System Design

Low-Level Design (LLD) Interview Preparation

Prudhvi Peyyala

Contents

1 Introduction

This document is a comprehensive guide to System Design and Low-Level Design (LLD) interview preparation. It covers essential OOP concepts, SOLID principles, design patterns, and practical LLD problem-solving strategies. Use this as a structured learning resource to master system design interviews.

2 Study Routine Pattern

If you can give **4–6+ hours** daily, follow this structure on heavy days:

- 1–2 hours: Theory (OOP, SOLID, design patterns, UML refresh)
- 2–3 hours: One full LLD problem (requirements → design → partial code)
- 1 hour: Review previous designs or watch one high-quality LLD discussion

3 Core Foundations

3.1 Object-Oriented Programming (Java)

- Revise: classes, interfaces, abstract classes, inheritance, polymorphism, composition.
- Implement 2–3 mini examples:
  - Shape hierarchy (Circle, Rectangle, Triangle).
  - Payment methods (Card, UPI, NetBanking).
  - Notification types (Email, SMS, Push).

3.1.1 The Four Pillars of OOP

- **Encapsulation:** Bundle data (attributes) and methods that operate on them into a single unit (class). Hide internal details using access modifiers (`private`, `protected`, `public`). Exposure only through getter/setter methods **ensures data integrity and allows validation**. Example: A `BankAccount` class keeps balance `private` and provides `withdraw()` with validation.
- **Abstraction:** Expose only relevant functionality, hide implementation complexity. Achieved via abstract classes and interfaces. Clients use *what* the object does, not *how* it does it. Example: `PaymentMethod` interface defines `pay()` method; subclasses (`CreditCard`, `UPI`) implement differently.

- **Inheritance:** Allows classes to inherit attributes and methods from parent classes, promoting code reuse and hierarchical organization. Subclasses extend or override parent behavior. Single inheritance in Java (unlike C++). Use `extends` keyword. Example: `SavingsAccount` extends `BankAccount` and inherits balance management but adds interest-specific logic.
- **Polymorphism:** Objects can take multiple forms. *Method overriding:* subclass provides its own implementation of a parent method (runtime polymorphism). *Method overloading:* same method name with different parameters (compile-time polymorphism). Enables writing flexible, extensible code. Example: Multiple `PaymentMethod` subclasses override `pay()` method differently.

3.1.2 Class Hierarchy: Class vs Interface vs Abstract Class

- **Concrete class:** Standard class with implementation. Can be instantiated (`new ClassName()`). Contains attributes and method implementations. Example: `CreditCardPayment` extends a common payment base.
- **Abstract class:** Partial implementation; cannot be instantiated directly. Can contain abstract methods (must be implemented by subclasses) and concrete methods. Use for related classes with common behavior. Keyword: `abstract`. Example: `abstract class PaymentMethod` defines common validation; subclasses implement `process()` differently.
- **Interface:** Pure contract; no implementation (before Java 8). All methods are abstract (implicitly). Use for unrelated classes to follow a common protocol (`Comparable`, `Serializable`). Multiple implementation supported (`implements`). Since Java 8, can include default and static methods. Example: `PaymentMethod` interface; `CreditCard`, `UPI`, `Wallet` all implement it.
- **When to use:** Concrete class (simple, direct instantiation) → abstract class (shared behavior among related classes) → interface (unrelated classes following same contract).

3.1.3 Composition vs Inheritance: Key Trade-offs

- **Composition (Has-A):** Object contains instances of other classes. Flexible, loose coupling. Change behavior at runtime by swapping composed objects. Preferred for complex systems. Example: `PaymentProcessor` *has-a* `PaymentMethod` (`private PaymentMethod method;`), set via constructor or setter.
- **Inheritance (Is-A):** Subclass inherits from parent. Tight coupling. Changes to parent affect all subclasses. Harder to test and maintain. Use only for true hierarchies (Vehicle → Car → SportsCar). Example: `SavingsAccount` is-a `BankAccount`.

- **Guideline: Favor composition:** Prefer composition over inheritance. More flexible and testable. Allows changing behavior without creating new subclasses. Example: Instead of `EmailNotification` extending `Notification`, create `Notification` with `private NotificationChannel channel;`.
- **Liskov Substitution Principle (LSP):** Subclass must be usable wherever parent is expected without breaking code. If inheritance violates LSP, use composition instead. Red flag: Subclass that disables or changes parent behavior.

3.1.4 Common OOP Anti-patterns (What to Avoid)

- **God class:** Single class doing too much. Violates SRP. Example: `Order` handling payments, inventory, notifications, and shipping. Solution: Split into `Order`, `PaymentProcessor`, `InventoryManager`, `NotificationService`.
- **Deep inheritance hierarchies:** Hard to understand and maintain. Max 2–3 levels usually. Example: `Animal` → `Mammal` → `Carnivore` → `Dog` → `Labrador`. Solution: Use composition or interfaces.
- **Leaky abstraction:** Implementation details seep out. Breaks encapsulation. Example: `List<String>` exposing that it uses `ArrayList`. Solution: Always expose only the interface, hide implementation (return `List` not `ArrayList`).
- **Mutable objects:** Objects that change state are hard to test and debug. Example: Shared mutable `List`. Solution: Prefer immutability or return defensive copies.
- **Missing null checks:** Can cause `NullPointerException`. Solution: Use `Optional` or validate inputs. Example: `if (object != null) { ... }` or `Optional.ofNullable(object).ifPresent(...)`.
- **Tight coupling:** Classes directly dependent on each other. Hard to test independently. Solution: Use interfaces and dependency injection.

