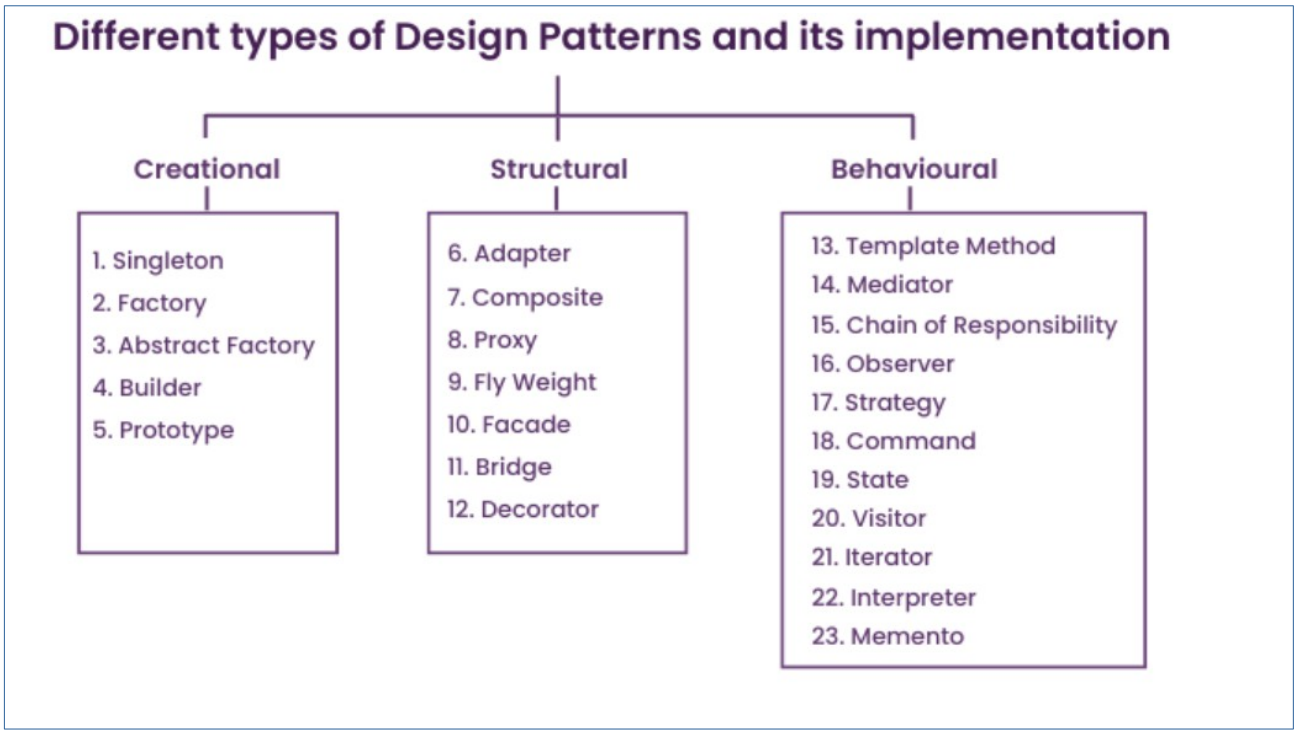
Design Patterns sind wiederverwendbare Lösungen für häufig auftretende Probleme in der Softwareentwicklung. Sie werden in verschiedene Kategorien unterteilt. Hier ist eine Übersicht:



1. Creational Patterns (Erzeugungsmuster)

Diese Muster befassen sich mit der **Erzeugung von Objekten** und bieten Mechanismen, um flexible und effiziente Objektinstanziierungen zu ermöglichen.

1. **Singleton**:
    - Stellt sicher, dass eine Klasse nur eine einzige Instanz hat und bietet einen globalen Zugriffspunkt darauf.
2. **Factory Method**:
    - Bietet eine Schnittstelle zur Erzeugung von Objekten, lässt aber Unterklassen entscheiden, welche Klasse instanziiert wird.
3. **Abstract Factory**:
    - Erzeugt eine Familie verwandter oder abhängiger Objekte, ohne deren konkrete Klassen anzugeben.
4. **Builder**:
    - Trennt die Konstruktion eines komplexen Objekts von seiner Repräsentation, sodass dasselbe Erstellungsverfahren unterschiedliche Objektdarstellungen erzeugen kann.
5. **Prototype**:
    - Erzeugt neue Objekte durch **Kopieren (Cloning)** eines bestehenden Objekts, anstatt dieses neu zu instanziieren.

