**Strategy Design Pattern**

1. It is Behavioral Design Pattern.

As we know Object has State and Behaviour.

Strategy pattern is based upon Open Closed design principle of SOLID principals.

**Strategy:-**

1. Encapsulates an algorithm in class.

2. When we have different implementations of an algorithm or behaviour.

3. Methods in class which have different behaviour and selected through switch case or if else case then we can

use Strategy design pattern.

4. Strategy classes implements interface and it will be same for all strategy classes . When we provide main object then

it will choose implemented class for that object.

Example- Class A and Class B implements interface C (Interface has processAlgo() method)

In Java :-

**Part 1:-**

Sort Algorithm , which could take object of Bubble sort , Quick Sort or merge sort.

Example :- Credit Card -> ICICI , SBI , Axis Algo:- provide different discount on credit cards. So they could have different algorithms based on card.

Another scenario could be : ICICI Credit card have different CCs.

Platinum , Titanium , Diamond and they could have different deductions.

**Part 2 :**

Based on some input we can decide which class should called. Every class has different behaviour.

Let's suppose : We are getting some Request from external sources and in Request we are getting different params. Example : Inputs are SMS , EMAIL , Data etc.

Based on above inputs we need to take decision or write own business logic.

It could have different algorithms or behaviour as per input parameters.

Then we apply Strategy Design pattern. Where we can define SMS , Email and Data classes and these classes can inherit same method say processAlgo().

**UML Diagram :**

<<interface>>

A

processAlgo()

Main Class

Strategy Object provided in main class

Class C

processAlgo()

Class B

processAlgo()

Example of Strategy from Java : java.util.Comparator.

JDK has examples ,

1. the first is Collection.sort(List, Comparator) method.

Where Comparator is Strategy and Collections.sort() is Context.

Because of this pattern, your sort method can sort any object, the object which doesn't exist when this method was written.

1. Another example is  java.util.Arrays#sort(T[], Comparator < ? super T > c) method which similar to Collections.sort() method, except need array in place of List.

**Disadvantage** :

If new Algorithm added then client side modification required.

**Java8:**

Java 8 has been released, the introduction of lambdas has made anonymous inner types more or less redundant. That means creating strategies in line is now a lot cleaner and easier.

Instead of choosing between concrete classes or anonymous inner types, let's try creating lambdas all in a single class:

public **interface** Discounter {

BigDecimal applyDiscount(BigDecimal amount);

static Discounter christmasDiscounter() {

return amount -> amount.multiply(BigDecimal.valueOf(0.9));

}

static Discounter newYearDiscounter() {

return amount -> amount.multiply(BigDecimal.valueOf(0.8));

}

static Discounter easterDiscounter() {

return amount -> amount.multiply(BigDecimal.valueOf(0.5));

}

}

**Leveraging Function Composition**

Let's modify our *Discounter* interface so it extends the ***UnaryOperator***interface, and then add a ***combine****()* method:

public interface Discounter extends UnaryOperator<BigDecimal> {

**default** Discounter combine(Discounter after) {

return value -> after.apply(this.apply(value));

}

}

Essentially, we are refactoring our *Discounter* and leveraging a fact that applying a discount is a function that converts a *BigDecimal* instance into another *BigDecimal* instance*,* allowing us to access predefined methods*.* **As the *UnaryOperator* comes with an *apply()* method, we can just replace *applyDiscount* with it.**

The *combine()* method is just an abstraction around applying one *Discounter* to the results of *this.* It uses the built-in functional *apply()* in order to achieve this.

Now, Let's try applying multiple *Discounters* cumulatively to an amount. We will do this by using the functional *reduce()* and our *combine():*

Discounter combinedDiscounter = discounters

.stream()

.reduce(v -> v, Discounter::combine);

combinedDiscounter.apply(...);

Pay special attention to the first *reduce* argument. When no discounts provided, we need to return the unchanged value. This can be achieved by providing an identity function as the default discounter.

This is a useful and less verbose alternative to performing a standard iteration. If we consider the methods we are getting out of the box for functional composition, it also gives us a lot more functionality for free.

