GOMBE STATE UNIVERSITY
FACULTY OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

COSC 307: OBJECT ORIENTED ANALYSIS AND DESIGN (OOAD)

Credit Units: 3

Course lecturer: Mamudu, Friday

Purpose of the Course

For acquiring skills in Object Oriented analysis and design systems.

Learning Outcomes

By the end of this course, learner should be able to:

i. Apply the basic principles of Object-Oriented Design including the roles, responsibilities and collaborations of classes in an object-oriented systems.
ii. Use the UML design tools such as: Use Case Diagrams, Sequence Diagrams, State Diagrams and Class Diagrams
iii. Explain the basics of object-oriented analysis and design with the use of UML.

Course Content

- Introduction – Scope, Objects, Development Paradigms, Development Phases.
- Introduction to analysis – Purpose, Models, Process.
- Object States – Instances, Classes, Attributes, Attribute Features, Constraints, Identifying Objects and Classes, Objects Relationships, Collections, Identifying Relationships.
- Object Dynamics – Describing Behavior, Transition Networks, Reducing Complexity, Object Interaction, Transitions, Sending and Receiving, Events Interaction Notations.
- Class Relationships – Property Inheritance, Subclasses, Multiple Inheritance, Sibling Relationships, Set Operations, Inheritance of Relations.
- Subclasses and Instances – Meta classes, Parametric Instances.
- Introduction to Design – Continuity, Transformation, Design Phases, Design Criteria, Managing Design.
- Attributes in Design – Defining Attributes, Concrete Attributes, Views, Exports, Composition and Inheritance.
- Relationships in Design – Designing Transitions, Interaction Designs, Dispatching, Coordination, Cluster Objects.
- Designing Passive Objects – Transformations. Storage Management. Passive objects in Java, Performance Optimization.

Introduction:

DEFINITION: Object Oriented Analysis and Design (OOAD) is an engineering approach that models a system as a set of inter-related objects.

Objects and Classes

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

Object

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

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

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

Class

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

The constituents of a class are −

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

In computer programming, *Objects*, or more precisely, the *classes* used to construct objects, are intended to be *reusable* software components.

There are string objects, output objects, input objects, graphics objects, date objects, time objects, audio objects, video objects, etc.

Almost any noun can be reasonably represented as a software object in terms of attributes (e.g., name, color and size) and behaviors (e.g., calculating, moving and communicating).

Using a modular, object-oriented design and implementation approach can make software-development more productive. Object-oriented programs are often easier to understand, correct and modify.

An Object-Oriented Car Analogy

In the real-world, there are many analogies of objects. Let us take a car as an example. A car is an object that can be used to transport people and goods. A car has an engine, four wheels, a number of doors, and a lot of other mechanical and electrical parts. These parts are hidden under the hood. A driver controls the car using a steering wheel, the gas and brake pedals, and many buttons and switches on the dashboard. If you are driving a car and want to make it go faster, you press its gas pedal with your foot.

The specific car you drive is one of many cars of a particular model that were made by a car factory according to a specific design or blueprint. The blueprint of the car may contain a set of engineering drawings and documents that specify the details of building the car. Specifically, the blueprint will include the steering wheel, gas and brake pedals, etc.

The gas pedal hides from the driver, or the user of the car, the complex mechanisms that actually make the car go faster, the brake pedal hides the mechanisms that slow the car, and the steering wheel hides the mechanisms that turn the car.

This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily and safely.

To be able to use a car to travel, the car has to be made by the factory.

In computer software, an object is used to keep and modify information. It is also made according to a blueprint and used by a user (which is typically another program). In the following, we explain a set of object-oriented programming concepts by drawing analogy to a car.

Methods and Classes

In Java, a class is a blueprint of a set of objects. It describes how the object is made of and what tasks the object is able to perform.

The performing of a task by a Java object is by a *method*. The method contains a sequence of Java statements that actually perform the task.

A method hides these statements from its user, just as the gas pedal of a car hides from the driver the mechanisms of making the car go faster.

In Java, a *class* is a programming module containing the methods that perform the class's tasks. Thus, the class is similar in concept to a car's design, which specifies the "methods" to accelerate, steer, brake, and so on.

Instantiation

Just as a factory has to construct a car from its blueprint before you can actually drive a car, a Java program must ask the Java compiler to construct an object of a class before it can make method calls to the object. This is called the instantiation of an object, and hence an object is referred to as an *instance* of its class. Technically, the instantiation of a Java object will allocate memory for the object to hold information and identify code sections that implement the methods.

Reuse

Just as a car's blueprint can be reused many times to build many cars, a Java class can be reused many times to build many objects. *Reusable classes* save programming time and effort and helps you build more reliable and effective systems. This is because existing classes often have gone through extensive testing, debugging and performance tuning. Just as the notion of interchangeable parts was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology. The reuse of a class is not limited to the original design. A class can be extended to add new constructs or modified to perform new tasks.

Method Calls

When you drive a car, pressing its gas pedal sends a message to the car and makes it to perform a task, namely, to go faster. Similarly, when using a Java object, you make *method calls* to an object. A method call sends a message to the object to perform a task by executing the method's statements. The message may include additional information that is needed to perform the task.

Attributes and Instance Variables

A car has many attributes, such as color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading), etc. These include the physical parts as well as the information about the parts. The car's attributes are specified in its blueprint. Every car has its own attributes. Each car knows how much gas is in its own gas tank, but not how much is in the tanks of other

cars. A Java object also has attributes. These attributes are information items that the object keeps while it is used in a program. The object keeps these attributes in its assigned memory which is organized like a C struct. Attributes are specified in the class of the object as *instance variables*. Each object knows its own attributes, but not the attributes of other objects.

Object-Oriented Analysis and Design

How will you create the code for your programs? You should follow a detailed analysis process for determining your project's *requirements* (i.e., defining what the system is supposed to do). You should develop a *design* that satisfies them (i.e., deciding how the system should do it). Object-oriented analysis and design (OOAD) views a systems as a group of interacting objects. Object-oriented programming (OOP) allows you to implement an object-oriented design as a working system. It provides a compiler that allows a programmer to define classes. The Unified Modeling Language (UML) is the most widely used graphical scheme for OOAD.

OOAD - Object Oriented Paradigm

A Brief History

The object-oriented paradigm took its shape from the initial concept of a new programming approach, while the interest in design and analysis methods came much later.

- The first object–oriented language was Simula (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center.
- In 1970, Alan Kay and his research group at Xerox PARK created a personal computer named Dynabook and the first pure object-oriented programming language (OOPL) - Smalltalk, for programming the Dynabook.
- In the 1980s, Grady Booch published a paper titled Object Oriented Design that mainly presented a design for the programming language, Ada. In the ensuing editions, he extended his ideas to a complete object–oriented design method.
- In the 1990s, Coad incorporated behavioral ideas to object-oriented methods.

The other significant innovations were Object Modelling Techniques (OMT) by James Rumbaugh and Object-Oriented Software Engineering (OOSE) by Ivar Jacobson.

Object-Oriented Analysis

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

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

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

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

- Identifying objects
- Organizing the objects by creating object model diagram

- Defining the internals of the objects, or object attributes
- Defining the behavior of the objects, i.e., object actions
- Describing how the objects interact

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

Object-Oriented Design

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

The implementation details generally include −

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

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

Object-Oriented Programming

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

The important features of object–oriented programming are −

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

Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

Grady Booch has defined object–oriented programming as *"a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships"*.

Structured Analysis	Object-Oriented Analysis
The main focus is on process and procedures of system.	The main focus in on data structure and real-world objects that are important.
It uses System Development Life Cycle (SDLC) methodology for different purposes like planning, analyzing, designing, implementing, and supporting an information system.	It uses Incremental or Iterative methodology to refine and extend our design.
It is suitable for well-defined projects with stable user requirements.	It is suitable for large projects with changing user requirements.
Risk while using this analysis technique is high and reusability is also low.	Risk while using this analysis technique is low and reusability is also high.
Structuring requirements include DFDs (Data Flow Diagram), Structured English, ER (Entity Relationship) diagram, CFD (Control Flow Diagram), Data Dictionary, Decision table/tree, State transition diagram.	Requirement engineering includes Use case model (find Use cases, Flow of events, Activity Diagram), the Object model (find Classes and class relations, Object interaction, Object to ER mapping), Statechart Diagram, and deployment diagram.
This technique is old and is not preferred usually.	This technique is new and is mostly preferred.

Advantages/Disadvantages of Object Oriented Analysis

Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	It cannot identify which objects would generate an optimal system design.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show the communications between the objects in the system.
It allows effective management of software complexity by the virtue of modularity.	All the interfaces between the objects cannot be represented in a single diagram.
It can be upgraded from small to large systems at a greater ease than in systems following structured analysis.	

Advantages/Disadvantages of Structured Analysis

Advantages	Disadvantages
As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA.	In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change.
It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems.	The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later.
The specifications in it are written in simple English language, and hence can be more easily analyzed by non-technical personnel.	It does not support reusability of code. So, the time and cost of development is inherently high.

Object Oriented Approach

In the object-oriented approach, the focus is on capturing the structure and behavior of information systems into small modules that combines both data and process. The main aim of Object Oriented Design (OOD) is to improve the quality and productivity of system analysis and design by making it more usable.

In analysis phase, OO models are used to fill the gap between problem and solution. It performs well in situation where systems are undergoing continuous design, adaption, and maintenance. It identifies the objects in problem domain, classifying them in terms of data and behavior.

The OO model is beneficial in the following ways −

- It facilitates changes in the system at low cost.
- It promotes the reuse of components.
- It simplifies the problem of integrating components to configure large system.

- It simplifies the design of distributed systems.

Elements of Object-Oriented System

Let us go through the characteristics of OO System −

- **Objects** − An object is something that is exists within problem domain and can be identified by data (attribute) or behavior. All tangible entities (student, patient) and some intangible entities (bank account) are modeled as object.
- **Attributes** − They describe information about the object.
- **Behavior** − It specifies what the object can do. It defines the operation performed on objects.
- **Class** − A class encapsulates the data and its behavior. Objects with similar meaning and purpose grouped together as class.
- **Methods** − Methods determine the behavior of a class. They are nothing more than an action that an object can perform.
- **Message** − A message is a function or procedure call from one object to another. They are information sent to objects to trigger methods. Essentially, a message is a function or procedure call from one object to another.

Structured Approach Vs. Object-Oriented Approach

The following table explains how the object-oriented approach differs from the traditional structured approach −

Structured Approach	Object Oriented Approach
It works with Top-down approach.	It works with Bottom-up approach.
Program is divided into number of submodules or functions.	Program is organized by having number of classes and objects.
Function call is used.	Message passing is used.
Software reuse is not possible.	Reusability is possible.
Structured design programming usually	Object oriented design programming done concurrently with

left until end phases.	other phases.
Structured Design is more suitable for offshoring.	It is suitable for in-house development.
It shows clear transition from design to implementation.	Not so clear transition from design to implementation.
It is suitable for real time system, embedded system and projects where objects are not the most useful level of abstraction.	It is suitable for most business applications, game development pr ojec expected to customize or extended.
DFD & E-R diagram model the data.	Class diagram, sequence diagram, state chart diagram, and use cases all contribute.
In this, projects can be managed easily due to clearly identifiable phases.	In this approach, projects can be difficult to manage due to uncertain transitions between phase.

Unified Modeling Language (UML)

UML is a visual language that lets you to model processes, software, and systems to express the design of system architecture. It is a standard language for designing and documenting a system in an object oriented manner that allow technical architects to communicate with developer.

It is defined as set of specifications created and distributed by Object Management Group. UML is extensible and scalable.

The objective of UML is to provide a common vocabulary of object-oriented terms and diagramming techniques that is rich enough to model any systems development project from analysis through implementation.

UML is made up of −

- **Diagrams** − It is a pictorial representations of process, system, or some part of it.
- **Notations** − It consists of elements that work together in a diagram such as connectors, symbols, notes, etc.

Uses of UML

UML is quite useful for the following purposes −

- Modeling the business process
- Describing the system architecture
- Showing the application structure
- Capturing the system behavior
- Modeling the data structure
- Building the detailed specifications of the system
- Sketching the ideas
- Generating the program code

Static Models

Static models show the structural characteristics of a system, describe its system structure, and emphasize on the parts that make up the system.

- They are used to define class names, attributes, methods, signature, and packages.
- UML diagrams that represent static model include class diagram, object diagram, and use case diagram.

Dynamic Models

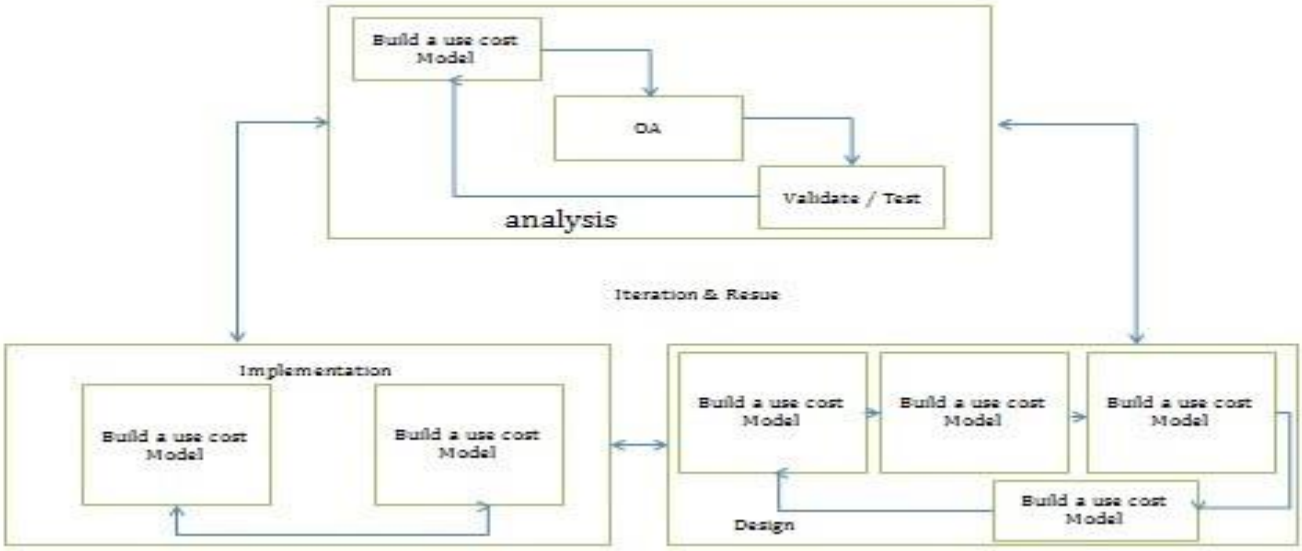
Dynamic models show the behavioral characteristics of a system, i.e., how the system behaves in response to external events.

- Dynamic models identify the object needed and how they work together through methods and messages.
- They are used to design the logic and behavior of system.
- UML diagrams represent dynamic model include sequence diagram, communication diagram, state diagram, activity diagram.

Object Oriented System Development Life Cycle

It consists of three macro processes −

- Object Oriented Analysis (OOA)
- Object oriented design (OOD)
- Object oriented Implementation (OOI)



Object Oriented Systems Development Activities

Object-oriented systems development includes the following stages −

- Object-oriented analysis
- Object-oriented design
- Prototyping
- Implementation
- Incremental testing

Object-Oriented Analysis

This phase concerns with determining the system requirements and to understand the system requirements build a **use-case model**. A use-case is a scenario to describe the interaction between user and computer system. This model represents the user needs or user view of system.

It also includes identifying the classes and their relationships to the other classes in the problem domain, that make up an application.

Object-Oriented Design

The objective of this phase is to design and refine the classes, attributes, methods, and structures that are identified during the analysis phase, user interface, and data access. This phase also identifies and defines the additional classes or objects that support implementation of the requirement.

Prototyping

Prototyping enables to fully understand how easy or difficult it will be to implement some of the features of the system.

It can also give users a chance to comment on the usability and usefulness of the design. It can further define a use-case and make use-case modeling much easier.

I**mplementation**

It uses either Component-Based Development (CBD) or Rapid Application Development (RAD).

Component-based development (CBD)

CODD is an industrialized approach to the software development process using various range of technologies like CASE tools. Application development moves from custom development to assembly of pre-built, pre-tested, reusable software components that operate with each other. A CBD developer can assemble components to construct a complete software system.

Rapid Application Development (RAD)

RAD is a set of tools and techniques that can be used to build an application faster than typically possible with traditional methods. It does not replace SDLC but complements it, since it focuses more on process description and can be combined perfectly with the object oriented approach.

Its task is to build the application quickly and incrementally implement the user requirements design through tools such as visual basic, power builder, etc.

Incremental Testing

Software development and all of its activities including testing are an iterative process. Therefore, it can be a costly affair if we wait to test a product only after its complete development. Here incremental testing comes into picture wherein the product is tested during various stages of its development.

OOA - Object Oriented Analysis

In the system analysis or object-oriented analysis phase of software development, the system requirements are determined, the classes are identified and the relationships among classes are identified.

The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modelling, dynamic modelling, and functional modelling.

Object Modelling

Object modelling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.

The process of object modelling can be visualized in the following steps −

- Identify objects and group into classes
- Identify the relationships among classes
- Create user object model diagram
- Define user object attributes
- Define the operations that should be performed on the classes
- Review glossary

Dynamic Modelling

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modelling.

Dynamic Modelling can be defined as “a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world”.

The process of dynamic modelling can be visualized in the following steps −

- Identify states of each object
- Identify events and analyze the applicability of actions
- Construct dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

Functional Modelling

Functional Modelling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling. The functional model corresponds to the data flow diagram of traditional structured analysis.

The process of functional modelling can be visualized in the following steps −

- Identify all the inputs and outputs
- Construct data flow diagrams showing functional dependencies
- State the purpose of each function
- Identify constraints
- Specify optimization criteria

Structured Analysis vs. Object Oriented Analysis

The Structured Analysis/Structured Design (SASD) approach is the traditional approach of software development based upon the waterfall model. The phases of development of a system using SASD are −

- Feasibility Study
- Requirement Analysis and Specification
- System Design
- Implementation
- Post-implementation Review

Now, we will look at the relative advantages and disadvantages of structured analysis approach and object-oriented analysis approach.

Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	It cannot identify which objects would generate an optimal system design.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show the communications between the objects in the system.
It allows effective management of software complexity by the virtue of modularity.	All the interfaces between the objects cannot be represented in a single diagram.
It can be upgraded from small to large systems at a greater ease than in systems following structured analysis.	

Advantages/Disadvantages of Structured Analysis

Advantages	Disadvantages
As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA.	In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change.
It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems.	The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later.
The specifications in it are written in simple English language, and hence can be more easily analyzed by non-technical personnel.	It does not support reusability of code. So, the time and cost of development is inherently high.

Programming Paradigms

A ***programming paradigm*** is a style, or "way," of programming

Programming philosophies, procedural vs structured programming.

