B.A.N.K.A- Clothing Store

Rewards Programs

BY: Amalia, Nick, Jaskiran(Kiran), Benjamin, Andreea

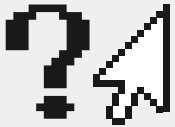
Agenda

WalkThrough	
Purpose	Purpose of B.A.N.K.A Clothing Store and Our Rewards Program
UML DIAGRAMS FOR DESIGN PATTERNS	UML Diagrams and Designs for each Design Pattern
Design Patterns with Java	Each Design Pattern Implemented with Java
Unit Test	Unit Tests and It’s Implementation
1 Component Test	Component Test
Diagrams	Context and Process Flow Diagrams

01

Purpose

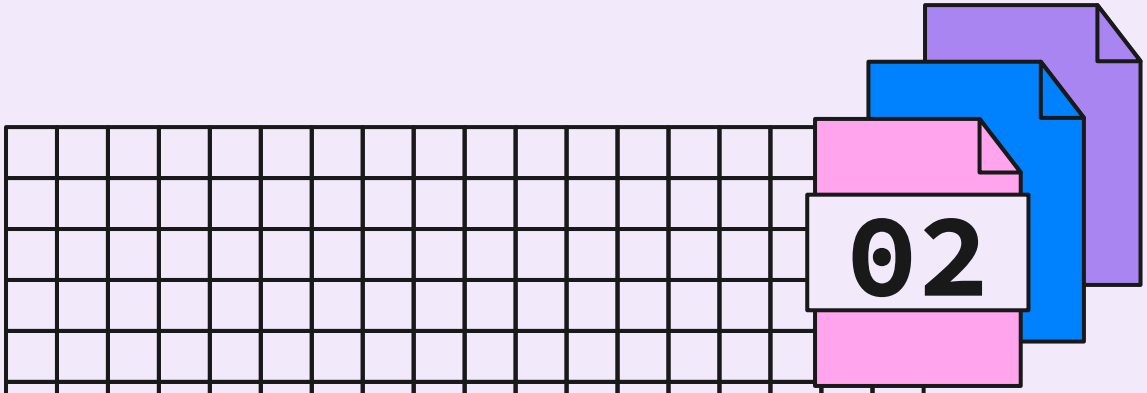
The Purpose of B.A.N.K.A



Purpose

B.A.N.K.A Clothing Line is a fashion brand that provides stylish and cost-effective clothing.
The company values customer satisfaction and loyalty, and has implemented a new reward system to enhance this:

- The reward system aims to encourage customers to engage with the brand by offering them reward points.
- Customers can earn points through actions such as making purchases, referring friends, engaging with social media, leaving product reviews, and marking special occasions.
- Reward points can be redeemed towards future purchases or other rewards.
- The reward system is intended to improve the customer experience, incentivize repeat purchases, and foster customer loyalty.

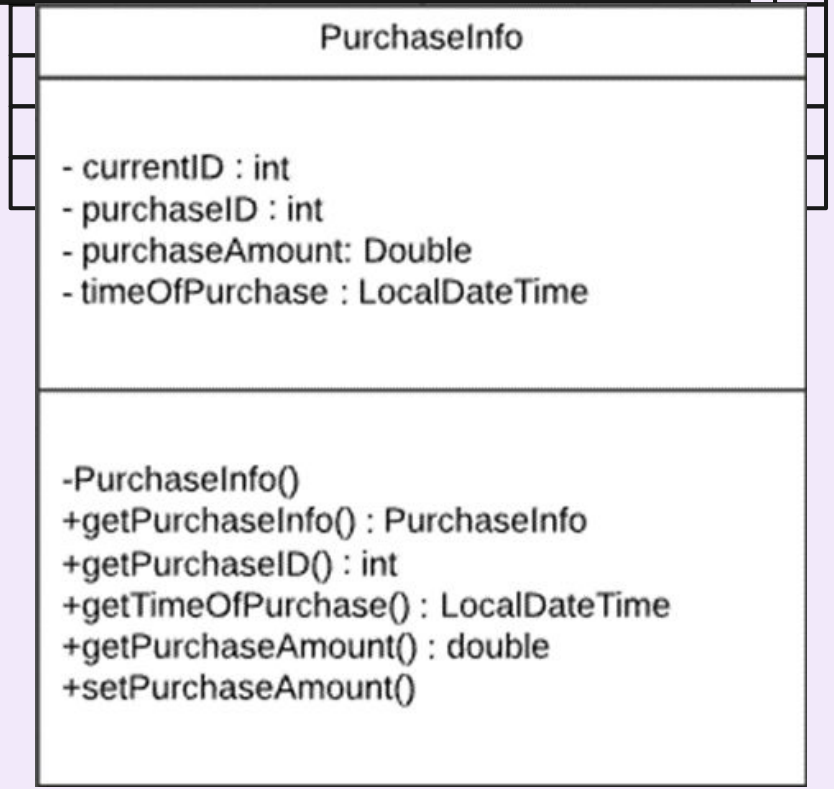


UML DIAGRAMS

UML Designs with our Patterns

UML Design description: Singleton

- Single class UML diagram
- Holds 4 private variables: currentID, purchaseID, purchaseAmount, timeOfPurchase
- Getters are available for purchaseID, purchaseAmount, and timeOfPurchase; the value of currentID cannot be retrieved or accessed by others as it is the center of the Singleton.
- Constructor is private, called only in the getter method getPurchaseInfo(), which returns a PurchaseInfo object and assigns it a unique ID.

