COMP8710 A1 - Vending Machine

For this assignment you will be modelling a vending machine. You will take the problem specification given in this assignment and implement it in Java. Your code should be clean, well-documented, and thoroughly tested. You should be able to demonstrate an understanding of fundamental concepts in object-oriented programming like encapsulation, abstraction, inheritance, and polymorphism.

1. Submission Instructions

- You will submit a zip file containing your project folder using the submission link on Moodle before the deadline. *A mark of zero is normally awarded to late or non-submission.*

  This zip file should contain all your Java code, in exactly the format and structure required to run it. Please verify before submission that your code will compile and run.

  If the code in the zip file does not compile as-is, it will receive no marks for functionality.

- Additionally, in your zip file you should include a README.txt file. This file should contain your name, login, and a description of your vending machine program. You should list all the interfaces and classes you created and write a short (one or two sentences) summary of what they do and why.

2. Problem Specification

2.1 Requirements

You are to model a vending machine that has the following characteristics:

- The machine accepts British pound sterling in the form of coins.
- The machine contains a variety of items.
- Users may deposit coins into the machine.
- Once enough money has been deposited, users may withdraw an item of their choice.
- Users may cancel their purchase and withdraw the money they have deposited.
- Once an item has been purchased, the remaining change is returned to the user.
- The owner of the vending machine can add new contents and deposit/withdraw money.

This document will go over these requirements in more details, but it will be ultimately up to you how you design your program. You will be evaluated on the design choices that you make. You should also develop some tests for your code and run experiments to convince yourself that it is functioning properly and meeting all the above requirements. See Section 4 for details about evaluation criteria.

2.2 Payment

The vending machine you are modelling should accept payments from users in the form of coins. For the purposes of this program, you should support the following kinds of coin payments: 2 pounds, 1 pounds, 50 pence, 20 pence, 10 pence, 5 pence, and 1 penny.

The user will deposit one or more of these forms of payment, and then purchase an item from the machine. If the amount of money deposited is insufficient to purchase that item, the machine must signal an error. Similarly, if change must be returned to the user but insufficient coins are in the machine, the machine must signal an error.

At any point prior to purchase, the user may request a refund of the money they have deposited. In this situation, the machine will return their coins to them.

2.3 Machine Contents

Each vending machine should have a name. It will contain several kinds of drinks. Each kind of drinks has a name, code, and price. The code consists of two digits, e.g. “01”. The user will enter the code to purchase an item from the machine, and the machine should signal an error if none of the selected item is in stock.

Here are some examples of items in a vending machine:

Code	Name	Price
01	Coke	1.25
02	Water	1.00
03	Milk	1.50

Because vending machines come in different sizes, the vending machine software must be configurable in terms of the maximum number of items that it can hold. However, once a vending machine has been set up, these limits should not change. Attempting to add contents to the machine exceeding the limits should signal an error.

2.4 Machine state

Your vending machine design must have some notion of state. For example, after a customer has deposited several coins, but not yet made a purchase, the state of the machine should account for the balance deposited but not spent. After a purchase does happen, the machine state will need to be updated to make note of the sold item and the accumulated money.

Read the requirements and think carefully about how the state of the machine must change over time.

2.5 Interactions

Interactions with the vending machine fall into two broad categories: customers purchasing items from the machine, and owners administering the machine.

- Purchasing
  - Customers may deposit coins.
  - Customers may request a refund of coins.
  - Customers may retrieve coins from the refund bucket.
  - Customers may retrieve a purchased item.
- Administering
  - Owners may add items to the machine.
  - Owners may deposit or withdraw money from the machine.

Your system should be carefully designed to distinguish between these kinds of interactions, e.g. by preventing customers from withdrawing all the money from the machine. You should think carefully how you might accomplish this by using class design and inheritance (where customers and owners interact with distinct subclasses, that have different available methods).

Obviously, not every interaction is valid at all times, e.g. a customer may not retrieve an item if insufficient money has been deposited. You should design your classes in such a way that such invalid actions are prevented.

