Lecture 1 Notes :

1. What is Low-Level Design (LLD)?

Definition: Designing the internal structure (“skeleton”) of an application by identifying classes/objects, their relationships, data flows, and how DSA solutions plug into this structure.

- **DSA**: Solves isolated problems (e.g. “find shortest path in an array/graph”) using algorithms like binary search, quicksort, Dijkstra’s, heaps, etc.
- **LLD**: Determines which objects exist in the system and how they interact, then applies DSA inside that structure.

2. Illustrative Story: Two Approaches to Building “**QuickRide**”

- Scenario: Build a ride-booking app (“QuickRide”) like Uber/Ola.

Anurag’s DSA-First Approach:

1. **Problem decomposition:**
- Map city intersections to graph nodes, roads to edges.
- Use Dijkstra’s algorithm to compute shortest route.
- Use a min-heap (priority queue) to match riders to closest drivers.

2. **Gaps:**
- No identification of classes/entities (User, Rider, Location, Notification, Payment).
- Omits data security (masking phone numbers).
- Missing integration points (notifications, payment gateways).
- No consideration for scaling to millions of users.

Maurya’s LLD-First Approach:

1. **Entity identification:**
- Objects: User, Rider, Location, NotificationService, PaymentGateway, etc.

2. **Define relationships & interactions:**
- How User and Rider connect via Location.
- How NotificationService and PaymentGateway integrate.

3. **Non-functional concerns:**
- Data security: Protect personal info.
- Scalability: Architect code to handle millions of users without performance collapse.

4. **Then apply DSA:**
- Embed shortest-path algorithm and driver-matching heap inside this object-oriented framework.

3. Core LLD Principles & Focus Areas

1. **Scalability**
- Handle large user volumes easily.
- Code structure should allow rapid, low-effort expansion (adding servers, features).

2. **Maintainability**
- New features shouldn't break existing ones.
- Code should be easy to debug and locate bugs.

3. **Reusability**
- Write loosely coupled, "plug-and-play" modules (e.g. generic notification or matching algorithms usable across apps like Zomato, Swiggy, Amazon delivery).

4. What LLD Is Not (vs. HLD)

- High-Level Design (HLD) focuses on system architecture, not code structure:
- Tech stack: Choice of languages/frameworks (e.g. Java Spring Boot).
- Database: SQL vs. NoSQL vs. hybrid.
- Server scaling & deployment: Autoscaling, load balancers, cost optimization on AWS/GCP.
- Cost considerations: Minimizing cloud/server expenses per load.

5. Summary & Takeaways

- DSA = Brain of an application: algorithms solve specific tasks.
- LLD = Skeleton: object models, class diagrams, code organization, and where algorithms plug in.
- HLD = Architecture: system-wide infrastructure, tech stack, databases, servers.

6. Key line to remember

"If DSA is the brain, LLD is the skeleton of your application."

Lecture 2 : OOPS (Abstraction & Encapsulation)

1. Why Did We Move Beyond Procedural Programming?

1.1 Early Languages

1. **Machine Language (Binary)**
    - Direct CPU instructions in 0s & 1s.
    - **Drawbacks**:
        - Extremely error-prone: one bit flip breaks the program.
        - Tedious to write and maintain.
        - No abstraction—every detail is manual.
2. **Assembly Language**
    - Introduced mnemonics (e.g. `MOV A, 61h`) instead of raw bits.
    - **Still hardware-tied**: code changes with CPU architecture.
    - **Scalability**: remains very limited for large systems.

1.2 Procedural (Structured) Programming

- **Features Introduced**:
    - **Functions** for code reuse
    - **Control structures**: `if-else`, `switch`, `for`/`while` loops
    - **Blocks** for grouping statements
- **Advantages**:
    - Improved readability over assembly.
    - Modularized small to mid-size programs.
- **Limitations**:
    - **Poor real-world mapping**: Difficult to model complex entities (e.g. a ride-booking system's users, drivers, payments).
    - **Data security gaps**: No built-in access control—everything is globally visible.
    - **Reusability & scalability**: Functions alone can't enforce consistent interfaces or safe extension.

2. Entering Object-Oriented Programming

- **Core Idea**: Model your application as **interacting objects** mirroring real-world entities.
- **Benefits**:
    - **Natural mapping** of domain concepts (User, Car, Ride).
    - **Secure data encapsulation**—control who can read or modify state.
    - **Code reuse** via inheritance and interfaces.
    - **Scalability** through loosely coupled modules.

3. Modeling Real-World Entities in Code

3.1 Objects, Classes, & Instances

- **Object**: A real-world “thing” with attributes and behaviors.
- **Class**: Blueprint defining those attributes (fields) and behaviors (methods).
- **Instance**: Concrete object in memory, created via the class.

4. Deep Dive: Pillar 1 – Abstraction

Definition:

> **Abstraction** hides unnecessary implementation details from the client and exposes only what is essential to use an object’s functionality.

4.1. Real-World Analogies

- **Driving a Car**
  - **What you do**: Insert key, press pedals, turn steering wheel.
  - **What you don’t need to know**: How the fuel-injection system works, how the transmission synchronizes gears, how the engine control unit computes ignition timing.
  - **Abstraction in action**: The car provides a simple interface (“start,” “accelerate,” “brake”) and conceals all mechanical complexity under the hood.
- **Using a TV or Laptop**
  - **What you do**: Press buttons on a remote or click icons.
  - **What you don’t need to know**: How the display panel refreshes, how the CPU executes machine code, how the OS schedules tasks.
  - **Abstraction in action**: A graphical interface abstracts away thousands of low-level operations.