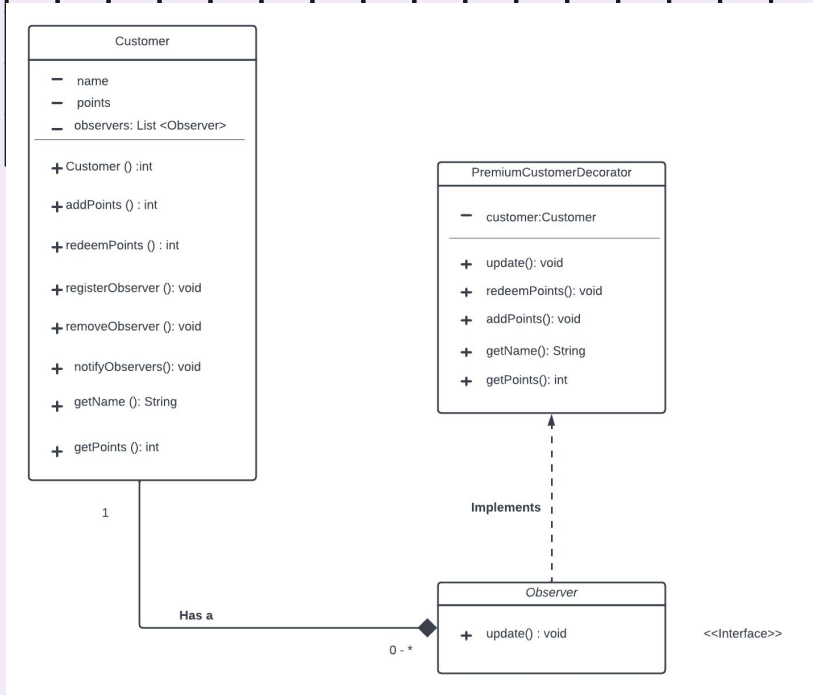


UML Design description: Decorator

The customer class has a one to many relationship with Observer

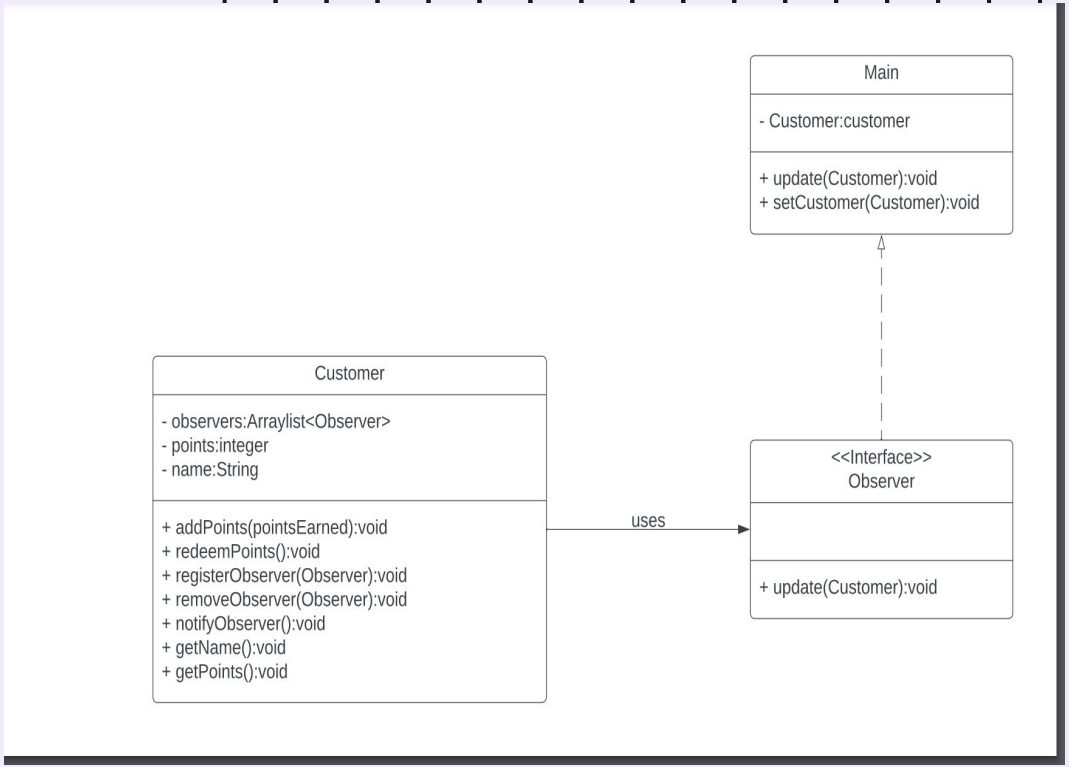
Premium Customer Decorator implements observer

The PremiumCustomerDecorator class decorates the customer class by adding an extra behavior to it such as checking if a member is a premium customer or not



