**Suryadatta College of Management Information Research & Technology (SCMIRT)**

**SOFTWARE ARCHITECTURE AND DESIGN PATTERNS**

**JOURNAL**

**Name: PRATEEK GANGURDE**

**CLASS: SY MSc (COMP. SCI.)**

1. **Write a JAVA Program to implement built-in support (java.util.Observable) Weather station with members temperature, humidity, pressure and methods mesurmentsChanged(), setMesurment(), getTemperature(), getHumidity(), getPressure()**

import java.util.Observable;

import java.util.Observer;

class CurrentConditionsDisplay implements Observer, DisplayElement {

Observable observable;

private float temperature;

private float humidity;

public CurrentConditionsDisplay(Observable observable) {

this.observable = observable;

observable.addObserver(this);

}

public void update(Observable obs, Object arg) {

if (obs instanceof WeatherData) {

WeatherData weatherData = (WeatherData)obs;

this.temperature = weatherData.getTemperature();

this.humidity = weatherData.getHumidity();

display();

}

}

public void display() {

System.out.println("Current conditions: " + temperature

+ "F degrees and " + humidity + "% humidity");

}

}

interface DisplayElement {

public void display();

}

class ForecastDisplay implements Observer, DisplayElement {

private float currentPressure = 29.92f;

private float lastPressure;

public ForecastDisplay(Observable observable) {

observable.addObserver(this);

}

public void update(Observable observable, Object arg) {

if (observable instanceof WeatherData) {

WeatherData weatherData = (WeatherData)observable;

lastPressure = currentPressure;

currentPressure = weatherData.getPressure();

display();

}

}

public void display() {

System.out.print("Forecast: ");

if (currentPressure > lastPressure) {

System.out.println("Improving weather on the way!");

} else if (currentPressure == lastPressure) {

System.out.println("More of the same");

} else if (currentPressure < lastPressure) {

System.out.println("Watch out for cooler, rainy weather");

}

}

}

class HeatIndexDisplay implements Observer, DisplayElement {

float heatIndex = 0.0f;

public HeatIndexDisplay(Observable observable) {

observable.addObserver(this);

}

public void update(Observable observable, Object arg) {

if (observable instanceof WeatherData) {

WeatherData weatherData = (WeatherData)observable;

float t = weatherData.getTemperature();

float rh = weatherData.getHumidity();

heatIndex = (float)

(

(16.923 + (0.185212 \* t)) +

(5.37941 \* rh) -

(0.100254 \* t \* rh) +

(0.00941695 \* (t \* t)) +

(0.00728898 \* (rh \* rh)) +

(0.000345372 \* (t \* t \* rh)) -

(0.000814971 \* (t \* rh \* rh)) +

(0.0000102102 \* (t \* t \* rh \* rh)) -

(0.000038646 \* (t \* t \* t)) +

(0.0000291583 \* (rh \* rh \* rh)) +

(0.00000142721 \* (t \* t \* t \* rh)) +

(0.000000197483 \* (t \* rh \* rh \* rh)) -

(0.0000000218429 \* (t \* t \* t \* rh \* rh)) +

(0.000000000843296 \* (t \* t \* rh \* rh \* rh)) -

(0.0000000000481975 \* (t \* t \* t \* rh \* rh \* rh)));

display();

}

}

public void display() {

System.out.println("Heat index is " + heatIndex);

}

}

class StatisticsDisplay implements Observer, DisplayElement {

private float maxTemp = 0.0f;

private float minTemp = 200;

private float tempSum= 0.0f;

private int numReadings;

public StatisticsDisplay(Observable observable) {

observable.addObserver(this);

}

public void update(Observable observable, Object arg) {

if (observable instanceof WeatherData) {

WeatherData weatherData = (WeatherData)observable;

float temp = weatherData.getTemperature();

tempSum += temp;

numReadings++;

if (temp > maxTemp) {

maxTemp = temp;

}

if (temp < minTemp) {

minTemp = temp;

}

display();

}

}

public void display() {

System.out.println("Avg/Max/Min temperature = " + (tempSum / numReadings)+ "/"

+ maxTemp + "/" + minTemp);

}

}

class WeatherData extends Observable {

private float temperature;

private float humidity;

private float pressure;

public WeatherData() { }

public void measurementsChanged() {

setChanged();

notifyObservers();