Procedural and structured programming

Procedural programming is a programming paradigm that has its roots in structured programming.

In procedural programming, programs are made up of **procedures**, also known as routines, subroutines, or functions. In this topic procedures will be referred to as **subroutines**. The term procedure is conventionally used to classify subroutines that do not return any value(s), whereas a *function* always has a return statement.

Each subroutine contains a series of computational steps that will be carried out when the subroutine is called. Many useful subroutines are built in to programming languages and there are often additional libraries of subroutines that can be imported to support particular programming tasks.

Some programming languages support only the procedural paradigm, others are purely object oriented or functional. If you have used Python, a popular programming language that supports all three of the paradigms mentioned, you will probably have started your programming journey by taking a procedural approach to program development.

A program in a **procedural language** is a list of instructions where each statement tells the computer to do something. It focuses on the procedure (function) & an algorithm is needed to perform the derived computation. When the program becomes larger, it is divided into function & each function has clearly defined purpose. Dividing the program into functions & module is one of the cornerstones of structured programming. Procedural languages are some of the common types of programming languages used by script and software programmers. They make use of functions, conditional statements, and variables to create programs that allow a computer to calculate and display the desired output.

Characteristics of Procedural Language-

- It focuses on processes rather than data.
- It takes a problem as a sequence of things to be done such as reading, calculating, and printing. Hence, a number of functions are written to solve a problem.
- A program is divided into a number of functions and each function has clearly defined purpose.
- Most of the functions share global data.
- Data moves openly around the system from function to function.

Structured programming (sometimes known as *modular programming*) is a programming paradigm that facilitates the creation of programs with readable code and reusable components. All modern programming

languages support structured programming, but the mechanisms of support, like the syntax of the programming languages, varies.

Where modules or elements of code can be reused from a library, it may also be possible to build structured code using modules written in different languages, as long as they can obey a common module interface or application program interface (API) specification. However, when modules are reused, it's possible to compromise data security and governance, so it's important to define and enforce a privacy policy controlling the use of modules that bring with them implicit data access rights.

Structured programming encourages dividing an application program into a hierarchy of modules or autonomous elements, which may, in turn, contain other such elements. Within each element, code may be further structured using blocks of related logic designed to improve readability and maintainability. These may include case, which tests a variable against a set of values; Repeat, while and for, which construct loops that continue until a condition is met. In all structured programming languages, an unconditional transfer of control, or goto statement, is deprecated and sometimes not even available.

Whereas, **structured programming language** allows a programmer to code a program by dividing the whole program into smaller units or modules. Structured programming is not suitable for the development of large programs and does not allow the reusability of any set of codes. It is a programming paradigm aimed at improving the quality, clarity, and access time of a computer program by the use of subroutines, block structures, for and while loops.

Characteristics of Structured Language:

- Structured programming is user-friendly and easy to understand.
- In this programming, programs are easier to read and learn.
- It avoids the increased possibility of data corruption.
- The main advantage of structured programming is reduced complexity.
- Increase the productivity of application program development.
- Application programs are less likely to contain logic errors.
- Errors are more easily found.

Now, **C** is called a structured programming language because to solve a large problem, C programming language divides the problem into smaller structural blocks each of which handles a particular responsibility. These structural blocks are–

- Decision-making blocks like if-else-elseif, switch-cases.
- Repetitive blocks like For-loop, While-loop, Do-while loop, etc.

- subroutines/procedures - functions.

The program which solves the entire problem is a collection of such structural blocks. Even a bigger structural block like a function can have smaller inner structural blocks like decisions and loops.

Procedural Programming:

Procedural Programming can be defined as a programming model which is derived from structured programming, based upon the concept of calling procedure. Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out. During a program's execution, any given procedure might be called at any point, including by other procedures or itself.

Languages used in Procedural Programming:

FORTRAN, ALGOL, COBOL, BASIC, Pascal and C. etc

Object Oriented Programming:

Object oriented programming can be defined as a programming model which is based upon the concept of objects. Objects contain data in the form of attributes and code in the form of methods. In object oriented programming, computer programs are designed using the concept of objects that interact with real world. Object oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

Languages used in Object Oriented Programming:

Java, C++, C#, Python,

PHP, JavaScript, Ruby, Perl,

Objective-C, Dart, Swift, Scala.

Procedural Oriented Programming	Object Oriented Programming
In procedural programming, program is divided into small parts called ***functions***.	In object oriented programming, program is divided into small parts called ***objects***.
Procedural programming follows ***top down approach***.	Object oriented programming follows ***bottom up approach***.
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private, public, protected etc.

Procedural Oriented Programming	Object Oriented Programming
Adding new data and function is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way for hiding data so it is ***less secure***.	Object oriented programming provides data hiding so it is ***more secure***.
In procedural programming, overloading is not possible.	Overloading is possible in object oriented programming.
In procedural programming, function is more important than data.	In object oriented programming, data is more important than function.
Procedural programming is based on ***unreal world***.	Object oriented programming is based on ***real world***.
Examples: C, FORTRAN, Pascal, Basic etc.	

What is Procedural Programming? [Definition]

Procedural Programming may be the first programming paradigm that a new developer will learn. Fundamentally, the procedural code is the one that directly instructs a device on how to finish a task in logical steps. This paradigm uses a linear top-down approach and treats data and procedures as two different entities. Based on the concept of a procedure call, Procedural Programming divides the program into procedures, which are also known as routines or functions, simply containing a series of steps to be carried out.

Simply put, Procedural Programming involves writing down a list of instructions to tell the computer what it should do step-by-step to finish the task at hand.

Key Features of Procedural Programming

The key features of procedural programming are given below:

- **Predefined functions:** A predefined function is typically an instruction identified by a name. Usually, the predefined functions are built into higher-level programming languages, but they are derived from the library or the registry, rather than the program. One example of a pre-defined function is

'charAt()', which searches for a character position in a string.

- **Local Variable**: A local variable is a variable that is declared in the main structure of a method and is limited to the local scope it is given. The local variable can only be used in the method it is defined in, and if it were to be used outside the defined method, the code will cease to work.

- **Global Variable**: A global variable is a variable which is declared outside every other function defined in the code. Due to this, global variables can be used in all functions, unlike a local variable.

- **Modularity**: Modularity is when two dissimilar systems have two different tasks at hand but are grouped together to conclude a larger task first. Every group of systems then would have its own tasks finished one after the other until all tasks are complete.

- **Parameter Passing:** Parameter Passing is a mechanism used to pass parameters to functions, subroutines or procedures. Parameter Passing can be done through 'pass by value', 'pass by reference', 'pass by result', 'pass by value-result' and 'pass by the name'.

Advantages and Disadvantages of Procedural Programming

Procedural Programming comes with its own set of pros and cons, some of which are mentioned below.

Advantages

- Procedural Programming is excellent for general-purpose programming
- The coded simplicity along with ease of implementation of compilers and interpreters
- A large variety of books and online course material available on tested algorithms, making it easier to learn along the way
- The source code is portable, therefore, it can be used to target a different CPU as well
- The code can be reused in different parts of the program, without the need to copy it
- Through Procedural Programming technique, the memory requirement also slashes
- The program flow can be tracked easily

Disadvantages

- The program code is harder to write when Procedural Programming is employed
- The Procedural code is often not reusable, which may pose the need to recreate the code if is needed to use in another application

- Difficult to relate with real-world objects
- The importance is given to the operation rather than the data, which might pose issues in some data-sensitive cases
- The data is exposed to the whole program, making it not so much security friendly

There are different types of programming paradigm as we mentioned before, which are nothing but a style of programming. It is important to understand that the paradigm does not cater to a specific language but to the way the program is written. Below is a comparison between Procedural Programming and Object-Oriented Programming.

What Is Object-Oriented Programming (OOP)

OOP is an approach to programming which recognizes life as we know it as a collection of objects, which work in tandem with each other to solve a particular problem at hand. The primary thing to know about OOP is encapsulation, which is the idea that each object which holds the program is self-sustainable, which means that all the components that make up the object are within the object itself. Now since each module within this paradigm is self-sustainable, objects can be taken from one program and used to resolve another problem at hand with little or no alterations.

Advantages

- Due to modularity and encapsulation, OOP offers ease of management
- OOP mimics the real world, making it easier to understand
- Since objects are whole within themselves, they are reusable in other programs

Disadvantages

- Object-Oriented programs tend to be slower and use up a high amount of memory
- Over-generalization
- Programs built using this paradigm may take longer to be created

Procedural Programming vs Object-Oriented Programming: Head to Head Comparison

On the other hand, Procedural Programming, unlike OOP, sheds focus on the steps which will be performed to complete a task, rather than the interaction between the objects. The tasks are broken down into subroutines, variables and data structures. At any point in time, these procedures can be called within the program execution.

Procedural Programming	**Object-Oriented Programming**

Uses immutable data	Uses mutable data
Follows the declarative programming model	Follows the imperative programming model
Extends support to parallel programming	Not suitable for parallel programming
The execution order of statements is not the primary focus	The execution order of statements is very important
Flow control is performed using function calls	Flow control is performed through conditional statements and loops
Uses recursion concept to iterate collective data	Uses loop concept to iterate collection data
No such side-effects of its functions	The method can have certain side-effects
The focus in Procedural Programming is on **'What You are Doing'**	The focus in Object-Oriented Programming is on **'How You are Doing It'**

Another highly used programming paradigm is Functional Programming. Functional Programming is way differs from both Procedural Programming and Object-Oriented Programming as it makes use of mathematical functions. Through this, the operations are performed only on the basis of the inputs that are entered, and they do not rely on temporary or hidden variables.

Advantages

- Functional Programming offers a protected environment
- While many other languages require a substantial amount of information in order to perform operations properly, function programming eliminates the need for a large amount of code needed to define states
- Since this paradigm is only dependent on the input arguments, there are no side-effects

Disadvantages

- Using Functional programming solely in commercial software development is not recommended and done
- It requires a large amount of memory and time
- It can prove to be less efficient than other paradigms

Advantages of structured programming

The primary advantages of structured programming are:

1. It encourages top-down implementation, which improves both readability and maintainability of code.
2. It promotes code reuse, since even internal modules can be extracted and made independent, residents in libraries, described in directories and referenced by many other applications.
3. It's widely agreed that development time and code quality are improved through structured programming.

These advantages are normally seen as compelling, even decisive, and nearly all modern software development employs structured programming.

Object-oriented programming (OOP). Defines a program as a set of objects or resources to which commands are sent. An object-oriented language will define a data resource and send it to process commands. For example, the procedural programmer might say "Print(object)" while the OOP programmer might say "Tell Object to Print".

Model-based programming. The most common example of this is database query languages. In database programming, units of code are associated with steps in database access and update or run when those steps occur. The database and database access structure will determine the structure of the code. Another example of a model-based structure is Reverse Polish Notation (RPN), a math-problem structure that lends itself to efficient solving of complex expressions. Quantum computing, just now emerging, is another example of model-based structured programming; the quantum computer demands a specific model to organize steps, and the language simply provides it.

Principles of OOP

Object-oriented programming is based on the following principles:

- Encapsulation. The implementation and state of each object are privately held inside a defined boundary, or class. Other objects do not have access to this class or the authority to make changes but are only able to call a list of public functions, or methods. This characteristic of data hiding provides greater program security and avoids unintended data corruption.

- Abstraction. Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. This concept helps developers more easily make changes and additions over time.
- Inheritance. Relationships and subclasses between objects can be assigned, allowing developers to reuse a common logic while still maintaining a unique hierarchy. This property of OOP forces a more thorough data analysis, reduces development time and ensures a higher level of accuracy.
- Polymorphism. Objects can take on more than one form depending on the context. The program will determine which meaning or usage is necessary for each execution of that object, cutting down the need to duplicate code.

Analysis: Finding the classes and objects

Overview

- Analysis is an attempt to build a model that describes the application domain -- developers do this
- Takes place after (or during) requirements specification
- The analysis model will typically consist of all three types of models discussed before:
    - Functional model (denoted with use cases)
    - Analysis object model (class and object idagrams)
    - Dynamic model
- At this level, note that we are still looking at the application domain.
    - This is not yet system design
    - However, many things discovered in analysis could translate closely into the system design
- Goal is to completely understand the application domain (the problem at hand, any constraints that must be adhered to, etc.)
    - New insights gained during analysis might cause requirements to be updated.
- Analysis activities include:
    - Identifying objects (often from use cases as a starting point)
    - Identfying associations between objects
    - Identifying general attributes and responsibilities of objects
    - Modeling interactions between objects
    - Modeling how individual objects change state -- helps identify operations
    - Checking the model against requirements, making adjustments, iterating through the process more than once

Finding the objects

- We often think of objects in code as mapping to some object we want to represent in the real world. Although this isn't always the case.
- Here are some categories of objects to look for:
    - **Entity objects** -- these represent persistent information tracked by a system. This is the closest parallel to "real world" objects.
    - **Boundary objects** -- these represent interactions between user and system. (For instance, a button, a form, a display)
    - **Control objects** -- usually set up to manage a given usage of the system. Often represent the control of some activity performed by a system
- UML diagrams can include a label known as a *stereotype*, above the class name in a class diagram. This would be placed inside << >> marks, like this:
    - <<entity>>
    - <<boundary>>
    - <<control>>
- Note: Different sources and/or "experts" will give other categorizations of types of objects
- There are some different popular techniques for identifying objects. Two traditional and popular ones that we will discuss are:
    - natural language analysis (i.e. parts of speech)
    - CRC cards
- It also helps to interact with domain experts -- these are people who are already well-versed in the realm being studied.
- Note that the goal in the analysis phase is NOT to find implementation specific objects, like HashTable or Stack.
    - This stage still models the application domain

Using natural language analysis

- Pioneered by Russell Abbott (1983), popularized by Grady Booch
- Not perfect, but coupled with other techniques, it's a good start
- This can be done from a general problem description, or better, from a use case or scenario
- Map parts of speech to object model components.
    - nouns usually map to classes, objects, or attributes
    - verbs usually map to operations or associations

Part of speech	model component	Examples
Proper noun	Instance (object)	Alice, Ace of Hearts
Common noun	Class (or attribute)	Field Officer, PlayingCard, value
Doing verb	Operation	Creates, submits, shuffles
Being verb	Inheritance	Is a kind of, is one of either
Having verb	Aggregation/Composition	Has, consists of, includes
Modal verb	Constraint	Must be
Adjective	Helps identify an attribute	a *yellow* ball (i.e. color)

Identifying different object types

Finding Entity Objects

- Some things to look for. These may be candidates for objects, or they may help identify objects:
    - Terms that are domain-specific in use cases
    - Recurring nouns
    - Real-world entities and activities tracked by system
- Use good naming conventions. Good to use names from the application domain -- they understand their own terminology best
- Example: In a ReportEmergency use case -- "A field officer sumits information to the system by filling out a form and pressing the 'Send Report' button"
    - FieldOfficer is a real world entity that interacts with the system
    - This is also likely an *actor* from the use case
    - As an actor, FieldOfficer is an external entity
    - But we see that the field officer submits *information* -- here's data to be tracked
    - We'll create the entity object type EmergencyReport, as that's the more common name for the information the officer submits (according to client)

Finding Boundary Objects

- Identify general user interface controls that initiate a use case
    - Note: Don't bother with the visual details here. This will evolve later
- Identify forms or windows for entering data into a system
- Identify messages used by system to respond to a user

Finding Control Objects

Control objects can help manage communication and interaction of other objects

- If a use case is conplex and involves many objects, create a control object to manage the use case
- Identify one control object per actor involved in a use case
- Life span of control object should last through the use case

CRC Cards

- A simple object-oriented analysis technique that includes the users and developers in the analysis process
- A CRC card is an index card with three parts:
  - *Class* -- name goes at the top of the card
  - *Responsibilities* -- as a list on the left side of the card
  - *Collaborators* -- as a list on the right side of the card
- Here's the layout:

ClassName	
Responsibility	*Collaborator*

-
- **Class**
  - Represents a type of object being modeled
  - One card per class
- **Responsibility**
  - Something that the class knows (keeps track of) or does
  - These should be the high-level responsibilities. Not trying to list out all member functions here
  - Example: class Mailbox in a voice mail system might have these responsibilities:
    - keep new and saved messages
    - manage the recorded greeting
- **Collaborator**
  - Another class that the current class has to work with to complete its responsibilities
  - Could be a class that has information we need

- Could be a class that helps perform a task
- Typically, we list a class as a collaborator if we (the current class) need to call upon it to help complete our own responsibilities
- Example: To succcessfully keep new and saved messages, the Mailbox class has to send them to a MessageQueue to be added and stored. So on the Mailbox card, we list MessageQueue as a collaborator

A CRC Card Session

- CRC cards can be used as a brainstorming technique, for the purpose of:
  - Identifying objects/classes
  - Identifying what each object's purpose is (responsibilities)
  - Discovering the dependencies and relationships between objects (collaborators
- A CRC card "session" involves users and developers:
  - *Domain experts/users* -- intended users of the system, people who know the business being modeled. Good to have a few of these
  - *Developer/Analyst* -- should have a couple members of the development team. People who understand OOP modeling and development processes
  - *Facilitator* -- one person who keeps things on track and progressing forward
- The process is based on going through use cases (or specific scenarios built from use cases), and using these to discover objects, responsibilities, and collaborators
- The general process:
  - Start with a scenario (usually representing a normal course through a use case)
  - Identify initial classes/objects and make cards for them (this is can often be done by picking out the nouns)
  - Going through a scenario helps identify responsibilities of a chosen object
  - Identify collaborations between objects that have been created
  - Sometimes, we'll identify a collaboration with a new object type that doesn't have a card yet -- this helps discover new classes
  - When new classes are created, walk through scenarios again to discover any new responsibilities and collaborators (it's an iterative process)
  - More use cases/scenarios will yield more classes, responsibilities, and collaborators
- Finding responsibilities
  - Look for verbs in the scenario descriptions. These often tell us what an object *does*

- Also ask what the class *knows*. This tells us what an object needs to store. Sometimes a primary responsibility of a class is management of certain unique information

- Finding collaborators
  - If a class has a responsibility that required it to get, or modify, information it doesn't have on its own, it will need to collaborate with another class
  - Most often, one class specifically initiates the collaboration
  - Usually, the collaboration is a request for information or a request to do something
  - The *initiator's* card should list the helper class as a collaborator
  - In this case, the initiator class *depends on* the collaborator class to accomplish its tasks

Class Diagrams

A class diagram is used to show the existence of classes and their relationships in the logical view of a system. A single class diagram represents a view of the class structure of a system. During analysis, we use class diagrams to indicate the common roles and responsibilities of the entities that provide the system's behavior. During design, we use class diagrams to capture the structure of the classes that form the system's architecture. The two essential elements of a class diagram are classes and their basic relationships.