3.1.5 Access Modifiers: Encapsulation Mechanism

- **public:** Accessible everywhere. Use for API methods. Example: `public void pay() { ... }`
- **protected:** Accessible in subclasses and same package. Use for helper methods meant for inheritance. Example: `protected void validate() { ... }`
- **private:** Accessible only within class. Use for internal implementation details. Example: `private void checkBalance() { ... }`
- **package-private (default):** Accessible in same package. Use rarely; be explicit about visibility when possible.

- **Principle:** Expose minimum necessary; keep everything `private` by default, upgrade visibility only when needed.

3.1.6 OOP Revision Checklist for System Design Interviews

- Before designing any system, ask: Which classes do I need? What attributes and methods for each?
- Apply the 4 pillars: Use **encapsulation** (private fields, public methods), **abstraction** (interfaces), **inheritance** (extends, but sparingly), **polymorphism** (override methods).
- Prefer **composition** over inheritance. If you think about inheritance, ask: Can I use composition instead?
- Keep classes focused: One responsibility (SRP). Small, cohesive classes beat large monolithic ones.
- Always think about extensibility: Use interfaces, not concrete classes. Code to contracts.
- Design error handling: Exceptions for failure cases. Define custom exceptions when useful.
- Avoid god classes and deep inheritance chains. Split responsibility early.
- Test your classes independently (unit tests). If hard to test, design is probably bad (tight coupling).
- During interview, *speak out loud*: "This `User` class handles authentication. I'll separate it into `User` (data) and `AuthService` (logic)."
- Remember: Good OOP design reduces bugs, eases testing, and impresses interviewers.

3.2 SOLID Principles

- Learn and internalize:
  - SRP, OCP, LSP, ISP, DIP with tiny Java examples.
- Practice: Take a "god class" and refactor it into multiple SRP-based classes with clear responsibilities.

3.2.1 Single Responsibility Principle (SRP)

- **Definition:** A class should have one and only one reason to change. Each class should have a single, well-defined responsibility.

- **Why it matters:** Reduces code complexity, makes testing easier, and improves maintainability. When a class has multiple responsibilities, changes to one can break another.
- **Example (Bad):**
  - `UserService` class handling user creation, authentication, and email notifications violates SRP.
  - Changes to email logic would affect user creation logic (tight coupling).
- **Example (Good):** Split into:
  - `User` class: Represents user data.
  - `UserService` class: Handles user creation and authentication.
  - `EmailService` class: Handles sending emails.
- **Java code snippet**:
  - Bad: `class User { login(); sendEmail(); calculateTax(); }`
  - Good: Separate classes with single responsibilities.

3.2.2 Open/Closed Principle (OCP)

- **Definition:** Software entities (classes, modules, functions) should be open for extension but closed for modification.
- **Why it matters:** Allows adding new functionality without changing existing code. Reduces risk of breaking existing features.
- **Example (Bad):**
  - `PaymentProcessor` class with hardcoded if–else for different payment types (CreditCard, PayPal, etc.).
  - Adding a new payment type requires modifying the class.
- **Example (Good):**
  - Create `PaymentMethod` interface with `pay()` method.
  - Each payment type (CreditCard, PayPal) implements the interface.
  - `PaymentProcessor` uses polymorphism; no changes needed when adding new payment types.
- **Java code snippet**:
  - `interface PaymentMethod { void pay(double amount); }`
  - `class CreditCard implements PaymentMethod { ... }`
  - `class PayPal implements PaymentMethod { ... }`

3.2.3 Liskov Substitution Principle (LSP)

- **Definition:** Subtypes must be substitutable for their base types. A subclass should not violate the contract of the parent class.
- **Why it matters:** Ensures that inheritance is used correctly. Prevents bugs caused by unexpected behavior in subclasses.
- **Example (Bad):**
  - `Bird` class with `fly()` method.
  - `Penguin` extends `Bird` but overrides `fly()` to throw an exception (penguins cannot fly).
  - Code expecting a `Bird` will break if it is actually a `Penguin`.
- **Example (Good):**
  - Create separate abstractions: `Bird` (basic attributes) and `FlyingBird` extends `Bird` with `fly()` method.
  - `Penguin` extends `Bird` without the `fly()` method contract.
- **Red flag:** If a subclass overrides a method to do nothing or throw an exception, LSP is violated.

