WIA1002/WIB1002 Data Structure

Tutorial: 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:

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

a. **Instance variables for**

Dispenser Class

- productName (String)
- productCost (double)
- numberOfItems (int)

Cash Register Class

- cashBalance (double)

Candy Machine Class

- candyDispenser (Dispenser)
- chipsDispenser (Dispenser)
- gumDispenser (Dispenser)
- cookiesDispenser (Dispenser)
- cashRegister (CashRegister)

b. **Methods / Operations for**

Dispenser Class

- Dispenser(String productName, double productCost, int numberOfItems): Argument constructor to initializes the dispenser
- getProductName(): returns the name of the product
- getProductCost(): returns the cost of the product
- getNumberOfItems(): returns the number of the items in the dispenser
- makeSale(): Reduce the number of items in the dispenser by 1
- setName(String productName): Accepts the parameter and set the name of the product
- setCost(double productCost): Accepts the parameter and set the cost of the product
- setItemCount(int numberOfItems): Accepts the parameter and set the number of items in the dispenser

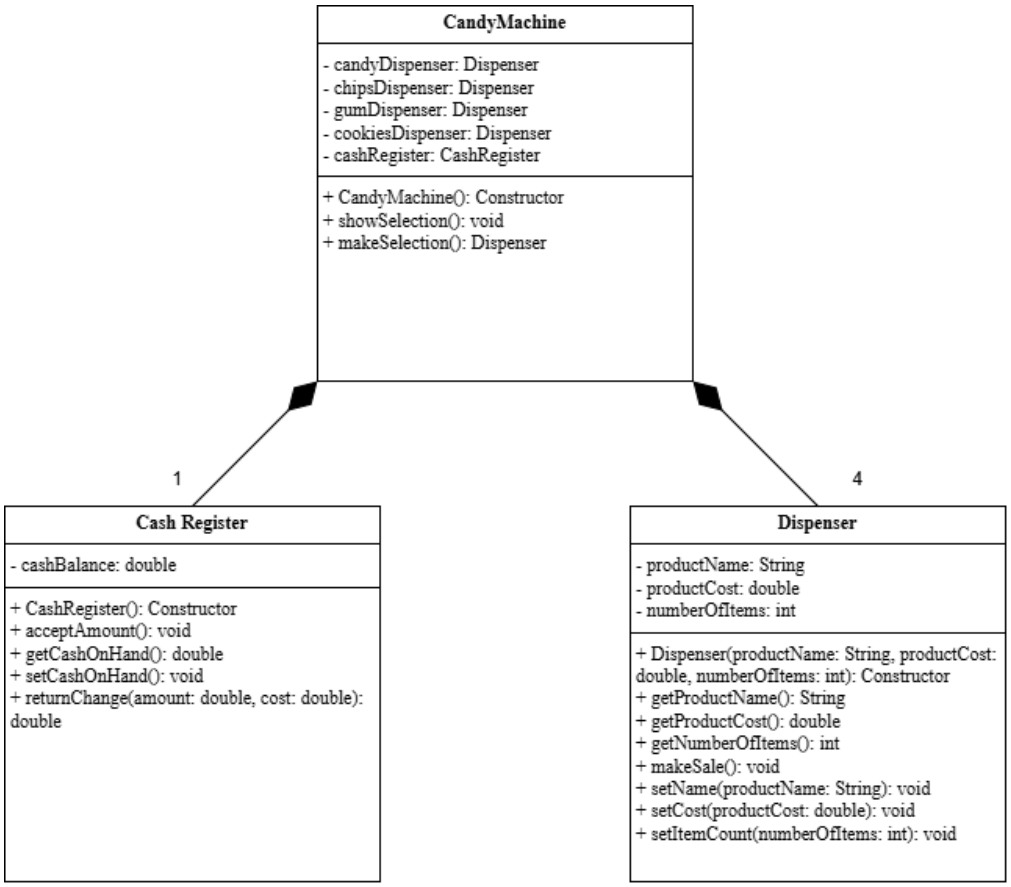
Cash Register Class

- CashRegister(): No-argument constructor that initializes the cash register with cashBalance with default value of 0
- acceptAmount(): Adds the amount of money to cashBalance
- getCashOnHand(): Returns the current amount of cashBalance
- setCashOnHand(): Set the amount of cashBalance
- returnChange(double amount, double cost): Calculates and returns change

Candy Machine Class

- CandyMachine(): No-argument constructor that initializes all dispensers and cash register
- showSelection(): Displays available products sold by the candy machine
- makeSelection(): Returns the Dispenser object based on the item selected

c. **UML Class Diagram**



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.

