**Behavioral Design Patterns**

**Vending Machine**

**GITHUB LINK:** [**https://github.com/gsiguenza12/CS5800-Homework/tree/main/Homework8**](https://github.com/gsiguenza12/CS5800-Homework/tree/main/Homework8)

**State and Chain of Responsibility**

Design a vending machine that dispenses different types of snacks based on:

1. User selection of snack item
2. The amount of money inserted
3. Availability of snacks in the Vending Machine

You should create a class to represent the state of the vending machine such as "Idle", "Waiting for Money", "Dispensing Snack", etc. Additionally, you will use the chain of responsibility design pattern to handle the requests for dispensing different types of snacks. A snack should only be dispensed if there is enough quantity available and enough money has been inserted by the user.

Your implementation should include the following class’s:

**Snack**– Represent all types of snacks that can be dispensed by the vending machine. Each Snack should have a name, price, and quantity.

**VendingMachine** - Should reference state of Vending Machine and SnackDispenser plus hold all snacks.

**StateOfVendingMachine**- Represents the different states of the vending machine. What are all the things vending machine can do?

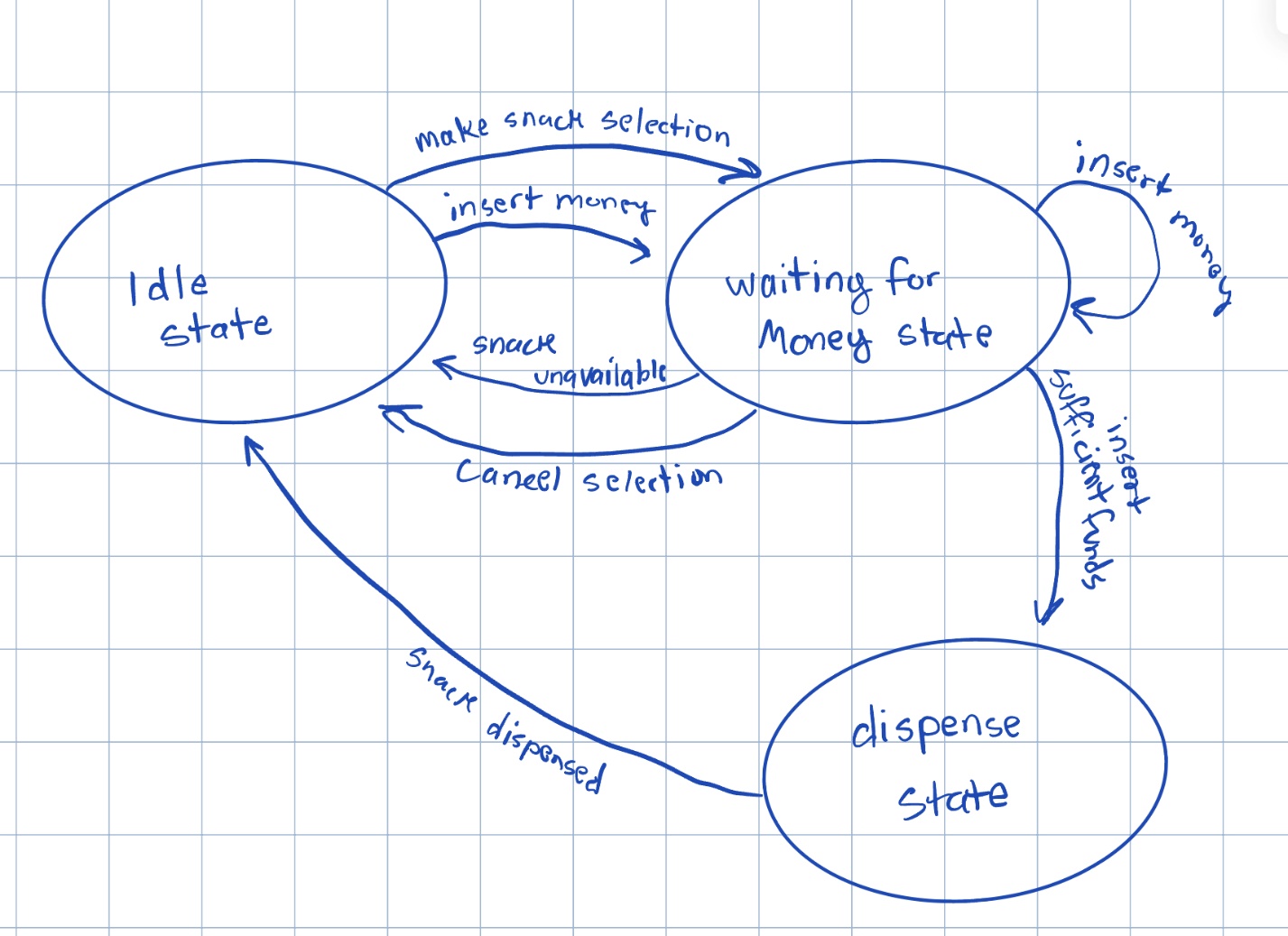
**SnackDispenseHandler**- Represents the chain of responsibility for dispensing different types of snacks. Handlers should have a reference to the next handler. Cycle through snacks until the proper one is dispensed.

