Understanding the Strategy Pattern and its Distinction from Normal Abstraction

1 Introduction

The Strategy Pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern enables the algorithm to vary independently from the clients that use it. In this document, we will explore the Strategy Pattern in detail, using first principles thinking, and distinguish it from normal abstraction.

2 Strategy Pattern

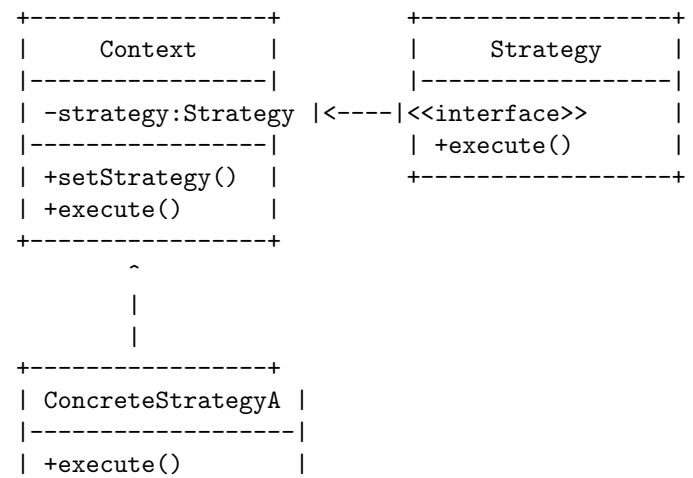
2.1 Components of the Strategy Pattern

- **Strategy Interface**: An interface common to all supported algorithms. This interface is used by the context object to call the algorithm defined by a concrete strategy.
- **Concrete Strategies**: Classes that implement the Strategy interface. Each class encapsulates a specific algorithm or behavior.
- **Context**: A class that uses a Strategy. The context maintains a reference to a Strategy object and delegates it to perform the algorithm.

2.2 Key Concepts

- **Encapsulation**: The Strategy Pattern encapsulates the details of each algorithm in separate classes.
- **Interchangeability**: Algorithms can be easily swapped in and out by changing the Strategy object within the context.
- **Extensibility**: New algorithms can be introduced without modifying the existing codebase, following the Open/Closed Principle.

2.3 UML Diagram



```
+--------------------+
        ^
        |
        |
+------------------+
| ConcreteStrategyB |
|-------------------|
| +execute()        |
+--------------------+
```

2.4 Example in Java

Consider a simple payment system where different payment methods can be selected at runtime.

Strategy Pattern in Java

```
interface PaymentStrategy {
    void pay(double amount);
}

class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paying " + amount + " using Credit Card.");
    }
}

class PayPalPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paying " + amount + " using PayPal.");
    }
}

class ShoppingCart {
    private PaymentStrategy strategy;

    public ShoppingCart(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void checkout(double amount) {
        strategy.pay(amount);
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart(new CreditCardPayment());
        cart.checkout(100);  // Output: Paying 100 using Credit Card.

        cart.setStrategy(new PayPalPayment());
        cart.checkout(200);  // Output: Paying 200 using PayPal.
    }
}
```

2.5 Benefits of the Strategy Pattern

- **Flexibility**: Allows algorithms to be changed without modifying the context.
- **Reusability**: Algorithms can be reused by different contexts.
- **Maintenance**: Encapsulating algorithms into separate classes makes the code easier to manage and extend.

2.6 Drawbacks of the Strategy Pattern

- **Overhead**: Introducing numerous strategy classes can add complexity and overhead.
- **Communication**: Strategies might require communication with context objects, leading to tightly coupled designs if not managed correctly.

2.7 When to Use

- When you have multiple related algorithms that need to be used interchangeably.
- When you want to avoid using conditional statements for algorithm selection.
- When you need to enable clients to choose different algorithms at runtime.

3 Strategy Pattern Using First Principles Thinking

First principles thinking involves breaking down complex problems into their most basic elements and then reassembling them from the ground up.

3.1 Identify the Core Problem

The core problem addressed by the Strategy Pattern is the need for a system to perform an operation in multiple ways.

