Unit - 3

Object orientation and Structural modeling https://www.geeksforgeeks.org/system-design/oops-object-oriented-design/

Class and object
Abstraction and encapsulation Method and messages
Interface, Inheritance and polymorphism

Structural Modeling :

Class Diagram and Object diagram Associations and links
Aggregation , Composition and containment Inherpitance, Sub Types and IS-A hierarchy

Object – Oriented Design:

In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities).

The state is distributed among the objects, and each object handles its state data.

For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data.

The tasks defined for one purpose cannot refer or change data of other objects. Objects have their internal data which represent their state.

Similar objects create a class. In other words, each object is a member of some class. Classes may inherit features from the superclass.

The different terms related to object design are:

1. Objects: All entities involved in the solution design are known as objects.

For example, person, banks, company, and users are considered as objects.

Every entity has some attributes associated with it and has some methods to perform on the attributes.

2. Classes: A class is a generalized description of an object. An object is an instance of a class.

3. Messages: Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function.

4. Abstraction: Abstraction is the removal of the irrelevant and the amplification of the essentials.

5. Encapsulation: Encapsulation is also called an information hiding concept.

The data and operations are linked to a single unit.

Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.

6. Inheritance: OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.

This property of OOD is called an inheritance.

7. Polymorphism: OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name.

This is known as polymorphism, which allows a single interface is performing functions for different types.

What is Object-Oriented Design (OOD)?

Object-oriented design (OOD) is a programming technique that solves software problems by building a system of interrelated objects. It makes use of the concepts of classes and objects, encapsulation, inheritance, and polymorphism to model real-world entities and their interactions. A system architecture that is modular, adaptable, and simple to understand and maintain is produced using OOD.

Importance of Object-Oriented Design (OOD) in System Design

Object-Oriented Design (OOD) is important in system design due to several key reasons:

- **Modularity**: OOD simplifies development and maintenance by decomposing complicated structures into smaller, more manageable components.
- **Reusability**: Objects and classes can be reused across different projects, reducing redundancy and saving time.
- **Scalability**: OOD facilitates system growth by making it simple to incorporate new objects without interfering with already-existing functionality.
- **Maintainability**: Encapsulation of data and behavior within objects simplifies troubleshooting and updates, enhancing system reliability.
- **Clear Mapping to Real-World Problems**: By modeling software after real-world entities and their interactions, OOD makes systems more intuitive and easier to understand.
- **Flexibility and Extensibility**: Through inheritance and polymorphism, OOD allows for extending and adapting systems with minimal changes, accommodating future requirements efficiently.

Key Principles of OOD

A number of fundamental principles support object-oriented design (OOD), helping in the development of reliable, expandable, and maintainable systems:

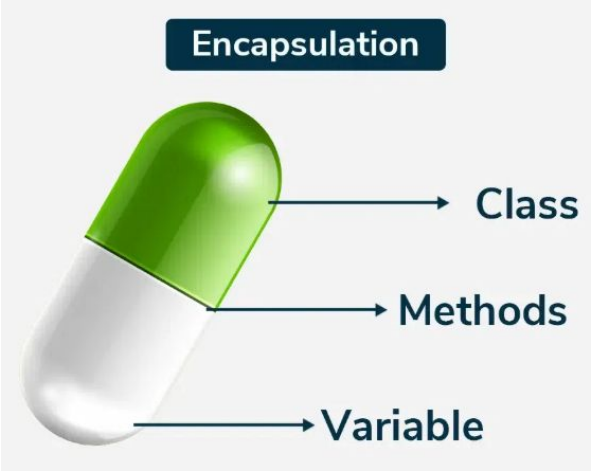
1. **Encapsulation**: Bundling data with methods that operate on the data, restricting direct access to some components and protecting object integrity.
2. **Abstraction**: Simplifying complex systems by modeling classes appropriate to the problem domain, highlighting essential features while hiding unnecessary details.
3. **Inheritance**: Establishing a hierarchy between classes, allowing derived classes to inherit properties and behaviors from base classes, promoting code reuse and extension.
4. **Polymorphism**: Enabling objects to be treated as instances of their parent class, allowing one interface to be used for a general class of actions, improving flexibility and integration.
5. **Composition**: Building complex objects by combining simpler ones, promoting reuse and flexible designs.
6. **SOLID Principles**:
    - **Single Responsibility Principle (SRP)**: A class should have one, and only one, reason to change.
    - **Open/Closed Principle (OCP)**: Classes should be open for extension but closed for modification.
    - **Liskov Substitution Principle (LSP)**: Subtypes must be substitutable for their base types without altering the correctness of the program.
    - **Interface Segregation Principle (ISP)**: Clients should not be forced to depend on interfaces they do not use.
    - **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules. Both should depend on abstractions.

Object-Oriented Design Concepts

A number of fundamental ideas are included in object-oriented design (OOD), which makes it easier to create software that is reliable, scalable, and maintainable. These are the main ideas, supported by examples and explanations.

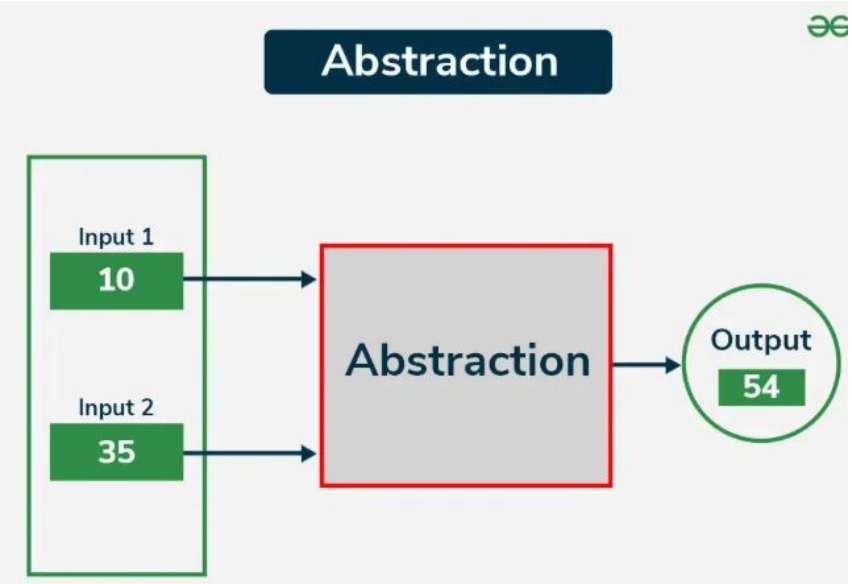
1. Encapsulation

Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit called a class. It restricts direct access to some of the object's components, which is a means of preventing accidental interference and misuse of the data.



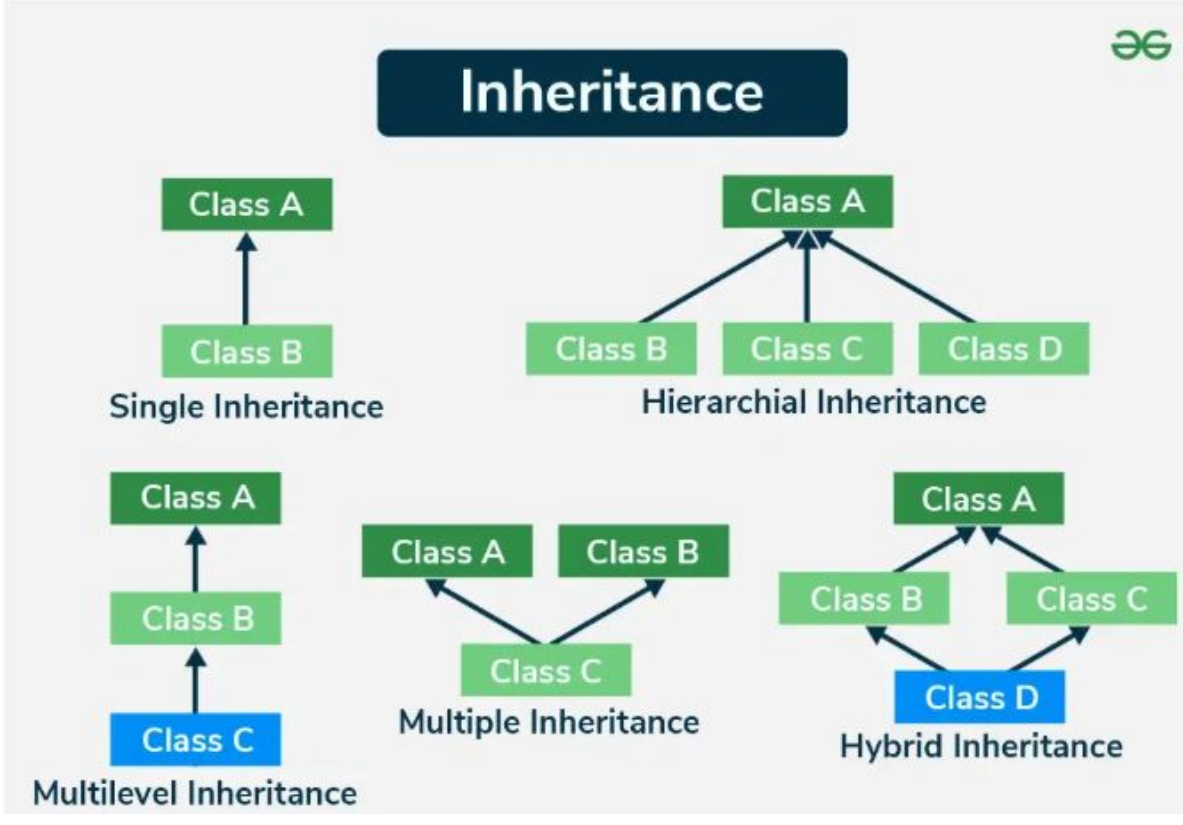
2. Abstraction

Abstraction involves hiding the complex implementation details and showing only the essential features of the object. This helps in managing complexity by allowing the designer to focus on the interactions at a higher level.



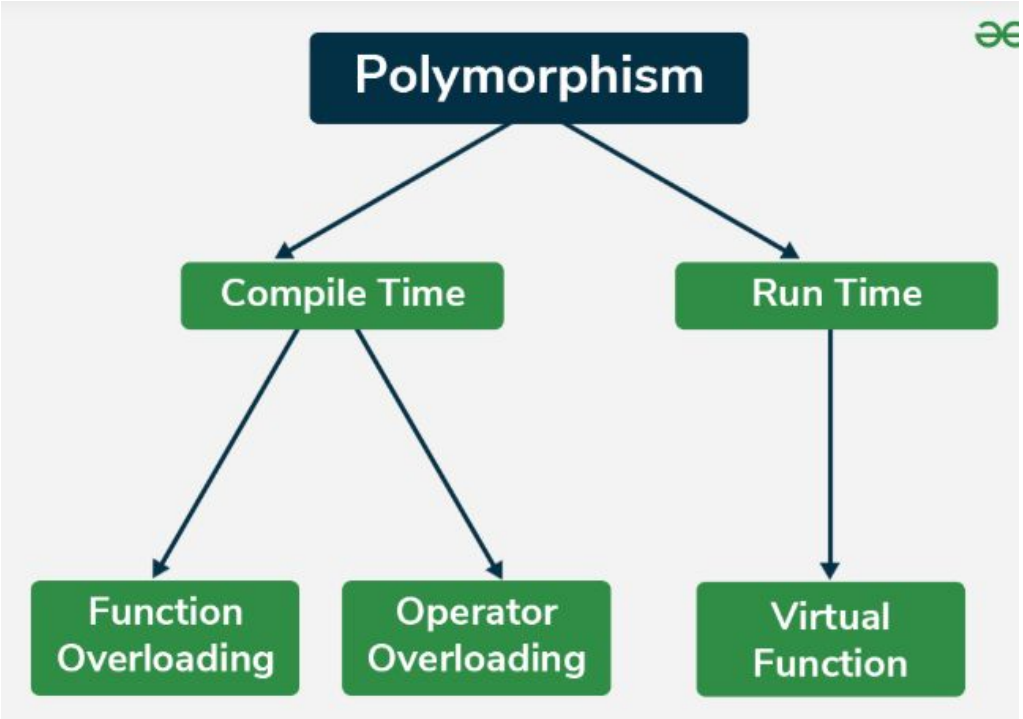
3. Inheritance

Inheritance is a mechanism where a new class inherits properties and behaviors (methods) from an existing class. This promotes code reuse and establishes a natural hierarchy between classes.



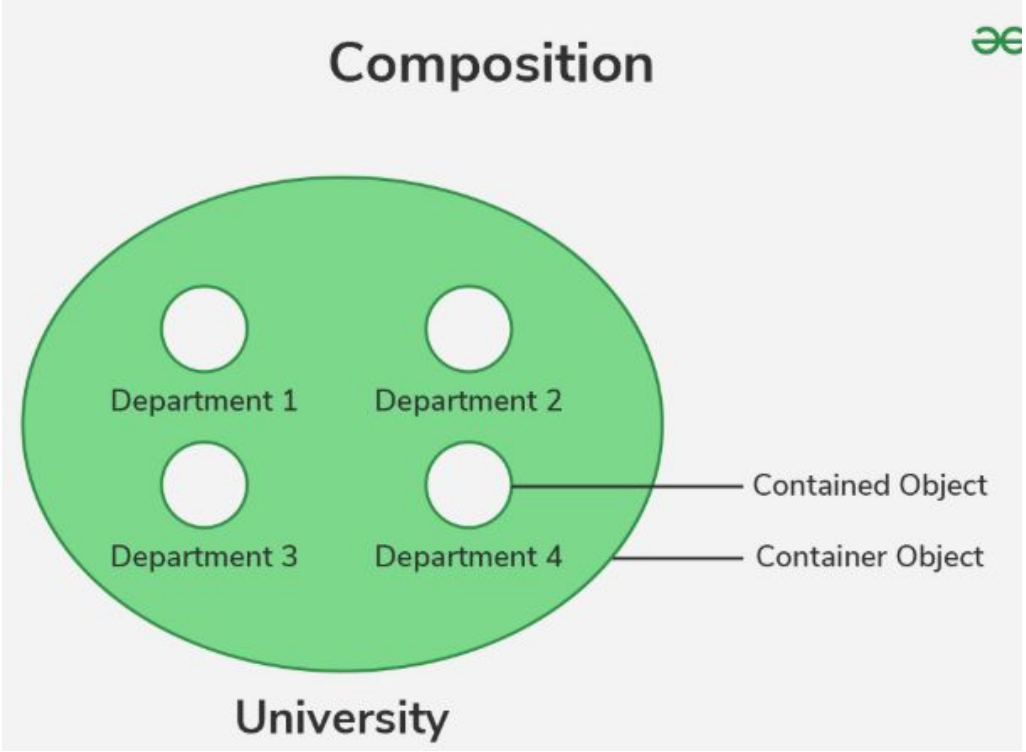
4. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common super class. It enables a single interface to represent different underlying data types and allows methods to use objects of various types.



5. Composition

Composition is a design principle where a class is composed of one or more objects of other classes, rather than inheriting from them. This promotes flexibility and reusability.



Design Patterns in Object-Oriented Design (OOD)

Design patterns in Object-Oriented Design (OOD) are proven solutions to common problems that arise in software design. These patterns provide templates that help to structure code in an efficient and maintainable way. Here are some of the most commonly used design patterns in OOD:

- **Creational Patterns:** Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. Some key creational patterns include:
    - **Singleton**: Ensures a class has only one instance and provides a global point of access to it.
    - **Factory Method**: Defines an interface for creating an object, but lets subclasses alter the type of objects that will be created.
- **Structural Patterns**: Structural patterns deal with object composition or structure, ensuring that if one part changes, the entire structure does not need to do so. Some key structural patterns include:
    - **Adapter**: Converts the interface of a class into another interface the clients expect. It lets classes work together that couldn't otherwise because of incompatible interfaces.
    - **Composite**: Composes objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly.
- **Behavioral Patterns**: Behavioral patterns deal with communication between objects, making it easier and more flexible. Some key behavioral patterns include:
    - **Observer**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
    - **Strategy**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.

