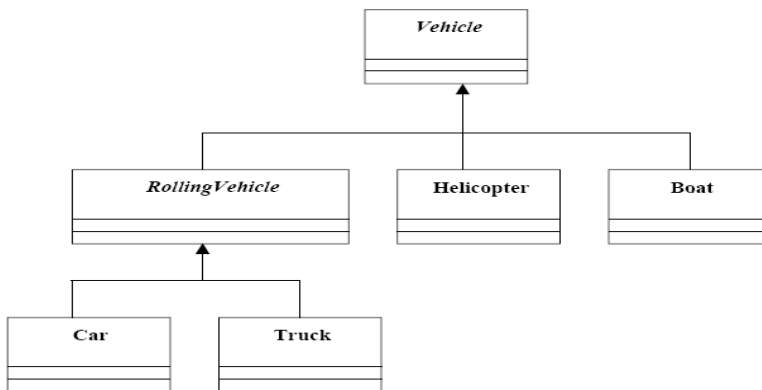


Fig. 2.19 Vehicle class