3.2.4 Interface Segregation Principle (ISP)

- **Definition:** Many client-specific interfaces are better than one general-purpose interface. Clients should not depend on methods they do not use.
- **Why it matters:** Prevents fat interfaces that force clients to implement unnecessary methods. Improves code flexibility.
- **Example (Bad):**
  - `Worker` interface with methods: `work()`, `eat()`, `sleep()`.
  - `Robot` implements `Worker` but has no need for `eat()` or `sleep()`.
  - `Robot` is forced to implement dummy versions of these methods.
- **Example (Good):**
  - `Workable` interface with `work()`.
  - `Eatable` interface with `eat()`.
  - `Sleepable` interface with `sleep()`.
  - `Human` implements `Workable`, `Eatable`, `Sleepable`.
  - `Robot` implements `Workable`.
- **Java code snippet**:

- Instead of `interface Worker { work(); eat(); sleep(); }`
- Use: `interface Workable { void work(); }` and `interface Eatable { void eat(); }`

3.2.5 Dependency Inversion Principle (DIP)

- **Definition:** High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces or abstract classes).
- **Why it matters:** Reduces coupling between modules, makes testing easier (mock dependencies), and improves flexibility.
- **Example (Bad):**
  - `PaymentProcessor` directly creates and uses `CreditCardPayment` class.
  - Tightly coupled; changing payment method requires changing `PaymentProcessor`.
- **Example (Good):**
  - `PaymentProcessor` depends on `PaymentMethod` interface (abstraction).
  - Pass the payment method via constructor (dependency injection).
  - Easy to swap payment methods without changing `PaymentProcessor`.
- **Java code snippet**:
  - Bad: `class PaymentProcessor { private CreditCardPayment payment = new CreditCardPayment(); }`
  - Good: `class PaymentProcessor { private PaymentMethod payment; public PaymentProcessor(PaymentMethod method) { this.payment = method; } }`

3.2.6 SOLID Principles Quick Checklist

- **SRP:** Does each class have one reason to change? Can you describe the class responsibility in one sentence?
- **OCP:** Can you add new functionality without modifying existing classes? Use interfaces and polymorphism.
- **LSP:** Can a subclass replace its parent without breaking code? Avoid overriding to throw exceptions.
- **ISP:** Are clients forced to depend on methods they do not use? Split fat interfaces.
- **DIP:** Are high-level modules directly creating low-level dependencies? Use dependency injection and interfaces.
- **Golden rule:** Code to interfaces, not implementations. This helps achieve all 5 principles.

3.3 UML Basics for LLD

- Focus on simple class diagrams:
  - Classes, attributes, methods.
  - Relationships: association, aggregation, composition.
- Draw small diagrams for:
  - Library (Book, Member, Loan).
  - User–Order–Product for a simple commerce flow.

3.3.1 What is UML? Why Use It?

- **Definition:** UML (Unified Modeling Language) is a standardized visual notation for designing and documenting software systems (a common visual language to explain a system). It helps communicate structure, behavior, and relationships between components before coding.
- **Why in LLD?** Clarifies system design, catches architectural issues early, communicates intent to team members, and creates a blueprint for implementation.
- **Scope for LLD:** Focus on class diagrams only. Ignore sequence diagrams, use case diagrams, and state diagrams for now.

3.3.2 Class Diagrams: The Core of LLD

- **What it shows:** Classes, their attributes, methods, and relationships.
- **Basic structure:** A box with three sections:
  - Top: Class name
  - Middle: Attributes (data members)
  - Bottom: Methods (behaviors)
- **Example:**

```
+---------------------+
|    BankAccount      |
+---------------------+
|- accountId: String  |
|- balance: double    |
|- owner: User        |
+---------------------+
|+ deposit(amount)    |
|+ withdraw(amount)   |
|- validatePin()      |
+---------------------+
```

3.3.3 Notation: Understanding Symbols

- **Access modifiers:**
  - `+` = public (accessible everywhere)
  - `-` = private (accessible only in this class)
  - `#` = protected (accessible in subclasses and same package)
  - ` ` = package-private (accessible in same package)
- **Types:** Write data types next to attributes. Example: `balance: double`, `name: String`, `age: int`.
- **Method signatures:** Include parameters and return types. Example: `withdraw(amount: double): boolean`.

3.3.4 Relationships: How Classes Connect

- **Association (Has-A):** One class uses or references another. Draw a simple line.
  - Example: `User` → `Account`. User has access to Account.
  - Optional label: `accesses`, `owns`.
  - Multiplicity: Write numbers at ends. `1 ---- *` means "one User to many Accounts."
- **Aggregation (Weak Has-A):** A part-of relationship where parts can exist independently. Draw a line with a hollow diamond at the container end.
  - Example: Aggregation example: Library and Book. Library contains books, but books can exist without a specific library.
  - Think: Library is a collection of books.
- **Composition (Strong Has-A):** A part-of relationship where parts cannot exist without the container. Draw a line with a filled diamond.
  - Example: Composition example: Car and Engine. If Car is destroyed, Engine is too.
  - Think: Order and OrderItems. Remove Order, OrderItems are meaningless.
- **Inheritance (Is-A):** A class extends another. Draw a line with a hollow triangle pointing to parent.
  - Example: Inheritance example: `SavingsAccount` and `BankAccount`. `SavingsAccount` is a `BankAccount` (extends parent class).
  - Use sparingly; prefer composition.

- **Dependency:** One class temporarily uses another (method parameter, local variable). Dashed line with arrow.
  - Example: `PaymentProcessor -.-> PaymentMethod`.

