WEEK 1

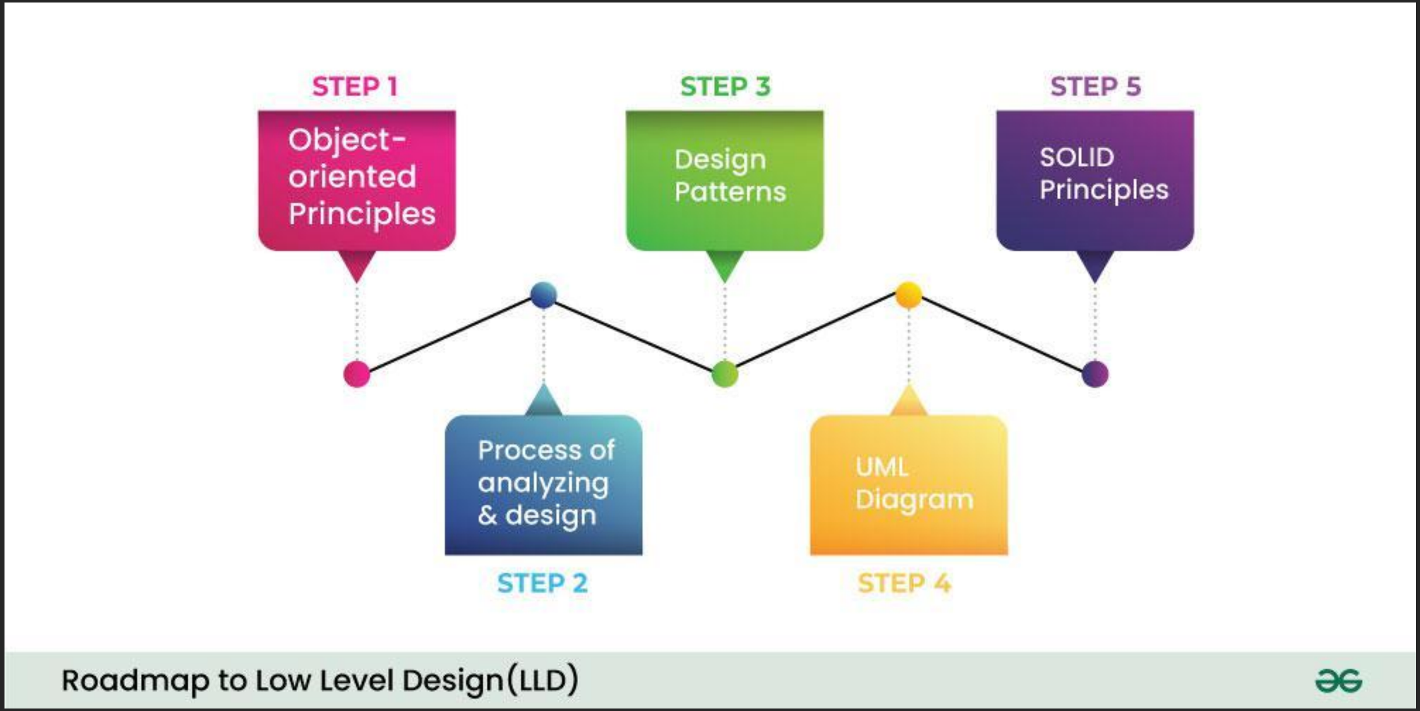
SYSTEM DESIGN

LOW_LEVEL DESIGN

Introduction to System Design

What is LLD?

LLD, or Low-Level Design, is a phase in the software development process where detailed system components and their interactions are specified, along with their implementation. It involves converting the high-level design into a more detailed blueprint, addressing specific algorithms, data structures, and interfaces. LLD serves as a guide for developers during coding, which ensures the accurate and efficient implementation of the system's functionality.


STEP 1
Object-oriented Principles
STEP 2
Process of analyzing & design
STEP 3
Design Patterns
STEP 4
UML Diagram
STEP 5
SOLID Principles
Roadmap to Low Level Design(LLD)

Main focus of LLD

- *Scalability* : The system should be designed so that it can handle increased load or complexity with minimal performance degradation or redesign.

- *Maintainability* : The code should be easy to understand, debug, and modify, allowing for efficient updates and fixes over time.

- *Reusability* (should not be tightly coupled) : Components should be modular and loosely coupled so they can be reused across different parts of the system without heavy dependencies.

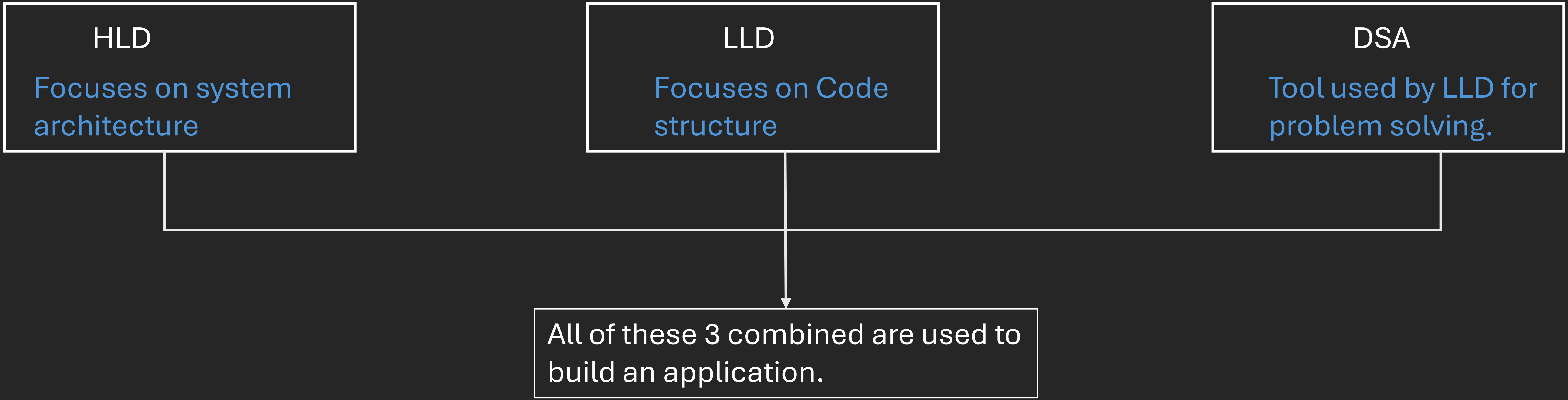
Modular:
Meaning: Code is organized into independent, self-contained units (modules), each handling a specific responsibility.
Why it matters: You can develop, test, and debug modules in isolation, making the system easier to manage.

Loosely Coupled:
Meaning: Components depend minimally on each other, so changes in one module have little or no impact on others.
Why it matters: Improves flexibility and makes code more adaptable and reusable.

What is not LLD?

HLD: High Level Design

- Tech Stack (Java/ spring boot)
- Data Base (SQL, noSQL)
- Server Scaling
- Cost Optimization


HLD
Focuses on system architecture
LLD
Focuses on Code structure
DSA
Tool used by LLD for problem solving.
All of these 3 combined are used to build an application.