4.2. Language-Level Abstraction

- **Control Structures as Abstraction**
  - Keywords like `if`, `for`, `while` let you express complex branching and loops without writing jump addresses or machine instructions.
  - The compiler translates these high-level constructs into assembly or machine code behind the scenes.

5. Code-Based Abstraction: Abstract Classes & Interfaces

5.1 Abstract Class Example (C++)

```
// Abstract interface for any Car type
class Car {
public:
  // Pure virtual methods - no implementation here
  virtual void startEngine() = 0;
  virtual void shiftGear(int newGear) = 0;
  virtual void accelerate() = 0;
  virtual void brake() = 0;
  virtual ~Car() {}
};
```

- **Key Points**
  - The `Car` class declares *what* operations must exist but hides *how* they work.
  - No code for `startEngine()`, etc., lives here—only signatures.
  - Clients use `Car*` pointers without needing concrete details.

5.2 Concrete Subclass Example

// See Code section for full Code example

6. Benefits of Abstraction

1. Simplified Interfaces: Clients focus on *what* an object does, not *how* it does it.
2. Ease of Maintenance: Internal changes (e.g., switching from a V6 to an electric motor) don't affect client code.
3. Code Reuse: Multiple concrete classes can implement the same abstract interface (e.g., `SportsCar`, `SUV`, `ElectricCar`).
4. Reduced Complexity: Large systems are easier to reason about when broken into abstract modules.

7. Deep Dive: Pillar 2 – Encapsulation

Definition:

> Encapsulation bundles an object's data (its state) and the methods that operate on that data into a single unit, and controls access to its inner workings.

7.1. Two Facets of Encapsulation

1. **Logical Grouping**
   - Data (fields) and behaviors (methods) that belong together live in the same "capsule" (class).

    - Example: A `Car` class encapsulates `engineOn`, `currentSpeed`, `shiftGear()`, `accelerate()`, etc., in one place.
2. **Data Security**
    - Restrict direct external access to sensitive fields to prevent invalid or unsafe operations.
    - Example: You can *read* the car's odometer but cannot directly set it back to zero.

7.2. Real-World Analogies

- **Medicine Capsule**
    - The capsule holds both the medicine (data) and its protective shell (access control).
    - You swallow the capsule without exposing its contents directly.
- **Car Odometer**
    - You can view the mileage but *cannot* tamper with it via the dashboard interface.

// See Code section for full Code example

7.3 Access Modifiers in C++

- **`public`**: Members are accessible everywhere.
- **`private`**: Members accessible only within the class itself.
- **`protected`**: Accessible in the class and its subclasses (for inheritance scenarios).

7.4. Getters & Setters with Validation

- **Purpose**: Allow controlled mutation with checks, rather than exposing fields blindly.

7.5. Encapsulation Benefits

1. **Robustness**: Prevents accidental or malicious misuse of internal state.
2. **Maintainability**: Internal changes (e.g., adding new constraints) do not ripple into client code.
3. **Clear Contracts**: Clients interact only via well-defined methods (the public API).
4. **Modularity**: Code is organized into self-contained units, easing testing and reuse.

Lecture 3 : Inheritance and Polymorphism

1. Inheritance

1.1 What is Inheritance?

- Real-world objects are often related in parent-child relationships.
- Example: Object A (Parent) and Object B (Child) share properties.
- In programming, this relationship is mimicked using **Inheritance**.

1.2 Real-Life Example: Car Hierarchy

- **Parent Class:** Car (Generic)
  - Common attributes:
    - Brand
    - Model
    - IsEngineOn
    - CurrentSpeed
  - Common behaviors:
    - startEngine()
    - stopEngine()
    - accelerate()
    - brake()
- **Child Classes:**
  - **ManualCar** (inherits Car)
    - Specific attribute: CurrentGear

        - Specific behavior: shiftGear()
    - **ElectricCar** (inherits Car)
        - Specific attribute: BatteryPercentage
        - Specific behavior: chargeBattery()

1.3 C++ Syntax

```
class ManualCar : public Car { ... };
class ElectricCar : public Car { ... };
```

- `public` inheritance maintains access specifiers.
- `private` and `protected` alter accessibility.

1.4 Access Specifiers in Inheritance

- **public:**
    - Public members stay public.
    - Protected members stay protected.
- **protected:**
    - Public and protected members become protected.
- **private:**
    - All inherited members become private.
- **Private members** of parent class are **never inherited**.

// See code section for full code example.

2. Polymorphism

2.1 What is Polymorphism?

- Derived from: **"Poly" (many) + "Morph" (forms)** = many forms.
- **One stimulus → different responses** based on object/situation.

2.2 Two Real-Life Scenarios:

- **Scenario 1:**
  - Different animals (Duck, Human, Tiger) all have a `run()` behavior.
  - Each performs it differently.
- **Scenario 2:**
  - Same human `run()`s differently based on context (tired vs chased).

2.3 Types of Polymorphism in Programming:

- **Static Polymorphism** – Compile-time
  - Achieved via **Method Overloading**
- **Dynamic Polymorphism** – Runtime
  - Achieved via **Method Overriding**

3. Static Polymorphism (Method Overloading)

- Same method name, different parameter lists.
- Overloaded method is resolved at **compile time**.

Example:

```
class ManualCar {
  void accelerate();                // no parameter
  void accelerate(int speed);     // with parameter
};
```

- Allows the same behavior to adapt based on passed arguments.

