**Design Patterns**

* Design Patterns are typical reusable solutions to commonly occurring problems in software design.
* Design Patterns are not finished code but templates or blueprints only that we can customize to solve a recurring design problem in our code.
* The pattern is not a specific piece of code, but a general concept for solving a particular problem. We can follow the pattern details & implement a solution that suits the realities of our own program.

**Design Pattern Vs Algorithm**

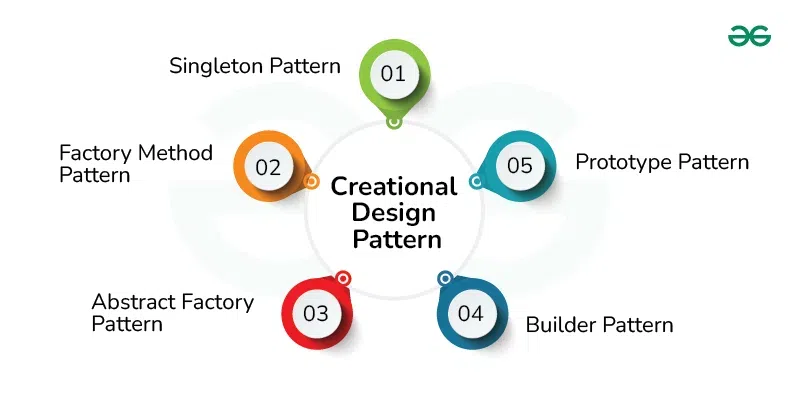
* Both concepts describe typical solutions to some known problems.
* An Algorithm always defines a clear set of actions that can achieve some goal.
* While a Pattern is more high-level description of a solution.

**Classification of Patterns**

* Design Patterns differ by their complexity, level of detail & scale of applicability to the entire system being designed.
* For e.g., Road Construction – We can make an intersection safer by either installing some traffic lights or building an entire multi-level interchange with underground passages for pedestrians.
* The most basic & low-level patterns are often called **Idioms**. They usually apply only to a single programming language.
* The most universal & high-level patterns are Architectural patterns. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.

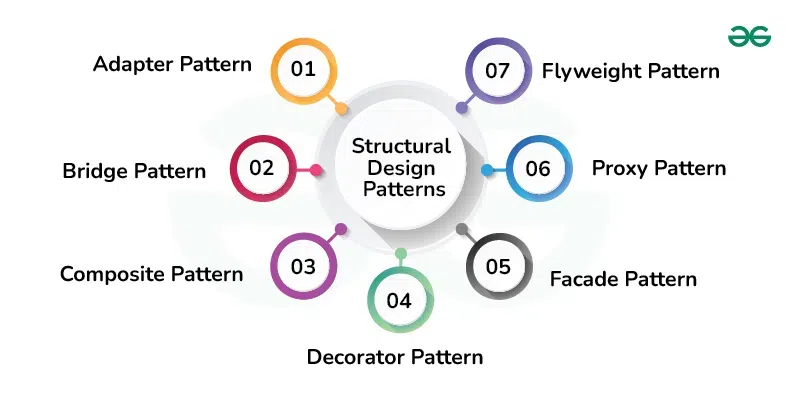
**1. Creational Patterns**

* They provide object creation mechanisms that increase flexibility & reuse of existing code.
* **Creational Design Patterns** focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed & represented.