3.3.5 Step-by-Step: Drawing a Class Diagram

1. **Identify classes:** What are the main entities? (User, Order, Payment, etc.)
2. **List attributes:** What data does each class hold? (userId, amount, status, etc.)
3. **List methods:** What behaviors? (placeOrder(), pay(), getBalance(), etc.)
4. **Add relationships:** How do classes interact? (User places Order, Order contains Items, etc.)
5. **Review:** Does it match the problem? Are relationships correct? Are responsibilities clear?

3.3.6 Common Mistakes to Avoid

- **Too many classes:** Do not model every tiny concept. Start with main entities.
- **Mixing association types:** Aggregation vs. composition vs. association. Think carefully: Can the part exist without the whole?
- **Missing multiplicity:** Always clarify: `1-to-1`, `1-to-many`, `many-to-many`.
- **Attribute as class:** Example: Do not make `UserId` a class. Make `id: String` an attribute of `User`.
- **God classes:** If a class has too many responsibilities, split it.

3.3.7 Quick Reference: When Drawing in Interviews

- Start with a simple, high-level diagram showing main entities and their relationships.
- Add attributes and methods as you discuss the design.
- Use clear naming: `User`, `Order`, `Payment` (not `U`, `O`, `P`).
- Draw on the whiteboard or paper clearly. Neat diagrams impress interviewers.
- Explain: "User has multiple Orders, each Order has multiple Items." Talking through relationships shows understanding.

3.4 Essential Design Patterns

Design patterns are reusable solutions to common problems in software design. Understanding and applying these patterns is crucial for system design interviews. This section covers the essential patterns used in LLD interviews, organized by their creational/structural/behavioral purpose.

Pattern documentation: Each pattern includes:

- **Intent:** Purpose and use case of the pattern.
- **Structure:** Key components and their relationships.
- **Example:** Practical Java implementation.
- **When to use:** Scenarios where the pattern is applicable.

(Minimum set)
For each pattern, note: intent (2–3 lines) plus one small Java example.

3.4.1 Creational

Creational patterns focus on object creation mechanisms. They encapsulate the instantiation process to make systems independent of how objects are composed and represented.

Singleton

- – **Intent:** Ensure a class has only one instance and provide a global point of access to it. Useful for shared resources like database connections, loggers, or configuration managers.
  – **Structure:** Private constructor, static instance, public static `getInstance()` method.
  – **Example (thread-safe):**

```
public class DatabaseConnection {
    private static DatabaseConnection instance;

    private DatabaseConnection() {}

    public static synchronized DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }
```

```
}
```

- **When to use:** Global configuration, logging, thread pools, cache. Note: Test-friendly alternatives use dependency injection.

Factory Method

- **Intent:** Define an interface for creating objects, but let subclasses decide which class to instantiate. Encapsulates object creation logic.
- **Structure:** Abstract creator class with abstract factory method; concrete creators implement the factory method to return specific product types.
- **Example:**

```
abstract class PaymentMethodFactory {
    abstract PaymentMethod createPayment();
}

class CreditCardFactory extends PaymentMethodFactory {
    @Override
    PaymentMethod createPayment() {
        return new CreditCard();
    }
}
```

Builder

- **Intent:** Construct complex objects step-by-step using a builder. Separates construction logic from representation, allowing different configurations. Useful for objects with many optional parameters.
- **Structure:** Builder class with fluent setter methods, each returning `this` for method chaining. A `build()` method returns the final constructed object.
- **Example:**

```
public class House {
    private String roof;
    private String walls;
    private int windows;

    private House(HouseBuilder builder) {
        this.roof = builder.roof;
```

```
        this.walls = builder.walls;
        this.windows = builder.windows;
    }

    public static class HouseBuilder {
        private String roof;
        private String walls;
        private int windows;

        public HouseBuilder withRoof(String roof) {
            this.roof = roof;
            return this;
        }

        public HouseBuilder withWalls(String walls) {
            this.walls = walls;
            return this;
        }

        public HouseBuilder withWindows(int windows) {
            this.windows = windows;
            return this;
        }

        public House build() {
            return new House(this);
        }
    }
}
```

- Usage: `House house = new House.HouseBuilder().withRoof("Tiles").withWalls("Brick").withW`
- **When to use:** Complex object creation with multiple parameters, optional fields, or when you want to avoid constructor telescoping (many constructors with different parameter combinations).

3.4.2 Structural

Structural patterns deal with object composition, creating relationships between entities in a way that enables new functionality. They help ensure that when one part changes, the entire structure does not need to do so.

