**WIA1002 DATA STRUCTURE**

**TUTORIAL 3： ADTs**

**Question 1:**

**Consider the following problem:**

A new candy machine is purchased for the cafeteria, but it is not working properly. The candy machine has four **dispensers** to hold and release **items** sold by the candy machine as well as a **cash register**. The machine sells four products—**candies**, **chips**, **gum**, and **cookies**—each stored in a separate dispenser. You have been asked to write a program for this candy machine so that it can be put into operation.

The program should do the following:

• *Show* the **customer** the different products sold by the **candy machine**.

• Let the **customer** *make* the selection.

• *Show* the **customer** the **cost of the item** selected.

• *Accept* the **money** from the **customer**.

• *Return* the **change**.

• *Release* the **item**, that is, *make* the sale.

You can see that the program you are about to write is supposed to deal with dispensers and cash registers. That is, the main objects are four dispensers and a cash register.

Because all the dispensers are of the same type, you need to create a class, say, Dispenser, to create the dispensers. Similarly, you need to create a class, say, CashRegister, to create a cash register. You will create the class CandyMachine containing the four dispensers, a cash register, and the application program.

Your tasks are to design ADTs to represent the three classes:

1. a. Identify the instance variables for each of the class (i.e. Dispenser, Cash Register, Candy Machine)
2. b. Identify the methods/operations for each of the class (i.e. Dispenser, Cash Register, Candy Machine)
3. c. Produce a UML class diagram to represent the three classes

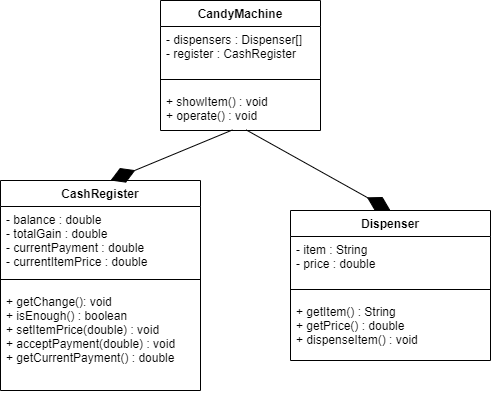
public class Main {  
 public static void main(String[] args) {  
 CandyMachine machine = new CandyMachine()**;** machine.operate()**;** }  
}

public class Dispenser {  
 private String item**;** private double price**;** Dispenser(String item**,** double price){  
 this.item=item**;** this.price=price**;** }  
 public String getItem(){  
 return item**;** }  
 public double getPrice(){  
 return price**;** }  
 public void dispenseItem(){  
 System.*out*.println(item+" is dispensed!")**;** }  
}

public class CashRegister {  
 private double balance**;** //money in machine  
 private double totalGain**;** private double currentPayment**;** private double currentItemPrice**;** CashRegister(){  
 this.balance=**100;** }  
 public void getChange(){  
 double change=currentPayment-currentItemPrice**;** if(change>balance){  
 System.*out*.println("Not enough balance!")**;** }  
 else{  
 balance-=change**;** currentPayment=currentItemPrice=**0;** //update the cP and CIP to 0 so next item can enter(?)

System.*out*.printf("Your change: RM %.2f\n"**,**change)**;** }  
 }  
 public boolean isEnough(){  
 return currentPayment>=currentItemPrice**;** }  
 public void setItemPrice(double currentItemPrice){  
 this.currentItemPrice=currentItemPrice**;** }  
 public void acceptPayment(double userPayment){  
 totalGain+=userPayment**;** currentPayment+=userPayment**;** }  
 public double getCurrentPayment(){  
 return this.currentPayment**;** }  
  
 public double getTotalGain() {  
 return this.totalGain**;** }  
}

import java.util.Scanner**;**public class CandyMachine {  
 public Dispenser [] dispensers**;** public CashRegister register**;** CandyMachine(){  
 this.dispensers=new Dispenser[**4**]**;** dispensers[**0**]= new Dispenser("Candy"**,2.50**)**;** dispensers[**1**]=new Dispenser("Chips"**,1.50**)**;** dispensers[**2**]=new Dispenser("Gum"**,1.00**)**;** dispensers[**3**]=new Dispenser("Cookies"**,3.00**)**;** this.register=new CashRegister()**;** }  
 public void showItem(){  
 System.*out*.println("Item\t\tPrice")**;** for(int i=**0;**i<dispensers.length**;**i++) {  
 System.*out*.printf("%d. %-10s RM %.2f\n"**,**i+1**,**dispensers[i].getItem()**,**dispensers[i].getPrice())**;** }  
 }  
 public void operate(){  
 Scanner sc=new Scanner(System.*in*)**;** System.*out*.println("The product: ")**;** showItem()**;** System.*out*.println("\nSelect the item")**;** int choice=sc.nextInt()-**1;** System.*out*.println(dispensers[choice].getItem()+" is RM"+dispensers[choice].getPrice())**;** register.setItemPrice(dispensers[choice].getPrice())**;** while(!register.isEnough()){  
 System.*out*.printf("Current payment: %.2f; Payment: "**,**register.getCurrentPayment())**;** //cP get 0register.acceptPayment(sc.nextDouble())**;** }  
 register.getChange()**;** dispensers[choice].dispenseItem()**;** sc.close()**;** }  
}