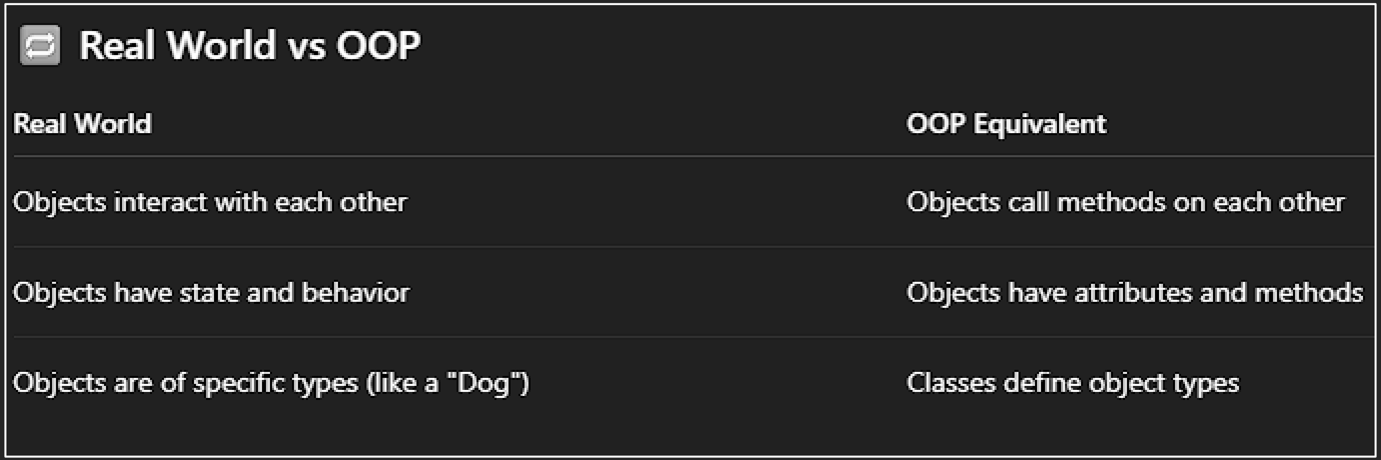
If DSA is the brain of an application, LLD is the skeleton.

OOPS: Abstraction, Encapsulation, Inheritance& Polymorphism

History of Programming:

1. Machine language (0/1) : Prone to error, tedious, not scalable
2. Assembly level language(e.g. MOV A 61H) : Prone to error, low scalability(because code is dependent on hardware), tedious.
3. Procedural Programming : It introduced Functions, loops, blocks(if-else, switch). It could do everything but at a small level. C is language based on procedural programming. But it was not able to solve complex problems.
4. OOPS : It introduced **real-world modelling**, introduced **Data Security**, and is highly **Scalable** and **Reusable.**

In real world, we have objects that interact with each other. We apply the same logic in OOPS.



Real World vs OOP

Real World	OOP Equivalent
Objects interact with each other	Objects call methods on each other
Objects have state and behavior	Objects have attributes and methods
Objects are of specific types (like a "Dog")	Classes define object types

REAL-WORLD MODELING IN OOPS

Let us understand this with an example of CAR. An object can have Characteristics and Behaviour(functions)

Characteristics	Behaviour(Functions)
- Engine	- Start()
- Brand	- Stop()
- Model	- gearshift()
- Wheels	- Accelerate()
- So on…	- Brake()

Now, if we have to convert this CAR in terms of programming, then we introduce the concept of Class.

```
Class Car{
    //code
};
```

This acts as a blueprint and we can create as many number of objects by calling the class.

In procedural programming, to represent a CAR, we would declare separate variables for each characteristic (like color, speed, model) and write individual functions for its behaviors. If we wanted to create multiple cars, we'd have to repeat this process for each one, making the code tedious, less readable, and hard to scale or reuse.

In contrast, OOP allows us to define a CAR blueprint (class) just once. We can then create multiple CAR objects from it, each with its own properties and behaviors. This makes the code more scalable, reusable, and better suited for real-world modeling.

PILLARS OF OOPS

We now know how to represent an object in programming, but it is not sufficient because in real-life real world objects can perform actions so we also need to learn how to map the actions in programming as well.

Pillars of OOPS:

- Abstraction: Hides complex implementation details and shows only essential features. Helps focus on what an object does rather than how it does it. Abstraction is about hiding complexity.

- Encapsulation: Bundles data and methods into a single unit (class) and restricts direct access. Protects the object's state by using **access modifiers** (like private, public, protected). Encapsulation is about protecting data.
  - Public: Accessible from anywhere in the program.
  - Private: Accessible only within the same class.
  - Protected: Accessible within the same class and by subclasses (even in different packages in some languages).

    Protected allows access to child classes, while private does not.

- Inheritance: Allows a class to acquire properties and behaviors of another class. Promotes code reuse and establishes a parent-child relationship.

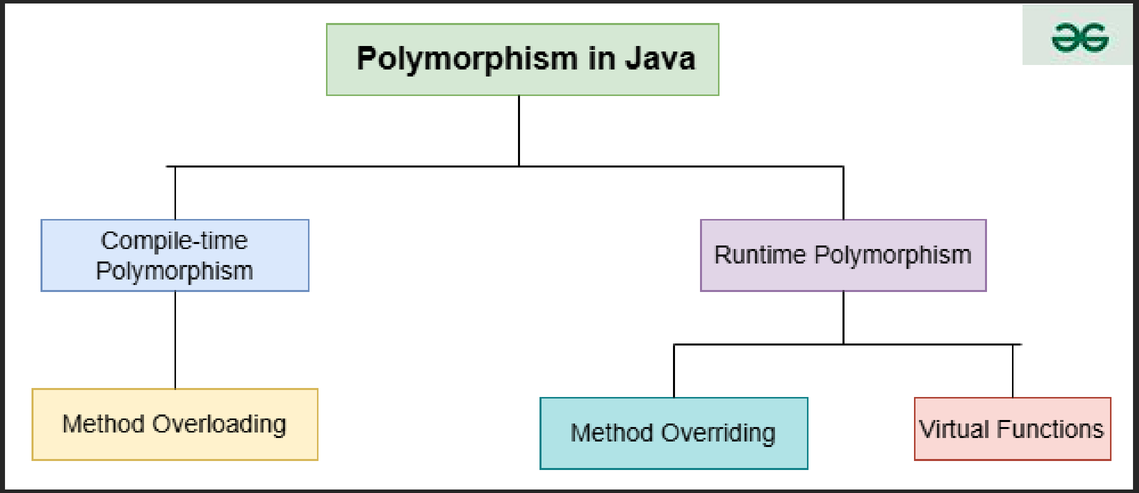
- Polymorphism: Allows objects to be treated as instances of their parent class. Enables one interface to perform different actions (method overriding/overloading). Polymorphism allows the same method or object to behave differently based on the context, specially on the project's actual runtime class. Poly (many) and morph (forms), this means one entity can take many forms.

ACCESS MODIFIERS			
	Within Class	From child class	Outside Class
Public	✓	✓	✓
Protected	✓	✓	X
Private	✓	X	X

Types of Polymorphism

1. Dynamic Polymorphism
(Runtime polymorphism)

2. Static Polymorphism
(Compile time polymorphism)



Dynamic Polymorphism (Runtime polymorphism)

It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

Method overriding is when a subclass provides its own version of a method that is already defined in its parent class, using the same name, return type, and parameters.

Easy explanation: Imagine you have a general class called Animal with a method makeSound() that says, "Some sound". Now, you create a subclass called Dog that also has a makeSound() method, but it says "Bark". When you call makeSound() on a Dog object, it will override the parent version and use the one in Dog, so it says "Bark" instead of "Some sound".

Why use it? To allow custom behavior in subclasses while keeping the same method name and structure as the parent class — this makes code more flexible and easier to manage (polymorphism).

Rules (common across most languages like Java, C++, etc.):

- The method must have the same name and parameters.
- The method must be inherited from the parent class.
- The subclass version replaces the parent's version when called on a subclass object.

Static Polymorphism (Compile time polymorphism)	This happens when multiple methods in the same class have the same name but different parameters which is known as method overloading. In **method overloading**, functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

Advantages of Polymorphism:

- Code reuse: Same method name, different behaviour.
- Flexible code: Easy to change or extend behaviour.
- Clean syntax: Improves code readability.
- Easy to extend: Add new classes without changing old code.
- Dynamic behaviour: Decides method at runtime.