}

public void setMeasurements(float temperature, float humidity, float pressure) {

this.temperature = temperature;

this.humidity = humidity;

this.pressure = pressure;

measurementsChanged();

}

public float getTemperature() {

return temperature;

}

public float getHumidity() {

return humidity;

}

public float getPressure() {

return pressure;

}

}

public class Main {

public static void main(String[] args) {

WeatherData weatherData = new WeatherData();

CurrentConditionsDisplay currentConditions = new

CurrentConditionsDisplay(weatherData);

StatisticsDisplay statisticsDisplay = new

StatisticsDisplay(weatherData);

ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

weatherData.setMeasurements(80, 65, 30.4f);

weatherData.setMeasurements(82, 70, 29.2f);

weatherData.setMeasurements(78, 90, 29.2f);

}

}

1. **Write a Java Program to implement I/O Decorator for converting uppercase letters to lower case letters.**

import java.io.\*;

import java.util.\*;

class LowerCaseInputStream extends FilterInputStream

{

public LowerCaseInputStream(InputStream in) {

super(in);

}

public int read() throws IOException {

int c = super.read();

return (c == -1 ? c : Character.toLowerCase((char)c));

}

public int read(byte[] b, int offset, int len) throws IOException {

int result = super.read(b, offset, len);

for (int i = offset; i < offset+result; i++) {

b[i] = (byte)Character.toLowerCase((char)b[i]);

}

return result;

}

}

public class Main {

public static void main(String[] args) throws IOException {

int c;

try {

InputStream in =

new LowerCaseInputStream(

new BufferedInputStream(

new FileInputStream("test.txt")));

while((c = in.read()) >= 0) {

System.out.print((char)c);

}

in.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

1. **Write a Java Program to implement Factory method for Pizza Store with createPizza(), orederPizza(), prepare(), Bake(), cut(), box(). Use this to create variety of pizza’s like NyStyleCheesePizza, ChicagoStyleCheesePizza etc.**

import java.util.ArrayList;

class ChicagoPizzaStore extends PizzaStore

{

Pizza createPizza(String item)

{

if (item.equals("cheese"))

{

return new ChicagoStyleCheesePizza();

}

else return null;

}

}

class ChicagoStyleCheesePizza extends Pizza

{

public ChicagoStyleCheesePizza()

{

name = "Chicago Style Deep Dish Cheese Pizza";

dough = "Extra Thick Crust Dough";

sauce = "Plum Tomato Sauce";

toppings.add("Shredded Mozzarella Cheese");

}

void cut()

{

System.out.println("Cutting the pizza into square slices");

}

}

class DependentPizzaStore

{ public Pizza createPizza(String style, String type)

{ Pizza pizza = null;

if (style.equals("NY"))

{

if (type.equals("cheese"))

{

pizza = new NYStyleCheesePizza();

}

}

if (style.equals("Chicago"))

{

if (type.equals("cheese"))

{

pizza = new ChicagoStyleCheesePizza();

}

}

else

{

System.out.println("Error: invalid type of pizza"); return null;

}

pizza.prepare();

pizza.bake();

pizza.cut();

pizza.box();

return pizza;

}

}

class NYPizzaStore extends PizzaStore

{

Pizza createPizza(String item)

{

if (item.equals("cheese"))

{

return new NYStyleCheesePizza();

}

else return null;

}

}

class NYStyleCheesePizza extends Pizza

{

public NYStyleCheesePizza()

{

name = "NY Style Sauce and Cheese Pizza";

dough = "Thin Crust Dough";

sauce = "Marinara Sauce";

toppings.add("Grated Reggiano Cheese");

}

}

abstract class Pizza

{

String name;

String dough;

String sauce;

ArrayList toppings = new ArrayList();

void prepare()

{

System.out.println("Preparing " + name);

System.out.println("Tossing dough...");

System.out.println("Adding sauce...");

System.out.println("Adding toppings: ");

for (int i = 0; i < toppings.size(); i++)

{

System.out.println(" " + toppings.get(i));

}

}

void bake()

{

System.out.println("Bake for 25 minutes at 350");

}

void cut()

{

System.out.println("Cutting the pizza into diagonal slices");

}

void box()

{

System.out.println("Place pizza in official PizzaStore box");

}

public String getName()

{

return name;

}

public String toString()

