Assignment 1

Vending Machine

COMP7007

<2025-10-29 Wed>

1 Problem Specification

1.1 Summary

For this assignment you will be modeling a **vending machine**. It is an individual coding assignment. You must write code to model a vending machine in Java.

You will take the problem specification given in this assignment, and you will write Java code to implement it. Your code should be clear, 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**. You should also think about how your code may be extended in the future, and design your classes to follow the **Open/Closed Principle** (open for extension, closed for modification). You may also want to consult the lecture slides on **design patterns** to determine if any of them may be applied to this problem.

1.2 Overall Design

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

- The machine accepts British pound sterling in the form of coins.
- The machine contains a variety of drinks and snacks.
- Users may deposit coins into the machine.
- Once a sufficient amount of money has been deposited, users may withdraw a drink or snack of their choice.
- Users may cancel their purchase and withdraw the money they have deposited.
- Once a drink or snack 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 arbitrarily, but ordinary users cannot.
- The vending machine has a small screen that displays text notifications for each action the machine makes.

- Multiple variations on the vending machine will be produced in the future, some of which may be larger or smaller and stock a different variety of options.

Your goal is to design a system that covers as many of these requirements as possible.

This document will go over these requirements in more detail, but it will be ultimately up to you how you design your program. You will be evaluated on the design choices that you make. See: Section 4. 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.

1.3 Payment

The machine you are modeling 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, 1 pence.

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 provided to the user but insufficient coins are in the machine, the machine must signal an error. The user **cannot** receive the product if they do not pay the required amount.

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.

1.4 Machine Contents

The contents of the vending machine fall into two categories: bottled drinks, and packaged snacks. Each machine will contain several kinds of drinks and several kinds of snacks. Each kind of drink and kind of snack will have price, as well as an associated code in the form of a four digit number. The user will enter this code to purchase an item from the machine, and the machine should signal an error if none of that kind of item is in stock.

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 (that is, there is a maximum number of each item that can be stocked in the machine). However, once a vending machine has been set up, these limits **should not** change. Attempting to add contents to the machine in excess of these limits should signal an error.

Vending machiens will stock the following items:

Type	Name	Code	Price
Drink	Coke	0001	2.00
Drink	Sprite	0002	2.00
Drink	Water	0003	1.50
Drink	Lemonade	0004	1.75
Snack	Crisps	1001	1.50
Snack	Peanuts	1002	1.50
Snack	Chocolate	1003	2.50
Snack	Candy	1004	2.00

1.5 Machine state

Your vending machine design must have some notion of **state**. For example, after a user 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 lost item and accumulated money.

Look at the requirements in Section 1.2 and think carefully about how the state of the machine must change over time.

1.6 Interactions

Interactions with the vending machine fall into two broad categories: ordinary users (buyers) purchasing items from the machine, and owners administering the machine. Your system should be carefully designed to distinguish between these kinds of interactions; for example, by preventing ordinary users from withdrawing all the money from the machine. How might you use class design and class design and/or design patterns to accomplish this?

Because we are doing an exercise in system design and not actually programming a real vending machine, you will need to write testing code that simulates the interactions with the machine. How you do this is up to you. For example, you may choose to simulate the vending machine display screen by printing text to the console.

Of course, not every interaction is valid at all times. A user may not retrieve an item if insufficient funds have been deposited, for example. You should design your classes in such a way that such invalid actions are prevented.

1.6.1 Purchasing

- Users may deposit coins.
- Users may request a refund of coins.
- Users may retrieve coins from the refund bucket.
- Users may retrieve a purchased item.

1.6.2 Administering

- Owners may add items to the machine.
- Owners may deposit or withdraw money from the machine.