Rules:

- Method name: Same
- Return type: Can be same or different (but not used for overloading)
- Parameters:
    - Vary in number **or** type

4. Dynamic Polymorphism (Method Overriding)

- Same method signature is redefined in child classes.
- Achieved using **virtual functions** in C++.
- Resolved at **runtime**.

Example:

```
class Car {
  virtual void accelerate() = 0; // Abstract
};

class ManualCar : public Car {
  void accelerate() override; // Manual-specific logic
};

class ElectricCar : public Car {
  void accelerate() override; // Electric-specific logic
};
```

5. Combined Use of OOP Pillars

- Final code demonstrates:

// See code section for full code example.

- **Abstraction** (Hiding implementation details)
- **Encapsulation** (Private/protected members)
- **Inheritance** (Manual/Electric inherit Car)
- **Polymorphism** (Method overriding & overloading)

Additional Concepts:

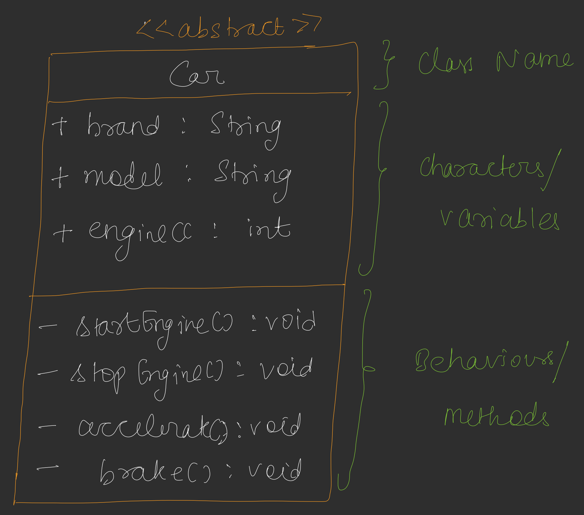
- **Protected:**
  - Inaccessible outside class, but accessible in child class.
- **Operator Overloading (Homework):**
  - Concept asked as homework: What is operator overloading?
  - Why is it available in C++ but not in Java/Python?

Conclusion & Practice

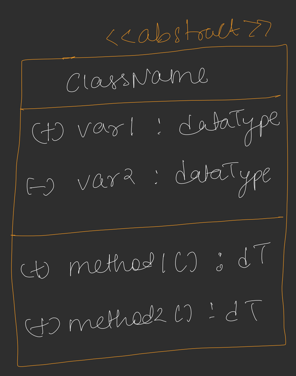
- Understanding OOPs is best done via **real-world relatable examples**.
- Practice suggestion: Modify/add features to existing car classes.
- **Homework:**
  1. Define **Operator Overloading**.
  2. Why is it not supported in Java/Python?

Class Diagram

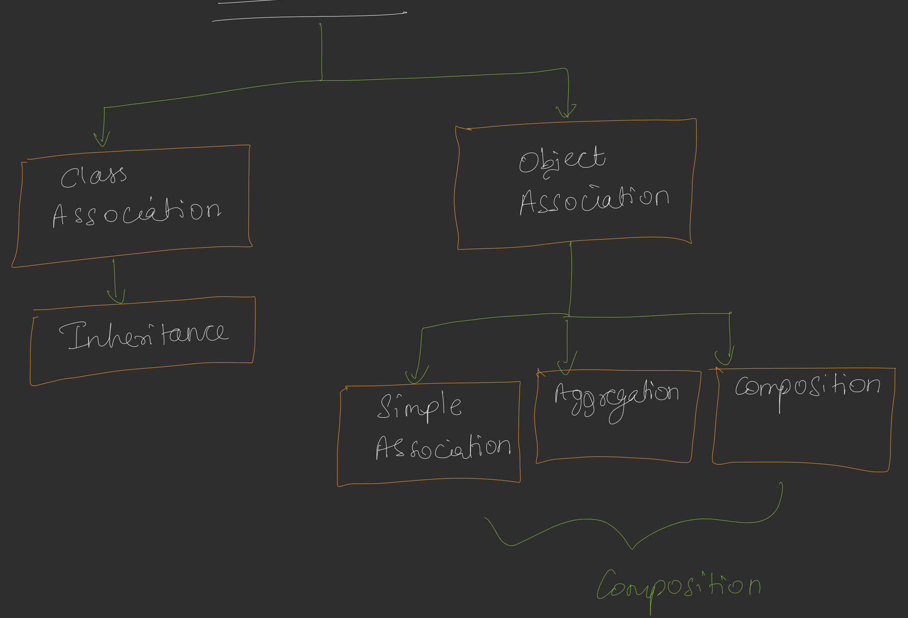
```
class Car {
    String brand;
    String model;
    int engineCC;

    startEngine();
    stopEngine();
    acclerate();
    brake();
}
```



Generic Class



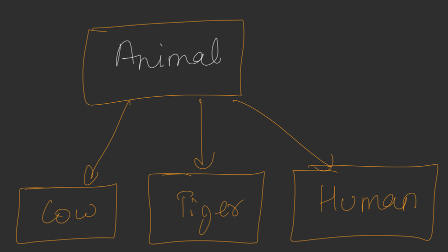
Associations

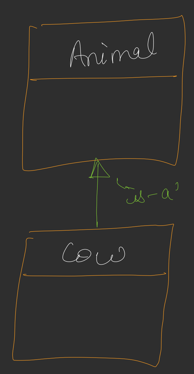


Inheritance

- (is-a) relationship
- Denoted by (——▷)

```
class A {
    method1();
}

class B : public A {
    method2();
}

main() {
    B* b = new B();
        b->method1();
        b->method2();
}
```