{

StringBuffer display = new StringBuffer();

display.append("---- " + name + " ----\n");

display.append(dough + "\n");

display.append(sauce + "\n");

for (int i = 0; i < toppings.size(); i++)

{

display.append((String )toppings.get(i) + "\n");

}

return display.toString();

}

}

abstract class PizzaStore

{

abstract Pizza createPizza(String item);

public Pizza orderPizza(String type)

{

Pizza pizza = createPizza(type);

System.out.println("--- Making a " + pizza.getName() + " ---");

pizza.prepare();

pizza.bake();

pizza.cut();

pizza.box();

return pizza;

}

}

public class Main

{

public static void main(String[] args)

{

PizzaStore nyStore = new NYPizzaStore();

PizzaStore chicagoStore = new ChicagoPizzaStore();

Pizza pizza = nyStore.orderPizza("cheese");

System.out.println("Poonam ordered a " + pizza.getName() + "\n");

pizza = chicagoStore.orderPizza("cheese");

System.out.println("Kadambari ordered a " + pizza.getName() + "\n");

pizza = nyStore.orderPizza("cheese");

}

}

/\* program for all pizza types\*/

import java.util.ArrayList;

class ChicagoPizzaStore extends PizzaStore

{

Pizza createPizza(String item)

{

if (item.equals("cheese"))

{

return new ChicagoStyleCheesePizza();

}

else if (item.equals("veggie"))

{

return new ChicagoStyleVeggiePizza();

}

else if (item.equals("clam"))

{

return new ChicagoStyleClamPizza();

}

else if (item.equals("pepperoni"))

{

return new ChicagoStylePepperoniPizza();

m.out.println("Cutting the pizza into square slices");

}

}

class ChicagoStyleClamPizza extends Pizza

{

public ChicagoStyleClamPizza()

{

name = "Chicago Style Clam Pizza";

dough = "Extra Thick Crust Dough";

sauce = "Plum Tomato Sauce";

toppings.add("Shredded Mozzarella Cheese");

toppings.add("Frozen Clams from Chesapeake Bay");

}

void cut()

{

System.out.println("Cutting the pizza into square slices");

}

}

class ChicagoStylePepperoniPizza extends Pizza

{

public ChicagoStylePepperoniPizza()

{

name = "Chicago Style Pepperoni Pizza";

dough = "Extra Thick Crust Dough";

sauce = "Plum Tomato Sauce";

toppings.add("Shredded Mozzarella Cheese");

toppings.add("Black Olives");

toppings.add("Spinach");

toppings.add("Eggplant");

toppings.add("Sliced Pepperoni");

}

void cut()

{

System.out.println("Cutting the pizza into square slices");

}

}

class ChicagoStyleVeggiePizza extends Pizza

{

public ChicagoStyleVeggiePizza()

{

name = "Chicago Deep Dish Veggie Pizza";

dough = "Extra Thick Crust Dough";

sauce = "Plum Tomato Sauce";

toppings.add("Shredded Mozzarella Cheese");

toppings.add("Black Olives");

toppings.add("Spinach");

toppings.add("Eggplant");

}

void cut()

{

System.out.println("Cutting the pizza into square slices");

}

}

class DependentPizzaStore

{ public Pizza createPizza(String style, String type)

{ Pizza pizza = null;

if (style.equals("NY"))

{

if (type.equals("cheese"))

{

pizza = new NYStyleCheesePizza();

}

else if (type.equals("veggie"))

{

pizza = new NYStyleVeggiePizza();

}

else if (type.equals("clam"))

{

pizza = new NYStyleClamPizza();

}

else if (type.equals("pepperoni"))

{

pizza = new NYStylePepperoniPizza();

}

}

else if (style.equals("Chicago"))

{

if (type.equals("cheese"))

{

pizza = new ChicagoStyleCheesePizza();

}

else if (type.equals("veggie"))

{

pizza = new ChicagoStyleVeggiePizza();

}

else if (type.equals("clam"))

{

pizza = new ChicagoStyleClamPizza();

}

else if (type.equals("pepperoni"))

{

pizza = new ChicagoStylePepperoniPizza();

}

}

else

{

System.out.println("Error: invalid type of pizza"); return null;

}

pizza.prepare();

pizza.bake();

pizza.cut();

pizza.box();

return pizza;

}

}

class NYPizzaStore extends PizzaStore

{

Pizza createPizza(String item)