Essentials: The Class Notation Figure 5–33 shows the icon used to represent a class in a class diagram and an example from our Hydroponics Gardening System. The class icon consists of three compartments, with the first occupied by the class name, the second by the attributes, and the third by the operations. A name is required for each class and must be unique to its enclosing namespace

A name is required for each class and must be unique to its enclosing namespace. By convention, the name begins in capital letters, and the space between multiple words is omitted. Again by convention, the first letter of the attribute and operation names is lowercase, with subsequent words starting in uppercase, and spaces are omitted just as in the class name. Since the class is the namespace for its attributes and operations, an attribute name must be unambiguous in the context of the class. So must an operation name, according to the rules in the chosen implementation language.

In UML classes and objects are depicted by boxes comprising of 3 compartments. The top displays the name of the class or object. The center displays the attributes and the bottom displays the operations.

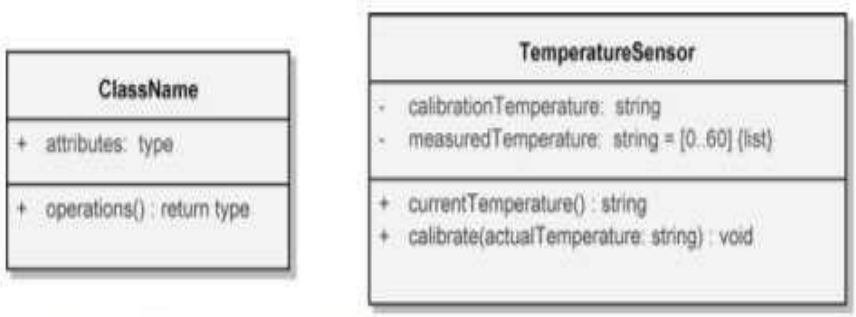


Figure 5–33 A General Class Icon and an Example for the Gardening System

The format of the attribute and operation specifications is shown here [29]:

■ Attribute specification format: visibility attribute Name: Type [multiplicity] = Default Value {property string} Figure 5–33 A General Class Icon and an Example for the Gardening System

■ Operation specification format: visibility operation Name (parameter Name: Type) : Return Type {property string}

Essentials: Class Relationships

Classes rarely stand alone; instead, they collaborate with other classes in a variety of ways. The essential connections among classes include association, generalization, aggregation, and composition

shown in Figure 3–3.

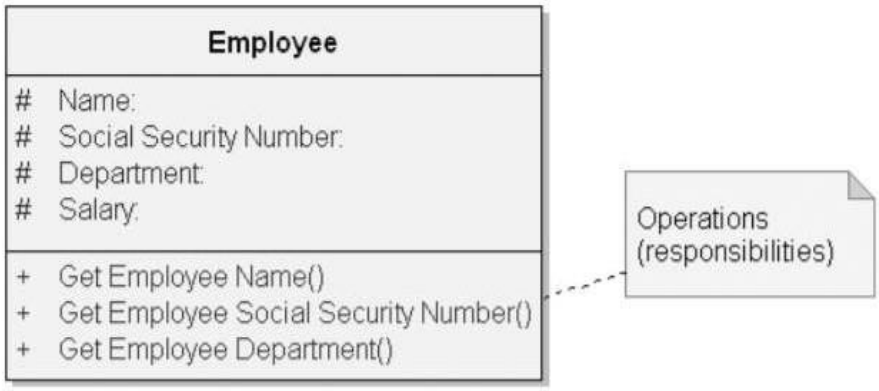


Figure 3–3 `Employee` Class with Protected Attributes and Public Operations

A class represents a collection of similar objects. An object is a person, place, thing, event, concept, screen, or report that is relevant to the system at hand. For example, Figure 2 shows a shipping/inventory control system with the classes such as Inventory Item, Order, Order Item, Customer, and Surface Address. The name of the class appears across the top of the card.

A responsibility is anything that a **class knows or does.** For example, customers have names, customer numbers, and phone numbers. These are the things that a **customer knows**. Customers also **order products, cancel orders, and make payments.** These are the things that a customer does. The things that a class knows and does constitute its responsibilities.

Responsibilities are shown on the left hand column of a CRC card. Sometimes a class will have a responsibility to fulfill, but will not have enough information to do it. When this happens it has to collaborate with other classes to get the job done. For example, an Order object has the responsibility to calculate it's total. Although it knows about the Order Item objects that are a part of the order, it doesn't know how many items were ordered (Order Item knows this) nor does it know the price of the item (Inventory Item knows this). To calculate the order total, the Order object collaborates with each Order Item object to calculate its own total, and then adds up all the totals to calculate the overall total. For each Order Item to calculate its individual total, it has to collaborate with Inventory Item to determine the cost of the ordered item, multiplying it by the number ordered (which it does know). The collaborators of a class are shown in the right-hand column of a CRC card.

CRC Models

A CRC model is a collection of CRC cards that represent whole or part of an application or problem domain. The most common use for CRC models, the one that this white paper addresses, is to gather and define the user requirements for an object-oriented application[1]. Figure 2 presents an example CRC model for a shipping/inventory control system, showing the CRC cards as they would be placed on a desk or work table. Note the placement of the cards: Cards that collaborate with one another are close to each other, cards that don't collaborate are not near each other.

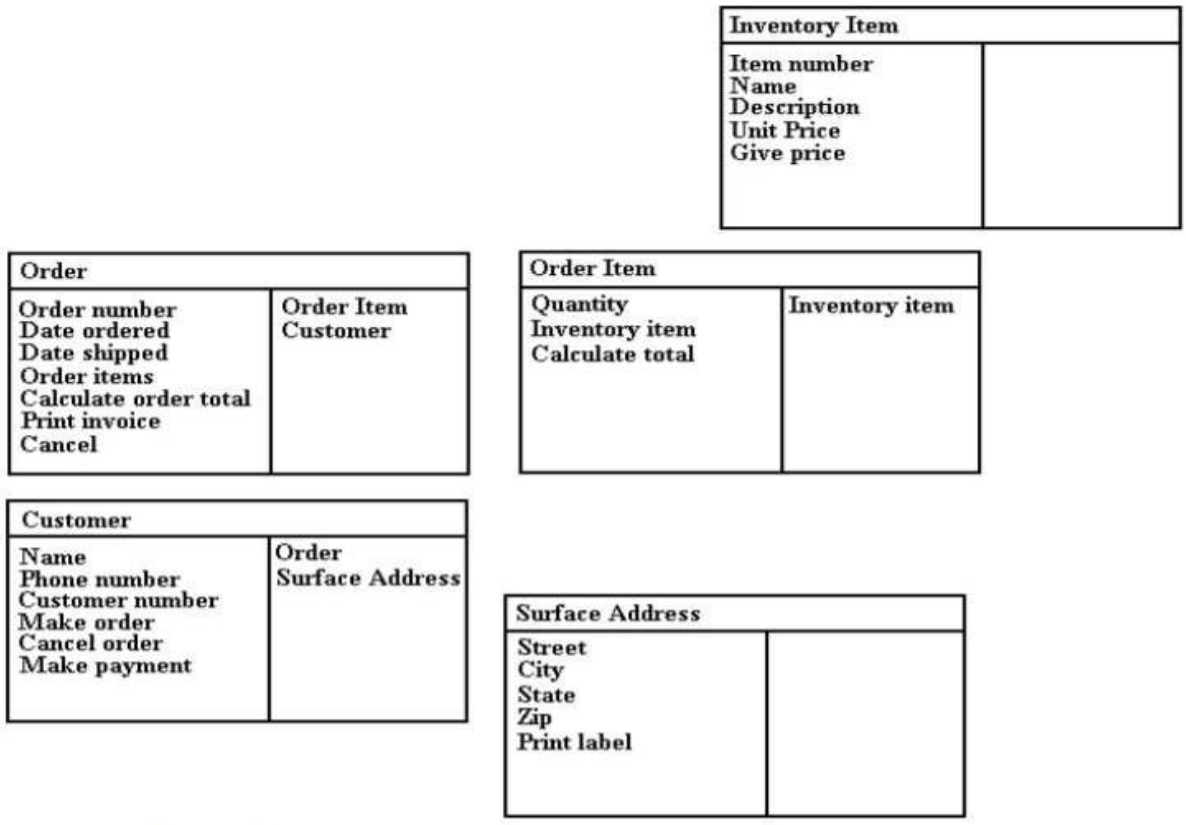


Figure 2. A CRC model for a simple shipping/inventory control system.

CRC MODELLING

The steps of CRC modeling are: ·

Find classes · Find responsibilities · Define collaborators · Define use-cases · Arrange the cards on the table

Finding Classes · Look for anything that interacts with the system, or is part of the system · Ask yourself "Is there a customer?" · Follow the money · Look for reports generated by the system · Look for any screens used in the system · Immediately prototype interface and report classes ·

Look for the three to five main classes right away · Create a new card for a class immediately · Use one or two words to describe the class · Class names are singular

Finding Responsibilities · Ask yourself what the class knows · Ask yourself what the class does · If you've identified a responsibility, ask yourself what class it "belongs" to · Sometimes get responsibilities that we won't implement, and that's OK · Classes will collaborate to fulfil many of their responsibilities

Defining Collaborators · Collaboration occurs when a class needs information that it doesn't have · Collaboration occurs when a class needs to modify information that it doesn't have · There will always be at least one initiator of any given collaboration · Sometimes the collaborator does the bulk of the work · Don't pass the buck · New responsibilities may be created to fulfill the collaboration

Behavior

No object exists in isolation. Rather, objects are acted on and themselves act on other objects. Thus, we may say the following: Behavior is how an object acts and reacts, in terms of its state changes and message passing. In other words, the behavior of an object represents its outwardly visible activity.

An operation is some action that one object performs on another in order to elicit a reaction. For example, a client might invoke the operations Of placing an order (invokes order class to react) Generally, a message is simply an operation that one object performs on another.

Relationships among Classes

Consider for a moment the similarities and differences among the following classes of objects: flowers, daisies, red roses, yellow roses, petals, and ladybugs. We can make the following observations.

■ A daisy is a kind of flower.

■ A rose is a (different) kind of flower.

■ Red roses and yellow roses are both kinds of roses.

■ A petal is a part of both kinds of flowers.

■ Ladybugs eat certain pests such as aphids, which may be infesting certain kinds of flowers.

From this simple example we conclude that classes, like objects, do not exist in isolation. Rather, for a particular problem domain, the key abstractions are usually related in a variety of interesting ways, forming the class structure of our design [21]. We establish relationships between two classes for one of two reasons. First, a class relationship might indicate some sort of sharing. For example, daisies and roses are both kinds of flowers, meaning that both have brightly colored petals, both emit a fragrance, and so on. Second, a class relationship might indicate some kind of semantic connection. Thus, we say that red roses and yellow roses are more alike than are daisies and roses, and daisies and roses are more closely related than are petals and flowers. Similarly, there is a symbiotic connection between ladybugs and flowers: Ladybugs protect flowers from certain pests, which in turn serve as a food source for the ladybug.

In all, there are three basic kinds of class relationships [22]. The first of these is generalization/specialization, denoting an "is a" relationship. For instance, a rose is a kind of flower, meaning that a rose is a specialized subclass of the more general class, flower. The second is whole/part, which denotes a "part of" relationship. Thus, a petal is not a kind of a flower; it is a part of a flower. The third is association, which denotes some semantic dependency among otherwise unrelated classes, such as between ladybugs and flowers. As another example, roses and candles are largely independent classes, but they both represent things that we might use to decorate a dinner table

CLASS RELATIONSHIPS

Inheritance

Inheritance, perhaps the most semantically interesting of these concrete relationships, exists to express generalization/specialization relationships. In our experience, however, inheritance is an insufficient means of expressing all of the rich relationships that may exist among the key abstractions in a given problem domain

Single Inheritance and Multiple inheritance

Simply stated, inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes

Aggregation

We also need aggregation relationships, which provide the whole/part relationships manifested in the class's instances. Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes.

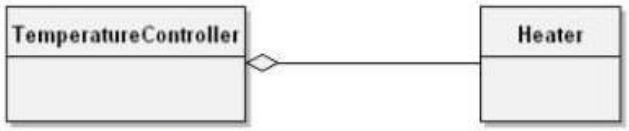


Figure 3–12 Aggregation

As we show in Figure 3–12, the class `TemperatureController` denotes the whole, and the class `Heater` is one of its parts. This corresponds exactly to the aggregation relationship among the instances of these classes illustrated earlier in Figure 3–6.

Physical Containment

In the case of the class TemperatureController, we have aggregation as containment by value, a kind of physical containment meaning that the Heater object does not exist independently of its enclosing TemperatureController instance. Rather, the lifetimes of these two objects are intimately connected: When we create an instance of TemperatureController, we also create an instance of the class Heater. When we destroy our TemperatureController object, by implication we also destroy the corresponding Heater object.

A less direct kind of aggregation is also possible, called **composition**, which is containment by reference. In this case, the class TemperatureController still denotes the whole, and an instance of the class Heater is still one of its parts, although that part must now be accessed indirectly. Hence, the lifetimes of these two objects are not so tightly coupled as before: We may create and destroy instances of each class independently.

Aggregation asserts a direction to the whole/part relationship. For example, the Heater object is a part of the TemperatureController object, and not vice versa. Of course, as we described in an earlier example, aggregation need not require physical containment. For example, although shareholders own stocks, a shareholder does not physically contain the owned stocks. Rather, **the lifetimes of these objects may be completely independent**, although there is still conceptually a whole/part relationship (each share is always a part of the shareholder's assets). Representation of this aggregation can be very indirect. This is still aggregation, although not physical containment. Ultimately, the litmus test for aggregation is this: If and only if there exists a whole/part relationship between two objects, we must have an aggregation relationship between their corresponding classes.

Aggregation is a specialized form of association between two or more objects in which each object has its own life cycle but there exists an ownership as well. Aggregation is a typical whole/part or parent/child relationship but it may or may not denote physical containment. An essential property of an aggregation relationship is that the whole or parent (i.e. the owner) can exist without the part or child and vice versa.

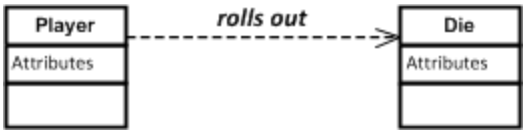
As an example, an employee may belong to one or more departments in an organization. However, if an employee's department is deleted, the employee object would not be destroyed but would live on. Note that the relationships between objects participating in an aggregation cannot be reciprocal—i.e., a department may "own" an employee, but the employee does not own the department.

Composition is a specialized form of aggregation. In composition, if the parent object is destroyed, then the child objects also cease to exist. Composition is actually a strong type of aggregation and is sometimes referred to as a "death" relationship. As an example, a house may be composed of one or more rooms. If the house is destroyed, then all of the rooms that are part of the house are also destroyed. The following code snippet illustrates a composition relationship between two classes, House and Room.

Dependencies

Aside from inheritance, aggregation, and association, there is another group of relationships called dependencies. A dependency indicates that an element on one end of the relationship, in some manner, depends on the element on the other end of the relationship. This alerts the designer that if one of these elements changes, there could be an impact to the other. There are many different kinds of dependency relationships (refer to the Object Management Group's latest UML specification for the full list [45]). You will often see dependencies used in architectural models (one system component, or package, is dependent on another) or at the implementation level (one module is dependent on another).

Dependency is often confused as Association. Dependency is normally created when you receive a reference to a class as part of a particular operation / method. Dependency indicates that you may invoke one of the APIs of the received class reference and any modification to that class may break your class as well. Dependency is represented by a dashed arrow starting from the dependent class to its dependency. Multiplicity normally doesn't make sense on a Dependency.



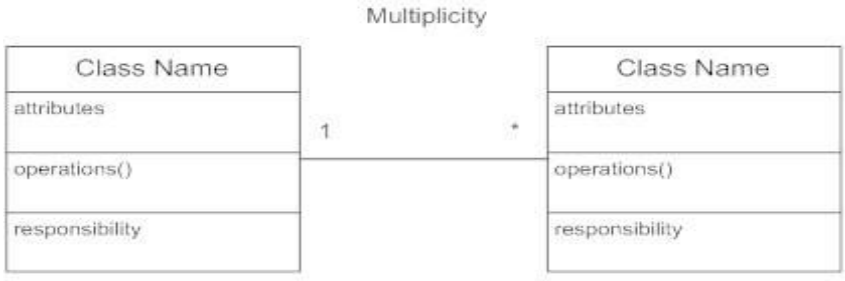
Association vs. Aggregation vs. Composition

Association	Aggregation	Composition
Association relationship is denoted using an arrow.	Aggregation relationship is denoted using a straight line with an empty arrowhead at one end.	Composition relationship is denoted using a straight line with a filled arrowhead at any one of the ends.
Association can exist between two or more classes in UML.	Aggregation is a part of an association relationship.	The composition is a part of an association relationship.
There can be one-one, one-many, many-one, and many-many association present between the association classes.	Aggregation is considered as a weak type of association.	The composition is considered as a strong type of association.
In an association relationship, one or more objects can be associated with each other.	In an aggregation relationship, objects that are associated with each other can remain in the scope of a system without each other.	In a composition relationship, objects that are associated with each other cannot remain in the scope without each other.
Objects are linked with each other.	Linked objects are not dependent upon the other object.	Objects are highly dependent upon each other.
In UML Association, deleting one element may or may not affect another associated element.	In UML Aggregation, deleting one element does not affect another associated element.	In UML Composition, deleting one element affects another associated element.
Example: A teacher is associated with multiple students. Or a teacher provides instructions to the students.	Example: A car needs a wheel, but it doesn't always require the same wheel. A car can function adequately with another wheel as well.	Example: A file is placed inside the folder. If one deletes the folder, then the file associated with that given folder is also deleted.

MULTIPLICITY IN CLASS DIAGRAMS

Multiplicity determines how many objects can participate in a relationship

e.g. order class is dependent on the customer class because it doesn't make sense to be able to create an order that doesn't belong to a particular customer. These symbols indicate the number of instances of one class linked to one instance of the other class. For example, one company will have one or more employees, but each employee works for just one company.



Indicator	Meaning
0..1	Zero or one
1	One only
0..*	0 or more
1..* *	1 or more
n	Only n (where n > 1)
0..n	Zero to n (where n >1)
1..n	One to n (where n > 1)

in the above example a customer can place many orders. As no multiplicity value is specified at the customer end, an implicit value of one is assumed, signifying that an order can have only one customer.

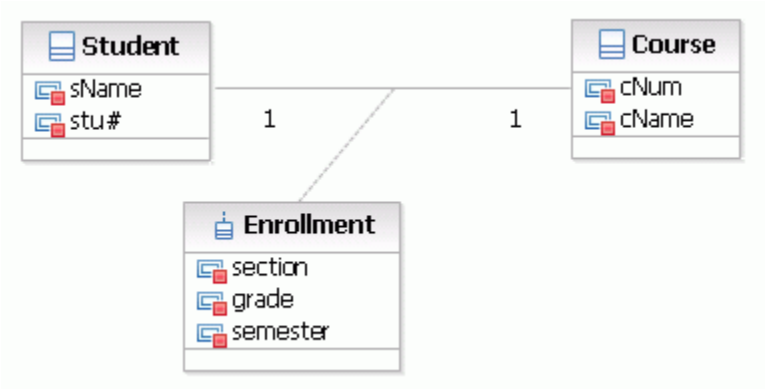
The following table lists the other possibilities for multiplicity

ASSOCIATIVE CLASSES /LINK CLASSES (MANY TO MANY RELATIONSHIP)

They are useful to keep important information about association itself e.g, a customer purchases a book.