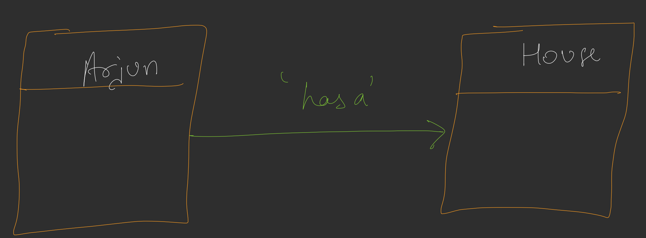




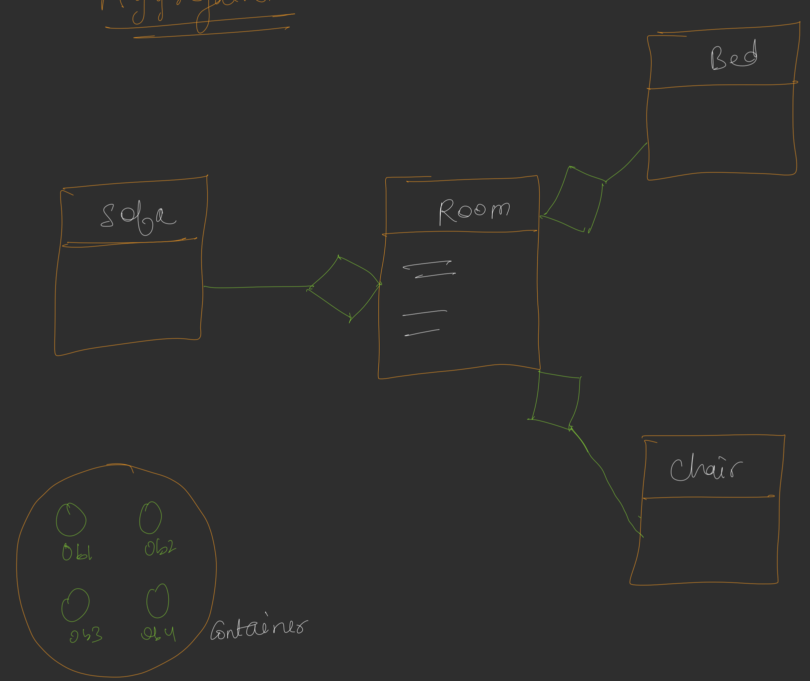
Composition

- (has-a) relationship
- Denoted by (——>)

