# Java Coding Challenge

Author: Lei Li
Date: 13/07/21

## 1. Thought Processes

This application will be used in an ATM or similar device that supports only $20 and $50 denominations. However, it needs to be well designed to allow further extensions, for example, the addition of a $10 denomination.

My overall concern is to make it reusable and emulate the real operation and transaction process as much as possible. It is essential to use Object Oriented Programming to structure the application with simple, reusable pieces of code and with well-defined public interfaces. The top-down approach has been used. Figure 1 is the sketch of the structure for this design and the relationship between the three classes.

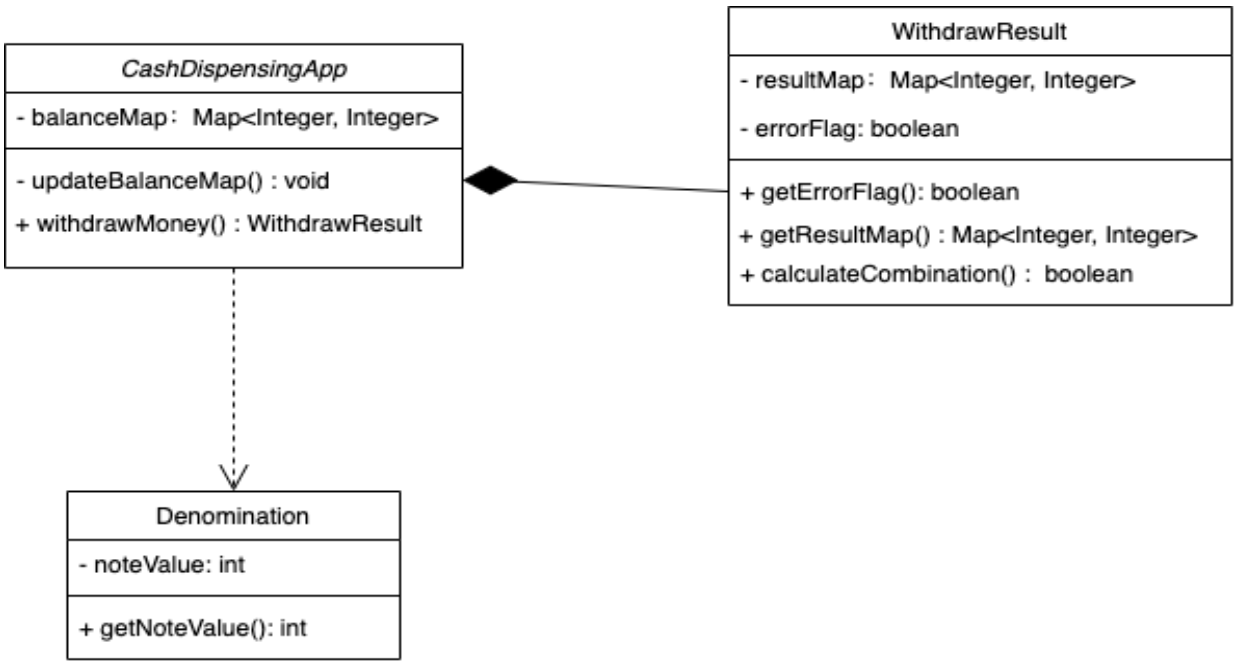


Figure1

The *Denomination* and *WithdrawResult* classes are implemented independently to guarantee low coupling and high cohesion, and it would be easy to either add various denominations or modify the algorithm without the need to modify the *CashDispensingApp* class.

### 1.1 *CashDispensingApp* class

It is also important to make sure the information is secure as a user shouldn't be able to inspect the data inside a device like an ATM machine. A HashMap is used to maintain the state of the balance, and the balance will be updated automatically after a successful transaction. The balance information is kept internal to the class by setting it as a private attribute, and the user will only interact with the public method *withdrawMoney*.

### 1.2 *Denomination* class

A device like an ATM machine should be able to recognize various denominations automatically. This class is used to emulate the banknotes, and it is implemented as an Enum class. In the initialisation process, the program will iterate through all the denominations and update the balance. Adding or removing denominations will not affect the performance of the program.

### 1.3 *WithdrawResult* class

Every transaction requested will return a result which is either successful or indicates failure. This class is implemented to do the calculation and record the result. An error flag will be set when a transaction fails, and a HashMap is used to obtain the combination of the denominations used to satisfy the request.

### 1.4 The algorithm

A dynamic programming algorithm is used to determine the note combination. It would be able to solve the problem even if other denominations were added. Although basic recursion would be sufficient for this problem, it has very high time complexity and is not reusable at all when other denominations are needed.

This is a classic knapsack problem where the requested amount by the user is the capacity, and the banknotes are the items whose weight and value are same as the face amount. Success is only indicated when full capacity can be achieved.

## 2. Future features

This implementation is a mix of required features and optional features, as the controller is persistent, and the application would be able to support all other denominations already.

With additional time, I would like to implement ATM balance notifications (for resupply of the ATM) and different error notifications. The application now will output the same error message for all possible transaction failures, which may not make much sense to the users.

I would also like to modify the *WithdrawResult* class to make it as simple as possible and maybe create another class “Transaction” to do all the calculation, and break the calculation down into smaller methods to guarantee maintainability.