Class Diagram | Unified Modeling Language (UML)

Last Updated : 03 Jan, 2025

A UML class diagram is a visual tool that represents the structure of a system by showing its classes, attributes, methods, and the relationships between them. It helps everyone involved in a project—like developers and designers—understand how the system is organized and how its components interact.

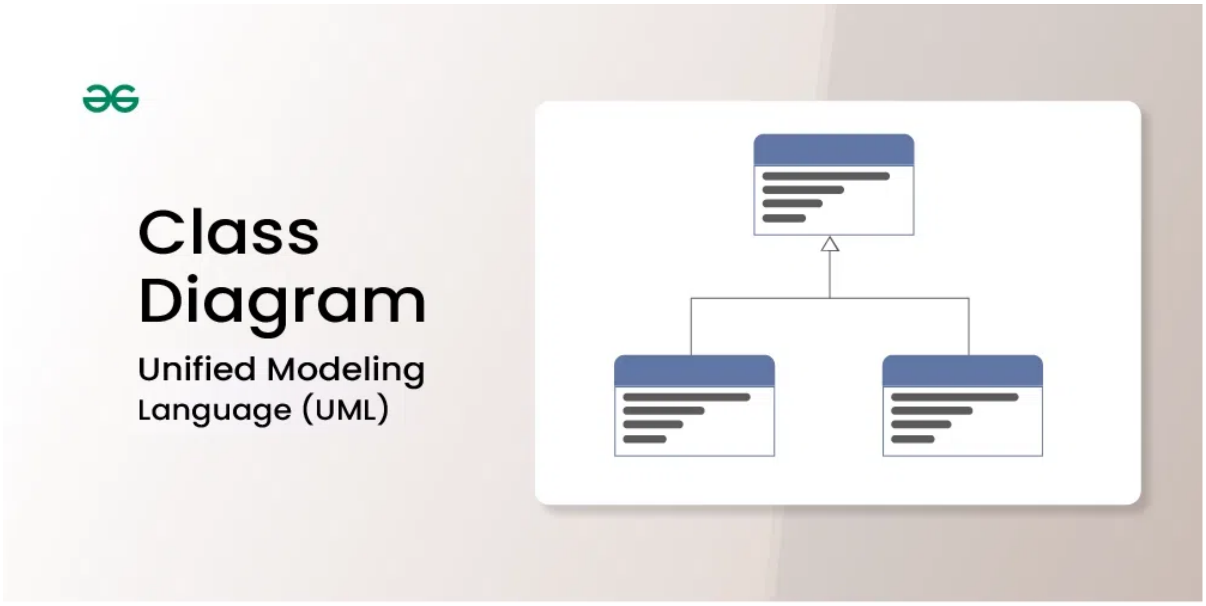


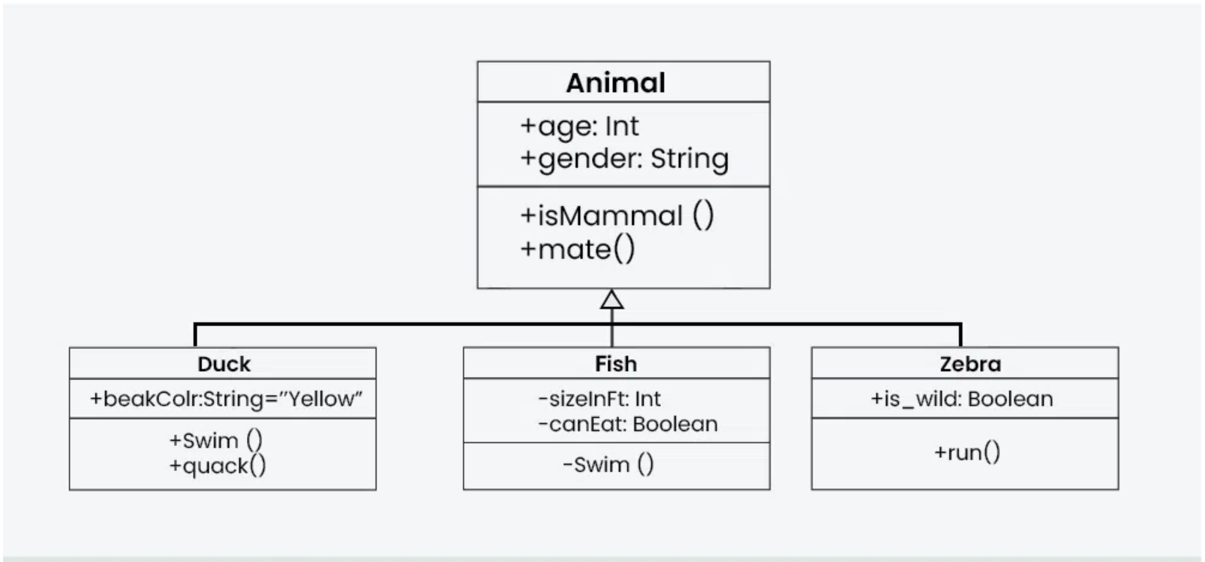
Table of Content

What are class Diagrams?

Class diagrams are a type of UML (Unified Modeling Language) diagram used in software engineering to visually represent the structure and relationships of classes within a system i.e. used to construct and visualize object-oriented systems.

In these diagrams, classes are depicted as boxes, each containing three compartments for the class name, attributes, and methods. Lines connecting classes illustrate associations, showing relationships such as one-to-one or one-to-many.