UML Diagrams for Visualizing OOD

Diagrams created using the Unified Modeling Language (UML) are useful tools for understanding and clarifying object-oriented system structure. They serve as a blueprint, showing the interactions between many components of a system, which makes difficult concepts simpler to understand and communicate.

Here are some common UML diagrams and how they contribute to object-oriented design:

- **Class Diagrams**: These diagrams show the structure of a system by highlighting the classes, their attributes, and relationships. This is useful in visualizing how data and behaviors are organized, which is the foundation of object-oriented design.
- **Sequence Diagrams**: Sequence diagrams illustrate how objects in a system interact over time. They show the flow of messages and actions, helping designers see the order in which things happen. This makes it easier to understand the dynamic behavior of a system.
- **State Diagrams**: State diagrams represent the various states of an object and the transitions between them. They help designers understand how objects respond to different events, which is key for designing systems that change over time.

By using these diagrams, developers can make sure everyone has a clear picture of the system's structure and behavior, making it easier to collaborate and avoid misunderstandings.

Common Challenges and Anti-Patterns in Object-Oriented Design (OOD)

While it offers strong software development concepts, object-oriented design (OOD) is not without its challenges and drawbacks. For systems to be productive and maintained, it is essential to recognize these difficulties and stay clear of typical anti-patterns. These are a few common OOD challenges and anti-patterns:

Common Challenges in Object-Oriented Design (OOD)

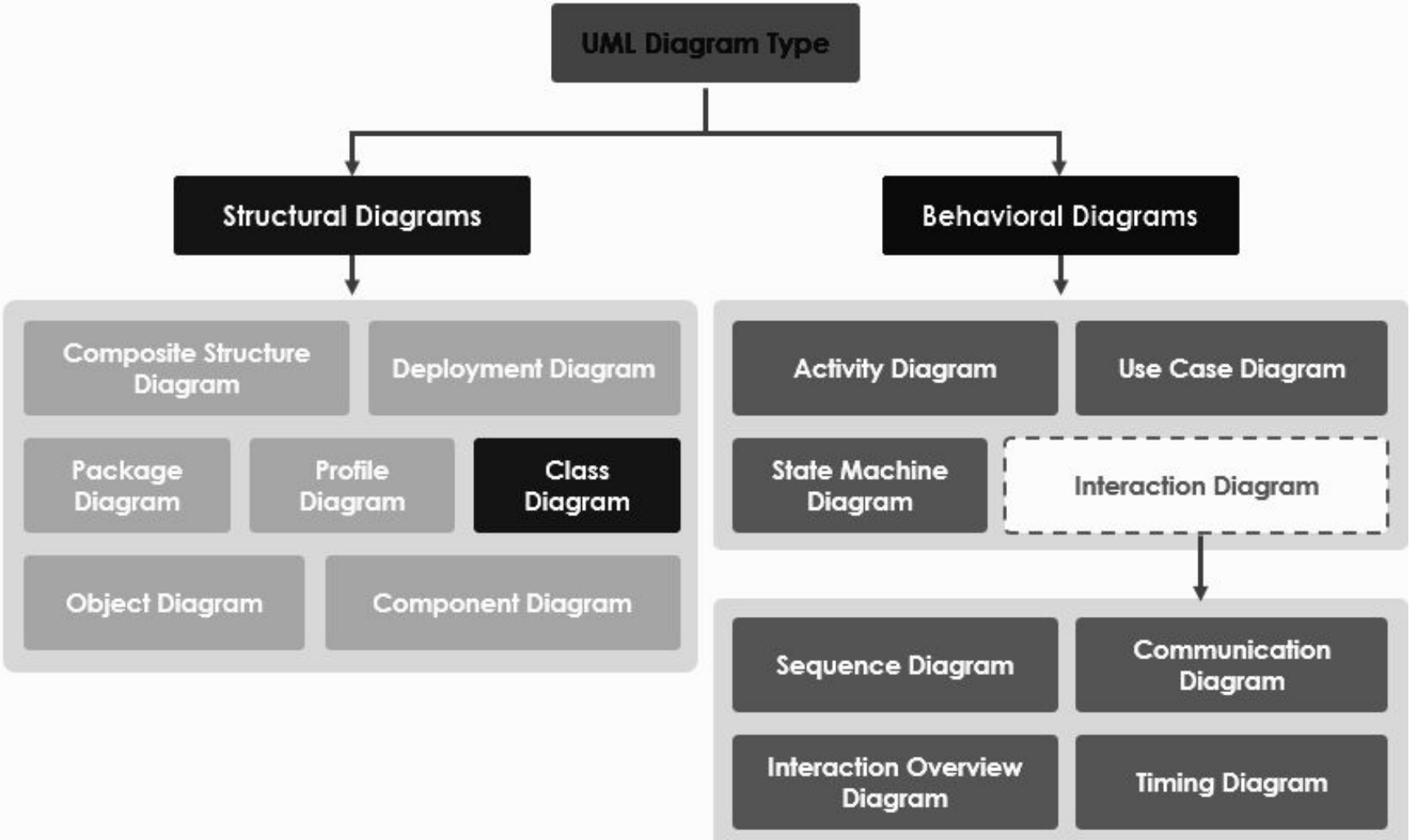
- Designing for potential future needs that may never arise, which adds needless complexity.
    - **Impact**: Leads to code that is harder to understand, maintain, and extend.
    - **Mitigation**: Focus on current requirements and implement extensibility only when there's a clear need.
- Not foreseeing future demands and modifications, which leads to an inflexible system.
    - **Impact**: Makes the system difficult to extend or modify.
    - **Mitigation**: Apply principles like SOLID and design patterns that facilitate flexibility and scalability.
- Ensuring that encapsulation does not excessively degrade performance.
    - **Impact**: Encapsulation can lead to additional layers of abstraction that may impact performance.
    - **Mitigation**: Use encapsulation judiciously and optimize critical performance paths as needed.
- Determining the right level of abstraction to balance simplicity and functionality.
    - **Impact**: Too much abstraction can obscure functionality; too little can lead to code duplication.
    - **Mitigation**: Aim for clear and concise abstractions that accurately represent the problem domain.

https://sparxsystems.com/resources/tutorials/uml2/class-diagram.html

Structural Modeling :

Class Diagram and Object diagram Associations and links

Aggregation , Composition and containment Inherpitance, Sub Types and IS-A hierarchy


UML Diagram Type
Structural Diagrams
Behavioral Diagrams
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Class Diagram
Object Diagram
Component Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram

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.

What are Object Diagrams?

An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them.

- An object diagram in UML is useful because it provides a clear and visual representation of specific instances of classes and their relationships at a particular point in time, aiding in understanding and communicating the structure and interactions within a system.
- In other words, “An object diagram in the Unified Modeling Language (UML), is a diagram that shows a complete or partial view of the structure of a modeled system at a specific time.

Object diagrams in UML are depicted using a simple and intuitive notations to show a snapshot of a system at a specific point in time, displaying instances of classes and their relationships.

Class Diagrams Vs. Object Diagrams

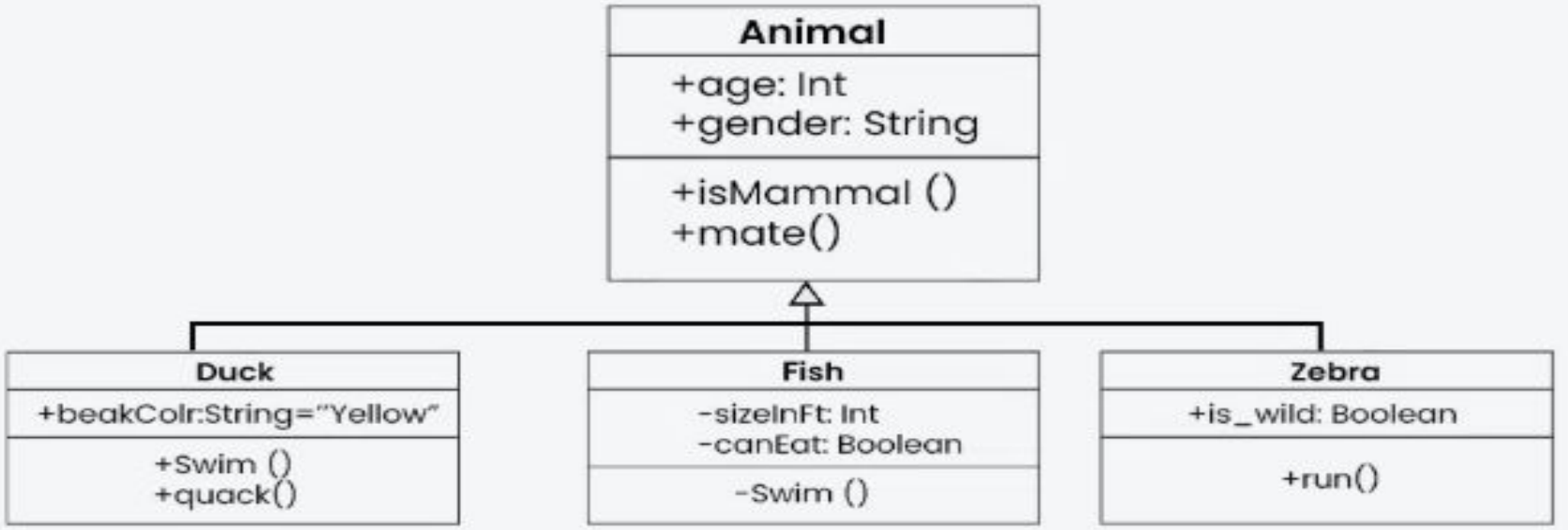
Below are the differences between class diagrams and object diagrams.

Aspect	Class Diagrams	Object Diagrams
Purpose	Represent the static structure of classes, their attributes, methods, and relationships in a system.	Represent a snapshot of objects and their relationships at a specific point in time.
Elements	Classes, attributes, operations, and relationships.	Objects, their attributes, and relationships between objects.
Relationship Type	Represents the static relationships between classes (e.g., inheritance, association, aggregation, composition).	Represents the relationships between specific instances of classes (objects).
Flexibility	Provides a more abstract and flexible view of the system, suitable for design and architecture.	Provides a more concrete and specific view of the system, suitable for understanding specific instances and their interactions.
Time Frame	Represents the structure that is valid for the entire lifecycle of the system.	Represents a specific snapshot or state of the system at a particular moment in time.
Usage	Used during the design phase to visualize and plan the structure of the system.	Used to illustrate specific scenarios or examples of how objects interact.

UML Class Diagram

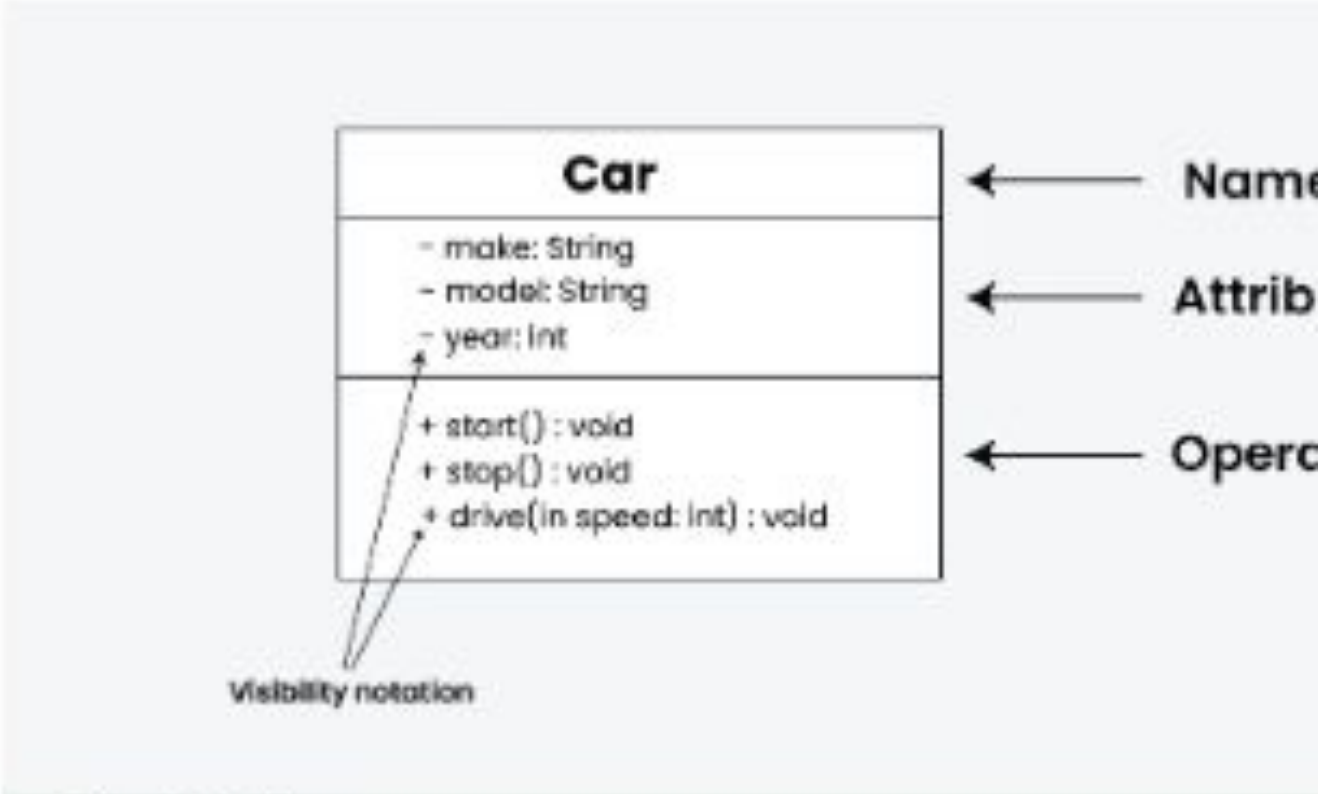
A UML class diagram visually represents the structure of a system by showing its classes, attributes, methods, and the relationships between them.

1. Helps everyone involved in a project—like developers and designers—understand how the system is organized and how its components interact.
2. Helps to communicate and document the structure of the software.



UML Class Notation

Classes are depicted as boxes, each containing three compartments for the class

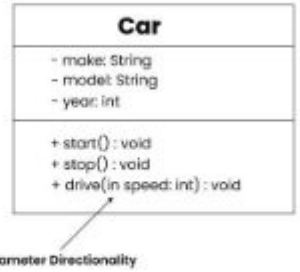


Class Notation

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.