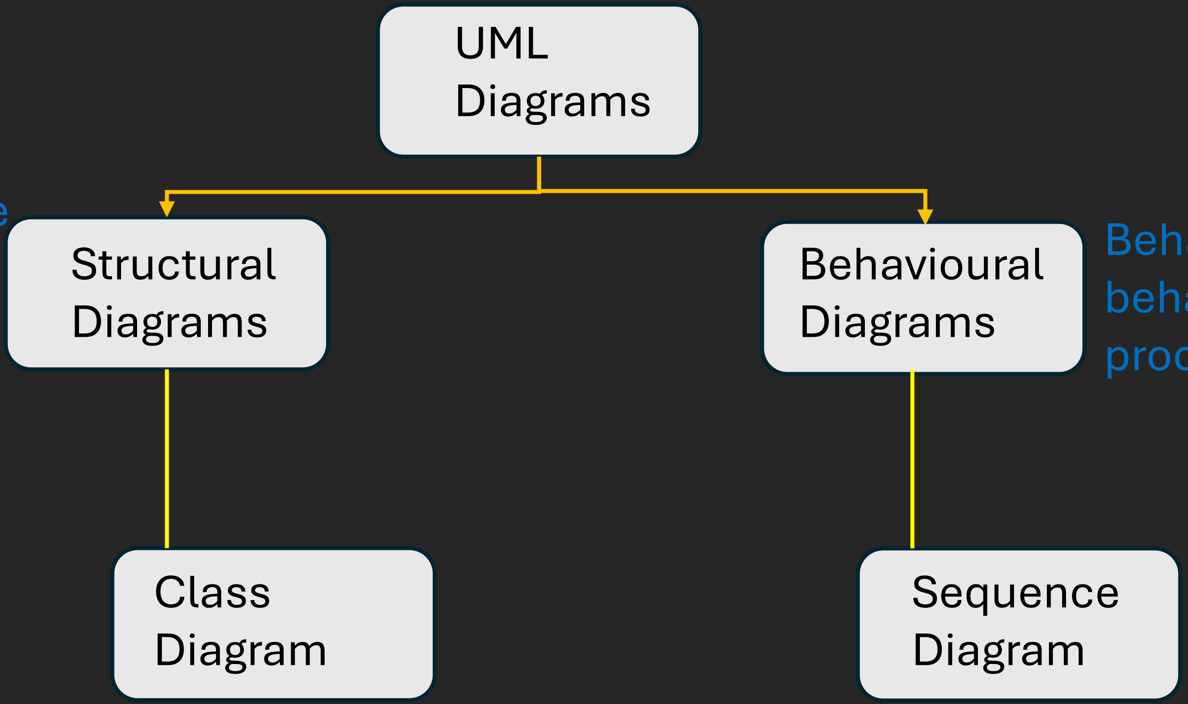
Disadvantages of Polymorphism:

- Hard to debug: Tracing method calls is tricky.
- Performance hit: Slight slowdown due to runtime decisions.
- Tight inheritance: Can cause rigid code structure.
- Confusing for beginners: Hard to grasp at first.

Unified Modelling Language (UML) Diagrams

We need UML (Unified Modeling Language) to visually represent and communicate complex system designs, facilitating better understanding and collaboration among stakeholders.

Structural diagrams show the **static** parts of a system (like classes, objects, and relationships).



Behavioral diagrams show the **dynamic** behavior or interactions over time (like processes, use cases, and workflows).

We have in total 14 diagrams, 7 for structural and 7 for dynamic. But for LLD we only need 2.

Class Diagram

A class diagram in UML is a structural diagram that represents the blueprint of a system by showing its classes, attributes, methods, and relationships.
Each class is drawn as a rectangle divided into three parts:

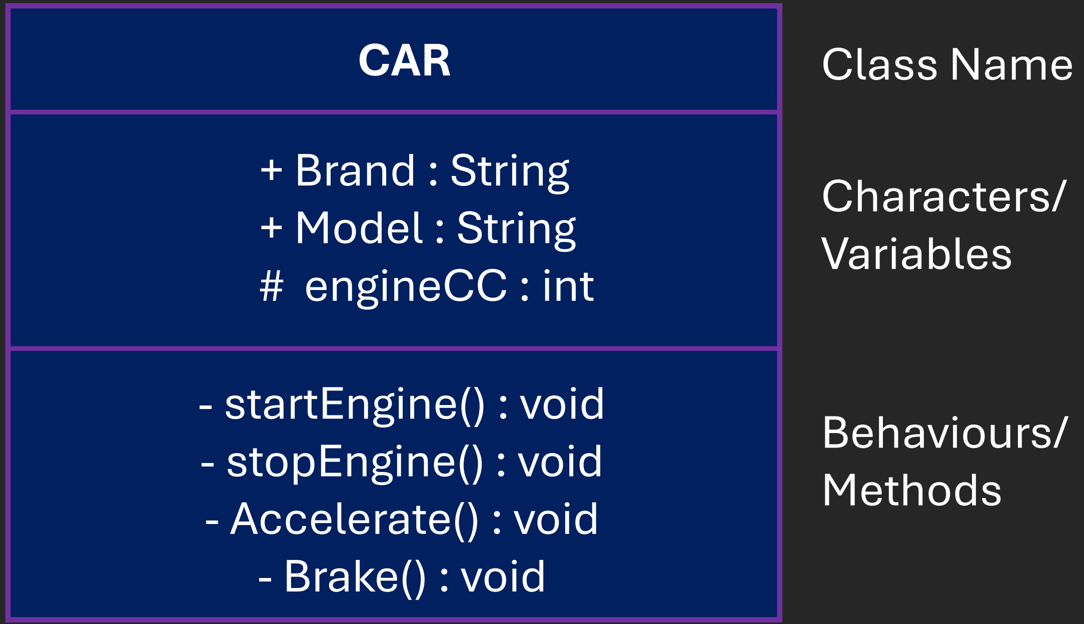
- the class name
- its attributes (variables) with the data-type
- and its methods (functions) with the data-type

It also illustrates how classes are connected — through associations (normal links), inheritance (generalization), dependencies, and aggregations/compositions (part-whole relationships).

Class diagrams help in understanding the static structure of a system and are widely used in object-oriented design to plan, communicate, and document software architecture.
It is composed of 2 major components: Class structure and association/ connection.

Representation of access-modifiers in Class Diagram:

1. Public → plus symbol (+)
2. Protected → hash symbol (#)
3. Private → minus symbol (-)



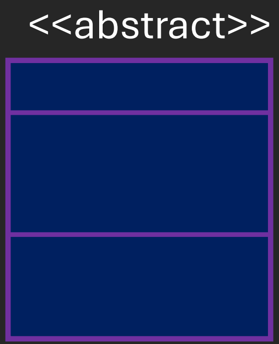
```
CAR
---------------------------
+ Brand : String
+ Model : String
#  engineCC : int
---------------------------
- startEngine() : void
- stopEngine() : void
- Accelerate() : void
- Brake() : void
```

We have 2 types of classes:

1. Abstract class

A class that cannot be instantiated and often contains abstract methods meant to be implemented by subclasses, i.e. (A class you can't create objects from; it's like a base idea meant to be built on). Example: Imagine a class called Vehicle with a method start(), but it doesn't say how to start — it just says "every vehicle must have a start method." You can't make a Vehicle object directly, but Car or Bike can inherit it and define their own version of start().

Denoted with <<abstract>> on top of the class.



2. Concrete Class

A class that can be instantiated and provides full implementation of all its methods, i.e. (A class you can create objects from; it has complete working code)

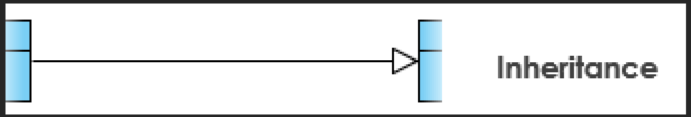
For concrete class we don't need to write anything on top.

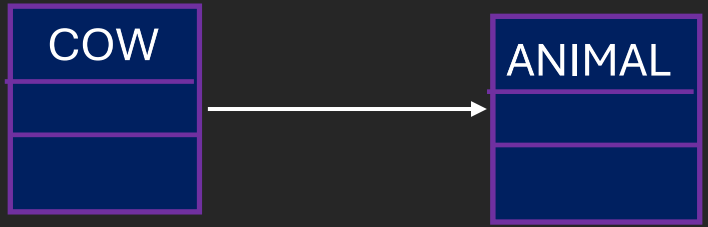
Representing Class Association in UML

Associations are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes. Associations are typically named using a verb or verb phrase which reflects the real-world problem domain.