Class diagrams provide a high-level overview of a system's design, helping to communicate and document the structure of the software. They are a fundamental tool in object-oriented design and play a crucial role in the software development lifecycle.



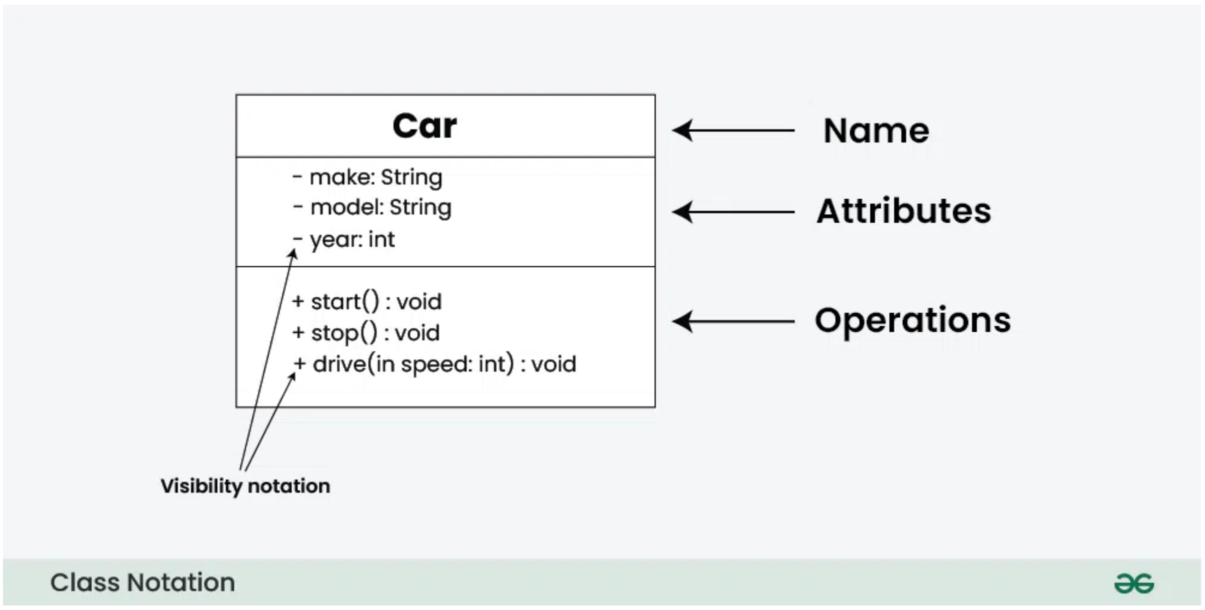
Class Diagram example

What is a class?

In object-oriented programming (OOP), a class is a blueprint or template for creating objects. Objects are instances of classes, and each class defines a set of attributes (data members) and methods (functions or procedures) that the objects created from that class will possess. The attributes represent the characteristics or properties of the object, while the methods define the behaviors or actions that the object can perform.

UML Class Notation

class notation is a graphical representation used to depict classes and their relationships in object-oriented modeling.



1. **Class Name:**
   - The name of the class is typically written in the top compartment of the class box and is centered and bold.

2. **Attributes:**
   - Attributes, also known as properties or fields, represent the data members of the class. They are listed in the second compartment of the class box and often include the visibility (e.g., public, private) and the data type of each attribute.

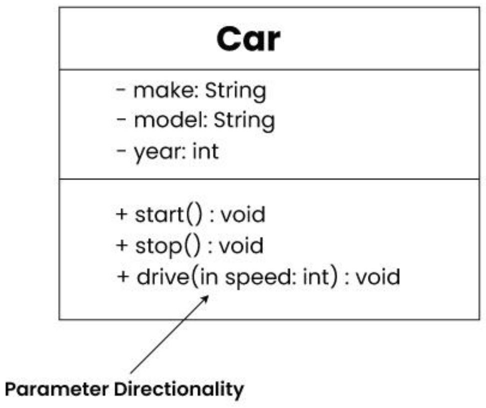
3. **Methods:**
   - Methods, also known as functions or operations, represent the behavior or functionality of the class. They are listed in the third compartment of the class box and include the visibility (e.g., public, private), return type, and parameters of each method.

4. **Visibility Notation:**
   - Visibility notations indicate the access level of attributes and methods. Common visibility notations include:
     - + for public (visible to all classes)
     - - for private (visible only within the class)
     - # for protected (visible to subclasses)

    - ~ for package or default visibility (visible to classes in the same package)

Parameter Directionality

In class diagrams, parameter directionality refers to the indication of the flow of information between classes through method parameters. It helps to specify whether a parameter is an input, an output, or both. This information is crucial for understanding how data is passed between objects during method calls.