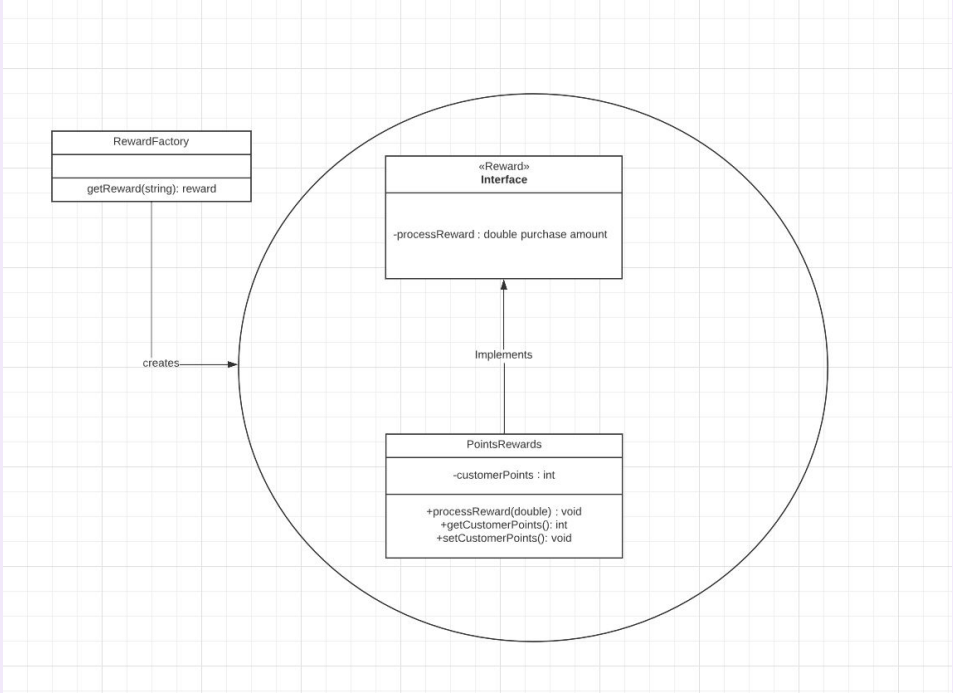
UML Design description: Observer

- The Main class implements the Observer interface.
- It includes the methods update () (gives the user an update on their information) and setCustomer().
- The customer class uses the Observer interface and uses the update () method as defined in Observer.
- The customer class has the methods registerObserver(), removeObserver(), and notifyObservers() interact with the attached observers.
- Whilst the addPoints(), redeemPoints(), getName() and getPoints() all deal with the customer and their attributes.

