What is UML?

UML stands for Unified Modeling Language and is used to model the Object-Oriented Analysis of a software system. UML is a way of visualizing and documenting a software system by using a collection of diagrams, which helps engineers, businesspeople, and system architects understand the behavior and structure of the system being designed.

Benefits of using UML:

1. Helps develop a quick understanding of a software system.
2. UML modeling helps in breaking a complex system into discrete pieces that can be easily understood.
3. UML's graphical notations can be used to communicate design decisions.
4. Since UML is independent of any specific platform or language or technology, it is easier to abstract out concepts.
5. It becomes easier to hand the system over to a new team.

Types of UML Diagrams:

These diagrams are organized into two distinct groups: structural diagrams and behavioral or interaction diagrams.

Structural UML diagrams

- **Class diagram**

Behavioral UML diagrams

- **Use case diagram**
- **Activity diagram**
- **Sequence diagram**

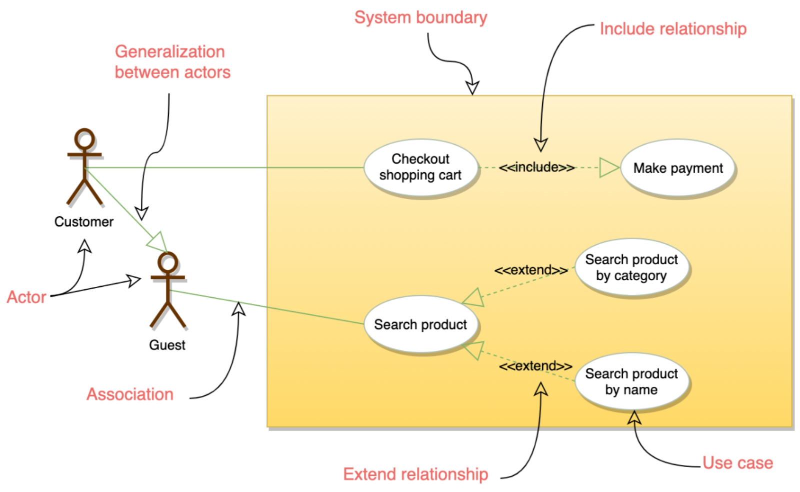
Use Case Diagrams

Use case diagrams describe a set of actions (called use cases) that a system should or can perform in collaboration with one or more external users of the system (called actors). Each use case should provide some observable and valuable result to the actors.

1. Use Case Diagrams describe the high-level functional behavior of the system.
2. It answers what system does from the user point of view.
3. Use case answers ‘What will the system do?’ and at the same time tells us ‘What will the system NOT do?’.

A use case illustrates a unit of functionality provided by the system. The primary purpose of the use case diagram is to help development teams visualize the functional requirements of a system, including the relationship of “actors” to the essential processes, as well as the relationships among different use cases.

To illustrate a use case on a use case diagram, we draw an oval in the middle of the diagram and put the name of the use case in the center of the oval. To show an actor (indicating a system user) on a use-case diagram, we draw a stick figure to the left or right of the diagram.



Sample use-case diagram for online shopping system

The different components of the use case diagram are:

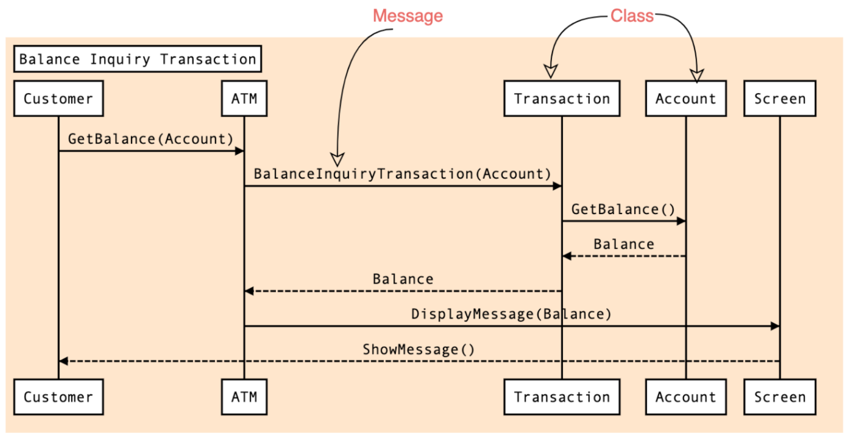
- **System boundary:** A system boundary defines the scope and limits of the system. It is shown as a rectangle that spans all use cases of the system.
- **Actors:** An actor is an entity who performs specific actions. These roles are the actual business roles of the users in a given system. An actor interacts with a use case of the system. For example, in a banking system, the customer is one of the actors.
- **Use Case:** Every business functionality is a potential use case. The use case should list the discrete business functionality specified in the problem statement.
- **Include:** Include relationship represents an invocation of one use case by another use case. From a coding perspective, it is like one function being called by another function.
- **Extend:** This relationship signifies that the extended use case will work exactly like the base use case, except that some new steps will be inserted in the extended use case.

Sequence diagram

Sequence diagrams describe interactions among classes in terms of an exchange of messages over time and are used to explore the logic of complex operations, functions or procedures. In this diagram, the sequence of interactions between the objects is represented in a step-by-step manner.

Sequence diagrams show a detailed flow for a specific use case or even just part of a particular use case. They are almost self-explanatory; they show the calls between the different objects in their sequence and can explain, at a detailed level, different calls to various objects.

A sequence diagram has two dimensions: The vertical dimension shows the sequence of messages in the chronological order that they occur; the horizontal dimension shows the object instances to which the messages are sent.



Sample sequence diagram for ATM balance inquiry

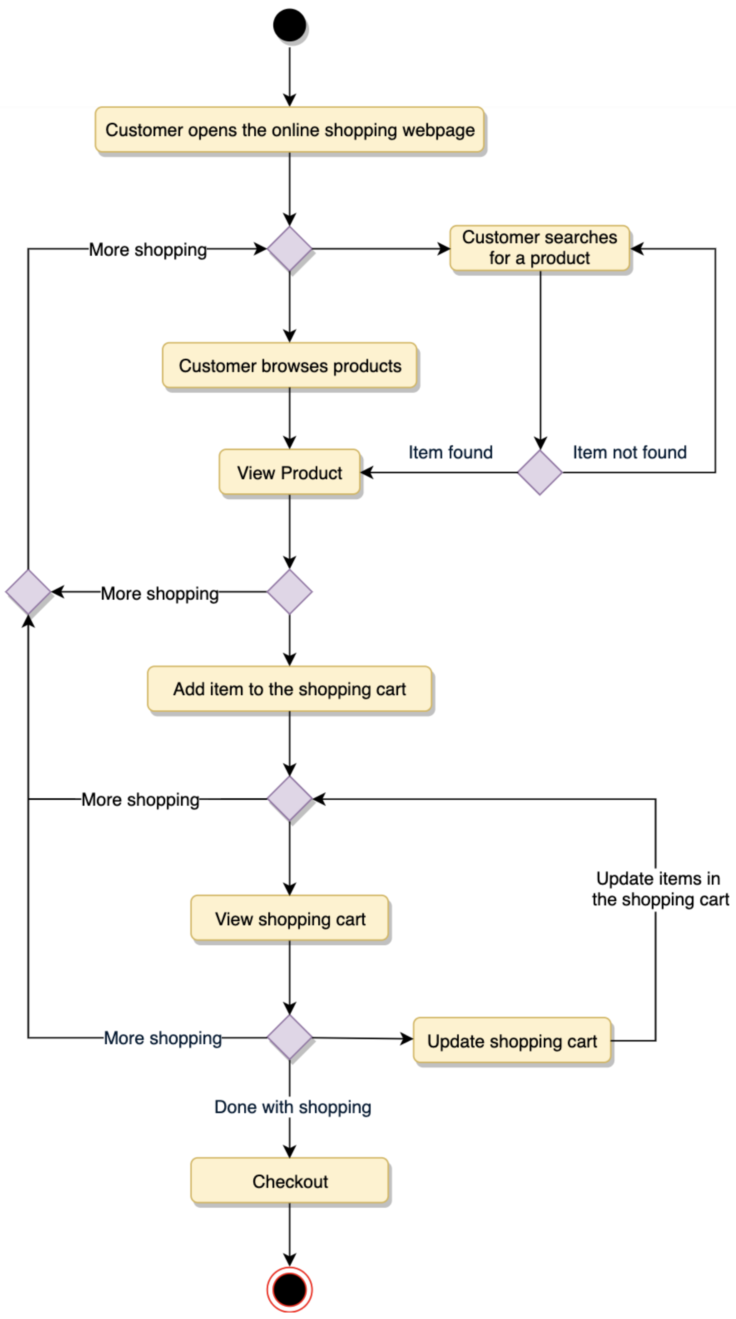
A sequence diagram is straightforward to draw. Across the top of your diagram, identify the class instances (objects) by putting each class instance inside a box (see above figure). If a class instance sends a message to another class instance, draw a line with an open arrowhead pointing to the receiving class instance and place the name of the message above the line. Optionally, for important messages, you can draw a dotted line with an arrowhead pointing back to the originating class instance; label the returned value above the dotted line.