class notation with parameter directionality

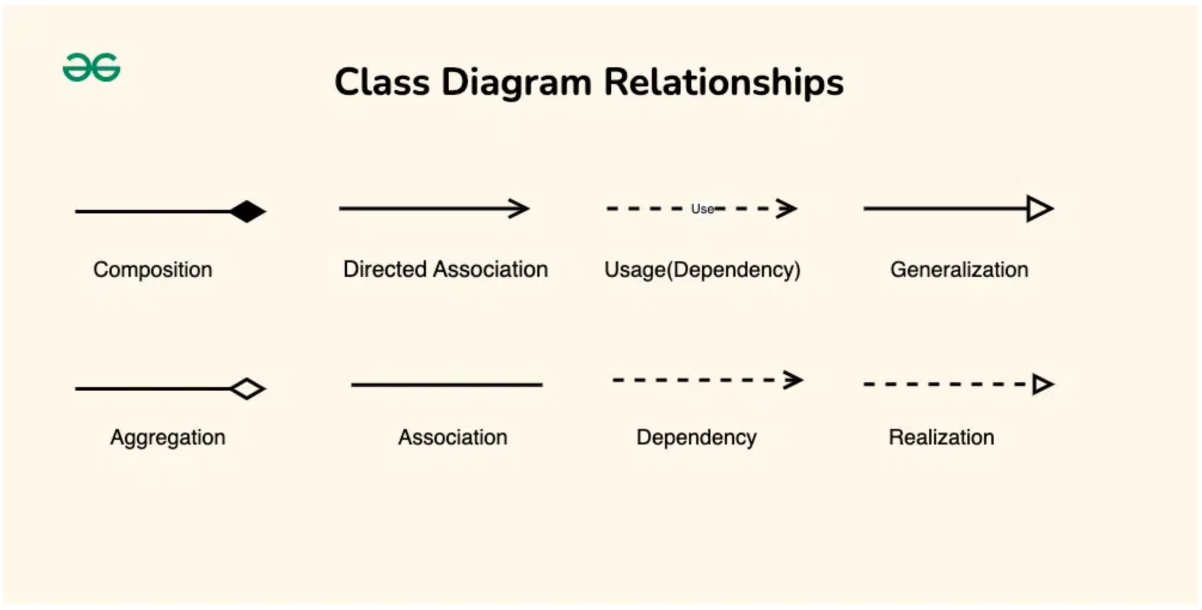
There are three main parameter directionality notations used in class diagrams:

- **In (Input):**
    - An input parameter is a parameter passed from the calling object (client) to the called object (server) during a method invocation.
    - It is represented by an arrow pointing towards the receiving class (the class that owns the method).
- **Out (Output):**
    - An output parameter is a parameter passed from the called object (server) back to the calling object (client) after the method execution.
    - It is represented by an arrow pointing away from the receiving class.
- **InOut (Input and Output):**
    - An InOut parameter serves as both input and output. It carries information from the calling object to the called object and vice versa.

- It is represented by an arrow pointing towards and away from the receiving class.

Relationships between classes

In class diagrams, relationships between classes describe how classes are connected or interact with each other within a system. There are several types of relationships in object-oriented modeling, each serving a specific purpose. Here are some common types of relationships in class diagrams:



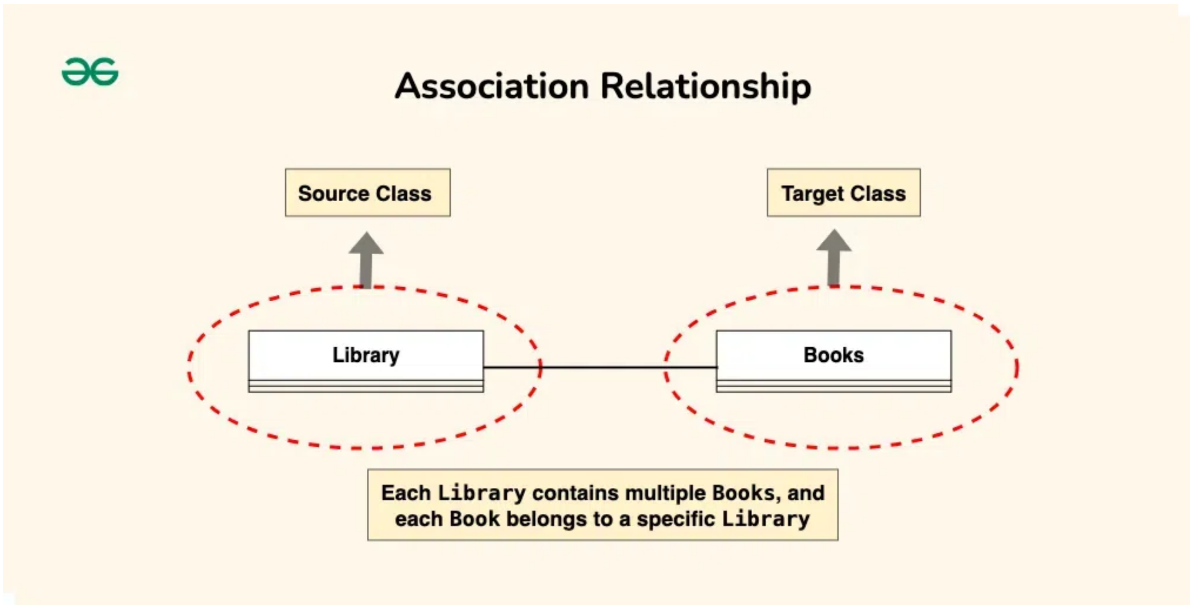
1. Association

An association represents a bi-directional relationship between two classes. It indicates that instances of one class are connected to instances of another class. Associations are typically depicted as a solid line connecting the classes, with optional arrows indicating the direction of the relationship.

Let’s understand association using an example:

> *Let’s consider a simple system for managing a library. In this system, we have two main entities: `Book` and `Library`. Each `Library` contains multiple `Books`, and each `Book` belongs to a specific `Library`. This relationship between `Library` and `Book` represents an association.*

The “Library” class can be considered the source class because it contains a reference to multiple instances of the “Book” class. The “Book” class would be considered the target class because it belongs to a specific library.