Aggregation since the cashRegister and dispensers are independently from candyMachine



Six types of relationship

Need different arrow

Association, aggregation(child can exist independently), composition(child can’t exist independently)

**Question 2:**

A bid for installing an air conditioner consists of the name of the company, a description of the unit, the performance of the unit, the cost of the unit, and the cost of installation. Design an ADT that represents a single bid for installing an air conditioning unit. Write a Java interface named BidInterface to specify the following ADT operations by stating its purpose, precondition, postcondition, parameters using javadoc-style comments:

* • Returns the name of the company making this bid.
* • Returns the description of the air conditioner that this bid is for.
* • Returns the capacity of this bid's AC in tons (1 ton = 12,000 BTU).
* • Returns the seasonal efficiency of this bid's AC (SEER).
* • Returns the cost of this bid's AC.
* • Returns the cost of installing this bid's AC.
* • Returns the yearly cost of operating this bid's AC.

Then design another ADT to represent a collection of bids. The second ADT should include methods to search for bids based on price and performance. Also note that a single company could make multiple bids, each with a different unit. Write a Java interface named BidCollectionInterface to specify the following ADT operations by stating its purpose, precondition, postcondition, parameters using javadoc-style comments:

* • Adds a bid to this collection.
* • Returns the bid in this collection with the best yearly cost.
* • Returns the bid in this collection with the best initial cost. The initial cost will be defined as the unit cost plus the installation cost.
* • Clears all of the items from this collection.
* • Gets the number of items in this collection.
* • Sees whether this collection is empty.

The **/\*\*** format is called a documentation comment in Java and is used to generate documentation automatically using tools like Javadoc.

The **/\*\*** format for a comment indicates that it is a documentation comment and not just a regular code comment. It's important to note that this format is specific to documentation comments and is not required for regular code comments.

Purpose, precondition, postcondition, parameter (need write in Javadoc)

public interface BidInterface {  
 */\*\*  
 \* Returns the name of the company making this bid.*

*\** ***postcondition: the name was returned*** *\** ***@return*** *the name of the company  
 \*/* public String companyName()**;** */\*\*  
 \* Returns the description of the air conditioner that this bid is for.  
 \** ***@return*** *the description of the air conditioner  
 \*/* public String description()**;** */\*\*  
 \* Returns the capacity of this bid's AC in tons (1 ton = 12,000 BTU).  
 \** ***@return*** *the capacity of the AC  
 \*/* public double getCapacity()**;** */\*\*  
 \* Returns the seasonal efficiency of this bid's AC (SEER).  
 \** ***@return*** *the seasonal efficiency of the AC  
 \*/* public double SEER()**;** */\*\*  
 \* Returns the cost of this bid's AC.  
 \** ***@return*** *the cost of the AC  
 \*/* public double bidCost()**;** */\*\*  
 \* Returns the cost of installing this bid's AC.  
 \** ***@return*** *the cost of installing the AC  
 \*/* public double installBidCost()**;** */\*\*  
 \* Returns the yearly cost of operating this bid's AC.*

*Postcondition: the yearly cost was returned  
 \** ***@return*** *the yearly cost of operating the AC  
 \*/* public double yearlyOperatingCost(double hourOperated,double energyCost)**;**}

public interface BidCollectionInterface <**E** extends BidInterface>{  
  
 */\*\*  
 \* Add a bid to this collection  
 \** ***@param*** *bidInterface bidInterface to be added  
 \*/* public void addBid(**E** bidInterface)**;** */\*\*  
 \* Return the bid in this collection with the best yearly cost  
 \** ***@return*** *the bid in this collection with the best yearly cost  
 \*/* public double bidBestYearlyCost()**;** */\*\*  
 \* Return the bid in this collection with the best initial cost.  
 \* The initial cost will be defined as the {****@code*** *unit cost} + {****@code*** *installation cost}  
 \** ***@return*** *bid the bid in this collection with the best initial cost  
 \*/* public double BestInitialCost()**;** */\*\*  
 \* Clear all the items from the collection  
 \*/* public void clear()**;** */\*\*  
 \* Get the number of items in this collection  
 \** ***@return*** *the number of items in this collection  
 \*/* public void getNofItem()**;** */\*\*  
 \* See whether this collection is empty  
 \** ***@return*** *{****@code*** *true} if the collection is empty, false otherwise  
 \*/* public void isEmpty()**;**}