{

if (item.equals("cheese"))

{

return new NYStyleCheesePizza();

}

else if (item.equals("veggie"))

{

return new NYStyleVeggiePizza();

}

else if (item.equals("clam"))

{

return new NYStyleClamPizza();

}

else if (item.equals("pepperoni"))

{

return new NYStylePepperoniPizza();

}

else return null;

}

}

class NYStyleCheesePizza extends Pizza

{

public NYStyleCheesePizza()

{

name = "NY Style Sauce and Cheese Pizza";

dough = "Thin Crust Dough";

sauce = "Marinara Sauce";

toppings.add("Grated Reggiano Cheese");

}

}

class NYStyleClamPizza extends Pizza

{

public NYStyleClamPizza()

{

name = "NY Style Clam Pizza";

dough = "Thin Crust Dough";

sauce = "Marinara Sauce";

toppings.add("Grated Reggiano Cheese");

toppings.add("Fresh Clams from Long Island Sound");

}

}

class NYStylePepperoniPizza extends Pizza

{

public NYStylePepperoniPizza()

{

name = "NY Style Pepperoni Pizza";

dough = "Thin Crust Dough";

sauce = "Marinara Sauce";

toppings.add("Grated Reggiano Cheese");

toppings.add("Sliced Pepperoni");

toppings.add("Garlic");

toppings.add("Onion");

toppings.add("Mushrooms");

toppings.add("Red Pepper");

}

}

class NYStyleVeggiePizza extends Pizza

{

public NYStyleVeggiePizza()

{

name = "NY Style Veggie Pizza";

dough = "Thin Crust Dough";

sauce = "Marinara Sauce";

toppings.add("Grated Reggiano Cheese");

toppings.add("Garlic");

toppings.add("Onion");

toppings.add("Mushrooms");

toppings.add("Red Pepper");

}

}

abstract class Pizza

{

String name;

String dough;

String sauce;

ArrayList toppings = new ArrayList();

void prepare()

{

System.out.println("Preparing " + name);

System.out.println("Tossing dough...");

System.out.println("Adding sauce...");

System.out.println("Adding toppings: ");

for (int i = 0; i < toppings.size(); i++)

{

System.out.println(" " + toppings.get(i));

}

}

void bake()

{

System.out.println("Bake for 25 minutes at 350");

}

void cut()

{

System.out.println("Cutting the pizza into diagonal slices");

}

void box()

{

System.out.println("Place pizza in official PizzaStore box");

}

public String getName()

{

return name;

}

public String toString()

{

StringBuffer display = new StringBuffer();

display.append("---- " + name + " ----\n");

display.append(dough + "\n");

display.append(sauce + "\n");

for (int i = 0; i < toppings.size(); i++)

{

display.append((String )toppings.get(i) + "\n");

}

return display.toString();

}

}

abstract class PizzaStore

{

abstract Pizza createPizza(String item);

public Pizza orderPizza(String type)

{

Pizza pizza = createPizza(type);

System.out.println("--- Making a " + pizza.getName() + " ---");

pizza.prepare();

pizza.bake();

pizza.cut();

pizza.box();

return pizza;

}

}

public class Main

{

public static void main(String[] args)

{

PizzaStore nyStore = new NYPizzaStore();

PizzaStore chicagoStore = new ChicagoPizzaStore();

Pizza pizza = nyStore.orderPizza("cheese");

System.out.println("Ethan ordered a " + pizza.getName() + "\n");

pizza = chicagoStore.orderPizza("cheese");

System.out.println("Joel ordered a " + pizza.getName() + "\n");

pizza = nyStore.orderPizza("clam");

System.out.println("Ethan ordered a " + pizza.getName() + "\n");

pizza = chicagoStore.orderPizza("clam");

System.out.println("Joel ordered a " + pizza.getName() + "\n");

pizza = nyStore.orderPizza("pepperoni");

System.out.println("Ethan ordered a " + pizza.getName() + "\n");

pizza = chicagoStore.orderPizza("pepperoni");

System.out.println("Joel ordered a " + pizza.getName() + "\n");

pizza = nyStore.orderPizza("veggie");

System.out.println("Ethan ordered a " + pizza.getName() + "\n");

pizza = chicagoStore.orderPizza("veggie");

System.out.println("Joel ordered a " +

pizza.getName() + "\n");

