### Strategy Design Pattern [Layman’s View]

Imagine you are planning a road trip; you can travel by car, by bus, or by bike. Each mode of transport is a strategy for reaching your destination. Similarly, in software development, the Strategy Design Pattern allows an object to choose from a family of algorithms, encapsulating each one, and making them interchangeable. The object's behavior changes at runtime depending on the strategy it employs, without altering its interface.

This design pattern falls under the category of behavioral patterns, as it's used to manage algorithms, relationships, and responsibilities between objects.

The Strategy Design Pattern is a fundamental concept within software design, offering a powerful way to enable flexible code that can adapt to changing requirements or scenarios. By embracing the principles of this pattern, developers can craft software that is more maintainable, scalable, and easier to understand. Let's break down the Strategy Design Pattern into simpler terms before diving into its technical intricacies, use cases, and potential pitfalls.

### Code Example

Let's illustrate the Strategy Design Pattern with a simple Java example: a text formatting application where the formatting strategy can be switched at runtime.

public interface TextFormatterStrategy {

    String format(String text);

}

**Concrete Strategies:**

public class UpperCaseFormatter implements TextFormatterStrategy {

    @Override

    public String format(String text) {

        return text.toUpperCase();

    }

}

public class LowerCaseFormatter implements TextFormatterStrategy {

    @Override

    public String format(String text) {

        return text.toLowerCase();

    }

}

**Context:**

public class TextEditor {

    private TextFormatterStrategy formatter;

    public TextEditor(TextFormatterStrategy formatter) {

        this.formatter = formatter;

    }

    public void setFormatter(TextFormatterStrategy formatter) {

        this.formatter = formatter;

    }

    public void publishText(String text) {

        System.out.println(formatter.format(text));

    }

}

**Usage:**

public class StrategyDemo {

    public static void main(String[] args) {

        TextEditor editor = new TextEditor(new UpperCaseFormatter());

        editor.publishText("First Strategy Example");

        editor.setFormatter(new LowerCaseFormatter());

        editor.publishText("Switched to Another Strategy");

    }

}

The Strategy Design Pattern consists of three main components:

* **Context:** The part of the application that uses a strategy. It maintains a reference to one of the concrete strategies and communicates with this strategy only via the strategy interface.
* **Strategy Interface:** This defines a common interface for all supported algorithms. It declares a method the context uses to execute a strategy.
* **Concrete Strategies:** These are classes that implement the strategy interface, each providing a different implementation of an algorithm or behaviour.

### When to Use

* **Flexibility in Algorithms:** When you have multiple algorithms for a specific task and want to switch between them dynamically.
* **Decoupling Algorithm from Context:** If your application needs to separate the implementation details of an algorithm from the code that uses it.
* **Avoiding Conditional Statements:** Instead of using numerous conditional operations to switch between different algorithms

### When Not to Use

* **Overhead for Simple Decisions:** If you only have a couple of algorithms and they are unlikely to change or grow, introducing the pattern may add unnecessary complexity.
* **Direct Communication:** When strategies need to communicate with each other or with the context frequently, it might complicate the design unnecessarily.

**Pitfalls**

* **Overhead:** Implementing the pattern can lead to additional complexity and overhead, especially for small projects.
* **Understanding the Design:** Misunderstanding the pattern can lead to incorrect implementations, such as creating strategies that are too granular or too broad.
* **Maintaining State:** Strategies that maintain state can introduce side effects or bugs if the state is not managed carefully.