Design patterns are a foundational concept in **Low-Level Design (LLD)** that help create scalable, maintainable, and extensible software systems. Let's break down your question into structured parts for a very detailed and complete understanding.

🔷 What Are Design Patterns in Low-Level Design (LLD)?

Design patterns are **reusable solutions to common software design problems** that occur frequently in real-world application development. In Low-Level Design, design patterns help structure classes, objects, and their interactions in a way that meets both functional and non-functional requirements.

Think of them like **blueprints** that developers can customize to solve a particular design issue in their codebase.

🔷 Why Do We Need Design Patterns in LLD?

✅ Benefits / Reasons:

1. **Reusability**: Solve problems using proven methods.
2. **Maintainability**: Code becomes easier to read and update.
3. **Scalability**: Helps build systems that are easy to extend and scale.
4. **Communication**: Acts as a common vocabulary among developers (e.g., “use Strategy Pattern here”).
5. **Avoids Anti-Patterns**: Helps avoid bad practices and poor design.
6. **Faster Development**: Leverage pre-existing patterns instead of reinventing solutions.

🔷 Types and Classifications of Design Patterns

Design patterns are primarily classified into **three main categories** based on their purpose:

1. Creational Patterns (How objects are created)

- Focus: Object instantiation process
- Goal: Make system independent of how objects are created, composed, and represented

2. Structural Patterns (How objects are composed)

- Focus: Class and object composition
- Goal: Ensure classes and objects work together even if they weren’t designed to do so

3. Behavioral Patterns (How objects interact)

- Focus: Communication between objects
- Goal: Help manage complex flow of control and data

🔷 Detailed Sub-classification with Use-Cases, Pros and Cons

🔶 1. Creational Design Patterns

Pattern	Description	Use-Case	Pros	Cons
Singleton	Ensures a class has only one instance and provides a global point of access	Logger, Configuration, DB connection	Controlled access, Lazy loading	Difficult to test, Global state
Factory Method	Defines an interface for creating an object but lets subclasses alter the type of objects that will be created	Creating shapes, GUI components	Decouples client from implementation, Easy to add new types	More classes created
Abstract Factory	Provides an interface to create families of related or dependent objects	UI themes, Cross-platform UI toolkits	Enforces consistency among products	Complex to implement
Builder	Separates object construction from its representation	Complex object creation like Pizza, Document, Car	Step-by-step construction, Immutable objects	More code overhead
Prototype	Clone objects without coupling to their specific classes	When object creation is costly (e.g., DB object)	Reduces cost of object creation	Complex with deep copies

---

🔶 2. Structural Design Patterns

Pattern	Description	Use-Case	Pros	Cons
Adapter	Allows incompatible interfaces to work together	Integrating legacy code with new systems	Reuse existing code	Adds extra layer
Bridge	Separates abstraction from implementation	Different UIs for different platforms	Increases flexibility	Increases complexity
Composite	Treats individual objects and compositions uniformly	File systems, GUI components	Easy to add new elements	Can make code harder to understand

Pattern	Description	Use-Case	Pros	Cons
Decorator	Adds responsibilities to objects dynamically	Logging, UI enhancements	Flexible alternative to subclassing	Many small objects
Facade	Provides simplified interface to a complex system	APIs, SDKs	Hides complexity, Easier to use	Can become a "god" object
Flyweight	Reduces memory usage by sharing objects	Game characters, Icons	Saves memory	Hard to manage shared state
Proxy	Provides a placeholder for another object to control access	Virtual proxy, Security proxy	Adds security, Lazy loading	Adds complexity, latency

🔶 3. Behavioral Design Patterns

Pattern	Description	Use-Case	Pros	Cons
Strategy	Encapsulates interchangeable algorithms	Sorting algorithms, Payment processing	Easy to switch algorithms	More objects, complexity
Observer	One-to-many dependency so when one object changes, others are notified	Event handling, Notification system	Loose coupling	Cascade updates
Command	Encapsulates a request as an object	Undo/Redo, GUI buttons	Supports queues, undoable actions	High number of classes
State	Allows an object to alter its behavior when its internal state changes	Traffic lights, UI component states	Localizes behavior changes	State explosion
Chain of Responsibility	Passes request along a chain of handlers	Logging, Authentication	Decouples sender and receiver	Hard to trace logic
Template Method	Defines the skeleton of an algorithm in a method	Payment flow, Game turns	Reuse code	Inflexible for complex scenarios
Iterator	Provides a way to access elements without exposing the underlying structure	Collections, Custom traversals	Unified access	Not suitable for parallel processing
Mediator	Reduces chaotic dependencies between objects	Chat system, Air traffic control	Reduces complexity	Central mediator can become complex

Pattern	Description	Use-Case	Pros	Cons
Memento	Captures and restores object’s internal state	Undo in text editors	Snapshot of object state	High memory cost
Visitor	Allows new operations to be added without changing the elements	Syntax tree processing	Easy to add operations	Hard to add new element classes
Interpreter	Defines grammar and interpretation of a language	SQL parsing, Expression evaluation	Simple grammar	Not efficient for complex languages

🔷 Visual Hierarchy

```
Design Patterns
├── Creational
│   ├── Singleton
│   ├── Factory Method
│   ├── Abstract Factory
│   ├── Builder
│   └── Prototype
├── Structural
│   ├── Adapter
│   ├── Bridge
│   ├── Composite
│   ├── Decorator
│   ├── Facade
│   ├── Flyweight
│   └── Proxy
└── Behavioral
    ├── Strategy
    ├── Observer
    ├── Command
    ├── State
    ├── Chain of Responsibility
    ├── Template Method
    ├── Iterator
    ├── Mediator
    ├── Memento
    ├── Visitor
    └── Interpreter
```

🔷 Summary Table

Type	Focus	Key Benefit
Creational	Object creation mechanisms	Flexibility in object creation
Structural	Class/object composition	Code reusability & scalability
Behavioral	Object communication	Better flow and responsibility separation