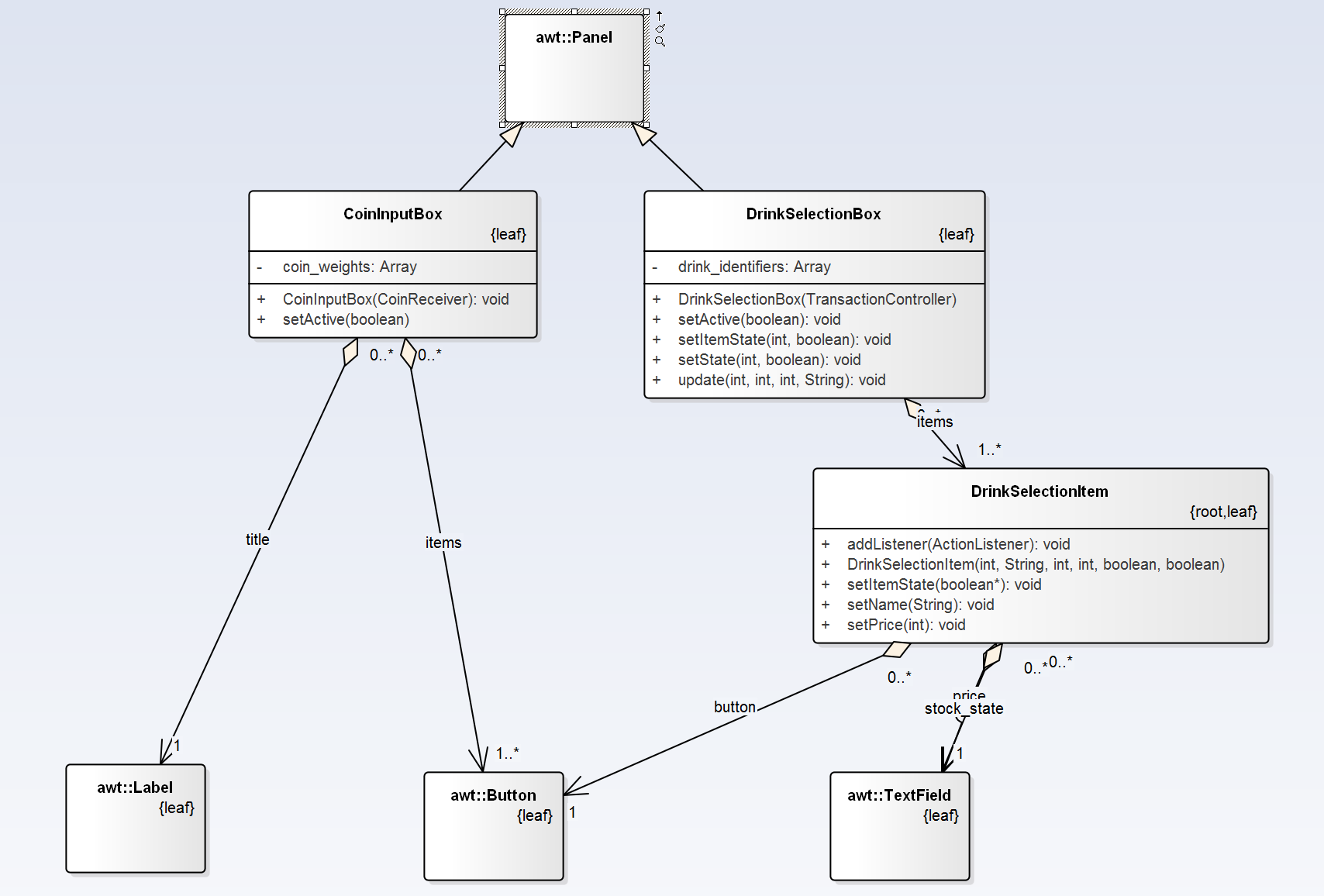
# 1. Design Problem:

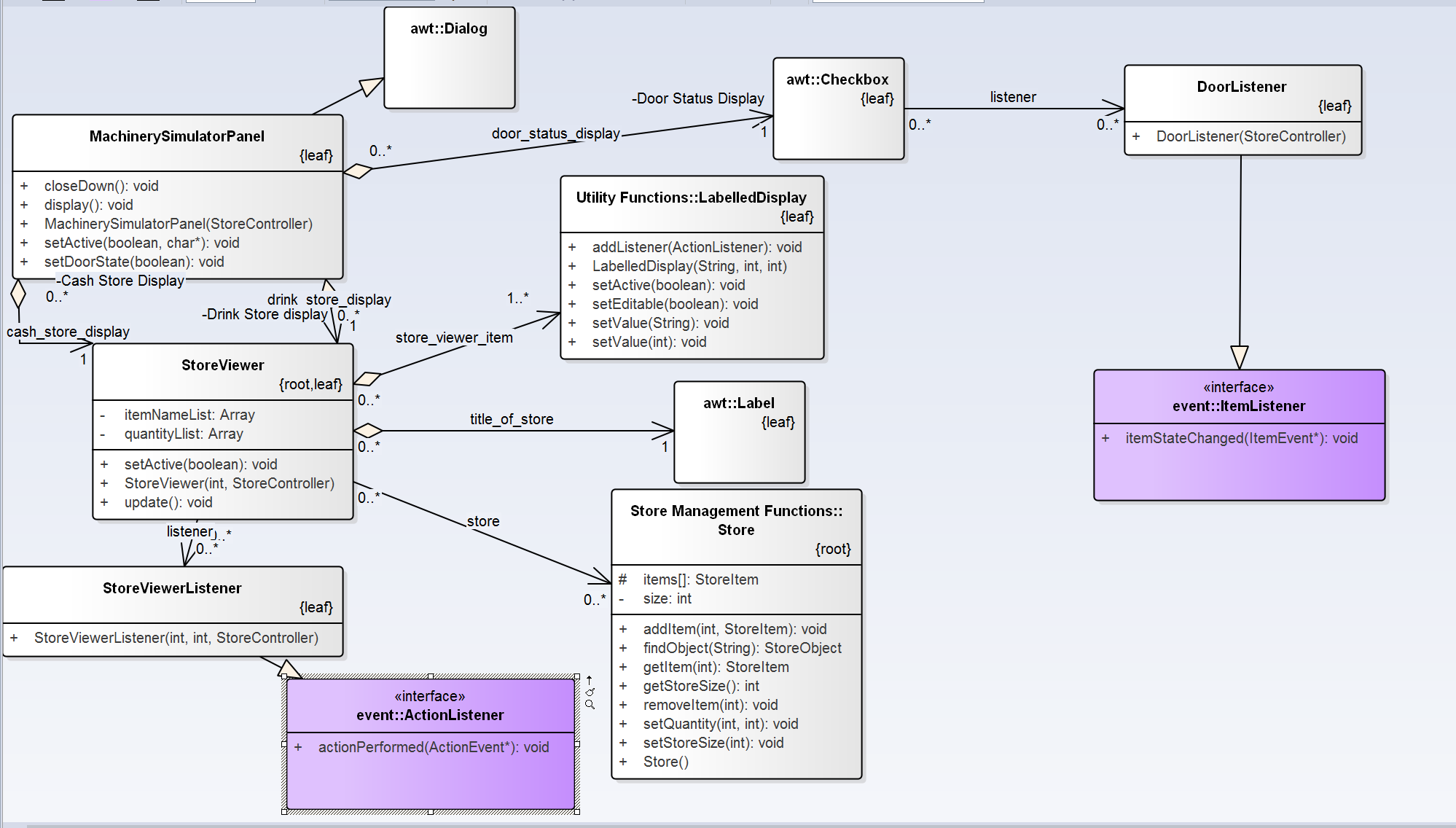
In this design GUI components need to know which operation in the respective controller to invoke. In here every listener is initializing controllers. Here client and service rendering receiver objects are coupled together. For example: According the current design DrinkSelectionListener class initializing Transaction Controller and its invoking StartTransaction() method and in the StoreViewerListener initialize Store Controller and it calls ChangeStoreQty(). SO this design contains tight coupling between UI and business Sub systems and UI need to know respective business components and adding new features is increase the coupling and complexity.

