Mediator Pattern

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.

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

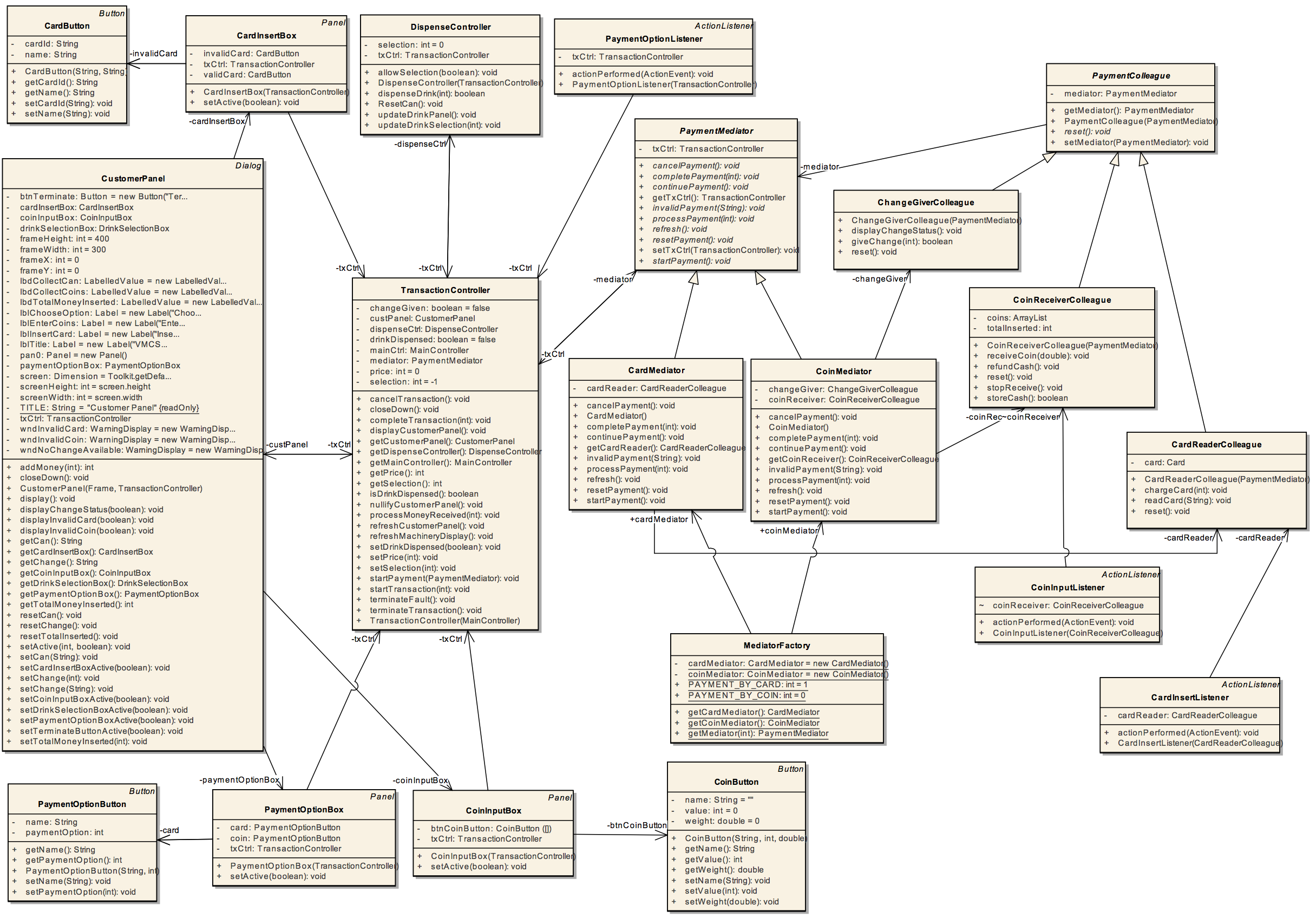
Motivation to choose a pattern that would solve the problem including support for new requirements or changes to existing problems

In this case Mediator Pattern is suitable.