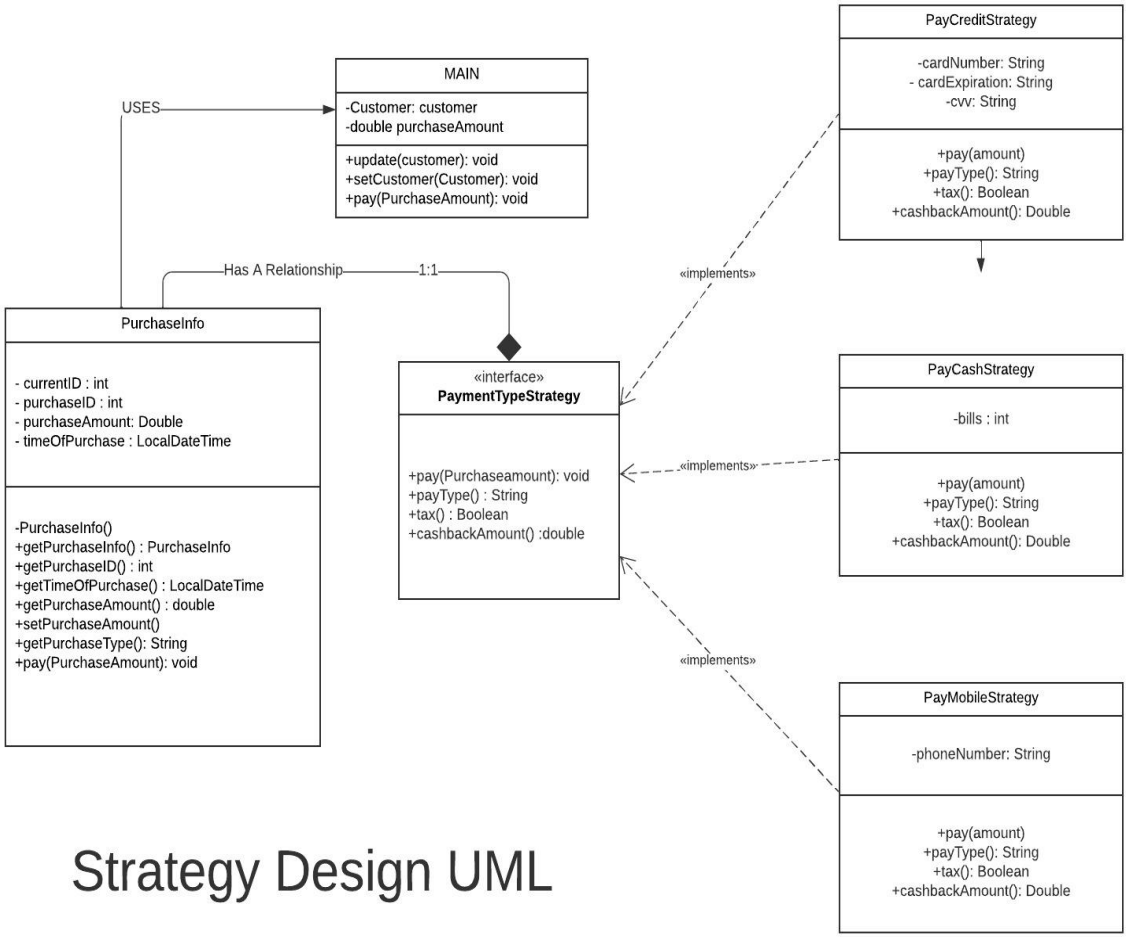


UML Design description: Factory

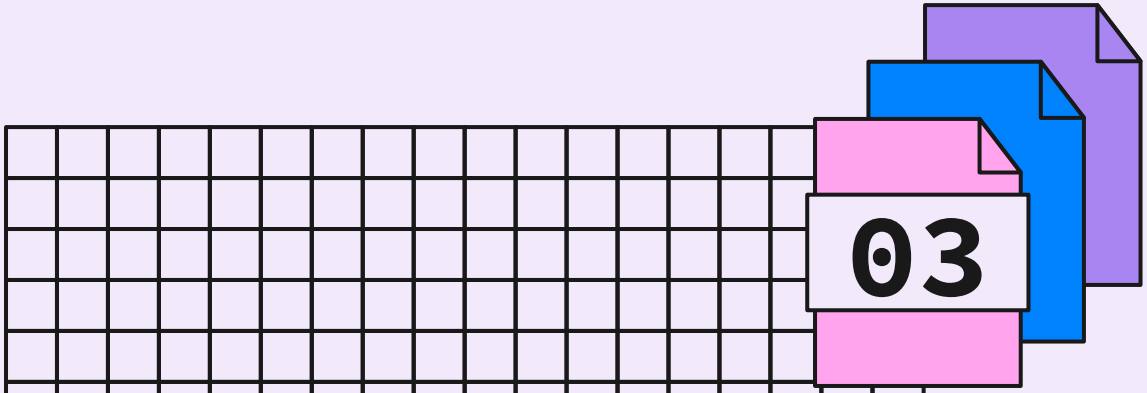
Factory design is extremely versatile, allowing for the creation of additional objects as seen with the RewardsFactory Class. It allows different types of rewards to be generated as needed while maintaining their individual ways of invoking processReward() that is unique to their class.



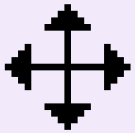
- The Purchase Info class reference to the Payment Type interface
- Payment Type interface defines the behavior abstraction for the payment types and declares the pay(), tax(), cashback(), type() methods
- The Mobile, Credit, and Cash classes are the implementations of the Payment Type interface



Strategy Design UML



Java Classes with our design



Java Classes with our Design

Main Function for Rewards

```java
import java.util.Scanner;

public class Main implements Observer {
  static final int POINTS_PER_DOLLAR = 10;
  private Customer customer;

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    // Get customer information from user input
    System.out.println("Welcome to the Rewards Program!");
    System.out.println("Please enter your name: ");
    String customerName = scanner.nextLine();
    int customerPoints = 0;

    // Create Customer object and register the RewardsProgram as an observer
    Customer customer = new Customer(customerName, customerPoints);
    Main rewardsProgram = new Main();
    customer.registerObserver(rewardsProgram);
    rewardsProgram.setCustomer(customer);

    // Main program loop
    while (true) {
      // Display customer information and menu options
      System.out.println("Customer: " + customer.getName());
      System.out.println("Points: " + customer.getPoints());
      System.out.println("1. Make a purchase");
      System.out.println("2. Redeem points");
      System.out.println("3. Exit");
      System.out.println("Please enter an option (1-3): ");

      // Get user input and perform action based on selected option
      int option = scanner.nextInt();
      switch (option) {
```

Main.java

```java
      switch (option) {
        case 1: // Make a purchase
          // Get purchase amount from user input
          System.out.println("Please enter the purchase amount: ");
          double purchaseAmount = scanner.nextDouble();
          // Calculate points earned and add them to the Customer
          int pointsEarned = (int) (purchaseAmount * POINTS_PER_DOLLAR);
          customer.addPoints(pointsEarned);
          break;
        case 2: // Redeem points
          customer.redeemPoints();
          break;
        case 3: // Exit
          System.exit(0);
          break;
        default:
          System.out.println("Invalid option. Please enter a number between 1 and 3.");
      }
    }
  }

  public void update(Customer customer) {
    System.out.println("Congratulations, " + customer.getName() + "!");
    System.out.println("You have earned " + customer.getPoints() + " points!");
  }

  public void setCustomer(Customer customer) {
    this.customer = customer;
  }
}
```

Singleton

```java
import java.time.LocalDateTime;
public class PurchaseInfo
{
    static private int _currentID = 1;
    private int purchaseID;
    private LocalDateTime timeOfPurchase;
    private double purchaseAmount;

    private PurchaseInfo()
    {
        this.timeOfPurchase = LocalDateTime.now();
    }
    static public PurchaseInfo getPurchaseInfo()
    {
        PurchaseInfo purchaseInfo = new PurchaseInfo();
        purchaseInfo.purchaseID = _currentID;
        _currentID++;
        return purchaseInfo;
    }
    public LocalDateTime getTimeOfPurchase()
    {
        return timeOfPurchase;
    }
    public int getPurchaseID()
    {
        return purchaseID;
    }
    public double getPurchaseAmount() {
        return purchaseAmount;
    }
    public void setPurchaseAmount(double purchaseAmount) {
        this.purchaseAmount = purchaseAmount;
    }
}
```

- The primary goal of the Singleton pattern is to restrict the instantiation of a class for a specific purpose (i.e: concealing variables and only changing their value whenever a certain event occurs, such as the instantiation of said Singleton class).
- PurchaseInfo is a class that provides extra purchase information that is **unique** to each purchase.
- These include a purchase ID, date of purchase, and the amount of said purchase.
- The constructor is private and can only be accessed through the getPurchaseInfo() method.
- The only time the purchase ID changes is when this method is called.

