STRATEGY

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Strategy pattern is also known as **Policy Pattern**. We define multiple algorithms and let client application pass the algorithm to be used as a parameter. One of the best example of strategy pattern is `Collections.sort()` method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways. For our example, we will try to implement a simple Shopping Cart where we have two payment strategies - using Credit Card or using PayPal. First of all we will create the interface for our strategy pattern example, in our case to pay the amount passed as argument. `PaymentStrategy.java`

```
package com.journaldev.design.strategy;

public interface PaymentStrategy {

	public void pay(int amount);
}
```

```
public class CreditCardStrategy implements PaymentStrategy {

	private String name;
	private String cardNumber;
	private String cvv;
	private String dateOfExpiry;

	public CreditCardStrategy(String nm, String ccNum, String cvv, String expiryDate){
		this.name=nm;
		this.cardNumber=ccNum;
		this.cvv=cvv;
		this.dateOfExpiry=expiryDate;
	}
	@Override
	public void pay(int amount) {
		System.out.println(amount +" paid with credit/debit card");
	}

}
```

```
public class PaypalStrategy implements PaymentStrategy {

	private String emailId;
	private String password;

	public PaypalStrategy(String email, String pwd){
		this.emailId=email;
		this.password=pwd;
	}

	@Override
	public void pay(int amount) {
		System.out.println(amount + " paid using Paypal.");
	}

}
```

```java
public class Item {

	private String upcCode;
	private int price;

	public Item(String upc, int cost){
		this.upcCode=upc;
		this.price=cost;
	}

	public String getUpcCode() {
		return upcCode;
	}

	public int getPrice() {
		return price;
	}

}
```

```java
public class ShoppingCart {

	//List of items
	List<Item> items;

	public ShoppingCart(){
		this.items=new ArrayList<Item>();
	}

	public void addItem(Item item){
		this.items.add(item);
	}

	public void removeItem(Item item){
		this.items.remove(item);
	}

	public int calculateTotal(){
		int sum = 0;
		for(Item item : items){
			sum += item.getPrice();
		}
		return sum;
	}

	public void pay(PaymentStrategy paymentMethod){
		int amount = calculateTotal();
		paymentMethod.pay(amount);
	}
}
```

Notice that payment method of shopping cart requires payment algorithm as argument and doesn't store it anywhere as instance variable. Let's test our strategy pattern example setup with a simple program. `ShoppingCartTest.java`

```java
package com.journaldev.design.strategy;

public class ShoppingCartTest {

	public static void main(String[] args) {
		ShoppingCart cart = new ShoppingCart();

		Item item1 = new Item("1234",10);
		Item item2 = new Item("5678",40);

		cart.addItem(item1);
		cart.addItem(item2);

		//pay by paypal
		cart.pay(new PaypalStrategy("myemail@example.com", "mypwd"));

		//pay by credit card
		cart.pay(new CreditCardStrategy("Pankaj Kumar", "1234567890123456", "786", "12/1
	}

}
```

Output of above program is:

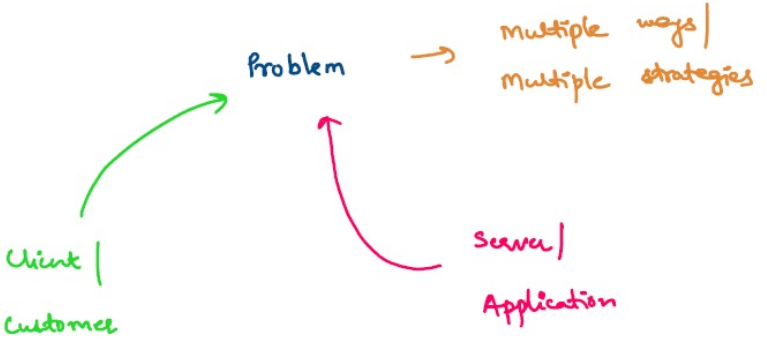
```
50 paid using Paypal.
50 paid with credit/debit card
```

