Strategy Design Pattern

Last Updated : 17 Dec, 2024

The Strategy Design Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable, allowing clients to switch algorithms dynamically without altering the code structure.

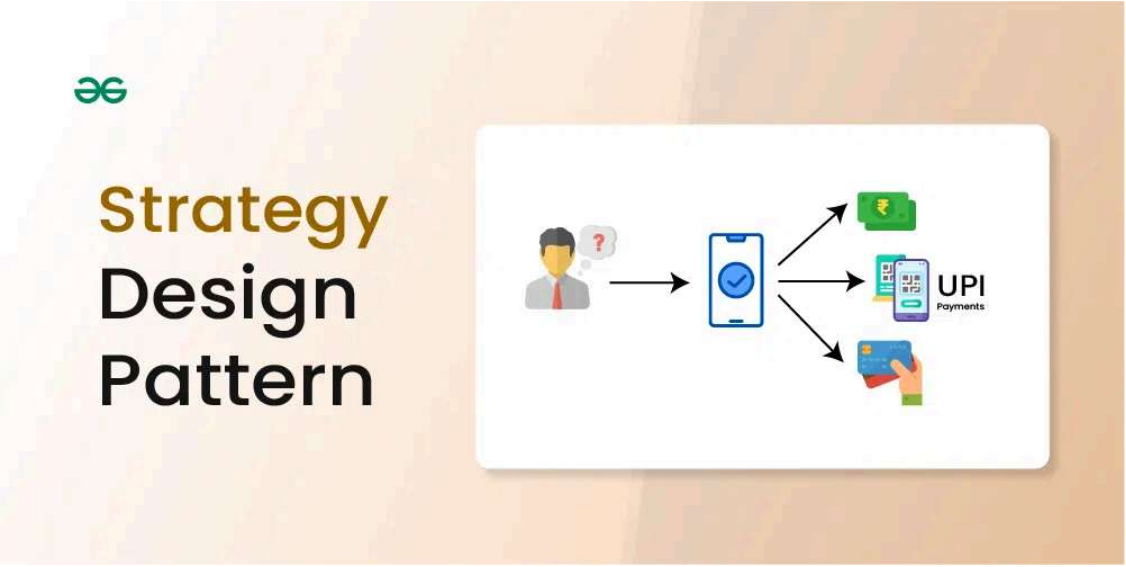


Table of Content

What is the Strategy Design Pattern?

The Strategy Design Pattern is a behavioral design pattern that allows you to define a family of algorithms or behaviors, put each of them in a separate class,

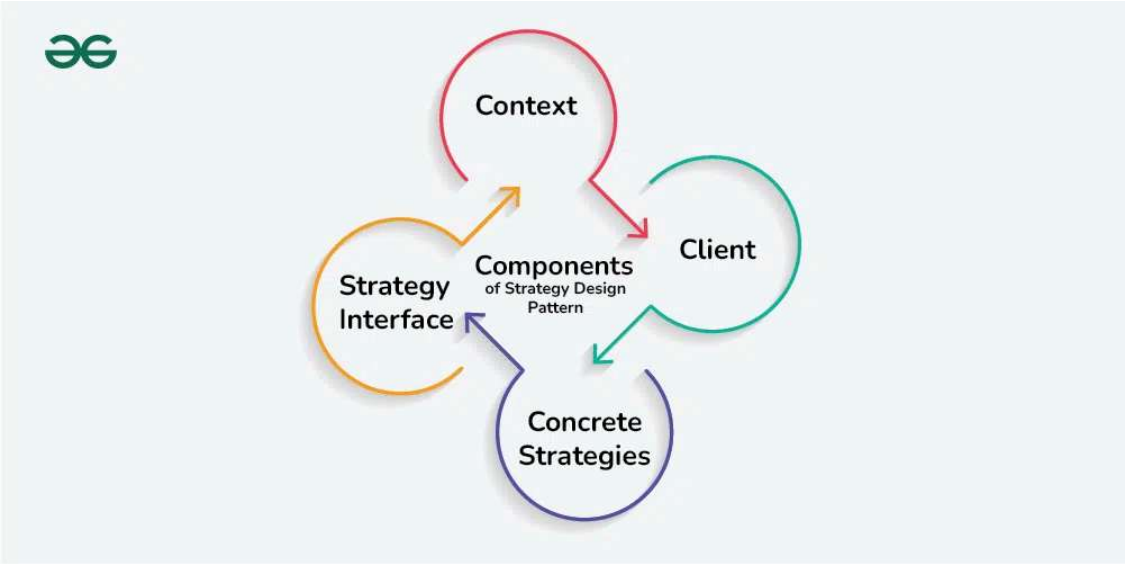
and make them interchangeable at runtime. This pattern is useful when you want to dynamically change the behavior of a class without modifying its code.