2 Examples and Hints

Unlike most homework assignments you have had in the past, you will **not** be given a template with classes already set up and told to fill in the blanks. 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 is some **sample** code 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 users interact with the machine. One way to define such an API is through an **Interface**, which might look something like this (remember, this may be incomplete):

```
/**
  * API for users interacting with the vending machine
  */
public interface VendingMachine {

    // User inserts a coin
    public void insertCoin(Coin coin);

    // Returns current balance of inserted coins
    public int amountDeposited();

    // User gets coins from return bucket
    public List<Coin> getReturnCoins();

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

    // User selects item
    public void selectItem(String code);

    // Returns currently selected item
    public String currentItem();

    // User purchases item
    // throws exception on error, puts change in return bucket
    public Item purchaseItem() throws MachinePurchaseException;

    // Returns the number of items in stock
    public int getStock(String code);

}
```

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 (see Section 1.3), perhaps you could consuder 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:

```
public interface AdminVendingMachine {

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

    // Add money to the machine
    public void depositCoins(List<Coin>);

    // Stocks an item
    // throws an exception if already full
    public void stockItem(Item e) throws MachineAdminException;

}
```

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

Of course, 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/optimizations you can make on top of the above code that have been left out of this example.

3 Evaluation Criteria

3.1 Testing requirements

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

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

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, add items and coins to it, and then have a user insert money, purchase items, etc. Every step of the way, you should print what is happening to the screen, and at the end you should print the state of the vending machine.

Try to make your testing as thorough as possible.

3.2 Design

3.2.1 Overall design

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

Of course, all of your code must successfully compile—no credit will be given to code that fails to compile—but, 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 classes, objects, interfaces, etc, to "model" the behavior of the vending machine.

3.2.2 How your design will be evaluated

Some amount of subjectivity is involved in the marking of your OOP design. This is a rough outline of what we will look for when evaluating your design.

Note that there are no absolute right answers here. You do not need to include every single thing covered in the lectures into your code—in fact, you shouldn't! The specification in prior sections is deliberately vague in places, to allow room for you to make your own choices. As explained in the instructions you will write a `README.txt` file explaining and justifying the choices you made.

Some guidelines about the evaluation of your design:

- If the specification says that something *should not change* then your Java code should enforce this. For example, a field marked `final` cannot be modified after it is initialised, or a protection Proxy can be used to selectively prevent mutation of parts of an object.
- Similarly, if the specification says that something *should not be accessible* by certain clients then your Java code should enforce this.
- Whenever possible, your Java code should *prevent* objects from ever containing invalid data/state. You might do this by limiting mutation of objects, and/or guarding mutation of objects to prevent bad data from entering.
- Your design should be *extensible* (as in, new features and functionality may be added) while minimising the changes that must be made to existing classes. For example, you should be careful to avoid stuffing a lot of behavior and data into big, monolithic classes.

Whatever choices you make, be sure to explain your reasoning in the accompanying `README.txt` file. If you want to make some assumptions about how your vending machine code will be used, document it in this file.

Your code will be judged by how well it matches the specification and follows the guidelines above. If your code implements all the required functionality but does not follow any of these guidelines, it will receive 0 of the 30% for the "design" portion of the marks. Code that successfully enforces all of the invariants mentioned in the guidelines above should receive at least 15. For a full 30, you should further demonstrate some thought about how your code and design would be extended in the future.

3.3 Assignment marking

The breakdown for marking is as follows:

- 50% – the functionality of your submission: your code should successfully support the requirements listed in Section 1.2—it should compile and run,

and it should perform correctly without crashing with an uncaught exception.

- 30% – the object-oriented design: how you designed your classes and interfaces, whether you followed the principles as explained in the lectures.
- 10% – the code quality: whether you have properly formatted and indented code, the comments and documentation, etc.
- 10% – the testing: your main method should test all the assignment requirements.

4 Submission Instructions

You will submit a `zip` file containing your project folder to **Moodle**. This `zip` file should contain **all** of your Java code, in exactly the format and structure required to run it. Please verify before submitting that your code will compile and run—if the code in the `zip` 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 username and a description of your vending machine program. You should list all the classes you created and write a short (one or two sentence) summary of what they do and why.

This is an **individual coding assignment**. You are expected to complete this assignment by yourself. You cannot share code or copy code from your classmates. You can consult whatever online sources you want but you cannot copy and paste entire blocks of code from the Internet or from AI. If you do consult example code from somewhere, you must ensure that you **cite** where you found the code and explain how you used it in your comments.