|  |
| --- |
|  |
| OODP Design Report |
|  |

|  |
| --- |
| **Team: FT02**  Xu Minsheng (A0134434M)  Liu Xinzhuo (A0136010A)  Xie Jiabao (A0109328E)  Lu Angxiao (A0136021Y)  2015-10-25 |

# Contents

[Contents 1](#_Toc433634035)

[1. Design Issues 3](#_Toc433634036)

[2. Solutions with Design Pattern 4](#_Toc433634037)

[2.1 Observer Pattern 4](#_Toc433634038)

[2.1.1 Description of the design problem 4](#_Toc433634039)

[2.1.2 Candidate design patterns considered 4](#_Toc433634040)

[2.1.3 Motivation to choose a pattern that would solve the problem including support for new requirement s or changes to existing problems 4](#_Toc433634041)

[2.1.4 Structure of the pattern (you should map the participants to your applications classes/objects) 5](#_Toc433634042)

[2.1.5 Collaborations among the participants (specific to your application objects) 6](#_Toc433634043)

[2.1.6 Implementation decision that you have taken 8](#_Toc433634044)

[2.2 State Pattern 9](#_Toc433634045)

[2.2.1 Description of the design problem 9](#_Toc433634046)

[2.2.2 Candidate design patterns considered 9](#_Toc433634047)

[2.2.3 Motivation to choose a pattern that would solve the problem including support for new requirement s or changes to existing problems 9](#_Toc433634048)

[2.2.4 Structure of the pattern (you should map the participants to your applications classes/objects) 10](#_Toc433634049)

[2.2.5 Collaborations among the participants (specific to your application objects) 12](#_Toc433634050)

[2.2.6 Implementation decision that you have taken 12](#_Toc433634051)

[2.3 Mediator Pattern 13](#_Toc433634052)

[2.3.1 Description of the design problem 13](#_Toc433634053)

[2.3.2 Candidate design patterns considered 13](#_Toc433634054)

[2.3.3 Motivation to choose a pattern that would solve the problem including support for new requirements or changes to existing problems 13](#_Toc433634055)

[2.3.4 Structure of the pattern 15](#_Toc433634056)

[2.3.5 Collaborations among participants 16](#_Toc433634057)

[2.3.6 Implementation decision 17](#_Toc433634058)

[2.4 Chain of Responsibility Pattern 17](#_Toc433634059)

[2.4.1 Description of the design problem 17](#_Toc433634060)

[2.4.2 Candidate design patterns considered 18](#_Toc433634061)

[2.4.3 Motivation to choose a pattern that would solve the problem including support for new requirements or changes to existing requirements 18](#_Toc433634062)

[2.4.4 Structure of the pattern (you should map the participants to your applications classes/objects), 19](#_Toc433634063)

[2.4.5 Collaborations among the participants (specific to your application objects) 20](#_Toc433634064)

[2.4.6 Implementation decisions that you have taken. 22](#_Toc433634065)

[3. Modified design document 23](#_Toc433634066)

[4. Team member Contributions 24](#_Toc433634067)

# Design Issues

According to the documents and project, our team have identified 4 design issues:

(1) There is too much coupling between StoreItem, MachineryController, DispenseController Objects.

(2) The TransactionController is too big to maintain and modify. In addition, the TransactionController has too many responsibilities. And in this class there are some code redundancy problems.

(3) There is tight coupling between TransactionController, CoinReceiver and ChangeGiver class, which means it is hard to maintain and extend.

(4) 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.

# Solutions with Design Pattern

## 2.1 Observer Pattern

### 2.1.1 Description of the design problem

There is too much coupling between StoreItem, MachineryController, DispenseController Objects.

It means that whenever there is any operation to change the quantity of StoreItem (Coin or Drink), for example if there is a need to add a use case to clear all drinks from the store for maintainer, the MachineryController and DispenseController may need to be called as a part of the operation.

Also if there is a need to add a use case to sync the storage data with an online data Centre, all operations that will change item’s quantity must be modified.

To sum up, it violates the Open for extension, Close for modification Principle.

### 2.1.2 Candidate design patterns considered

Since the problem is caused by tight coupling between class, and it is a behavioral issue, the candidate patterns are

* Observer
* Mediator

### 2.1.3 Motivation to choose a pattern that would solve the problem including support for new requirement s or changes to existing problems

In this case Observer Pattern is suitable.

The nature of problem is how to assure the consistency of StoreItem’s quantity between multiple Objects without making the classes tightly coupled.

By introducing Observer Pattern, it reduces the coupling because no matter what operation that change the StoreItem’s quantity, the operation itself does not need to worry about make the related objects to be consistent, hence it does not need to know what are these related objects.

Furthermore, it becomes easier to extend to support new requirement for both side (Subject and Observer) due to both depend on abstraction. Changes to each side will not impact another.

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



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

Establish Observation:



Notify:



### 2.1.6 Implementation decision that you have taken

1. Mapping subjects to their observers.

A hashmap is used to store the mapping between subjects to observer.

Each subject object has a list of observer reference. We consider it is fine because there is not a lot of subjects and observers that would incur storage overhead.