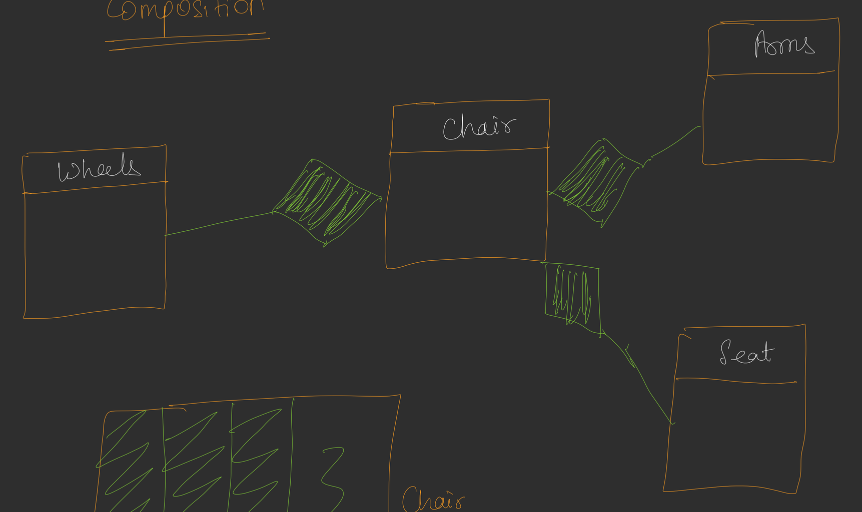
Simple Association

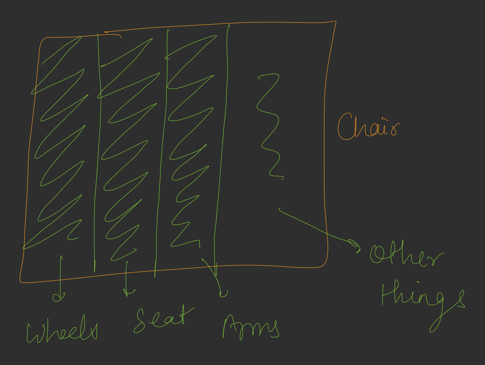


Aggregation



Composition

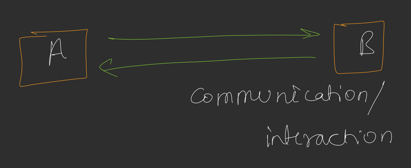




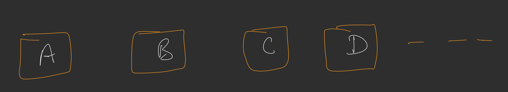
Composition in Code

```
Class A {
    method1();
}

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

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

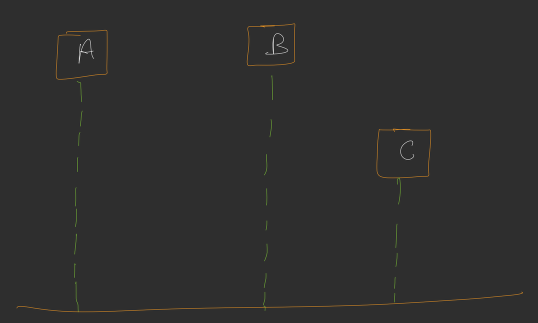
Sequence Diagrams



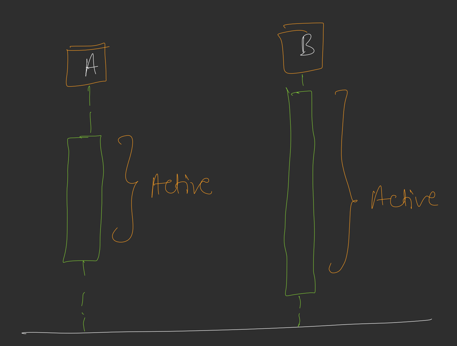
① Representing Object.



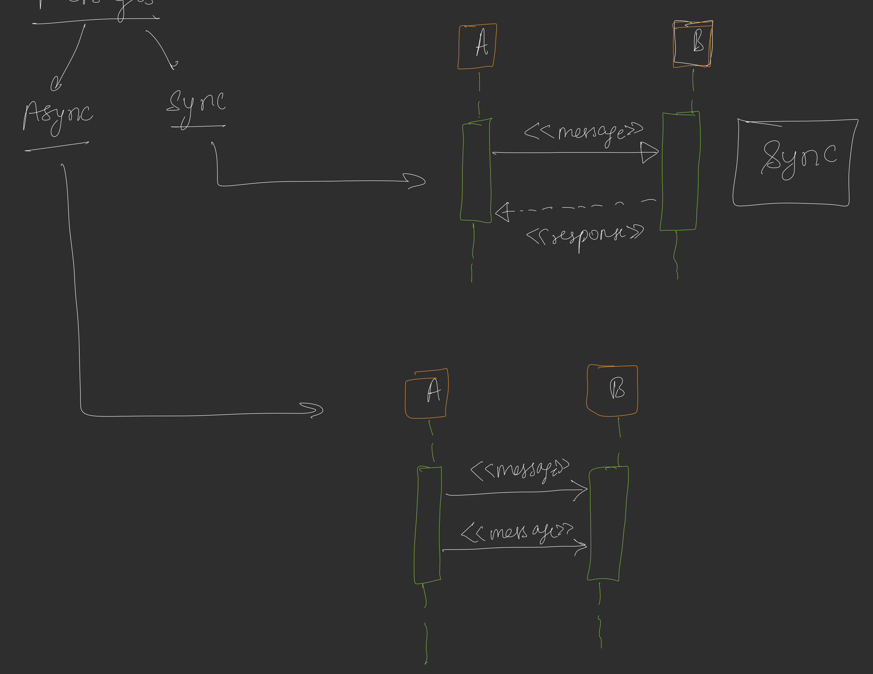
② Lifeline



③ Activation Bar

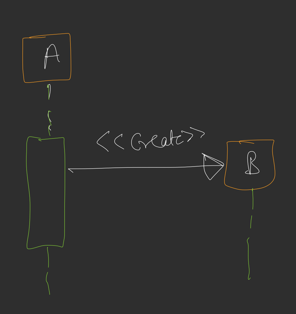


④ Messages

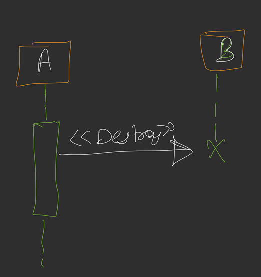


⑤ Create & Destroy Message

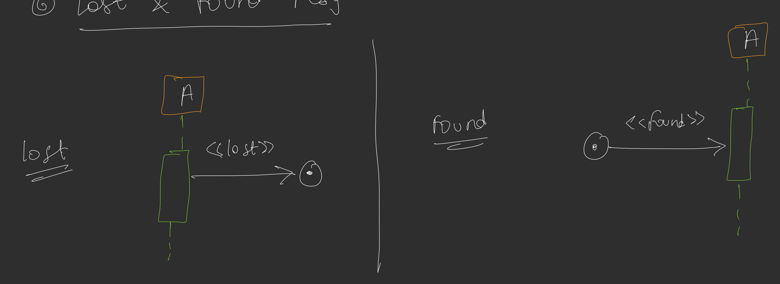
Create →



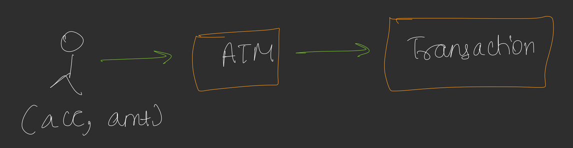
Destroy →



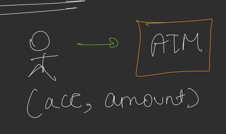
⑥ Lost & Found Msg

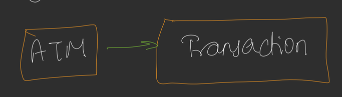


ATM Transaction flow



Use Case

① 

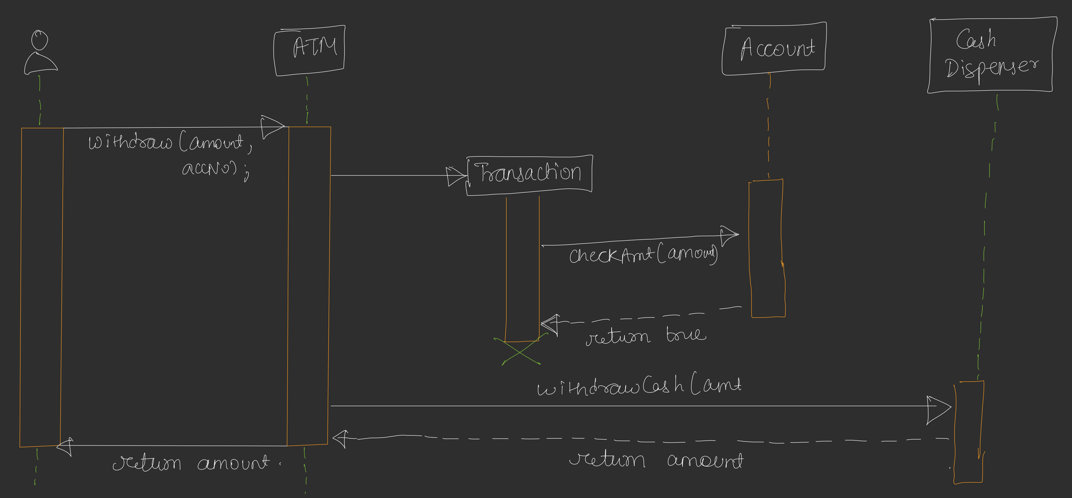
② 

① verify ATM pin
② verify Account
③ Tell Cash Dispenser for money.

Sequence Diagram

① Use Case
② Objects
- ATM
- User
- Transaction
- Account
- Cash Dispenser

③ Draw Diagram



alt : (if-else)
Option : (if)
loop : for/while

Lecture 6 : SOLID Principles Part-2

1. Recap of SOLID Principles

Before diving into the remaining two principles (Interface Segregation and Dependency Inversion), a quick recap:

1. **Single Responsibility Principle (SRP)**
   • A class should have only one reason to change—i.e., one responsibility.
2. **Open/Closed Principle (OCP)**
   • Software entities (classes, modules, functions) should be open for extension but closed for modification.
3. **Liskov Substitution Principle (LSP)**
   • Subtypes must be substitutable for their base types without altering the correctness of the program.

We have already covered SRP, OCP, and LSP conceptually. What follows is a **detailed breakdown of LSP guidelines**, then full explanations of **Interface Segregation Principle (ISP)** and **Dependency Inversion Principle (DIP)** with illustrative examples.

2. Deep Dive: Liskov Substitution Principle (LSP)

> **Definition:** *"Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program."*

2.1 Why LSP "Breaks" Often

- Inheritance ensures that subclasses have the same methods, but not necessarily the same behavior or contractual guarantees.
- Without clear rules, a subclass may override a method incorrectly (e.g., throwing unexpected exceptions, changing return values or method signatures), causing client code to fail.

2.2 Three Categories of LSP Rules

LSP compliance hinges on three broad categories of rules, each with sub-rules:

1. **Signature Rules**
2. **Property Rules**
3. **Method Rules**

2.3 Signature Rules

Ensure that method overrides preserve the *contractual interface* of the parent:

1. **Method Argument Rule**
   - The overridden method in the subclass must accept the same argument types as the parent, or *wider* (a “broader” type up the inheritance chain).
   - *Example:* If the parent method takes a `String`, the child override must also take `String` (or a supertype, e.g., `Object`), never an unrelated type like `Integer`.
2. **Return Type Rule**
   - The subclass’s return type must be the same as the parent’s, or *narrower* (a subtype).