In UML diagrams, an association class is a class that is part of an association relationship between two other classes. You can attach an association class to an association relationship to provide additional information about the relationship. An association class is identical to other classes and can contain operations, attributes, as well as other associations. For example, a class called Student represents a student and has an association with a class called Course, which represents an educational course. The Student class can enroll in a course. An association class called Enrollment further defines the relationship between the Student and Course classes by providing section, grade, and semester information related to the association relationship.

As the following figure illustrates, an association class is connected to an association by a dotted line.



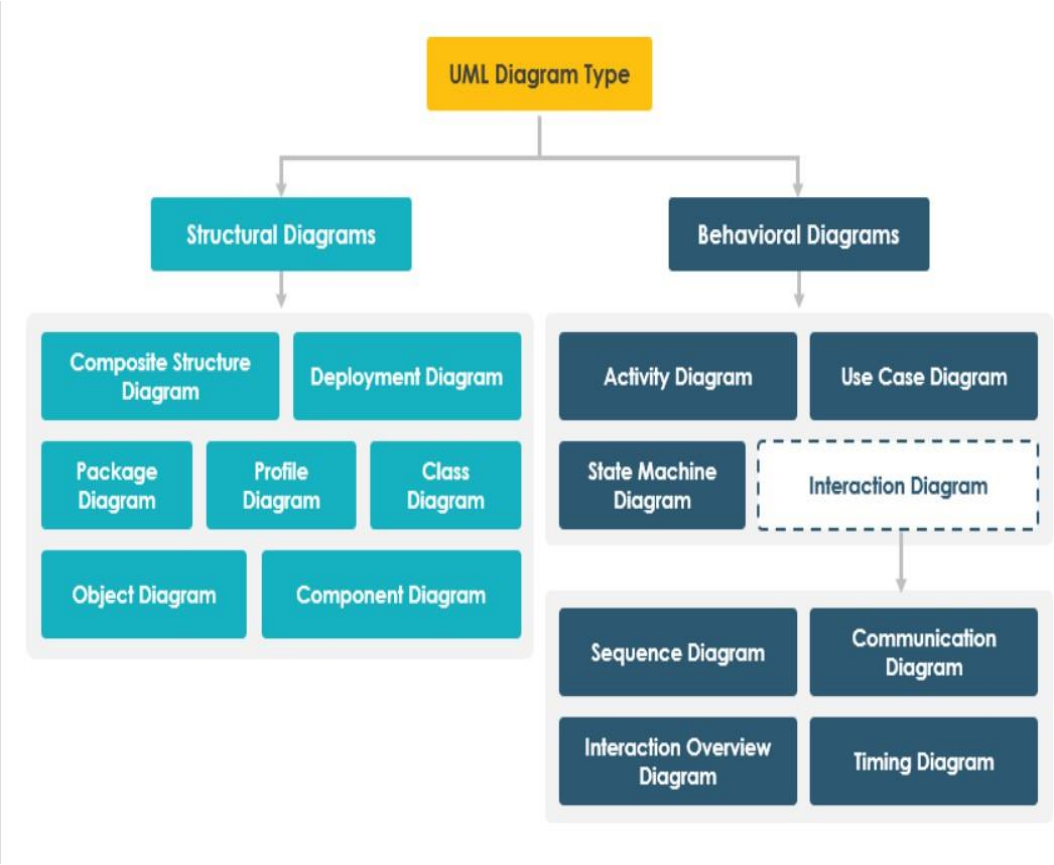
UML

Having a well-defined and expressive notation is important to the process of software development. First, a standard notation makes it possible for an analyst or developer to describe a scenario or formulate an architecture and then unambiguously communicate those decisions to others.

UML (Unified Modeling Language) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

There are two broad categories of diagrams and they are again divided into subcategories −

- Structural Diagrams
- Behavioral Diagrams



Structural Diagrams

The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable.

These static parts are represented by classes, interfaces, objects, components, and nodes. The four structural diagrams are −

- Class diagram

- Object diagram
- Component diagram
- Deployment diagram

Class Diagram

Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations, and collaboration. Class diagrams basically represent the object-oriented view of a system, which is static in nature.

Active class is used in a class diagram to represent the concurrency of the system.

Class diagram represents the object orientation of a system. Hence, it is generally used for development purpose. This is the most widely used diagram at the time of system construction.

Object Diagram

Object diagrams can be described as an instance of class diagram. Thus, these diagrams are more close to real-life scenarios where we implement a system.

Object diagrams are a set of objects and their relationship is just like class diagrams. They also represent the static view of the system.

The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from a practical perspective.

Component Diagram

Component diagrams represent a set of components and their relationships. These components consist of classes, interfaces, or collaborations. Component diagrams represent the implementation view of a system.

During the design phase, software artifacts (classes, interfaces, etc.) of a system are arranged in different groups depending upon their relationship. Now, these groups are known as components.

Finally, it can be said component diagrams are used to visualize the implementation.

Deployment Diagram

Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed.

Deployment diagrams are used for visualizing the deployment view of a system. This is generally used by the deployment team.

Note − If the above descriptions and usages are observed carefully then it is very clear that all the diagrams have some relationship with one another. Component diagrams are dependent upon the classes, interfaces, etc. which are part of class/object diagram. Again, the deployment diagram is dependent upon the components, which are used to make component diagrams.

Behavioral Diagrams

Any system can have two aspects, static and dynamic. So, a model is considered as complete when both the aspects are fully covered.

Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspect can be further described as the changing/moving parts of a system.

UML has the following five types of behavioral diagrams −

- Use case diagram
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram

Use Case Diagram

Use case diagrams are a set of use cases, actors, and their relationships. They represent the use case view of a system.

A use case represents a particular functionality of a system. Hence, use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as **actors**.

Sequence Diagram

A sequence diagram is an interaction diagram. From the name, it is clear that the diagram deals with some sequences, which are the sequence of messages flowing from one object to another.

Interaction among the components of a system is very important from implementation and execution perspective. Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality.

Collaboration Diagram

Collaboration diagram is another form of interaction diagram. It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects and links.

The purpose of collaboration diagram is similar to sequence diagram. However, the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction.

Statechart Diagram

Any real-time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system. Statechart diagram is used to represent the event driven state change of a system. It basically describes the state change of a class, interface, etc. State chart diagram is used to visualize the reaction of a system by internal/external factors.

Activity Diagram

Activity diagram describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched. Activities are nothing but the functions of a system. Numbers of activity diagrams are prepared to capture the entire flow in a system. Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.

Note − Dynamic nature of a system is very difficult to capture. UML has provided features to capture the dynamics of a system from different angles. Sequence diagrams and collaboration diagrams are isomorphic; hence they can be converted from one another without losing any information. This is also true for State chart and activity diagram.

Dynamic view- describes a system over time

Static view -It does not describe the time dependent behavior of the system

USE CASE DAGRAMS

The figure below shows a class diagram from the box office application. The diagram contains part of ticket-selling domain model. It shows several important classes, such as customer, reservation, ticket and performance. Customers may have many reservations, but each reservation is made by one customer. Reservations are two kinds: subscription series and individual reservations. Both reserve tickets; in one case only one ticket in the other case several tickets. Every ticket is part of a subscription series or an individual reservation. But not both. Every performance has many tickets available, each with a unique seat number. A performance can be identified by a show, date and time.

Classes can be described at various levels of precision and concreteness. In the early satges of desing, the model captures the more logical aspects of the problem. In the later stages , the model captures the design decisions and implementation details.

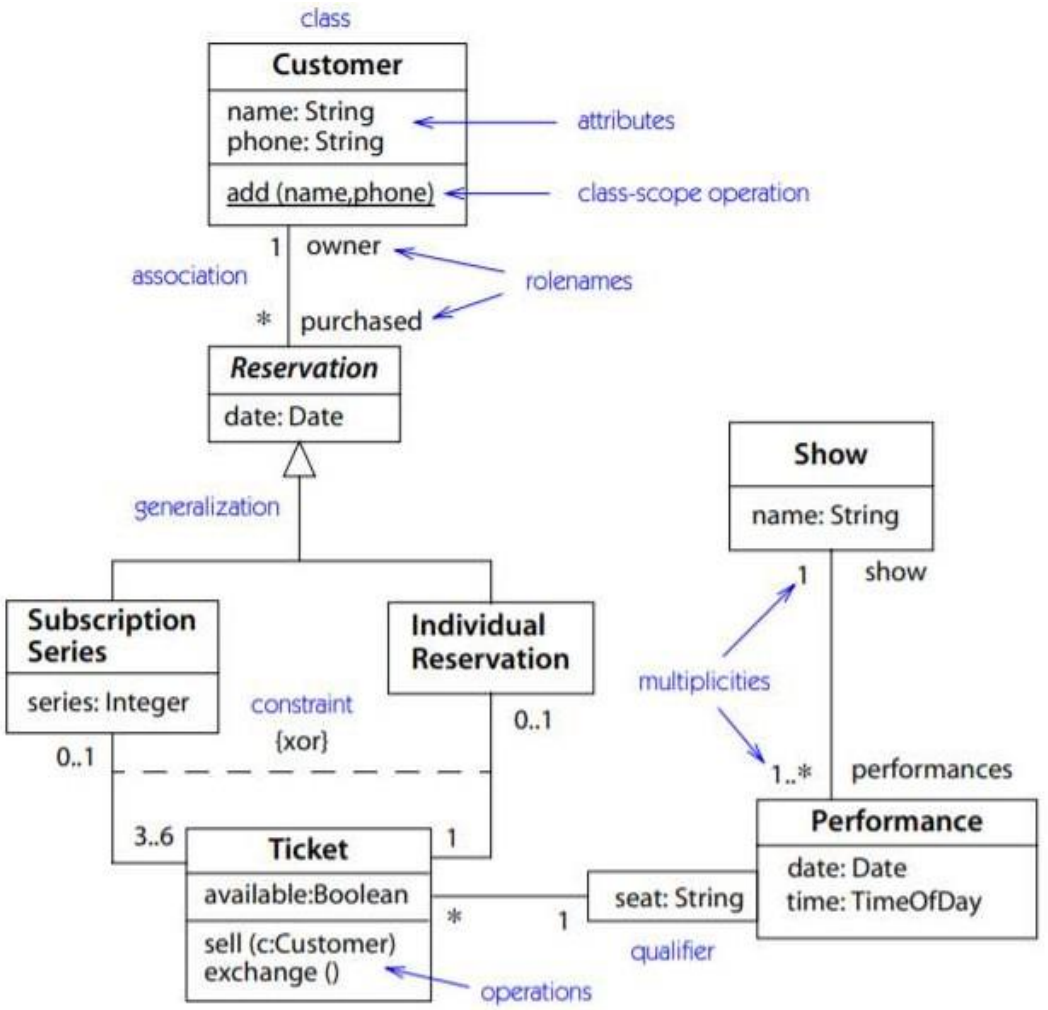


Figure 3-1. *Class diagram*

A use case is a methodology used in system analysis to identify, clarify, and organize system requirements.

A use case diagram models different types of users interact with the system to solve a problem. As such, it describes the goals of the users, the interactions between the users and the system, and the required behavior of the system in satisfying these goals.

A use-case model is a model of all the useful ways to use a system. It allows you to very quickly scope the system – what is included and what is not – and give the team a comprehensive picture of what the system will do. It lets you do this without getting bogged down in the details of the requirements or the internals of the system. With a little experience it is very easy to produce use-case models for even the most complex systems, creating an easily accessible big picture that makes the scope and goals of the system visible to everyone involved.

Its high level and doesn't show any details (DFDLEVEL 0)

Use cases define interactions between external actors and the system to attain particular goals. A use case diagram contains four main components

A use-case diagram consists of a number of model elements. The most important model elements are:

Actor

Actors are usually individuals involved with the system defined according to their roles. The actor can be a human or other external system. something with a behavior or role. e.g. a person. Another system or organization.

Use Case

A use case describes how actors uses a system to accomplish a particular goal. Use cases are typically initiated by a user to fulfill goals describing the activities and variants involved in attaining the goal. **Relationship**

The relationships between and among the actors and the use cases.A line between Actors and use cases.

System Boundary

The system boundary defines the system of interest in relation to the world around it.A box that sets the system scope to use cases.

UML CASE DIAGRAMS are ideal for:

- Representing the goals of system user interaction
- Defining and organizing functional requirements in a system
- Specifying the content of requirements of a system
- Modelling the basic flow of events

FINDING ACTORS

- These are external objects that produce / consume data.
- Must serve as sources and destinations for data
- Must be external to the system e.g humans, machines, external systems, organizational units etc.
- Play business roles in the system

FINDING ACTORS

Ask the following questions:

- Who are the system's primary users? That stimulate system to read
- Who requires system support for daily tasks?
- Who are system's secondary users? respond to system requirements?
- What or how does the system handle?
- Which other system interact with the system in question?
- Do nay entities interacting with the system perform multiple roles as actors?
- Which other entities (human or otherwise) might have an interest in the system's output?

Elements of a use case

- Actor is someone interacting with use case (system functionality) named by a noun
- Actor triggers use case
- Actor has responsibility towards the system(inputs) actor have responsibility towards the

system(outputs)

USE CASE

- System functions (process automated or manual)
- Named by verbs
- Each actor must be linked to a use case, while some use-cases may not be linked to actors.

example: A business wishes to automate some of its sales procedures. Preliminary interviews reveal that there are a number of staff roles in the Sales department. A salesperson can place orders on behalf of customers and check the status of these orders. A technical salesperson has the same duties, but additionally is able to provide customers with detailed technical advice (which we would not expect an ordinary salesperson to be able to do). A sales supervisor is a salesperson, with the additional responsibility of creating new customer accounts and checking their credit-worthiness. It is reasonable to assume that Salesperson is a Generalization of Technical Salesperson and Sales Supervisor, because the technical salesperson and sales supervisor have all the properties of a salesperson, and some extra.

We can construct an outline use case model to show these relationships. We will assume for the sake of simplicity that the use cases are “Place Order”, “Check Order”, “Create Account”, “Check Credit”, and “Technical Advice”. Without generalization, we obtain the following model:

A use case example, without generalization

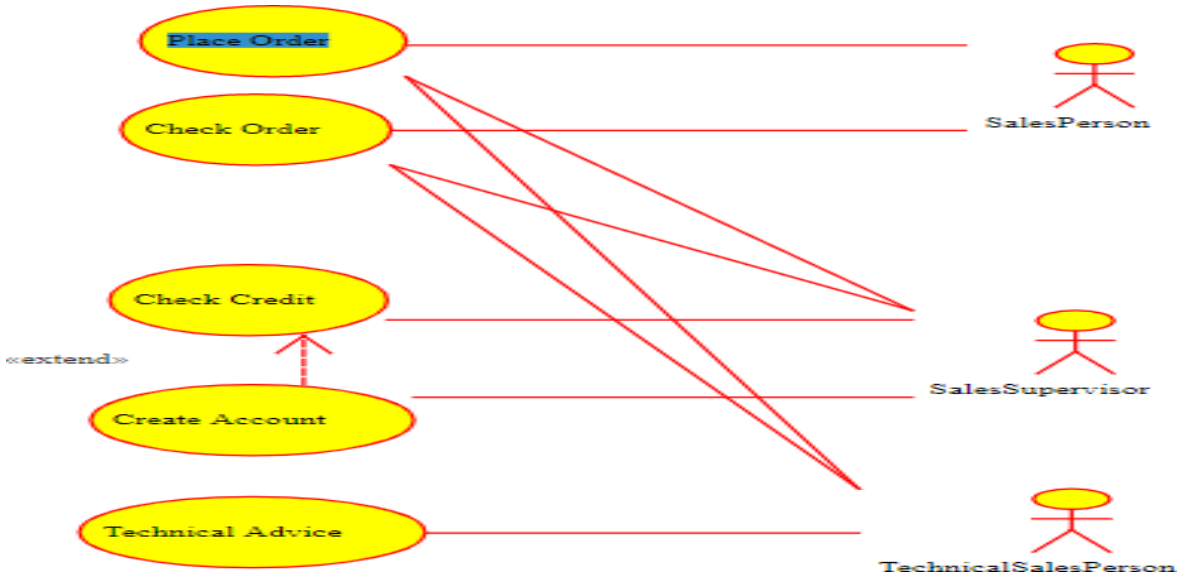
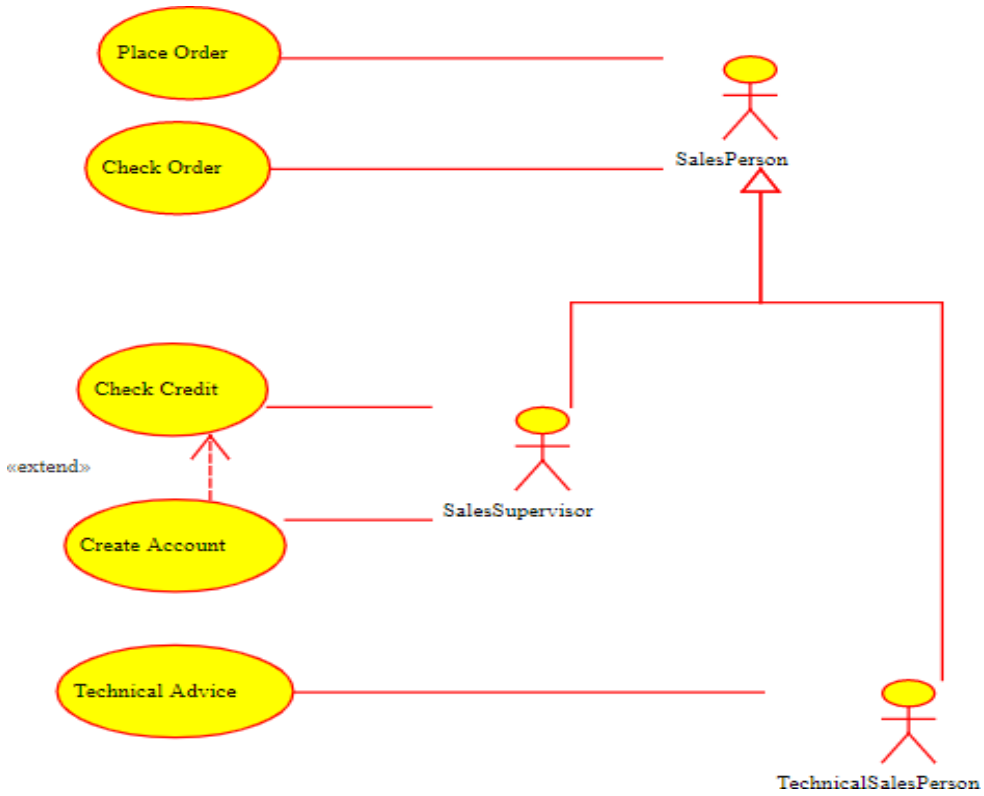


Figure 3.8. Use case example, with generalization



A use case example

A retail business wishes to automate some of its sales procedures. The retailer buys items in bulk from various manufacturers and re-sells them to the public at a profit. Preliminary interviews reveal that there are number of staff roles in the Sales department. A salesperson can place orders on behalf of customers and check the status of these orders. A technical salesperson has the same duties, but additionally is able to provide customers with detailed technical advice (which we would not expect an ordinary salesperson to be able to do). A sales supervisor is a salesperson, with the additional responsibility of creating new customer accounts and checking their credit-worthiness. A dispatcher is responsible for collecting the goods ordered from the warehouse and packing them for dispatch to the customer. To assist in this operation, the computer system should be able to produce a list of unpacked orders as well as delete the orders from the list that the dispatcher has packed. All staff are able to find general details of the products stocked, including stock levels and locations in the warehouse. A re-ordering clerk is responsible for finding out which products are out of stock in the warehouse, and placing orders for these products from the manufacturers. If these products are required to satisfy an outstanding order, they are considered to be "priority" products, and are ordered first. The system should be able to advise the re-order clerk of which products are "priority" products. A stock clerk is responsible for placing items that arrive from manufacturers in their correct places in the warehouse. To do this the clerk needs to be able to find the correct warehouse location for each product from the computer system. Currently, the same person in the business plays the roles of stock clerks and re-order clerk.