1. Observing more than one subject

In this case, the Machinery Controller needs to observer all StoreItems of Coin and Drink. It may not be so efficient that once a state of one item changed, all items must be updated. Thus we choose to override the notifyObservers method of both CashStoreItem and DrinksStoreItem to pass a parameter to specify which type of Item is changed, only objects that are interesting in this type will be updated.

But to improve performance furthermore, the Machinery Controller has to figure which particular item is changed and to update the related object, will make the codes more complicate.

1. Who triggers the update

Making the subject to be responsible for calling the Notify is more preferable. Even though this approach may lower the performance by consecutive updates, the end result is consistent. And the benefit is we can avoid the bug caused from client forget to call Notify.

1. Dangling references to deleted subjects

Since the observer references are stored as subject’s attribute, when a subject is deleted, its attribute will be deleted as well.

1. Making sure subject state is self-consistent before notification

Subject inherit the Java Lib class Observable, and Notify is the last operation in methods which change its state.

1. Avoiding observer-specific update protocols

In this case, by inheriting Java Lib class Observable, the subject’s reference is passed to observer regardless of it is needed or not for simplicity. So that the observer can decide whether to use it or not.

1. Specifying modifications of interest explicitly

As mentioned in 2. , another parameter of notifyObserver method is used to distinguish the type of changed item.

1. Encapsulating complex update semantics

A StoreItemChangeManager class is introduced to maintain the mapping between StoreItem and its observer. It is a mediator will be responsible for trigger observer to update when receive notify from subject. It implements the Singleton to assure unique and provide globally access.

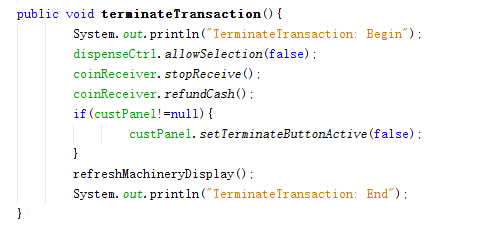
1. Combining the subject and Observer classes

Not suitable. In this case there is no multiple inheritance, make abstract subject and observer combined will expose methods to both ConcreteSubject and ConcreteObserver which is unnecessary and confusing.

## State Pattern

### Description of the design problem

1. For current design, most of controllers are big, especially for the TransactionController. In this situation, it is difficult to modification and maintenance. In another word if a programmer wants to modify TransactionController or something which is associated with it. Nearly, all usages of it need to be retested. It violates the Open for extension, Close for modification Principle.
2. The TransactionController has too many responsibilities. It need to control the transaction process, maintenance process. It violates the Single Responsibility Principle.
3. Code redundancy problem, in some situation, some codes are useless. Here is an example:



This method is used to terminate transaction. If a customer didn’t start a transaction, do the machine need to refund cash? Absolutely not, so the code “coinReceiver.refundCash()” is useless.

### 2.2.2 Candidate design patterns considered

In this case, we need to choose a design pattern, which can:

* Make the TransactionController easily to maintenance and modification.
* Divide the responsibilities.
* Define situations and behaviors.

According to the requirements, the design pattern considered is only State Pattern.

### 2.2.3 Motivation to choose a pattern that would solve the problem including support for new requirement s or changes to existing problems

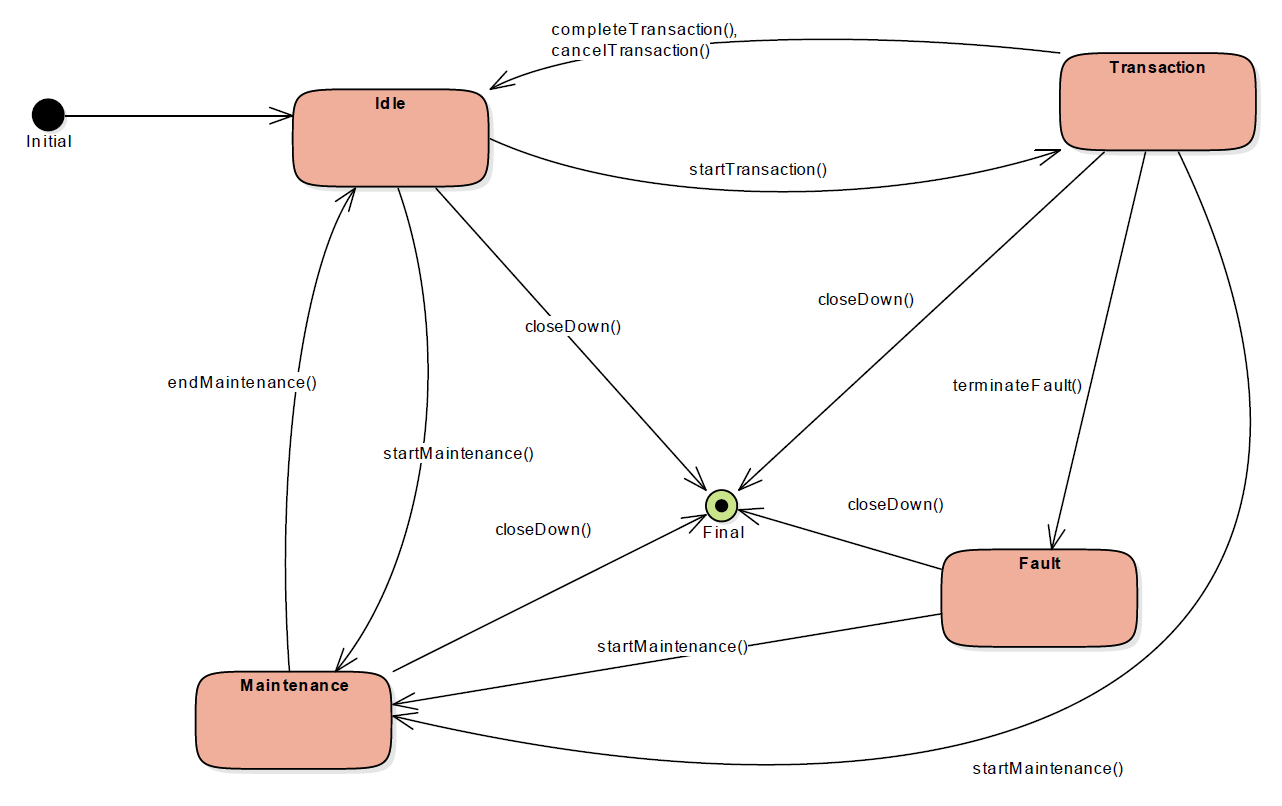
In this case, State Pattern is suitable.