3.2 Decompose the Problem

Break down the problem into fundamental components:

- **Operation**: A task that needs to be performed, but can be done in various ways (e.g., payment processing, sorting).
- **Algorithms**: Different methods to perform the operation (e.g., credit card payment, PayPal payment, bubble sort, quicksort).
- **Interchangeability**: The ability to swap algorithms without changing the core operation logic.
- **Encapsulation**: Keeping the details of each algorithm separate to avoid complex interdependencies.

3.3 Identify the Basic Elements

The basic elements required to solve the problem are:

- **Strategy Interface**: Declares a method for the operation that all algorithms must implement.
- **Concrete Strategies**: Implementations of the Strategy Interface, each encapsulating a specific algorithm.
- **Context**: Uses a Strategy to perform the operation, maintaining a reference to a Strategy and delegating the operation to it.

3.4 Reassemble from the Ground Up

Reassemble the basic elements into a coherent solution:

Reconstructed Strategy Pattern in Java

```
interface PaymentStrategy {
    void pay(double amount);
}

class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paying " + amount + " using Credit Card.");
    }
}

class PayPalPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paying " + amount + " using PayPal.");
    }
}

class ShoppingCart {
    private PaymentStrategy strategy;

    public ShoppingCart(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void checkout(double amount) {
        strategy.pay(amount);
    }
}
```

3.5 Analyze the Reconstructed Solution

- **Flexibility**: The ShoppingCart class can easily switch between different payment methods without modifying its code.
- **Encapsulation**: Each payment method is encapsulated in its own class, making the system easier to maintain and extend.
- **Interchangeability**: Payment methods can be swapped dynamically at runtime, providing flexibility.

3.6 Evaluate the Solution Against the Core Problem

The reconstructed solution addresses the core problem by ensuring different algorithms can be used interchangeably, are encapsulated, and can be extended without modifying existing code.

4 Distinction from Normal Abstraction

While both the Strategy Pattern and normal abstraction involve creating general interfaces to abstract away specific details, they serve different purposes and are applied in distinct ways.

4.1 Normal Abstraction

4.1.1 Definition

Abstraction involves hiding the complex implementation details and showing only the essential features of an object.

4.1.2 Purpose

- Simplify the interface for interacting with complex systems.
- Separate what an object does from how it does it.
- Allow different implementations to be used interchangeably through a common interface.

4.1.3 Usage

Defining abstract classes or interfaces that declare methods without implementing them. Concrete classes implement these methods, providing specific behaviors.

4.1.4 Example

Imagine a Shape class with an abstract method `draw()`.

Normal Abstraction in Java

```
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a Circle.");
    }
}

class Square extends Shape {
    void draw() {
        System.out.println("Drawing a Square.");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape square = new Square();

        circle.draw();  // Output: Drawing a Circle.
        square.draw();  // Output: Drawing a Square.
    }
}
```

4.2 Strategy Pattern vs. Normal Abstraction

4.2.1 Key Differences

- **Purpose**:
  - **Abstraction**: Focuses on simplifying interfaces and separating the definition of behavior from its implementation.

  - **Strategy Pattern**: Focuses on selecting and swapping algorithms dynamically, promoting flexibility and adherence to the Open/Closed Principle.

- **Usage Context**:
  - **Abstraction**: Generally used to define a general interface or base class that multiple subclasses can implement or extend.
  - **Strategy Pattern**: Specifically used when there are multiple ways to perform a particular task, and the method needs to be selected or changed at runtime.

- **Design Intent**:
  - **Abstraction**: Simplifies and generalizes functionality. It's more about creating a hierarchy and reuse.
  - **Strategy Pattern**: Provides a way to change the behavior of a class by changing the algorithm it uses, allowing for more dynamic and flexible design.

- **Encapsulation**:
  - **Abstraction**: Encapsulates behavior in a base class or interface.
  - **Strategy Pattern**: Encapsulates behavior in separate strategy classes, allowing them to be interchangeable.

5 Summary

While both abstraction and the Strategy Pattern involve interfaces and implementation, they address different needs in software design. Abstraction is about defining general behavior and hiding details, whereas the Strategy Pattern is about selecting among multiple algorithms dynamically to achieve flexible and maintainable code. The Strategy Pattern builds on the concept of abstraction by providing a way to dynamically change the behavior at runtime through interchangeable strategy objects.