There are 2 types of Associations:
Class Association , Object Association

1. **Class Associations:**
   - Inheritance: Inheritance, also called generalization in UML, is a "is-a" relationship between a parent (super) class and a child (sub) class. It means the child class inherits attributes and methods from the parent class and can also add or override its own. Shown with a solid line and a hollow arrowhead pointing from the child class to the parent class.



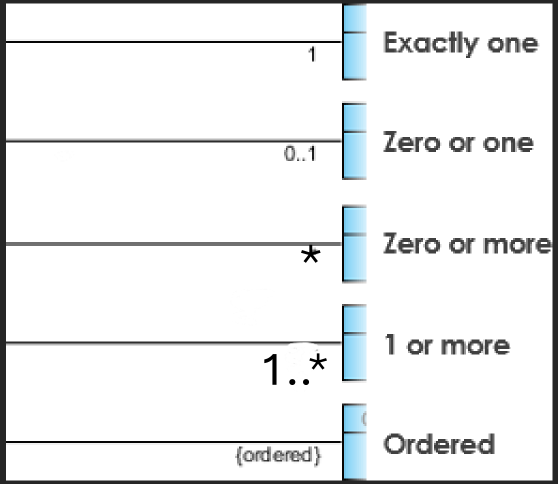


2. Object Associations:

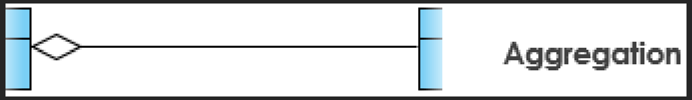
2.A Simple associations: A basic relationship between two classes where one class uses or interacts with another. It shows that objects of one class are connected to objects of another, without implying ownership. A Student uses a Library Card. UML symbol: Solid line between classes, **optionally** with multiplicity (e.g., 1..*).

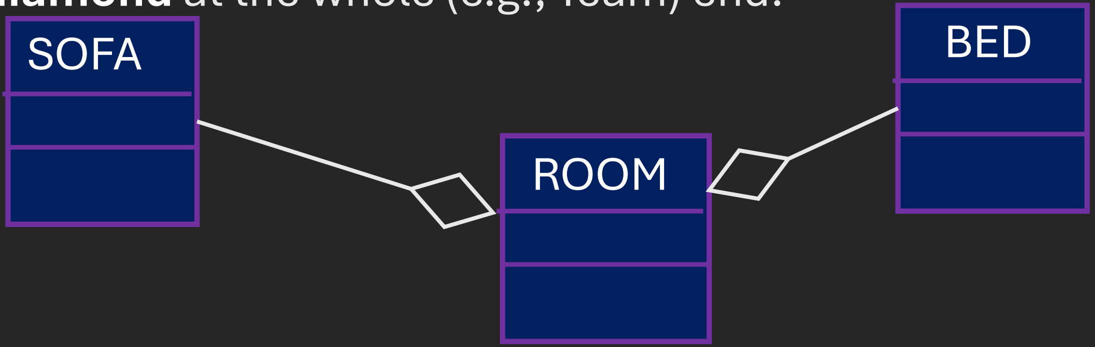
Cardinality is expressed in terms of:

- one to one
- one to many
- many to many

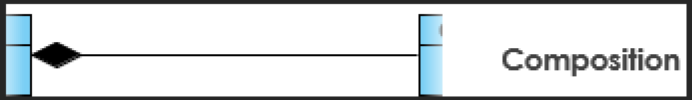


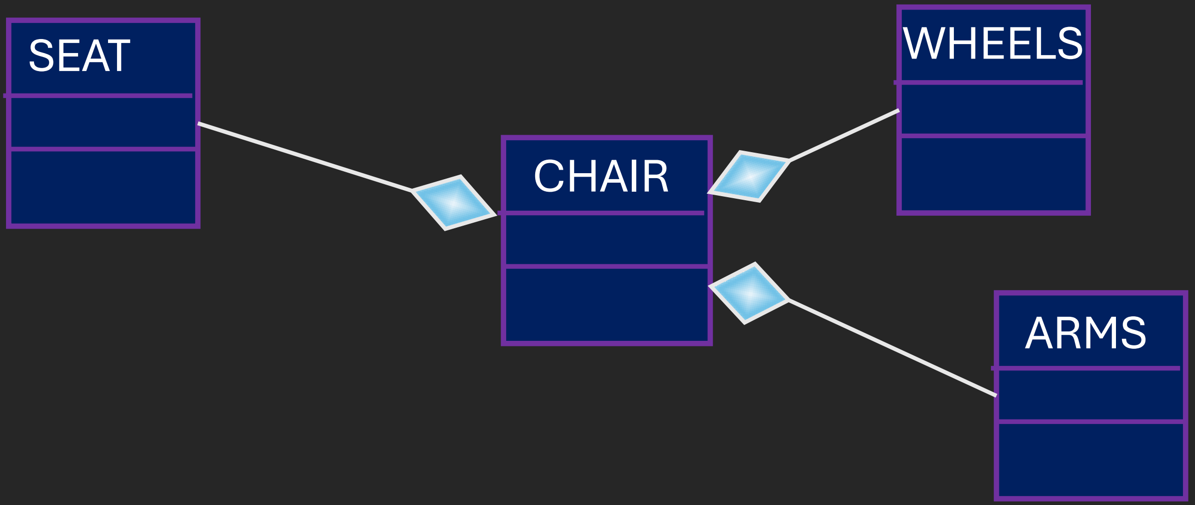
2.B Aggregation: A "has-a" relationship where one class is made up of other classes, but the parts can exist independently of the whole. It shows a weak ownership. 🧱 Example: A Team has Players, but players can exist without the team. UML symbol: Solid line with a **hollow diamond** at the whole (e.g., Team) end.





2.C Composition: A strong "has-a" relationship where one class is made up of parts that cannot exist independently of the whole. It shows strong ownership and lifecycle dependency. 🧱 Example: A House has Rooms, and if the house is destroyed, the rooms are too. Solid line with a filled (black) diamond at the whole (e.g., House) end.





Representing composition in code

```
Class A{
   method1();
}

Class B{
 A * a;
 B() { a = new A(); };  //constructor
 method2();
}
```

```
main() {
      B * b = new B();
      b → method2();
      b → a  → method1();
}
```

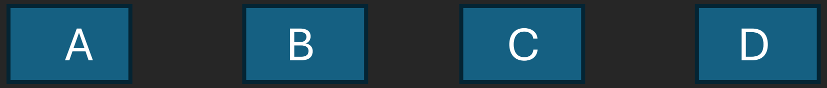
Sequence Diagram

A sequence diagram in UML is a behavioral diagram that shows how objects interact over time by displaying the order of messages exchanged between them. It focuses on the sequence of events in a specific scenario, often representing a use case.

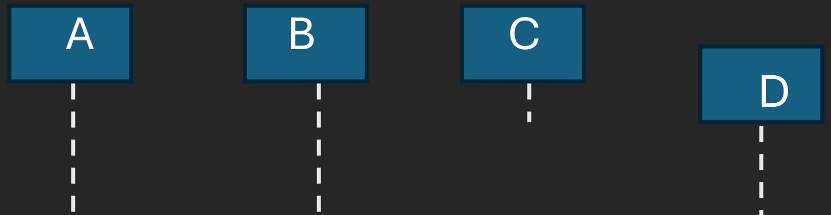
Objects (or classes) are shown as boxes at the top, with lifelines extending downward, and messages (method calls, returns) are represented by arrows between lifelines.

It helps visualize the flow of control, understand object collaboration, and detect timing or logic issues in a system. Sequence diagrams are especially useful in modeling real-time operations, interactions, or workflows.

