## **Chain of Responsibility**

## Description of the design problem

* 1. The method called **giveChange(int changeRequired)** in **ChangeGiver** class uses a **for** loop to give changes. It manually starts the loop from the last **CashStoreItem** in **CashStore** just because its coin value is “1$” which is the highest value in the **items** lists. So we understand it wants to give changes from the high value coin to low value coin. However, this is not a good practice because the sequence of **CashStoreItem** depends on the **CashPropertyFile.txt** so this is very unreasonable. We also do not think using **for** loop for each give changes action is a good practice. So we want find a way to resolve those problems.

## Candidate design patterns considered

* Composite design pattern

Composite is often used in tree structures represent part-whole hierarchies. So we want use it to traverse the **CashStoreItem**s, a component's parent can act as one **CashStoreItem**’s successor. However to use composite patterns, we need new Composite class, leaf class and component class. The relationship between each **CashStoreItem** is not pure part-whole hierarchies. Though this pattern can be used for recursive problems, we do not think add more classes and change the hierarchy is a good approach.

* Chain of Responsibility

This issue is a behavioral issue, chain of responsibility is used when people want to issue a request to one of several objects without specifying the receiver explicitly.

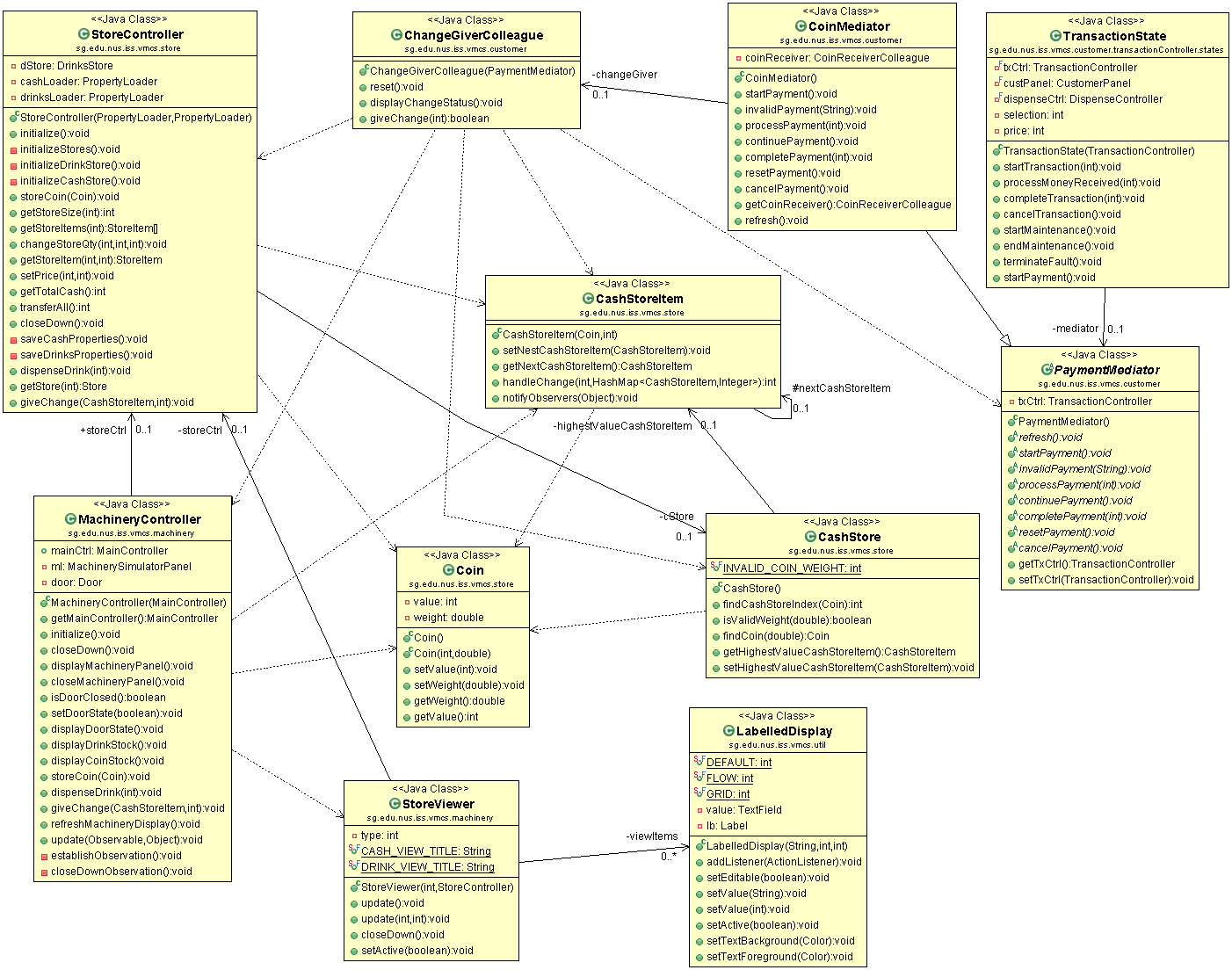
## Motivation to choose a pattern that would solve the problem including support for new requirements or changes to existing requirements

Chain of Responsibility is suitable for this problem.

