## Behavior Design Pattern:

Behavioral design patterns are a set of solutions to common problems related to how objects in a software system communicate and work together to accomplish a task or goal.

They focus on the behavior of objects and how they interact with each other, and provide ways to manage complex interactions between objects or ensure that different parts of a system work together seamlessly.

There are several behavioral design patterns, including:

1. Observer Pattern: This pattern defines a one-to-many relationship between objects, so that when one object changes state, all its dependents are notified and updated automatically. This pattern is useful when you have objects that need to be notified when something changes, but you don't want to tightly couple them together.
2. Strategy Pattern: This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows you to select an algorithm at runtime, based on the situation. This pattern is useful when you have a set of algorithms that can be used interchangeably, depending on the context.
3. Command Pattern: This pattern encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. This pattern is useful when you need to separate the object that invokes a command from the object that knows how to execute it.
4. Template Method Pattern: This pattern defines the skeleton of an algorithm in a base class, but lets subclasses override specific steps of the algorithm without changing its structure. This pattern is useful when you have a set of similar algorithms with slight variations.
5. Iterator Pattern: This pattern provides a way to access the elements of an aggregate object sequentially, without exposing its underlying representation. This pattern is useful when you have a collection of objects that you want to iterate over without exposing the internal implementation of the collection.

### Strategy Design Pattern:

The Strategy Pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows you to select an algorithm at runtime, based on the situation.

##### Example 1:

Suppose you are building a shopping cart system for an online store, where customers can buy different types of products. Each product type has a different shipping cost calculation algorithm. For example, some products may have free shipping, while others may have a fixed shipping cost or a percentage-based shipping cost.

In this case, you can use the Strategy Pattern to encapsulate each shipping cost calculation algorithm into its own class. Each class implements a common interface or abstract class that defines the calculation method.

For example, you can have three classes: FreeShippingCostCalculator, FixedShippingCostCalculator, and PercentageShippingCostCalculator. Each class would have its own implementation of the calculateShippingCost() method, based on the specific algorithm.

Then, you can create a ShippingCostCalculator class that accepts a specific shipping cost calculation algorithm as a parameter. This class would use the algorithm to calculate the shipping cost for a given product.

Finally, you can create a ShoppingCart class that uses the ShippingCostCalculator to calculate the total shipping cost for all the products in the cart.

By using the Strategy Pattern, you can easily switch between different shipping cost calculation algorithms at runtime, without changing the implementation of the ShoppingCart class. For example, you can select the FreeShippingCostCalculator for products with free shipping or the PercentageShippingCostCalculator for products with a percentage-based shipping cost.

| public interface ShippingCostCalculator {  double calculateShippingCost(double totalPrice); } public class FreeShippingCostCalculator implements ShippingCostCalculator {  @Override  public double calculateShippingCost(double totalPrice) {  return 0.0; // Free shipping for all products  } }  public class FixedShippingCostCalculator implements ShippingCostCalculator {  private double fixedCost;   public FixedShippingCostCalculator(double fixedCost) {  this.fixedCost = fixedCost;  }   @Override  public double calculateShippingCost(double totalPrice) {  return fixedCost; // Fixed cost for all products  } }  public class PercentageShippingCostCalculator implements ShippingCostCalculator {  private double percentage;   public PercentageShippingCostCalculator(double percentage) {  this.percentage = percentage;  }   @Override  public double calculateShippingCost(double totalPrice) {  return totalPrice \* percentage; // Percentage of the total price  } } public class ShippingCostContext {  private ShippingCostCalculator calculator;   public ShippingCostContext(ShippingCostCalculator calculator) {  this.calculator = calculator;  }   public double calculateShippingCost(double totalPrice) {  return calculator.calculateShippingCost(totalPrice);  } } import java.util.ArrayList; import java.util.List;  public class ShoppingCart {  private List < Product > products = new ArrayList < > ();   public void addProduct(Product product) {  products.add(product);  }   public void removeProduct(Product product) {  products.remove(product);  }   public double getTotalPrice() {  double totalPrice = 0.0;  for (Product product: products) {  totalPrice += product.getPrice();  }  return totalPrice;  }   public double getShippingCost(ShippingCostCalculator calculator) {  double totalPrice = getTotalPrice();  ShippingCostContext context = new ShippingCostContext(calculator);  return context.calculateShippingCost(totalPrice);  } } public class Main {  public static void main(String[] args) {  // Create some products  Product product1 = new Product("Product 1", 10.0);  Product product2 = new Product("Product 2", 20.0);  Product product3 = new Product("Product 3", 30.0);   // Add products to the cart  ShoppingCart cart = new ShoppingCart();  cart.addProduct(product1);  cart.addProduct(product2);  cart.addProduct(product3);   // Calculate the total price and shipping cost  double totalPrice = cart.getTotalPrice();  double shippingCost = cart.getShippingCost(new PercentageShippingCostCalculator(0.1)); // 10% of the total price  double totalCost = totalPrice + shippingCost;   // Display the results  System.out.println("Total Price: $" + totalPrice);  System.out.println("Shipping Cost: $" + shippingCost);  System.out.println("Total Cost: $" + totalCost);  } } |
| --- |