2. Structural Patterns (Strukturmuster)

Diese Muster befassen sich mit der **Komposition von Klassen und Objekten**, um neue Funktionalitäten zu schaffen oder bestehende zu verbessern.

6. **Adapter**:
    - Konvertiert die Schnittstelle einer Klasse in eine andere, die ein Client erwartet. Es ermöglicht das Zusammenspiel von Klassen, die sonst inkompatibel wären.
7. **Composite**:
    - Ermöglicht es, Objekte in **Baumstrukturen** zu organisieren, um Teil-Ganzes-Hierarchien darzustellen. Ein Client kann mit einzelnen Objekten und Kompositionen gleich umgehen.
8. **Proxy**:
    - Stellt ein Stellvertreterobjekt bereit, das den Zugriff auf ein anderes Objekt kontrolliert (z. B. für Zugriffskontrolle, Caching oder Lazy Initialization).
9. **Flyweight**:
    - Minimiert die Speicherverwendung, indem wiederkehrende Objekte geteilt werden, wenn sie mehrfach verwendet werden, anstatt sie neu zu erstellen.
10. **Facade**:

- Bietet eine einfache Schnittstelle zu einem **komplexen Subsystem**, indem es viele Schnittstellen in einer einzigen vereinheitlicht.

11. **Bridge**:

- Trennt die **Abstraktion** einer Klasse von ihrer Implementierung, sodass beide unabhängig voneinander verändert werden können.

12. **Decorator**:

- Fügt einem Objekt zur Laufzeit dynamisch zusätzliche Funktionalitäten hinzu, indem es das Objekt in eine Dekorator-Klasse "verpackt".

3. Behavioral Patterns (Verhaltensmuster)

Diese Muster befassen sich mit der **Kommunikation zwischen Objekten** und der Zuweisung von Verantwortlichkeiten zwischen Objekten.

13. **Template Method**:

- Definiert das **Gerüst eines Algorithmus** in einer Methode, delegiert aber einige Schritte an Unterklassen, ohne die Struktur des Algorithmus zu ändern.

14. **Mediator**:

- Kapselt die Kommunikation zwischen Objekten in einem **zentralen Mediator**, um direkte Abhängigkeiten zwischen Objekten zu vermeiden.

15. **Chain of Responsibility**:

- Organisiert mehrere Objekte in eine Kette, durch die eine Anfrage weitergereicht wird, bis eines der Objekte die Anfrage verarbeitet.

16. **Observer**:

- Definiert eine **1**
  ängigkeit zwischen Objekten, sodass eine Änderung eines Objekts automatisch alle abhängigen Objekte informiert und aktualisiert.

17. **Strategy**:

- Definiert eine **Familie von Algorithmen**, die austauschbar sind. Der Algorithmus wird zur Laufzeit ausgewählt, ohne die Client-Klasse zu ändern.

18. **Command**:

- Kapselt eine Anforderung als Objekt, das Parameter unterstützt und später aufgerufen, protokolliert oder wiederholt werden kann.

19. **State**:

- Ermöglicht einem Objekt, sein Verhalten zu ändern, wenn sich sein **internes Zustand** ändert, sodass das Objekt wie eine andere Klasse aussieht.

20. **Visitor**:

- Ermöglicht es, eine neue Operation auf Objekten einer Struktur hinzuzufügen, ohne deren Klassen zu ändern.

21. **Iterator**:

- Bietet eine **Möglichkeit, über Elemente** einer Sammlung (z. B. Liste oder Array) zu iterieren, ohne die interne Darstellung der Sammlung offenzulegen.

22. **Interpreter**:

- Definiert eine Grammatik zur Darstellung von Problemen und stellt einen **Interpreter bereit**, um diese Probleme zu lösen.

23. **Memento**:

- Ermöglicht es, den internen Zustand eines Objekts zu speichern und später wiederherzustellen, ohne die Kapselung zu brechen.

Diese Design Patterns bieten bewährte Lösungen für verschiedene wiederkehrende Probleme in der Softwareentwicklung und werden häufig verwendet, um die **Wartbarkeit, Flexibilität und Erweiterbarkeit** von Code zu verbessern.