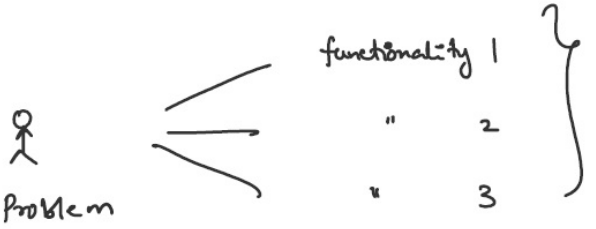
THIS MAKES US UNDERSTAND THE PATTERN

The Strategy pattern provides several advantages, including:

1. Flexibility: The Strategy pattern allows the behavior of an object to be changed dynamically at runtime by selecting different algorithms.
2. Modularity: The pattern encapsulates the algorithms in separate classes, making it easy to add or remove algorithms without affecting other parts of the code.
3. Testability: The pattern makes it easy to test the different algorithms separately, without affecting the overall behavior of the code.
4. Open-Closed Principle: The Strategy pattern follows the Open-Closed Principle, which states that a class should be open for extension but closed for modification.
5. However, the Strategy pattern can also have some drawbacks, including increased complexity due to the use of multiple classes, and potential performance issues if the selection of algorithms is done frequently at runtime.

Used when we have multiple ways to solve a problem

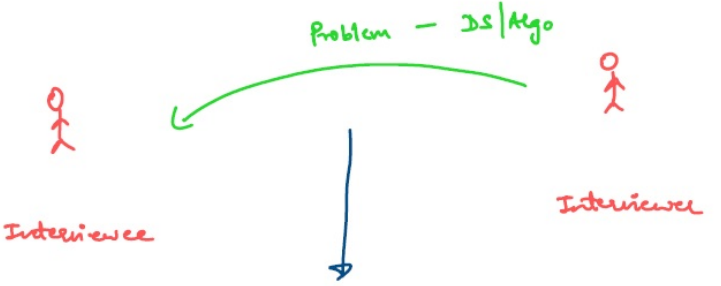




In strategy pattern, a class behavior or its algorithm can be changed at run time.

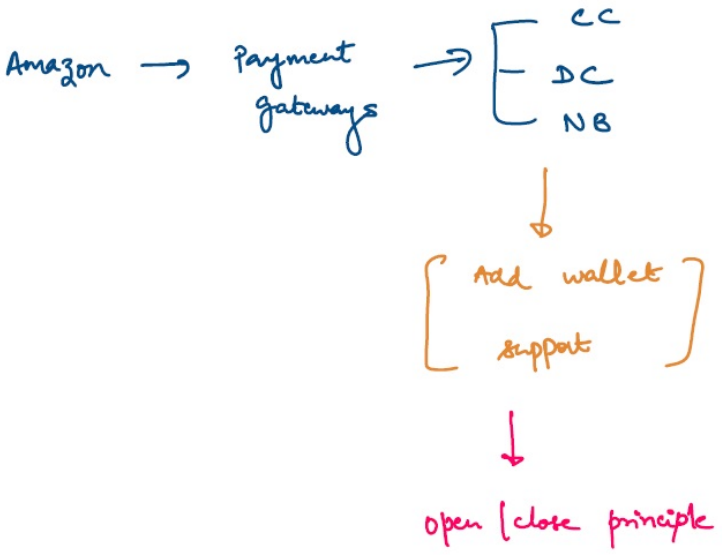
like we change the way Collections.sort() works at runtime by just providing comparator

In strategy design pattern, we create objects which represent various strategies and a context object whose behaviour varies as per its strategy object. This strategy object changes the executing algorithm of the context object.



1. Brute force
2. Optimize with extra space
3. Optimize with no extra space.

various strategies



follows open/close principle as we payment methods can be added later without chnaging the initial code .we can extend to multiple payment methods