Characteristics of this design pattern

This pattern exhibits several key characteristics, such as:

- **Defines a family of algorithms:** The pattern allows you to encapsulate multiple algorithms or behaviors into separate classes, known as strategies.
- **Encapsulates behaviors:** Each strategy encapsulates a specific behavior or algorithm, providing a clean and modular way to manage different variations or implementations.
- **Enables dynamic behavior switching:** The pattern enables clients to switch between different strategies at runtime, allowing for flexible and dynamic behavior changes.
- **Promotes object collaboration:** The pattern encourages collaboration between a context object and strategy objects, where the context delegates the execution of a behavior to a strategy object.

Components of the Strategy Design Pattern



1. Context

A class or object known as the Context assigns the task to a strategy object and contains a reference to it.

- It serves as an intermediary between the client and the strategy, offering an integrated approach for task execution without exposing every detail of the process.
- The Context maintains a reference to a strategy object and calls its methods to perform the task, allowing for interchangeable strategies to be used.

2. Strategy Interface

An abstract class or interface known as the Strategy Interface specifies a set of methods that all concrete strategies must implement.

- As a kind of agreement, it guarantees that all strategies follow the same set of rules and are interchangeable by the Context.
- The Strategy Interface promotes flexibility and modularity in the design by establishing a common interface that enables decoupling between the Context and the specific strategies.

3. Concrete Strategies

Concrete Strategies are the various implementations of the Strategy Interface. Each concrete strategy provides a specific algorithm or behavior for performing the task defined by the Strategy Interface.

- Concrete strategies encapsulate the details of their respective algorithms and provide a method for executing the task.
- They are interchangeable and can be selected and configured by the client based on the requirements of the task.

4. Client

The Client is responsible for selecting and configuring the appropriate strategy and providing it to the Context.

- It knows the requirements of the task and decides which strategy to use based on those requirements.
- The client creates an instance of the desired concrete strategy and passes it to the Context, enabling the Context to use the selected strategy to perform the task.

Communication between the Components

In the Strategy Design Pattern, communication between the components occurs in a structured and decoupled manner. Here's how the components interact with each other:

- **Client to Context**:
    - The Client, which knows the requirements of the task, interacts with the Context to initiate the task execution.
    - The Client selects an appropriate strategy based on the task requirements and provides it to the Context.
    - The Client may configure the selected strategy before passing it to the Context if necessary.
- **Context to Strategy**:
    - The Context holds a reference to the selected strategy and delegates the task to it.
    - The Context invokes a method on the strategy object, triggering the execution of the specific algorithm or behavior encapsulated within the strategy.
- **Strategy to Context**:
    - Once the strategy completes its execution, it may return a result or perform any necessary actions.
    - The strategy communicates the result or any relevant information back to the Context, which may further process or utilize the result as needed.
- **Strategy Interface as Contract**:
    - The Strategy Interface serves as a contract that defines a set of methods that all concrete strategies must implement.
    - The Context communicates with strategies through the common interface, promoting interchangeability and decoupling.
- **Decoupled Communication**:
    - Since the components' communication is decoupled, the Context is not required to be aware of the exact details of how each strategy carries out the task.

- As long as they follow the same interface, strategies can be switched or replaced without affecting the client or other strategies.

Real-World Analogy of Strategy Design Pattern

Imagine you're planning a trip to a new city, and you have several options for getting there: by car, by train, or by plane. Each mode of transportation offers its own set of advantages and disadvantages, depending on factors such as cost, travel time, and convenience.

- **Context**: You, as the traveler, represent the context in this analogy. You have a specific goal (reaching the new city) and need to choose the best transportation strategy to achieve it.
- **Strategies**: The different modes of transportation (car, train, plane) represent the strategies in this analogy. Each strategy (mode of transportation) offers a different approach to reaching your destination.
- **Interface**: The interface in this analogy is the set of common criteria you consider when choosing a transportation mode, such as cost, travel time, and convenience.
- **Flexibility**: Just as the Strategy Design Pattern allows you to dynamically switch between different algorithms at runtime, you have the flexibility to choose a transportation mode based on your specific requirements and constraints.
- **Dynamic Selection**: The Strategy Design Pattern allows you to dynamically select the best strategy (transportation mode) based on changing circumstances. For instance, if your initial flight is canceled due to bad weather, you can quickly switch to an alternative mode of transportation

Strategy Design Pattern Example(with implementation)

Problem Statement:

Let's consider a sorting application where we need to sort a list of integers. However, the sorting algorithm to be used may vary depending on factors such as the size of the list and the desired performance characteristics.