Association Relationship
Source Class
Target Class
Library
Books
Each Library contains multiple Books, and each Book belongs to a specific Library

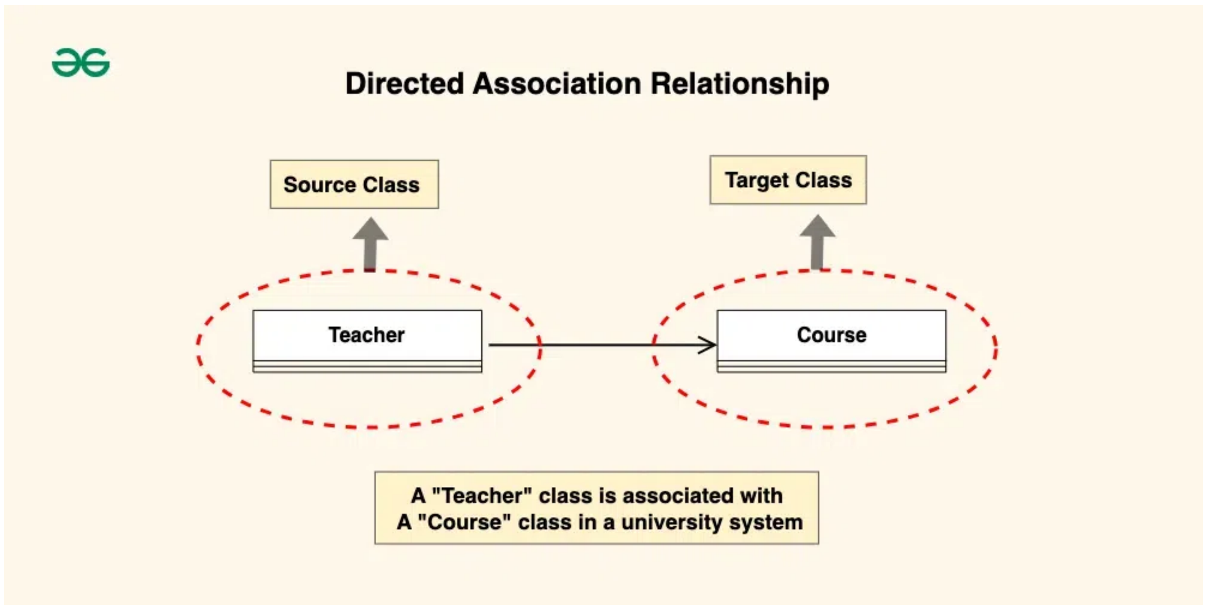
2. Directed Association

A directed association in a UML class diagram represents a relationship between two classes where the association has a direction, indicating that one class is associated with another in a specific way.

- In a directed association, an arrowhead is added to the association line to indicate the direction of the relationship. The arrow points from the class that initiates the association to the class that is being targeted or affected by the association.
- Directed associations are used when the association has a specific flow or directionality, such as indicating which class is responsible for initiating the association or which class has a dependency on another.

Consider a scenario where a “Teacher” class is associated with a “Course” class in a university system. The directed association arrow may point from the “Teacher” class to the “Course” class, indicating that a teacher is associated with or teaches a specific course.

- The source class is the “Teacher” class. The “Teacher” class initiates the association by teaching a specific course.
- The target class is the “Course” class. The “Course” class is affected by the association as it is being taught by a specific teacher.

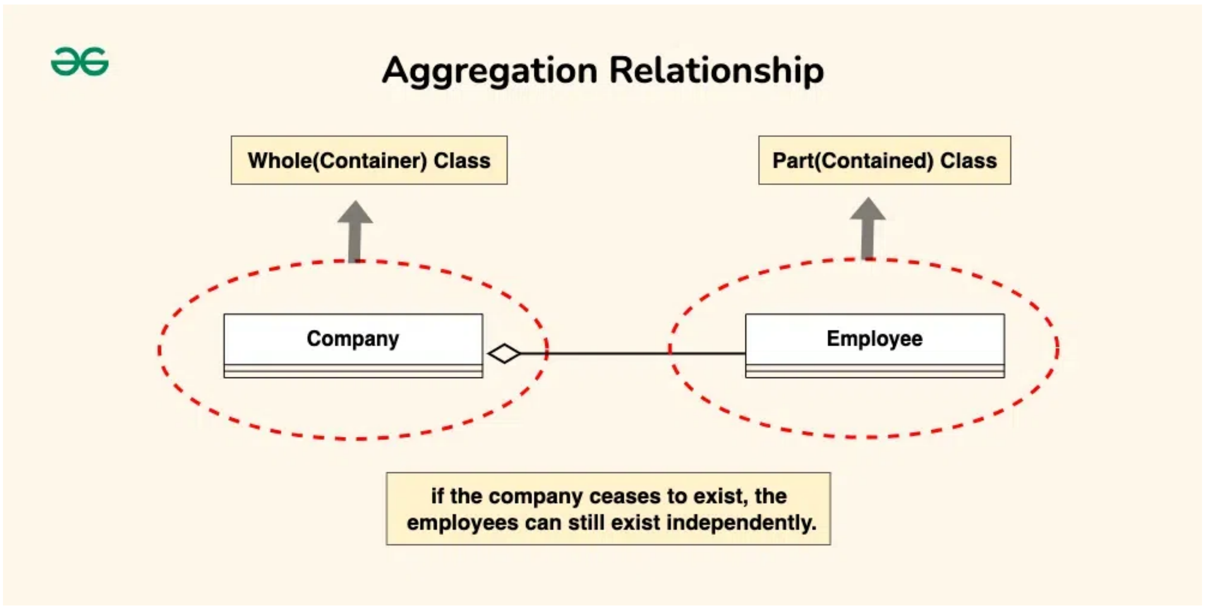

Directed Association Relationship
Source Class
Target Class
Teacher
Course
A "Teacher" class is associated with
A "Course" class in a university system