Decorator

```java
public class PremiumCustomerDecorator implements Observer {
    private Customer customer;

    public PremiumCustomerDecorator(Customer customer) {
        this.customer = customer;
    }

    @Override
    public void update(Customer customer) {
        int currentPoints = customer.getPoints();
        if (currentPoints > 10000) {
            System.out.println("Congratulations, " + customer.getName() + "!");
            System.out.println("You are now a premium customer!");
        }
    }

    public void redeemPoints() {
        if (customer.getPoints() >= 1000) {
            double rewardAmount = customer.getPoints() / 100.0;
            customer.addPoints((int) (-1000)); // deduct 1000 points from the customer's balan
            System.out.println("Congratulations! You redeemed " + rewardAmount + " dollars.");
        } else {
            System.out.println("Sorry, you need at least 1000 points to redeem rewards.");
        }
    }

    public void addPoints(int pointsEarned) {
        customer.addPoints(pointsEarned);
    }

    public String getName() {
        return customer.getName();
    }

    public int getPoints() {
        return customer.getPoints();
    }

    public boolean isPremiumCustomer() {
        return customer.getPoints() >= 10000;
    }
}
```

The constructor initializes the customer object that decorator will modify

Method is called when the customer's points are updated

Method allows the customer to redeem their points for rewards

Method allows the customer to add points to their account

Method returns the customer's name

Method returns the customer's current point balance

Indicates whether the customer is premium or not based on their current points

Observer- Customer

```java
// define the Subject interface
// in this case Customer is the subject

import java.util.ArrayList;
import java.util.List;

public class Customer {
    private String name;
    private int points;
    private List<Observer> observers = new ArrayList<>();

    public Customer(String name, int points) {
        this.name = name;
        this.points = points;
    }

    public void addPoints(int pointsEarned) {
        points += pointsEarned;
        notifyObservers();
    }

    public void redeemPoints() {
        if (points >= 1000) {
            double rewardAmount = points / 100.0;
            points = 0;
            notifyObservers();
            System.out.println("Congratulations! You redeemed " + rewardAmount + " dollars.");
        } else {
            System.out.println("Sorry, you need at least 1000 points to redeem rewards.");
        }
    }

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
```

```java
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(this);
        }
    }

    public String getName() {
        return name;
    }

    public int getPoints() {
        return points;
    }
}
//The Customer class maintains a list of observers and has methods to add points and redeem points.

//
```

- the primary function of the observer pattern is to provide a way for multiple objects to observe and react to a change in the code; without the need for the observed object to know anything about its observers making it more flexible to maintain.
- The Customer class **stores information** about a customer, provides methods to add and redeem points, and implements the Observer pattern to allow for notification of changes to the customer's point balance.
- The Constructor is Public and can be called when there's a need to create an instance of a customer with a name and a point balance, and when there is a need to add or redeem points for that customer.

Observer- Customer

Observer.java

```java
public interface Observer {
    void update(Customer customer);
}

//define the Observer interface. In our case, the RewardsProgram class will be the Observer.

```

The Observer interface is defined to have a single method "update" that takes a Customer object as a parameter.

Factory

```java
class pointsCustomer {
    3 usages
    private String name;
    4 usages
    private PointsReward pointsReward;

    4 usages
    public pointsCustomer(String name, int initialPoints) {
        this.name = name;
        this.pointsReward = new PointsReward();
        this.pointsReward.setCustomerPoints(initialPoints);
    }

    1 usage
    public String getName() {
        return name;
    }

    no usages
    public void setName(String name) {
        this.name = name;
    }

    1 usage
    public PointsReward getPointsReward() {
        return pointsReward;
    }
    no usages
    public void setPointsReward(PointsReward pointsReward){
        this.pointsReward = pointsReward;
    }
}
```

```java
// Reward interface
3 usages  1 implementation
interface Reward {
    4 usages  1 implementation
    void processReward(double purchaseAmount);
}
```

```java
class PointsReward implements Reward {
    4 usages
    private int customerPoints;

    3 usages
    PointsReward() { this.customerPoints = 0; }

    4 usages
    @Override
    public void processReward(double purchaseAmount) {
        if (purchaseAmount < 0) {
            System.out.println("Invalid purchase amount. Please enter a positive value.");
            return;
        }
        int POINTS_PER_DOLLAR = 10;
        int pointsEarned = (int) (purchaseAmount * POINTS_PER_DOLLAR);
        customerPoints += pointsEarned;
        System.out.println("You earned " + pointsEarned + " points for this purchase.");
    }

    6 usages
    public int getCustomerPoints() { return customerPoints; }

    2 usages
    public void setCustomerPoints(int customerPoints) {
        if (customerPoints < 0) {
            System.out.println("Invalid points value. Please enter a positive value.");
            return;
        }
        this.customerPoints = customerPoints;
    }
}
```

The Factory Design pattern provides an interface for creating objects in a superclass, allowing subclasses to determine the type of object that will be created. Its extremely versatile.

In this case the interface has processReward is Overriding processReward to handle rewards using points, this can be modified for various other types of Rewards such as cashback or special promotions.

Strategy

- For **our project**, in the main the customer is able to make a decision to **purchase**, which is the **only** way to **acquire reward points**. My design **helps** take different payment methods **ensuring that all customers can buy** from the shop that will **increase revenue and can get a chance to earn points**
- It **separates the behavior from the main class** and **encapsulate it into separate classes** which **allows for greater flexibility** and **maintainability** since different payment options require different info (diff inputs&if/else statements) from a customer
- It allows for the **open-closed principle** to be applied: **more** payment types (**classes**) can be easily added **without changing the existing** paymentType **interface**
- It involves **creating a family of algorithms** that perform **a specific task**
- Promotes code **reusability**, methods are overwritten if necessary for the pay option
- Helps to **eliminate duplicate** code, some payments ask for same info and now will avoid
- Strategies are completely **independent** & **unaware** of each other