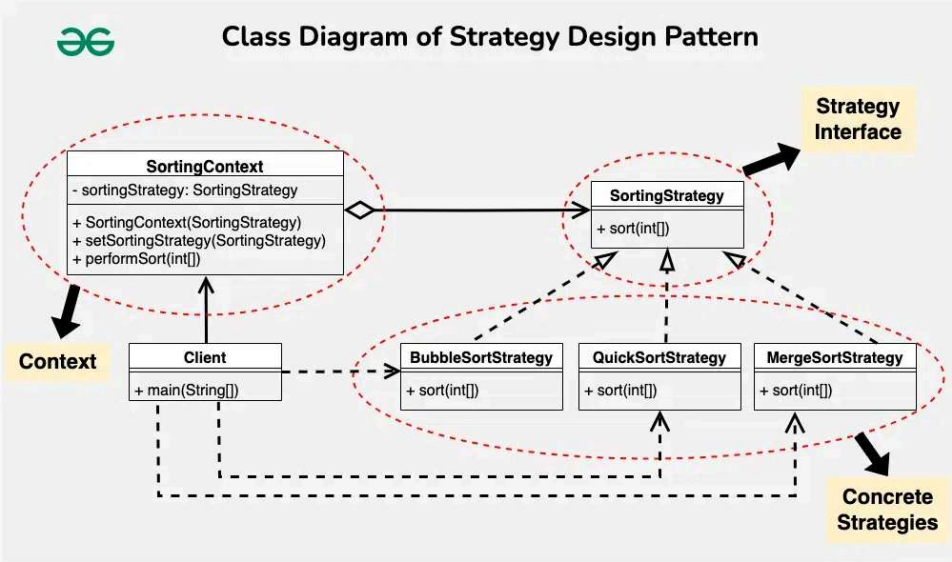
Challenges Without Using Strategy Pattern:

- **Limited Flexibility**: Implementing sorting algorithms directly within the main sorting class can make the code inflexible. Adding new sorting algorithms or changing existing ones would require modifying the main class, which violates the Open/Closed Principle.
- **Code Duplication**: Without a clear structure, you may end up duplicating sorting logic to handle different algorithms.
- **Hard-Coded Logic**: Implementing sorting logic directly within the main sorting class can make the code rigid and difficult to extend or modify.

How Strategy Pattern helps to solve above challenges?

Here's how the Strategy Pattern helps:

- **Code Reusability**: By encapsulating sorting algorithms into separate strategy classes, you can reuse these strategies across different parts of the system. This reduces code duplication and promotes maintainability.
- **Flexibility and Extensibility**: The Strategy Pattern makes it simple to adapt or add new sorting algorithms without changing the existing code. Since each strategy is independent, it is possible to change or expand it without impacting other system components.
- **Separation of Concerns**: The Strategy Pattern separates sorting logic into distinct strategy classes, which encourages a clear division of responsibilities. As a result, the code is easier to test, and maintain.



Below is the implementation of the above example:

1. Context(SortingContext)

```
public class SortingContext {
    private SortingStrategy sortingStrategy;

    public SortingContext(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }

    public void setSortingStrategy(SortingStrategy sortingStr
{
        this.sortingStrategy = sortingStrategy;
    }

    public void performSort(int[] array) {
        sortingStrategy.sort(array);
    }
}
```

2. Strategy Interface(SortingStrategy)

```
public interface SortingStrategy {
```

```
    void sort(int[] array);
}
```

3. Concrete Strategies

```
// BubbleSortStrategy
public class BubbleSortStrategy implements SortingStrategy
{
    @Override
    public void sort(int[] array) {
        // Implement Bubble Sort algorithm
        System.out.println("Sorting using Bubble Sort");
    }
}

// MergeSortStrategy
public class MergeSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        // Implement Merge Sort algorithm
        System.out.println("Sorting using Merge Sort");
    }
}

// QuickSortStrategy
public class QuickSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        // Implement Quick Sort algorithm
        System.out.println("Sorting using Quick Sort");
    }
}
```