3. Aggregation

Aggregation is a specialized form of association that represents a “whole-part” relationship. It denotes a stronger relationship where one class (the whole) contains or is composed of another class (the part). Aggregation is represented by a diamond shape on the side of the whole class. In this kind of relationship, the child class can exist independently of its parent class.

Let’s understand aggregation using an example:

> *The company can be considered as the whole, while the employees are the parts. Employees belong to the company, and the company can have multiple employees. However, if the company ceases to exist, the employees can still exist independently.*



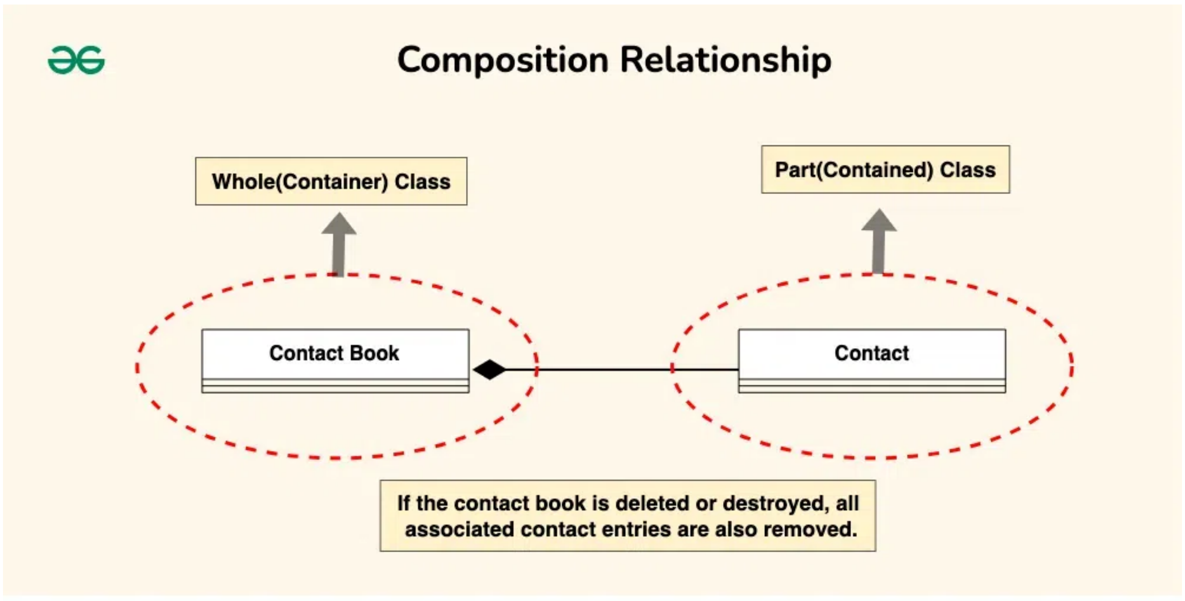
4. Composition

Composition is a stronger form of aggregation, indicating a more significant ownership or dependency relationship. In composition, the part class cannot exist independently of the whole class. Composition is represented by a filled diamond shape on the side of the whole class.

Let’s understand Composition using an example:

Imagine a digital contact book application. The contact book is the whole, and each contact entry is a part. Each contact entry is fully owned and managed by the contact book. If the contact book is deleted or destroyed, all associated contact entries are also removed.

This illustrates composition because the existence of the contact entries depends entirely on the presence of the contact book. Without the contact book, the individual contact entries lose their meaning and cannot exist on their own.