Adapter

- **Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces. It is often called a *wrapper*.[web:41][web:57]
- **Structure:**
  - `Target` interface that the client expects.
  - `Adaptee` class with an incompatible interface.
  - `Adapter` implementing `Target` and internally using an `Adaptee` instance to translate calls.
- **Example (Payment gateway adapter):**

```
interface PaymentProcessor {
    void pay(double amount);
}

// Third-party library (incompatible interface)
class ExternalGateway {
    void makePayment(double total) {
        System.out.println("Paying " + total + " via external gateway");
    }
}

// Adapter
class ExternalGatewayAdapter implements PaymentProcessor {
    private final ExternalGateway gateway;

    ExternalGatewayAdapter(ExternalGateway gateway) {
        this.gateway = gateway;
    }

    public void pay(double amount) {
        gateway.makePayment(amount);
    }
}

// Usage:
PaymentProcessor processor =
    new ExternalGatewayAdapter(new ExternalGateway());
processor.pay(1000);
```

- **When to use:** Integrating legacy or third-party code whose interfaces you cannot change, reusing existing classes in a new API, or bridging DTO/domain models across service boundaries.[web:41][web:54]

Bridge

- **Intent:** Decouple an abstraction from its implementation so that the two can vary independently. Bridge is like a "designed up-front" adapter that separates what an object does from how it is implemented.[web:41][web:51]
- **Structure:**
  - `Abstraction` defining the high-level operations and holding a reference to an `Implementor`.
  - `RefinedAbstraction` subclasses that extend behavior.
  - `Implementor` interface for low-level operations.
  - `ConcreteImplementor` classes providing platform-specific or variant-specific implementations.
- **Example (Device + Remote):**

```
interface Device {
    void turnOn();
    void turnOff();
}

class Tv implements Device {
    public void turnOn()  { System.out.println("TV ON"); }
    public void turnOff() { System.out.println("TV OFF"); }
}

class Radio implements Device {
    public void turnOn()  { System.out.println("Radio ON"); }
    public void turnOff() { System.out.println("Radio OFF"); }
}

// Abstraction
class Remote {
    protected Device device;
    Remote(Device device) { this.device = device; }

    void togglePower(boolean on) {
        if (on) device.turnOn(); else device.turnOff();
    }
}

// Usage:
Remote remote = new Remote(new Tv());
remote.togglePower(true);
```

- **When to use:** Multiple dimensions of variation (e.g., different devices and different controllers, different renderers and shapes), avoiding combinatorial explosion of subclasses like `LargeRedCircle`, `SmallBlueSquare`, etc.[web:41][web:51]

Composite

- **Intent:** Compose objects into tree structures to represent part–whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.[web:41][web:43]
- **Structure:**
  - `Component` interface defining common operations.
  - `Leaf` representing simple elements with no children.
  - `Composite` representing complex elements that have children and implement operations by delegating to them.
- **Example (File system tree):**

```
interface FileSystemNode {
    void print(String indent);
}

class FileLeaf implements FileSystemNode {
    private final String name;

    FileLeaf(String name) { this.name = name; }

    public void print(String indent) {
        System.out.println(indent + "- " + name);
    }
}

class Directory implements FileSystemNode {
    private final String name;
    private final List<FileSystemNode> children = new ArrayList<>();

    Directory(String name) { this.name = name; }

    void add(FileSystemNode node) { children.add(node); }

    public void print(String indent) {
```

```
        System.out.println(indent + "+ " + name);
        for (FileSystemNode child : children) {
            child.print(indent + "  ");
        }
    }
}

// Usage:
Directory root = new Directory("root");
root.add(new FileLeaf("readme.txt"));
Directory src = new Directory("src");
src.add(new FileLeaf("Main.java"));
root.add(src);
root.print("");
```

- **When to use:** Menus/submenus, UI widget trees, organization hierarchies, file systems, or any part–whole structure where you want to treat a single object and a group of objects through the same interface.[web:41][web:43]

Decorator

- **Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.[web:41][web:51]
- **Structure:**
  - `Component` interface defining base operations.
  - `ConcreteComponent` with core behavior.
  - `Decorator` implementing `Component` and holding a `Component` reference.
  - `ConcreteDecorator` classes that add behavior before or after delegating to the wrapped component.
- **Example (Coffee with add-ons):**

```
interface Coffee {
    double cost();
    String description();
}
```

```
class BasicCoffee implements Coffee {
    public double cost() { return 50; }
    public String description() { return "Coffee"; }
}

abstract class CoffeeDecorator implements Coffee {
    protected final Coffee coffee;
    CoffeeDecorator(Coffee coffee) { this.coffee = coffee; }
}

class MilkDecorator extends CoffeeDecorator {
    MilkDecorator(Coffee coffee) { super(coffee); }
    public double cost() { return coffee.cost() + 10; }
    public String description() { return coffee.description() + ", Milk"; }
}

class SugarDecorator extends CoffeeDecorator {
    SugarDecorator(Coffee coffee) { super(coffee); }
    public double cost() { return coffee.cost() + 5; }
    public String description() { return coffee.description() + ", Sugar"; }
}

// Usage:
Coffee c = new SugarDecorator(new MilkDecorator(new BasicCoffee()));
System.out.println(c.description() + " costs " + c.cost());
```

- **When to use:** Adding optional features (logging, caching, compression, encryption) around core functionality, stacking multiple behaviors without creating many subclasses like `LoggedCachedService`, `CachedCompressedService`, etc.[web:41][web:51]

Facade

- **Intent:** Provide a unified, simplified interface to a set of interfaces in a subsystem. Facade reduces coupling and hides complexity behind a single entry point.[web:41][web:51]
- **Structure:**
  - Subsystem classes with complex APIs and dependencies among themselves.
  - `Facade` class that wraps and orchestrates subsystem calls, exposing a simpler high-level API to clients.