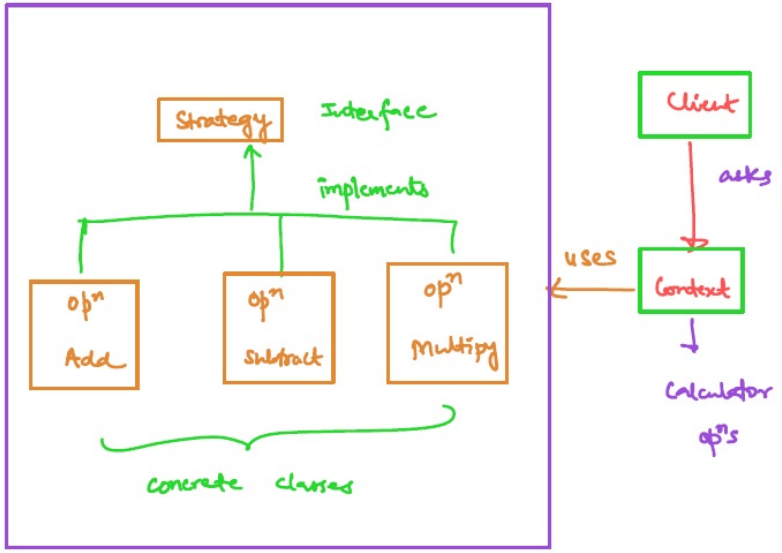
In this pattern we provide family of functionality

Benefits / Advantages –

1. It makes it easier to extend and incorporate new behavior without changing the application.

2. It defines each behavior within its own class, eliminating the need for conditional statements.

3. It provides a substitute for sub-classing.



in prev eg shoppingCart was the context

usages of strategy design pattern –

1. when we need the different variations of an algorithm.
2. when the multiple classes differ only in their behaviors.

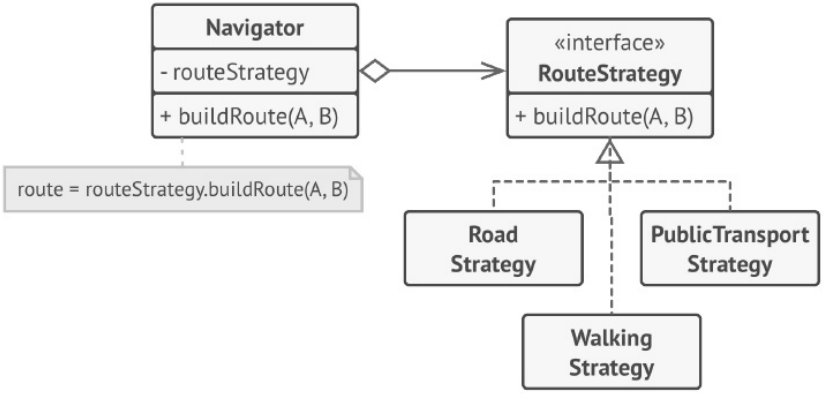
same implementation was given by maam

The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*.

The original class, called *context*, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

The context isn't responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. In fact, the context doesn't know much about strategies. It works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy.

This way the context becomes independent of concrete strategies, so you can add new algorithms or modify existing ones without changing the code of the context or other strategies.



Route planning strategies.

⚖ Pros and Cons

- ✓ You can swap algorithms used inside an object at runtime.
- ✓ You can isolate the implementation details of an algorithm from the code that uses it.
- ✓ You can replace inheritance with composition.
- ✓ *Open/Closed Principle*. You can introduce new strategies without having to change the context.
- ✕ If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
- ✕ Clients must be aware of the differences between strategies to be able to select a proper one.
- ✕ A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you'd have used the strategy objects, but without bloating your code with extra classes and interfaces.

⇄ Relations with Other Patterns

- **Bridge**, **State**, **Strategy** (and to some degree **Adapter**) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.

- **Command** and **Strategy** may look similar because you can use both to parameterize an object with some action. However, they have very different intents.
  - You can use *Command* to convert any operation into an object. The operation's parameters become fields of that object. The conversion lets you defer execution of the operation, queue it, store the history of commands, send commands to remote services, etc.
  - On the other hand, *Strategy* usually describes different ways of doing the same thing, letting you swap these algorithms within a single context class.

- **Decorator** lets you change the skin of an object, while **Strategy** lets you change the guts.

suppose interviewer asks you to design to payment gateway !! dont go directly to strategy design pattern so ask him what are payment options !! and so on ask hium questions!!

steps:-

1.create a interface which you want to extend which tells various strategies

2.create concreate classes of the inteface

3.create context class here we have a shopping cart

4.create client class