   - *Covariant returns* are allowed (e.g., parent returns `Animal`; child can return `Dog`), but not contravariant (e.g., child cannot return `Object` if the parent returns `Animal`).
3. **Exception Rule**
   - The subclass may throw fewer or more specific exceptions than the parent, but never broader exceptions that the client is not expecting.
   - *Example:* If the parent method declares it throws `RuntimeError`, the child can throw `IndexOutOfBoundsException` (a subtype) but not a totally unrelated exception like `OutOfMemoryError` if it isn’t within that hierarchy.

2.4 Property Rules

Ensure that the subclass preserves key “properties” of the parent class:

1. **Class Invariant**
   - Any invariant (a condition that must always hold true) specified on the parent must not be violated by the subclass.
   - *Example:* A `BankAccount` class may mandate that `balance >= 0`. A subclass `CheatAccount` that allows negative balances breaks this invariant and thus violates LSP.
2. **History Constraint**
   - The subclass must preserve the “history” or lifecycle behavior of the parent. It cannot remove or disable operations that clients expect to always work.
   - *Example:* A `FixedDepositAccount` (subclass) that throws an exception on every withdrawal violates the parent’s guarantee that withdrawal is always allowed.

2.5 Method Rules

Ensure that method-specific preconditions and postconditions remain consistent:

1. **Precondition (Method Rule – Before Execution)**
   - Preconditions specify what must be true *before* a method executes.
   - A subclass may *weaken* (make less strict) the precondition (accept a broader range of inputs), but must not *strengthen* it (require more than the parent).
   - *Example:* Parent requires `0 ≤ x ≤ 5`; child can accept `0 ≤ x ≤ 10` (weaker), but not `0 ≤ x ≤ 3` (stronger), or clients that supply `x = 7` would fail.
2. **Postcondition (Method Rule – After Execution)**
   - Postconditions specify what must be true *after* a method completes.
   - A subclass may *strengthen* the postcondition (guarantee more), but must not *weaken* it (guarantee less).
   - *Example:* Parent `brake()` method guarantees "speed decreases"; a subclass `HybridCar` may also increase battery charge (strengthening), but must never leave speed unchanged or increased (weakening).

2.6 Key Takeaways for LSP

- Always check whether a subclass truly *behaves* like its parent, not just whether it *compiles*.
- Remember: **Signature**, **Property**, and **Method** rules each have clearly defined sub-rules—use these as a checklist when designing hierarchies.
- Violations often manifest as unexpected exceptions, incorrect return values, or broken invariants.

---

3. Interface Segregation Principle (ISP)

Definition: *"Clients should not be forced to depend on interfaces they do not use."*
Key Idea: It's better to have many small, client-specific interfaces than one large, general-purpose interface.

3.1 The Problem with "Fat" Interfaces

- A single interface/class that includes every conceivable method (e.g., both 2D and 3D shape operations) forces some implementers to override methods they don't need.
- Unneeded methods often either throw exceptions or remain unimplemented, hurting maintainability and violating SRP.

3.2 Illustrative Example: Shapes

“Fat” Interface Approach