It localizes state-specific behavior and partitions behavior for different states. The State pattern puts all behavior associated with a particular state into one object. Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclass, so the responsibilities have been divided in different states. And the code redundancy problem can also be solved, because in different states, different behaviors can be defined. Using State Pattern, although the number of class and object is increased, it has imposed structure on the code and makes its intent clearer.

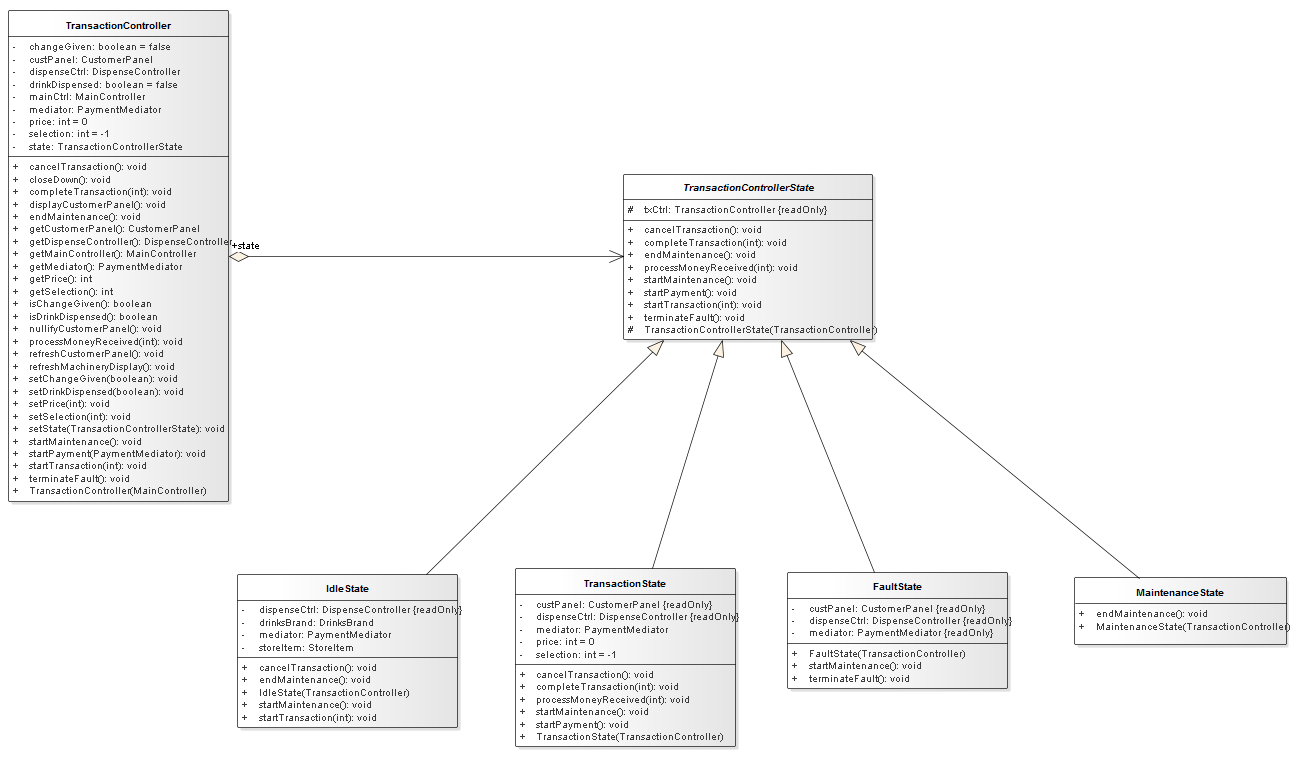
It makes state transitions explicit. Introducing separate object for different states makes the transitions more explicit. In addition, State objects can protect the Context from inconsistent internal states because state transitions are atomic.