##### Example 2:

How to use the Strategy Pattern to calculate the best route on Google Maps using different settings:

First, let's create the interface RouteCalculator that defines the common method calculateRoute():

| public interface RouteCalculator {  String calculateRoute(Location start, Location end); } |
| --- |

Next, let's create the concrete classes that implement this interface for each type of route calculation:

| public class FastestRouteCalculator implements RouteCalculator {  @Override  public String calculateRoute(Location start, Location end) {  // Calculate the fastest route using Google Maps API  return "Fastest route from " + start + " to " + end;  } }  public class ShortestRouteCalculator implements RouteCalculator {  @Override  public String calculateRoute(Location start, Location end) {  // Calculate the shortest route using Google Maps API  return "Shortest route from " + start + " to " + end;  } }  public class ScenicRouteCalculator implements RouteCalculator {  @Override  public String calculateRoute(Location start, Location end) {  // Calculate the scenic route using Google Maps API  return "Scenic route from " + start + " to " + end;  } } |
| --- |

Now, let's create the RouteCalculatorContext class that accepts a specific RouteCalculator and uses it to calculate the best route:

| public class RouteCalculatorContext {  private RouteCalculator calculator;    public RouteCalculatorContext(RouteCalculator calculator) {  this.calculator = calculator;  }    public String calculateRoute(Location start, Location end) {  return calculator.calculateRoute(start, end);  } } |
| --- |

Finally, let's create the GoogleMaps class that uses the RouteCalculatorContext to calculate the best route using different settings:

| public class GoogleMaps {  private RouteCalculatorContext context;    public GoogleMaps(RouteCalculator calculator) {  this.context = new RouteCalculatorContext(calculator);  }    public void setRouteCalculator(RouteCalculator calculator) {  this.context = new RouteCalculatorContext(calculator);  }    public String calculateRoute(Location start, Location end) {  return context.calculateRoute(start, end);  } } |
| --- |

Here's how you can use these classes to calculate the best route on Google Maps using different settings:

| public class Main {  public static void main(String[] args) {  // Create a Google Maps instance with the default calculator (fastest route)  GoogleMaps maps = new GoogleMaps(new FastestRouteCalculator());    // Calculate the best route from New York to Boston using the default calculator  Location start = new Location("New York");  Location end = new Location("Boston");  String route = maps.calculateRoute(start, end);  System.out.println(route);    // Change the calculator to the shortest route  maps.setRouteCalculator(new ShortestRouteCalculator());    // Calculate the best route from New York to Boston using the shortest route calculator  route = maps.calculateRoute(start, end);  System.out.println(route);    // Change the calculator to the scenic route  maps.setRouteCalculator(new ScenicRouteCalculator());    // Calculate the best route from New York to Boston using the scenic route calculator  route = maps.calculateRoute(start, end);  System.out.println(route);  } } |
| --- |

##### Example 3:

Let's say we have a mobile application that needs to send push notifications to its users. The application needs to support multiple platforms such as iOS, Android, and Windows. Each platform has its own way of sending push notifications. We can use the Strategy Pattern to implement this functionality.