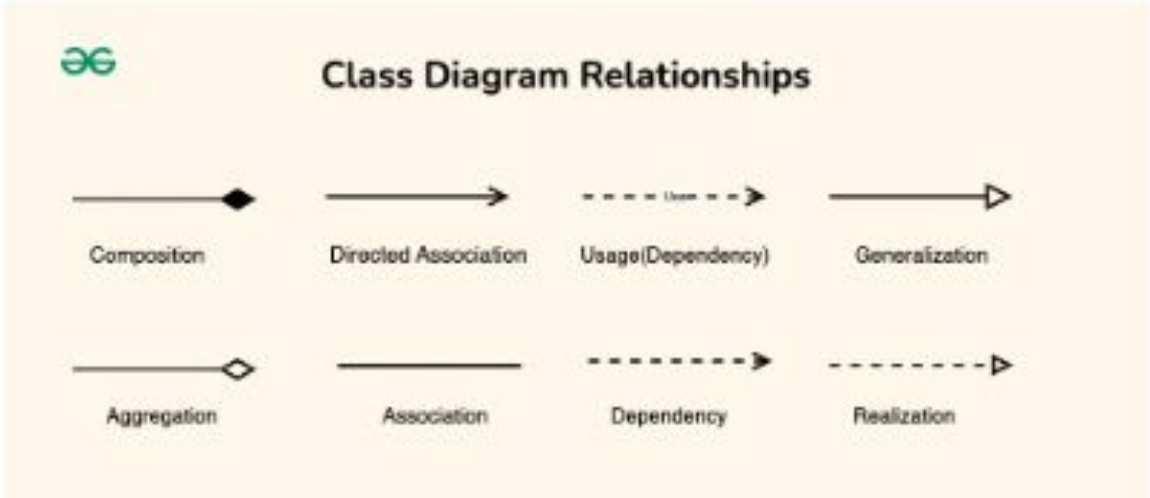
class notation with parameter directionality

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. 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.

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.

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.

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.

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.

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.

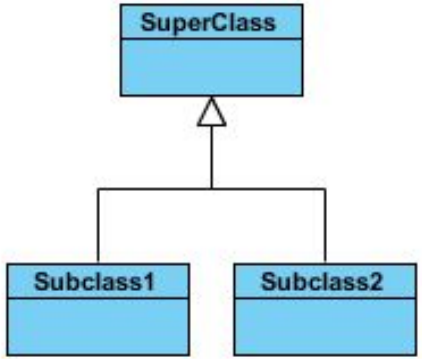
7. Dependency Relationship

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.

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.

Relationship Type	Graphical Representation
Inheritance (or Generalization): • Represents an "is-a" relationship. • An abstract class name is shown in italics. • SubClass1 and SubClass2 are specializations of Super Class. • A solid line with a hollow arrowhead that point from the child to the parent class	

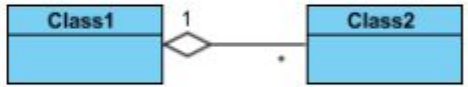
Simple Association:

- A structural link between two peer classes.
- There is an association between Class1 and Class2
- A solid line connecting two classes

Aggregation:

A special type of association. It represents a "part of" relationship.

- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.
- A solid line with an unfilled diamond at the association end connected to the class of composite



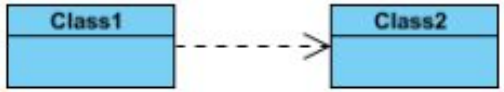
Composition:

A special type of aggregation where parts are destroyed when the whole is destroyed.

- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.
- A solid line with a filled diamond at the association connected to the class of composite

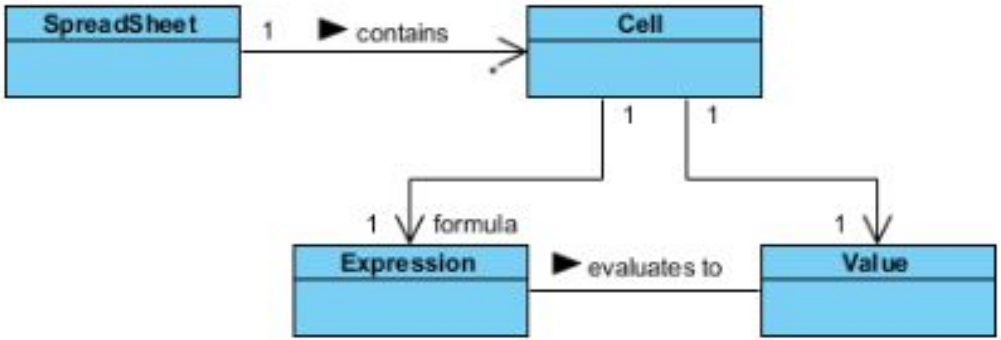
Dependency:

- Exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2
- A dashed line with an open arrow



Relationship Names

- Names of relationships are written in the middle of the association line.
- Good relation names make sense when you read them out loud:
  - "Every spreadsheet **contains** some number of cells",
  - "an expression **evaluates to** a value"
- They often have a **small arrowhead to show the direction** in which direction to read the relationship, e.g., expressions evaluate to values, but values do not evaluate to expressions.



Relationship - Roles

- A role is a directional purpose of an association.
- Roles are written at the ends of an association line and describe the purpose played by that class in the relationship.
  - E.g., A cell is related to an expression. The nature of the relationship is that the expression is the **formula** of the cell.

Access for each of these visibility types is shown below for members of different classes.

Access Right	public (+)	private (-)	protected (#)	Package (~)
Members of the same class	yes	yes	yes	yes
Members of derived classes	yes	no	yes	yes
Members of any other class	yes	no	no	in same package

Visibility of Class attributes and Operations

In object-oriented design, there is a notation of visibility for attributes and operations.

UML identifies four types of visibility: public, protected, private, and package.

The +, -, # and ~ symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.

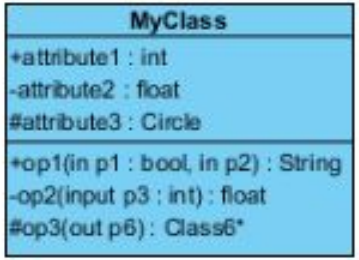
+ denotes public attributes or operations

- denotes private attributes or operations

# denotes protected attributes or operations

~ denotes package attributes or operations

Class Visibility Example

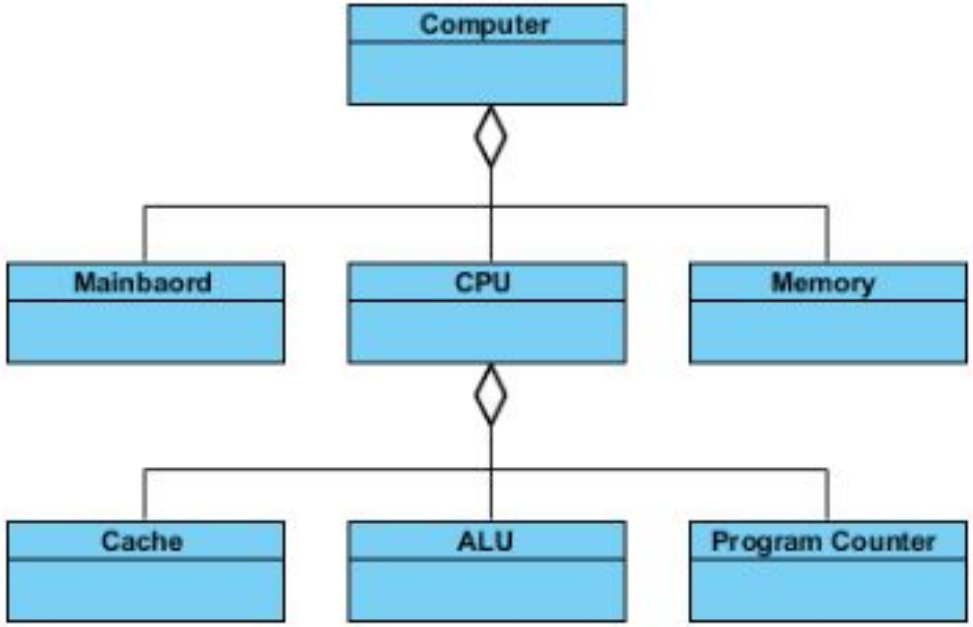


In the example above:

- attribute1 and op1 of MyClassName are public
- attribute3 and op3 are protected.
- attribute2 and op2 are private.

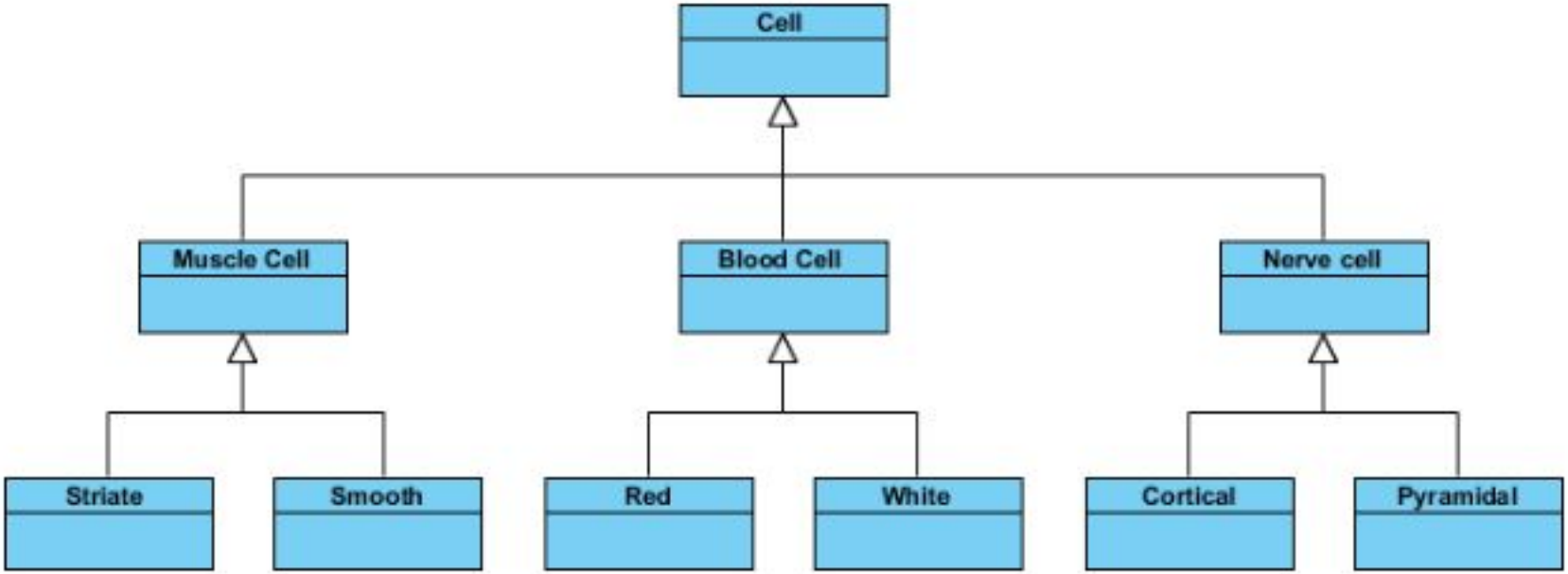
Aggregation Example - Computer and parts

- An aggregation is a special case of association denoting a "consists-of" hierarchy
- The aggregate is the parent class, the components are the children classes



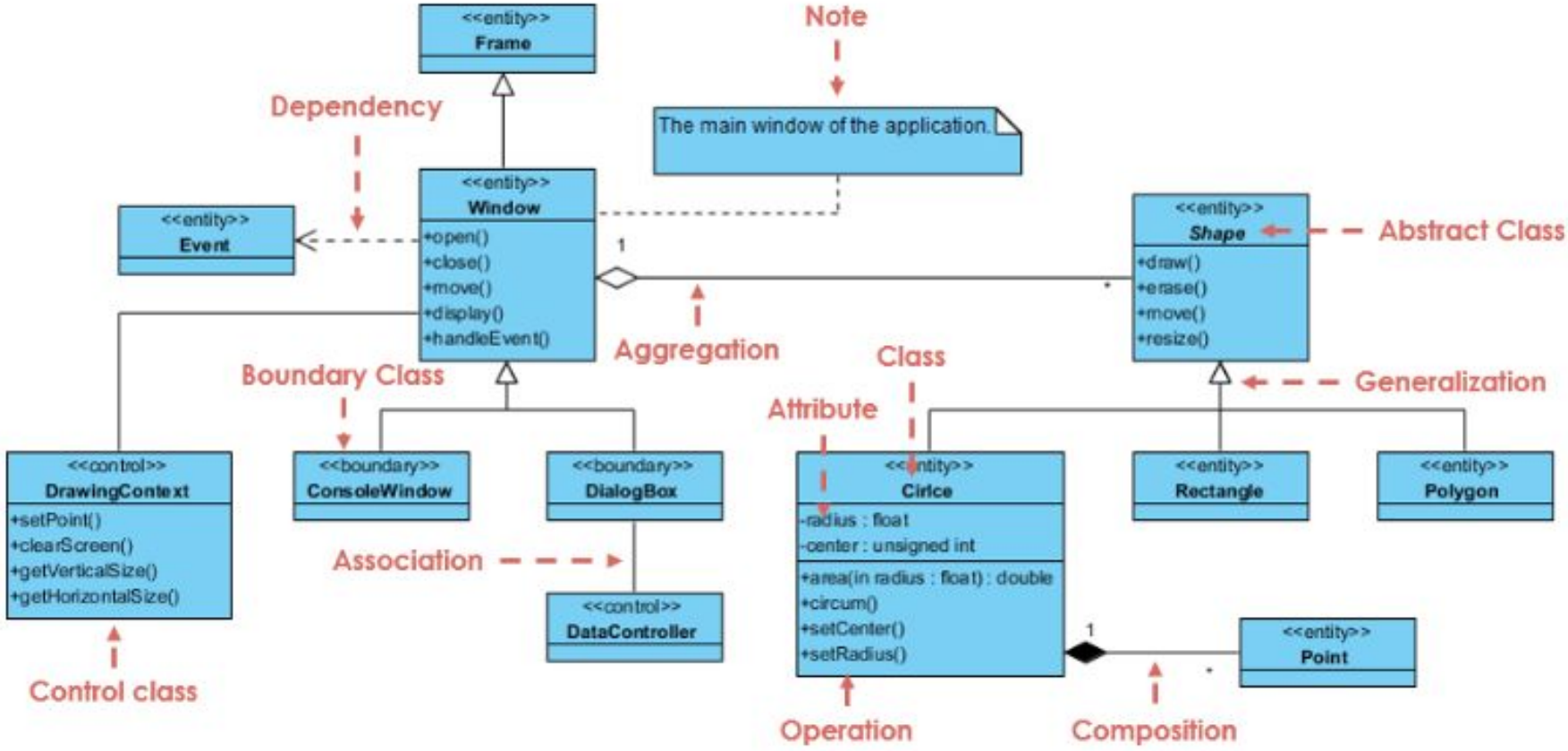
Inheritance Example - Cell Taxonomy

- Inheritance is another special case of an association denoting a "kind-of" hierarchy
- Inheritance simplifies the analysis model by introducing a taxonomy
- The child classes inherit the attributes and operations of the parent class.



Class Diagram - Diagram Tool Example

A class diagram may also have notes attached to classes or relationships. Notes are shown in grey.



In the example above:

We can interpret the meaning of the above class diagram by reading through the points as following.

1. Shape is an abstract class. It is shown in Italics.
2. Shape is a superclass. Circle, Rectangle and Polygon are derived from Shape. In other words, a Circle is-a Shape. This is a generalization / inheritance relationship.
3. There is an association between DialogBox and DataController.
4. Shape is part-of Window. This is an aggregation relationship. Shape can exist without Window.
5. Point is part-of Circle. This is a composition relationship. Point cannot exist without a Circle.
6. Window is dependent on Event. However, Event is not dependent on Window.
7. The attributes of Circle are radius and center. This is an entity class.
8. The method names of Circle are area(), circum(), setCenter() and setRadius().
9. The parameter radius in Circle is an in parameter of type float.
10. The method area() of class Circle returns a value of type double.
11. The attributes and method names of Rectangle are hidden. Some other classes in the diagram also have their attributes and method names hidden.

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.

