

Design Patterns

Design Patterns – Quick Reference Notes

1. What Is a Design Pattern?

Aspect	Description
Definition	A reusable solution to a common software design problem. It describes <i>what</i> to do, not <i>how</i> to code it.
Purpose	Improves code readability, promotes best practices, and speeds up development by providing proven templates.
Scope	Architectural (e.g., MVC), creational, structural, and behavioral patterns.

2. Classification of Patterns

Category	Goal	Typical Use Cases	Common Patterns
Creational	Object creation mechanisms that increase flexibility and reuse.	When object creation is complex, needs to be delayed, or must vary at runtime.	Singleton, Factory Method, Abstract Factory, Builder, Prototype
Structural	Organizing classes and objects to form larger structures while keeping them flexible.	When you need to compose objects, hide implementation details, or adapt interfaces.	Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
Behavioral	Managing algorithms, communication, and responsibilities among objects.	When you need clear interaction protocols, state-dependent behavior, or decoupled communication.	Observer, Strategy, Command, Iterator, Mediator, Memento, State, Template Method, Visitor, Chain of Responsibility

3. Creational Patterns

3.1 Singleton

Design Patterns

Ensures a class has only one instance and provides a global access point.

- **When to use:** Shared resources (e.g., logger, configuration).
- **Key points:** Private constructor, static instance, lazy vs. eager initialization, thread?safe implementation.

3.2 Factory Method

Defines an interface for creating an object, but lets subclasses decide which class to instantiate.

- **When to use:** Class creation should be delegated to subclasses.
- **Key points:** Creator class with `abstract factoryMethod()`, concrete `ConcreteCreator` overrides it.

3.3 Abstract Factory

Provides an interface for creating families of related objects without specifying concrete classes.

- **When to use:** Multiple related products (e.g., UI widgets for different OS).
- **Key points:** `AbstractFactory` with methods for each product, concrete factories implement them.

3.4 Builder

Separates construction of a complex object from its representation.

- **When to use:** Objects with many optional parameters or step?wise construction.
- **Key points:** `Builder` interface, concrete builders, `Director` orchestrates building, fluent API for chaining.

3.5 Prototype

Creates new objects by copying an existing instance (prototype).

- **When to use:** Object creation is expensive; you need many similar objects.
 - **Key points:** Implement `clone()` (shallow vs. deep copy), prototype registry.
-

Design Patterns

4. Structural Patterns

4.1 Adapter

Converts one interface into another that a client expects.

- **When to use:** Integrating incompatible third-party libraries.
- **Key points:** Class adapter (multiple inheritance) vs. object adapter (composition).

4.2 Bridge

Decouples an abstraction from its implementation so the two can vary independently.

- **When to use:** Multiple dimensions of variation (e.g., UI + platform).
- **Key points:** Abstraction holds a reference to Implementor.