Activity Diagrams

We use Activity Diagrams to illustrate the flow of control in a system. An activity diagram shows the flow of control for a system functionality; it emphasizes the condition of flow and the sequence in which it happens. We can also use an activity diagram to refer to the steps involved in the execution of a use case.

Activity diagrams illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operations.

Following is an activity diagram for a user performing online shopping:


Customer opens the online shopping webpage
More shopping
Customer searches for a product
Customer browses products
Item found
Item not found
View Product
More shopping
Add item to the shopping cart
More shopping
Update items in the shopping cart
View shopping cart
More shopping
Update shopping cart
Done with shopping
Checkout

What is the difference between Activity diagram and Sequence diagram?

Activity diagram captures the process flow. It is used for functional modeling. A functional model represents the flow of values from external inputs, through operations and internal data stores, to external outputs.

Sequence diagram tracks the interaction between the objects. It is used for dynamic modeling, which is represented by tracking states, transitions between states, and the events that trigger these transitions.

Class Diagram

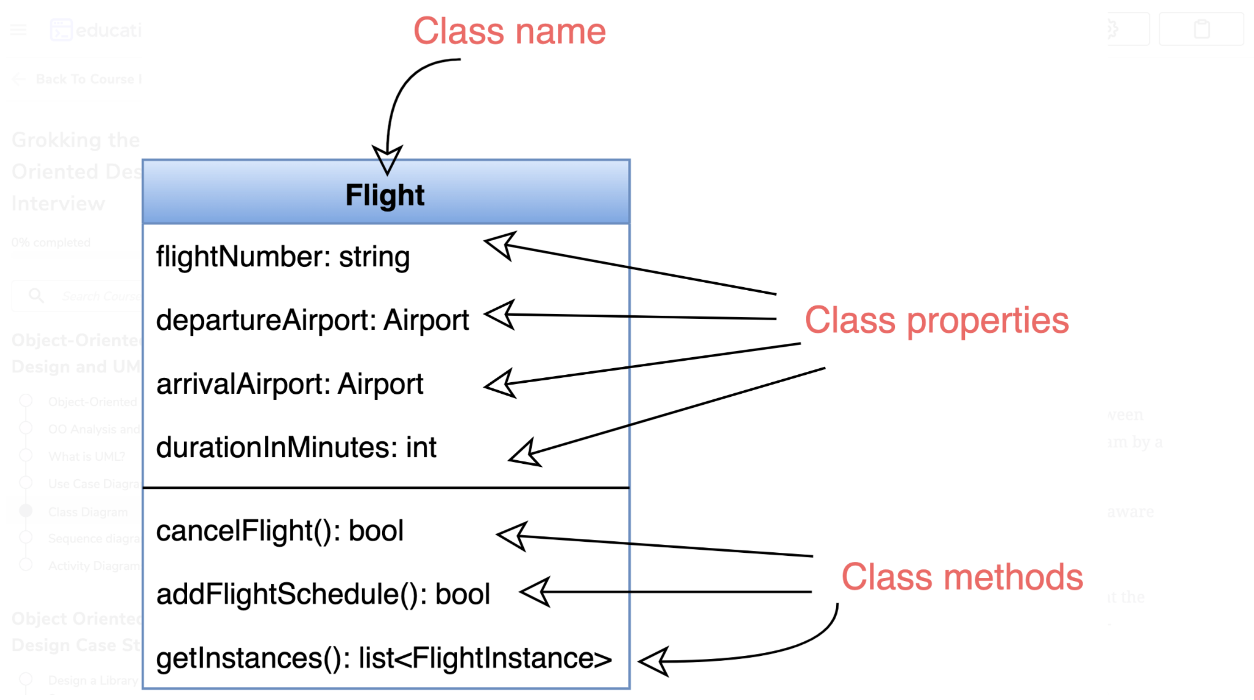
Class diagram is the backbone of object-oriented modeling - it shows how different entities (people, things, and data) relate to each other. In other words, it shows the static structures of the system.