3. Examples and Hints

For this assignment, you need to come up with the classes yourself. This may be an intimidating assignment, if you are not accustomed to writing code on your own. To help get you started, here are some sample codes that you can draw some inspiration from. You are *not* required to use any of this, nor is it necessarily the best or most optimal design.

You will want to come up with the API for how customer interact with the vending machine. One way to define such an API is through an Interface, which might look something like this:

```
/**
* API for customer interacting with the vending machine
*/
public interface VMCustomerAPI {

    // Customer inserts a coin
    void insertCoin(Coin coin);

    // Returns current balance of inserted coins
    int getCurrentBalance();

    // Customer selects item
    // throws exception if not found
    void selectItem(String code) throws PurchaseException;

    // Returns currently selected item’s code
    String getItemCode();

    // Customer requests a refund, coins to be placed in return bucket
    void requestRefund();

    // Customer requests purchasing selected item
    // throws exception on error, puts item in return bucket
    public void requestPurchaseItem() throws PurchaseException;

    // Customer requests change, coins to be placed in return bucket
    void requestChange();

    // Customer collects refund, or item and change, from return bucket
    // Update states of the vending machine and return bucket
    void collect();

}
```

If you choose to go with an interface like this, then whatever class you design that implements the interface needs to have fields to track the machine state, including the current amount deposited, the currently selected item, the amount in the return bucket, the stock of each item, etc..

Furthermore, if you were to use an interface like this, you will also need to (at the very least) decide what Item and Coin should be. Because Coin was specified as one of several distinct things as specified in the section “Payment”, perhaps you could consider using an enum.

Additionally, you will want to come up with an API for how the owner interacts with the machine. One such interface might look like this:

```
/**
* API for owner interacting with the vending machine
*/
public interface VMAdminAPI {

    // Take the money out of the machine
    List<Coin> withdrawCoins();

    // Add money to the machine
    void addCoin(Coin money);

    // Add an item in the machine
    // throws an exception if already full
    void addItem(Item item) throws VMFullException;

}
```

You may find that other methods are necessary for the owner’s interface.

Obviously, in both cases there are other ways you might want to design an interface for a vending machine, and you are not forced to use these. There may be improvements and optimizations you can make on top of the above code that have been left out of this example.

4. Evaluation Criteria

4.1 Marking scheme

The breakdown for marking is as follows:

- **Functionality of your submission [50%]**: your code should successfully support all the requirements as specified in this assignment. It should compile and run, and it should perform correctly without crashing with an uncaught exception.
- **Object-oriented design [20%]**: how well you designed your classes and interfaces, whether you followed the principles as explained in the lectures.
- **Testing [20%]**: your main() method should test all the assignment requirements.
- **Code quality [10%]**: your code should be clean and easy to understand. It should be properly formatted and indented with appropriate comments and documentation.

4.2 Design and code quality

You will primarily be evaluated on the design decisions that you make, and the quality of your Java code.

- All your code must successfully compile. *No credit will be given to code that fails to compile.*
- As this is an “advanced” Java programming course, you are also expected to document and format your code to make it clean and readable.

- You should use what you have learned about designing classes and object-oriented programming to complete this assignment. Your code should use interfaces, classes, objects, etc, to “model” the behaviour of the vending machine as specified.

4.3 Testing requirements

As part of your program, you should write some code to test out the various parts of your system. This should include testing all the different possible interactions between the machine and users, i.e. customers and owners.

If you are familiar with JUnit, you may use that for implementing your tests. However, this is not required for this assignment.

You are required to have a main() method that runs a sample series of actions on the vending machine, for testing purposes. The main method should create a vending machine, and have an admin add items and coins to it; and then have a customer to insert money, purchase items, etc.; also have an admin to stock items, withdraw money, etc.. Every step of the way, you should print what is happening to the terminal, and at the end you should print the state of the vending machine.

Try to make your testing as thorough as possible.