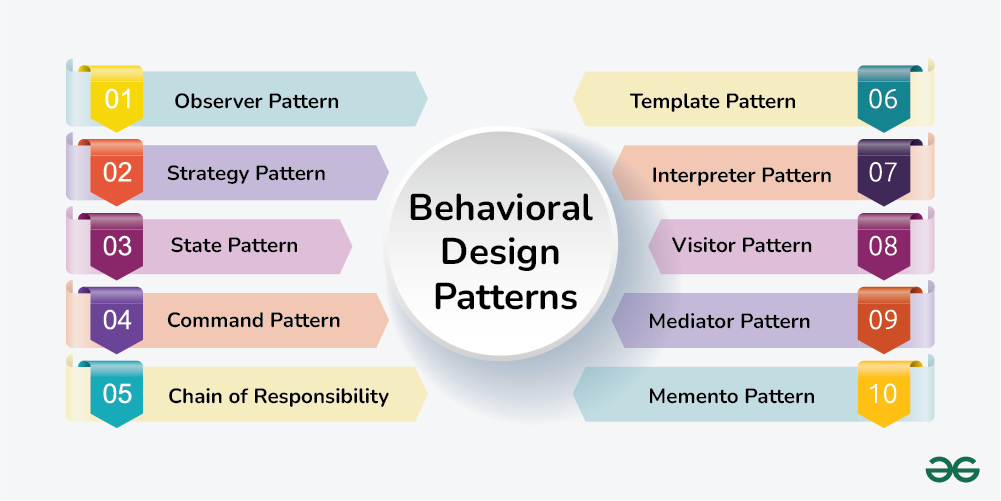
**2. Structural Patterns**

* They explain how to assemble objects & classes into larger structures, while keeping these structure flexible & efficient.
* Structural Design Patterns solves problems related to how classes & objects are composed/assembled to form larger structures which are efficient & flexible in nature.
* They use inheritance to compose interfaces or implementations.



**3. Behavioral Patterns**

* They take care of effective communication & the assignment of responsibilities b/w objects.
* Behavioral Design Patterns are concerned with algorithms & the assignment of responsibilities b/w objects.
* They describe not just patterns of objects or classes but also the patterns of communication b/w them.
* These patterns characterize complex control flow that’s difficult to follow at run-time.



**Creational Design Patterns**

**1. Factory Method Design Pattern**

* Factory method design pattern provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

2. Abstract Factory Method Design Pattern

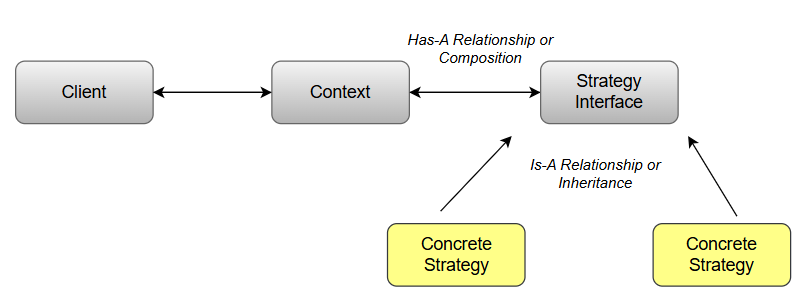
3. Builder Design Pattern

* Builder design pattern lets us construct complex objects step by step. This pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

**Strategy Design Pattern**

* Strategy Design pattern is a **behavioral Design pattern** that lets us dynamically choose or change the behavior of an object by encapsulating it into different strategies rather than sticking with one.
* In simpler terms, this pattern provides a way to extract the behavior of an object into separate classes that can be swapped in & out at runtime.
* This enable the object to be more flexible & reusable, as different strategies can be easily added or modified without changing the client’s core code.
* It is based on the principle of Composition over Inheritance.

**Components of Strategy Design Pattern**



**Note:**

* Communications b/w the components happen in a structured & decoupled manner i.e.; the context is not required to be aware of the exact behavior of each strategy.
* As long as they follow the same interface, strategies can be switched without affecting the client or other strategies.

**1. Client**

* Client is responsible for configuring the appropriate strategy based on the requirement & providing it to the context.
* It creates an instance of the desired concrete strategy & passes it to the context, enabling the context to use the selected strategy to perform the task.

**2. Context**

* Context acts as a mediator between the clients & strategies and it maintains a reference to a strategy object & calls its methods to perform the task without exposing the actual strategies behavior to client.

**3. Strategy Interface**

* Strategy interface enables decoupling between the context & the specific/concrete strategies by ensuring that all the strategies follow the same set of rules & can be interchangeable used by the context.
* It can be an abstract class or interface that specifies a set of methods that all concrete strategies must implement.

**4. Concrete Strategies**

* Concrete strategies are the various implementations of the Strategy interface with each concrete strategy defining a specific algorithm or behavior to the task/method defined by the Strategy interface.
* They are interchangeable & can be selected by the client based on the task requirement.