Brief use case specifications

- **Check Stock**: Allows a user to check the levels of stock of any item held in the warehouse, and where that item is shelved. A particularly important scenario is that of obtaining a list a stock items for which the stock level is zero, that is, of which there is no stock in the warehouse.

- **Place Order**: A salesperson places an order on behalf of a client. This has the effect of making information about the order available to Dispatch Service. The order remains on the system until it has been packed and dispatched.

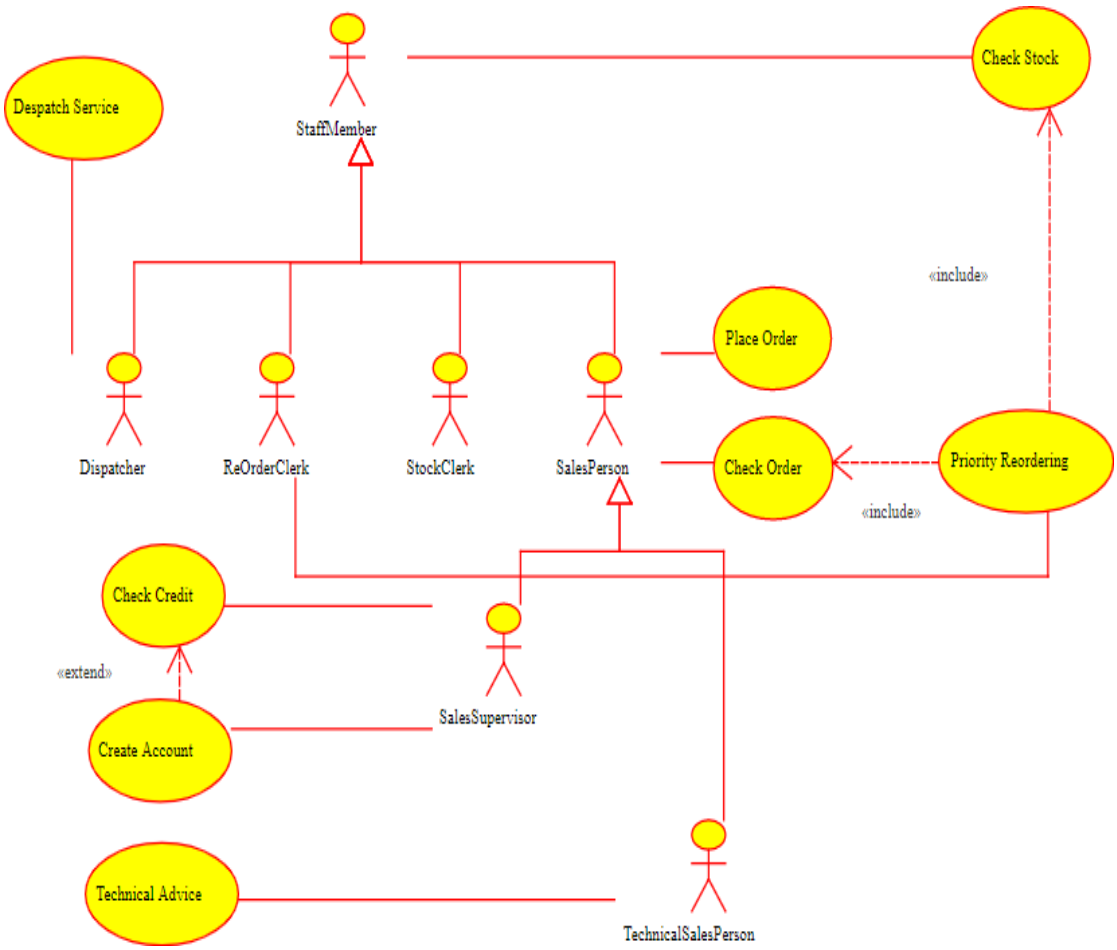
- **Dispatch Service**: Allows a list of outstanding orders to be obtained, and updated when orders are packed. An order is not available to this service if it cannot be satisfied because there is not enough stock in the warehouse.

- **Priority Reordering**: Obtains a list of items that need to be re-ordered urgently, as the business cannot satisfy its own orders without them. This use case makes use of Check Stock (to determine if

an item is out of stock) and Check Order (to determine what stock is required to satisfy all current orders)

- **Check Order**: Obtains details about any outstanding orders, including what stock items are required to satisfy the order. This use case is used by salespeople to advise customers, and by the Priority Reorder use case to determine which items of stock must be replaced quickly.
- **Check Credit**: Used to find out whether it is safe to extend credit facilities to a client. This use case refers to external credit reference agencies (not shown).
- **Create Account**: Used to register a new customer. If a customer asks for credit facilities, this use case includes Check Credit. Otherwise it doesn't have to.
- **Technical Advice**: Provides technical specifications for selected products. Used by technical sales staff to provide advice to customers.

SOLUTION



CASE STUDY

A Library Management System is a software built to handle the primary housekeeping functions of a library. Libraries rely on library management systems to manage asset collections as well as relationships with their members. Library management systems help libraries keep track of the books and their checkouts, as well as members' subscriptions and profiles.

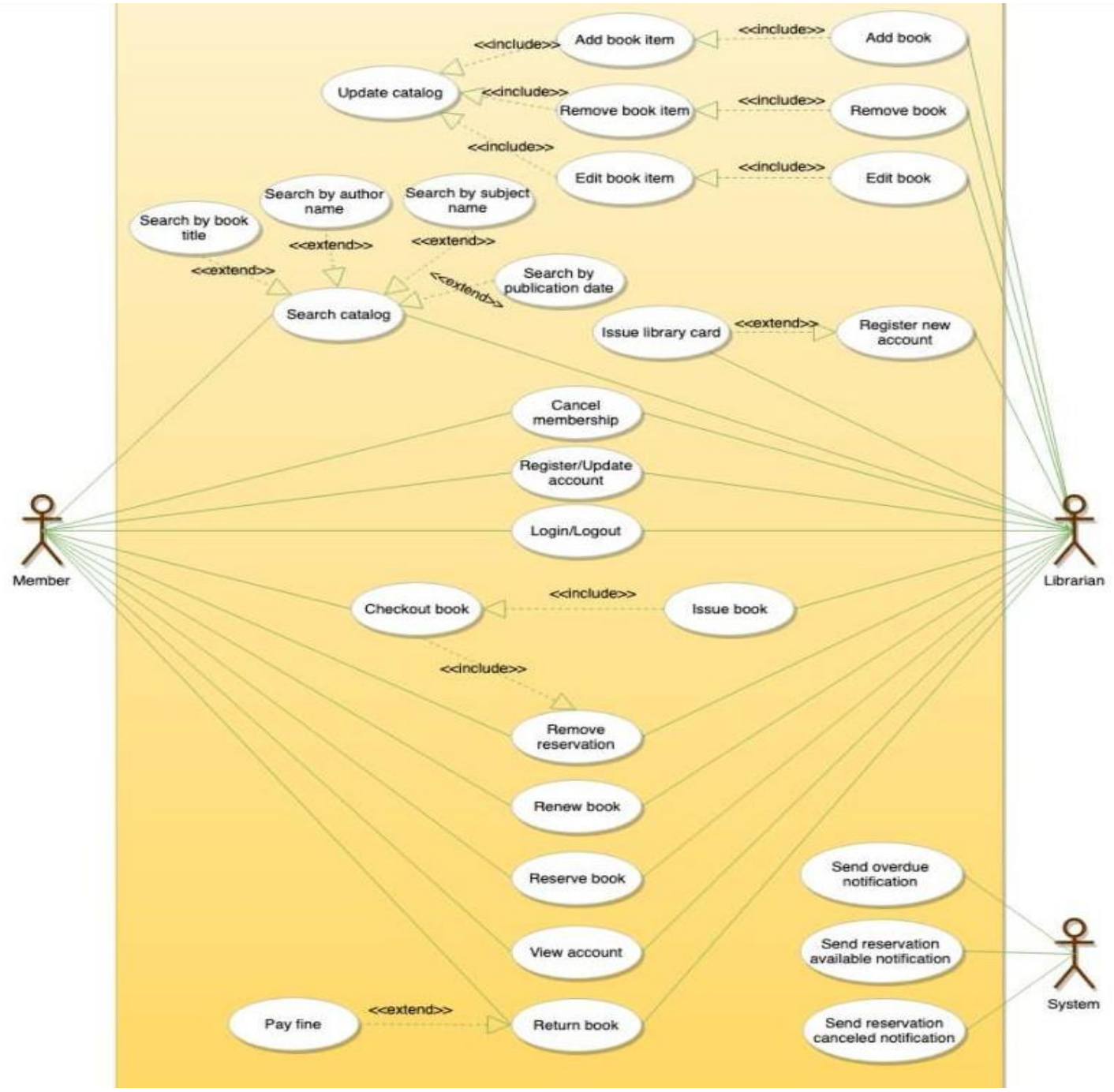
Library management systems also involve maintaining the database for entering new books and recording books that have been borrowed with their respective due dates

We have three main actors in our system:

- **Librarian:** Mainly responsible for adding and modifying books, book items, and users. The Librarian can also issue, reserve, and return book items.
- **Member:** All members can search the catalog, as well as check-out, reserve, renew, and return a book.
- **System:** Mainly responsible for sending notifications for overdue books, canceled reservations, etc.

Here are the top use cases of the Library Management System:

- **Add/Remove/Edit book:** To add, remove or modify a book or book item.
- **Search catalog:** To search books by title, author, subject or publication date.
- **Register new account/cancel membership:** To add a new member or cancel the membership of an existing member.
- **Check-out book:** To borrow a book from the library.
- **Reserve book:** To reserve a book which is not currently available.
- **Renew a book:** To re-borrow an already checked-out book.
- **Return a book:** To return a book to the library which was issued to a member.



Places to look for actors :

- Who uses the system?
- Who gets information from the system?
- Who provides information to the system?
- What other systems use the system?
- Who installs, starts up, or maintain the system?

Relationship in use cases:

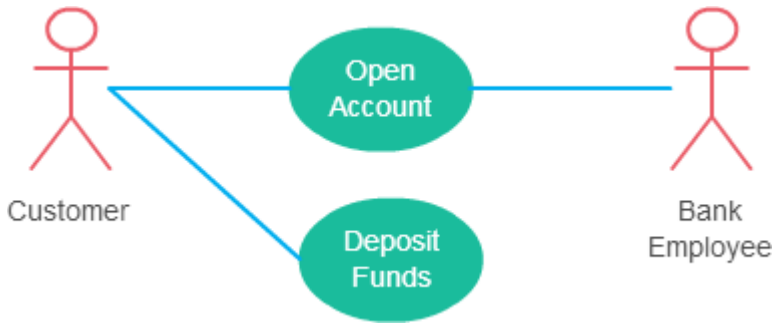
There can be 5 relationship types in a use case diagram.

- Association between actor and use case
- Generalization of an actor
- Extend between two use cases
- Include between two use cases
- Generalization of a use case

Association Between Actor and Use Case

This one is straightforward and present in every use case diagram. Few things to note.

- An actor must be associated with at least one use case.
- An actor can be associated with multiple use cases.
- Multiple actors can be associated with a single use case.

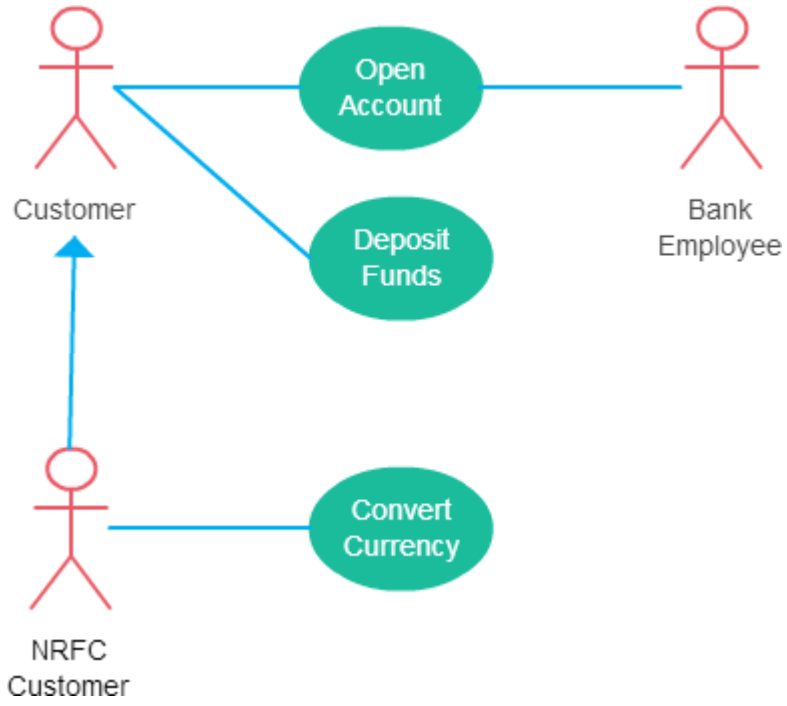


Different ways association relationship appears in use case diagrams

Check out the use case diagram guidelines for other things to consider when adding an actor.

Generalization of an Actor

Generalization of an actor means that one actor can inherit the role of the other actor. The descendant inherits all the use cases of the ancestor. The descendant has one or more use cases that are specific to that role. Let's expand the previous use case diagram to show the generalization of an actor.



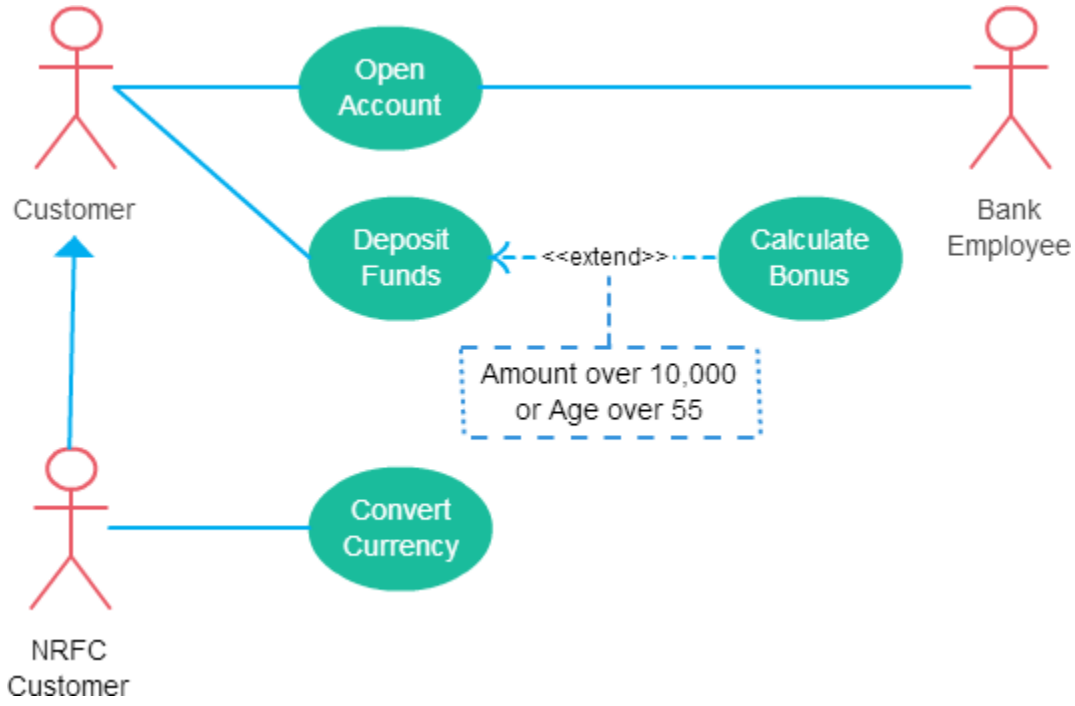
A generalized actor in an use case diagram

Extend Relationship Between Two Use Cases

Many people confuse the extend relationship in use cases. As the name implies it extends the base use case and adds more functionality to the system. Here are a few things to consider when using the <<**extend**>> relationship.

- **The extending use case is dependent on the extended (base) use case**. In the below diagram the "Calculate Bonus" use case doesn't make much sense without the "Deposit Funds" use case.
- **The extending use case is usually optional** and can be triggered conditionally. In the diagram, you can see that the extending use case is triggered only for deposits over 10,000 or when the age is over 55.
- **The extended (base) use case must be meaningful on its own**. This means it should be independent and must not rely on the behavior of the extending use case.

Lets expand our current example to show the <<extend>> relationship.



Extend relationship in use case diagrams

Although extending use case is optional most of the time it is not a must. An extending use case can have non-optional behavior as well. This mostly happens when your modeling complex behaviors.

For example, in an accounting system, one use case might be “Add Account Ledger Entry”. This might have extending use cases “Add Tax Ledger Entry” and “Add Payment Ledger Entry”. These are not optional but depend on the account ledger entry. Also, they have their own specific behavior to be modeled as a separate use case.

Extend is used when a use case adds steps to another first-class use case.

For example, imagine "Withdraw Cash" is a use case of an Automated Teller Machine (ATM). "Assess Fee" would extend Withdraw Cash and describe the *conditional* "extension point" that is instantiated when the ATM user doesn't bank at the ATM's owning institution. Notice that the basic "Withdraw Cash" use case stands on its own, without the extension.