Vending machine starts in idle state. User should select a snack, selectSnack(), from Vending Machine. Vending machine should wait for user to insert money, insertMoney(). If money inserted is >= price then dispense item if enough quantity or return money. To dispense snack, the VendingMachine class should call the dispenseSnack() method on the current state object. The state object will handle the event and transition to the appropriate state based on the availability of snacks.

Create **driver** with 6 different snacks and Chain of Responsibility in this order ->Coke, Pepsi, Cheetos, Doritos, KitKat, and Snickers.

Your driver should include at least one case where the quantity hits 0 with snickers.

HINT: Draw a graph showing states of Vending Machine. Draw pointed arrows between states showing what must occur for the state to transition, move from one state to another.



**SCREENSHOTS OF OUTPUT:**

A black rectangular object with white and blue spots

Description automatically generated with medium confidence

**CODE TEST COVERAGE REPORT:**

A screenshot of a computer

Description automatically generated

**UML**

**Class Diagram**

Your assignment is to go back to the **text editor** in which you utilized the flyweight design pattern and create a UML Class Diagram.

*NOTE*: Use the solution provided to you in the lecture notes.

Class UML:

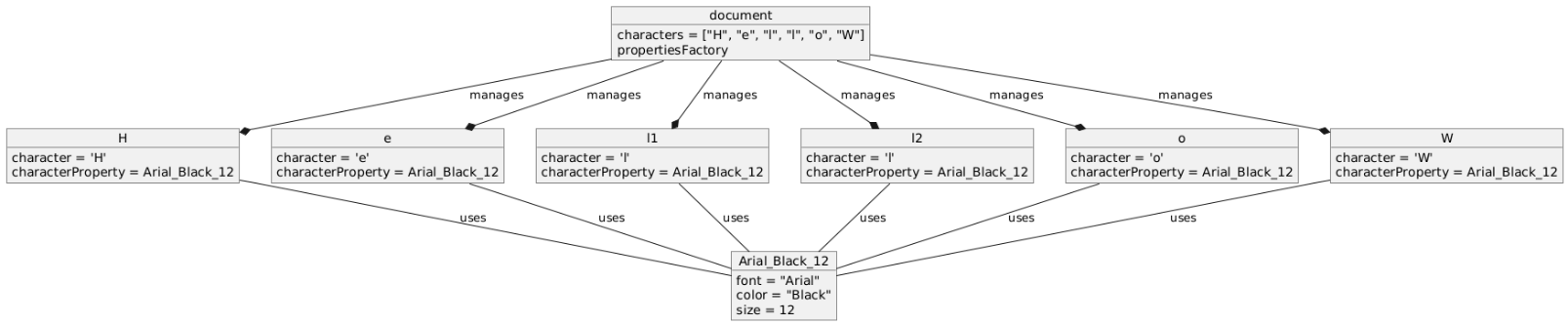
A black and white image of a pen

Description automatically generated with medium confidence

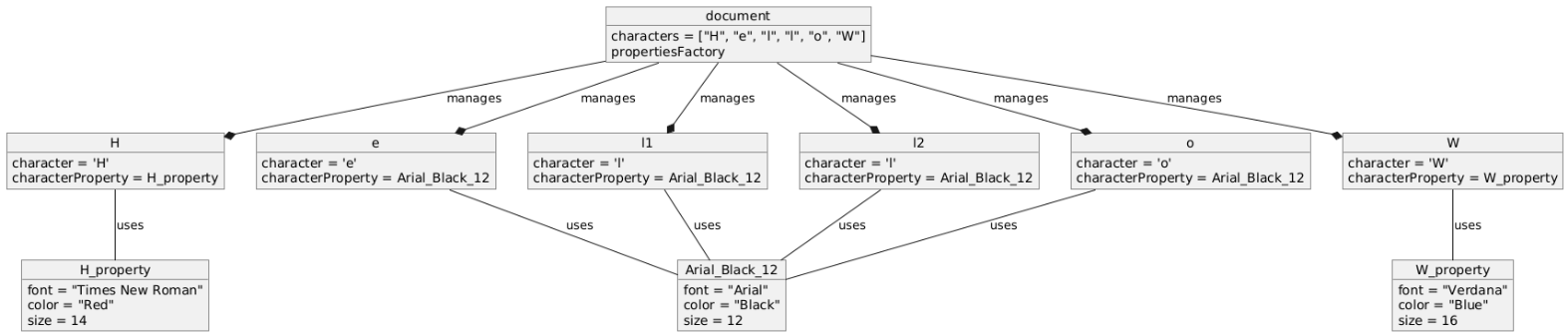
**Object Diagram**

Create the UML Object Diagram at two points.

1. After HelloWorldCS5800 is entered but, just before the Character Properties are modified.



1. And another Object Diagram just after the character properties H and W are modified.



Use any UML editor you prefer: PlantUML, LucidCharts, Visio…etc

**SOURCE CODE:**

package com.company;  
  
*/\*\*  
 \* State and Chain of Responsibility  
 \* Vending machine starts in idle state. User should select a snack, selectSnack(),  
 \* from Vending Machine.  
 \* Vending machine should wait for user to insert money, insertMoney().  
 \*  
 \* If money inserted is >= price  
 \* then dispense item if enough quantity or return money.  
 \* To dispense snack, the VendingMachine class should call the dispenseSnack()  