Strategy

Helps add new payment types to the system without changing interface defines a common set of methods that each payment type implementation must implement, such as pay, tax, type, cashback methods

```java
public interface PaymentTypeStrategy {

    //This will handle each types of
    no usages  3 implementations
    public void pay(double amount);
    //This will just provide us with
    1 usage  3 implementations
    public String payType();

    //This will calculate will let us
    no usages  3 implementations
    public boolean tax();
    //This will let us know if a cust
    no usages  3 implementations
    public double cashbackAmount();
```

Example of Credit Card Payment option

All payments all have unique features such as this one
Pay method will have each ask the customer/user for different inputs to use the different payment option

```java
3 usages
public PayCreditStrategy(String cardNumber, String cardExpiration, String cvv) {
    this.cardNumber = cardNumber;
    this.cardExpiration = cardExpiration;
    this.cvv = cvv;

}
no usages
@Override
public void pay(double amount) {
    // Code to handle the credit card payment
    Scanner scanner = new Scanner(System.in);
    System.out.println("Please enter your credit card number: ");
    String c = scanner.nextLine();
    System.out.println("Please enter the expiration date (MM/YY): ");
    String e= scanner.nextLine();
    System.out.println("Please enter the CVV: ");
    String v = scanner.nextLine();
    PayCreditStrategy card= new PayCreditStrategy(c,e,v);

    //Checks if credit card is valid
    if(valid(card)) {

        System.out.println("Paying " + amount + " using credit card " + card.cardNumber);
    }
    else
        System.out.println("Credit Card is invalid");
}

1 usage
@Override
```

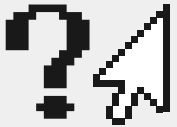
```java
public class PayCreditStrategy implements PaymentTypeStrategy {
```

```java
public class PayMobileStrategy implements PaymentTypeStrategy {
```

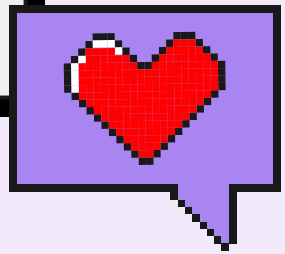
```java
public class PayCashStrategy implements PaymentTypeStrategy{
```

04

Unit test



Unit test with our Implemented Design



Singleton

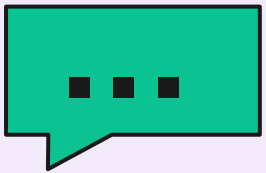
```
import java.time.LocalDateTime;

import static org.junit.jupiter.api.Assertions.*;

class PurchaseInfoTest {

    @org.junit.jupiter.api.Test
    void getPurchaseInfo()
    {
        PurchaseInfo info = PurchaseInfo.getPurchaseInfo();
        assertTrue(info.getPurchaseID() == 1); //test 1
        info = PurchaseInfo.getPurchaseInfo();
        assertTrue(info.getPurchaseID() == 2); //test 1
        assertTrue(info.getTimeOfPurchase().getDayOfYear() ==
LocalDateTime.now().getDayOfYear()); //test 2
    }
}
```

- The first test tests the IDs of purchases. The ID should always begin at 1 and is incremented each time new purchase info is acquired.
- The second test tests if the date information is accurate by retrieving the day of the LocalDateTime object created when the purchase info was created and comparing it to the day retrieved from calling LocalDateTime.now().



Decorator

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class PremiumCustomerDecoratorTest {

    @Test
    public void testIsPremiumCustomer() {
        Customer customer = new Customer("Maria", 0);

        PremiumCustomerDecorator premiumCustomerDecorator = new PremiumCustomerDecorator(customer);

        premiumCustomerDecorator.addPoints(11000);

        premiumCustomerDecorator.update(customer);

        assertTrue(premiumCustomerDecorator.getPoints() == 1000);
    }

    // Test to check if a customer with less than 10000 points does not become a premium customer
    @Test
    public void testIsNotPremiumCustomer() {
        // Create a new customer with 0 points
        Customer customer = new Customer("Henry", 0);

        // Wrap the customer object in a premium customer decorator object
        PremiumCustomerDecorator premiumCustomerDecorator = new PremiumCustomerDecorator(customer);

        // Add 5000 points to the customer's account
        premiumCustomerDecorator.addPoints(5000);

        // Update the premium customer decorator with the customer object
        premiumCustomerDecorator.update(customer);

        // Check if Henry is not a premium customer
        assertFalse(premiumCustomerDecorator.getPoints() == 1000);
    }
}
```

Test to check if a customer with more than and a customer with less than 10000 points becomes a premium customer

- Create a new customer named Maria with 0 points
- Wrap the customer object in a premium customer decorator object
- Add 11000 points to the customer's account
- Update the premium customer decorator with the customer object
- Check if Maria is now a premium customer

Observer Unit Test

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class CustomerTest1 {
    @Test
    public void testAddPoints() {
        Customer customer = new Customer("Alice", 0);
        customer.addPoints(100);
        assertEquals(100, customer.getPoints());
    }

    @Test
    public void testRedeemPoints() {
        Customer customer = new Customer("Bob", 1500);
        customer.redeemPoints();
        assertEquals(0, customer.getPoints());
    }
}
```

- In testAddPoints(), a new Customer object is created with an initial point balance of 0, and 100 points are added using the addPoints() method. The test then checks if the customer's point balance is updated to 100 using the assertEquals method.
- In testRedeemPoints(), a new Customer object is created with an initial point balance of 1500, and the redeemPoints() method is called to redeem all the points for a reward.
- The test then checks if the customer's point balance is updated to 0 using the assertEquals method.