}

}

1. **Write a Java Program to implement Singleton pattern for multithreading**.

public class Main {

public static void main(String ar[]) {

Test1 t = new Test1();

Test1 t2 = new Test1();

Test1 t3 = new Test1();

Thread tt = new Thread(t);

Thread tt2 = new Thread(t2);

Thread tt3 = new Thread(t3);

Thread tt4 = new Thread(t);

Thread tt5 = new Thread(t);

tt.start();

tt2.start();

tt3.start();

tt4.start();

tt5.start();

}

}

final class Test1 implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + " : " +

Single.getInstance().hashCode());

}

}

}

class Single {

private final static Single sing = new Single();

private Single() {

}

public static Single getInstance() {

return sing;

}

}

1. **Write a Java Program to implement command pattern to test Remote Control.**

interface Command {

public void execute();

}

class Light {

public void on(){

System.out.println("Light is on");

}

public void off()

{

System.out.println("Light is off");

}

}

class LightOnCommand implements Command {

Light l1;

public LightOnCommand(Light a) {

this.l1 = a;

}

public void execute() {

l1.on();

}

}

class LightOffCommand implements Command {

Light l1;

public LightOffCommand(Light a) {

this.l1 = a;

}

public void execute() {

l1.off();

}

}

class SimpleRemoteControl {

Command slot;

public SimpleRemoteControl() {}

public void setCommand(Command command) {

slot = command;

}

public void buttonWasPressed() {

slot.execute();

}

}

public class Main {

public static void main(String[] args) {

SimpleRemoteControl r1 = new SimpleRemoteControl();

Light l1 = new Light();

LightOnCommand lo = new LightOnCommand(l1);

r1.setCommand(lo);

r1.buttonWasPressed();

LightOffCommand lO = new LightOffCommand(l1);

r1.setCommand(lO);

r1.buttonWasPressed();

}

}

1. **Write a Java Program to implement undo command to test Ceiling fan.**

interface Command {

public void execute();

}

class CeilingFan {

public void on(){

System.out.println("Ceiling Fan is on");

}

public void off()

{

System.out.println("Ceiling Fan is off");

}

}

class CeilingFanOnCommand implements Command {

CeilingFan c;

public CeilingFanOnCommand(CeilingFan l) {

this.c = l;

}

public void execute() {

c.on();

}

}

class CeilingFanOffCommand implements Command {

CeilingFan c;

public CeilingFanOffCommand(CeilingFan l) {

this.c = l;

}

public void execute() {

c.off();

}

}

class SimpleRemoteControl {

Command slot;

public SimpleRemoteControl() {}

public void setCommand(Command command) {

slot = command;

}

public void buttonWasPressed() {

slot.execute();

}

}

public class Main {

public static void main(String[] args) {

SimpleRemoteControl remote = new SimpleRemoteControl();

CeilingFan ceilingFan=new CeilingFan();

CeilingFanOnCommand ceilingFanOn =new CeilingFanOnCommand(ceilingFan);

remote.setCommand(ceilingFanOn);

remote.buttonWasPressed();

CeilingFanOffCommand ceilingFanOff =new CeilingFanOffCommand(ceilingFan);

remote.setCommand(ceilingFanOff);

remote.buttonWasPressed();

}

}

**7. Write a Java Program to implement Adapter pattern for Enumeration iterator.**

import java.util.\*;

class EnumerationIterator implements Iterator {

Enumeration enumeration;

public EnumerationIterator(Enumeration enumeration) {

this.enumeration = enumeration;

}

public boolean hasNext() {

return enumeration.hasMoreElements();

}

public Object next() {

return enumeration.nextElement();

}

public void remove() {

throw new UnsupportedOperationException();

}

}

public class Main {

public static void main (String args[]) {

Vector v = new Vector(Arrays.asList(args));

Iterator iterator = new EnumerationIterator(v.elements());

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

}

}

**8. Write a Java Program to implement Iterator Pattern for Designing Menu like Breakfast, Lunch or Dinner Menu.**

import java.util.Iterator;

public interface Menu {

public Iterator<?> createIterator();

String name;

public String getName() {

return name;

}

}

public class MenuItem {

String name;

String description;

boolean vegetarian;

double price;

public MenuItem(String name,

String description,

boolean vegetarian,

double price)