 \* method on the current state object. The state object will handle the event  
 \* and transition to the appropriate state based on the availability of snacks.  
 \*  
 \* Create driver with 6 different snacks and Chain of Responsibility in this order  
 \* ->Coke, Pepsi, Cheetos, Doritos, KitKat, and Snickers.  
 \*  
 \* Your driver should include at least one case where the quantity hits 0 with snickers.  
 \*/*public class Main {  
  
 public static void main(String[] args) {  
 // Create driver with 6 different snacks and Chain of Responsibility in this order ->Coke, Pepsi, Cheetos, Doritos, KitKat, and Snickers.  
 SnackDispenseHandler chain = new CokeDispenseHandler(new PepsiDispenseHandler(new CheetosDispenseHandler(new DoritosDispenseHandler(new KitKatDispenseHandler(new SnickersDispenseHandler(null))))));  
 VendingMachine machine = new VendingMachine(chain);  
  
 // add snacks to vendingmachine  
 machine.addSnack(new Snack("Coke", 2.99,10));  
 machine.addSnack(new Snack("Pepsi",2.99,10));  
 machine.addSnack(new Snack("Cheetos",3.75,10));  
 machine.addSnack(new Snack("Doritos",3.75,10));  
 machine.addSnack(new Snack("KitKat",1.99,10));  
 machine.addSnack(new Snack("Snickers",1.99,1));  
  
 // buying the last snickers  
 machine.selectSnack("Snickers");  
 machine.insertMoney(2.99);  
  
 // attempting to purchase another snickers  
 machine.selectSnack("Snickers");  
 machine.insertMoney(2.99);  
  
 System.*out*.println(machine.getSnacks());  
 }  
}

package com.company;  
  
import java.util.HashMap;  
import java.util.Map;  
  
*/\*\*  
 \* Should reference STATE of Vending Machine and SNACKDISPENSER plus hold all snacks.  
 \*/*public class VendingMachine {  
 private final Map<String, Snack> snacks = new HashMap<>();  
 private final SnackDispenseHandler handlerChain;  
 private VendingMachineState currentState;  
 private Snack selectedSnack;  
 private double currentMoney;  
  
 public double getCurrentMoney() {  
 return currentMoney;  
 }  
  
 public void setCurrentMoney(double currentMoney) {  
 this.currentMoney = currentMoney;  
 }  
  
 public VendingMachine(SnackDispenseHandler chain) {  
 this.handlerChain = chain; // set chain of command  
 currentState = new IdleState(this); // Vending machine starts in the Idle state  
 currentState.handleRequest();  
 }  
  