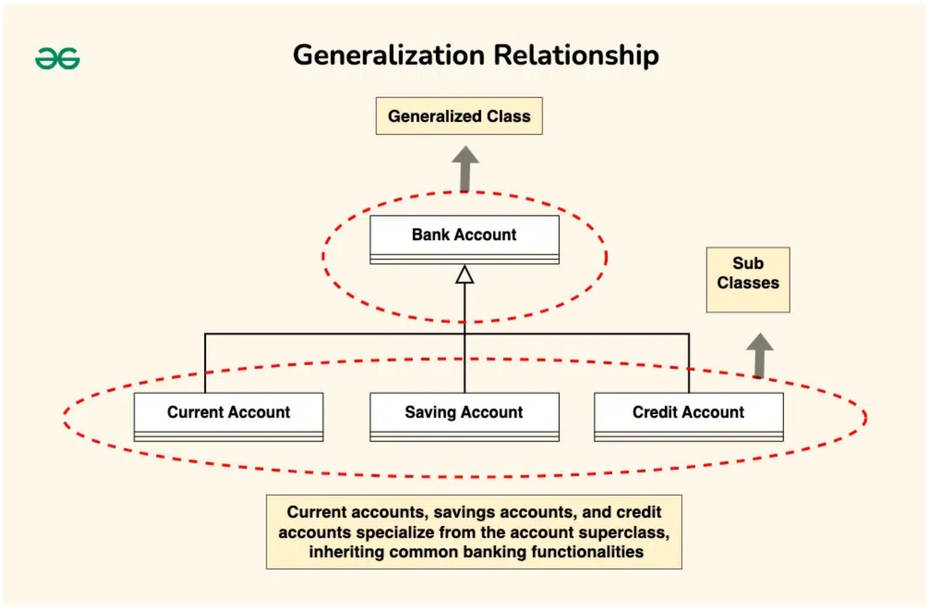

Composition Relationship
Whole(Container) Class
Part(Contained) Class
Contact Book
Contact
If the contact book is deleted or destroyed, all associated contact entries are also removed.

5. Generalization(Inheritance)

Inheritance represents an “is-a” relationship between classes, where one class (the subclass or child) inherits the properties and behaviors of another class (the superclass or parent). Inheritance is depicted by a solid line with a closed, hollow arrowhead pointing from the subclass to the superclass.

In the example of bank accounts, we can use generalization to represent different types of accounts such as current accounts, savings accounts, and credit accounts.

The Bank Account class serves as the generalized representation of all types of bank accounts, while the subclasses (Current Account, Savings Account, Credit Account) represent specialized versions that inherit and extend the functionality of the base class.


Generalization Relationship
Generalized Class
Bank Account
Sub Classes
Current Account
Saving Account
Credit Account
Current accounts, savings accounts, and credit accounts specialize from the account superclass, inheriting common banking functionalities

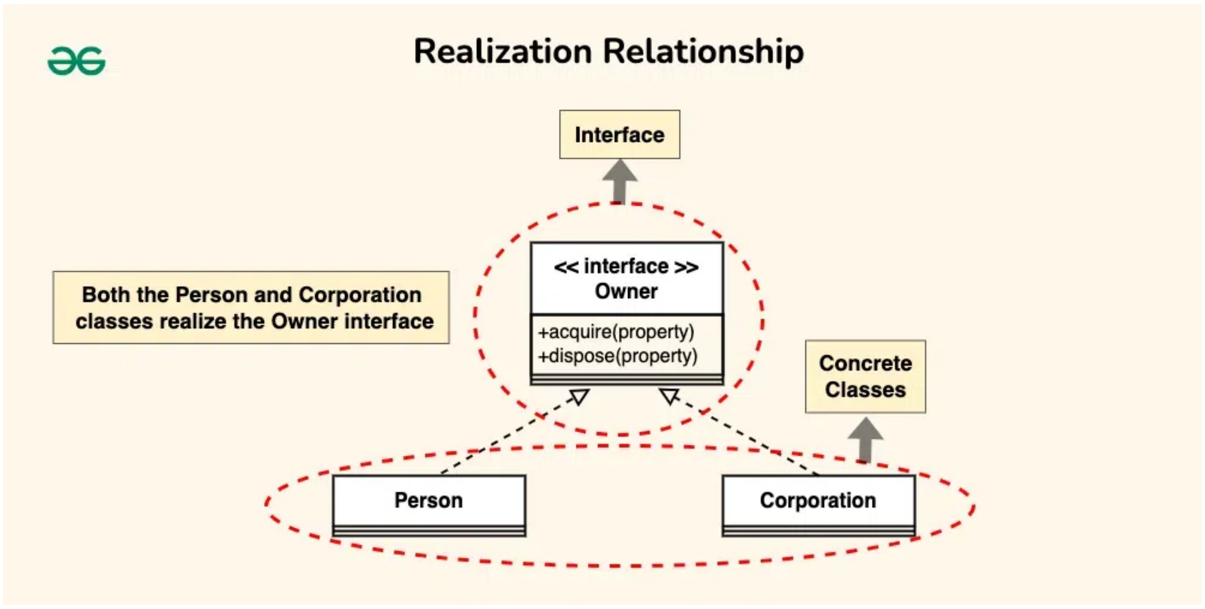
6. Realization (Interface Implementation)

Realization indicates that a class implements the features of an interface. It is often used in cases where a class realizes the operations defined by an interface. Realization is depicted by a dashed line with an open arrowhead pointing from the implementing class to the interface.

Let's consider the scenario where a "Person" and a "Corporation" both realizing an "Owner" interface.

- **Owner Interface:** This interface now includes methods such as "acquire(property)" and "dispose(property)" to represent actions related to acquiring and disposing of property.
- **Person Class (Realization):** The Person class implements the Owner interface, providing concrete implementations for the "acquire(property)" and "dispose(property)" methods. For instance, a person can acquire ownership of a house or dispose of a car.
- **Corporation Class (Realization):** Similarly, the Corporation class also implements the Owner interface, offering specific implementations for the "acquire(property)" and "dispose(property)" methods. For example, a corporation can acquire ownership of real estate properties or dispose of company vehicles.

Both the Person and Corporation classes realize the Owner interface, meaning they provide concrete implementations for the "acquire(property)" and "dispose(property)" methods defined in the interface.