**Use cases**

* Avoid Code Duplication – Suppose multiple concrete classes have same functionality, then they can be encapsulated to one generic strategy class.
* Multiple algorithms: e.g., Sorting algorithm – Different sorting algos can be encapsulated into separate strategies & passed to an object that needs sorting.
* Encapsulating algorithms
* Runtime selection
* Reducing conditional statements
* Testing & Extensibility
* Validation rules
* Text formatting
* Database access
* Payment strategy

**Benefits**

* Improved code flexibility
* Better code reusability
* Encourages better coding practices
* Simplifies testing

**Disadvantages**

* The application must be aware of all the strategies to select the right one for the right solution.
* The Strategy interface defines a set of features, some of which might be not relevant for some concrete strategies. ----> *Violating the Liskov Substitution Principle* (its solution can be used)
* **Imp:** In most cases, the client/application configures the context with the required strategy object. Therefore, the client needs to create & maintain 2 objects instead of one.

**Best Practices for implementing the Strategy Design Pattern**

* Keep the interface simple & focused on a single responsibility
* Encapsulate any stateful behavior in the concrete strategy classes, rather than in the context class.
* Use Dependency injection to pass the concrete strategy to the context class, rather than creating it directly in the context class.
* Use an enum or a factory class to provide a centralized place for creating & managing concrete strategy objects.

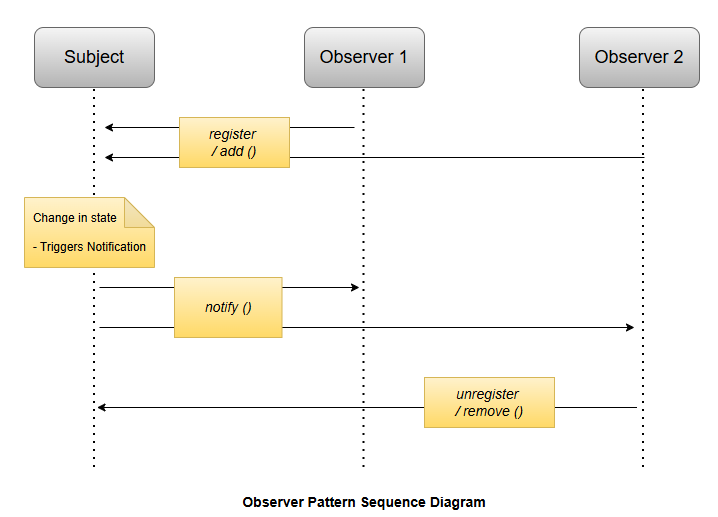
Example 1: Vehicle

|  |
| --- |
| **Scenarios:**  Suppose we’ve a “Vehicle” class with drive () method & multiple classes extend Vehicle class & according to their requirement, drive () method is being overridden by these inheritors.   * class **Vehicle**: drive () method – Sysout (“Normal Drive Capability”) * class **SportsVehicle** extends **Vehicle**: Overrides drive () method – Sysout (“Sports Capability”) * class **PassengerVehicle** extends **Vehicle**: uses parent Vehicle class drive () method definition * class **OffRoadVehicle** extends **Vehicle**: Overrides drive () method – Sysout (“Sports Capability”) * class **GoodsVehicle** extends **Vehicle**: uses parent Vehicle class drive () method definition     **Problem:**  Above, we can see SportsVehicle & OffRoadVehicle classes have common definition that is resulting in code duplication.  **Solution:**  To avoid code duplication, we can use Strategy Design pattern.  **Step 1:** Client class will provide the Vehicle Context object.  **Note:** In most of the cases client provides the vehicle context object along with Strategy object but in this specific example, the VehicleContext extensions are providing the actual Strategy object.  **Step 2:** Based on the Vehicle Context object, a strategy object will be returned.  **Step 3:** Based on the strategy object, the specific method/behavior will be called. |

**Observer Design Pattern (Publish-Subscribe Pattern)**

* Observer Design pattern is a behavioral design pattern that lets us define a subscription mechanism to notify multiple objects (called Observers) about any changes in the state of the object they’re observing i.e., one to many dependencies between objects: Observable & Observers.
* In this pattern, many observers (subscriber objects) observe a particular subject (publisher object). Observers register with a subject to be notified when a change is made inside that subject.
* *These objects are loosely coupled* as an observer object can register or unregister from a subject at any point of time.

**Components of Observer Design Pattern**



**1. Subject / Observable / Publisher**

* Subject maintains a list of observers & also provides method to register, unregister & notify observers about any changes in the state of subject / observable / publisher object.
* It can be interface or abstract class.