Factory

```java
@Test
public void testProcessRewardEarnPoints() { // checks if expected points is 10*purchase amount
    double purchaseAmount = 10.0;
    int expectedPoints = 100;

    pointsReward.processReward(purchaseAmount);

    assertEquals(expectedPoints, pointsReward.getCustomerPoints(), message: "Points should be 100 after processing the reward.");
}

@Test
public void testProcessRewardEarnZeroPoints() {
    double purchaseAmount = 0.0;
    int expectedPoints = 0;

    pointsReward.processReward(purchaseAmount);

    assertEquals(expectedPoints, pointsReward.getCustomerPoints(), message: "Points should be 0 after processing the reward.");
}
```

The tests are rather straightforward the first test is to ensure that the points are acquired according to the predefined rate.

Second test is checking if rewards are set correctly after being used in this case setting it to 0.

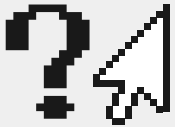
Strategy

- Verify that each strategy class correctly implements the payment logic for its respective payment type
- Help catch any bugs or errors in the strategy pattern implementation before it is deployed to production, which can save time and money in the long run
- Provides reliability in the purchase system's payment functionality, which can help improve the customer experience and ensure that reward points can be acquired

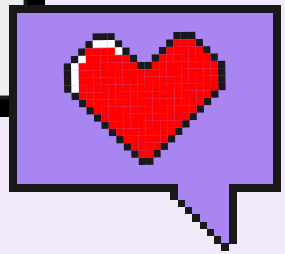
```
public class PayStrategyTest {
    no usages
    @Test
    void mobilePayCheck() {
        var test1 = new PayMobileStrategy( phoneNumber: "3474447777");
        assertEquals("Mobile Tap-to-Pay", test1.payType());
    }
    no usages
    @Test
    void cashPayCheck() {
        var test2 = new PayCashStrategy();
        assertEquals("Cash", test1.payType());
    }
    no usages
    @Test
    void creditPayInvalidCheck() {
        var test3 = new PayCreditStrategy( cardNumber: "1234567890",  cardExpiration: "1/23",  cvv: "12");
        assertFalse(test3.valid(test3));
    }
    no usages
    @Test
    void cashbackAmountCheck() {
        var test4 = new PayCreditStrategy( cardNumber: "1234567890123456",  cardExpiration: "12/23",  cvv: "123");
        assertEquals(0.03, test5.cashbackAmount());
    }
```

04

Component Test

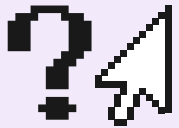


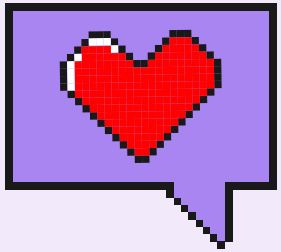
Component Test for our Design Patterns



Singleton

A customer purchases 10 items, the 10^{th} item should have a purchase ID of 10.

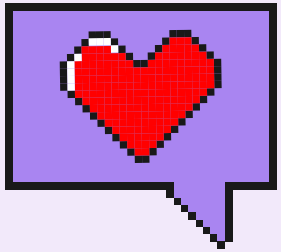




Decorator

Test to check if a premium customer gets extra points

- Create a new customer with 5000 points
- Check if premium customer
- Check if customer would get extra points for being a premium customer



Observer

A Customer purchases 11 items, so the 11th item so the customer should have 1,500 Points

Factory

Create a new instance and test the interaction between Customer and PointsReward. Using a method called testInteraction you can check if the customer makes a purchase and see if the processReward() updates the customers points correctly.

```
public void testRewardFactoryIntegration() {
    // Define a purchase amount and the expected points after processing the reward
    double purchaseAmount = 50.0;
    int expectedPoints = 500;

    // Process the reward using the Reward object
    reward.processReward(purchaseAmount);

    // Cast the Reward object to a PointsReward object and assert that the customer points are updated as expected
    PointsReward pointsReward = (PointsReward) reward;
    assertEquals(expectedPoints, pointsReward.getCustomerPoints(), message: "Customer points should be 500 after processing reward");
}
```

Strategy

A Customer uses a credit card to pay, must have the correct attributes of a credit card:

```
void creditPayInvalidCheck() {
    var test3 = new PayCreditStrategy( cardNumber: "1234567890",  cardExpiration: "1/23",  cvv: "12");
    assertFalse(test3.valid(test3));
}
```

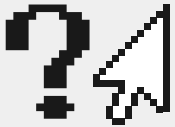
A Customer uses tap to pay, must have a phone number attribute:

```
void mobilePayCheck() {
    var test1 = new PayMobileStrategy( phoneNumber: "3474447777");
    assertEquals("Mobile Tap-to-Pay", test1.payType());
}
```