{

this.name = name;

this.description = description;

this.vegetarian = vegetarian;

this.price = price;

}

public String getName() {

return name;

}

public String getDescription() {

return description;

}

public double getPrice() {

return price;

}

public boolean isVegetarian() {

return vegetarian;

}

}

public class PancakeHouseMenu implements Menu {

ArrayList<MenuItem> menuItems;

public PancakeHouseMenu() {

name = "BREAKFAST";

menuItems = new ArrayList<MenuItem>();

addItem("K&B's Pancake Breakfast",

"Pancakes with scrambled eggs, and toast",

true,

2.99);

addItem("Regular Pancake Breakfast",

"Pancakes with fried eggs, sausage",

false,

2.99);

addItem("Blueberry Pancakes",

"Pancakes made with fresh blueberries, and blueberry syrup",

true,

3.49);

addItem("Waffles",

"Waffles, with your choice of blueberries or strawberries",

true,

3.59);

}

public void addItem(String name, String description,

boolean vegetarian, double price)

{

MenuItem menuItem = new MenuItem(name, description, vegetarian, price);

menuItems.add(menuItem);

}

public ArrayList<MenuItem> getMenuItems() {

return menuItems;

}

public Iterator<MenuItem> createIterator() {

return menuItems.iterator();

}

// other menu methods here

}

import java.util.Iterator;

public class DinerMenu implements Menu {

static final int MAX\_ITEMS = 6;

int numberOfItems = 0;

MenuItem[] menuItems;

public DinerMenu() {

name = "LUNCH";

menuItems = new MenuItem[MAX\_ITEMS];

addItem("Vegetarian BLT",

"(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);

addItem("BLT",

"Bacon with lettuce & tomato on whole wheat", false, 2.99);

addItem("Soup of the day",

"Soup of the day, with a side of potato salad", false, 3.29);

addItem("Hotdog",

"A hot dog, with saurkraut, relish, onions, topped with cheese",

false, 3.05);

addItem("Steamed Veggies and Brown Rice",

"Steamed vegetables over brown rice", true, 3.99);

addItem("Pasta",

"Spaghetti with Marinara Sauce, and a slice of sourdough bread",

true, 3.89);

}

public void addItem(String name, String description,

boolean vegetarian, double price)

{

MenuItem menuItem = new MenuItem(name, description, vegetarian, price);

if (numberOfItems >= MAX\_ITEMS) {

System.err.println("Sorry, menu is full! Can't add item to menu");

} else {

menuItems[numberOfItems] = menuItem;

numberOfItems = numberOfItems + 1;

}

}

public MenuItem[] getMenuItems() {

return menuItems;

}

public Iterator<MenuItem> createIterator() {

return new DinerMenuIterator(menuItems);

//return new AlternatingDinerMenuIterator(menuItems);

}

public

// other menu methods here

}

import java.util.Iterator;

public class DinerMenuIterator implements Iterator<MenuItem> {

MenuItem[] list;

int position = 0;

public DinerMenuIterator(MenuItem[] list) {

this.list = list;

}

public MenuItem next() {

MenuItem menuItem = list[position];

position = position + 1;

return menuItem;

}

public boolean hasNext() {

if (position >= list.length || list[position] == null) {

return false;

} else {

return true;

}

}

public void remove() {

if (position <= 0) {

throw new IllegalStateException

("You can't remove an item until you've done at least one next()");

}

if (list[position-1] != null) {

for (int i = position-1; i < (list.length-1); i++) {

list[i] = list[i+1];

}

list[list.length-1] = null;

}

}

}

public class Waitress {

ArrayList<Menu> menus;

public Waitress(ArrayList<Menu> menus) {

this.menus = menus;

}

public void printMenu() {

Iterator<?> menuIterator = menus.iterator();

System.out.print(MENU\n----\n);

while(menuIterator.hasNext()) {

Menu menu = (Menu)menuIterator.next();

System.out.print("\n" + menu.getName() + "\n");

printMenu(menu.createIterator());

}

}

void printMenu(Iterator<?> iterator) {

while (iterator.hasNext()) {

MenuItem menuItem = (MenuItem)iterator.next();

System.out.print(menuItem.getName() + ", ");

System.out.print(menuItem.getPrice() + " -- ");

System.out.println(menuItem.getDescription());

}

}

}

public class MenuTestDrive {

public static void main(String args[]) {

PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();