Purpose of Class Diagrams

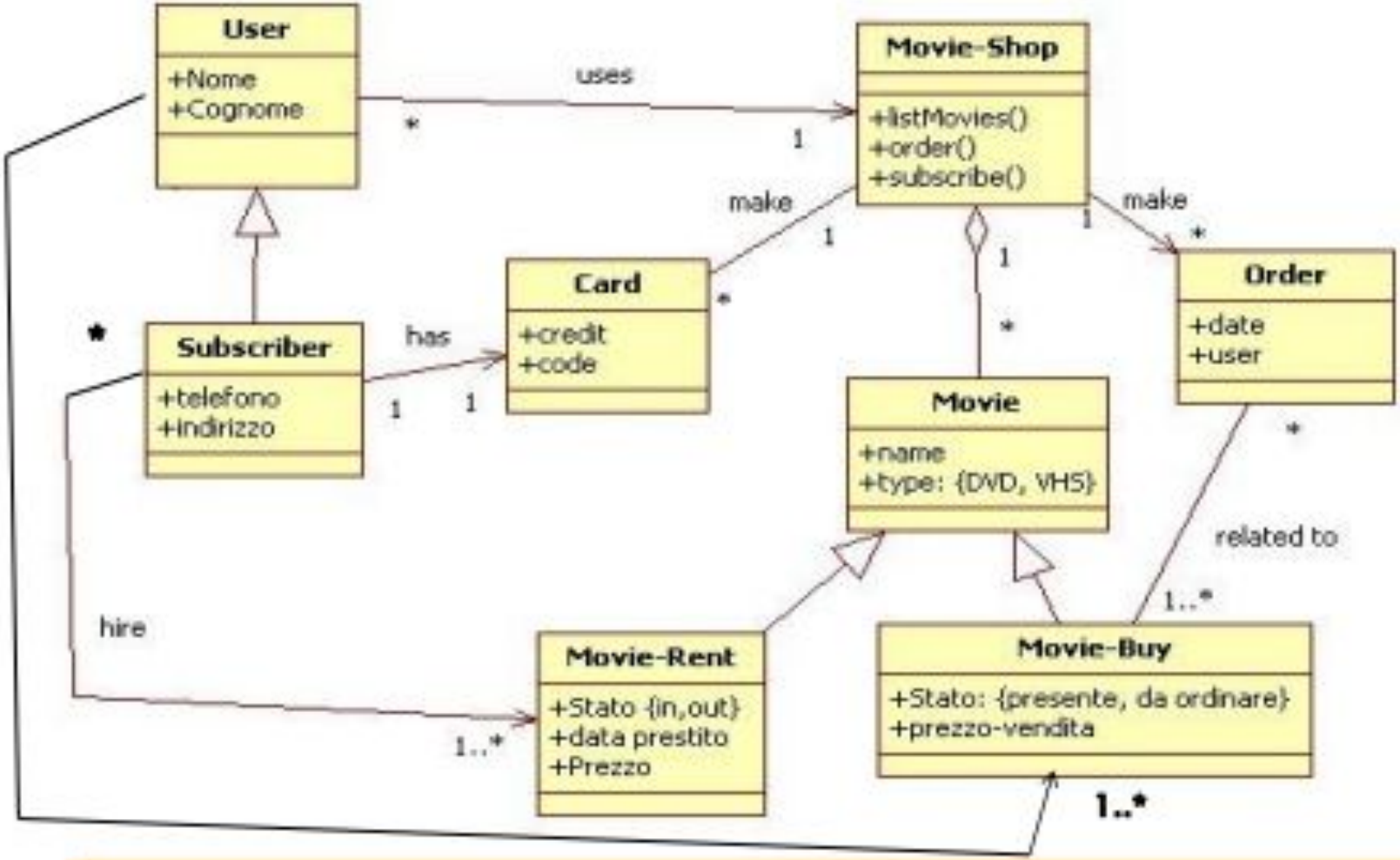
The main purpose of using class diagrams is:

- 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.

Solved Examples:

Movie- Shop

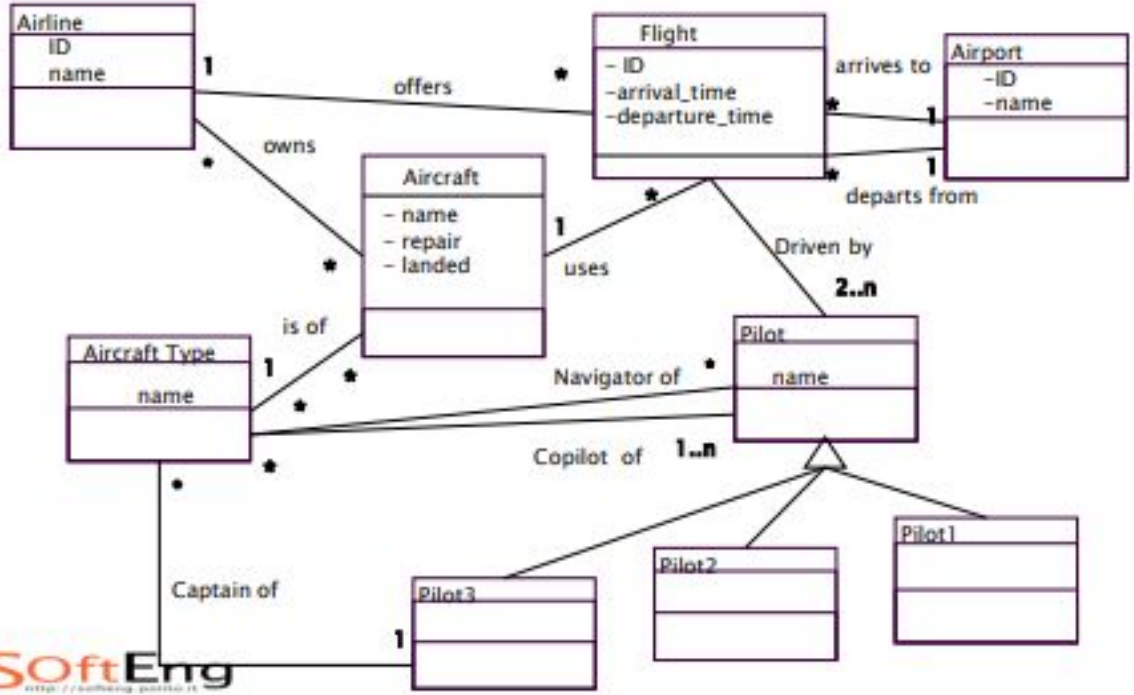
- Design a system for a movie-shop, in order to handle ordering of movies and browsing of the catalogue of the store, and user subscriptions with rechargeable cards.
- Only subscribers are allowed hiring movies with their own card.
- Credit is updated on the card during rent operations.
- Both users and subscribers can buy a movie and their data are saved in the related order.
- When a movie is not available it is ordered .


User
+Nome
+Cognome
uses
*
1
Movie-Shop
+listMovies()
+order()
+subscribe()
make
1
1
make
1
*
Order
+date
+user
Card
+credit
+code
*
1
*
*
Subscriber
+telefono
+indirizzo
has
1
1
Movie
+name
+type: {DVD, VHS}
*
related to
1..*
hire
Movie-Rent
+Stato {in,out}
+data prestito
+Prezzo
1..*
Movie-Buy
+Stato: {presente, da ordinare}
+prezzo-vendita
1..*

Flights

We want to model a system for management of flights and pilots.
An airline operates flights. Each airline has an ID.
Each flight has an ID a departure airport and an arrival airport: an airport as a unique identifier.
Each flight has a pilot and a co-pilot, and it uses an aircraft of a certain type; a flight has also a departure time and an arrival time.
An airline owns a set of aircrafts of different types.
An aircraft can be in a working state or it can be under repair.
In a particular moment an aircraft can be landed or airborne.
A company has a set of pilots: each pilot has an experience level: 1 is minimum, 3 is maximum.
A type of aeroplane may need a particular number of pilots, with a different role (e.g.: captain, co-pilot, navigator): there must be at least one captain and one co-pilot, and a captain must have a level 3.

Flights – solution



Question 1:

A company consists of departments. Departments are located in one or more offices. One office acts as a headquarter. Each department has a manager who is recruited from the set of employees. Your task is to model the system for the company.

Task:

Draw a class diagram which consists of all the classes in your system their attributes and operations, relationships between the classes, multiplicity specifications, and other model elements that you find appropriate.

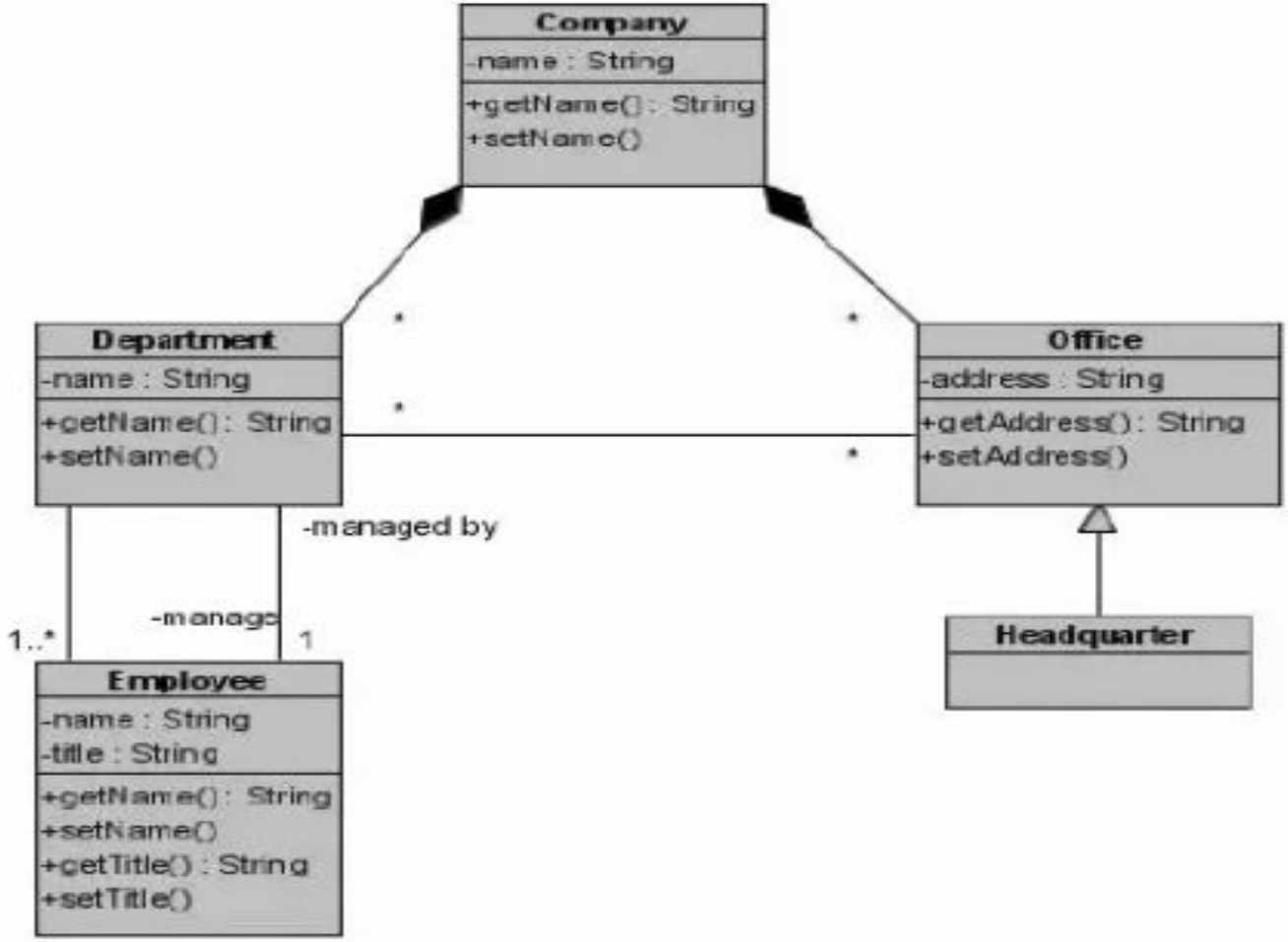


Figure 1 – Company consists of Departments Class Diagram

Question 2:

A token-ring based local-area-network (LAN) is a network consisting of nodes, in which network packets are sent around. Every node has a unique name within the network, and refers to its next node. Different kinds of nodes exist: workstations are originators of messages; servers and printers are network nodes that can receive messages. Packets contain an originator, a destination and content, and are sent around on a network. A LAN is a circular configuration of nodes.

Task:

Draw a class diagram which consists of all the classes in your system their attributes and operations, relationships between the classes, multiplicity specifications, and other model elements that you find appropriate.

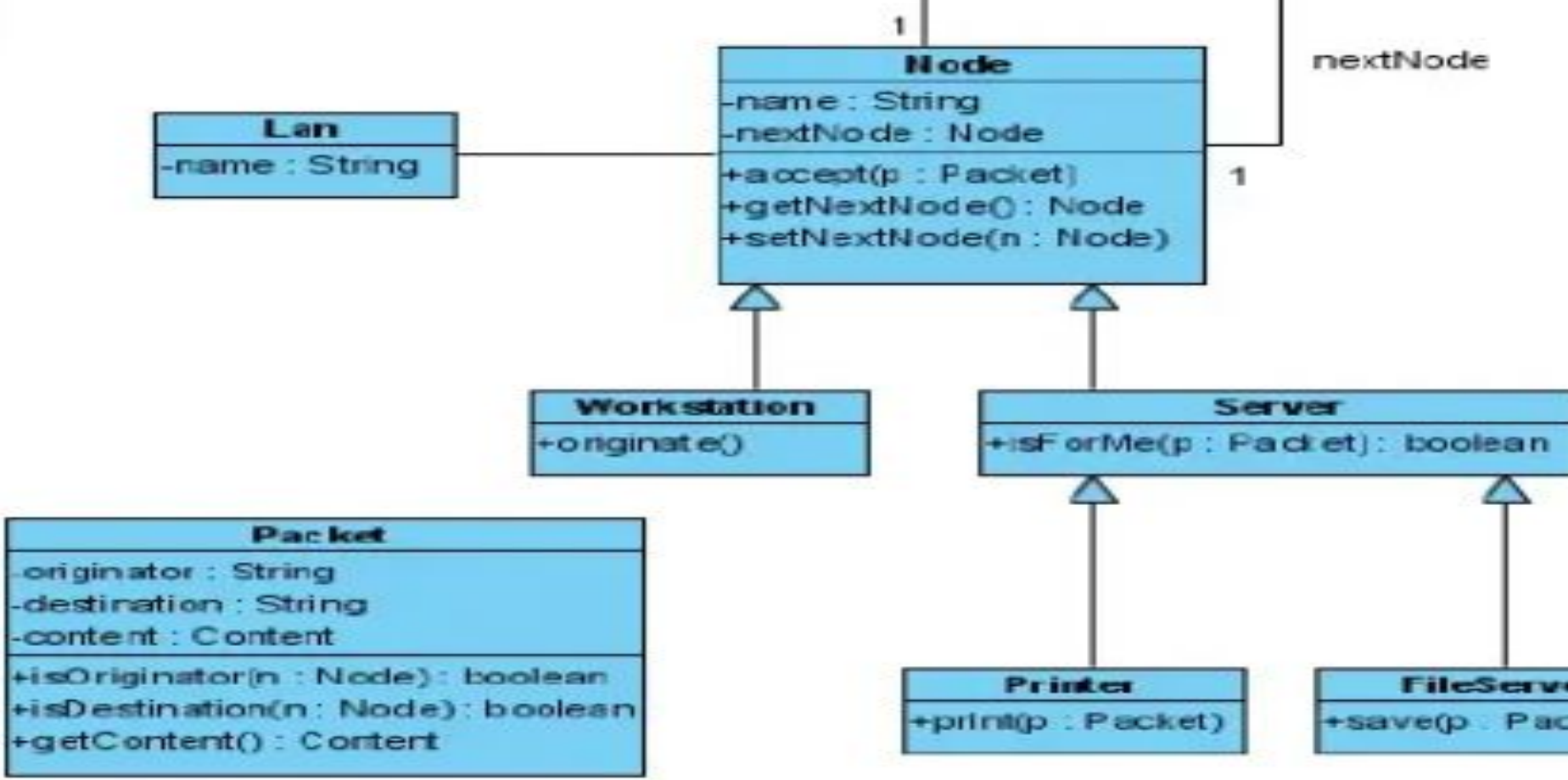


Figure 2 – A token-ring based local-area-network Class Diagram

Solution:

Question 3:

Model a LAN in which we find two kinds of printers: ASCII printers can only print ASCII documents. PostScript printers can print either ASCII or Postscript documents. The documents are sent around as content of packets.