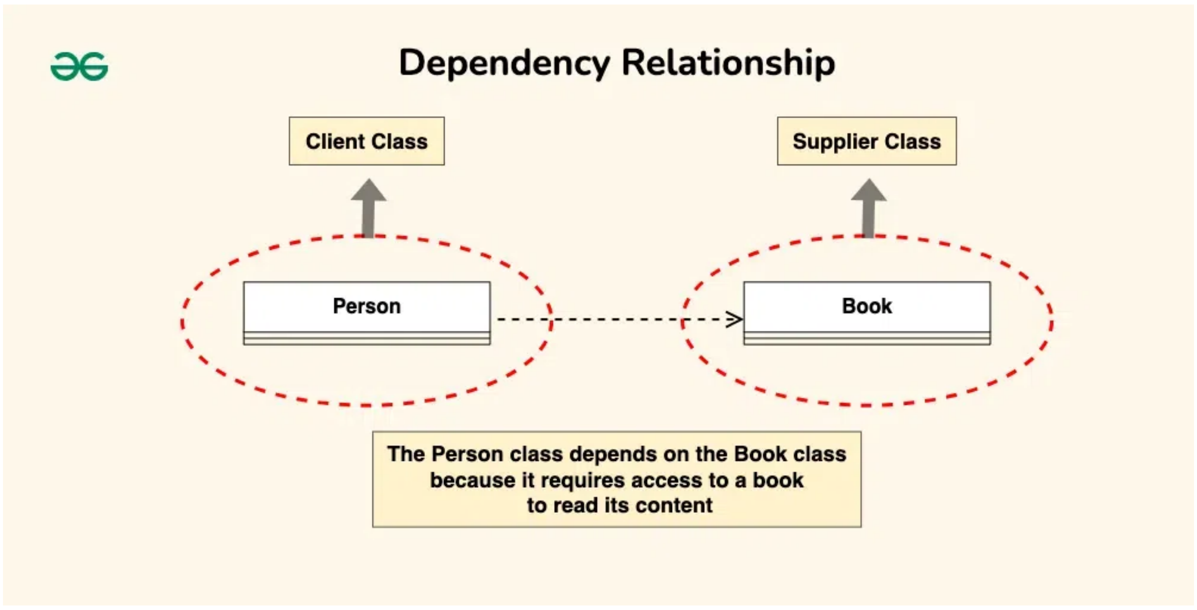
7. Dependency Relationship

A dependency exists between two classes when one class relies on another, but the relationship is not as strong as association or inheritance. It represents a more loosely coupled connection between classes. Dependencies are often depicted as a dashed arrow.

Let's consider a scenario where a Person depends on a Book.

- **Person Class:** Represents an individual who reads a book. The Person class depends on the Book class to access and read the content.
- **Book Class:** Represents a book that contains content to be read by a person. The Book class is independent and can exist without the Person class.

The Person class depends on the Book class because it requires access to a book to read its content. However, the Book class does not depend on the Person class; it can exist independently and does not rely on the Person class for its functionality.


Dependency Relationship
Client Class
Supplier Class
Person
Book
The Person class depends on the Book class because it requires access to a book to read its content

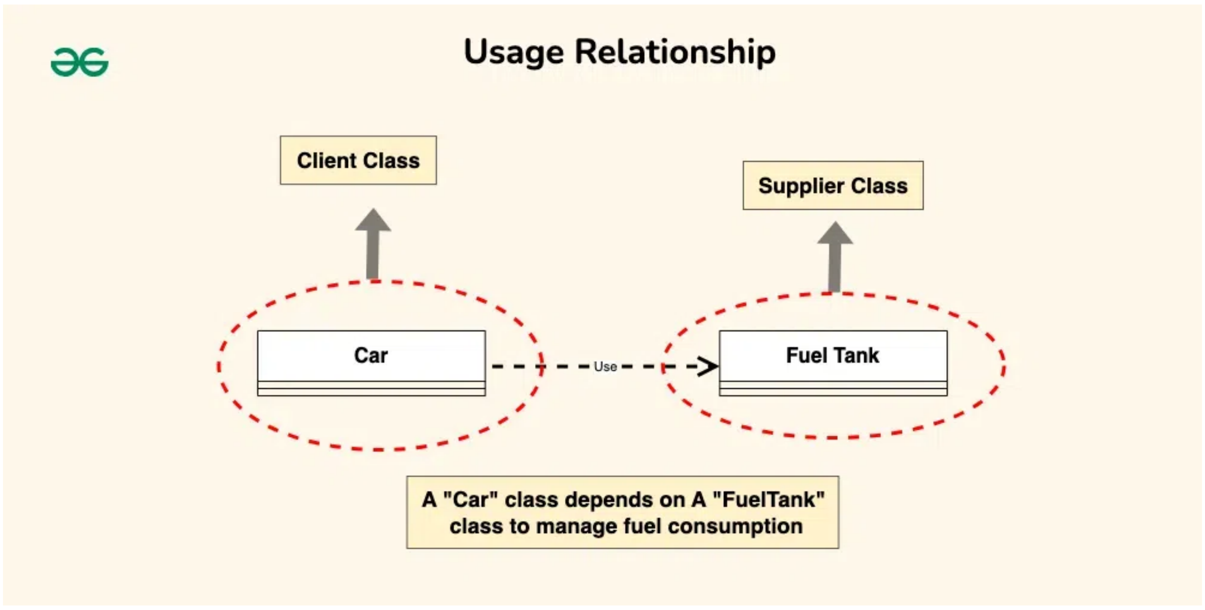
8. Usage(Dependency) Relationship

A usage dependency relationship in a UML class diagram indicates that one class (the client) utilizes or depends on another class (the supplier) to perform certain tasks or access certain functionality. The client class relies on the services provided by the supplier class but does not own or create instances of it.