 public void addSnack(Snack snack){  
 snacks.put(snack.getName(),snack);  
 }  
  
 public void selectSnack(String snackName) {  
 System.*out*.println("Snack selected: " + snackName);  
 if (snacks.containsKey(snackName) && snacks.get(snackName).getQuantity() > 0) {  
 this.selectedSnack = snacks.get(snackName);  
 System.*out*.println("Setting selection to: " + snackName);  
 currentState = new WaitingForMoneyState(this);  
 System.*out*.println("Price for selection: $" + selectedSnack.getPrice());  
 System.*out*.println("Please insert money");  
 } else {  
 System.*out*.println("Snack not available. Please select another snack.");  
 }  
 }  
  
 public Snack getSelectedSnack() {  
 return selectedSnack;  
 }  
  
 public void invokeHandlerChain(Snack snack){  
 System.*out*.println("Handler chain invoked...");  
 handlerChain.handle(snack);  
 }  
  
 public void insertMoney(double money) {  
 // add money to vending machine  
 currentMoney += money;  
 currentState.handleRequest();  
 }  
  
 public VendingMachineState getCurrentState() {  
 return currentState;  
 }  
  
 public void setCurrentState(VendingMachineState state) {  
 this.currentState = state;  
 }  
  
 public Map<String, Snack> getSnacks() {  
 return snacks;  
 }  
  
}

package com.company;  
  
*/\*\*  
 \* Base class contains boilerplate code common to all handlers  
 \*/*public abstract class SnackDispenseHandler {  
 protected SnackDispenseHandler next;  
  
 public SnackDispenseHandler(SnackDispenseHandler next){this.next = next;}  
  
 public void handle(Snack requestSnack){  
 if(next != null){  
 next.handle(requestSnack);  
 }  
 }  
}

package com.company;  
  
public class CheetosDispenseHandler extends SnackDispenseHandler {  
 public CheetosDispenseHandler(SnackDispenseHandler next){super(next);}  
  
 @Override  
 public void handle(Snack requestSnack) {  
 if (requestSnack.getName().equalsIgnoreCase("Cheetos")) {  
 System.*out*.println("Dispensing Cheetos");  
 requestSnack.dispense();  
 } else if (next != null) {  
 System.*out*.println("I was passed from CheetosDispenseHandler");  
 next.handle(requestSnack);  
 }  
 }  
}

package com.company;  
  
public class DoritosDispenseHandler extends SnackDispenseHandler {  
 public DoritosDispenseHandler(SnackDispenseHandler next){super(next);}  
  
 @Override  
 public void handle(Snack requestSnack) {  
 if (requestSnack.getName().equalsIgnoreCase("Doritos")) {  
 System.*out*.println("Dispensing Doritos");  
 requestSnack.dispense();  
 } else if (next != null) {  
 System.*out*.println("I was passed from DoritosDispenseHandler");  
 super.handle(requestSnack);  
 }  
 }  
}

package com.company;  
  
public class KitKatDispenseHandler extends SnackDispenseHandler {  
 public KitKatDispenseHandler(SnackDispenseHandler next){super(next);}  
  
 @Override  
 public void handle(Snack requestSnack) {  
 if (requestSnack.getName().equalsIgnoreCase("KitKat")) {  
 System.*out*.println("Dispensing KitKat");  
 requestSnack.dispense();  
 } else if (next != null) {  
 System.*out*.println("I was passed from KitKatDispenseHandler");  
 super.handle(requestSnack);  
 }  
 }  
}

package com.company;  
  
public class SnickersDispenseHandler extends SnackDispenseHandler {  
 public SnickersDispenseHandler(SnackDispenseHandler next){super(next);}  
 @Override  
 public void handle(Snack requestSnack) {  
 if (requestSnack.getName().equalsIgnoreCase("Snickers")) {  
 System.*out*.println("Dispensing Snickers");  
 requestSnack.dispense();  
 } else if (next != null) {  
 System.*out*.println("I was passed from SnickersDispenseHandler");  
 super.handle(requestSnack);  
 }  
 }  
}

package com.company;  
  
public class PepsiDispenseHandler extends SnackDispenseHandler {  
 public PepsiDispenseHandler(SnackDispenseHandler next){super(next);}  
  