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

Statechart Diagra:

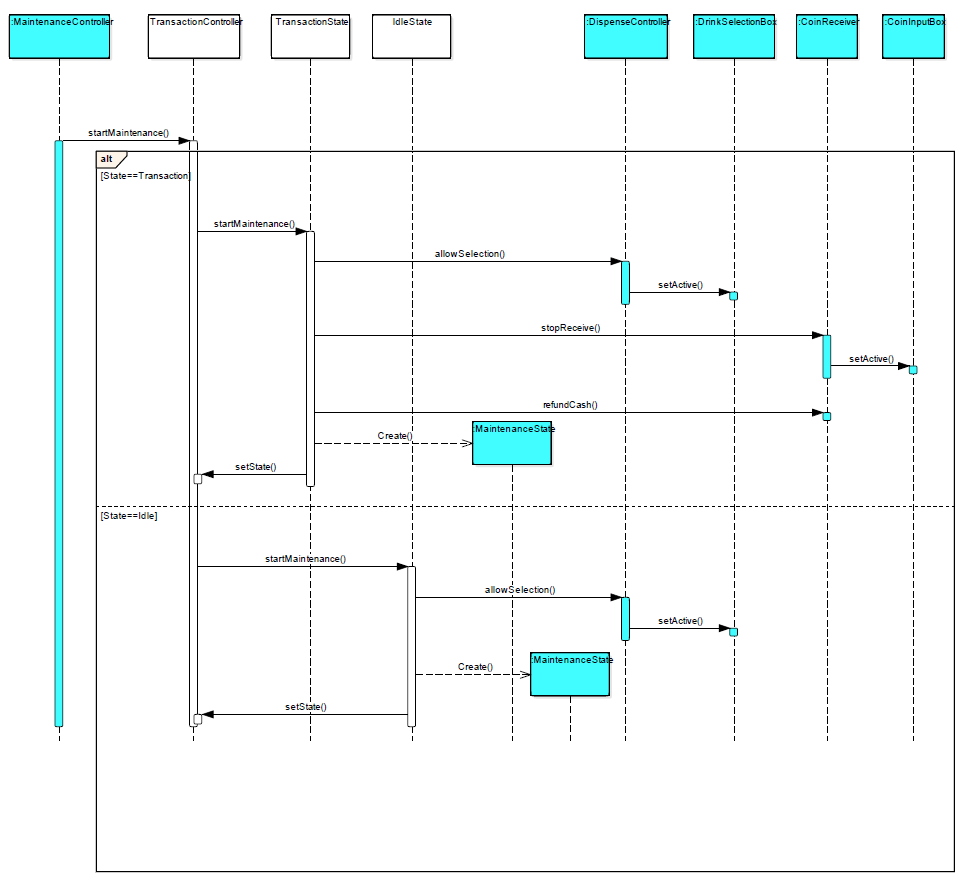


Class Diagram:



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

Sequence Diagram Start Maintenance (Maintainer Logs-in):



### 2.2.6 Implementation decision that you have taken

1. Who defines the state transitions?

In this case, using the State subclasses themselves specify their successor state is better, because in this way it is more flexible and appropriate when to make the transition.

Decentralizing the transition logic in this way makes it easy to modify or extend the logic by defining new State subclasses, but it will cause the dependencies between subclasses.

1. A table based alternation.

Using a table can change the transition criteria by modifying data instead of changing codes, however:

* A table look-up is often less efficient than a virtual function call.
* Putting transition logic into a uniform, tabular format makes the transition criteria less explicit, therefore explicit and therefore harder to understand.
* It is usually difficult to add actions to accompany the state transitions.

The virtual class is considered to implement the State Pattern because in this case it focuses on the state-specific behavior, not defining state transitions.

1. Creating and destroying State objects

Actually, there are two options: (1) to create State objects only when they are needed and destroy them thereafter versus (2) creating them ahead of time and never destroying them.

For option 1, it is preferable when the states that will be entered are not know at run-time and context s change state infrequently.

Option 2 is better when state changes occur rapidly. It can avoid destroying states, which means there are no destruction costs at all.

In this case, Option 1 is accepted. Because for Option 2, the Context must keep references that might be entered.

1. Using dynamic inheritances.

This is not possible in most object-oriented programming language. Exceptions include Self and other delegation-based programming languages. This mechanism lets objects change their behavior and amounts to changing their class.

## Mediator Pattern

### 2.3.1 Description of the design problem

There is tight coupling between TransactionController, CoinReceiver and ChangeGiver class, which means it is hard to maintain and extend. It is about making payment by coins. When developers modify current payment or add new payment, they have to make clear all the processes of the above three classes or add new methods in TransactionController class if they need new payment option so that the TransactionController would be too fat.

In a word, it violates the Open for extension, Close for modification Principle.

