Name: **Ximan Liu**

Let us consider 2 SALES RECEIPT classes: RECEIPT1 and RECEIPT2. RECEIPT1 has a method that displays the message “Receipt Body 1”. RECEIPT2 has a method that displays the message “Receipt Body 2”. A SALES RECEIPT object has exactly 1 header and 1 footer. There are 2 possible HEADER classes (HEADER1 and HEADER2) and 2 possible FOOTER classes (FOOTER1 and FOOTER2).

* HEADER1 class has a method that displays the message “Heade1r 1”.
* HEADER2 class has a method that displays the message “Header 2”.
* FOOTER1 class has a method that displays the message “Footer 1”.
* FOOTER2 class has a method that displays the message “Footer 2”.

HEADER1 should be used with RECEIPT1 and FOOTER1. HEADER2 should be used with RECEIPT2 and FOOTER2.

We would like to use the abstract factory pattern to create sales receipts.

1. Give the UML class diagram (5 points).
2. Provide the implementation code of the UML class diagram given in question 1) in the programming language of your choice. The object created by the client code should display the following message: ”Header 1 Receipt Body 1 Footer 1” (20 points).

A close up of text on a white background

Description automatically generated

// AbstractFactory.java

public abstract class AbstractFactory {

public abstract AbstractHeader CreateHeader();

public abstract AbstractFooter CreateFooter();

public abstract AbstractReceipt CreateReceipt();

}

// AbstractFooter.java

public abstract class AbstractFooter {

public abstract void DisplayName(AbstractFooter a);

}

// AbstractReceipt.java

public abstract class AbstractReceipt {

public abstract void DisplayName(AbstractReceipt a);

}

// AbstractHeader.java

public abstract class AbstractHeader {

public abstract void DisplayName(AbstractHeader a);

}

// ConcreteReceipt1.java

public class ConcreteReceipt1 extends AbstractFactory {

public AbstractHeader CreateHeader() {

return new HeaderReceipt1();

}

public AbstractFooter CreateFooter() {

return new FooterReceipt1();

}

public AbstractReceipt CreateReceipt() {

return new ReceiptReceipt1();

}

}

// ConcreteReceipt2.java

public class ConcreteReceipt2 extends AbstractFactory {

public AbstractHeader CreateHeader() {

return new HeaderReceipt2();

}

public AbstractFooter CreateFooter() {

return new FooterReceipt2();

}

public AbstractReceipt CreateReceipt() {

return new ReceiptReceipt2();

}

}

// HeaderReceipt1.java

public class HeaderReceipt1 extends AbstractHeader {

@Override

public void DisplayName(AbstractHeader a) {

System.out.println("Header 1 ");

}

}

// HeaderReceipt2.java

public class HeaderReceipt2 extends AbstractHeader{

@Override

public void DisplayName(AbstractHeader a) {

System.out.println("Header 2 ");

}

}

// FooterReceipt1.java

public class FooterReceipt1 extends AbstractFooter {

@Override

public void DisplayName(AbstractFooter a) {

System.out.println("Footer 1");

}

}

// FooterReceipt2.java

public class FooterReceipt2 extends AbstractFooter{

@Override

public void DisplayName(AbstractFooter a) {

System.out.println("Footer 2 ");

}

}

// ReceiptReceipt1

public class ReceiptReceipt1 extends AbstractReceipt {

@Override

public void DisplayName(AbstractReceipt a) {

System.out.println("Receipt Body 1");

}

}

// ReceiptReceipt2.java

public class ReceiptReceipt2 extends AbstractReceipt {

@Override

public void DisplayName(AbstractReceipt a) {

System.out.println("Receipt Body 2");

}

}

// Main.java

public class Main {

private AbstractHeader AbstractHeader;

private AbstractFooter AbstractFooter;

private AbstractReceipt AbstractReceipt;

public Main(AbstractFactory factory) {

AbstractHeader = factory.CreateHeader();

AbstractFooter = factory.CreateFooter();

AbstractReceipt = factory.CreateReceipt();

}

public void run() {

AbstractHeader.DisplayName(AbstractHeader);

AbstractReceipt.DisplayName(AbstractReceipt);

AbstractFooter.DisplayName(AbstractFooter);

}

public static void main(String[] args) {

AbstractFactory R1 = new ConcreteReceipt1();

Main m = new Main(R1);

m.run();

}

}

A system uses three (3) temperature sensors produced by the same vendor V1 to monitor the condition of a hardware device at a specific time:

* + The first sensor is of type TS1.
  + The second sensor is of type TS2.
  + The third sensor is of type TS3.