Include is used to extract use case fragments that are *duplicated* in multiple use cases. The included use case cannot stand alone and the original use case is not complete without the included one. This should be used sparingly and only in cases where the duplication is significant and exists by design (rather than by coincidence).

For example, the flow of events that occurs at the beginning of every ATM use case (when the user puts in their ATM card, enters their PIN, and is shown the main menu) would be a good candidate for an include.

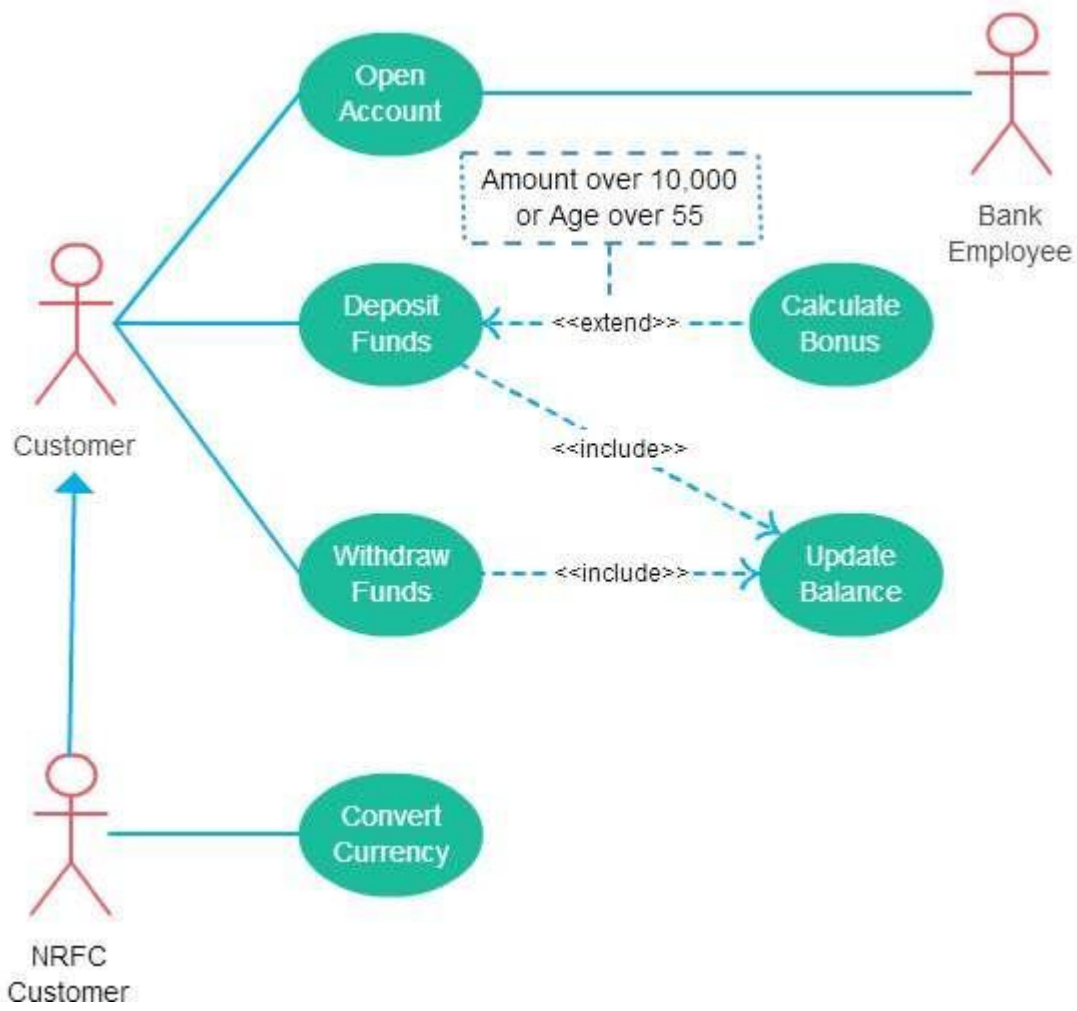
Include Relationship between Two Use Cases

Include relationship show that the behavior of the included use case is part of the including (base) use case.

The main reason for this is to reuse common actions across multiple use cases. In some situations, this is done to simplify complex behaviors. Few things to consider when using the <<include>> relationship.

- The base use case is incomplete without the included use case.
- The included use case is mandatory and not optional.

Lest expand our banking system use case diagram to show include relationships as well.



Includes is usually used to model common behavior

For some further reading regarding the difference between extend and include relationships in use case diagrams

Generalization of a Use Case

This is similar to the generalization of an actor. The behavior of the ancestor is inherited by the descendant. This is used when there is common behavior between two use cases and also specialized behavior specific to each use case.
For example, in the previous banking example, there might be a use case called “Pay Bills”. This can be generalized to “Pay by Credit Card”, “Pay by Bank Balance” etc.

I hope you found this article about **use case relationships** helpful and useful. You can use our diagramming tool to easily create use case diagrams online. As always if you have any questions don’t hesitate to ask them in the comments section.

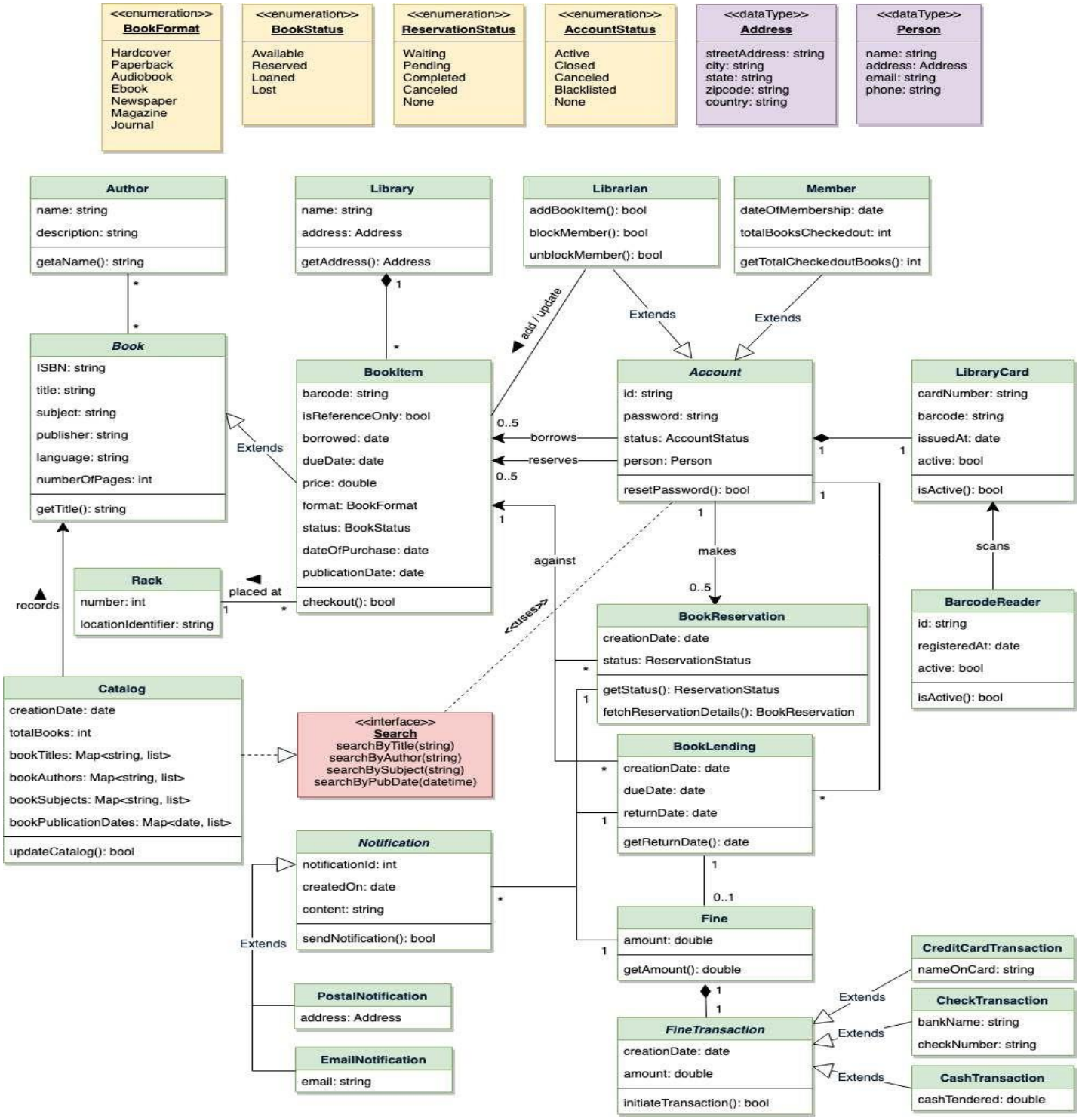
Class diagram

#

Here are the main classes of our Library Management System:

- **Library:** The central part of the organization for which this software has been designed. It has attributes like ‘Name’ to distinguish it from any other libraries and ‘Address’ to describe its location.
- **Book:** The basic building block of the system. Every book will have ISBN, Title, Subject, Publishers, etc.
- **BookItem:** Any book can have multiple copies, each copy will be considered a book item in our system. Each book item will have a unique barcode.
- **Account:** We will have two types of accounts in the system, one will be a general member, and the other will be a librarian.
- **LibraryCard:** Each library user will be issued a library card, which will be used to identify users while issuing or returning books.
- **BookReservation:** Responsible for managing reservations against book items.
- **BookLending:** Manage the checking-out of book items.

- **Catalog:** Catalogs contain list of books sorted on certain criteria. Our system will support searching through four catalogs: Title, Author, Subject, and Publish-date.
- **Fine:** This class will be responsible for calculating and collecting fines from library members.
- **Author:** This class will encapsulate a book author.
- **Rack:** Books will be placed on racks. Each rack will be identified by a rack number and will have a location identifier to describe the physical location of the rack in the library.
- **Notification:** This class will take care of sending notifications to library members.


<<enumeration>>
BookFormat
Hardcover
Paperback
Audiobook
Ebook
Newspaper
Magazine
Journal
<<enumeration>>
BookStatus
Available
Reserved
Loaned
Lost
<<enumeration>>
ReservationStatus
Waiting
Pending
Completed
Canceled
None
<<enumeration>>
AccountStatus
Active
Closed
Canceled
Blacklisted
None
<<dataType>>
Address
streetAddress: string
city: string
state: string
zipcode: string
country: string
<<dataType>>
Person
name: string
address: Address
email: string
phone: string
Author
name: string
description: string
getaName(): string
Library
name: string
address: Address
getAddress(): Address
Librarian
addBookItem(): bool
blockMember(): bool
unblockMember(): bool
Member
dateOfMembership: date
totalBooksCheckedout: int
getTotalCheckedoutBooks(): int
Book
ISBN: string
title: string
subject: string
publisher: string
language: string
numberOfPages: int
getTitle(): string
BookItem
barcode: string
isReferenceOnly: bool
borrowed: date
dueDate: date
price: double
format: BookFormat
status: BookStatus
dateOfPurchase: date
publicationDate: date
checkout(): bool
Account
id: string
password: string
status: AccountStatus
person: Person
resetPassword(): bool
LibraryCard
cardNumber: string
barcode: string
issuedAt: date
active: bool
isActive(): bool
Rack
number: int
locationIdentifier: string
BarcodeReader
id: string
registeredAt: date
active: bool
isActive(): bool
BookReservation
creationDate: date
status: ReservationStatus
getStatus(): ReservationStatus
fetchReservationDetails(): BookReservation
Catalog
creationDate: date
totalBooks: int
bookTitles: Map<string, list>
bookAuthors: Map<string, list>
bookSubjects: Map<string, list>
bookPublicationDates: Map<date, list>
updateCatalog(): bool
<<interface>>
Search
searchByTitle(string)
searchByAuthor(string)
searchBySubject(string)
searchByPubDate(datetime)
BookLending
creationDate: date
dueDate: date
returnDate: date
getReturnDate(): date
Notification
notificationId: int
createdOn: date
content: string
sendNotification(): bool
Fine
amount: double
getAmount(): double
PostalNotification
address: Address
EmailNotification
email: string
FineTransaction
creationDate: date
amount: double
initiateTransaction(): bool
CreditCardTransaction
nameOnCard: string
CheckTransaction
bankName: string
checkNumber: string
CashTransaction
cashTendered: double
add / update
Extends
Extends
Extends
borrows
reserves
0..5
0..5
placed at
records
against
makes
0..5
<<uses>>
scans
Extends
Extends
Extends
Extends

ACTIVITY DIAGRAMS

It is basically a flow chart to represent the flow of one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. The flow can be sequential, branched or concurrent. Activity diagrams deal with all types of flow control by using different elements such as Fork, Join etc. Activity diagrams captures the dynamic behavior of the system. It shows massage flows from one activity to another.

The purpose of an activity diagram can be described as:

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

Activity Diagrams can be used for:

- Modelling business Requirements
- Modeling workflow by using activities
- High level understanding of systems functionality
- Investigating business requirements at a later stage.

Basic Activity Diagrams NOTATIONS and SYMBOLS

The various components used in the diagram and the standard notations are explained below.

Activity Diagram Notations –

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

Figure – notation for initial state or start state

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

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

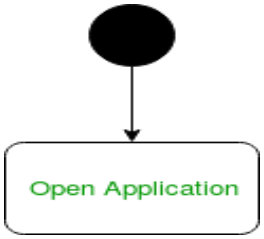


Figure – initial state symbol being used

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

Figure – notation for an activity state

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

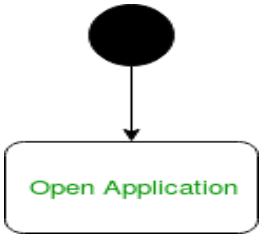


Figure – activity state symbol being used

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

Figure – notation for control Flow

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

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

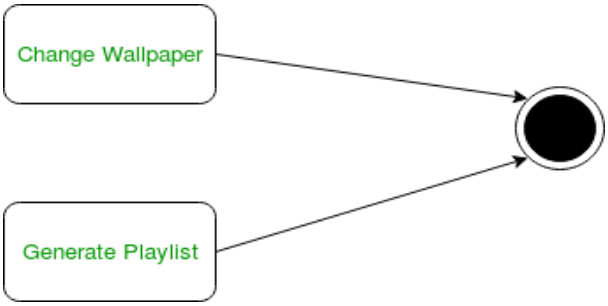


Figure – using action flows for transitions

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

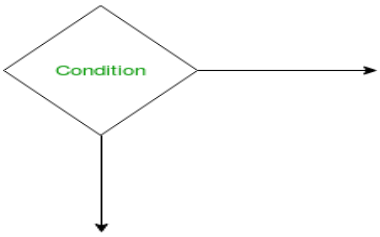


Figure – notation for decision node

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

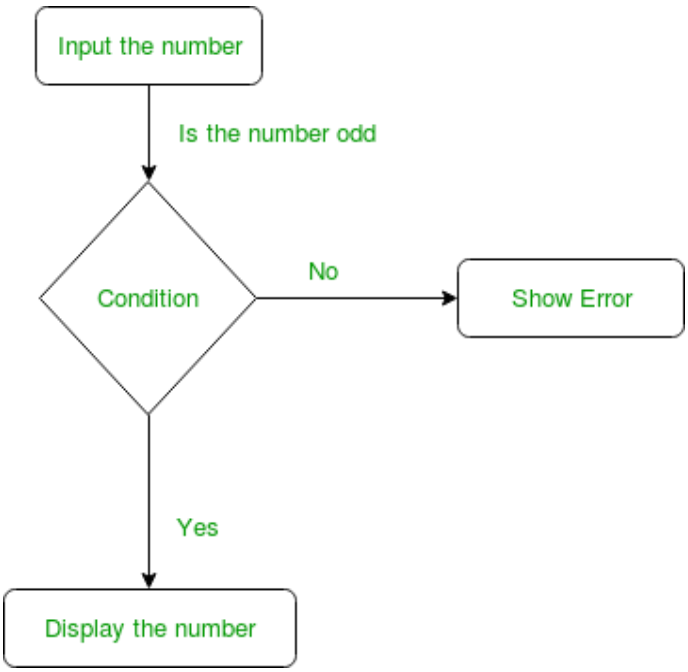


Figure – an activity diagram using decision node

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

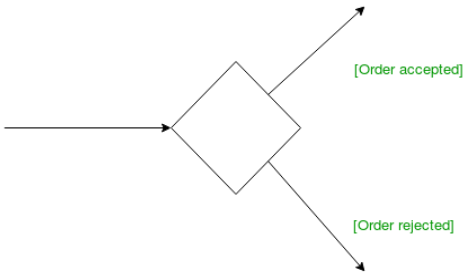


Figure – guards being used next to a decision node

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

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

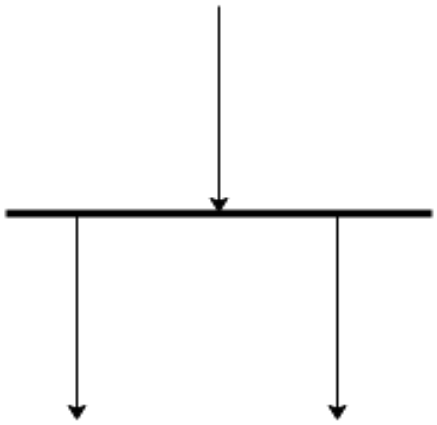


Figure – fork notation

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

For example: In the example below, the activity of making coffee can be split into two concurrent activities and hence we use the fork notation.