Task:

Adapt the class diagram from the previous example to support this behavior. Don't forget attributes and operations, relationships between the classes, multiplicity specifications, and other model elements that you find appropriate.

Solution

Figure 3 shows the class diagram in considering additional printing requirements. Note that no case statements or explicit checks are needed when printing documents: the double dispatch mechanism takes care of that. The pieces of code illustrate the mechanism. Figure 4 shows the double dispatch mechanism in action using a sequence diagram. Sophie sends a packet on the network, with as destination the laser printer. The printer is the next node, so it immediately arrives at the destination. The double dispatch mechanism is highlighted.

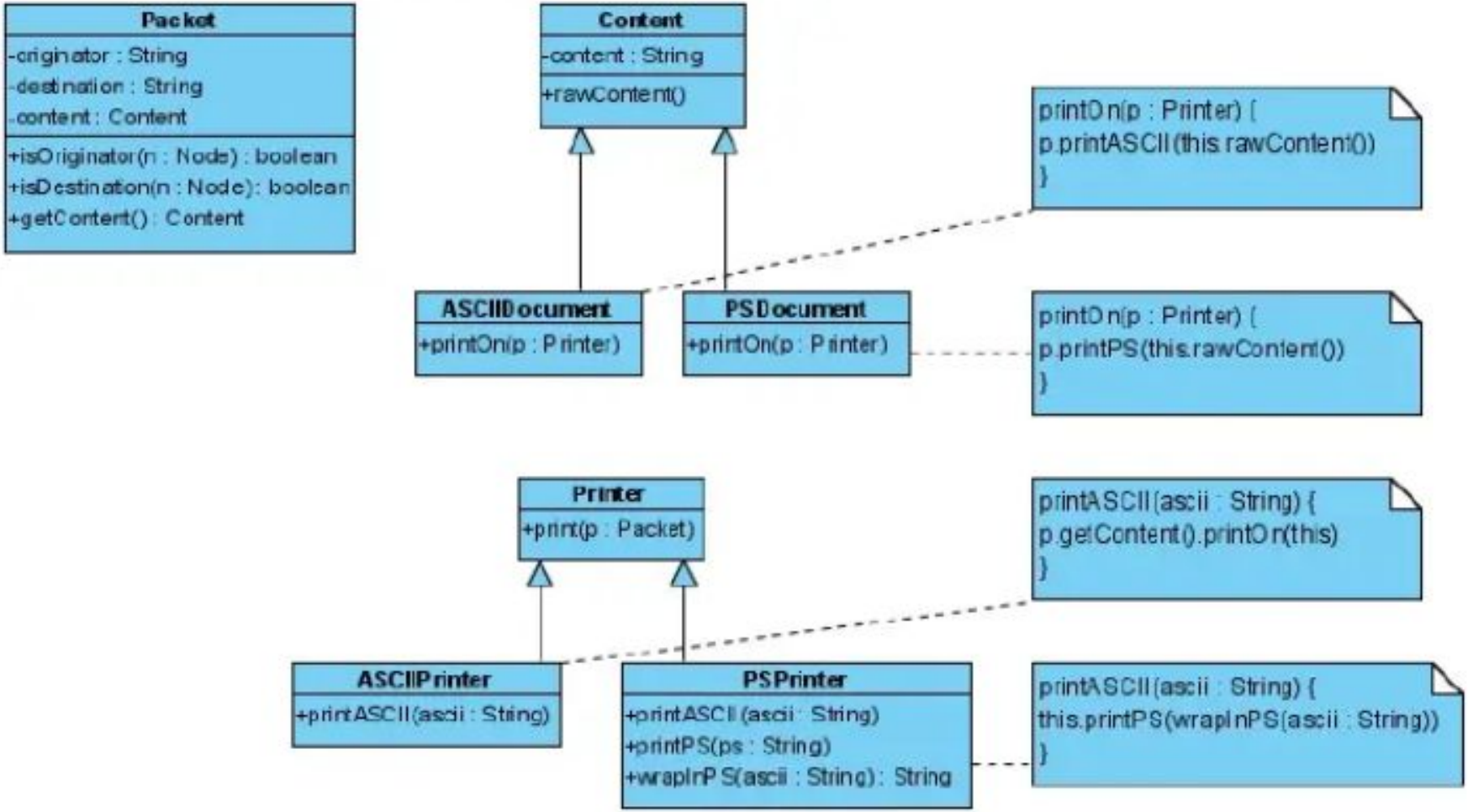


Figure 3 – Printing Refined Class Diagram

https://youtu.be/iLsJ0Ix_dho?si=gtXmGdBP_98BQKuO

Srs example

https://youtu.be/PtJmjPkrSUE?si=Rmqe6HZ55N184Ls4

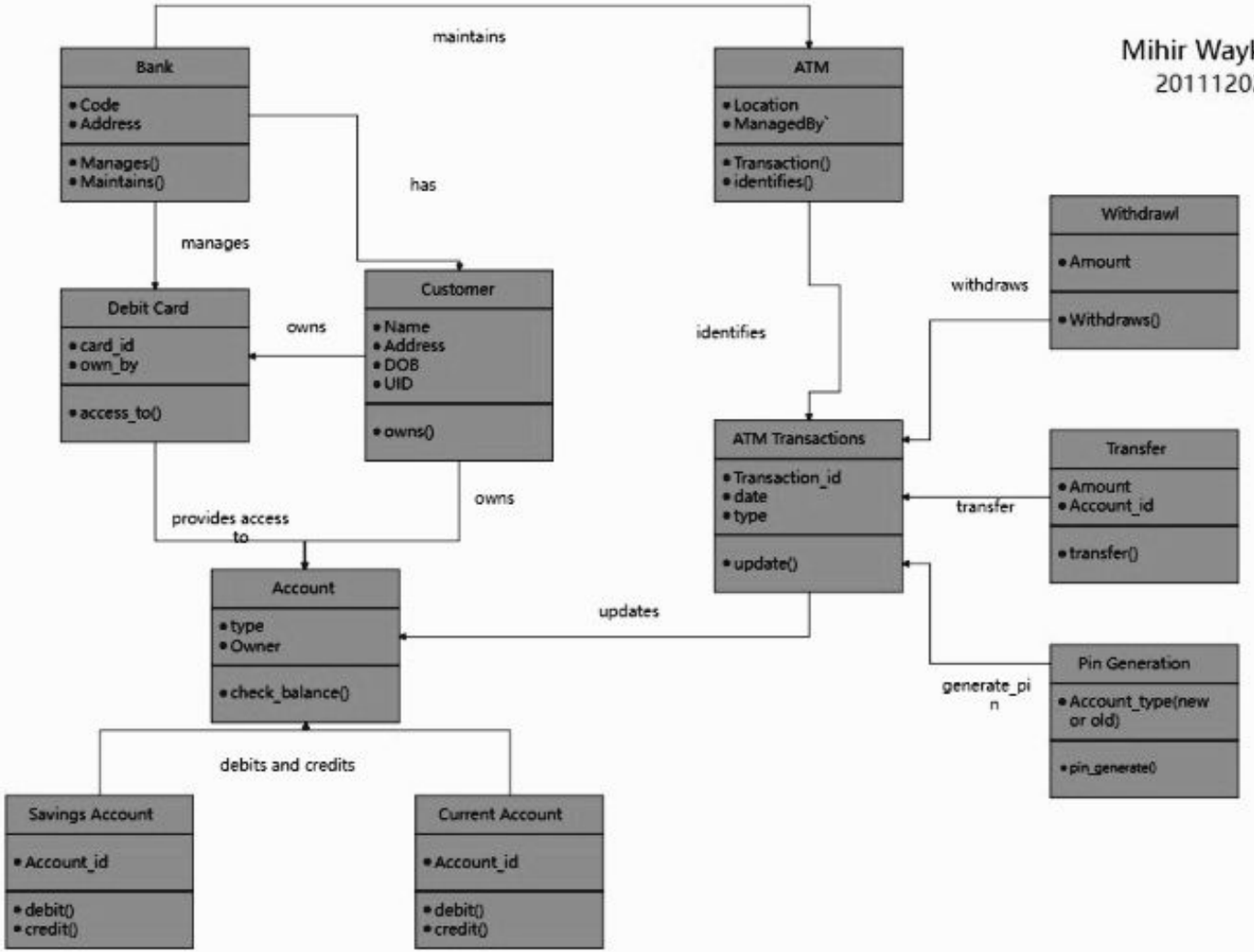
srs-example.pdf | Web Development | Internet
https://share.google/GKC5xLBYWeBJZXnwW

Class Diagram for ATM

This ATM class diagram maps the structure and attributes of automated teller machine operations. It demonstrates relationships between multiple classes, providing a comprehensive overview of ATM functionality. Use this template as-is or customize it to meet your specific requirements.

https://edraw.wondershare.com/example-uml-class-diagram.html?srsltid=AfmBOoo9AJc3LnIGZtjJ2JORgef275rM7vSRuNQpbGQ2s1n1nH4Wxs5h

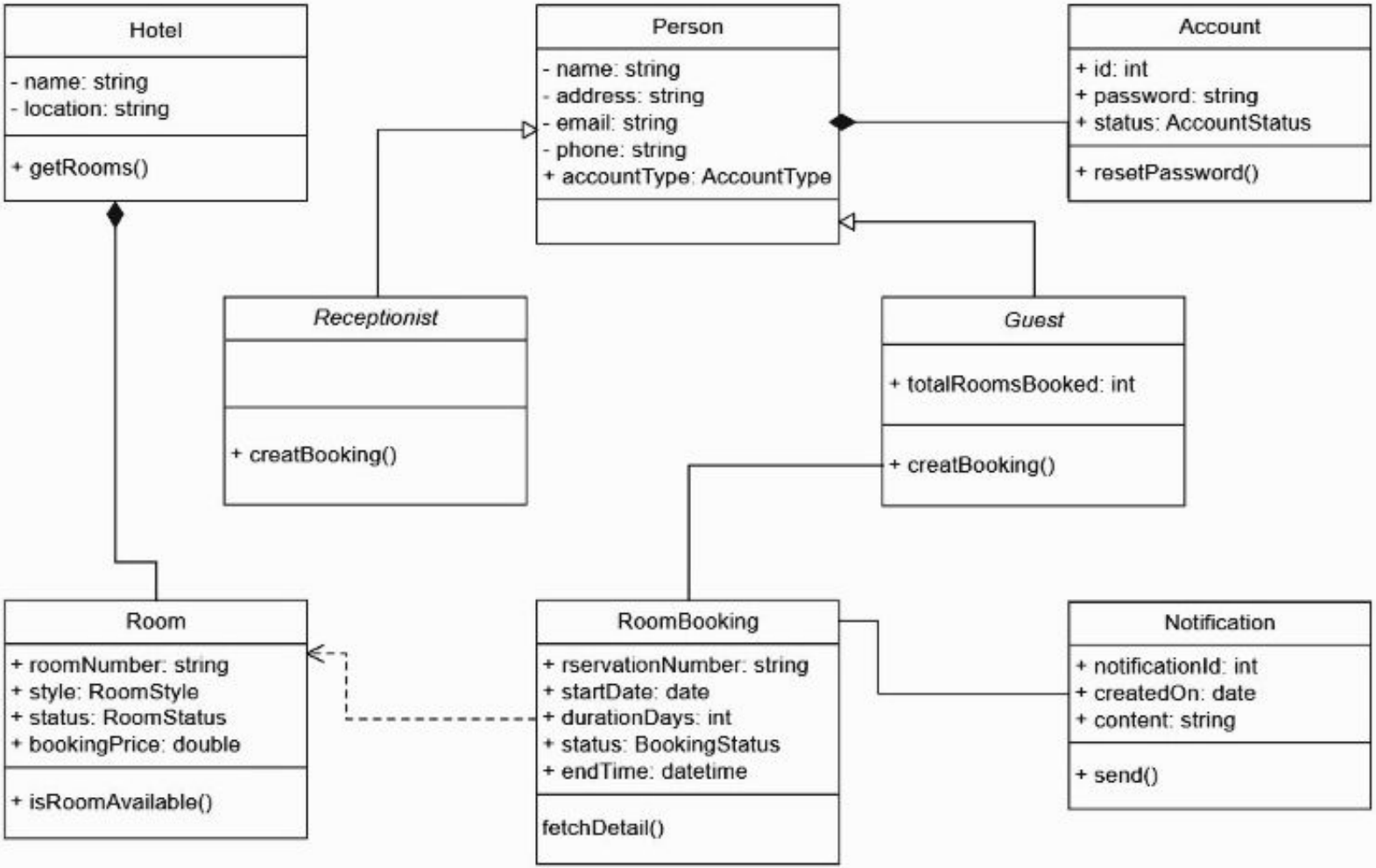
Mihir Waykole
201112028



Class Diagram for Hotel Management System

This hotel management class diagram meticulously connects all classes through arrows to illustrate their relationships. Easily customize this comprehensive diagram by adding additional classes to match your specific hotel management needs.