1. Object
In the UML, an object in a sequence diagram is drawn as a rectangle containing the name of the object.

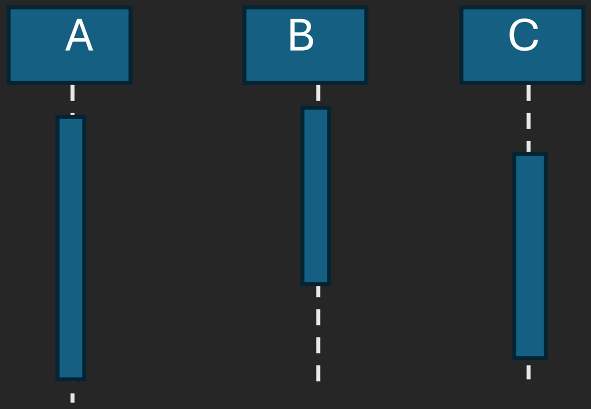


2. Lifeline
A lifeline is represented by dashed vertical line drawn below each object. These indicate the existence of the object. Length of lifeline denotes duration of the object.



3. Activation Bar

An activation bar (also called a focus of control) in a sequence diagram shows the period an object is actively performing an action or executing a process. It's drawn as a thin vertical rectangle on the lifeline. It starts when a message is received and ends when the task is done.

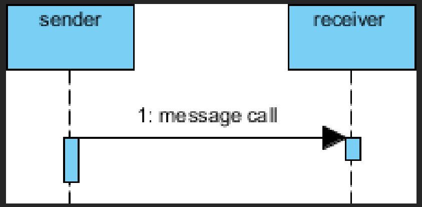


4. Messages

A message in a sequence diagram represents communication between objects, typically a method call or signal. It's shown as an arrow from the sender's lifeline to the receivers. The arrow type (solid, dashed) indicates if it's a call, return, or asynchronous message. Types of messages:

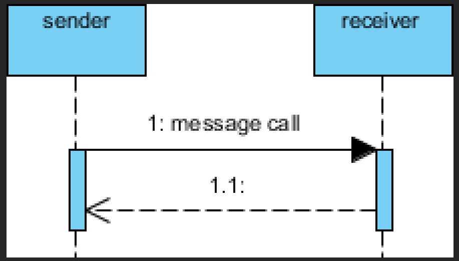
1. Synchronous

Shown as a solid line with a **filled arrowhead**. It is a regular message call used for normal communication between sender and receiver.



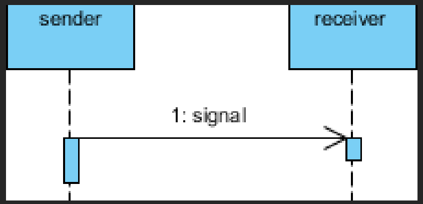
2. Return

A return message uses a **dashed line with an open arrowhead**.



3. Asynchronous

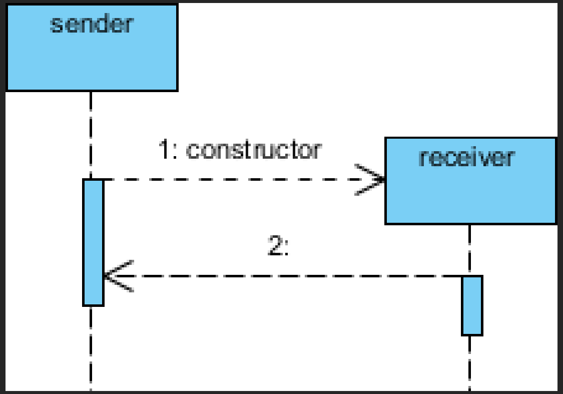
An asynchronous message has a solid line with an **open arrowhead**. A signal is an asynchronous message that has no reply.



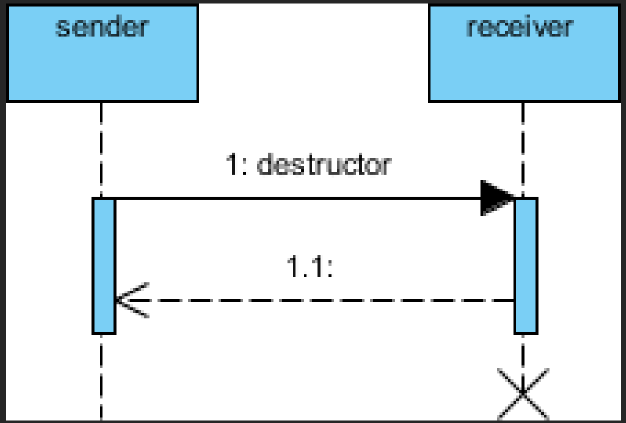
5. Creation and Destruction Messages

Participants do not necessarily live for the entire duration of a sequence diagram's interaction. Participants can be created and destroyed according to the messages that are being passed.

A **constructor message** creates its receiver. The sender that already exist at the start of the interaction are placed at the top of the diagram. Targets that are created during the interaction by a constructor call are automatically placed further down the diagram.

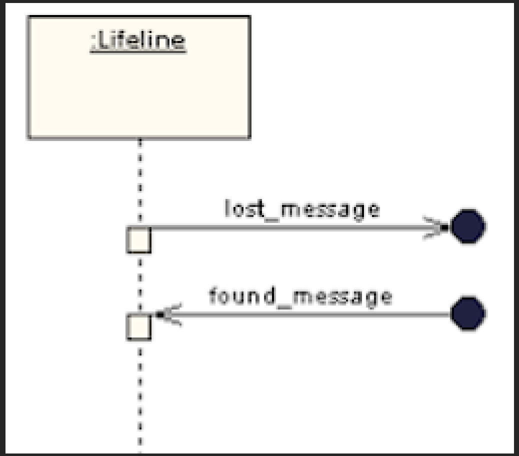


A **destructor message** destroys its receiver. There are other ways to indicate that a target is destroyed during an interaction. Only when a target's destruction is set to 'after destructor' do you have to use a destructor.



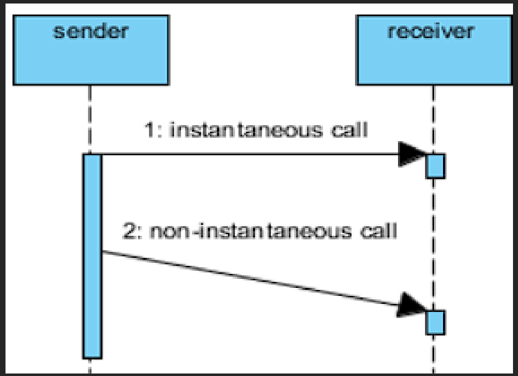
6. Lost and Found Messages

A **lost message** is a message sent from an object, but the receiver is unknown or not shown in the diagram. It's used when you know a message is sent, but not where it ends up.
A **found message** is a message that appears to come from an unknown source and is received by an object. It's used when the sender is outside the system or not shown in the diagram.



7. Non instantaneous message

Messages are often considered to be instantaneous, thus, the time it takes to arrive at the receiver is negligible. The messages are drawn as a horizontal arrow. To indicate that it takes a certain while before the receiver actually receives a message, a slanted arrow is used.

