

i.Creational DP

- a.Singleton dp-Eager & Lazy initialization
- b.Prototype dp
- c.Factory dp
- d.Builder dp
- e.Abstract Factory dp

ii.Structural DP

- a.Decorator dp
- b.Composite dp
- c.Proxy dp



iii.Behavioral DP




- a.Chain of Responsibility dp
- b.Command dp
- c.Observer dp
- d.State dp
- e.Strategy dp

can you please tell their structure & draw uml diagram & differentiate them in tabular form & tell their pros & cons & tell when & how to use them?



To provide a thorough breakdown of these design patterns, I'll first give a **tabular comparison** between them, including **structure, UML diag.** ↓ **s, pros and cons, and when and how to use them.**

I. Creational Design Patterns

Pattern	Structure	UML Diagram	Pros	Cons	When to Use
Singleton (Eager & Lazy)	Ensure a class has only one instance and provides a global point of access.	 Singlet on UML	Eager: Simple to implement. Lazy: Instance only when needed.	Eager: Memory overhead. Lazy: Thread synchronization issues.	When only one instance of a class is needed to coordinate actions across a system (e.g., logger, configuration manager).
Prototype	Clone existing instances to create new ones.	 Prototy pe UML	Reduces object creation cost, good for expensive creation.	Cloning can be complex, deep vs shallow copy issues.	When object creation is expensive and similar objects are created frequently.

Factory Method	Create objects without exposing the creation logic to the client.	 Factory UML	Loose coupling, flexibility in subclassing.	More classes can lead to complexity.	When the class needs to delegate the instantiation of objects to subclasses.
Builder	Build complex objects step-by-step.	 Builder UML	Easier to create complex objects, improves readability.	Requires additional classes for director and builder.	When an object requires many parts to be created step by step (e.g., constructing a large, complex object like a house).
Abstract Factory	Provide an interface for creating families of related objects without specifying their concrete classes.	 Abstract Factory UML	Ensures related objects are used together, promotes loose coupling.	Complexity increases with the number of product families.	When you need to create families of related objects (e.g., different OS-specific UI components).

II. Structural Design Patterns

Pattern	Structure	UML Diagram	Pros	Cons	When to Use
Decorator	Adds behavior or responsibilities to an individual object dynamically.	 Decorator UML	Flexible, allows for extending functionality at runtime.	Can introduce complexity with many decorators.	When you want to dynamically add responsibilities to objects without modifying the existing code (e.g., adding scrollbars to windows).
Composite	Composes objects into tree-like structures to represent part-whole hierarchies.	 Composite UML	Simplifies client code, treats individual objects and compositions uniformly.	Complexity can arise from tree structures.	When you need to represent a tree structure (e.g., a file system, GUI components like menus).

Proxy

Provide a surrogate or placeholder for another object to control access to it.








Controls access to objects, can provide lazy initialization.

Adds complexity due to indirection.

When you need to control access to an object (e.g., remote proxy, virtual proxy for memory optimization).

III. Behavioral Design Patterns

Pattern	Structure	UML Diagram	Pros	Cons	When to Use
Chain of Responsibility	Passes a request along a chain of handlers.	 Chain of Responsibility UML	Decouples sender and receiver.	Can be hard to debug, no guarantee the request will be handled.	When multiple objects can handle a request and the handler isn't determined until runtime.
Command	Encapsulates a request as an object.	 Command UML	Supports undo/redo, decouples sender and receiver.	Can result in a large number of classes.	When you need to parametrize objects with operations (e.g., GUI button click actions).

Observer	Defines a one-to-many dependency where changes in one object are automatically notified to others.	 Observer UML	Promotes loose coupling, automatic update of dependent objects.	Can lead to memory leaks if observers are not properly managed.	When multiple objects need to be notified of changes in one object (e.g., event systems, notifications).
State	Allows an object to change its behavior when its state changes.	 State UML	Simplifies state-based behavior.	Can result in many state classes.	When an object needs to change its behavior based on internal state (e.g., a vending machine's various states).
Strategy	Encapsulates a family of algorithms, allowing them to be interchangeable.	 Strategy UML <div>↓</div>	Flexibility in choosing algorithms, avoids large conditional statements.	Increased number of classes, client must be aware of strategies.	When you need multiple algorithms for a specific task (e.g., sorting algorithms).

Detailed Description and Use Cases

1. Singleton Pattern:

- **Eager Initialization:** Create the instance when the class is loaded. Suitable when the singleton will be used immediately.
- **Lazy Initialization:** Only create the instance when it's first needed, ensuring no unnecessary memory usage. Suitable for resource-heavy singletons that are not always needed.

2. Prototype Pattern:

- Use when object creation is resource-intensive (e.g., network objects, database connections) or when similar objects are frequently needed. Cloning them saves time and resources.

3. Factory Method Pattern:

- Use when the exact type of object to be created is not known until runtime (e.g., GUI components in a cross-platform application).

4. Builder Pattern:

- Useful when constructing complex objects, especially when some fields are optional or should be created in a specific order (e.g., building a car or assembling a complex user profile object).

5. Abstract Factory Pattern:

- Best used when you need to ensure that families of related objects are created together, ensuring consistency (e.g., creating a UI theme with consistent styles across components).
-

6. Decorator Pattern:

- Ideal for adding new responsibilities to objects dynamically, such as extending functionalities of user interface elements like buttons or scroll bars.

7. Composite Pattern:

- Use when you need to treat individual objects and compositions of objects uniformly (e.g., representing a menu with submenus and items).

8. Proxy Pattern:

- Use when you need to control access to an object, either for security reasons or to reduce memory usage (e.g., lazy loading images in a large dataset).

9. Chain of Responsibility Pattern:

- Suitable when multiple handlers can process the request, but the specific handler is not known until runtime (e.g., event-handling systems, request validation).

10. Command Pattern:

- Use when you need to decouple the invoker from the object that performs the action, making it easier to support undo/redo functionality (e.g., text editor commands, GUI actions).

11. Observer Pattern:

- Use in scenarios where you need multiple objects to react to the state changes of another object (e.g., MVC architectures, notification systems).

12. State Pattern:

- Best used when an object needs to behave differently based on its internal state (e.g., a traffic light system or vending machine).

13. Strategy Pattern:

- Use when you need to choose between multiple algorithms at runtime (e.g., sorting algorithms or payment processing strategies).