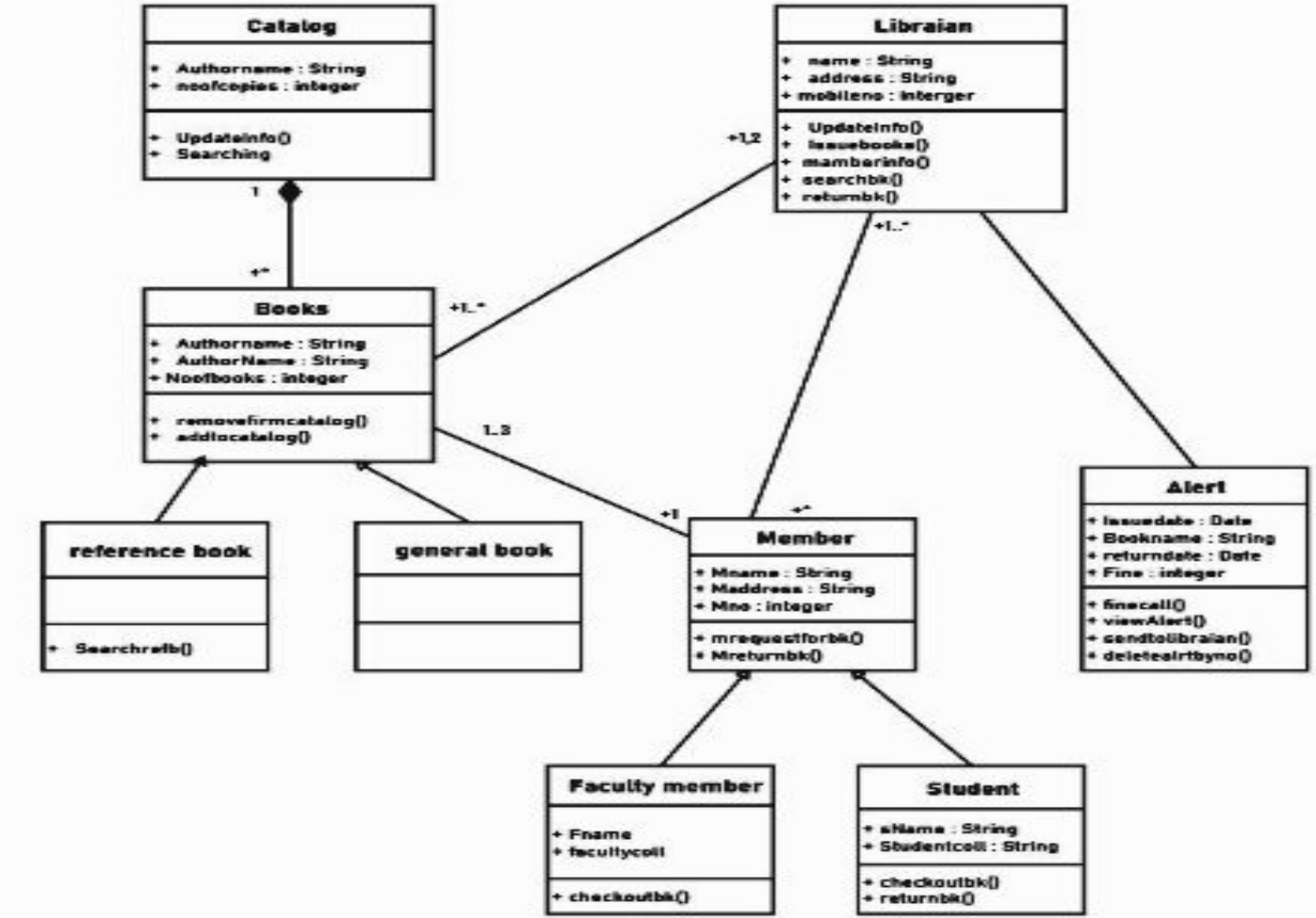
Hotel Management System



Class Diagram for Library Management System

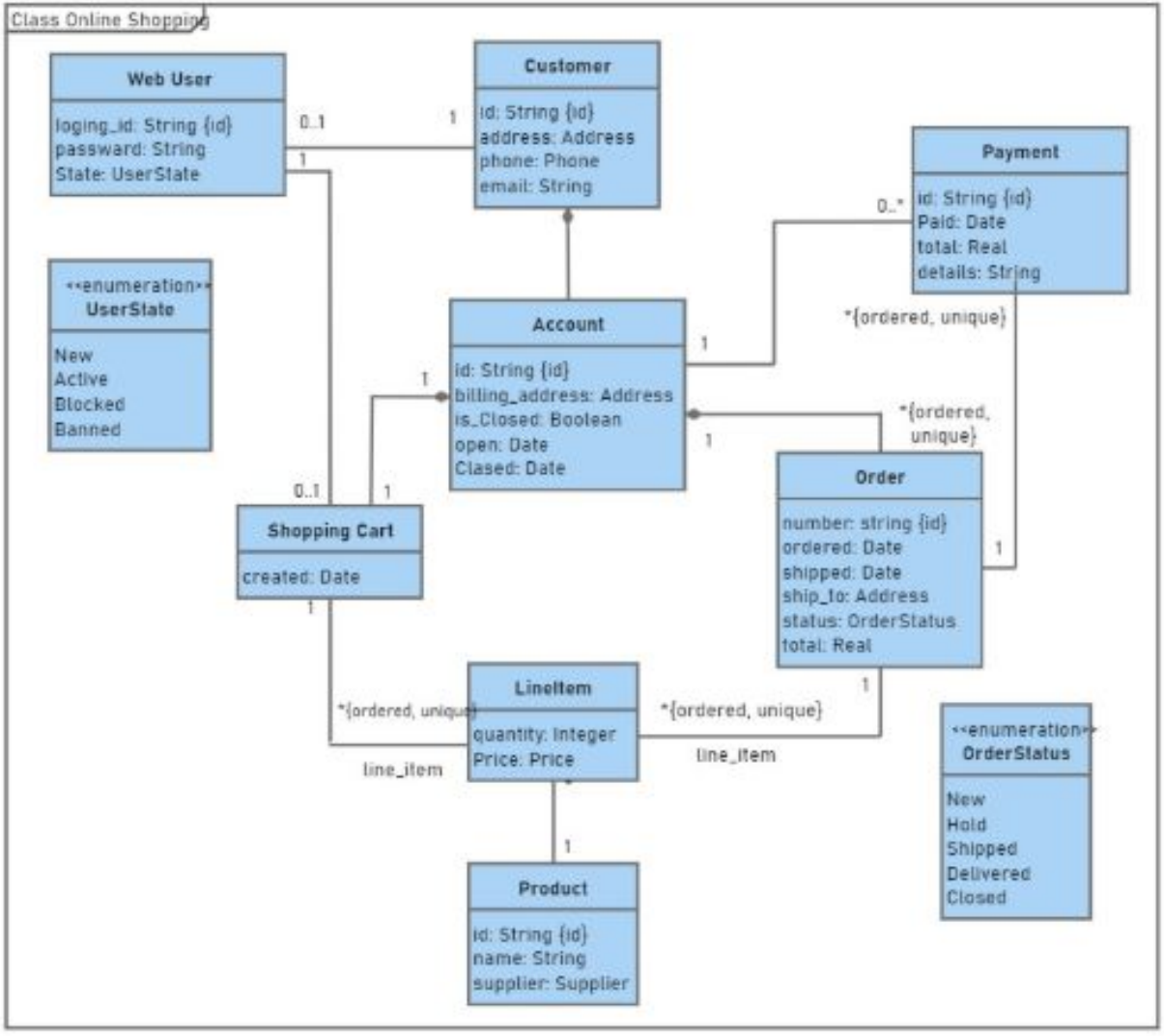
This library management system class diagram features multiple classes including user, librarian, book, and account. It details the attributes and operations of each class while demonstrating their interrelationships within the library management context.

Library System UML Diagram



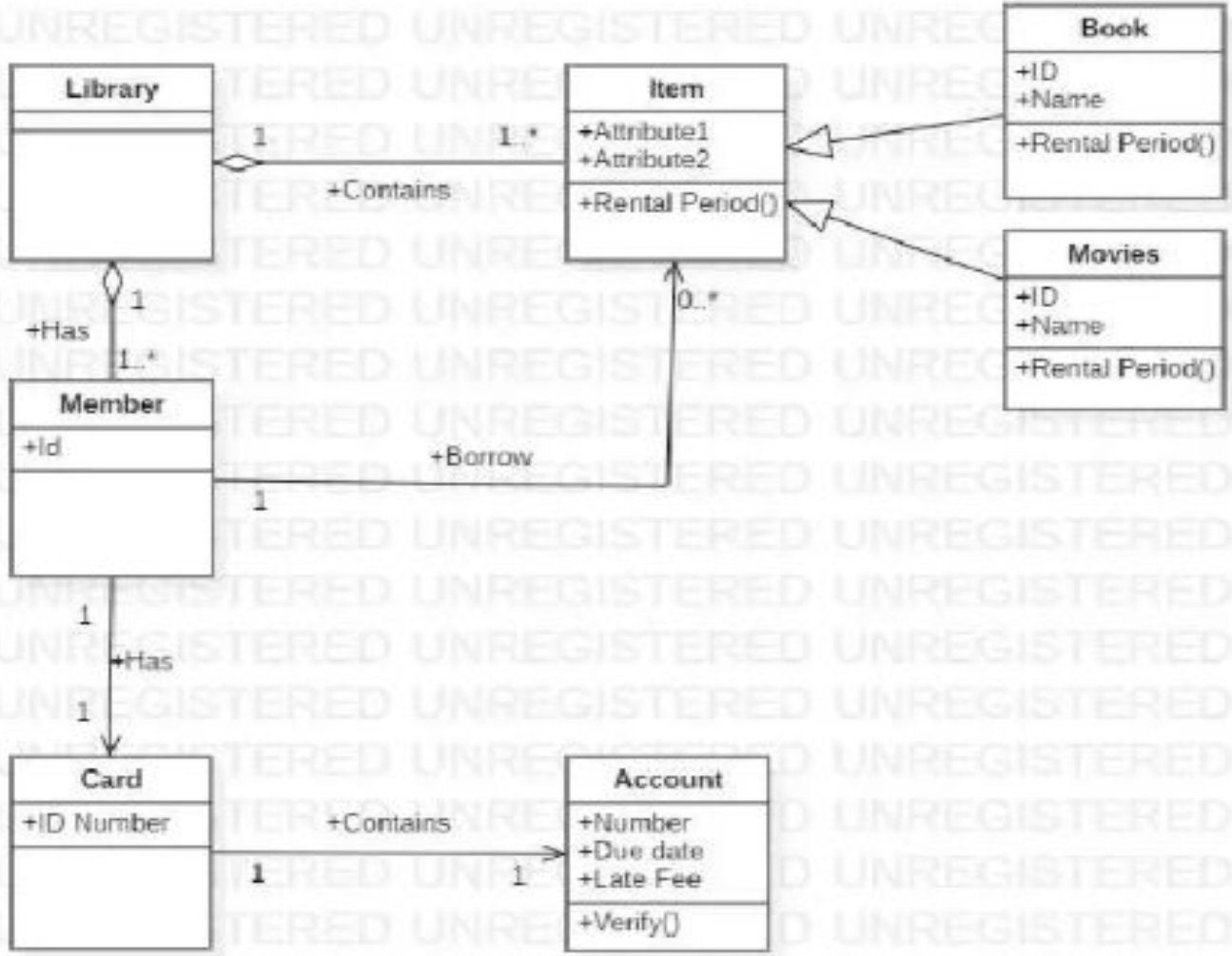
Class Diagram for Online Shopping

This online shopping class diagram presents the domain model for e-commerce systems. It clearly illustrates how classes like user and account interact to facilitate order placement and shipment processes, making it accessible for both software engineers and business analysts.


Class Online Shopping
Web User
loging_id: String {id}
password: String
State: UserState
Customer
id: String {id}
address: Address
phone: Phone
email: String
Payment
id: String {id}
Paid: Date
total: Real
details: String
<<enumeration>>
UserState
New
Active
Blocked
Banned
Account
id: String {id}
billing_address: Address
is_Closed: Boolean
open: Date
Clased: Date
Shopping Cart
created: Date
Order
number: string {id}
ordered: Date
shipped: Date
ship_to: Address
status: OrderStatus
total: Real
LineItem
quantity: Integer
Price: Price
Product
id: String {id}
name: String
supplier: Supplier
<<enumeration>>
OrderStatus
New
Hold
Shipped
Delivered
Closed
0..1
1
0..*
*{ordered, unique}
line_item

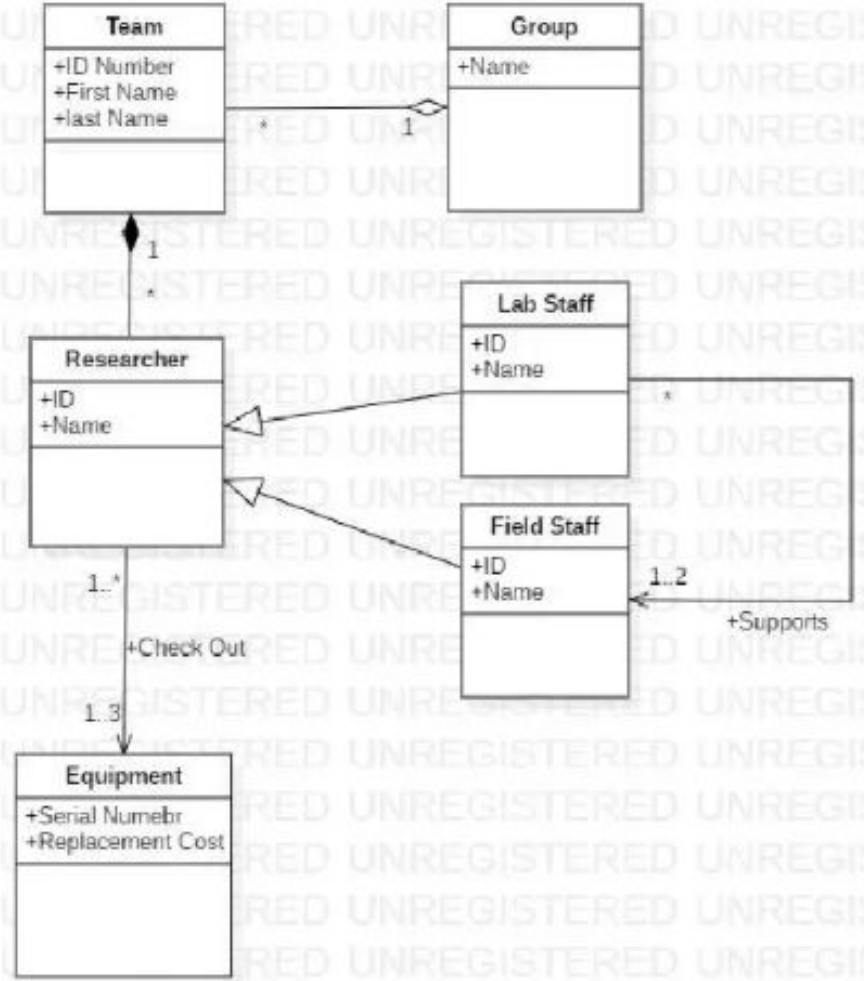
Task 1: Create a class diagram for a web based public library. A public library typically stores a collection of books, movies, or other library items available to be borrowed by people living in a community. Each library member typically has a library account and a library card with the account's ID number, which she can use to identify herself to the library. A member's library account records which items the member has borrowed and the due date for each borrowed item. Each type of item has a default rental period, which determines the item's due date when the item is borrowed. If a member returns an item after the item's due date, the member owes a late fee specific for that item, an amount of money recorded in the member's library account

Solution:


Library
Item
+Attribute1
+Attribute2
+Rental Period()
Book
+ID
+Name
+Rental Period()
Movies
+ID
+Name
+Rental Period()
1
1..*
+Contains
+Has
1
1..*
0..*
Member
+Id
+Borrow
1
1
+Has
1
Card
+ID Number
+Contains
1
1
Account
+Number
+Due date
+Late Fee
+Verify()

Task 2: Create a class diagram based on the following description. You have been asked to build a management system for a group of archeologists. The group is comprised of multiple teams of researchers. Each team has a letter ID (e.g., team A, team B). Each researcher belongs to one of the teams, and has an ID number, a first name, and a last name. There are two types of researchers: field and lab staff. Each field staff member has a favorite region (string). Each lab researcher supports up to 2 field researchers. Some researchers may not be supported by a lab researcher. The company also manages an inventory of equipment. Researchers of any type may check out up to 3 pieces of equipment. Each piece of equipment has a serial number and replacement cost.