- **Example (Video conversion):**

```
class VideoFile { /* ... */ }
class Codec { /* ... */ }
class CodecFactory {
    Codec extract(VideoFile file) { /* ... */ return new Codec(); }
}
class BitrateReader {
    static VideoFile read(VideoFile file, Codec codec) { /* ... */ return file; }
    static VideoFile convert(VideoFile file, Codec codec) { /* ... */ return file; }
}
class AudioMixer {
    VideoFile fix(VideoFile file) { /* ... */ return file; }
}

// Facade
class VideoConverterFacade {
    public VideoFile convert(String fileName, String format) {
        VideoFile file = new VideoFile();
        CodecFactory factory = new CodecFactory();
        Codec sourceCodec = factory.extract(file);
        VideoFile buffer = BitrateReader.read(file, sourceCodec);
        VideoFile result = BitrateReader.convert(buffer, sourceCodec);
        AudioMixer mixer = new AudioMixer();
        return mixer.fix(result);
    }
}

// Usage:
VideoConverterFacade converter = new VideoConverterFacade();
VideoFile converted = converter.convert("movie.mkv", "mp4");
```

- **When to use:** Large subsystems (payment, messaging, search, media processing) where clients should not know internal details; creating "entry points" for layered architectures (API facade over domain, domain facade over infrastructure).[web:41][web:51][web:54]

Proxy

- **Intent:** Provide a surrogate or placeholder for another object to control access to it. Proxy and Decorator look similar structurally but differ in purpose: proxy focuses on access control, while decorator focuses on behavior extension.[web:41][web:45]
- **Structure:**

- `Subject` interface implemented by both `RealSubject` and `Proxy`.
- `RealSubject` doing the real work.
- `Proxy` holding a reference to `RealSubject`, managing its lifecycle or access, and forwarding calls as needed.

- **Example (Lazy-loading remote service):**

```
interface Image {
    void display();
}

class RealImage implements Image {
    private final String fileName;

    RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading " + fileName);
    }

    public void display() {
        System.out.println("Displaying " + fileName);
    }
}

class ImageProxy implements Image {
    private final String fileName;
    private RealImage realImage;

    ImageProxy(String fileName) { this.fileName = fileName; }

    public void display() {
        if (realImage == null) {
            realImage = new RealImage(fileName); // lazy load
        }
        realImage.display();
    }
}

// Usage:
Image img = new ImageProxy("photo.png");
img.display(); // loads + displays
```

```
img.display(); // only displays (already loaded)
```

- **When to use:** Lazy loading (virtual proxy), access control (protection proxy), remote calls (remote proxy), caching, logging around expensive or remote resources, without changing client code.[web:41][web:45][web:51]

3.4.3 Behavioral

Behavioral patterns characterize the ways objects collaborate and distribute responsibility. They define communication patterns and control flow between objects, enabling flexible and dynamic behavior.

Chain of Responsibility

- **Intent:** Pass a request along a chain of handlers where each handler either processes the request or forwards it to the next handler. This decouples senders from specific receivers and supports flexible routing of responsibilities.
- **Structure:**
  - `Handler` interface or abstract class with `setNext(Handler)` and `handle(Request)` methods.
  - `ConcreteHandler` classes that either handle the request or delegate to `next`.
  - A client that links handlers into a chain and submits requests to the head.
- **Example (Logging chain):**

```
abstract class Logger {
    private Logger next;

    public Logger setNext(Logger next) {
        this.next = next;
        return next;
    }

    public void log(String level, String message) {
        if (canHandle(level)) {
            write(message);
        } else if (next != null) {
            next.log(level, message);
        }
```

```
    }

    protected abstract boolean canHandle(String level);
    protected abstract void write(String message);
}

class InfoLogger extends Logger {
    protected boolean canHandle(String level) { return "INFO".equals(level); }
    protected void write(String message) { System.out.println("INFO: " + message); }
}

class ErrorLogger extends Logger {
    protected boolean canHandle(String level) { return "ERROR".equals(level); }
    protected void write(String message) { System.err.println("ERROR: " + message); }
}

// Usage:
Logger logger = new InfoLogger();
logger.setNext(new ErrorLogger());
logger.log("ERROR", "Something failed");
```

- **When to use:** Validation pipelines, middleware chains (auth, rate limiting), logging, approval workflows, or any request-processing pipeline where handlers can be added or reordered without changing the client.