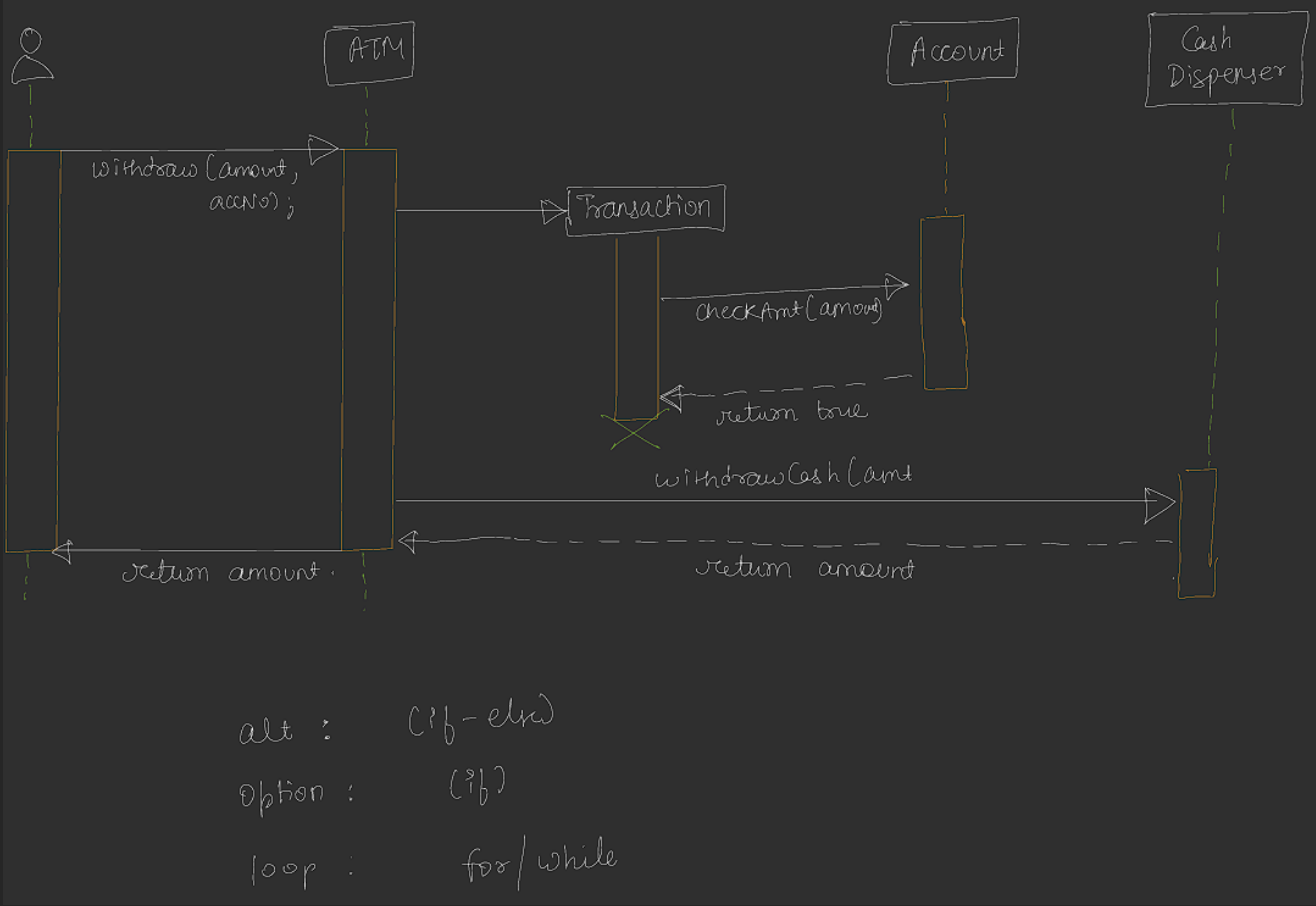


Steps to Make a Sequence Diagram

1. Identify actors and objects involved in the use case (e.g., ATM, Account, Transaction).
2. Define a use case scenario, like "Withdraw Cash", and list the message flow between the objects step-by-step.
3. Start constructing Sequence Diagram.

- **alt**: Represents if-else conditional logic.
- **opt**: Represents an optional single condition.
- **loop**: Repeats a block multiple times.

This sequence diagram models the cash withdrawal process from an ATM. The user requests withdrawal via the ATM, which delegates to the Transaction system. The Transaction checks the balance with the Account and, if valid, commands the Cash Dispenser to release the amount. Messages and returns show the interaction flow, and the X marks the end of the transaction's activation. Alt/loop/option notes hint at how conditions or repetitions can be modeled if needed.



SOLID Design Principles

While building an application, we might faces issues related to Code Maintainability, Readability and Bugs.
Solid design principles provide guidelines to build software systems that are scalable, maintainable, testable, and easy to understand and evolve.

The SOLID principles are five essential guidelines that enhance software design, making code more maintainable and scalable. They include

1. ***Single Responsibility Principle (SRP)***
2. ***Open/Closed Principle (OCP)***
3. ***Liskov Substitution Principle (LSP)***
4. ***Interface Segregation Principle (ISP)***
5. ***Dependency Inversion Principle (DIP)***

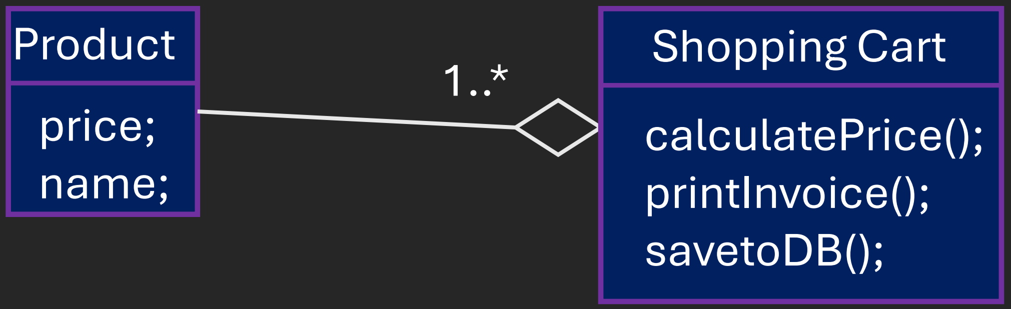
S
Single Responsibility
O
Open/closed
L
Liskov substitution
I
Interface segregation
D
Dependency inversion

Why is it a big deal?

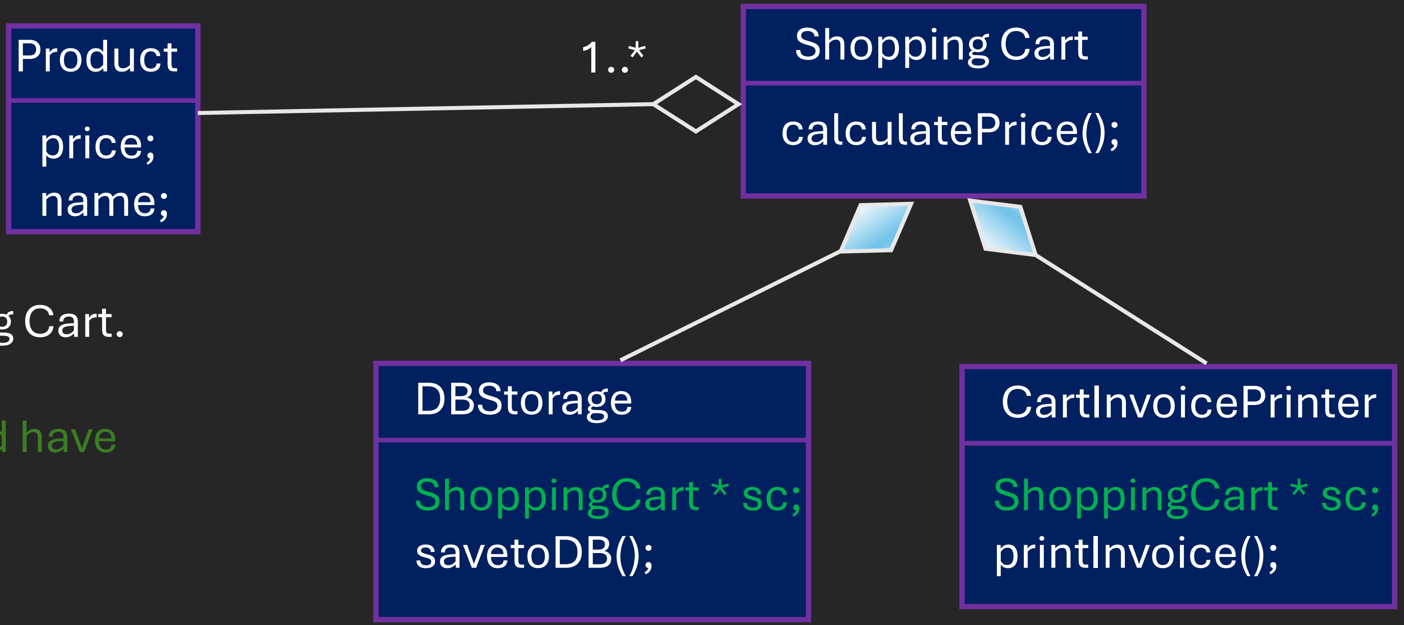
- Maintainability :Easy to modify and extend code.
- Scalability: Supports growth with minimal rework.
- Reusability & Modularity: Use components across multiple systems.
- Team Collaboration: Enables smooth teamwork and onboarding.
- Testability & Reliability: Simplifies testing and reduces bugs.
- Encapsulation of Complexity: Hides internal details from users.
- Foundation for Best Practices: Encourages clean, proven design approaches.