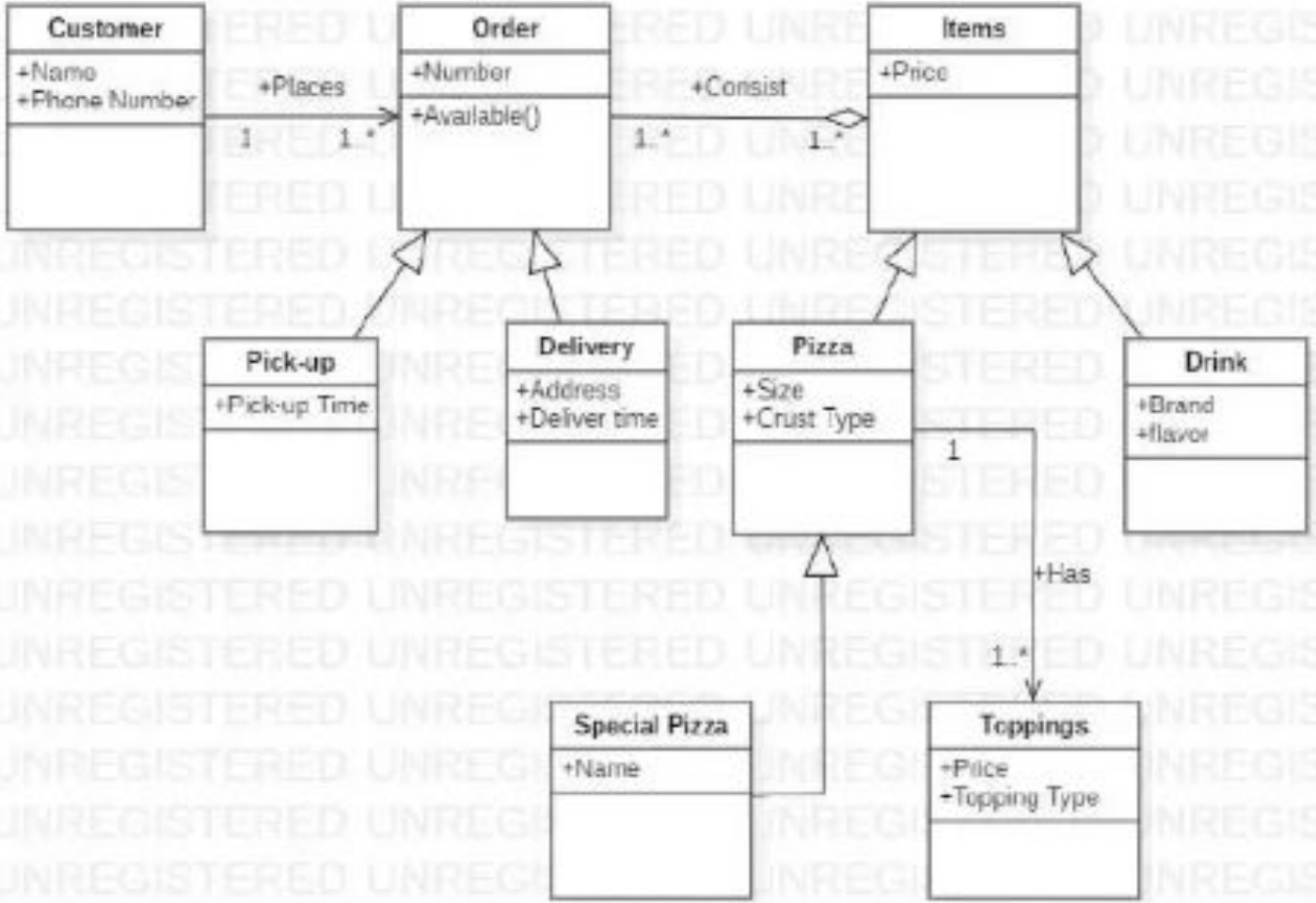
Solution:


Team
+ID Number
+First Name
+last Name
Group
+Name
*
1
1
*
Researcher
+ID
+Name
Lab Staff
+ID
+Name
*
Field Staff
+ID
+Name
1..2
+Supports
1..*
+Check Out
1..3
Equipment
+Serial Numebr
+Replacement Cost

Task 3: Imagine that you are tasked with developing a system for a pizza shop. Given the following description, create a class diagram (in the form of a UML class diagram). Include all conceptual classes, attributes, associations, and generalization relationships mentioned in the descriptions. Label all associations and include all multiplicities.

A customer places orders. A customer has a name and phone number. There are two types of orders: pick-up and delivery. A pick-up order has a pick-up time. A delivery order has an address and deliver-by time. All orders consist of a set of items. There are two types of items: pizzas and drinks. All items have a price. A pizza has a size and a crust type. A pizza also has a number of toppings. A topping has a topping type and a price. Some pizzas are special pizzas that have a name (e.g., "Hawaiian" or "Meat Lovers"). A drink has a brand and a flavor

Solution:


Customer
+Name
+Phone Number
+Places
1
1..*
Order
+Number
+Available()
+Consist
1..*
1..*
Items
+Price
Pick-up
+Pick-up Time
Delivery
+Address
+Deliver time
Pizza
+Size
+Crust Type
Drink
+Brand
+flavor
1
+Has
1..*
Special Pizza
+Name
Toppings
+Price
+Topping Type

Problem Statement

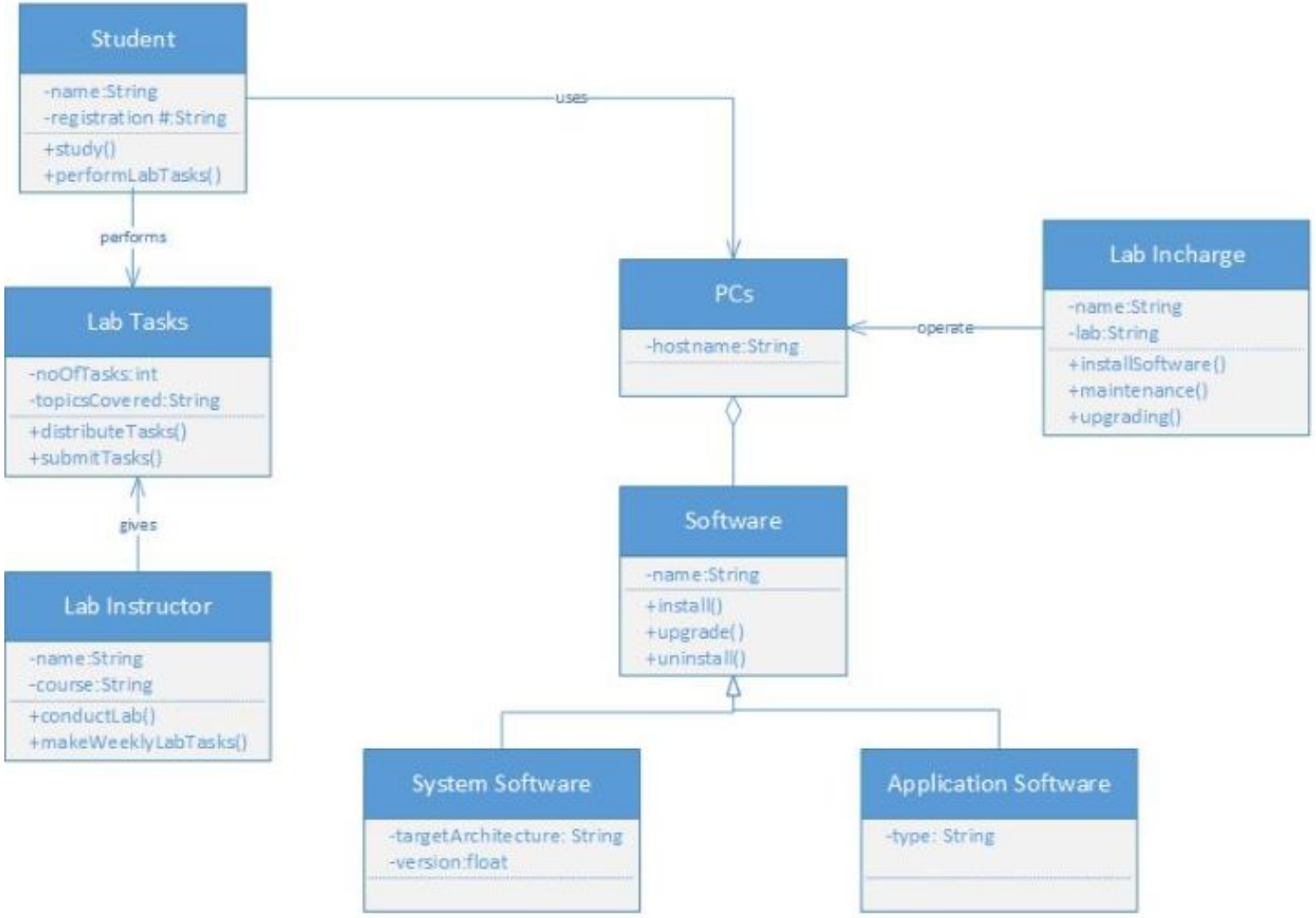
"Develop a UML class diagram that models the working of computer labs in your department. It relates Students, Lab Instructor, weekly Lab Tasks, PCs, and Software installed on the PCs. There are two types of software installed i.e. System Software and Application Software. All the PCs are connected through LAN with a server which is being operated by a lab in charge. The lab in charge is also responsible for installation and maintenance of System and Application Software for all the systems in the lab."

The tasks to be performed are:

1. Extract the main objects (entities) for above system.
2. Find the necessary attributes and functions that need to be associated with each object (You are required to mention at most three attributes and one functions for a class).
3. Identify the relationships between these objects.
4. Construct a final comprehensive Class diagram showing all objects and their relationships along with their attributes and functions.

Important things to consider:

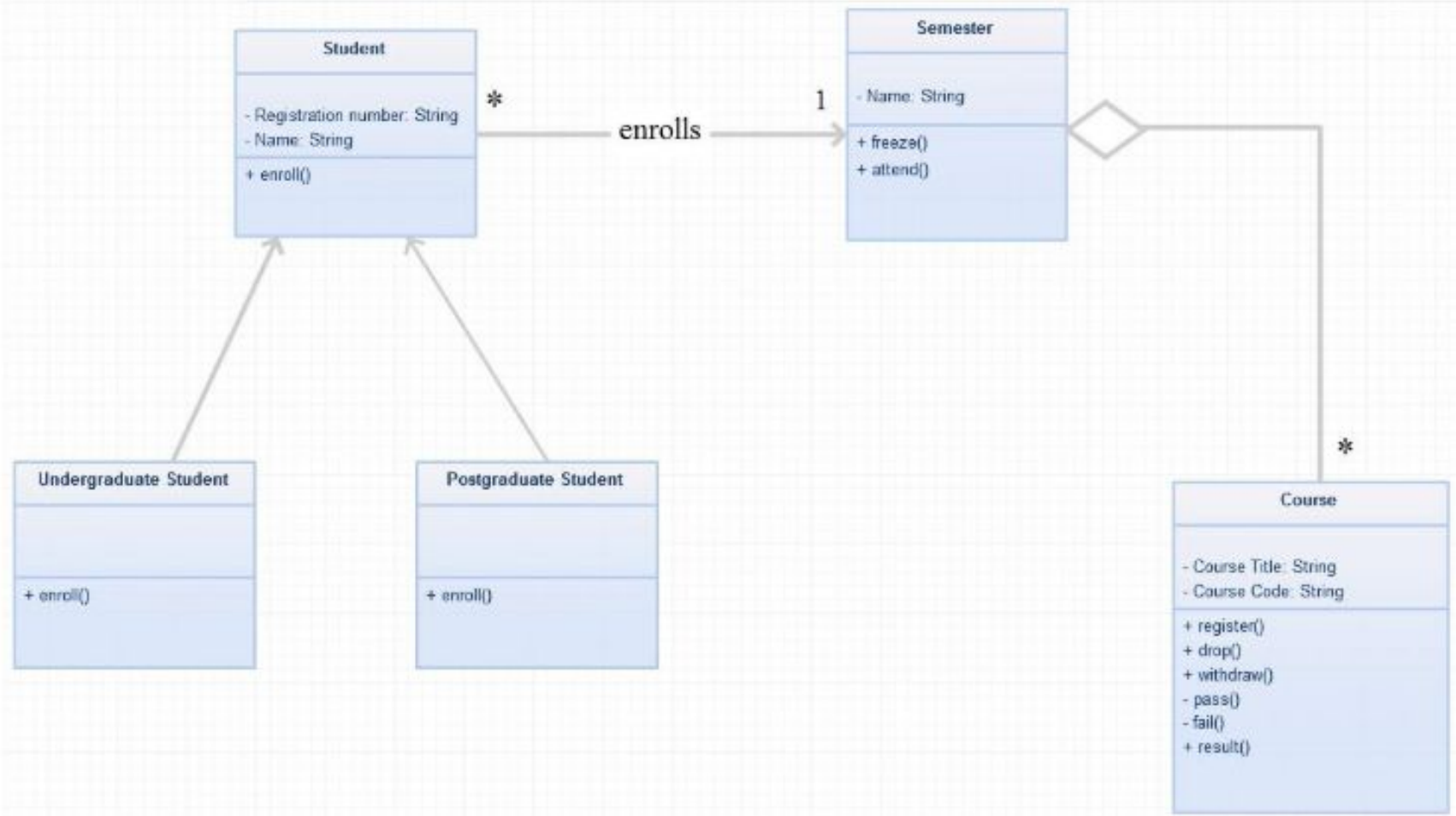
`You have to use standard UML notations for objects, classes, and their associations.` **Note:**


Student
-name:String
-registration #:String
+study()
+performLabTasks()
uses
performs
Lab Tasks
-noOfTasks:int
-topicsCovered:String
+distributeTasks()
+submitTasks()
gives
Lab Instructor
-name:String
-course:String
+conductLab()
+makeWeeklyLabTasks()
PCs
-hostname:String
operate
Lab Incharge
-name:String
-lab:String
+installSoftware()
+maintenance()
+upgrading()
Software
-name:String
+install()
+upgrade()
+uninstall()
System Software
-targetArchitecture: String
-version:float
Application Software
-type: String

Problem Statement

"Develop a Student Registration System in which student gets enroll in a semester. The semester contains courses. Each course has a title and course Code. The course can be registered, dropped, withdraw, passed and failed by the students. Also a semester may be freezed or attended. There are two kinds of students; Undergraduate Students (for BCS & MCS) and Postgraduate Students (for MPhil & PhD). Each type of students enroll in different ways."

Solution


Student
- Registration number: String
- Name: String
+ enroll()
*
enrolls
1
Semester
- Name: String
+ freeze()
+ attend()
*
Undergraduate Student
+ enroll()
Postgraduate Student
+ enroll()
Course
- Course Title: String
- Course Code: String
+ register()
+ drop()
+ withdraw()
- pass()
- fail()
+ result()

UML-class-diagram---Course-Information-System

Converting a problem statement into class diagram.

Problem Statement

"Develop an application for a university to provide an interface of viewing course information for staff, students and managers.

The lecturers and courses details are listed in a prospectus in which courses or lecturers can be added or removed. Courses at the college comprise a number of units at two levels, with students normally taking level one units in the first year of their course and progressing level two units in the second. Part time students may take two or more years to complete all their units at a given level. Level one units are assessed on course work grades, but level two units are assessed with examinations. Some courses also include a compulsory foundation unit where particular technical skills need to be established.

The system will provide lists of lecturers, courses and units, and access to more detailed information about which units are taught on which courses. Individual lecturer records may be queried for information such as their office room number, e-mail address and subject specialisms, as well as reporting the units that a lecturer teaches. Each course in the prospectus can be identified by a unique course code, and consists of a number of units at each level. Every course has one lecturer acting as course leader, who is responsible for administering that course and updating unit descriptions. Management functions of the prospectus allow lecturers and courses to be added to or removed from the system."

The tasks to be performed are:

1. Extract the main objects (entities) for above system.
2. Find the necessary attributes and functions that need to be associated with each object (You are required to mention at most three attributes and one functions for a class).
3. Identify the relationships between these objects.
4. Construct a final comprehensive Class diagram showing all objects and their relationships along with their attributes and functions.