Example :

Reference : Baeldung

|  |
| --- |
| public interface Discounter extends UnaryOperator<BigDecimal> { |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| default Discounter combine(Discounter after) { |
|  |

|  |
| --- |
| return value -> after.apply(this.apply(value)); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| static Discounter christmas() { |
|  |

|  |
| --- |
| return (amount) -> amount.multiply(BigDecimal.valueOf(0.9)); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| static Discounter newYear() { |
|  |

|  |
| --- |
| return (amount) -> amount.multiply(BigDecimal.valueOf(0.8)); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| static Discounter easter() { |
|  |

|  |
| --- |
| return (amount) -> amount.multiply(BigDecimal.valueOf(0.5)); |
|  |

|  |
| --- |
| } |
|  |

}

|  |
| --- |
| import static com.baeldung.strategy.Discounter.christmas; |
|  |

|  |
| --- |
| import static com.baeldung.strategy.Discounter.easter; |
|  |

|  |
| --- |
| import static com.baeldung.strategy.Discounter.newYear; |
|  |

|  |
| --- |
| import static org.assertj.core.api.Assertions.assertThat; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| public class StrategyDesignPatternUnitTest { |
|  |

|  |
| --- |
| @Test |
|  |

|  |
| --- |
| public void shouldDivideByTwo\_WhenApplyingStaffDiscounter() { |
|  |

|  |
| --- |
| Discounter staffDiscounter = new EasterDiscounter(); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| final BigDecimal discountedValue = staffDiscounter |
|  |

|  |
| --- |
| .apply(BigDecimal.valueOf(100)); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| assertThat(discountedValue) |
|  |

|  |
| --- |
| .isEqualByComparingTo(BigDecimal.valueOf(50)); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| @Test |
|  |

|  |
| --- |
| public void shouldDivideByTwo\_WhenApplyingStaffDiscounterWithAnonyousTypes() { |
|  |

|  |
| --- |
| Discounter staffDiscounter = new Discounter() { |
|  |

|  |
| --- |
| @Override |
|  |

|  |
| --- |
| public BigDecimal apply(BigDecimal amount) { |
|  |

|  |
| --- |
| return amount.multiply(BigDecimal.valueOf(0.5)); |
| |  | | --- | |  |  |  | | --- | |  | |  |  |  | | --- | |  |  |  | | --- | |  | |  |  |  | | --- | | final BigDecimal discountedValue = staffDiscounter | |  |  |  | | --- | | .apply(BigDecimal.valueOf(100)); | |  |  |  | | --- | |  | |  |  |  | | --- | | assertThat(discountedValue) | |  |  |  | | --- | | .isEqualByComparingTo(BigDecimal.valueOf(50)); | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | | @Test | |  |  |  | | --- | | public void shouldDivideByTwo\_WhenApplyingStaffDiscounterWithLamda() { | |  |  |  | | --- | | Discounter staffDiscounter = amount -> amount.multiply(BigDecimal.valueOf(0.5)); | |  |  |  | | --- | |  | |  |  |  | | --- | | final BigDecimal discountedValue = staffDiscounter | |  |  |  | | --- | | .apply(BigDecimal.valueOf(100)); | |  |  |  | | --- | |  | |  |  |  | | --- | | assertThat(discountedValue) | |  |  |  | | --- | | .isEqualByComparingTo(BigDecimal.valueOf(50)); | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | | @Test | |  | | final Discounter combinedDiscounter = discounters | | |  | |  |  | | --- | | .stream() |  |  | | --- | | public void shouldApplyAllDiscounts() { | |  |  |  | | --- | | List<Discounter> discounters = Arrays.asList(christmas(), newYear(), easter()); | |  |  |  | | --- | |  | |  |  |  | | --- | | BigDecimal amount = BigDecimal.valueOf(100); | |  |  |  | | --- | |  | |  |  |  | | --- | |  |  |  | | --- | | .reduce(v -> v, Discounter::combine); | |  |  |  | | --- | |  | |  |  |  | | --- | | combinedDiscounter.apply(amount); | |  |  |  | | --- | | } | |  |  |  | | --- | |  | |  |  |  | | --- | | @Test | |  |  |  | | --- | | public void shouldChainDiscounters() { | |  |  |  | | --- | | final Function<BigDecimal, BigDecimal> combinedDiscounters = Discounter | |  |  |  | | --- | | .christmas() | |  |  |  | | --- | | .andThen(newYear()); | |  |  |  | | --- | |  | |  |  |  | | --- | | combinedDiscounters.apply(BigDecimal.valueOf(100)); | |  |  |  | | --- | | } | |  |  Java 8 Functional Interfaces It is not hard to find more examples of the Strategy design pattern in the JDK, and it is even simpler starting from version 8. Making a quick reference search for the new functional interfaces (Predicate, Function, Consumer, Supplier etc) makes it clear that this pattern is used literally everywhere.  Just one example: ArrayList defines a method forEach, that takes as parameter a Consumer. The ArrayList class has no knowledge of the specific Consumers, it is only aware of the interface. The client of the collection has the responsibility to pass in the desired implementation of that interface. Of course, these Consumers usually do not manifest as a class, but are rather represented as lambda expressions.  HashMap and ConcurrentHashMap also rely heavily on the functional interfaces Function and BiFunction for computing and merging entries. |