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

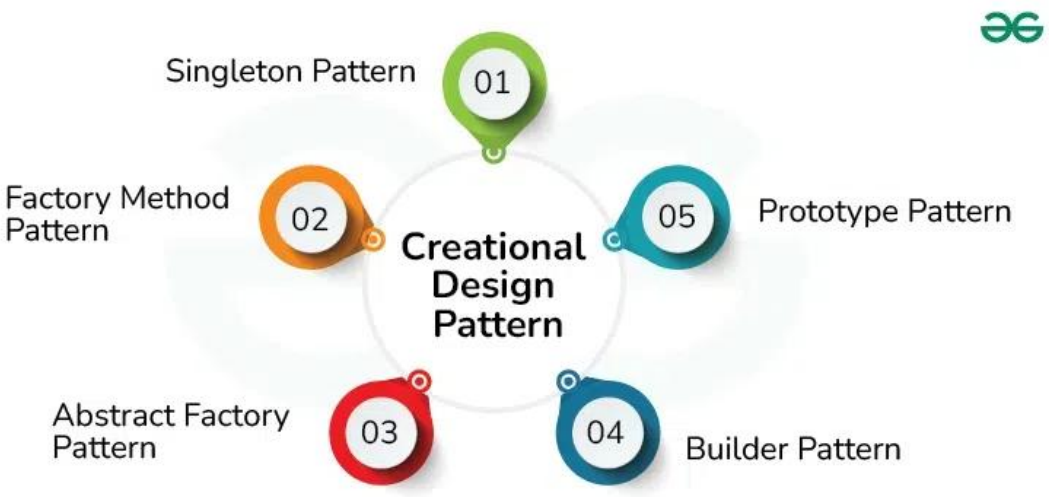
- Both concepts describe typical solutions to some known problems.
- An Algorithm always defines a clear set of actions that can achieve some goal.
- While a Pattern is more high-level description of a solution.

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.

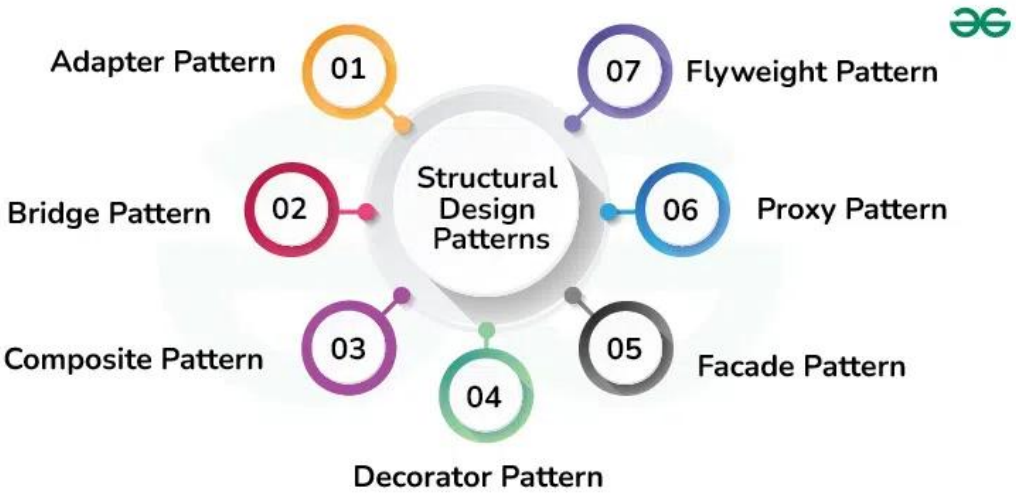
1. Creational Patterns

- 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.