### 2.3.2 Candidate design patterns considered

Since the problem is caused by tight coupling between classes, and it is a behavioral issue, the candidate patterns are

* Observer
* Mediator

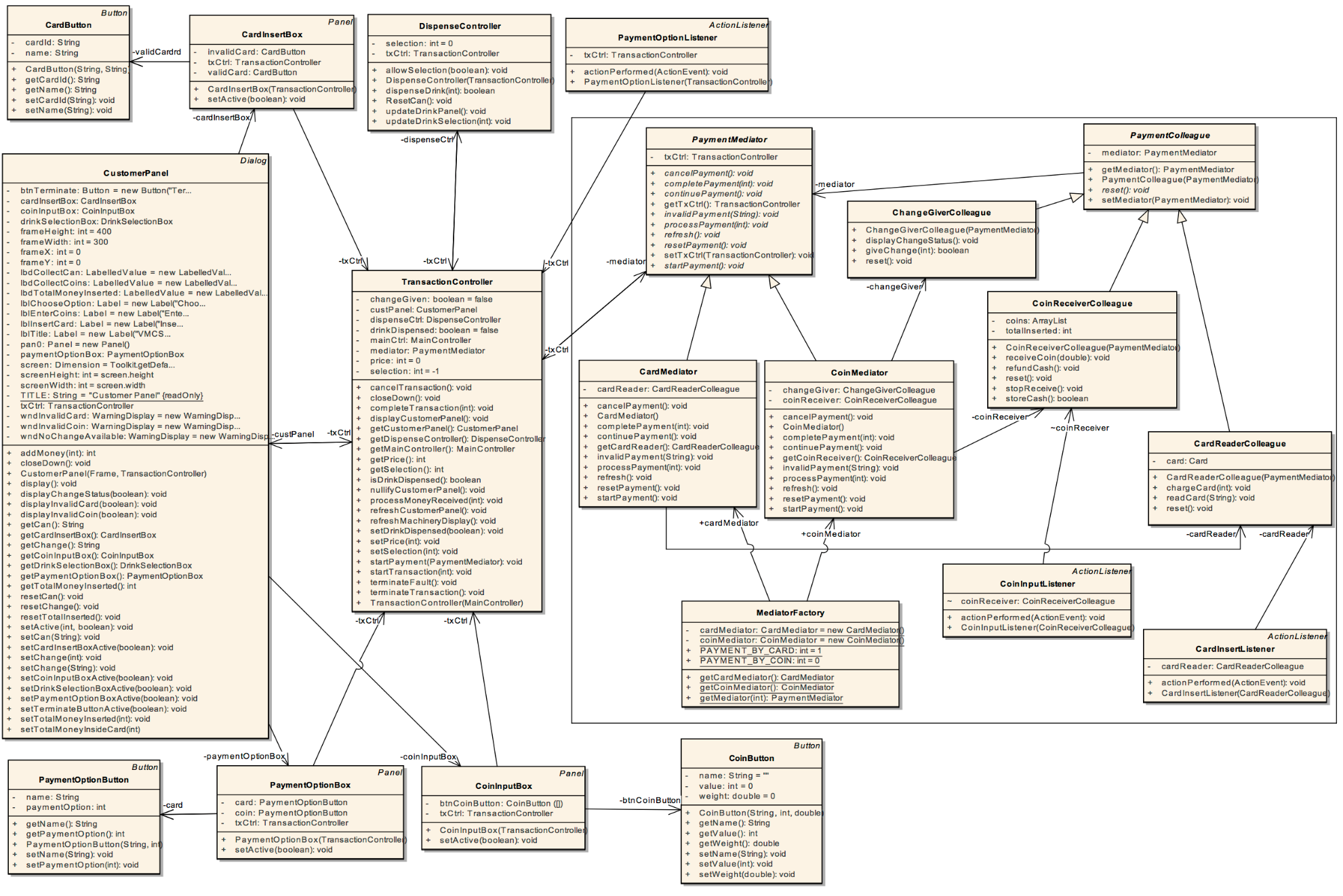
### 2.3.3 Motivation to choose a pattern that would solve the problem including support for new requirements or changes to existing problems

In this case Mediator Pattern is suitable.