Command

- **Intent:** Encapsulate a request as an object so you can parameterize clients with different requests, queue or log requests, and support undo/redo.
- **Structure:**
  - `Command` interface with a single method like `execute()`.
  - `ConcreteCommand` classes holding a reference to a `Receiver` and implementing `execute()` by calling receiver methods.
  - `Invoker` that triggers commands.
  - `Receiver` that performs the actual work.
- **Example (Remote control):**

```
interface Command {
    void execute();
```

```
}

class Light {
    void on()  { System.out.println("Light ON"); }
    void off() { System.out.println("Light OFF"); }
}

class LightOnCommand implements Command {
    private final Light light;
    LightOnCommand(Light light) { this.light = light; }
    public void execute() { light.on(); }
}

class RemoteControl {
    private Command button;
    void setCommand(Command command) { this.button = command; }
    void pressButton() { button.execute(); }
}

// Usage:
Light light = new Light();
Command onCommand = new LightOnCommand(light);
RemoteControl remote = new RemoteControl();
remote.setCommand(onCommand);
remote.pressButton();
```

- **When to use:** Task queues, job schedulers, audit logs (store commands as data), GUI actions (menu items, buttons), and undoable operations by storing inverse commands.

Iterator

- **Intent:** Provide a standard way to traverse elements of a collection without exposing its underlying representation.
- **Structure:**
  - `Iterator` interface with methods like `hasNext()` and `next()`.
  - `ConcreteIterator` implementing traversal logic.
  - `Aggregate` (collection) with a factory method to create iterators.
- **Example (Custom collection):**

```
interface MyIterator<T> {
    boolean hasNext();
    T next();
}

class NameCollection {
    private final String[] names = {"Alice", "Bob", "Charlie"};

    public MyIterator<String> iterator() {
        return new MyIterator<String>() {
            int index = 0;
            public boolean hasNext() { return index < names.length; }
            public String next() { return names[index++]; }
        };
    }
}

// Usage:
NameCollection c = new NameCollection();
MyIterator<String> it = c.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

- **When to use:** Custom collections or aggregate objects where you want multiple traversal strategies, and to hide internal representation (array, list, tree) from client code.

Observer

- **Intent:** Define a one-to-many dependency so that when one object (subject) changes state, all its dependents (observers) are notified and updated automatically.
- **Structure:**
  - `Subject` interface with `attach`, `detach`, and `notify` methods.
  - `Observer` interface with an `update()` method.
  - `ConcreteSubject` storing state and notifying observers on change.
  - `ConcreteObserver` reading state from subject and reacting.
- **Example (Weather station):**

```
interface Observer {
    void update(float temperature);
}

interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notifyObservers();
}

class WeatherStation implements Subject {
    private final List<Observer> observers = new ArrayList<>();
    private float temperature;

    public void setTemperature(float temp) {
        this.temperature = temp;
        notifyObservers();
    }

    public void attach(Observer o) { observers.add(o); }
    public void detach(Observer o) { observers.remove(o); }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(temperature);
        }
    }
}

class Display implements Observer {
    public void update(float temperature) {
        System.out.println("New temperature: " + temperature);
    }
}

// Usage:
WeatherStation station = new WeatherStation();
station.attach(new Display());
station.setTemperature(30.5f);
```

- **When to use:** Event systems, pub–sub, UI bindings, cache invalidation listeners, and any scenario where multiple parts of the system must react to changes in shared state without tight coupling.

State

- **Intent:** Allow an object to change its behavior when its internal state changes; the object appears to change its class.
- **Structure:**
  - `State` interface declaring behavior methods (for example, domain-specific actions).
  - `ConcreteState` classes implementing behavior for each state.
  - `Context` holding a reference to a `State` and delegating calls to it; the state can switch the context's current state.
- **Example (Order lifecycle):**

```
interface OrderState {
    void next(OrderContext ctx);
    void cancel(OrderContext ctx);
}

class NewState implements OrderState {
    public void next(OrderContext ctx) { ctx.setState(new ShippedState()); }
    public void cancel(OrderContext ctx) { ctx.setState(new CancelledState()); }
}

class ShippedState implements OrderState {
    public void next(OrderContext ctx) { ctx.setState(new DeliveredState()); }
    public void cancel(OrderContext ctx) {
        System.out.println("Cannot cancel, already shipped");
    }
}

class CancelledState implements OrderState {
    public void next(OrderContext ctx) { /* no-op */ }
    public void cancel(OrderContext ctx) { /* no-op */ }
}

class DeliveredState implements OrderState {
    public void next(OrderContext ctx) { /* no-op */ }
    public void cancel(OrderContext ctx) { /* no-op */ }
}

class OrderContext {
    private OrderState state = new NewState();
```

```
    void setState(OrderState state) { this.state = state; }
    void next() { state.next(this); }
    void cancel() { state.cancel(this); }
}
```

- **When to use:** Objects with clear finite states (order, ticket, document workflow) where behavior depends on state, especially when you want to avoid big `switch`/`if-else` blocks on an enum.

Strategy

- **Intent:** Define a family of algorithms, encapsulate each one, and make them interchangeable so the algorithm can vary independently from clients using it.
- **Structure:**
  - `Strategy` interface declaring the algorithm method (for example, `pay`, `compress`, `sort`).
  - `ConcreteStrategy` classes implementing different algorithms.
  - `Context` holding a `Strategy` reference and delegating work to it.
- **Example (Payment strategy):**

```
interface PaymentStrategy {
    void pay(double amount);
}

class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paying " + amount + " via Credit Card");
    }
}

class UpiPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paying " + amount + " via UPI");
    }
}

class PaymentContext {
    private PaymentStrategy strategy;
    PaymentContext(PaymentStrategy strategy) { this.strategy = strategy; }
```