 @Override  
 public void handle(Snack requestSnack) {  
 if (requestSnack.getName().equalsIgnoreCase("Pepsi")) {  
 System.*out*.println("Dispensing Pepsi");  
 requestSnack.dispense();  
 } else if (next != null) {  
 System.*out*.println("I was passed from PepsiDispenseHandler");  
 super.handle(requestSnack);  
 }  
 }  
}

package com.company;  
  
public class CokeDispenseHandler extends SnackDispenseHandler {  
 public CokeDispenseHandler(SnackDispenseHandler next){super(next);}  
 @Override  
 public void handle(Snack requestSnack) {  
 if (requestSnack.getName().equalsIgnoreCase("Coke")) {  
 System.*out*.println("Dispensing Coke");  
 requestSnack.dispense();  
 } else if (next != null) {  
 System.*out*.println("I was passed from CokeDispenseHandler");  
 super.handle(requestSnack);  
 }  
 }  
}

package com.company;  
  
public class IdleState extends VendingMachineState {  
 public IdleState(VendingMachine vendingMachine) {  
 super(vendingMachine);  
 }  
  
 @Override  
 public void handleRequest() {  
 System.*out*.println("Machine is idle. Please select a snack.");  
 }  
}

package com.company;  
  
public class WaitingForMoneyState extends VendingMachineState {  
 public WaitingForMoneyState(VendingMachine vendingMachine) {  
 super(vendingMachine);  
 }  
  
 @Override  
 public void handleRequest() {  
 System.*out*.println("Machine is in waiting for money state...");  
 Snack snack = vendingMachine.getSelectedSnack();  
 if (vendingMachine.getCurrentMoney() >= snack.getPrice() && snack.getQuantity() > 0) {  
 System.*out*.println("Money accepted. Switching machine to dispense state");  
 vendingMachine.setCurrentState(new DispensingState(vendingMachine));  
 vendingMachine.getCurrentState().handleRequest();  
 } else {  
 System.*out*.println("Waiting for money. Please insert the correct amount.");  
 }  
 }  
}

package com.company;  
  
public class DispensingState extends VendingMachineState { // concrete state class provides its own implementation for state specific methods  
 public DispensingState(VendingMachine vendingMachine) {  
 super(vendingMachine);  
 }  
  
 @Override  
 public void handleRequest() {  
 System.*out*.println("The machine is in dispensing state...");  
 Snack selectedSnack = vendingMachine.getSelectedSnack();  
 if (selectedSnack != null && selectedSnack.getQuantity() > 0) {  
 vendingMachine.invokeHandlerChain(selectedSnack);  
 } else {  
 System.*out*.println(vendingMachine.getSelectedSnack().getName() + " is out of stock, please make another choice.");  
 }  
 System.*out*.println("Switching machine back to idle state...");  
 vendingMachine.setCurrentState(new IdleState(vendingMachine)); // switch current state back to idle state  
 vendingMachine.getCurrentState().handleRequest();  
 }  
}

**TESTS:**

package com.company;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
  
import java.util.Map;  
  
public class VendingMachineTest {  
 private VendingMachine vendingMachine;  
 private SnackDispenseHandler mockHandler;  
 private Snack snack;  
  
 @BeforeEach  
 public void setUp() {  
 mockHandler = new CokeDispenseHandler(new PepsiDispenseHandler(new CheetosDispenseHandler(new DoritosDispenseHandler(new KitKatDispenseHandler(new SnickersDispenseHandler(null))))));  
 vendingMachine = new VendingMachine(mockHandler);  
 snack = new Snack("Chips", 1.5, 10);  
 vendingMachine.addSnack(snack);  
 }  
  
 @Test  
 public void testInitialState() {  
 *assertTrue*(vendingMachine.getCurrentState() instanceof IdleState);  
 }  
  
 @Test  
 public void testAddSnack() {  
 Snack newSnack = new Snack("Chocolate", 2.0, 5);  
 vendingMachine.addSnack(newSnack);  
 Map<String, Snack> snacks = vendingMachine.getSnacks();  
 *assertTrue*(snacks.containsKey("Chocolate"));  
 *assertEquals*(newSnack, snacks.get("Chocolate"));  
 }  
  
 @Test  
 public void testSelectSnack() {  
 vendingMachine.selectSnack("Chips");  
 *assertEquals*(snack, vendingMachine.getSelectedSnack());  
 *assertTrue*(vendingMachine.getCurrentState() instanceof WaitingForMoneyState);  
 }  
  