**2. Concrete Subject**

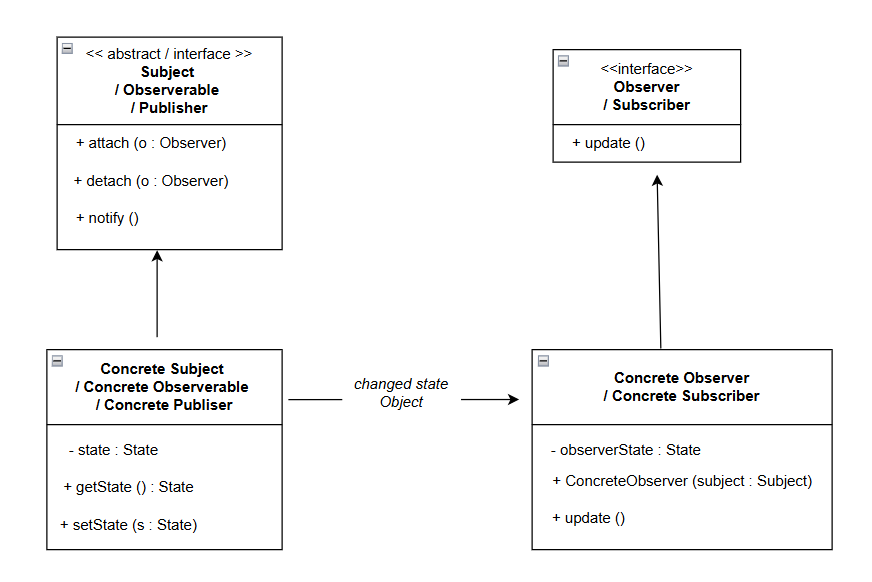
* Concrete subjects are specific implementations of the subject interface.
* They hold the actual state or data under observation & notify the subscribed observers about any changes in the state.
* E.g., if a Weather Station is the Subject, Specific Weather Stations in different locations would be Concrete Subjects.

**3. Observer / Subscriber interface**

* Observer defines an interface with an update method to ensure that concrete observers receive updates from the subject in a consistent way.
* It can be interface or abstract class.

**4. Concrete Observer**

* Concrete observers are specific implementations of the observer interface.
* They get registered to the concrete subjects & react when notified of a state change i.e., when the subject’s state changes, the concrete observer’s update () method is invoked, allowing it to take appropriate actions.
* E.g., A Weather app on smartphone can be a concrete observer that reacts to changes from a weather station.



**When to use the Observer Design Pattern?**

Below is when to use observer design pattern:

* When you need one object to notify multiple others about changes.
* When you want to keep objects loosely connected, so they don’t rely on each other’s details.
* When you want observers to automatically respond to changes in the subject’s state.
* When you want to easily add or remove observers without changing the main subject.
* When you’re dealing with event systems that require various components to react without direct connections.

**When not to use the Observer Design Pattern?**

Below is when not to use observer design pattern:

* When the relationships between objects are simple and don’t require notifications.
* When performance is a concern, as many observers can lead to overhead during updates.
* When the subject and observers are tightly coupled, as it defeats the purpose of decoupling.
* When number of observers is fixed and won’t change over time.
* When the order of notifications is crucial, as observers may be notified in an unpredictable sequence.

Example1 – Weather Application

|  |
| --- |
| **Use case:**  Design a weather application that will used on TV & Mobile to get current weather update from multiple weather stations.   * TV Display will receive update from Weather station 1 & 2 * Mobile Display will receive update from Weather station 1   **Implementation:**    Key points:   * Observer: **PhoneDisplay** observes **WeatherStation 1** and **TvDisplay** observes **WeatherStation 1** & **WeatherStation 2** * **WeatherStation** setWeather () will trigger the notifyObservers () method * 2 ways to consume weather update in Observers  1. Pass the state/message through update method arguments 2. ***Imp****: Observer class can have state/message as property & initialized through constructor injection i.e., Has\_A Relationship* |

**Decorator / Wrapper Design Pattern**

* Decorator design pattern is a structural design pattern that let us add new behaviour to individual objects dynamically, without affecting the behaviour of other objects from the same class.

**Components of Decorator Design Pattern**

**1. Component Interface**

* It specifies the operations that can be performed on the objects & it is common interface for both concrete components & decorators.
* This can be an abstract class or interface.