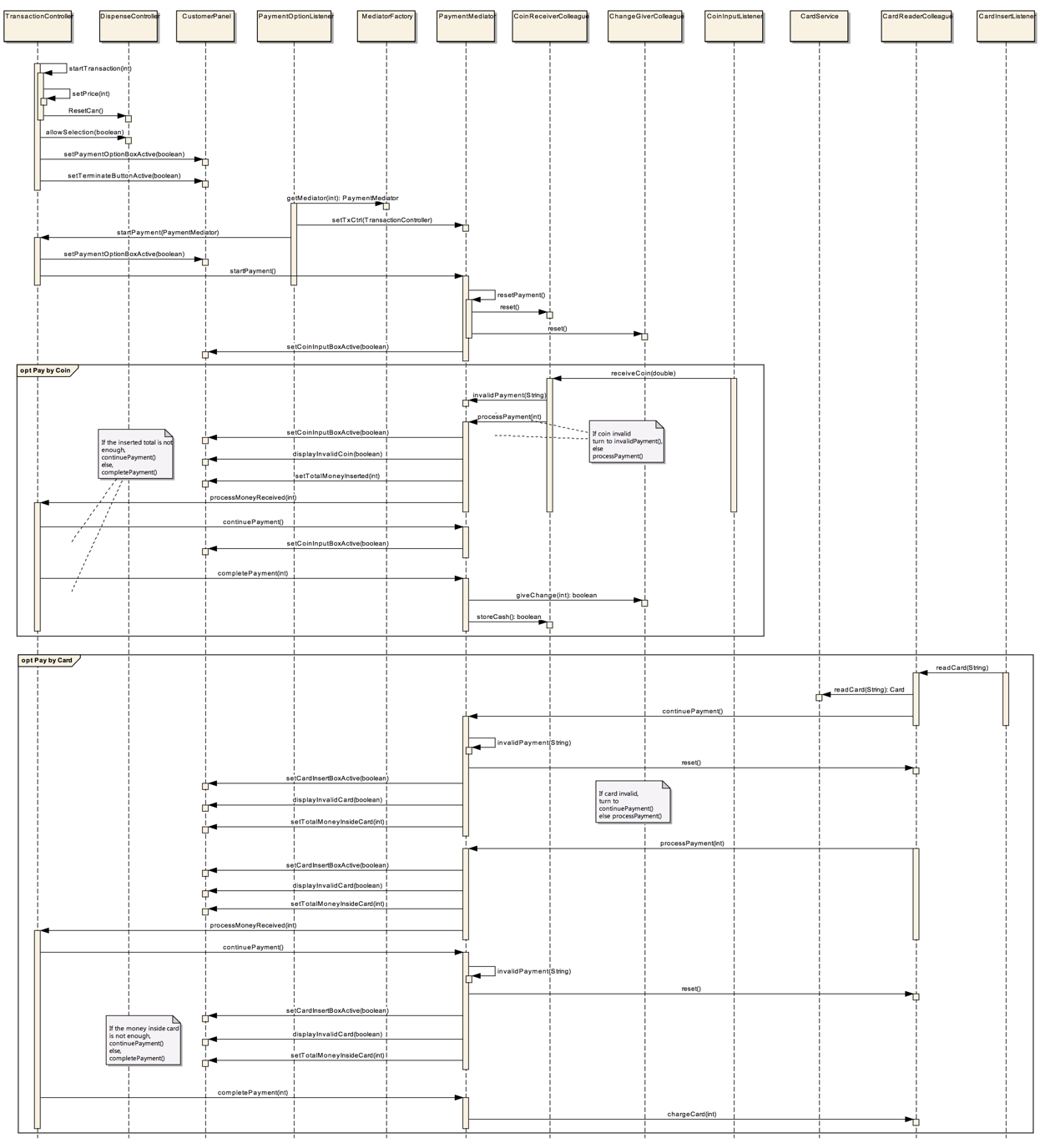
The nature of problem is how to decouple the referred classes so that the TransactionController does not need to change and the develop does not need to know what the TransactionController do when we add new payment option or modify the process of current payment.

Using Mediator Pattern, we only need to focus on what should be done for particular payment option rather than what TransactionController should do and what it needs for different payment options. It becomes easier to modify and extend.

### 2.3.4 Structure of the pattern



### 2.3.5 Collaborations among participants



### 2.3.6 Implementation decision

1. Abstract Mediator

To support multiple payment options, the TransactionController does not need to know what kind of payment option was selected and what the selected option would do actually so that when we add new payment option or modify current payment option we don't need to modify TransactionController. So the abstract mediator is necessary. However, mediator must know TransactionController because the mediator would change involved UI through TransactionController.

2. Encapsulating different complex payment process in different derived mediator classes

In this case, the communication between mediator and colleague is quite complex because choosing different payment option causes different UI active and uses different colleague classes. For example, the CoinMediator will interactive with CoinInputBox and use CoinReceiverColleague and ChangeGiverColleague.

3. Abstract Colleague but each derived Mediator knows its related derived Colleague

Abstract colleague assure all derived colleague classes have the same attribute and same operations and also can have their own different attributes and operations. Because all the colleague classes must know mediator class and have some required methods like reset. They also need some different behaviors like receiving coin for CoinReceiverColleague class and reading card for CardReaderColleague class.

And because of different behaviors for different payment options each derived mediator class should know its related derived colleague classes such as the CoinMediator knows CoinReceiverColleague and ChangeGiverColleague.

4. Factory provides the derived colleague classes

From the view of simulator, the coin receiver and card reader should be always existed and unique. And from the implementation, the coin input button need to know CoinReceiverColleague and the card insert button need the CardReaderColleague but the TransactionController does not have the Mediator and does not know Colleague before client choose payment option. So we use Factory to instantiate a CoinMediator and a CardMediator and to provide with them after client choose the payment option.

## Chain of Responsibility Pattern

### 2.4.1 Description of the design problem

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.