```
// See Code for example
```

1. **Problem:** `Square` and `Rectangle` are forced to implement `volume()`, leading to stubs or exceptions.

3.3 ISP Solution: Segregate into Two Interfaces

2DShape

```
class TwoDShape {
  double area();
}
class Square :public TwoDShape { ... }
class Rectangle : public TwoDShape { ... }
```

3DShape

```
class ThreeDShape {
public:
    virtual double area() = 0;
    virtual double volume() = 0;
};

class Cube : public ThreeDShape {
    // ...
};
```

Benefits:

- Each implementer only deals with methods it actually uses.
- Code is cleaner, adheres to SRP, and avoids unnecessary stubs or exceptions.

---

4. Dependency Inversion Principle (DIP)

Definition:

1. High-level modules should not depend on low-level modules; both should depend on abstractions.
2. Abstractions should not depend on details; details should depend on abstractions.

4.1 The Problem with Direct Coupling

- A high-level class (e.g., `UserService`) that directly calls concrete low-level classes (`SqlDatabase`, `MongoDatabase`) becomes tightly coupled.
- Changing the low-level implementation (e.g., swapping MongoDB for Cassandra) forces modifications in the high-level class—violating OCP.

4.2 DIP Solution: Introduce an Abstraction Layer

Define an Abstraction

```
class Persistence {
public:
  virtual void save(const User& u) = 0;
};
```

Make Low-Level Classes Depend on the Abstraction

```
class SqlDatabase : public Persistence { ... override save(...) ... }
class MongoDatabase : public Persistence { ... override save(...) ... }
```

High-Level Module Depends Only on the Abstraction

```
class UserService {
private:
  Persistence* db;   // injected dependency
public:
  UserService(Persistence* p) : db(p) { }
  void storeUser(const User& u) { db->save(u); }
};
```

Dependency Injection

- At runtime, instantiate `UserService` with either `new SqlDatabase(…)` or `new MongoDatabase(…)` (or a future `CassandraDatabase`), without changing `UserService` itself.

4.3 Real-World Analogy

- A company CEO (high-level) doesn't instruct individual developers (low-level) directly. Instead, a manager (abstraction) relays requirements.
- The CEO depends only on the manager's interface; developers depend on the manager for directives. Swapping out developers doesn't affect the CEO's workflow.

---

5. Final Thoughts & Trade-Offs

- **SOLID principles are guidelines, not hard laws.** In practice, business requirements and performance constraints may necessitate trade-offs.
- Adhering to these principles generally leads to more **maintainable**, **scalable**, and **extensible** code—but balance is key.
- Whenever you find yourself violating one principle, check whether it's in service of a higher-priority need (e.g., performance) and document your reasoning.

By following these LSP guidelines and applying ISP and DIP judiciously, you'll write cleaner, more robust object-oriented code that stands the test of evolving requirements.

Lecture 6 : SOLID Principles Part-2

1. Recap of SOLID Principles

Before diving into the remaining two principles (Interface Segregation and Dependency Inversion), a quick recap:

1. **Single Responsibility Principle (SRP)**
   • A class should have only one reason to change—i.e., one responsibility.
2. **Open/Closed Principle (OCP)**
   • Software entities (classes, modules, functions) should be open for extension but closed for modification.
3. **Liskov Substitution Principle (LSP)**
   • Subtypes must be substitutable for their base types without altering the correctness of the program.

We have already covered SRP, OCP, and LSP conceptually. What follows is a **detailed breakdown of LSP guidelines**, then full explanations of **Interface Segregation Principle (ISP)** and **Dependency Inversion Principle (DIP)** with illustrative examples.

2. Deep Dive: Liskov Substitution Principle (LSP)

> **Definition:** *"Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program."*

2.1 Why LSP "Breaks" Often

- Inheritance ensures that subclasses have the same methods, but not necessarily the same behavior or contractual guarantees.
- Without clear rules, a subclass may override a method incorrectly (e.g., throwing unexpected exceptions, changing return values or method signatures), causing client code to fail.

2.2 Three Categories of LSP Rules

LSP compliance hinges on three broad categories of rules, each with sub-rules:

1. **Signature Rules**
2. **Property Rules**
3. **Method Rules**

2.3 Signature Rules

Ensure that method overrides preserve the *contractual interface* of the parent:

1. **Method Argument Rule**
   - The overridden method in the subclass must accept the same argument types as the parent, or *wider* (a "broader" type up the inheritance chain).
   - *Example:* If the parent method takes a `String`, the child override must also take `String` (or a supertype, e.g., `Object`), never an unrelated type like `Integer`.
2. **Return Type Rule**
   - The subclass's return type must be the same as the parent's, or *narrower* (a subtype).
   - *Covariant returns* are allowed (e.g., parent returns `Animal`; child can return `Dog`), but not contravariant (e.g., child cannot return `Object` if the parent returns `Animal`).
3. **Exception Rule**
   - The subclass may throw fewer or more specific exceptions than the parent, but never broader exceptions that the client is not expecting.
   - *Example:* If the parent method declares it throws `RuntimeError`, the child can throw `IndexOutOfBoundsException` (a subtype) but not a totally unrelated exception like `OutOfMemoryError` if it isn't within that hierarchy.

2.4 Property Rules

Ensure that the subclass preserves key "properties" of the parent class:

1. **Class Invariant**
   - Any invariant (a condition that must always hold true) specified on the parent must not be violated by the subclass.
   - *Example:* A `BankAccount` class may mandate that `balance >= 0`. A subclass `CheatAccount` that allows negative balances breaks this invariant and thus violates LSP.
2. **History Constraint**
   - The subclass must preserve the "history" or lifecycle behavior of the parent. It cannot remove or disable operations that clients expect to always work.
   - *Example:* A `FixedDepositAccount` (subclass) that throws an exception on every withdrawal violates the parent's guarantee that withdrawal is always allowed.

2.5 Method Rules

Ensure that method-specific preconditions and postconditions remain consistent:

1. **Precondition (Method Rule – Before Execution)**
   - Preconditions specify what must be true *before* a method executes.
   - A subclass may *weaken* (make less strict) the precondition (accept a broader range of inputs), but must not *strengthen* it (require more than the parent).
   - *Example:* Parent requires `0 ≤ x ≤ 5`; child can accept `0 ≤ x ≤ 10` (weaker), but not `0 ≤ x ≤ 3` (stronger), or clients that supply `x = 7` would fail.
2. **Postcondition (Method Rule – After Execution)**
   - Postconditions specify what must be true *after* a method completes.
   - A subclass may *strengthen* the postcondition (guarantee more), but must not *weaken* it (guarantee less).
   - *Example:* Parent `brake()` method guarantees "speed decreases"; a subclass `HybridCar` may also increase battery charge (strengthening), but must never leave speed unchanged or increased (weakening).

2.6 Key Takeaways for LSP

- Always check whether a subclass truly *behaves* like its parent, not just whether it *compiles*.
- Remember: **Signature**, **Property**, and **Method** rules each have clearly defined sub-rules—use these as a checklist when designing hierarchies.
- Violations often manifest as unexpected exceptions, incorrect return values, or broken invariants.

---

3. Interface Segregation Principle (ISP)

Definition: *"Clients should not be forced to depend on interfaces they do not use."*
Key Idea: It's better to have many small, client-specific interfaces than one large, general-purpose interface.

3.1 The Problem with "Fat" Interfaces

- A single interface/class that includes every conceivable method (e.g., both 2D and 3D shape operations) forces some implementers to override methods they don't need.
- Unneeded methods often either throw exceptions or remain unimplemented, hurting maintainability and violating SRP.

3.2 Illustrative Example: Shapes

“Fat” Interface Approach