TS1, TS2, and TS3 accept as input a date and time (given in Eastern Standard Time or EST) and return a temperature in Fahrenheit. Time is in the 24-hour time keeping system (e.g., 23:59:59). V1 supplies a class to interface with its sensors:

Class TS1 {

Public double getTemp(date d, time t) { // returns Temp in Fahrenheit

// t is in EST time

// t is in the 24-hour time keeping system (e.g., 23:59:59)

// Code Goes here } }

Class TS2 {

Public double getTemp(date d, time t) {// returns Temp in Fahrenheit

// t is in EST time

// t is in the 24-hour time keeping system (e.g., 23:59:59)

// Code Goes here } }

Class TS3 {

Public double getTemp(date d, time t) {// returns Temp in Fahrenheit

// t is in EST time

// t is in the 24-hour time keeping system (e.g., 23:59:59)

// Code Goes here } }

We would like to add a third sensor of type TS4 provided by vendor V2. TS4 accepts as inputs a year, month, day, and time (specified in Pacific Standard Time or PST) and returns a temperature in 1/10 of Celsius. Time is specified in the AM/PM format (12-hour clock). Below is the interface for TS4:

Class TS4 {

Public double getTS4Temp(year y, month m, day d, time t, string when) {

// returns Temp in 1/10 Celsius

// Fahrenheit = (Celsius -32) \* 5/9

// t is in PST time

// t is specified in the AM/PM format (12-hour clock)

// PST = EST – 3

// when=”AM” or when=”PM”

// Code Goes here } }

1. Give the UML class diagram for the application using the object adapter pattern. (5 points)
2. Provide the implementation code of the UML class diagram given in question 1) in the programming language of your choice. The client application should read a date and time from the user and compute the average temperature of the four sensors at that date and time. (20 points)

A close up of text on a white background

Description automatically generated

// TS.java

public abstract class TS {

public TS() {};

public double getTemp(Date d, Time t){

return 0.0;

}

}

// TS1.java

public class TS1 extends TS {

public double getTemp(Date d, Time t){

return 0.0;

}

}

// TS2.java

public class TS2 extends TS {

public double getTemp(Date d, Time t){

return 0.0;

}

}

// TS3.java

public class TS3 extends TS {

public double getTemp(Date d, Time t){

return 0.0;

}

}

// TSAdapter.java

public class TSAdapter extends TS {

private double temp = 100;

private TS4 TS4;

public TSAdapter() {

TS4 = new TS4();

}

public double getTemp(Date d, Time t){

TS4.getTS4Temp();

}

}

// TS4.java

public class TS4 {

public TS4() {}

public double getTS4Temp(Year y, Month m, Date d, Time t, String when) {

return 0.0;

}

}

Let us consider a PIZZA class which has a method that displays the message “Pizza”. A PIZZA object may be decorated by 3 kinds of toppings.

* + OLIVE class has a method that displays the message “With Olive”.
  + PEPPER class has a method that displays the message “With Pepper”.
  + MASHROOM class has a method that displays the message “With Mashroom”.

We would like to use the decorator pattern to create pizzas with toppings.

1. Give the UML class diagram. (5 points)
2. Provide the implementation code of the UML class diagram given in question 1) in the programming language of your choice. The object created by the client code should display the following message: ”Pizza With Pepper With Olive With Mashroom”. (20 points)

A close up of text on a white background

Description automatically generated

// Decorator.java

public abstract class Decorator extends pizza {

public abstract String run();

}

// pizza.java

public abstract class pizza {

public abstract String run();

}

// base.java

public class base extends pizza {

public String run() {

return "Pizza ";

}

}

// mashroom.java

public class mashroom extends Decorator {

pizza pizza;

public mashroom(pizza pizza) {

this.pizza = pizza;