DinerMenu dinerMenu = new DinerMenu();

ArrayList<Menu> menus = new ArrayList<Menu>();

menus.add(pancakeHouseMenu);

menus.add(dinerMenu);

Waitress waitress = new Waitress(menus);

waitress.printMenu();

}

}

**9. Write a Java Program to implement State Pattern for Gumball Machine. Create instance variable that holds current state from there, we just need to handle all actions, behaviors and state transition that can happen. For actions we need to implement methods to insert a quarter, remove a quarter, turning the crank and display gumball.**

public interface State {

public void insertQuarter();

public void ejectQuarter();

public void turnCrank();

public void dispense();

public void refill();

}

public class NoQuarterState implements State {

GumballMachine gumballMachine;

public NoQuarterState(GumballMachine gumballMachine) {

this.gumballMachine = gumballMachine;

}

public void insertQuarter() {

System.out.println("You inserted a quarter");

gumballMachine.setState(gumballMachine.getHasQuarterState());

}

public void ejectQuarter() {

System.out.println("You haven't inserted a quarter");

}

public void turnCrank() {

System.out.println("You turned, but there's no quarter");

}

public void dispense() {

System.out.println("You need to pay first");

}

public void refill() { }

public String toString() {

return "waiting for quarter";

}

}

public class GumballMachine {

State soldOutState;

State noQuarterState;

State hasQuarterState;

State soldState;

State state;

int count = 0;

public GumballMachine(int numberGumballs) {

soldOutState = new SoldOutState(this);

noQuarterState = new NoQuarterState(this);

hasQuarterState = new HasQuarterState(this);

soldState = new SoldState(this);

this.count = numberGumballs;

if (numberGumballs > 0) {

state = noQuarterState;

} else {

state = soldOutState;

}

}

public void insertQuarter() {

state.insertQuarter();

}

public void ejectQuarter() {

state.ejectQuarter();

}

public void turnCrank() {

state.turnCrank();

state.dispense();

}

void releaseBall() {

System.out.println("A gumball comes rolling out the slot...");

if (count != 0) {

count = count - 1;

}

}

int getCount() {

return count;

}

void refill(int count) {

this.count += count;

System.out.println("The gumball machine was just refilled; it's new count is: " + this.count);

state.refill();

}

void setState(State state) {

this.state = state;

}

public State getState() {

return state;

}

public State getSoldOutState() {

return soldOutState;

}

public State getNoQuarterState() {

return noQuarterState;

}

public State getHasQuarterState() {

return hasQuarterState;

}

public State getSoldState() {

return soldState;

}

public String toString() {

StringBuffer result = new StringBuffer();

result.append("\nMighty Gumball, Inc.");

result.append("\nJava-enabled Standing Gumball Model #2004");

result.append("\nInventory: " + count + " gumball");

if (count != 1) {

result.append("s");

}

result.append("\n");

result.append("Machine is " + state + "\n");

return result.toString();

}

}

public class HasQuarterState implements State {

GumballMachine gumballMachine;

public HasQuarterState(GumballMachine gumballMachine) {

this.gumballMachine = gumballMachine;

}

public void insertQuarter() {

System.out.println("You can't insert another quarter");

}

public void ejectQuarter() {

System.out.println("Quarter returned");

gumballMachine.setState(gumballMachine.getNoQuarterState());

}

public void turnCrank() {

System.out.println("You turned...");

gumballMachine.setState(gumballMachine.getSoldState());

}

public void dispense() {

System.out.println("No gumball dispensed");

}

public void refill() { }

public String toString() {

return "waiting for turn of crank";

}

}

public class SoldState implements State {

GumballMachine gumballMachine;

public SoldState(GumballMachine gumballMachine) {

this.gumballMachine = gumballMachine;

}

public void insertQuarter() {

System.out.println("Please wait, we're already giving you a gumball");

}

public void ejectQuarter() {

System.out.println("Sorry, you already turned the crank");

}

public void turnCrank() {

System.out.println("Turning twice doesn't get you another gumball!");

}

public void dispense() {

gumballMachine.releaseBall();

if (gumballMachine.getCount() > 0) {

gumballMachine.setState(gumballMachine.getNoQuarterState());

} else {

System.out.println("Oops, out of gumballs!");

gumballMachine.setState(gumballMachine.getSoldOutState());