```
// See Code for example
```

1. **Problem:** `Square` and `Rectangle` are forced to implement `volume()`, leading to stubs or exceptions.

3.3 ISP Solution: Segregate into Two Interfaces

2DShape

```
class TwoDShape {
  double area();
}
class Square :public TwoDShape { ... }
class Rectangle : public TwoDShape { ... }
```

3DShape

```
class ThreeDShape {
public:
    virtual double area() = 0;
    virtual double volume() = 0;
};

class Cube : public ThreeDShape {
    // ...
};
```

Benefits:

- Each implementer only deals with methods it actually uses.
- Code is cleaner, adheres to SRP, and avoids unnecessary stubs or exceptions.

---

4. Dependency Inversion Principle (DIP)

Definition:

1. High-level modules should not depend on low-level modules; both should depend on abstractions.
2. Abstractions should not depend on details; details should depend on abstractions.

4.1 The Problem with Direct Coupling

- A high-level class (e.g., `UserService`) that directly calls concrete low-level classes (`SqlDatabase`, `MongoDatabase`) becomes tightly coupled.
- Changing the low-level implementation (e.g., swapping MongoDB for Cassandra) forces modifications in the high-level class—violating OCP.

4.2 DIP Solution: Introduce an Abstraction Layer

Define an Abstraction

```
class Persistence {
public:
  virtual void save(const User& u) = 0;
};
```

Make Low-Level Classes Depend on the Abstraction

```
class SqlDatabase : public Persistence { ... override save(...) ... }
class MongoDatabase : public Persistence { ... override save(...) ... }
```

High-Level Module Depends Only on the Abstraction

```
class UserService {
private:
  Persistence* db;   // injected dependency
public:
  UserService(Persistence* p) : db(p) { }
  void storeUser(const User& u) { db->save(u); }
};
```

Dependency Injection

- At runtime, instantiate `UserService` with either `new SqlDatabase(…)` or `new MongoDatabase(…)` (or a future `CassandraDatabase`), without changing `UserService` itself.

4.3 Real-World Analogy

- A company CEO (high-level) doesn't instruct individual developers (low-level) directly. Instead, a manager (abstraction) relays requirements.
- The CEO depends only on the manager's interface; developers depend on the manager for directives. Swapping out developers doesn't affect the CEO's workflow.

---

5. Final Thoughts & Trade-Offs

- **SOLID principles are guidelines, not hard laws.** In practice, business requirements and performance constraints may necessitate trade-offs.
- Adhering to these principles generally leads to more **maintainable**, **scalable**, and **extensible** code—but balance is key.
- Whenever you find yourself violating one principle, check whether it's in service of a higher-priority need (e.g., performance) and document your reasoning.

By following these LSP guidelines and applying ISP and DIP judiciously, you'll write cleaner, more robust object-oriented code that stands the test of evolving requirements.