BidInterface interface

```
/**
An interface to calculate a bid for installing an air conditioner
*/
public interface BidInterface {

   /** Returns the name of the company making this bid.
    *  Precondition: none
    *  Postcondition: the name was returned
    *  @return the name of the company making this bid.
    */
   public String getCompanyName();

   /** Returns the description of the air conditioner that this bid is for.
    *  Precondition: none
    *  Postcondition: the description was returned
    *  @return the description of the air conditioner
    */
   public String getDescription();

   /** Returns the capacity of this bid's AC in tons (1 ton = 12,000 BTU).
    *  Precondition: none
    *  Postcondition: the capacity of this bid's AC in tons was returned
    *  @return the capacity of this bid's AC in tons
    */
   public double getCapacity();

   /** Returns the seasonal efficiency of this bid's AC (SEER).
    *  Precondition: none
    *  Postcondition: the seasonal efficiency of this bid's AC (SEER) was returned
    *  @return the seasonal efficiency of this bid's AC (SEER)
    */
   public double getSEER();

   /** Returns the cost of this bid's AC.
    *  Precondition: none
    *  Postcondition: the cost of this bid's AC was returned
    *  @return the cost of this bid's AC.
    */
   public double getACCost();

   /** Returns the cost of installing this bid's AC.
```

```
    *  Precondition: none
    *  Postcondition: the cost of installing this bid's AC was returned
    *  @return the cost of installing this bid's AC
    */
   public double getInstallationCost();

   /** Returns the yearly cost of operating this bid's AC.
    *  Precondition: none
    *  Postcondition: the yearly cost of operating this bid's AC was returned
    *  @param hoursOperated Average number of hours the unit operates per year.
    *  @param energyCost Cost in dollars per kilowatt hour.
    *  @return the yearly cost of operating this bid's AC in dollars,
    * cost = 12 * tons * energyCost * hoursOperated / SEER
    */
   public double getYearlyOperationCost(double hourOperated, double energyCost);

} // end BidInterface
```

BidCollectionInterface interface

```
/**
An interface to represent  collection of bids
 */
public interface BidCollectionInterface {

   /** Adds a bid to this collection
    *  Precondition: none
    *  Postcondition: The bid was added at the end of the collection
    *  @param bid the bid to dd to this collection
    */
   public void addBid(BidInterface bid);


   /** Returns the bid in this collection with the best yearly cost.
    *  Precondition: The collection is not empty
    *  Postcondition: The bid with the lowest yearly cost was returned
    *  @param averageHours Average hours of operation per year
    *  @param energyCost Cost in dollars per kilowatt hour
    *  @return the bid in this collection with the best yearly cost.
    */
   public BidInterface bestYearlyCost(double averageHours, double energyCost);
```

```
    /** 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.
     *  Precondition: The collection is not empty
     *  Postcondition: The bid with the lowest initial cost was returned
     *  @return the bid in this collection with the best initial cost.
     */
    public BidInterface bestInitialCost();

    /** Clears all of the items from this collection.
     *  Precondition: none
     *  Postcondition: The collection is empty
     */
    public void clear();

    /** Gets the number of items in this collection.
     *  Precondition: none
     *  Postcondition: The collection is unchanged
     *  @return the number of items in this collection
     */
    public int getSize();

    /** Sees whether this collection is empty.
     *  Precondition: none
     *  Postcondition: The collection is unchanged
     *  @return True if there are no items in the bid collection, false otherwise
     */
    public boolean isEmpty();

} // end BidCollectionInterface
```