First, let's create the interface PushNotificationStrategy that defines the common method sendPushNotification():

| public interface PushNotificationStrategy {  void sendPushNotification(String message); } |
| --- |

Next, let's create the concrete classes that implement this interface for each platform:

| public class IOSPushNotificationStrategy implements PushNotificationStrategy {  @Override  public void sendPushNotification(String message) {  // Send push notification using iOS push notification service  System.out.println("Sending push notification to iOS devices: " + message);  } }  public class AndroidPushNotificationStrategy implements PushNotificationStrategy {  @Override  public void sendPushNotification(String message) {  // Send push notification using Android push notification service  System.out.println("Sending push notification to Android devices: " + message);  } }  public class WindowsPushNotificationStrategy implements PushNotificationStrategy {  @Override  public void sendPushNotification(String message) {  // Send push notification using Windows push notification service  System.out.println("Sending push notification to Windows devices: " + message);  } } |
| --- |

Now, let's create the PushNotificationContext class that accepts a specific

| PushNotificationStrategy and uses it to send push notifications: public class PushNotificationContext {  private PushNotificationStrategy strategy;    public PushNotificationContext(PushNotificationStrategy strategy) {  this.strategy = strategy;  }    public void setStrategy(PushNotificationStrategy strategy) {  this.strategy = strategy;  }    public void sendPushNotification(String message) {  strategy.sendPushNotification(message);  } } |
| --- |

Finally, let's create the MobileApp class that uses the PushNotificationContext to send push notifications to its users on different platforms:

| public class MobileApp {  private PushNotificationContext context;    public MobileApp(PushNotificationStrategy strategy) {  this.context = new PushNotificationContext(strategy);  }    public void setPushNotificationStrategy(PushNotificationStrategy strategy) {  this.context.setStrategy(strategy);  }    public void sendPushNotification(String message) {  context.sendPushNotification(message);  } } |
| --- |

Here's how you can use these classes to send push notifications to users on different platforms:

| public class Main {  public static void main(String[] args) {  // Create a mobile app instance with the default push notification strategy (iOS)  MobileApp app = new MobileApp(new IOSPushNotificationStrategy());    // Send push notification to all users using the default strategy (iOS)  app.sendPushNotification("New update available!");    // Change the push notification strategy to Android  app.setPushNotificationStrategy(new AndroidPushNotificationStrategy());    // Send push notification to all users using the Android strategy  app.sendPushNotification("New update available!");    // Change the push notification strategy to Windows  app.setPushNotificationStrategy(new WindowsPushNotificationStrategy());    // Send push notification to all users using the Windows strategy  app.sendPushNotification("New update available!");  } } |
| --- |

### Observer Design Pattern:

The Observer Design Pattern is typically used in scenarios where there is a one-to-many relationship between objects, such that changes to one object need to be reflected in one or more other objects. Here are some common use cases where the Observer Pattern can be applied:

#### Examples:

##### Event handling:

When an event occurs, all the registered observers are notified and the appropriate action is taken.

| import java.util.ArrayList; import java.util.List;  public class Button {  private List<ClickListener> clickListeners = new ArrayList<>();   public void addClickListener(ClickListener listener) {  clickListeners.add(listener);  }   public void click() {  for (ClickListener listener : clickListeners) {  listener.onClick();  }  } }  public interface ClickListener {  void onClick(); }  public class DisplayUpdater implements ClickListener {  public void onClick() {  System.out.println("Updating display...");  } }  public class Calculator implements ClickListener {  public void onClick() {  System.out.println("Performing calculation...");  } }  public class Main {  public static void main(String[] args) {  Button button = new Button();  button.addClickListener(new DisplayUpdater());  button.addClickListener(new Calculator());  button.click();  } } |
| --- |

In this example, we have a Button class that has a list of ClickListener objects. When the button is clicked, it notifies all the registered ClickListener objects by calling their onClick method. The Main class creates a new Button object, registers two ClickListener objects with it, and then clicks the button.