S: Single Responsibility Principle

A class should have only one reason to change.
A class should do only one thing.



In this example, we can see that Shopping Cart is breaking the SRP because it is taking multiple responsibilities.
So, in future, if we want to update savetoDB() or printInvoice() we will have to make changes in this class. So, there are multiple reasons to change the class.
So, it breaks SRP !
To solve this, we use Composition.

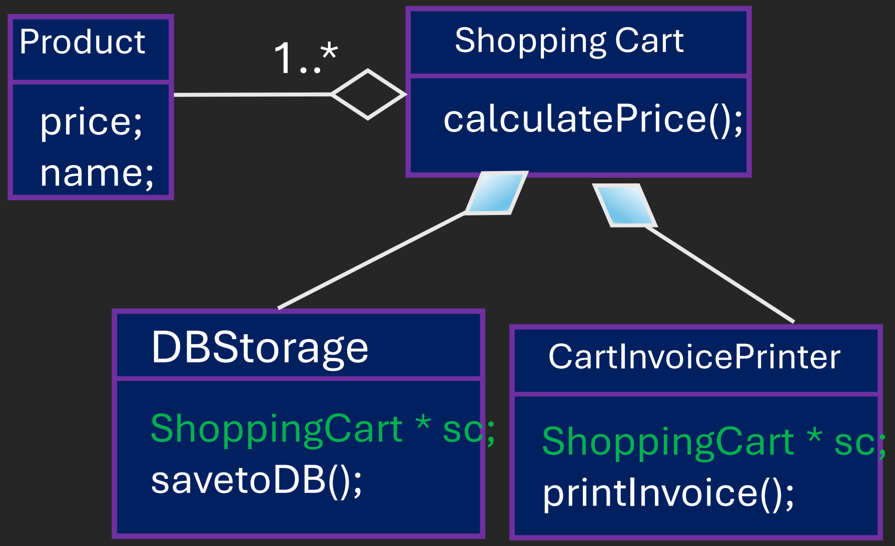


Both DBStorage and CartInvoicePrinter "has-a" relationship with Shopping Cart and can't exist without Shopping Cart.
ShoppingCart * sc; stores the reference of Shopping Cart.

Now if we have to change a logic in future, we would have to make changes to a single class.
Now it follows SRP !

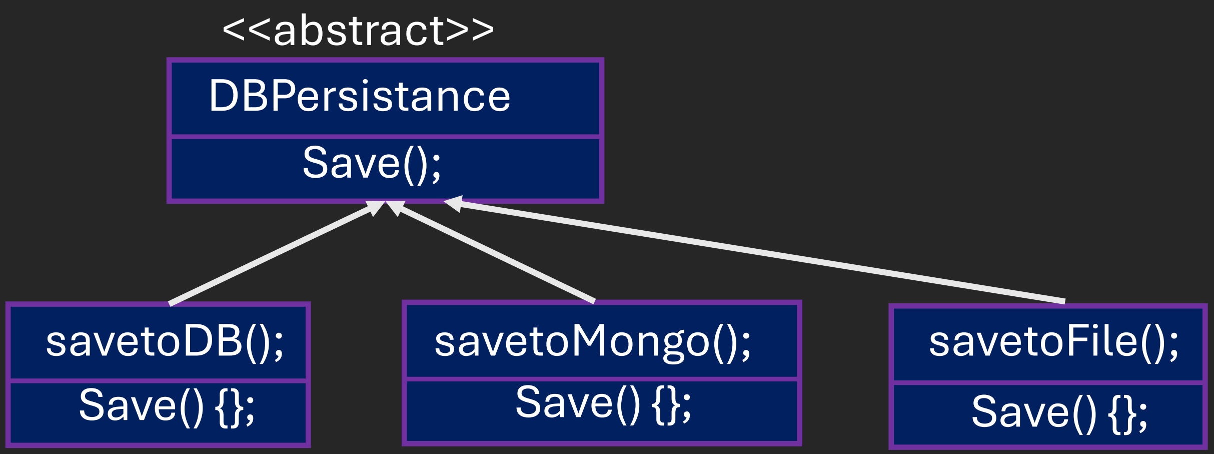
O: Open / Close Principle

A class should be open for extension (like adding new feature) but close for modification (like changing the current code).



Let's, understand this. In the previous example, let's say we want to store in DB but also want to store in Mongo and want to store the same in a File as well.
But we can't insert those functions in DBStorage.
It will break the OCP !
Because, to add those methods, we made changes to existing class. It should have had been closed for modification.

To handle this without breaking OCP we need Abstraction, Inheritance and Polymorphism. Also, our current DBStorage class is concrete. We will have to make it abstract.
In the 2nd method, we have solved this issue.
First, we made the class as abstract initialized a method 'save' and created 3 classes which **inherit** from DBPersistance and can declare their own save methods.
And DBPersistance will have the same 'has-a' Relationship with shopping cart.
Now it satisfies OCP !



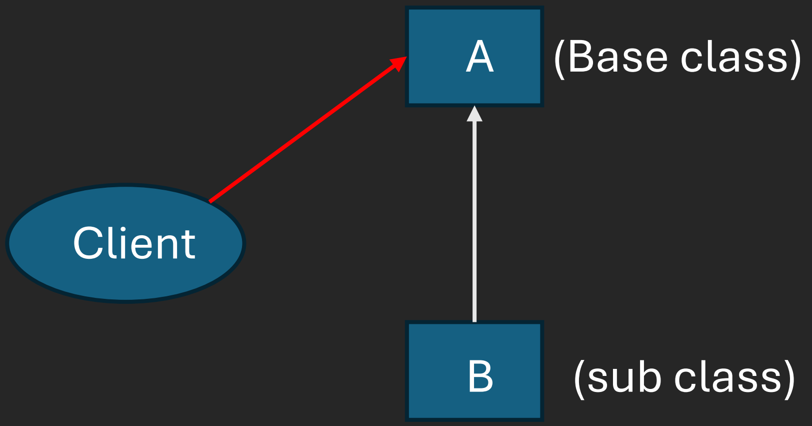
L: Liskov Substitution Principle

Sub-classes should be substitutable for their base class.
It means if a client uses a subclass in place of a base class, the program must continue to work **correctly and consistently**.