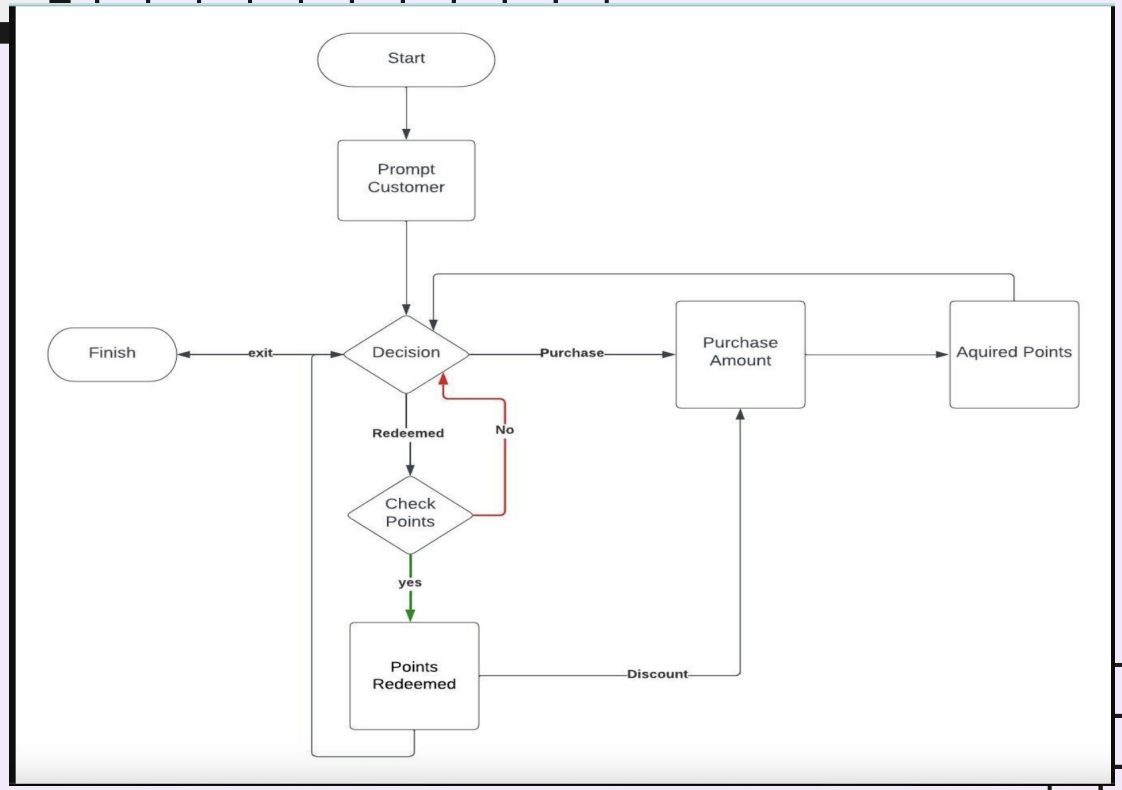
05

Diagrams

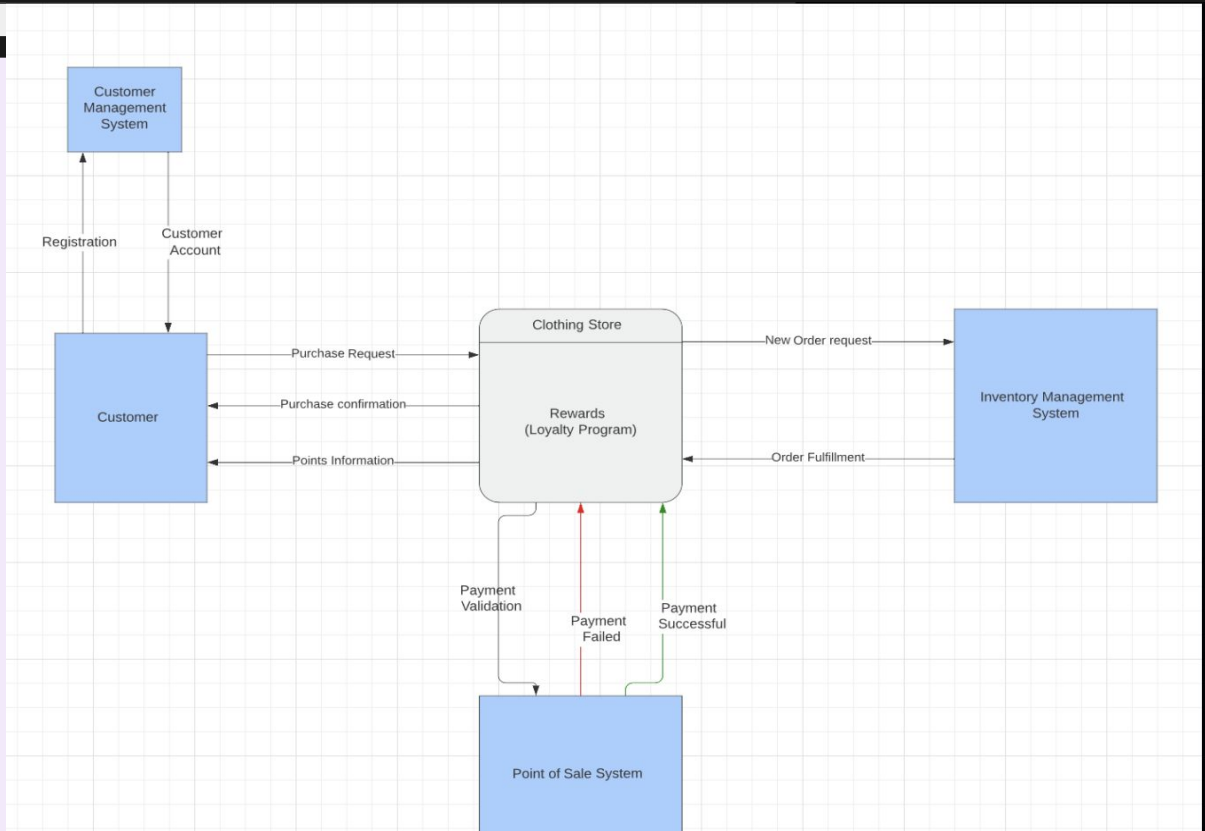
brief sentence about the Diagrams



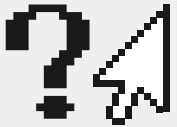
Process Flow Diagram



Context Diagram



Requirement Doc



Required Document

Required Document:
https://docs.google.com/document/d/1BGbyxPyNGu_TVgSDzxWvhVLX1KN_0CeyYLC4410rX7g/edit?usp=sharing