```
    void setStrategy(PaymentStrategy strategy) { this.strategy = strategy; }
    void checkout(double amount) { strategy.pay(amount); }
}

// Usage:
PaymentContext ctx = new PaymentContext(new CreditCardPayment());
ctx.checkout(1000);
ctx.setStrategy(new UpiPayment());
ctx.checkout(500);
```

- **When to use:** Multiple interchangeable algorithms (discounts, pricing, routing, serialization) and to replace conditional logic (`if (type == A) ... else if (type == B) ...`) with polymorphism.

Template Method

- **Intent:** Define the skeleton of an algorithm in a base class, deferring some steps to subclasses so they can vary parts of the algorithm without changing its overall structure.
- **Structure:**
  - `AbstractClass` with a `templateMethod()` containing the high-level algorithm steps.
  - Abstract "hook" methods that subclasses must implement (or optionally override).
  - `ConcreteClass` subclasses implementing the variable steps.
- **Example (Data processor):**

```
abstract class DataProcessor {

    // template method
    public final void process() {
        readData();
        transformData();
        writeData();
    }

    protected abstract void readData();
    protected abstract void transformData();
    protected abstract void writeData();
}
```

```
class CsvDataProcessor extends DataProcessor {
    protected void readData() { System.out.println("Read CSV"); }
    protected void transformData() { System.out.println("Transform CSV data"); }
    protected void writeData() { System.out.println("Write CSV"); }
}

class JsonDataProcessor extends DataProcessor {
    protected void readData() { System.out.println("Read JSON"); }
    protected void transformData() { System.out.println("Transform JSON data"); }
    protected void writeData() { System.out.println("Write JSON"); }
}

// Usage:
DataProcessor p = new CsvDataProcessor();
p.process();
```

- **When to use:** Algorithms with fixed high-level steps but varying details (report generation, ETL, parsing), where you want to enforce a standard workflow while allowing customizations in certain steps.

4 LLD Interview Template

Use this 6–step approach for every problem (write it like a checklist):

1. Clarify requirements and constraints.
2. Identify entities/classes.
3. Define attributes and relationships.
4. Define methods/behaviors and main flows.
5. Apply SOLID principles and suitable patterns.
6. Walk through main flows and edge cases end-to-end.

5 Core Practice Problems

5.1 Parking Lot System

- Use the 6–step template to design a Parking Lot (multiple iterations).
- Focus on:
  - Vehicle types, parking spots, tickets, payment.
  - Strategy pattern for pricing or allocation if needed.
- Implement core classes in Java (no DB/UI).

5.2 Library Management System

- Entities: Book, Member, Loan, Fine, Librarian.
- Core operations:
  - Search book, issue book, return book, calculate fine.
- Emphasize relationships and responsibilities between classes.

5.3 Simple Game Designs

- Tic Tac Toe or Snake & Ladder.
- Focus on clean modeling and game flow rather than complex algorithms.
- Implement a fully working Java version to build confidence.

5.4 Splitwise-Style Expense Sharing

- Clarify: groups, expenses, different split types (equal, exact, percentage).
- Key classes: User, Group, Expense, Split; use Strategy for split type behavior.
- Implement core flows and briefly note how persistence would be handled.

5.5 Movie Ticket Booking (BookMyShow)

- Entities: Theatre, Screen, Show, Seat, Booking, Payment.
- Use patterns:
  - Factory for payment methods.
  - Strategy for pricing/discounts.
- Implement flows: search show, select seat, lock seats, confirm, pay; think about seat clashes and locks.

5.6 Online Shopping Cart / E-commerce Cart

- Entities: User, Product, Cart, CartItem, Order, Payment, Inventory.
- Focus on extensibility:
  - Discount strategies.
  - Multiple payment options.
- Model how cart transitions to order and how inventory updates.

6 Advanced Topics and Depth

6.1 Concurrency and Edge Cases in LLD

- Think about thread safety for bookings, counters, shared resources.
- For Parking Lot / BookMyShow:
  - Identify race conditions (same seat, same spot, same resource).
  - Consider locking, optimistic concurrency, or queues as solutions.

6.2 Additional Practice Problems

Pick 2–3 extra problems and solve them under 45–60 minutes using the same 6–step template:

- Elevator System.
- Notification Service (Email/SMS/Push with retries, priority).
- LRU Cache (combine DSA with LLD: interfaces plus internal data structures).

7 Mock Interviews and Revision

7.1 Mock Rounds

- Choose a fresh problem each session (e.g., Food Delivery, Ride Sharing, Meeting Scheduler).
- Timebox: 45–60 minutes.
- Practice speaking as if in a real interview:
  - Requirements → entities → diagram → flows → patterns → edge cases.

7.2 Polishing and Summaries

- Revisit 3–4 core designs: Parking Lot, Splitwise, BookMyShow, Shopping Cart.
- For each, prepare:
  - A 1–2 minute "elevator pitch" explanation.
  - A list of reusable patterns and standard sentences you can use in interviews (e.g., how you apply SRP, where Strategy fits, how you handle concurrency).