Let's, understand this with an example. Here B is a child class and inherits from A.
Suppose class A has methods : m1(), m2(), m3();
B can inherit these methods. Also, B has its own methods : m4(), m5();

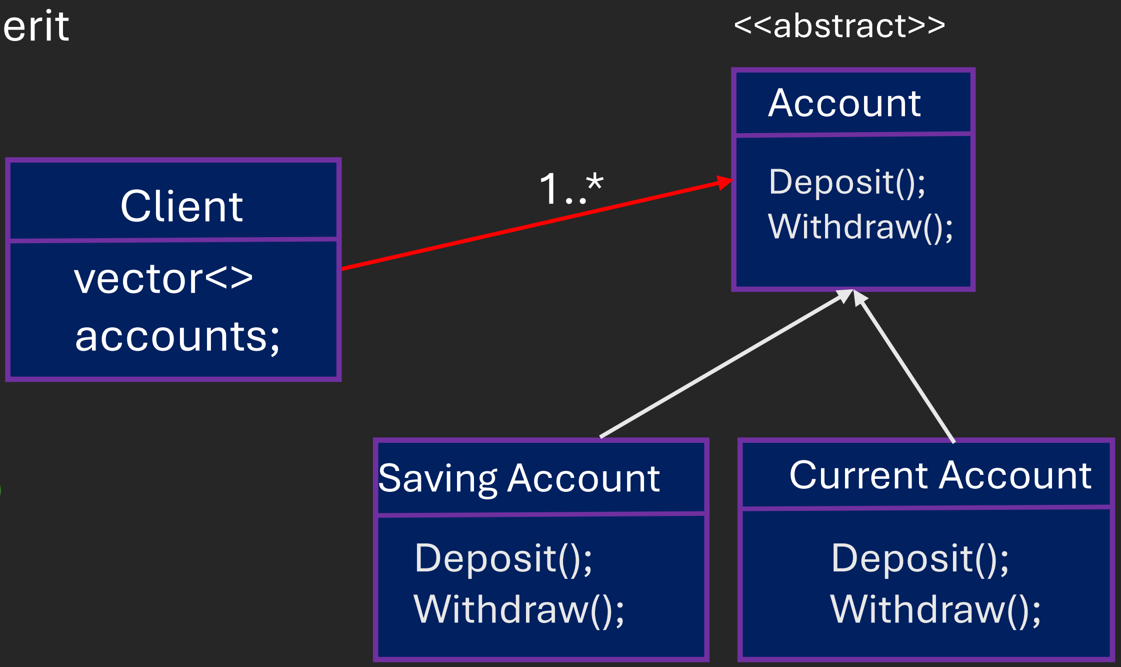
Now, if a client expects ‘A’ let’s say using a method ‘randomMethod’.

```
randomMethod (A * a) {
    a → m1();
    a → m2();
    a → m3();
}
```



Now, if we pass (A * b) instead of (A * a) in the parameter of the same function, it should still be able to call methods m1(), m2, m3().

In this example, account is an abstract class, and its children can inherit from the base class.
Also, client has 1 or more accounts.



Now let's say, bank has opened new type of account, 'fixed deposit' which gives more interest, but it doesn't allow for withdraw(), but since it is a type of account, and it can also be considered as a child class for account, and manually define the withdraw() function inside the child-class so that it doesn't follow the parent.

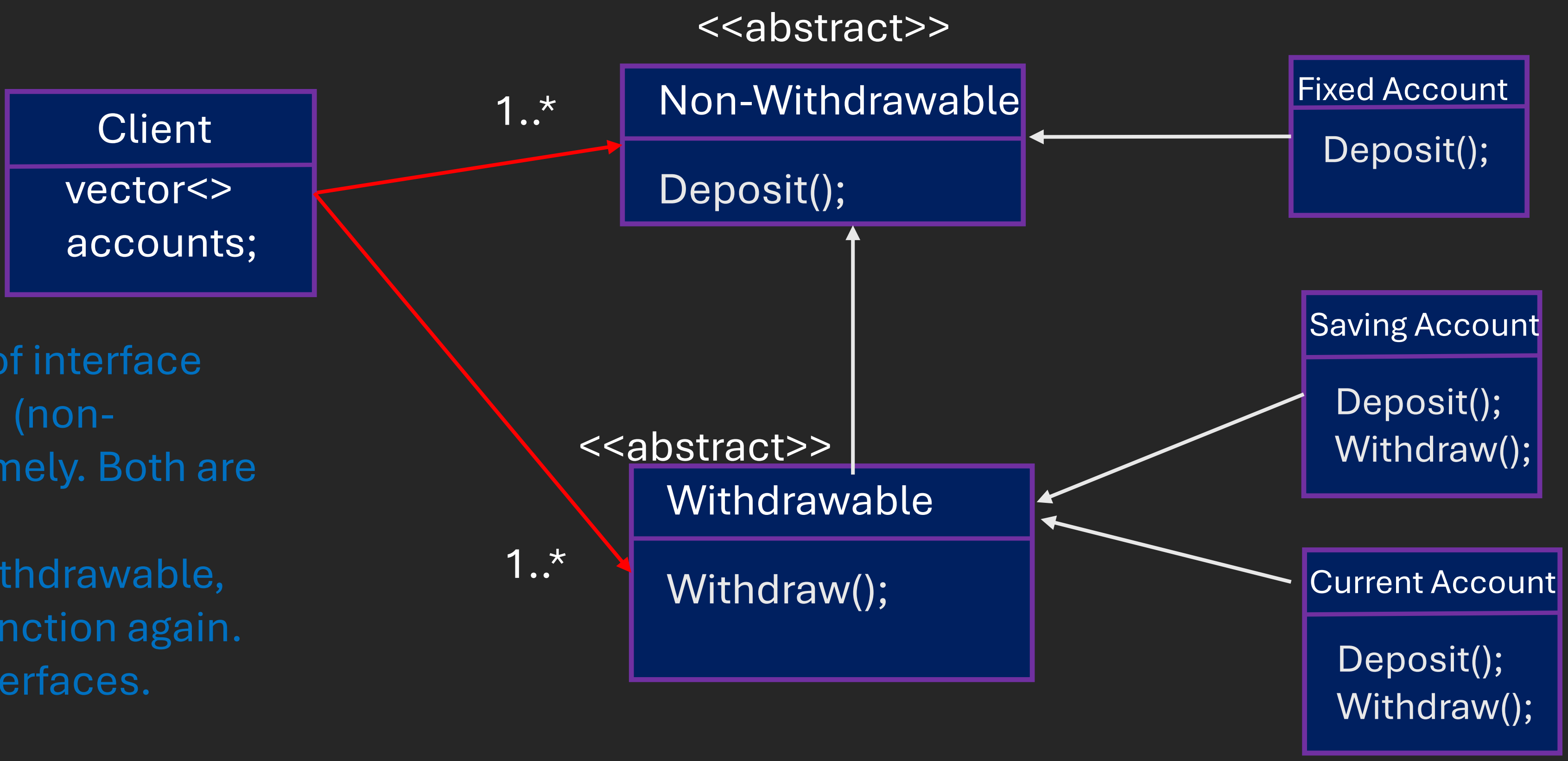
But the problem is, client is only aware of base –class 'Accounts' and that it can perform both deposit() and withdraw() but he won't be able to perform the withdraw for this 'fixed deposit'.
Here, Liskov substitution principle breaks ! To solve this, we have 2 ways. First method is naïve; second method is optimal.

- In the naïve method, what we can do is, add a condition, if the account == "fixed deposit" then don't call withdraw(); else call both methods.
- But, due to this what happens is, now Client gets tightly coupled with Account and its children, but client should only be able to call Interface (base-class). While this might seem to solve the issue temporarily, it introduces a serious design flaw: the client now needs to know about specific subclasses of the Account class.
- This violates the principle of polymorphism and creates tight coupling between the client and the subclasses. **The client is no longer working with just the abstraction (Account) but is directly dependent on concrete implementations.**
- **As a result, Client was modified; hence it also broke Open Close Principle !**

Why this matters:

- The client should only rely on the base class interface.
- It shouldn't care which subclass it's dealing with.
- If the client must write conditions for specific subclasses, it leads to fragile code, poor extensibility, and higher maintenance.

SECOND SOLUTION



Instead of considering 'fixed deposit' of interface (account), now we create 2 interfaces (non-withdrawable) and (withdrawable) namely. Both are abstract classes.
Since, withdrawable is child of non-withdrawable, we don't need to mention Deposit() function again.
Client can communicate with both interfaces.

- Every relationship type belongs to "is-a" relationship.
- But relationship between client and interfaces is of type "has-a" relationship.