 @Test  
 public void testSelectUnavailableSnack() {  
 vendingMachine.selectSnack("Candy");  
 *assertNull*(vendingMachine.getSelectedSnack());  
 *assertTrue*(vendingMachine.getCurrentState() instanceof IdleState);  
 }  
  
 @Test  
 public void testInsertMoney() {  
 vendingMachine.selectSnack("Chips");  
 vendingMachine.insertMoney(1.5);  
 *assertEquals*(1.5, vendingMachine.getCurrentMoney());  
 }  
  
 @Test  
 public void testInvokeHandlerChain() {  
 vendingMachine.selectSnack("Chips");  
 vendingMachine.insertMoney(1.5);  
 }  
  
 @Test  
 public void testIdleState() {  
 VendingMachineState idleState = new IdleState(vendingMachine);  
 vendingMachine.setCurrentState(idleState);  
 idleState.handleRequest();  
 *assertTrue*(vendingMachine.getCurrentState() instanceof IdleState);  
 }  
  
 @Test  
 public void testWaitingForMoneyState() {  
 VendingMachineState waitingForMoneyState = new WaitingForMoneyState(vendingMachine);  
 vendingMachine.setCurrentState(waitingForMoneyState);  
 *assertTrue*(vendingMachine.getCurrentState() instanceof WaitingForMoneyState);  
 }  
  
 @Test  
 public void testDispensingState() {  
 vendingMachine.selectSnack("Chips");  
 VendingMachineState dispensingState = new DispensingState(vendingMachine);  
 vendingMachine.setCurrentState(dispensingState);  
 *assertTrue*(vendingMachine.getCurrentState() instanceof DispensingState);  
 }  
}

package com.company;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
  
public class SnackTest {  
 private Snack snack;  
  
 @BeforeEach  
 public void setUp() {  
 snack = new Snack("Chips", 1.5, 10);  
 }  
  
 @Test  
 public void testGetName() {  
 *assertEquals*("Chips", snack.getName());  
 }  
  
 @Test  
 public void testGetPrice() {  
 *assertEquals*(1.5, snack.getPrice());  
 }  
  
 @Test  
 public void testGetQuantity() {  
 *assertEquals*(10, snack.getQuantity());  
 }  
  
 @Test  
 public void testDispense() {  
 snack.dispense();  
 *assertEquals*(9, snack.getQuantity());  
 }  
  
 @Test  
 public void testSetQuantity() {  
 snack.setQuantity(5);  
 *assertEquals*(5, snack.getQuantity());  
 }  
  
 @Test  
 public void testSetPrice() {  
 snack.setPrice(2.0);  
 *assertEquals*(2.0, snack.getPrice());  
 }  
  
 @Test  
 public void testToString() {  
 String expected = "Snack{name='Chips', price=1.5, quantity=10}";  
 *assertEquals*(expected, snack.toString());  
 }  
}

**PLANT UML CODE:**  
@startuml  
  
object document {  
 characters = ["H", "e", "l", "l", "o", "W"]  
 propertiesFactory  
}  
  
object H {  
 character = 'H'  
 characterProperty = H\_property  
}  
  
object e {  
 character = 'e'  
 characterProperty = Arial\_Black\_12  
}  
  
object l1 {  
 character = 'l'  
 characterProperty = Arial\_Black\_12  
}  
  
object l2 {  
 character = 'l'  
 characterProperty = Arial\_Black\_12  
}  
  
object o {  
 character = 'o'  
 characterProperty = Arial\_Black\_12  
}

'modification to the puml code to show part 2'   
object W {  
 character = 'W'  
 characterProperty = W\_property  
}  
  
object H\_property {  
 font = "Times New Roman"  
 color = "Red"  
 size = 14  
}  
  
object W\_property {  
 font = "Verdana"  
 color = "Blue"  
 size = 16  
}  
  
object Arial\_Black\_12 {  
 font = "Arial"  
 color = "Black"  
 size = 12  
}  
  
document --\* H : manages  
document --\* e : manages  
document --\* l1 : manages  
document --\* l2 : manages  
document --\* o : manages  
document --\* W : manages  
  
H -- H\_property : uses  
e -- Arial\_Black\_12 : uses  
l1 -- Arial\_Black\_12 : uses  
l2 -- Arial\_Black\_12 : uses  
o -- Arial\_Black\_12 : uses  
W -- W\_property : uses  
  
@enduml