### 2.4.2 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.

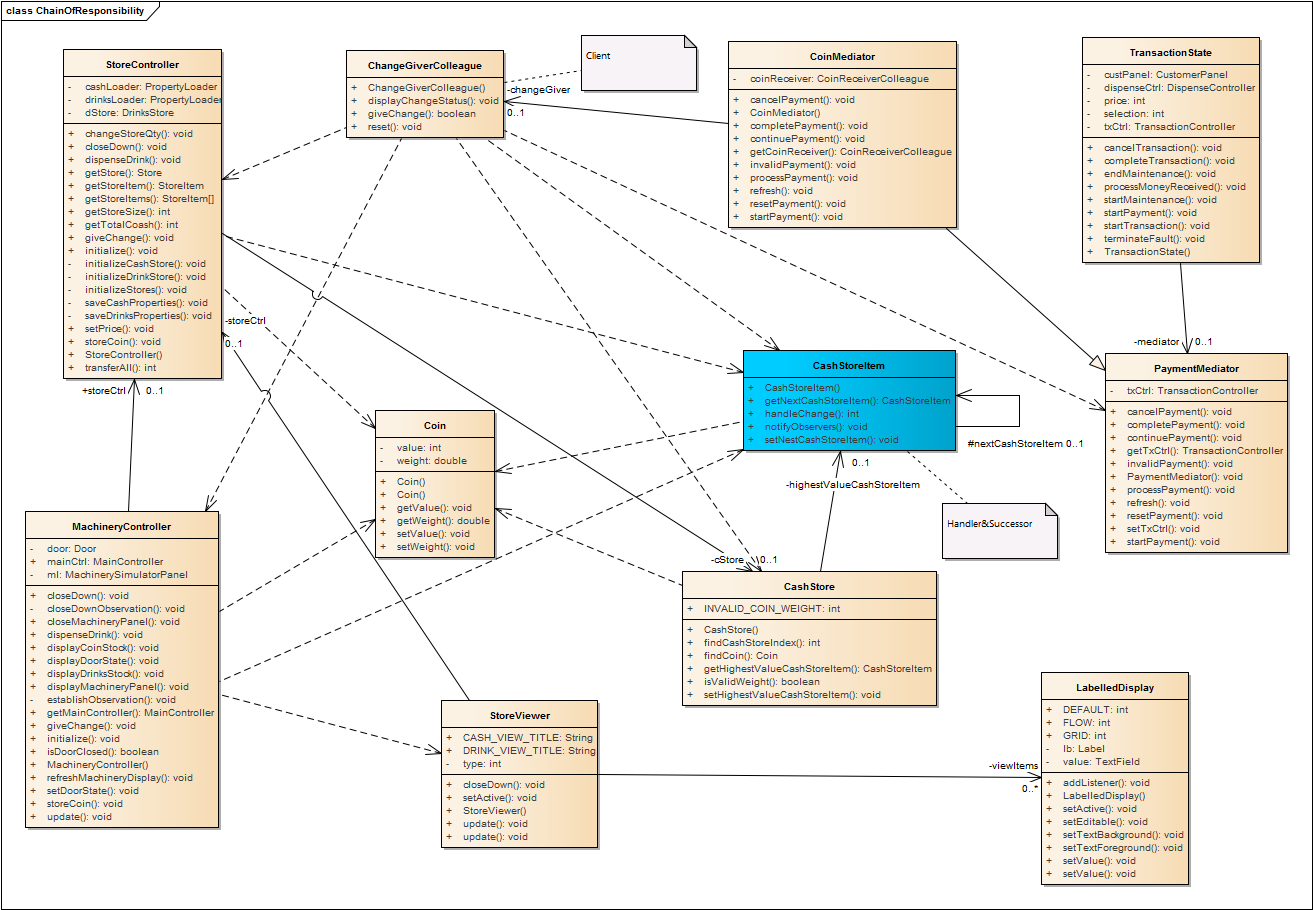
### 2.4.3 Motivation to choose a pattern that would solve the problem including support for new requirements or changes to existing requirements

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.

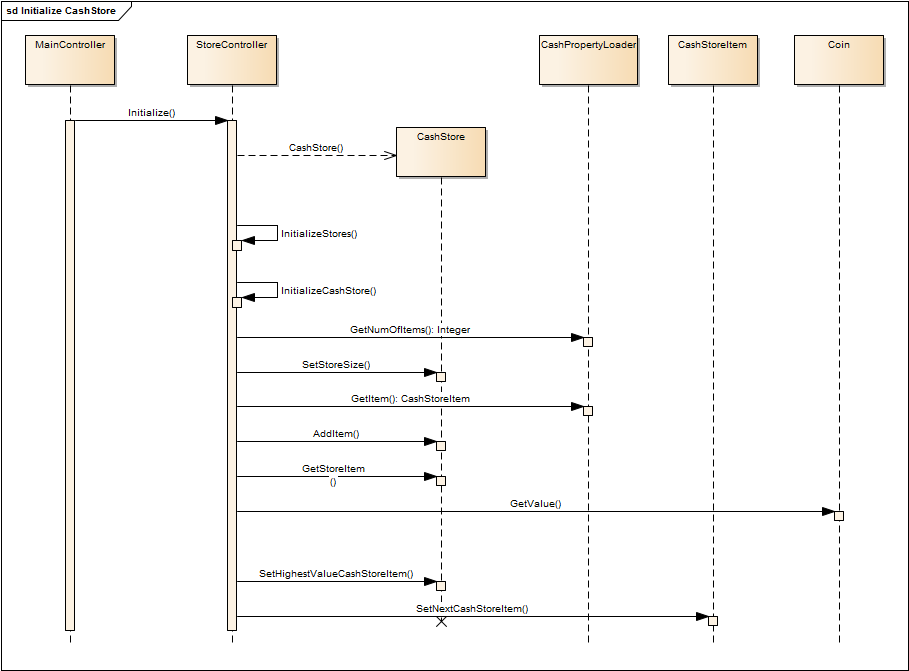
By using this pattern, we de-couple the **ChangeGiver** class. The **ChangeGiver** class now just need know the first **CashStoreItem**(1$), then begin the give changes action.

