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

Sample Solution:

a. Identify the instance variables for each of the class

Dispenser: To make the sale, at least one item must be in the dispenser and the customer must know the cost of the product. Therefore, the instance variables of a dispenser are:

- Product cost - `cost`
- Number of items in the dispenser - `numberOfItems`

Cash Register: The cash register accepts money and returns change. Therefore, the cash register has only one instance variable, which we call `cashOnHand`.

Candy Machine: The class CandyMachine has four dispensers and a cash register. You can name the four dispensers by the items they store. Therefore, the candy machine has five instance variables:

- four dispensers – `candy`, `chip`, `gum`, and `cookie`
- a cash register – `cRegister`

b. Identify the methods/operations for each of the class (i.e. Dispenser, Cash Register, Candy Machine)

The relevant verbs are *show* (selection), *make* (selection), *show* (cost), *accept* (money), *return* (change), and *make* (sale).

- The verbs *show* (selection) and *make* (selection) relate to the candy machine.
- The verbs *show* (cost) and *make* (sale) relate to the dispenser.
- Similarly, the verbs accept (money) and *return* (change) relate to the cash register.

Dispenser: The verb *show* (cost) applies to either printing or retrieving the value of the data member cost. The verb *make* (sale) applies to reducing the number of items in the dispenser by 1. Of course, the dispenser has to be nonempty. You must also provide an operation to set the cost and the number of items in the dispenser. Thus, the operations for a dispenser object are:

- `getCount`: Retrieve the number of items in the dispenser.
- `getProductCost`: Retrieve the cost of the item.
- `makeSale`: Reduce the number of items in the dispenser by 1.
- `setCost`: Set the cost of the product.
- `setNumberOfItems`: Set the number of items in the dispenser.

Cash Register: The verb *accept* (money) applies to updating the money in the cash register by adding the money deposited by the customer. Similarly, the verb *return* (change) applies to reducing the money in the cash register by returning the overpaid amount (by the customer) to

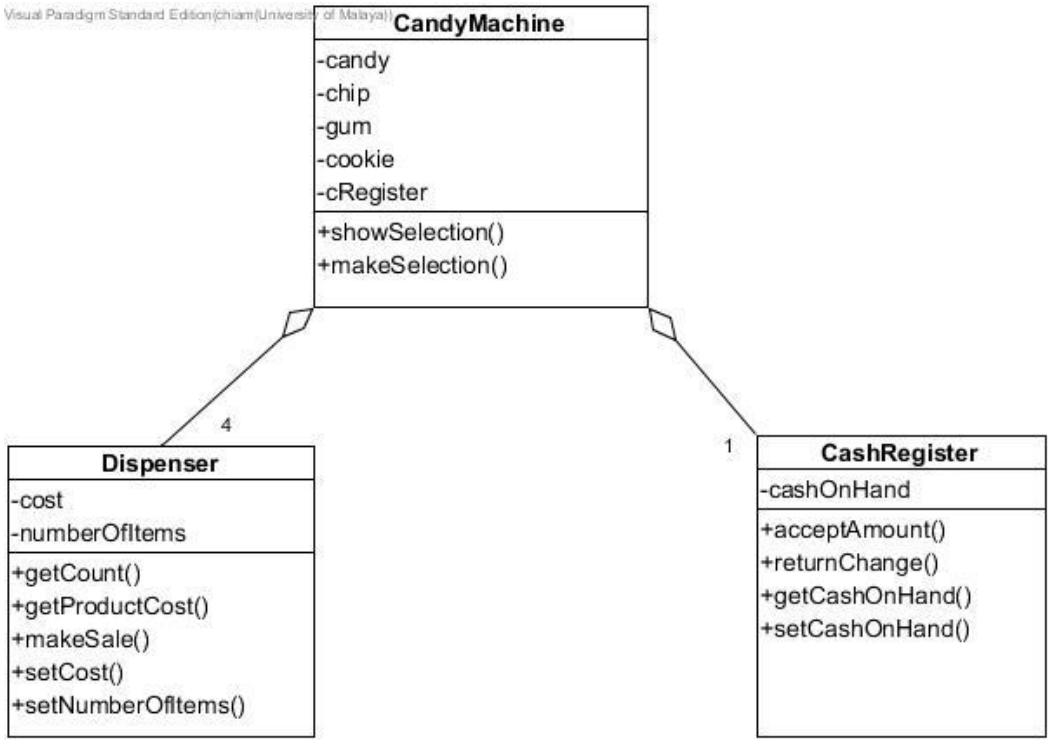
the customer. You also need to (initially) set the money in the cash register and retrieve the money from the cash register. Thus, the possible operations on a cash register are:

- `acceptAmount`: Update the amount in the cash register.
- `returnChange`: Return the change.
- `getCashOnHand`: Retrieve the amount in the cash register.
- `setCashOnHand`: Set the amount in the cash register.

Candy Machine: The verbs *show* (selection) and *make* (selection) apply to the candy machine. Thus, the two possible operations are:

- `showSelection`: Show the number of products sold by the candy machine.
- `makeSelection`: Allow the customer to select the product.

c. Sample 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.

Sample Solution:

BidInterface:

```
/**
   This is an interface for a single bid for installing an air conditioning
   unit.
*/
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 the 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 performance was returned.
            @return  The cooling capacity of the AC unit in tons. */
      public double getCapacity();

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

      /**   Returns the cost of this bid's AC.
            Precondition:  None.
            Postcondition:  The AC cost was returned.
            @return  The cost of the unit in dollars. */
      public double getUnitCost();

      /**   Returns the cost of installing this bid's AC.
            Precondition:  None.
            Postcondition:  The installation cost was returned.
            @return  The cost of installation in dollars. */
      public double getInstallationCost();

      /**   Returns the yearly cost of operating this bid's AC.
            Precondition:  None.
            Postcondition:  The yearly cost was returned.
            @param hoursOperated  Average number of hours the unit
            operates per year.
            @param energyCost Cost in dollars per kilowatt hour.
            @return The cost for the year in dollars,
             cost = 12 * tons * energyCost * hoursOperated / SEER. */
```

```
	public double getYearlyCost(double hoursOperated, double energyCost);

} // end BidInterface
```

BidCollectionInterface:

```
/**
This is an interface for a collection of objects each of which is a Bid for
installing an air conditioning unit.
*/

public interface BidCollectionInterface
{
	/**	Adds a bid to this collection.
		Precondition:  None
		Postcondition:  The bid was added at the end of the collection.
		@param toAdd  The bid to add. */
	public void add(BidInterface toAdd);

	/**	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  A bid with the lowest 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  A bid with the lowest 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. */
	public int getLength();

	/**	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
```