4. Client Component

```
public class Client {
```

```
    public static void main(String[] args) {
        // Create SortingContext with BubbleSortStrategy
        SortingContext sortingContext = new SortingContext(new BubbleSortStrategy());
        int[] array1 = {5, 2, 9, 1, 5};
        sortingContext.performSort(array1); // Output: Sorting using Bubble Sort

        // Change strategy to MergeSortStrategy
        sortingContext.setSortingStrategy(new MergeSortStrategy());
        int[] array2 = {8, 3, 7, 4, 2};
        sortingContext.performSort(array2); // Output: Sorting using Merge Sort

        // Change strategy to QuickSortStrategy
        sortingContext.setSortingStrategy(new QuickSortStrategy());
        int[] array3 = {6, 1, 3, 9, 5};
        sortingContext.performSort(array3); // Output: Sorting using Quick Sort
    }
}
```

Complete code for the above example

Below is the complete code for the above example:

```
// SortingContext.java
class SortingContext {
    private SortingStrategy sortingStrategy;

    public SortingContext(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }

    public void setSortingStrategy(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }
```

```
12
13      public void performSort(int[] array) {
14          sortingStrategy.sort(array);
15      }
16  }
17
18  // SortingStrategy.java
19  interface SortingStrategy {
20      void sort(int[] array);
21  }
22
23  // BubbleSortStrategy.java
24  class BubbleSortStrategy implements SortingStrategy {
25      @Override
26      public void sort(int[] array) {
27          // Implement Bubble Sort algorithm
28          System.out.println("Sorting using Bubble Sort");
29          // Actual Bubble Sort Logic here
30      }
31  }
32
33  // MergeSortStrategy.java
34  class MergeSortStrategy implements SortingStrategy {
35      @Override
36      public void sort(int[] array) {
37          // Implement Merge Sort algorithm
38          System.out.println("Sorting using Merge Sort");
39          // Actual Merge Sort Logic here
40      }
41  }
42
43  // QuickSortStrategy.java
44  class QuickSortStrategy implements SortingStrategy {
45      @Override
46      public void sort(int[] array) {
47          // Implement Quick Sort algorithm
48          System.out.println("Sorting using Quick Sort");
49          // Actual Quick Sort Logic here
50      }
51  }
52
53  // Client.java
54  public class Client {
```

```
    public static void main(String[] args) {
        // Create SortingContext with BubbleSortStrategy
        SortingContext sortingContext = new SortingContext(new BubbleSortStrategy());
        int[] array1 = {5, 2, 9, 1, 5};
        sortingContext.performSort(array1); // Output: Sorting using Bubble Sort

        // Change strategy to MergeSortStrategy
        sortingContext.setSortingStrategy(new MergeSortStrategy());
        int[] array2 = {8, 3, 7, 4, 2};
        sortingContext.performSort(array2); // Output: Sorting using Merge Sort

        // Change strategy to QuickSortStrategy
        sortingContext.setSortingStrategy(new QuickSortStrategy());
        int[] array3 = {6, 1, 3, 9, 5};
        sortingContext.performSort(array3); // Output: Sorting using Quick Sort
    }
}
```

Output

```
Sorting using Bubble Sort
Sorting using Merge Sort
Sorting using Quick Sort
```

When to use the Strategy Design Pattern?

Here are some situations where you should consider using the Strategy pattern:

- **Multiple Algorithms**: When you have multiple algorithms that can be used interchangeably based on different contexts, such as sorting algorithms (bubble sort, merge sort, quick sort), searching algorithms, compression algorithms, etc.
- **Encapsulating Algorithms**: When you want to encapsulate the implementation details of algorithms separately from the context that uses

them, allowing for easier maintenance, testing, and modification of algorithms without affecting the client code.

- **Runtime Selection**: When you need to dynamically select and switch between different algorithms at runtime based on user preferences, configuration settings, or system states.
- **Reducing Conditional Statements**: When you have a class with multiple conditional statements that choose between different behaviors, using the Strategy pattern helps in eliminating the need for conditional statements and making the code more modular and maintainable.
- **Testing and Extensibility**: When you want to facilitate easier unit testing by enabling the substitution of algorithms with mock objects or stubs. Additionally, the Strategy pattern makes it easier to extend the system with new algorithms without modifying existing code.

When not to use the Strategy Design Pattern?