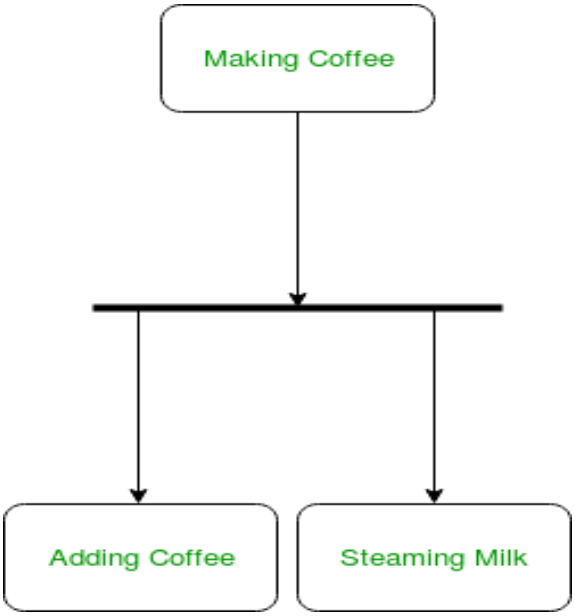


Figure – a diagram using fork

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

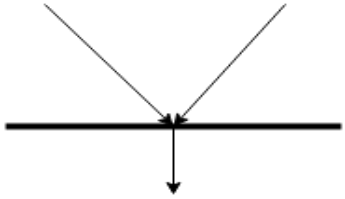


Figure – join notation

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

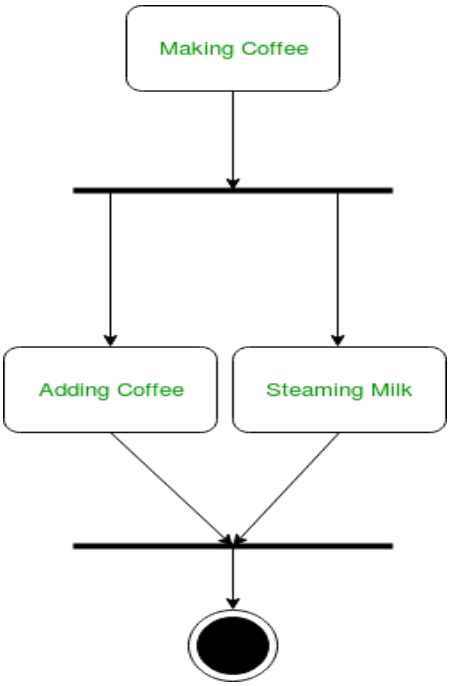


Figure – a diagram using join notation

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

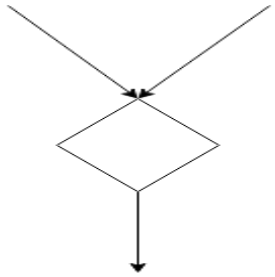


Figure – merge notation

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

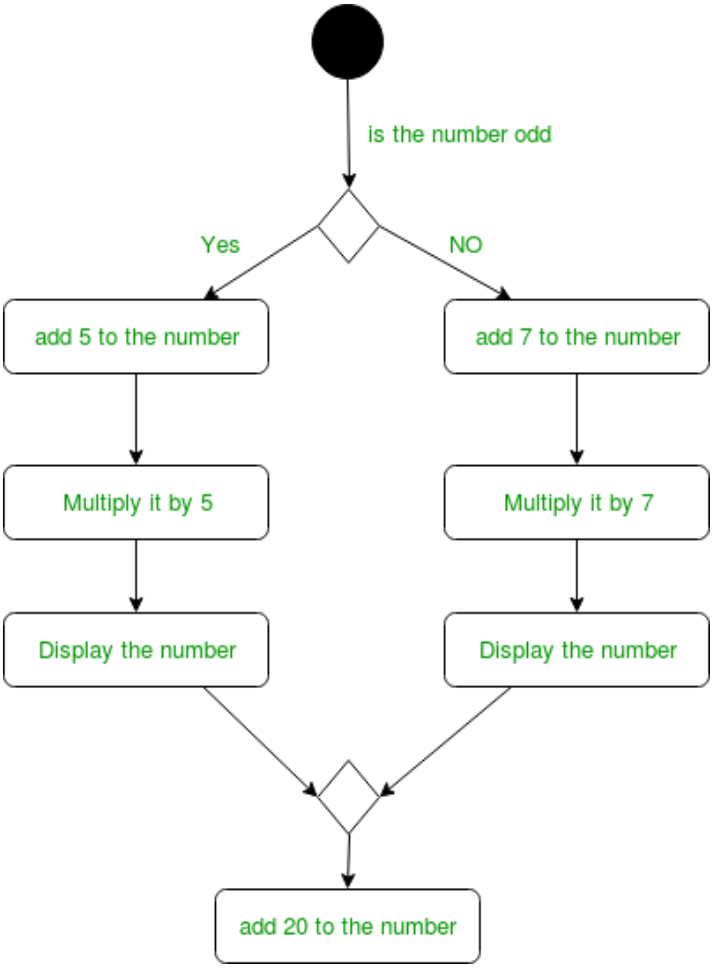


Figure – an activity diagram using merge notation

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

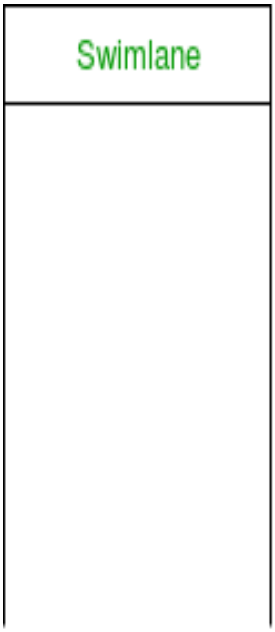


Figure – swimlanes notation

We use a rectangular column to represent a swimlane as shown in the figure above.

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

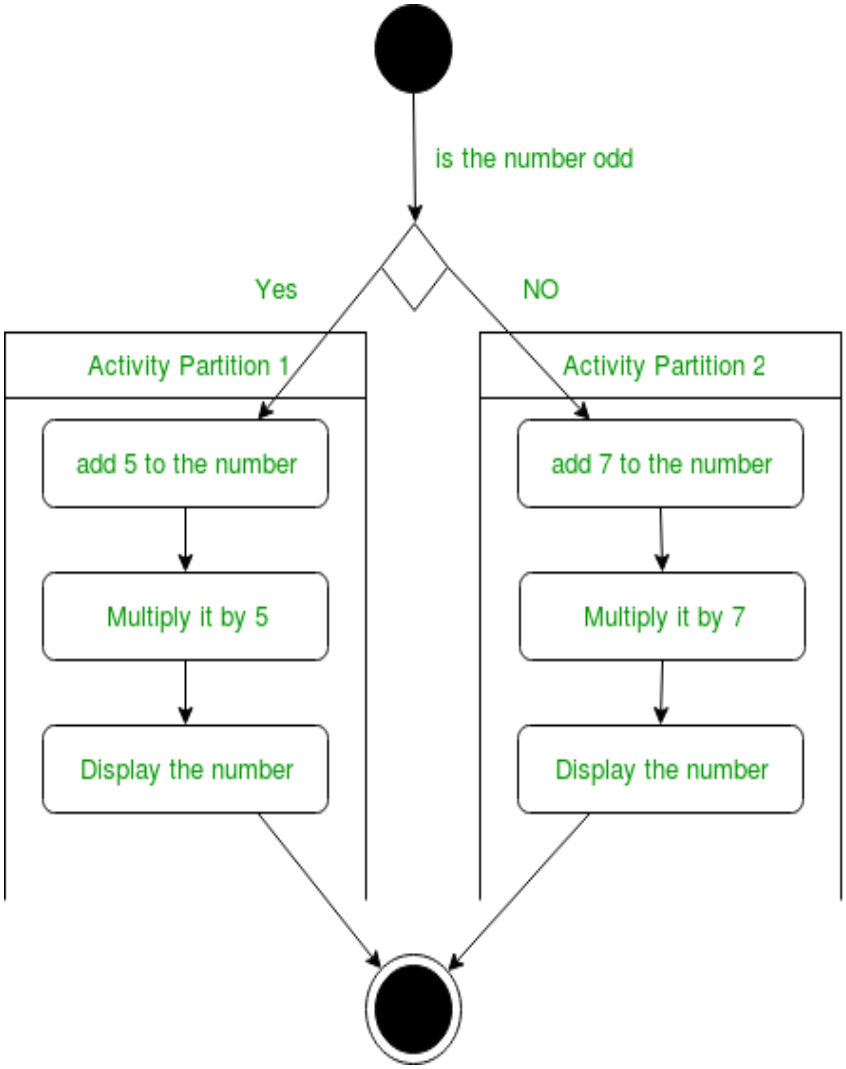


Figure – an activity diagram making use of swimlanes

10. **Time Event –**

Figure – time event notation

We can have a scenario where an event takes some time to complete. We use an hourglass to represent a time event.

For example – Let us assume that the processing of an image takes takes a lot of time. Then it can be represented as shown below.

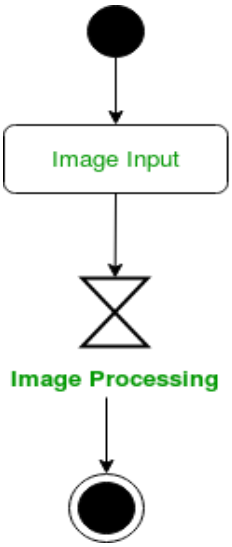


Figure – an activity diagram using time event

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

Figure – notation for final state

How to Draw an activity diagram –

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

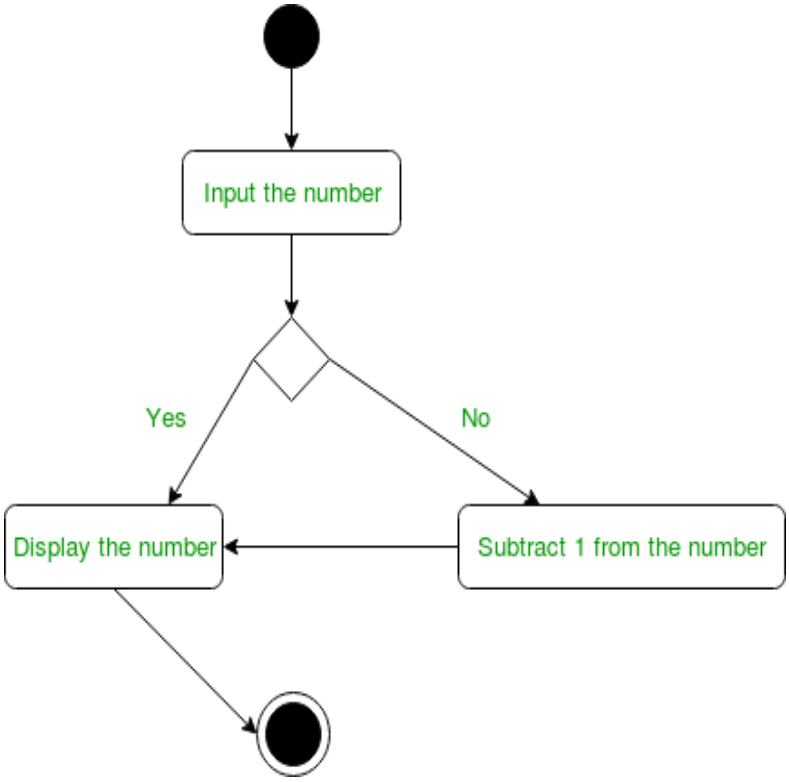


Figure – an activity diagram

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

Difference between an Activity diagram and a Flowchart –

Flowcharts were typically invented earlier than activity diagrams. Non programmers use Flow charts to model workflows. For example: A manufacturer uses a flow chart to explain and illustrate how a particular product is manufactured. We can call a flowchart a primitive version of an activity diagram. Business processes where decision making is involved is expressed using a flow chart.

So, programmers use activity diagrams (advanced version of a flowchart) to depict workflows. An activity diagram is **used by developers** to understand the flow of programs on a high level. It also enables them to figure out constraints and conditions that cause particular events. A flow chart converges into being an activity diagram if complex decisions are being made.

Brevity is the soul of wit. We need to convey a lot of information with clarity and make sure it is short. So an activity diagram helps people on both sides i.e. Businessmen and Developers to interact and understand systems.

Uses of an Activity Diagram –

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

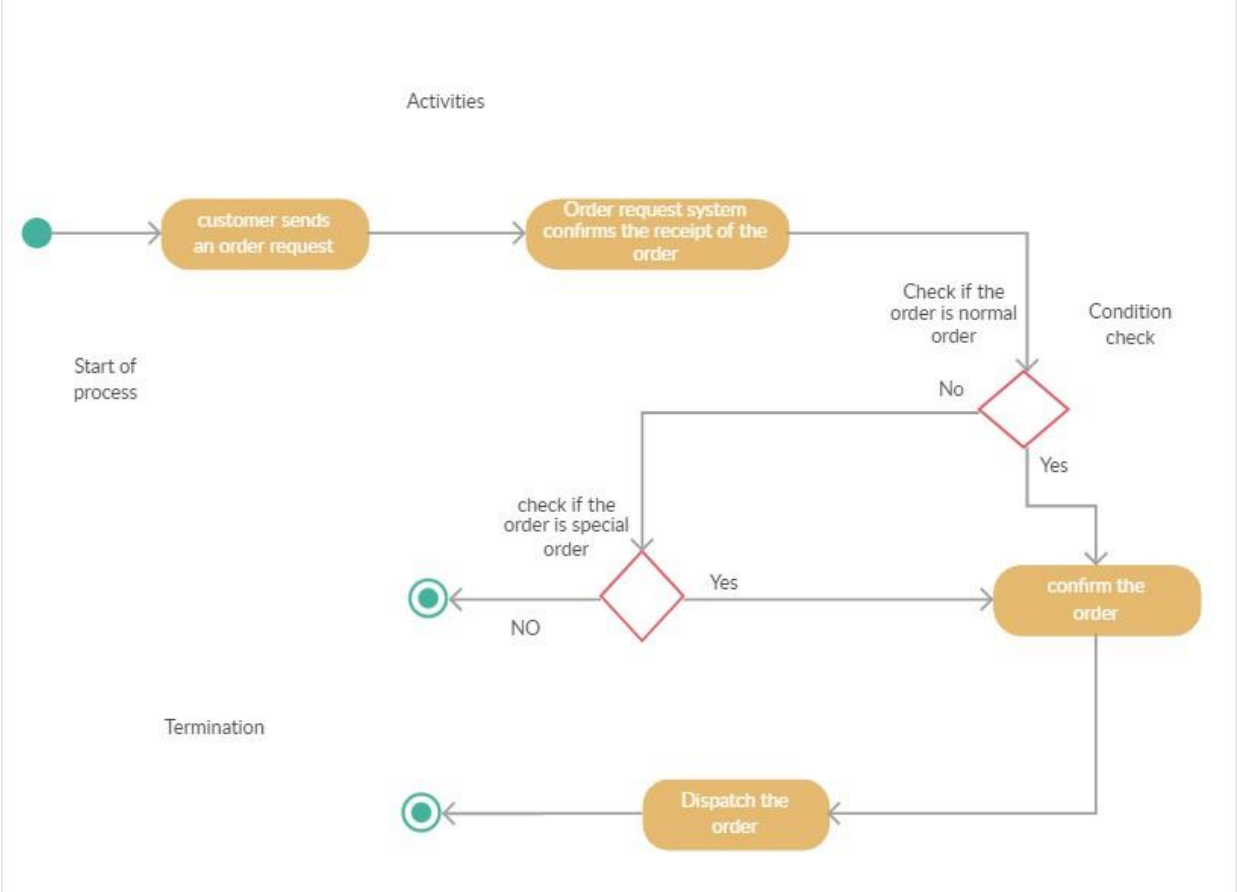
Before drawing an activity diagrams we should identify the following:

- **A**ctivities
- Association
- Conditions
- Constraints

Example of order management system

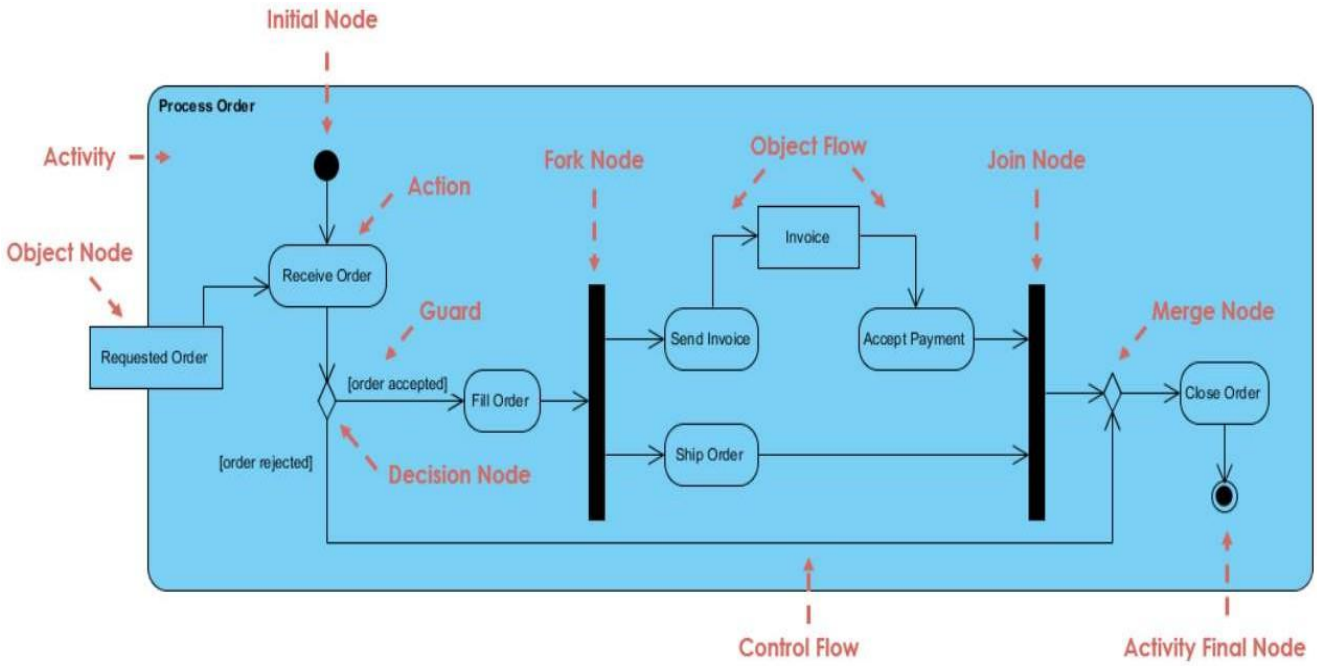
The form main activities:

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

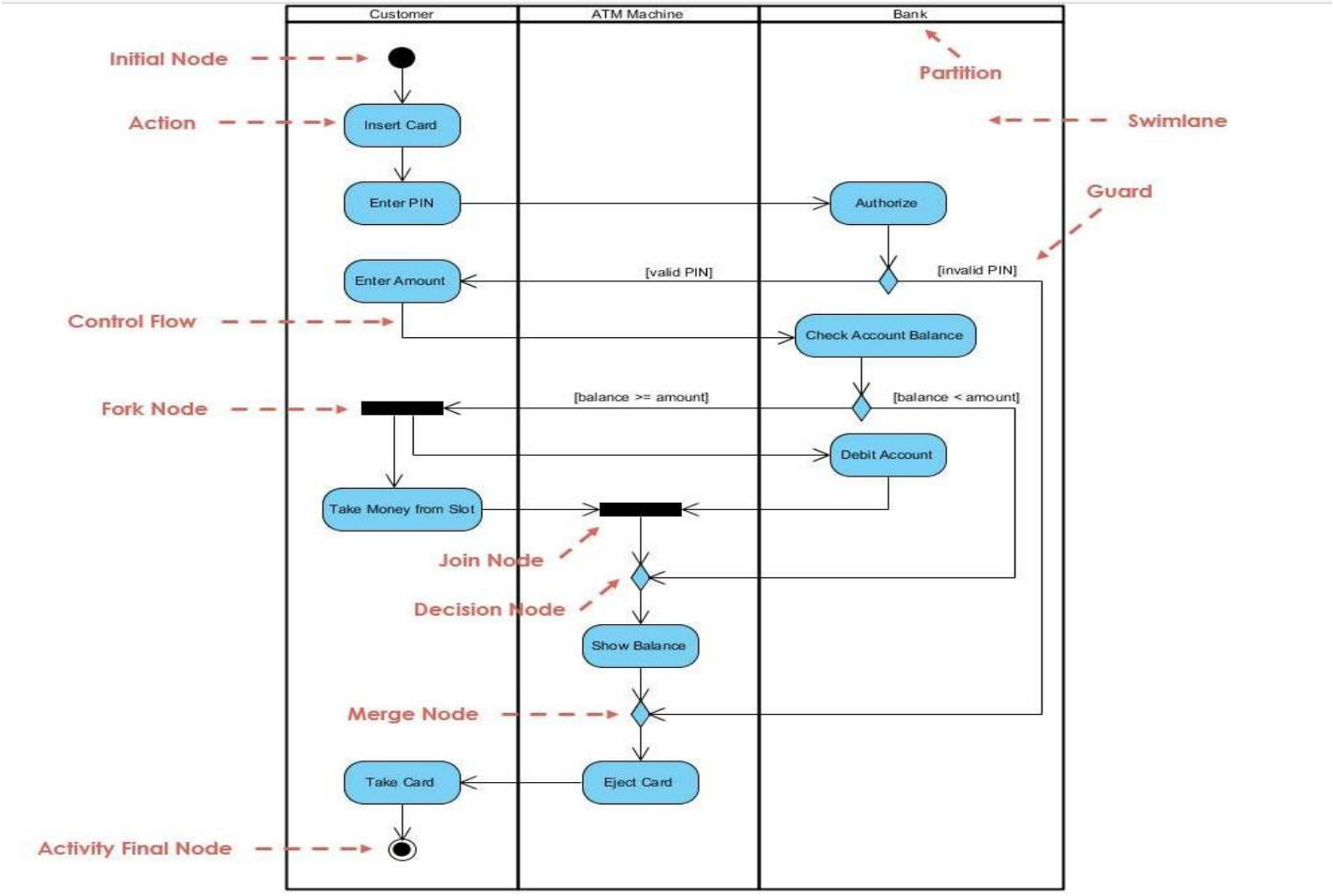


PROCESSING ORDER EXAMPLE

Requested order is in-out parameter of the activity. After order is completed and all required information is filled in , payment is accepted and order is shipped.