##### User interface components:

When the state of a component changes, such as a checkbox being checked, all the registered observers are notified and the appropriate action is taken.

| import java.util.ArrayList; import java.util.List;  public class Checkbox {  private boolean checked;  private List<ChangeListener> changeListeners = new ArrayList<>();   public void addChangeListener(ChangeListener listener) {  changeListeners.add(listener);  }   public boolean isChecked() {  return checked;  }   public void setChecked(boolean checked) {  this.checked = checked;  for (ChangeListener listener : changeListeners) {  listener.onChange(checked);  }  } }  public interface ChangeListener {  void onChange(boolean checked); }  public class DisplayUpdater implements ChangeListener {  public void onChange(boolean checked) {  System.out.println("Checkbox is now " + (checked ? "checked" : "unchecked"));  } }  public class DatabaseUpdater implements ChangeListener {  public void onChange(boolean checked) {  System.out.println("Updating database with checkbox state...");  } }  public class Main {  public static void main(String[] args) {  Checkbox checkbox = new Checkbox();  checkbox.addChangeListener(new DisplayUpdater());  checkbox.addChangeListener(new DatabaseUpdater());  checkbox.setChecked(true);  } }  In this example, we have a Checkbox class that has a boolean checked field and a list of ChangeListener objects. When the checked field is updated, it notifies all the registered ChangeListener objects by calling their onChange method with the new value of checked. The Main class creates a new Checkbox object, registers two ChangeListener objects with it, and then sets the checkbox to be checked. |
| --- |

##### Stock market data:

When the value of a stock changes, all the registered observers are notified and the appropriate action is taken, such as updating a graph or triggering an alert.

| import java.util.ArrayList; import java.util.List;  public class Stock {  private double value;  private List < ValueChangeListener > valueChangeListeners = new ArrayList < > ();   public void addValueChangeListener(ValueChangeListener listener) {  valueChangeListeners.add(listener);  }   public double getValue() {  return value;  }   public void setValue(double value) {  this.value = value;  for (ValueChangeListener listener: valueChangeListeners) {  listener.onValueChange(value);  }  } }  public interface ValueChangeListener {  void onValueChange(double newValue); }  public class GraphUpdater implements ValueChangeListener {  public void onValueChange(double newValue) {  System.out.println("Updating graph with new value: " + newValue);  } }  public class EmailSender implements ValueChangeListener {  public void onValueChange(double newValue) {  System.out.println("Sending email with new value: " + newValue);  } }  /\* To complete the code, add the missing closing bracket and create an instance of the Stock class with some initial value, and add the GraphUpdater and EmailSender as listeners to the Stock object.  \*/ public class Main {  public static void main(String[] args) {  Stock stock = new Stock();  stock.setValue(100.0); // Set an initial value   GraphUpdater graphUpdater = new GraphUpdater();  EmailSender emailSender = new EmailSender();   stock.addValueChangeListener(graphUpdater);  stock.addValueChangeListener(emailSender);   // Update the value of the stock  stock.setValue(110.0);  } } |
| --- |

##### Weather monitoring:

When the weather changes, all the registered observers are notified and the appropriate action is taken, such as updating a display or triggering an alert.