Here are some situations where you should consider not using the Strategy pattern:

- **Single Algorithm**: If there is only one fixed algorithm that will be used throughout the lifetime of the application, and there is no need for dynamic selection or switching between algorithms, using the Strategy pattern might introduce unnecessary complexity.
- **Overhead**:If the overhead of implementing multiple strategies outweighs the benefits, especially in simple scenarios where direct implementation without the Strategy pattern is more straightforward and clear.
- **Inflexible Context**: If the context class tightly depends on a single algorithm and there is no need for flexibility or interchangeability, using the Strategy pattern may introduce unnecessary abstraction and complexity.

Advantages of the Strategy Design Pattern

Below are the advantages of the strategy design pattern:

- A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.

- By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
- The application can switch strategies at run-time.
- Strategy enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
- Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

Disadvantages of the Strategy Design Pattern

Below are the disadvantages of the strategy design pattern:

- The application must be aware of all the strategies to select the right one for the right situation.
- Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.
- In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.

Next Article

Template Method Design Pattern

Similar Reads

Software Design Patterns Tutorial

Software design patterns are important tools developers, providing proven solutions to common problems encountered during software development. This article will act as tutorial to help you understand the concep...

9 min read

Complete Guide to Design Patterns

Design patterns help in addressing the recurring issues in software design and provide a shared vocabulary for developers to communicate and collaborate effectively. They have been documented and refined over tim...

11 min read

Types of Software Design Patterns

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our...

9 min read

1. Creational Design Patterns

Creational Design Patterns

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, and represented. Creational...

4 min read

Types of Creational Patterns

2. Structural Design Patterns

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships betwee...

7 min read

Types of Structural Patterns

3. Behvioural Design Patterns

Behavioral Design Patterns

Behavioral design patterns are a category of design patterns that focus on the interactions and communication between objects. They help define how objects collaborate and distribute responsibility among them, makin...

5 min read

3. Types of Behvioural Patterns

Chain of Responsibility Design Pattern

The Chain of Responsibility design pattern is a behavioral design pattern that allows an object to pass a request along a chain of handlers. Each handler in the chain decides either to process the request or to pass...

10 min read

Command Design Pattern

The Command Design Pattern is a behavioral design pattern that turns a request into a stand-alone object called a command. With the help of this pattern, you can capture each component of a request, including th...

10 min read

Interpreter Design Pattern

The Interpreter Design Pattern is a behavioral design pattern used to define a language's grammar and provide an interpreter to process statements in that language. It is useful for parsing and executing...

10 min read

Mediator Design Pattern

The Mediator Design Pattern simplifies communication between multiple objects in a system by centralizing their interactions through a mediator. Instead of objects interacting directly, they communicate via a mediato...

7 min read

Memento Design Pattern

The Memento Design Pattern is a behavioral pattern that helps save and restore an object's state without exposing its internal details. It is like a "snapshot" that allows you to roll back changes if something goes...

6 min read

Observer Design Pattern

The Observer Design Pattern is a behavioral design pattern that defines a one-to-many dependency between objects. When one object (the subject) changes state, all its dependents (observers) are notified...

8 min read

State Design Pattern

The State design pattern is a behavioral software design pattern that allows an object to alter its behavior when its internal state changes. It achieves this by encapsulating the object's behavior within different state...

11 min read

Strategy Design Pattern

The Strategy Design Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable, allowing clients to switch algorithms dynamically without altering the code structure. Tabl...

11 min read

Template Method Design Pattern

The Template Method Design Pattern is a behavioral design pattern that provides a blueprint for organizing code, making it flexible and easy to extend. With this pattern, you define the core steps of an algorithm in a...

8 min read

Visitor design pattern

An object-oriented programming method called the Visitor design pattern makes it possible to add new operations to preexisting classes without changing them. It improves the modularity and maintainability of...

7 min read

Top Design Patterns Interview Questions [2024]

A design pattern is basically a reusable and generalized solution to a common problem that arises during software design and development. Design patterns are not specific to a particular programming language or...

9 min read

Software Design Pattern in Different Programming Languages

Java Design Patterns Tutorial

Design patterns in Java refer to structured approaches involving objects and classes that aim to solve recurring design issues within specific contexts. These patterns offer reusable, general solutions to common problems...

8 min read

Python Design Patterns Tutorial

Design patterns in Python are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Modern C++ Design Patterns Tutorial

Design patterns in C++ help developers create maintainable, flexible, and understandable code. They encapsulate the expertise and experience of seasoned software architects and developers, making it easier f...

7 min read

JavaScript Design Patterns Tutorial

Design patterns in Javascipt are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

Software Design Pattern Books

Software Design Pattern in Development

Some other Popular Design Patterns

Design Patterns in Different Languages