- In UML class diagrams, usage dependencies are typically represented by a dashed arrowed line pointing from the client class to the supplier class.
- The arrow indicates the direction of the dependency, showing that the client class depends on the services provided by the supplier class.

Consider a scenario where a “Car” class depends on a “FuelTank” class to manage fuel consumption.

- The “Car” class may need to access methods or attributes of the “FuelTank” class to check the fuel level, refill fuel, or monitor fuel consumption.
- In this case, the “Car” class has a usage dependency on the “FuelTank” class because it utilizes its services to perform certain tasks related to fuel management.


Usage Relationship
Client Class
Supplier Class
Car
Use
Fuel Tank
A "Car" class depends on A "FuelTank" class to manage fuel consumption

Purpose of Class Diagrams

The main purpose of using class diagrams is:

- This is the only UML that can appropriately depict various aspects of the OOPs concept.
- Proper design and analysis of applications can be faster and efficient.
- It is the base for deployment and component diagram.
- It incorporates forward and reverse engineering.

Benefits of Class Diagrams

Below are the benefits of class diagrams:

- Class diagrams represent the system's classes, attributes, methods, and relationships, providing a clear view of its architecture.
- They shows various relationships between classes, such as associations and inheritance, helping stakeholders understand component connectivity.
- Class diagrams serve as a visual tool for communication among team members and stakeholders, bridging gaps between technical and non-technical audiences.
- They guide developers in coding by illustrating the design, ensuring consistency between the design and actual implementation.
- Many development tools allow for code generation from class diagrams, reducing manual errors and saving time.

How to draw Class Diagrams

Drawing class diagrams involves visualizing the structure of a system, including classes, their attributes, methods, and relationships. Here are the steps to draw class diagrams:

- **Step 1: Identify Classes:**
    - Start by identifying the classes in your system. A class represents a blueprint for objects and should encapsulate related attributes and methods.
- **Step 2: List Attributes and Methods:**
    - For each class, list its attributes (properties, fields) and methods (functions, operations). Include information such as data types and visibility (public, private, protected).
- **Step 3: Identify Relationships:**
    - Determine the relationships between classes. Common relationships include associations, aggregations, compositions, inheritance, and dependencies. Understand the nature and multiplicity of these relationships.
- **Step 4: Create Class Boxes:**
    - Draw a rectangle (class box) for each class identified. Place the class name in the top compartment of the box. Divide the box into compartments for attributes and methods.
- **Step 5: Add Attributes and Methods:**
    - Inside each class box, list the attributes and methods in their respective compartments. Use visibility notations (+ for public, – for private, # for protected, ~ for package/default).
- **Step 6: Draw Relationships:**
    - Draw lines to represent relationships between classes. Use arrows to indicate the direction of associations or dependencies. Different line types or notations may be used for various relationships.
- **Step 7: Label Relationships:**

    - Label the relationships with multiplicity and role names if needed. Multiplicity indicates the number of instances involved in the relationship, and role names clarify the role of each class in the relationship.
- **Step 8: Review and Refine:**
    - Review your class diagram to ensure it accurately represents the system's structure and relationships. Refine the diagram as needed based on feedback and requirements.

Use cases of Class Diagrams

Below are the use cases of class diagrams:

- Class diagrams shows the static structure of a software system, showcasing classes, attributes, methods, and relationships.
- They help visualize and organize the components of the system, serving as a blueprint for implementation.
- They encourage discussions about the system's design, promoting a shared understanding among team members.
- Many development tools support code generation from class diagrams, enabling developers to create code skeletons efficiently.
- This capability minimizes manual coding efforts and helps ensure that the implementation aligns with the design.

Comment More info Advertise with us

Next Article

Object Diagrams | Unified Modeling Language (UML)

Similar Reads

What are UML Diagrams

Unified Modeling Language (UML) Diagrams

Unified Modeling Language (UML) is a general-purpose modeling language. The main aim of UML is to define a standard way to visualize the way a system has been designed. It is quite similar to blueprints...

14 min read

UML Full Form

The full form of UML is "Unified Modeling Language". It is a general-purpose modeling language. The main aim of UML is to define a standard way to visualize how a system has been designed. It is quite...

3 min read

Structural Diagrams

Class Diagram | Unified Modeling Language (UML)

A UML class diagram is a visual tool that represents the structure of a system by showing its classes, attributes, methods, and the relationships between them. It helps everyone involved in a projectâ€”like...

12 min read

Object Diagrams | Unified Modeling Language (UML)

Object diagrams are a visual representation in UML (Unified Modeling Language) that illustrates the instances of classes and their relationships within a system at a specific point in time. They display...

8 min read

Deployment Diagram in Unified Modeling Language(UML)

A Deployment Diagram is a type of Structural UML Diagram that shows the physical deployment of software components on hardware nodes. It illustrates the mapping of software components onto the...

9 min read

Package Diagram â€“ Unified Modeling Language (UML)

A package diagram is a type of structural diagram in UML (Unified Modeling Language) that organizes and groups related classes and components into packages. It visually represents the dependencies and...

7 min read

Behavioral Diagrams

Behavioral Diagrams | Unified Modeling Language(UML)

Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them. So UML becomes essential to communicate with non-...

7 min read