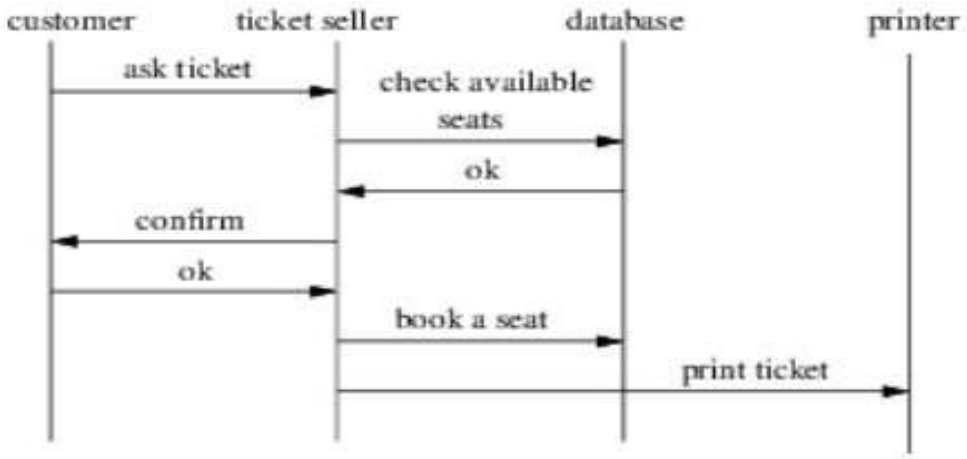
ACTIVITY DIAGRAMS WITH SWIMLANES



SEQUENCE DIAGRAMS

A sequence diagram describes how objects or a group of objects interact within a system. Interacting objects can be classes , program components or real world instances such as customer who is buying a train ticket.

Example: an interaction diagram between a customer, ticket seller, a database and a printer.



An interaction between a customer, ticket seller, database and printer.

Lifelines:

They represent either roles or object instances that participants in the sequence being modelled. Lifelines are drawn as a box with a dashed line descending from the center of the bottom adge.

Basic Sequence Diagram Notations

Class Roles or Participants

Class roles describe the way an object will behave in context. Use the UML object symbol to illustrate class roles, but don't list object attributes.

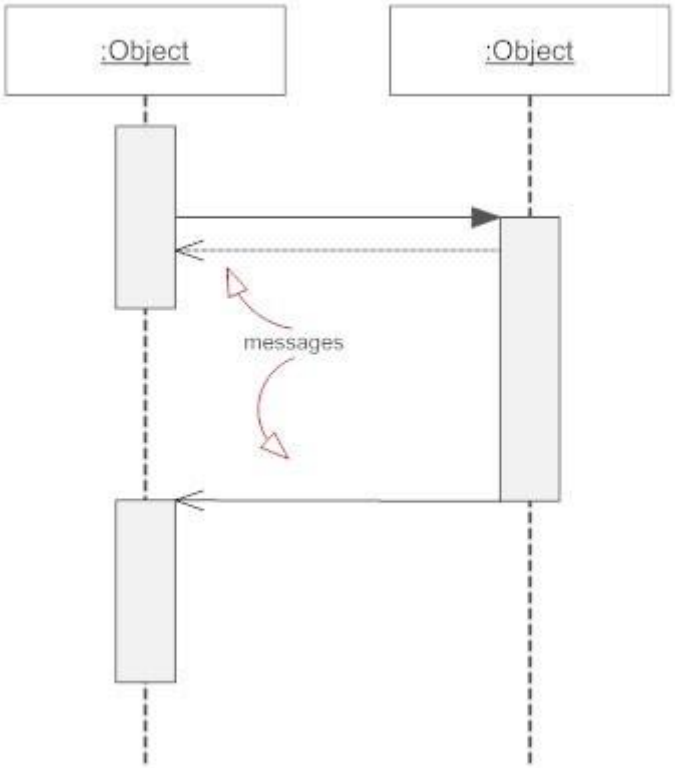
Activation or Execution Occurrence

Activation boxes represent the time an object needs to complete a task. When an object is busy executing a process or waiting for a reply message, use a thin gray rectangle placed vertically on its lifeline.

Activation or Execution Occurrence

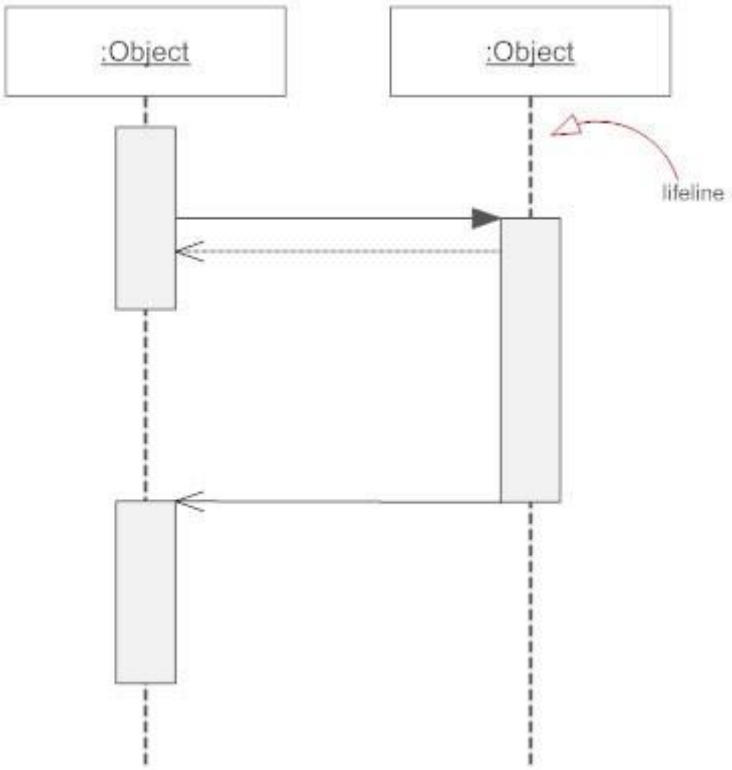
Messages

Messages are arrows that represent communication between objects. Use half-arrowed lines to represent asynchronous messages. Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks. For message types, see below.



Lifelines

Lifelines are vertical dashed lines that indicate the object's presence over time.



Destroying Objects

Objects can be terminated early using an arrow labeled "<< destroy >>" that points to an X. This object is removed from memory. When that object's lifeline ends, you can place an X at the end of its lifeline to denote a destruction occurrence.

Loops

A repetition or loop within a sequence diagram is depicted as a rectangle. Place the condition for exiting the loop at the bottom left corner in square brackets [ ].

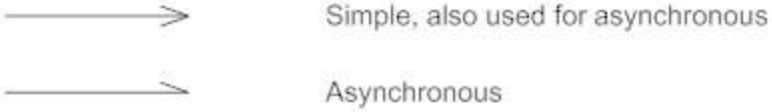
Types of Messages in Sequence Diagrams

Synchronous Message

A synchronous message requires a response before the interaction can continue. It's usually drawn using a line with a solid arrowhead pointing from one object to another.

Asynchronous Message

Asynchronous messages don't need a reply for interaction to continue. Like synchronous messages, they are drawn with an arrow connecting two lifelines; however, the arrowhead is usually open and there's no return message depicted.

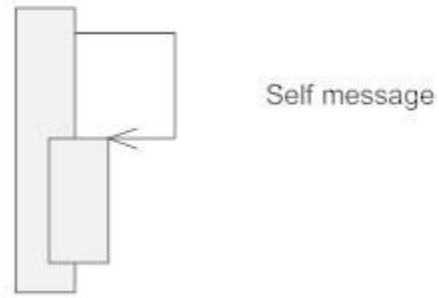


Reply or Return Message

A reply message is drawn with a dotted line and an open arrowhead pointing back to the original lifeline.

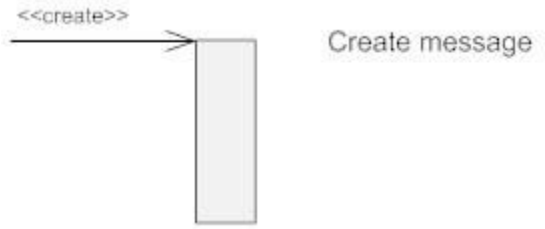
Self Message

A message an object sends to itself, usually shown as a U shaped arrow pointing back to itself.



Create Message

This is a message that creates a new object. Similar to a return message, it's depicted with a dashed line and an open arrowhead that points to the rectangle representing the object created.



Delete Message

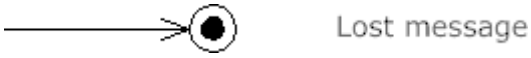
This is a message that destroys an object. It can be shown by an arrow with an x at the end.

Found Message

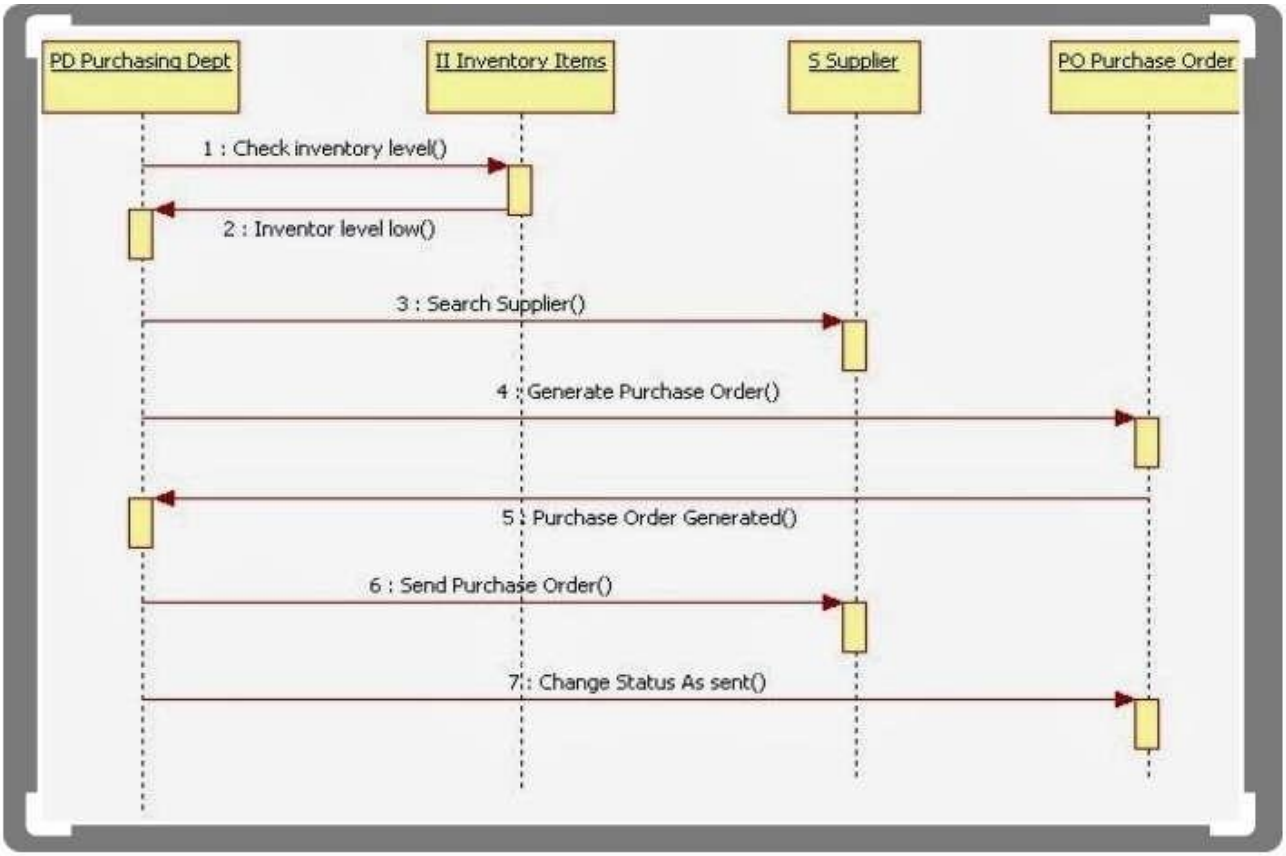
A message sent from an unknown recipient, shown by an arrow from an endpoint to a lifeline.

Lost Message

A message sent to an unknown recipient. It's shown by an arrow going from a lifeline to an endpoint, a filled circle or an x.



Example of a sequence diagram for order processing



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

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

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

Purpose of Statechart Diagrams

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

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

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

Following are the main purposes of using Statechart diagrams −

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

How to Draw a Statechart Diagram?

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

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

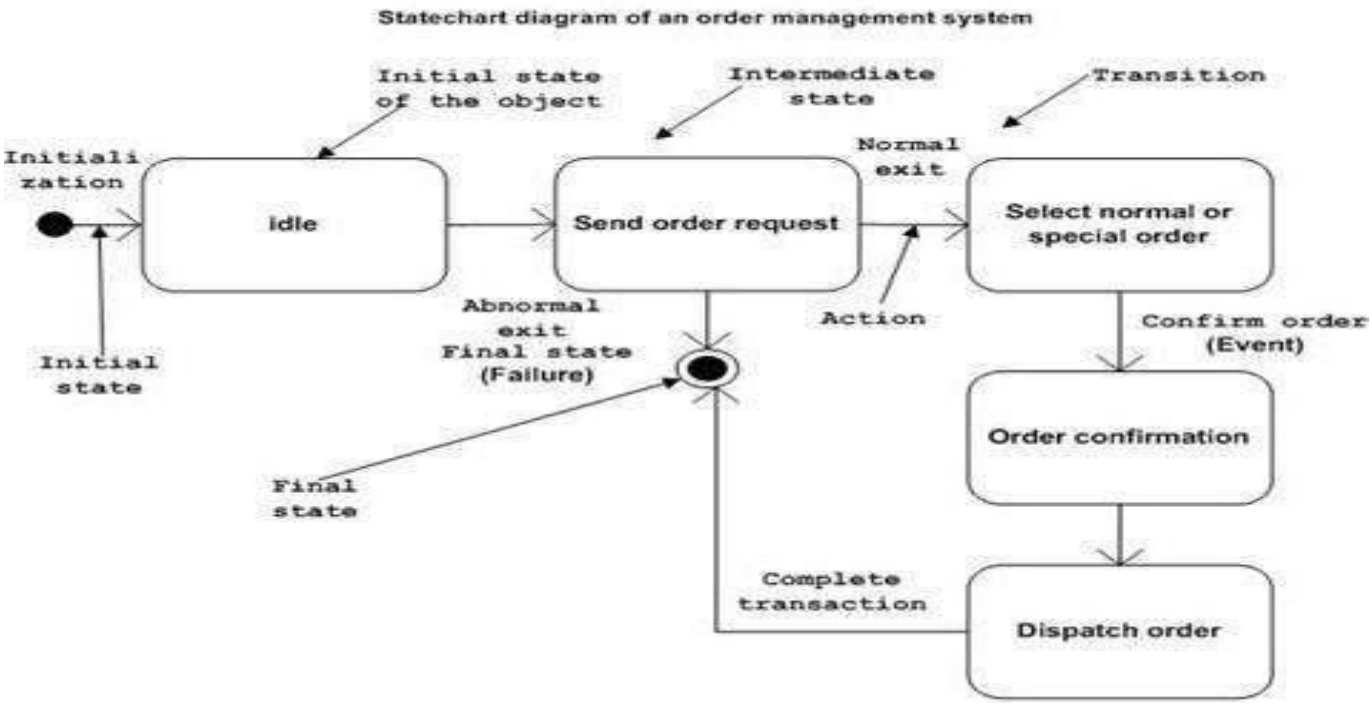
Before drawing a Statechart diagram we should clarify the following points −

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

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

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

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



Where to Use Statechart Diagrams?

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

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

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

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

The main usage can be described as −

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

Recommended Textbooks

1. Arlow, J., and Neustadt, I. (2005). UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd ed), Upper Saddle River NJ. Addison Wesley.
2. Booch G. et al. (2007). Object-Oriented Analysis and Design with applications (3rd ed). Upper saddle River, NJ. Addison Wesley.
3. Deacon, J. (2004). Object-Oriented Analysis and Design: A pragmatic Approach .Upper Saddle River NJ. Addison Wesley.
4. Dennis de Champeaux, Object-Oriented Systems Development, Addison Wesley
5. Larman, C. (2005). Applying UML and patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3ed ed). Upper Saddle River NJ. Pearson Prentice Hall.
6. Oestereich, B. (2002). Developing Software with UML: Object-Oriented Analysis and Design in Practice (2nded.). Upper Saddle River, NJ: Addison Wesley.
7. Richard C. L. and William M. T. (2002). Practical Object-Oriented Development with UML and Java.