| import java.util.ArrayList; import java.util.List;  interface Observer {  void update(float temperature, float humidity, float pressure); }  interface Subject {  void registerObserver(Observer observer);  void removeObserver(Observer observer);  void notifyObservers(); }  class WeatherData implements Subject {  private List<Observer> observers;  private float temperature;  private float humidity;  private float pressure;    public WeatherData() {  observers = new ArrayList<>();  }   public void registerObserver(Observer observer) {  observers.add(observer);  }   public void removeObserver(Observer observer) {  observers.remove(observer);  }   public void notifyObservers() {  for (Observer observer : observers) {  observer.update(temperature, humidity, pressure);  }  }   public void measurementsChanged() {  notifyObservers();  }   public void setMeasurements(float temperature, float humidity, float pressure) {  this.temperature = temperature;  this.humidity = humidity;  this.pressure = pressure;  measurementsChanged();  } }  class CurrentConditionsDisplay implements Observer {  private float temperature;  private float humidity;    public void update(float temperature, float humidity, float pressure) {  this.temperature = temperature;  this.humidity = humidity;  display();  }    public void display() {  System.out.println("Current conditions: " + temperature + "F degrees and " + humidity + "% humidity");  } }  public class WeatherStation {  public static void main(String[] args) {  WeatherData weatherData = new WeatherData();    CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay();    weatherData.registerObserver(currentDisplay);    weatherData.setMeasurements(80, 65, 30.4f);  weatherData.setMeasurements(82, 70, 29.2f);  weatherData.setMeasurements(78, 90, 29.2f);  } } |
| --- |

In this example, the WeatherData class is the subject that the observers (in this case, the CurrentConditionsDisplay) register with and receive updates from. The WeatherData class keeps track of the temperature, humidity, and pressure, and calls the measurementsChanged method to notify its observers whenever these values change.

The CurrentConditionsDisplay class implements the Observer interface and defines the update method, which is called by the WeatherData subject whenever there is a change in the weather measurements. The display method is called to show the updated weather conditions.

In the WeatherStation class, we create a new instance of WeatherData and CurrentConditionsDisplay, register the display as an observer of the weather data, and then simulate updates to the weather measurements by calling the setMeasurements method on the weather data object. The CurrentConditionsDisplay object automatically updates and displays the new weather conditions.

##### Database management systems:

When the state of a database changes, such as a record being updated, all the registered observers are notified and the appropriate action is taken, such as updating a cache or triggering a backup.

| public class DatabaseRecord {  private int id;  private String name;  private String email;   // constructor, getters and setters } public interface DatabaseRecordObserver {  void onRecordUpdated(DatabaseRecord record); }  public class CacheUpdater implements DatabaseRecordObserver {  public void onRecordUpdated(DatabaseRecord record) {  // Update the cache with the new record data  // ...  System.out.println("Cache updated with new record: " + record.getId());  } }  public class UserNotifier implements DatabaseRecordObserver {  public void onRecordUpdated(DatabaseRecord record) {  // Send a notification to the user with the new record data  // ...  System.out.println("User notified about new record: " + record.getId());  } } import java.util.ArrayList; import java.util.List;  public class Database {  private List<DatabaseRecordObserver> observers = new ArrayList<>();   public void addRecordObserver(DatabaseRecordObserver observer) {  observers.add(observer);  }   public void updateRecord(DatabaseRecord record) {  // Save the updated record to the database  // ...   // Notify all observers that the record has been updated  for (DatabaseRecordObserver observer : observers) {  observer.onRecordUpdated(record);  }  } } public class Main {  public static void main(String[] args) {  // Create a new database  Database database = new Database();   // Register the CacheUpdater and UserNotifier as observers  CacheUpdater cacheUpdater = new CacheUpdater();  UserNotifier userNotifier = new UserNotifier();  database.addRecordObserver(cacheUpdater);  database.addRecordObserver(userNotifier);   // Update a record  DatabaseRecord record = new DatabaseRecord(1, "John Doe", "johndoe@example.com");  database.updateRecord(record);  } } |
| --- |

When the updateRecord method is called on the Database object, it will notify all its observers that the record has been updated. The CacheUpdater and UserNotifier objects will receive the notification and perform their specific actions. In this example, the CacheUpdater will update the cache and print a message to the console, and the UserNotifier will send a notification to the user and print a message to the console.

In general, the Observer Pattern is useful in any situation where multiple objects need to be kept in sync with one another, without tightly coupling them together.