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

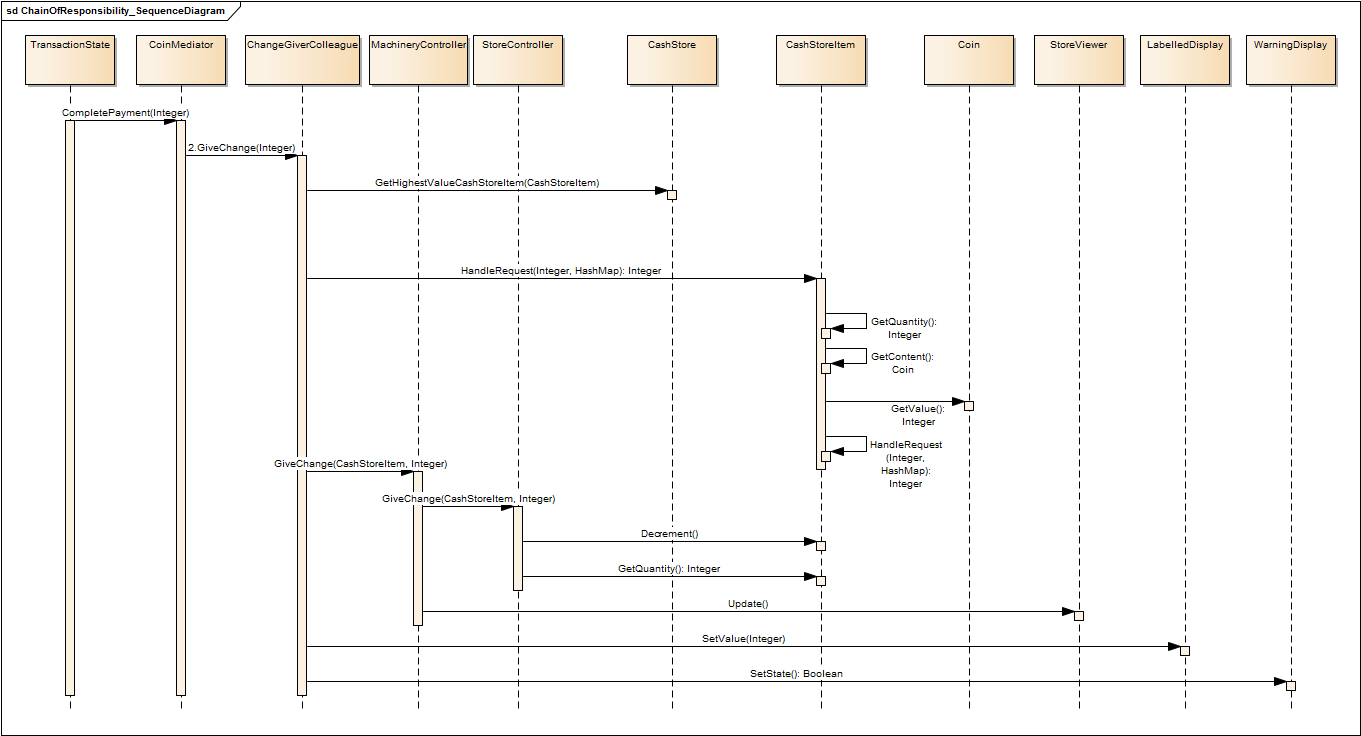


### 2.4.5 Collaborations among the participants (specific to your application objects)

Initialize the cash store:



Give Change:



### 2.4.6 Implementation decisions that you have taken.

1. Implementing the successor chain. There are two possible ways to implement the successor chain:

* Define new links (usually in the Handler, but Concrete Handlers could define them instead).
* Use existing links.

We create a new links in a List of **CashStoreItem**s, because they do not have relationships at the beginning, and we do not want to create 5 more coin classes, so we decided to create links between objects in a list.

2. 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.

3. 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.

4. 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.

5. 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.

# Modified design document

# Team member Contributions

|  |  |  |  |
| --- | --- | --- | --- |
| Problem Number | Design Pattern | Design | Implement |
| 1 | Observer | Xu Minsheng | Xu Minsheng |
| 2 | State | Liu Xinzhuo | Liu Xinzhuo |
| 3 | Mediator | Xie Jiabao | XieJiabao |
| 4 | Chain of responsibility | Lu Angxiao | Lu Angxiao |