Requirements

Your program must have the following features:

Features

- The program should **display all of the items** and their respective prices when the program starts, along with an **option to exit the program**.
- The user must put in **some amount of money before an item can be selected**.
- Only **one item can be vended at a time**.
- If the user selects an item that costs more than the amount the user put into the vending machine, the program should display a message indicating **insufficient funds** and then **redisplay the amount the user had put into the machine.**
- If the user selects an item that costs equal to or less than the amount of money that the user put in the **vending machine, the program should display the change returned to the user**. Change must be displayed as the number of quarters, dimes, nickels, and pennies returned to the user.
- **Vending machine items must be stored in a file**. Inventory for the vending machine must be read from this file when the program starts and must be written out to this file just before the program exits. The program must track the following properties for each item:
    - **Item name**
    - **Item cost**
    - **Number of items in inventory**
- **When an item is vended, the program must update the inventory level appropriately. If the machine runs out of an item, it should no longer be available as an option to the user.** However, the items that have an inventory level of zero must still be read from and written to the inventory file and stored in memory.

Guidelines

- This application must follow the MVC pattern used for all previous labs (App class, Controller, View, Service Layer, DAO) – this includes the use of constructor based dependency injection.
- You must have a full set of unit tests for your DAO and Service Layer components.
- You must use BigDecimal for all monetary calculations where applicable.

- You must use application specific exceptions and your application must fail gracefully under all conditions (i.e. not displaying a stack trace when an exception is thrown). At a minimum you should have the following application specific exceptions thrown by your Service Layer:
    - One that is thrown when the user tries to purchase an item but doesn't deposit enough money (i.e. `InsufficientFundsException`)
    - One that is thrown when the user tries to purchase an item that has zero inventory (i.e. `NoItemInventoryException`)
- Use enums to represent the values of different coins.
- Include an Audit DAO to log events and the time they occurred.

Description of my implementation

My implementation attempts to use MVC with a service layer.

It has two DAOs - the **AuditDAO** and the **VendingDAO**. The AuditDAO just writes out notifications of reading, writing, and vending to a text file. It uses `java.time.LocalDateTime` to get the current time for logging. I have written five interfaces implemented by the - **AuditDAO**, **VendingController**, **VendingDAO**, **VendingServiceLayer**, and **VendingView**.

Even though the application is command line, I decided to still have a separate view interface to stick to MVC better. The View this time takes more responsibility for data validation from the user - I thought that in a real application, it would likely have a GUI which by default would provide some level of input validation e.g. buttons would only allow certain options to be selected.

Compared to my last assignment (DVD Library), my Controller delegates more responsibility to the View and the Service Layer and gives them more trust to do what they're supposed to. The Controller simply tells the View or the Service Layer to do something and handles any exceptions thrown.

The **VendingController** has a startup and exit routine which both involve using the service layer to use the DAO to read and write to a file. Then, in a do while loop, it will repeatedly ask the view to show the menu options and depending on the response, will either display the items in the machine, purchase an item, or exit.

Other classes

App: simply kicks off the application. Contains the main method to run.

VendingItem: The data transfer object which represents an item in the vending machine. Actually represents a family of items i.e. all those with the same name because it stores the quantity for us.

Coins: an Enum with values to represent coins.

Menu Options: an Enum with values to represent the options for the menu screen so that if extra features are added, then nothing else needs to change.

Payment: A class to handle the validation (items in stock and enough funds deposited) and to return a HashMap<Coins, Integer> representing the change provided to the user. It is used by the Service Layer when purchasing the item. It takes in a HashMap representing the user's coin input and converts it to a BigDecimal to perform the checks. Then it will calculate the change due and calculate the coins to be returned and these are passed back up to the Controller.

DAOTest: tests to check that the DAO operations - reading and writing from/to the file - work as expected.

ServiceTest: tests to ensure that the operations performed in the Service Layer, are working as expected.

Exceptions

DAOException: covers any issue with reading or writing from the storage. This could be a file not found problem or a formatting problem.

InsufficientFundsException, NoItemInventoryException: Thrown by the Payment class and taken all the way up to the controller, when the user hasn't entered enough money or when the item stock level is zero.

ItemNotFoundException: Thrown by the Service Layer when the item searched for does not exist.

MenuChoiceException: Thrown by the View when the user inputs an invalid selection.