}

public String run() {

return pizza.run() + "With Mashroom";

}

}

// olive.java

public class olive extends Decorator {

pizza pizza;

public olive(pizza pizza) {

this.pizza = pizza;

}

public String run() {

return pizza.run() + "With Olive ";

}

}

// pepper.java

public class pepper extends Decorator

{

pizza pizza;

public pepper(pizza pizza) {

this.pizza = pizza;

}

public String run() {

return pizza.run() + "With Pepper ";

}

}

// Main.java

public class Main {

public static void main(String args[]) {

pizza pizza = new base();

pizza = new pepper(pizza);

pizza = new olive(pizza);

pizza = new mashroom(pizza);

System.out.println(pizza.run());

}

}

Let us consider the builder pattern where Product has four possible parts Part1, Part2, Part3, and Part4 (all integer). Assume we have three concrete builders:

ConcreteBuilder1, ConcreteBuilder2, and ConcreteBuilder3. ConcreteBuilder1 assign 11, 12, 13, and 14 to

Part1, Part2, Part 3, and Part4 respectively. ConcreteBuilder2 assign 21, 22, 23, and 24 to Part1, Part2, Part 3, and Part4 respectively. ConcreteBuilder3 assign 31, 32, 33, and 34 to Part1, Part2, Part3, and Part4 respectively.

1. Give the UML diagram.
2. Implement the UML class diagram given in 1) in the programming language of your choice.

A close up of a map

Description automatically generated

// ProductBuilder.java

public abstract class ProductBuilder {

public abstract void Part1();

public abstract void Part2();

public abstract void Part3();

public abstract void Part4();

public abstract Product Number();

}

// Product.java

public class Product {

public int Part1;

public int Part2;

public int Part3;

public int Part4;

public void run() {

System.out.println("Part1: " + Part1);

System.out.println("Part2: " + Part2);

System.out.println("Part3: " + Part3);

System.out.println("Part4: " + Part4);

System.out.println(" ");

}

}

// ProductEngineer.java

public class ProductEngineer {

public void makeProduct(ProductBuilder productBuilder) {

productBuilder.Part1();

productBuilder.Part2();

productBuilder.Part3();

productBuilder.Part4();

}

}

// ConcreteBuilder1.java

public class ConcreteBuilder1 extends ProductBuilder {

private Product product = new Product();

public void Part1() {

product.Part1 = 11;

}

public void Part2() {

product.Part2 = 12;

}

public void Part3() {

product.Part3 = 13;

}

public void Part4() {

product.Part4 = 14;

}

public Product Number() {

return product;

}

}

// ConcreteBuilder2.java

public class ConcreteBuilder2 extends ProductBuilder {

private Product product = new Product();

public void Part1() {

product.Part1 = 21;

}

public void Part2() {

product.Part2 = 22;

}

public void Part3() {

product.Part3 = 23;

}

public void Part4() {

product.Part4 = 24;

}

public Product Number() {

return product;

}

}

// ConcreteBuilder3.java

public class ConcreteBuilder3 extends ProductBuilder {

private Product product = new Product();

public void Part1() {

product.Part1 = 31;

}

public void Part2() {

product.Part2 = 32;

}

public void Part3() {

product.Part3 = 33;

}

public void Part4() {

product.Part4 = 34;

}

public Product Number() {

return product;

}

}

// Main.java

public class Main {

public static void main (String[] args) {

ProductEngineer factory = new ProductEngineer();

ProductBuilder ConcreteBuilder1 = new ConcreteBuilder1();

ProductBuilder ConcreteBuilder2 = new ConcreteBuilder2();

ProductBuilder ConcreteBuilder3 = new ConcreteBuilder3();

factory.makeProduct(ConcreteBuilder1);

factory.makeProduct(ConcreteBuilder2);

factory.makeProduct(ConcreteBuilder3);

Product doneConcreteBuilder1 = ConcreteBuilder1.Number();

Product doneConcreteBuilder2 = ConcreteBuilder2.Number();

Product doneConcreteBuilder3 = ConcreteBuilder3.Number();

doneConcreteBuilder1.run();

doneConcreteBuilder2.run();

doneConcreteBuilder3.run();

}

}