**2. Concrete Component**

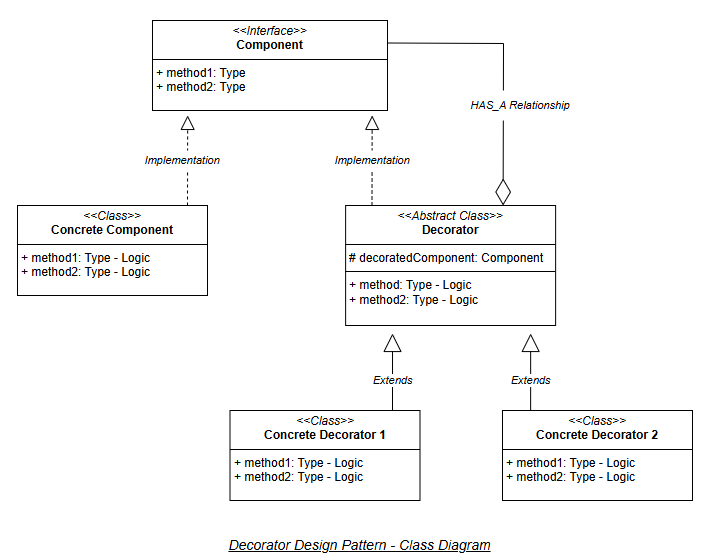
* These are the basic classes that implement the Component interface & add new behavior or responsibilities to Concrete Component objects.

**3. Decorator** (HAS\_A Relationship + IS\_A Relationship)

* Decorator is an abstract class that has a reference to component object & also implements the Component interface.
* Decorators are responsible for adding new behaviors to the wrapped Component object.

**4. Concrete Decorators**

* These are the concrete classes that extends the Decorator class & add a specific behaviors or responsibilities to the already existing Concrete Component objects.



**Advantages of Decorator Design Pattern**

* **Composition over Inheritance**
* Unlike traditional inheritance, which can lead to a deep & inflexible class hierarchy, the decorator pattern uses composition.
* We can compose objects with different decorators to achieve desired functionality, avoiding the drawbacks of inheritance, such as tight coupling & rigid hierarchies.
* **Open-Closed Principle**
* Using this pattern, we can introduce new functionality to an existing class without changing its source code.
* **Flexibility**
* It allows us to add or remove responsibilities (i.e., behaviors) from objects at runtime.
* This flexibility makes it easy to create complex object structures with varying combinations of behaviors.
* **Reusable Code**
* Decorators are reusable components. We can create a library of decorator classes & apply them to different objects & classes as needed, reducing code duplication.
* **Dynamic Behavior Modification**
* Decorators can be applied or removed at runtime, providing dynamic behavior modification for objects.
* This is particularly useful when we need to adapt an object’s behavior based on changing requirements or user preferences.
* **Clear Code Structure**
* The Decorator pattern promotes a clear & structured design, making it easier for developers to understand how different features & responsibilities are added to objects.

**Disadvantages of Decorator Design Pattern**

* **Order of Decoration**
* The order in which decorators are applied can affect the final behavior of the object.
* If decorators are not applied in correct order, it can lead to unexpected results.
* Managing the order of decorators can be challenging, especially in complex scenarios.
* **Increased number of classes**
* When using the decorator pattern, we often end up with a large number of small, specialized decorator classes.
* This can lead to a proliferation of classes in our codebase, which may increase maintenance overhead.
* **Potential for Overuse**
* As it’s easy to add decorators to objects, there is a risk of overusing the decorator pattern, making the codebase unnecessarily complex.
* It’s important to use decorators effectively & only when they genuinely add value to the design.
* **Complexity**
* As we add more decorators to an object, the code can become more complex & harder to understand.
* The nesting of decorators can make the codebase difficult to navigate & debug, especially when there are many decorators involved.
* **Limited Support in some languages**
* Some Programming language may not provide convenient support for implementing decorators.
* Implementing the pattern can be more verbose (uses more words) & less intuitive in such language.

Example: Coffee Shop Application

|  |
| --- |
| **Scenario:**  Suppose we’re building a coffee shop application where customers can order different types of coffee.   * Each coffee can have various optional add-ons such as milk, sugar, whipped cream etc. * We want to implement a system where we can dynamically add these add-ons to a coffee order without modifying the coffee classes themselves. |