}

}

public void refill() { }

public String toString() {

return "dispensing a gumball";

}

}

public class SoldOutState implements State {

GumballMachine gumballMachine;

public SoldOutState(GumballMachine gumballMachine) {

this.gumballMachine = gumballMachine;

}

public void insertQuarter() {

System.out.println("You can't insert a quarter, the machine is sold out");

}

public void ejectQuarter() {

System.out.println("You can't eject, you haven't inserted a quarter yet");

}

public void turnCrank() {

System.out.println("You turned, but there are no gumballs");

}

public void dispense() {

System.out.println("No gumball dispensed");

}

public void refill() {

gumballMachine.setState(gumballMachine.getNoQuarterState());

}

public String toString() {

return "sold out";

}

}

public class GumballMachineTestDrive {

public static void main(String[] args) {

GumballMachine gumballMachine = new GumballMachine(2);

System.out.println(gumballMachine);

gumballMachine.insertQuarter();

gumballMachine.turnCrank();

System.out.println(gumballMachine);

gumballMachine.insertQuarter();

gumballMachine.turnCrank();

gumballMachine.insertQuarter();

gumballMachine.turnCrank();

gumballMachine.refill(5);

gumballMachine.insertQuarter();

gumballMachine.turnCrank();

System.out.println(gumballMachine);

}

}

**10. Write a java program to implement Adapter pattern to design Heart Model to Beat Model.**

interface BeatModelInterface {

void initialize();

void on();

void off();

void setBPM(int bpm);

int getBPM();

void registerObserver(BeatObserver o);

void removeObserver(BeatObserver o);

void registerObserver(BPMObserver o);

void removeObserver(BPMObserver o);

}

class BeatModel implements BeatModelInterface, MetaEventListener {

Sequencer sequencer;

ArrayList beatObservers = new ArrayList();

ArrayList bpmObservers = new ArrayList();

int bpm = 90;

// other instance variables here

public void initialize() {

setUpMidi();

buildTrackAndStart();

}

public void on() {

sequencer.start();

setBPM(90);

}

public void off() {

setBPM(0);

sequencer.stop();

}

public void setBPM(int bpm) {

this.bpm = bpm;

sequencer.setTempoInBPM(getBPM());

notifyBPMObservers();

}

public int getBPM() {

return bpm;

}

void beatEvent() {

notifyBeatObservers();

}

// Code to register and notify observers

// Lots of MIDI code to handle the beat

}

class DJView implements ActionListener, BeatObserver, BPMObserver {

BeatModelInterface model;

ControllerInterface controller;

JFrame viewFrame;

JPanel viewPanel;

BeatBar beatBar;

JLabel bpmOutputLabel;

public DJView(ControllerInterface controller, BeatModelInterface model) {

this.controller = controller;

this.model = model;

model.registerObserver((BeatObserver)this);

model.registerObserver((BPMObserver)this);

}

public void createView() {

// Create all Swing components here

}

public void updateBPM() {

int bpm = model.getBPM();

if (bpm == 0) {

bpmOutputLabel.setText("offline");

} else {

bpmOutputLabel.setText("Current BPM: " + model.getBPM());

}

}

public void updateBeat() {

beatBar.setValue(100);

}

}

interface ControllerInterface {

void start();

void stop();

void increaseBPM();

void decreaseBPM();

void setBPM(int bpm);

}

class BeatController implements ControllerInterface {

BeatModelInterface model;

DJView view;

public BeatController(BeatModelInterface model) {

this.model = model;

view = new DJView(this, model);

view.createView();

view.createControls();

view.disableStopMenuItem();

view.enableStartMenuItem();

model.initialize();

}

public void start() {

model.on();

view.disableStartMenuItem();

view.enableStopMenuItem();

}

public void stop() {

model.off();

view.disableStopMenuItem();

view.enableStartMenuItem();

}

public void increaseBPM() {

int bpm = model.getBPM();

model.setBPM(bpm + 1);

}

public void decreaseBPM() {

int bpm = model.getBPM();

model.setBPM(bpm - 1);

}

public void setBPM(int bpm) {

model.setBPM(bpm);

}

}

public class Main {

public static void main (String[] args) {

BeatModelInterface model = new BeatModel();

ControllerInterface controller = new BeatController(model);

}

}

**11. Design simple HR Application using Spring Framework**