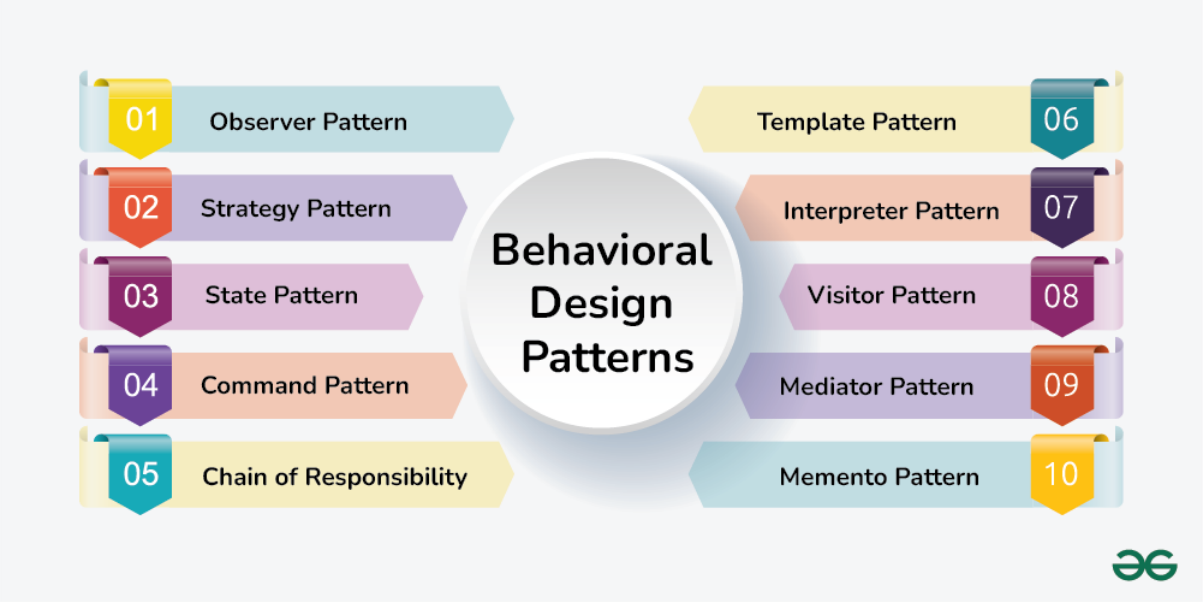
2. Structural Patterns

- 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

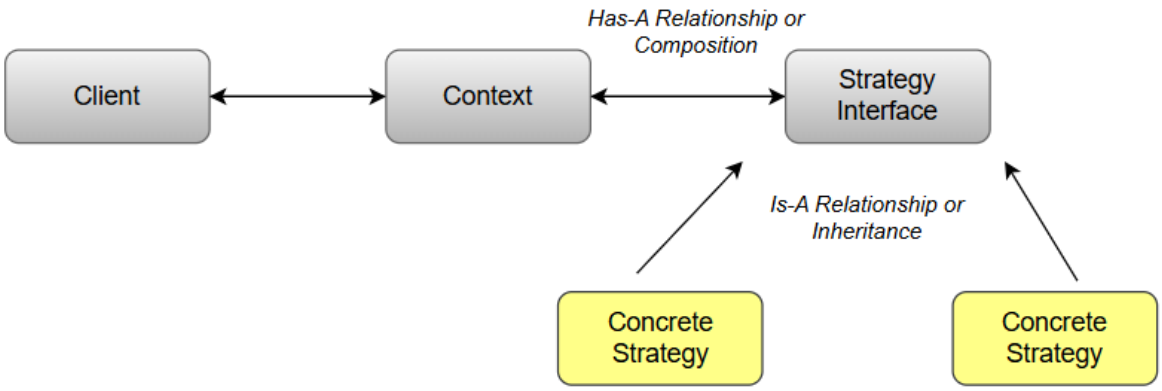
3. Builder Design Pattern

- 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

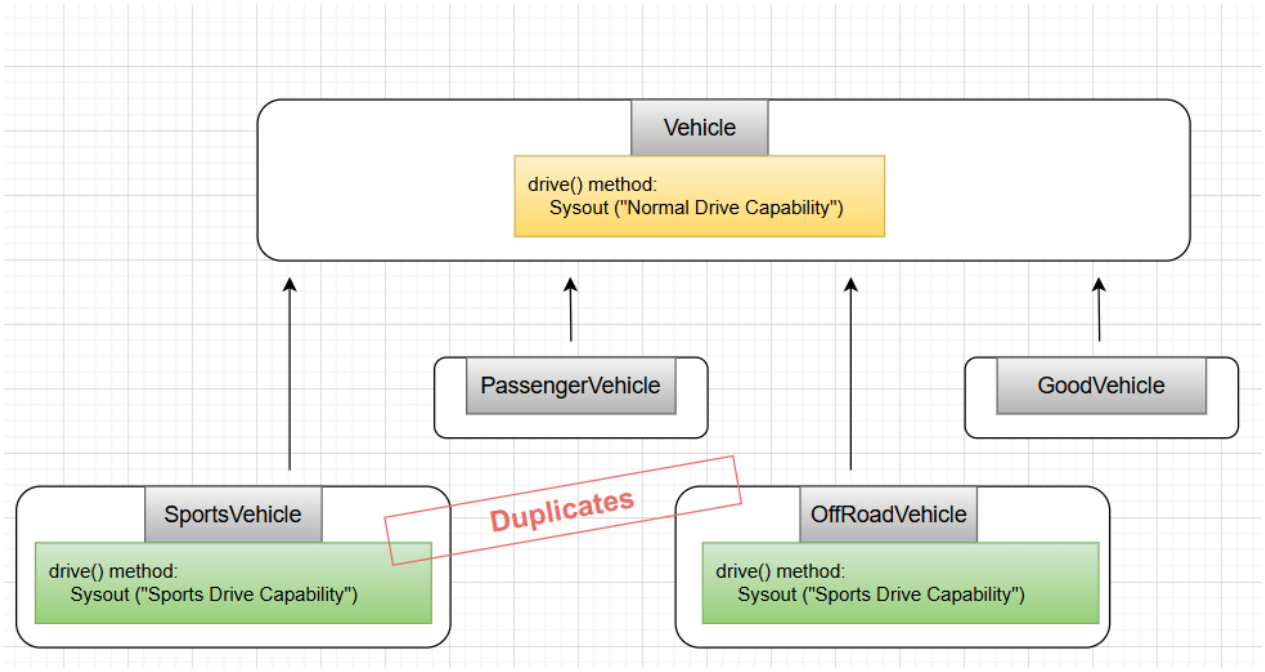
- Keep the interface simple & focused on a single responsibility
- Encapsulate any stateful behavior in the concrete strategy classes, rather than in the context class.
- Use Dependency injection to pass the concrete strategy to the context class, rather than creating it directly in the context class.
- 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.