A class diagram describes the attributes and operations of a class and also the constraints imposed on the system. Class diagrams are widely used in the modeling of object-oriented systems because they are the only UML diagrams that can be mapped directly to object-oriented languages.

The purpose of the class diagram can be summarized as:

1. Analysis and design of the static view of an application;
2. To describe the responsibilities of a system;
3. To provide a base for component and deployment diagrams; and,
4. Forward and reverse engineering.

A class is depicted in the class diagram as a rectangle with three horizontal sections, as shown in the figure below. The upper section shows the class's name (Flight), the middle section contains the properties of the class, and the lower section contains the class's operations (or "methods").



These are the different types of relationships between classes:

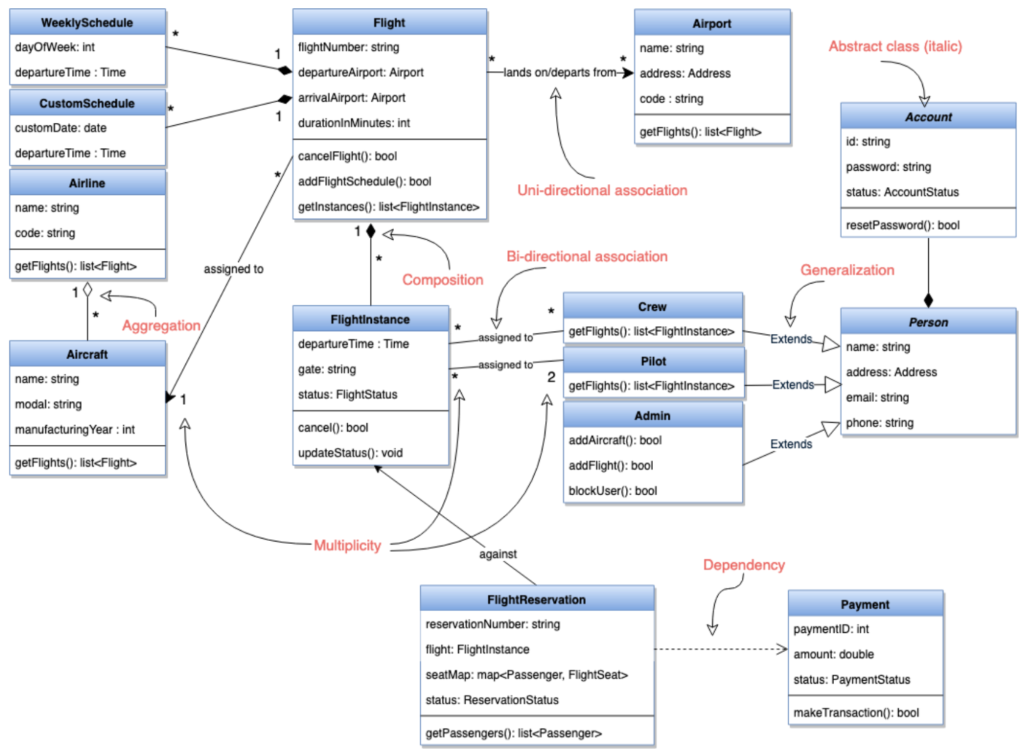
Association: If two classes in a model need to communicate with each other, there must be a link between them. This link can be represented by an association. Associations can be represented in a class diagram by a line between these classes with an arrow indicating the navigation direction.