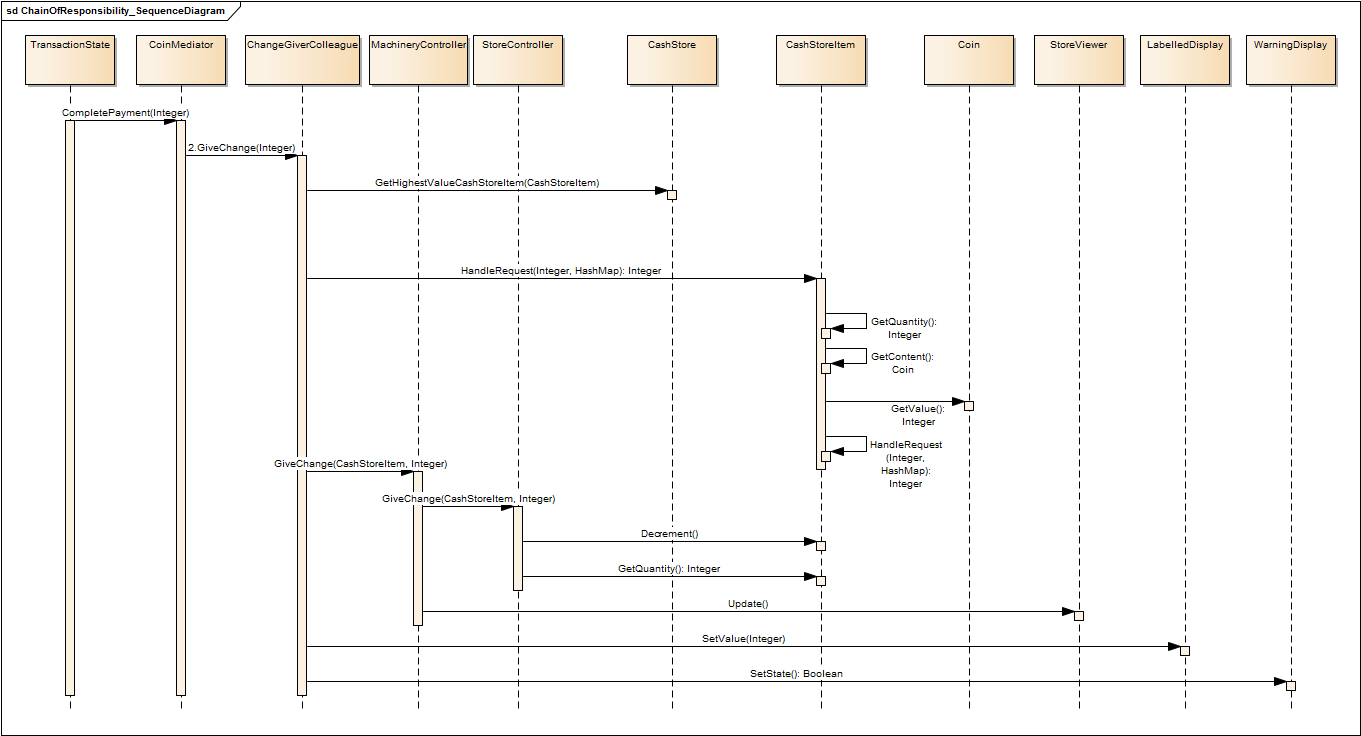
By using Chain of Responsibility, **giveChange** method no need to use **for** loop to execute give change action. We now just need to get one **CashStoreItem** as the head of the chain, then send the give change request by using **handleRequest** method to its successor on the chain. We do not need to know who the successor is because the chain was built when the **CashStore** was initialized.

By using this pattern, we de-couple the **ChangeGiver** class. The **ChangeGiver** class now just need know the first **CashStoreItem**(1$), then begin the give changes action.

## Structure of the pattern (you should map the participants to your applications classes/objects),



## Collaborations among the participants (specific to your application objects)



## Implementation decisions that you have taken.

1. Where to build the chain?

At the beginning, we want to build the chain in the **giveChange** method, but we think it is important to make this method simple and only let it do what it should do. We want to build the chain only once. So, we build the chain in the **initialCashStore()** method.

1. Sort the **CashStoreItem** from 1$ to 5C

The best way to give change is to give from the high value because this save the coins. But the original code does not sort coin, just using the sequence in the property file which is not reasonable. So we decide to create a List and sort the cash item no matter how many items in the property file and what the value is, we can solve the problem by the List.

When we build the chain, we just get the first object in this List and set the **handleRequest** in the sequence of the List.

1. Using index or Object in the process of giveChange?

The original code uses index to get the **CashStoreItem** in items list. But if the sequence of items changed, or the property file changed, the result will be wrong. So we find it’s better to use **CashStoreItem** as the parameter of the **giveChange(CashStoreItem item, int numOfCoins)** method.

1. Give the change in every handleRequest or Collect the results and Give changes together after the handleRequest ends?

At the beginning, we used **giveChange** method in the process of **handleRequest**. But we consider that **handleRequest** is only need to get the required quantity of coin, no need to do the **giveChange** action. So we finally create a HashMap object to collect the quantity required for each **CashStoreItem**. After the chain finished, we use the HashMap to get the Key which is the **CashStoreItem** and the value which is the required quantity. Then using a **for** loop to give changes.