Important things to consider:

```
You have to use standard UML notations for objects, classes, and their associations.
```

Note:

Besides problem statement; objects, properties, and functions of a system can also be extracted from domain knowledge. You have to consider all OOP concepts like abstraction, encapsulation, inheritance, association, generalization, and specialization wherever applicable.

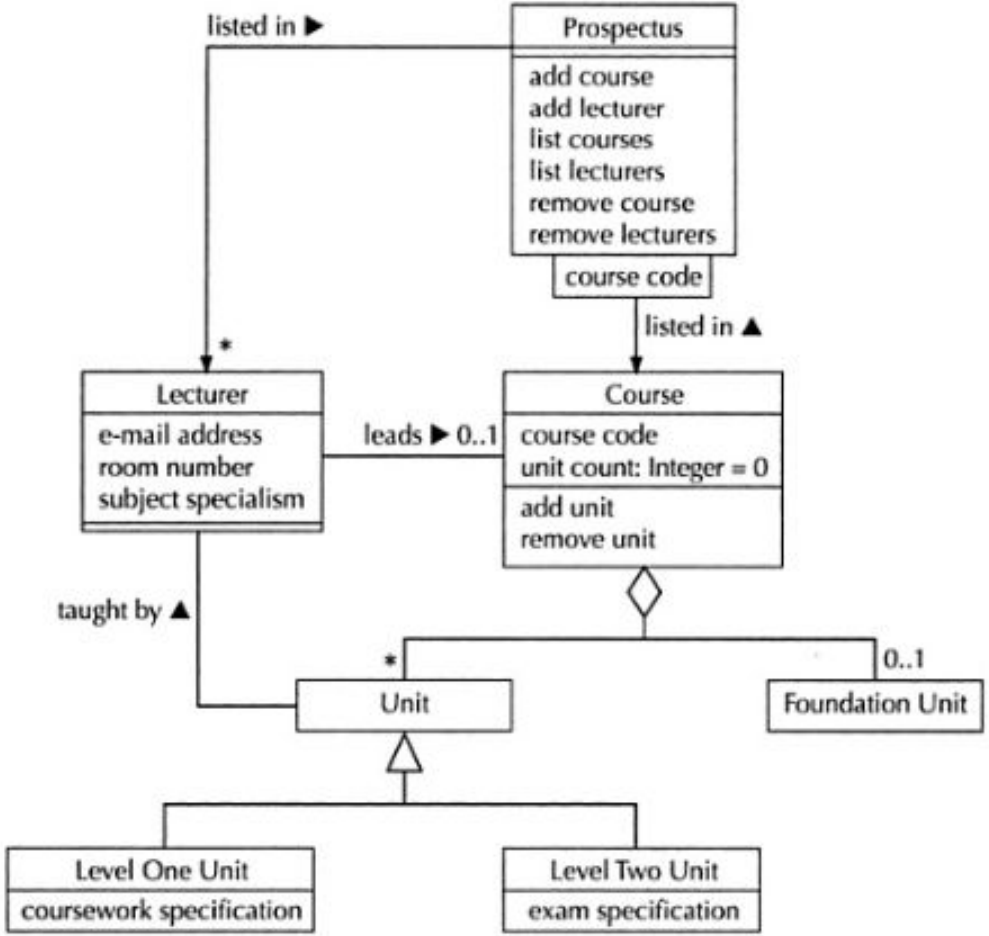


Fig. : The class diagram showing associations and their multiplicity, also aggregations, inheritance and some attributes and operations.

Task 1 - Problem Statement to UML Class conversion

"Develop a Banking System in which customer can open account. The account has the functionality of deposit, withdraw and get balance. There are two kinds of account; Current Account and Saving Account. Each kind of accounts withdraw in different ways. The account is identified by account number."

The tasks to be performed are:

1. Extract the main objects (entities) for above system.
2. Find the necessary attributes and functions that need to be associated with each object (You are required to mention at most three attributes and one functions for a class).
3. Identify the relationships between these objects.
4. Construct a final comprehensive Class diagram showing all objects and their relationships along with their attributes and functions.

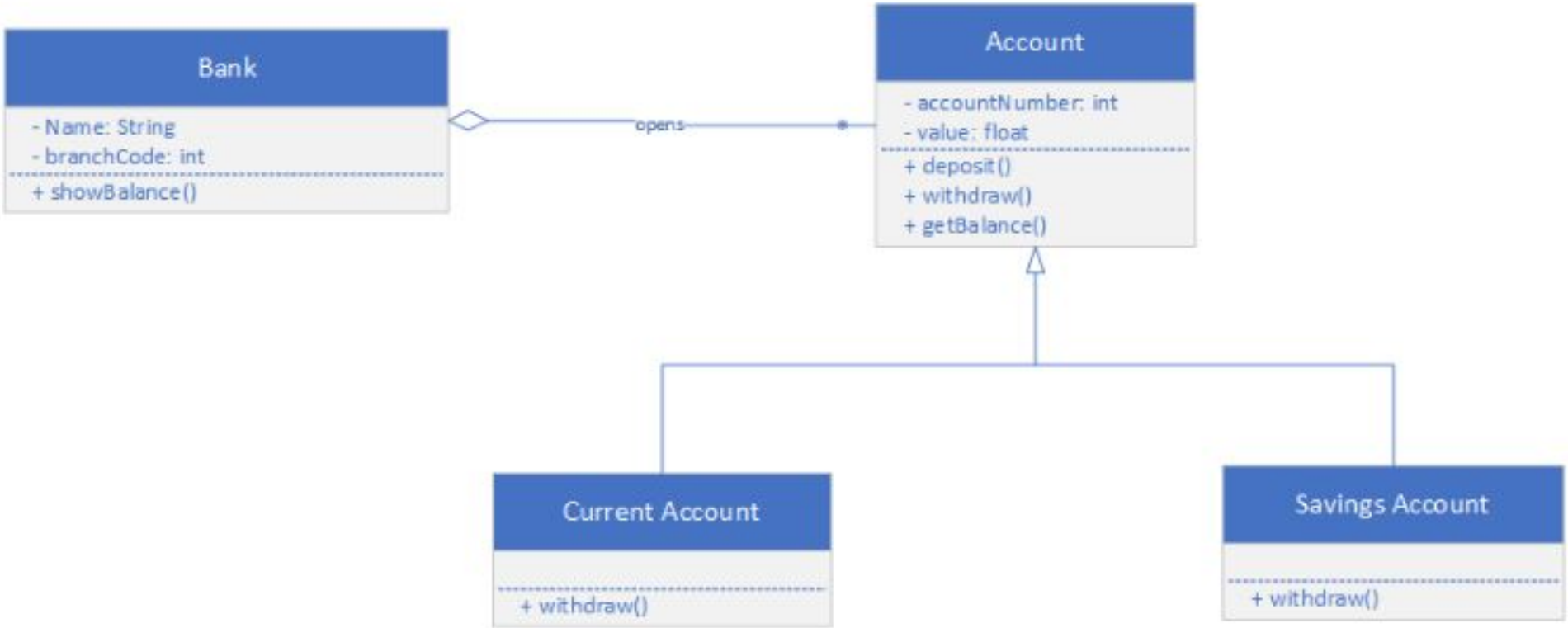
Important things to consider:

`You have to use standard UML notations for objects, classes, and their associations.`

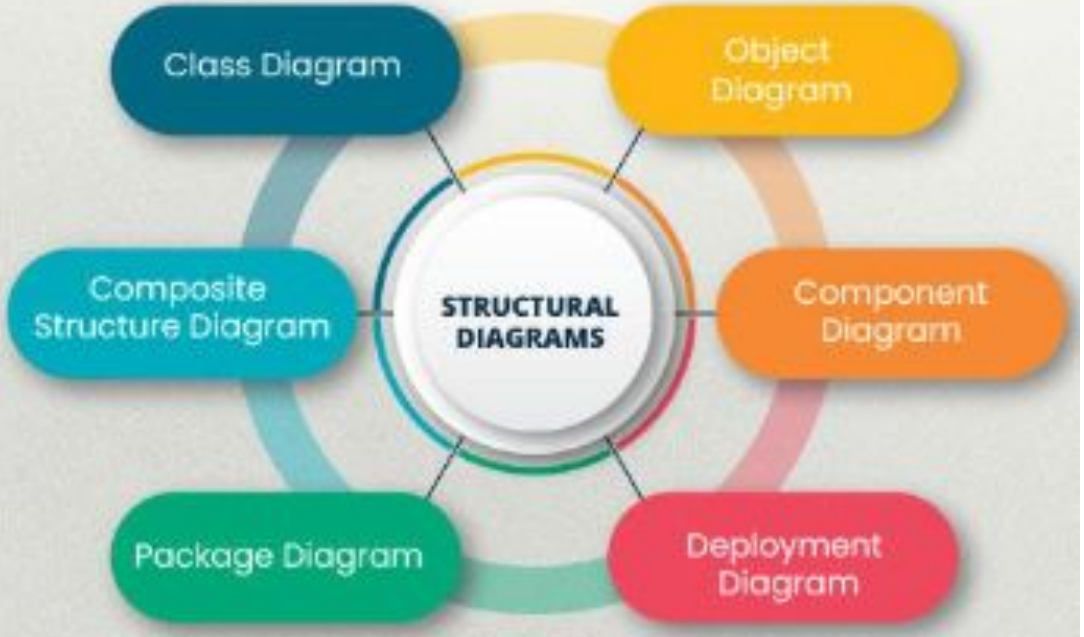
Note:

Besides problem statement; objects, properties, and functions of a system can also be extracted from domain knowledge. You have to consider all OOP concepts like abstraction, encapsulation, inheritance, association, generalization, and specialization wherever applicable.

Solution


Bank
- Name: String
- branchCode: int
+ showBalance()
opens
Account
- accountNumber: int
- value: float
+ deposit()
+ withdraw()
+ getBalance()
Current Account
+ withdraw()
Savings Account
+ withdraw()

Structural Diagrams



Structural UML diagrams

Structural UML diagrams illustrate the organization of a system by depicting its components, such as classes, objects, and packages. They represent the elements that make up the system and the relationships between them.

Types of Structural UML diagrams

1. Class Diagram

- Class diagrams are the main building blocks of every object-oriented method.
- The class diagram can be used to show the classes, relationships, interface, association, and collaboration. UML is standardized in class diagrams.
- Since classes are the building block of an application that is based on OOPs, the class diagram has an appropriate structure to represent the classes, inheritance, relationships, and everything that OOPs have in their context.

2. Package Diagram

- A package diagram is a type of Unified Modeling Language (UML) diagram mainly used to represent the organization and the structure of a system in the form of packages.
- A package is used as a container to organize the elements present in the system into a more manageable unit.
- It is very useful to represent the system's architecture and design as a cohesive unit and concise manner.

3. Object Diagram

- An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them.
- Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behavior of the system at a particular instant.
- Object diagrams are vital to portray and understand functional requirements of a system. In other words, “An object diagram in the Unified Modeling Language (UML), is a diagram that shows a complete or partial view of the structure of a modelled system at a specific time.

4. Component Diagram

- Component diagrams are used to represent how the physical components in a system have been organized. We use them for modelling implementation details.
- Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development.
- Component Diagrams become essential to use when we design and build complex systems.
- Interfaces are used by components of the system to communicate with each other.

5. Composite structure diagram

- A composite structure diagram represents relationship between parts and their configuration which determine how the classifier (class, a component, or a deployment node) behaves.
- They represent internal structure of a structured classifier making the use of parts, ports, and connectors.
- They are similar to class diagrams except they represent individual parts in detail as compared to the entire class.

6. Deployment diagram

- Deployment Diagrams are used to represent system hardware and its software.It tells us what hardware components exist and what software components run on them.
- We illustrate system architecture as distribution of software artifacts over distributed targets.
- They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations.

Benefits of using Structural UML diagrams

Structural diagrams are like visual guides that help everyone on a software team see how the different parts of a program fit together. They make planning, building, and fixing things much easier.

- **Clear Picture for Everyone:**
    - Structural UML diagrams are like visual maps that show how different parts of a computer program or system are connected.
    - It's a clear picture that helps everyone understand how things fit together.
- **Helps Design and Plan:**
    - These diagrams are like blueprints for building a house.
    - They help the people designing the software see how different pieces will work together.
- **Easy to Explain and Share:**
    - Structural diagrams make it easy to explain how a computer program works and share ideas with the team.
- **Keeps Things Organized:**
    - These diagrams help organize the different parts of a computer program.
    - It keeps everything in order, making it easier to understand and update.
- **Saves Time and Fixes Problems Early:**
    - Before building something big, like a bridge, engineers use drawings to catch any problems early. Structural diagrams do the same for computer programs.
    - They help find and fix issues before spending lots of time writing the actual code.

Challenges faced in developing Structural UML diagrams

- **Abstraction Complexity:**
  - Abstracting complex systems into simplified diagrams can be challenging.
  - Representing complicated relationships and dependencies between classes or components may lead to reduced clarity.
- **Maintaining Consistency with Code:**
  - Ensuring that UML diagrams remain consistent with the actual codebase over time can be challenging.
  - Code changes may not always be reflected in the diagrams, leading to discrepancies.
- **Choosing the Right Level of Detail:**
  - Striking the right balance between providing sufficient detail and avoiding overwhelming complexity is a common challenge.
  - Including too much detail can hinder readability, while too little may lead to ambiguity.
- **Expressing Dynamic Aspects in a Static Diagram:**
  - Structural diagrams are inherently static, making it challenging to represent dynamic aspects such as behavior or state transitions.
  - Capturing dynamic behavior may require additional diagrams like behavior diagrams.
- **Handling Change Management:**
  - Structural diagrams may need frequent updates to reflect changes in the system. Managing and communicating these changes effectively can be challenging, especially in large projects.
- **Balancing Simplicity and Completeness:**
  - Striking a balance between simplicity and completeness is crucial.
  - Over-simplified diagrams may lack necessary details, while overly detailed diagrams may become overwhelming.
- **Ensuring Consistent Notation:**
  - Inconsistent use of notation and symbols across diagrams can lead to confusion. Different team members may interpret symbols differently, affecting the overall understanding.
- **Tooling and Accessibility:**
  - Different team members might be familiar with different tools, and not all team members may have easy access to specialized UML tools.

Best Practices for developing Structural UML diagrams

- **Keep It Simple:**
  - Avoid unnecessary complexity, and only include information that is essential for understanding the structure of the system.
- **Use Consistent Naming Conventions:**
  - Employ consistent and meaningful names for classes, attributes, methods, and other elements.
  - Follow a naming convention that is widely accepted and understood within your development team.
- **Organize Elements Logically:**
  - Arrange elements in a logical and organized manner. Group related classes or components, and use visual alignment to convey relationships.
- **Use Proper Notation:**
  - Clearly represent associations, generalizations, aggregations, compositions, and other relationships using the appropriate symbols.
- **Avoid Overcrowding:**
  - Avoid overcrowding the diagram with too many elements.
  - If the diagram becomes too dense, consider breaking it into smaller, more focused diagrams or using packages to group related classes.
- **Keep Diagrams Updated:**
  - Regularly update diagrams to reflect the current state of the system. Outdated diagrams can lead to confusion and misinterpretation.
- **Validate Against Code:**
  - Periodically validate your UML diagrams against the actual code to ensure consistency.
  - Tools that support code generation or synchronization can be helpful in maintaining alignment between code and diagrams.
- **Review and Validate:**
  - Conduct regular reviews of your diagrams with team members to ensure accuracy and understanding. Validate that the diagrams effectively represent the system's structure.