- By default, associations are always assumed to be bi-directional; this means that both classes are aware of each other and their relationship. In the diagram below, the association between Pilot and FlightInstance is bi-directional, as both classes know each other.
- By contrast, in a uni-directional association, two classes are related - but only one class knows that the relationship exists. In the below example, only Flight class knows about Aircraft; hence it is a uni-directional association

Multiplicity Multiplicity indicates how many instances of a class participate in the relationship. It is a constraint that specifies the range of permitted cardinalities between two classes. For example, in the diagram below, one FlightInstance will have two Pilots, while a Pilot can have many

FlightInstances. A ranged multiplicity can be expressed as “0…*” which means “zero to many" or as “2…4” which means “two to four”.

We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association. The below diagram, demonstrates that a FlightInstance has exactly two Pilots but a Pilot can have many FlightInstances.



Sample class diagram for flight reservation system

Aggregation: Aggregation is a special type of association used to model a “whole to its parts” relationship. In a basic aggregation relationship, the lifecycle of a PART class is independent of the WHOLE class’s lifecycle. In other words, aggregation implies a relationship where the child can exist independently of the parent. In the above diagram, Aircraft can exist without Airline.

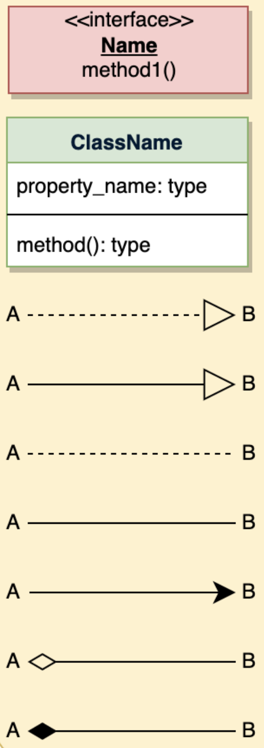
Composition: The composition aggregation relationship is just another form of the aggregation relationship, but the child class's instance lifecycle is dependent on the parent class's instance lifecycle. In other words, Composition implies a relationship where the child cannot exist independent of the parent. In the above example, WeeklySchedule is composed in Flight which means when Flight lifecycle ends, WeeklySchedule automatically gets destroyed.

Generalization: Generalization is the mechanism for combining similar classes of objects into a single, more general class. Generalization identifies commonalities among a set of entities. In the above diagram, Crew, Pilot, and Admin, all are Person.

Dependency: A dependency relationship is a relationship in which one class, the client, uses or depends on another class, the supplier. In the above diagram, FlightReservation depends on Payment.

Abstract class: An abstract class is identified by specifying its name in *italics*. In the above diagram, both Person and Account classes are abstract classes.

UML conventions



Interface: Classes implement interfaces, denoted by Generalization.

Class: Every class can have properties and methods.
Abstract classes are identified by their *Italic* names.

Generalization: A implements B.

Inheritance: A inherits from B. A "is-a" B.

Use Interface: A uses interface B.

Association: A and B call each other.

Uni-directional Association: A can call B, but not vice versa.

Aggregation: A "has-an" instance of B. B can exist without A.

Composition: A "has-an" instance of B. B cannot exist without A.