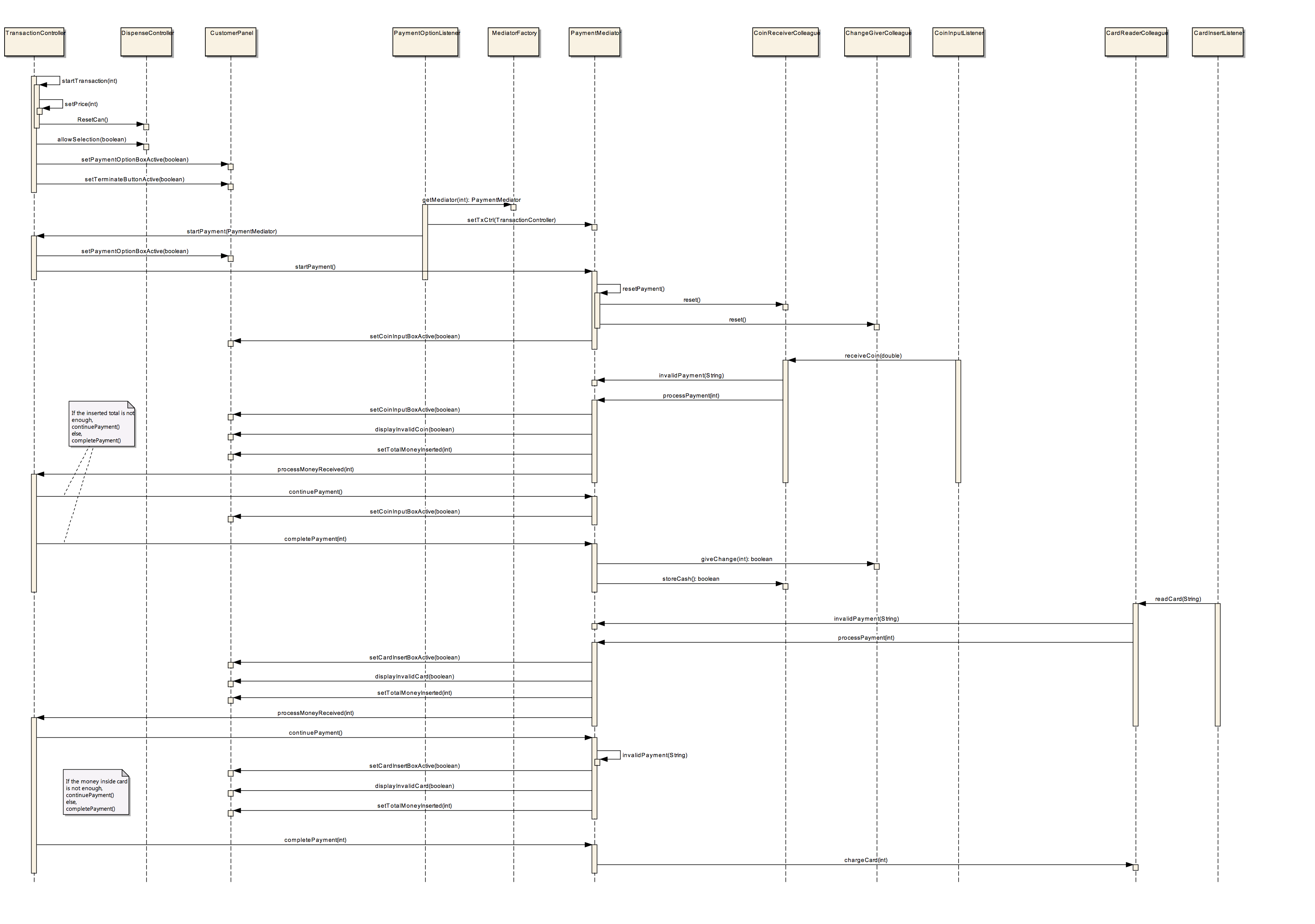
The nature of problem is how to decouple the referred classes so that the TransactionController does not need to change and the develop does not need to know what the TransactionController do when we add new payment option or modify the process of current payment.

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.

Structure of the pattern



Collaborations among participants



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.