# 2. Class Diagrams (Before Applying Command Pattern)

## 2.1 Drink Selection

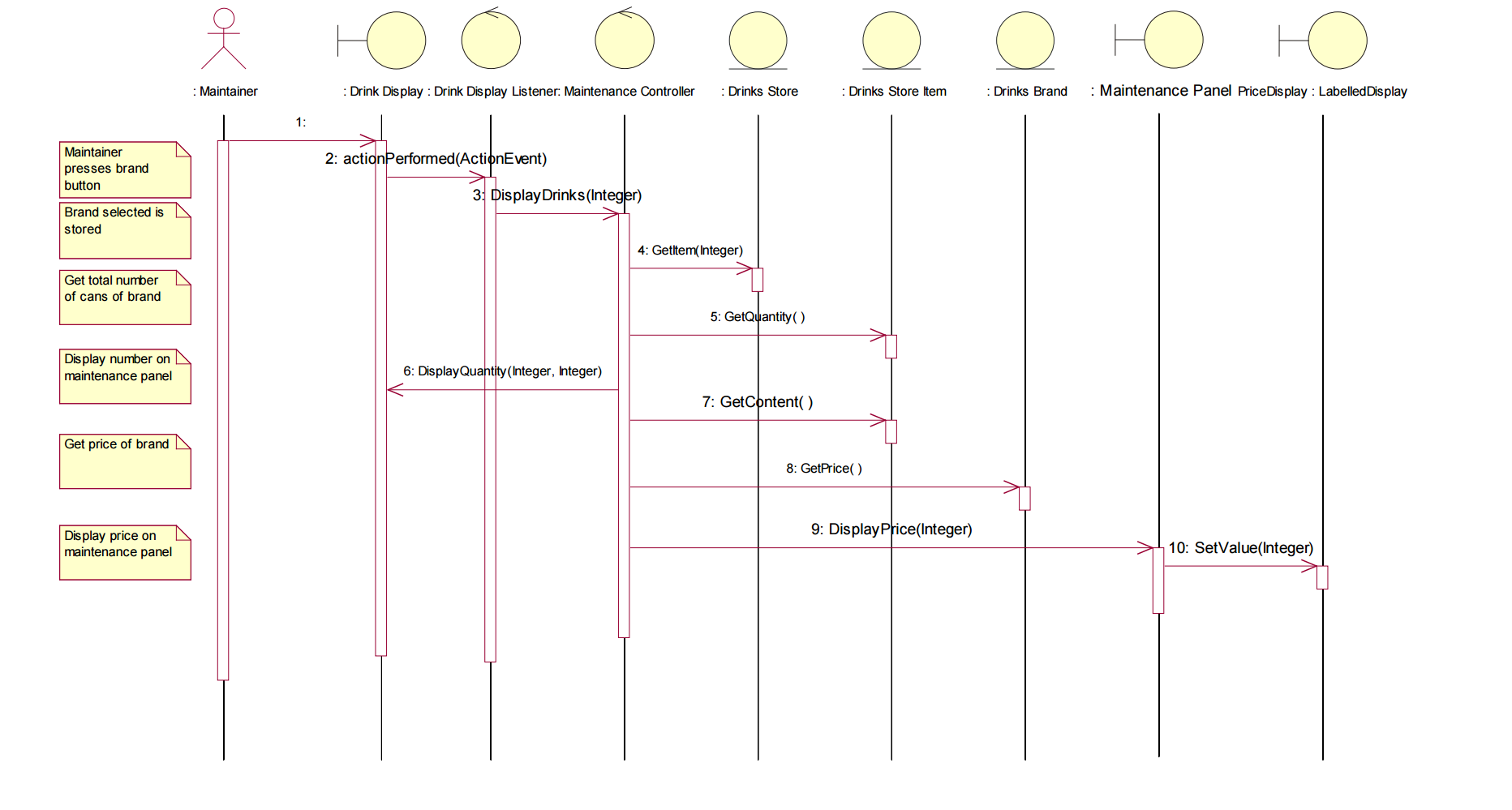


## 2.2 Change Cash Store

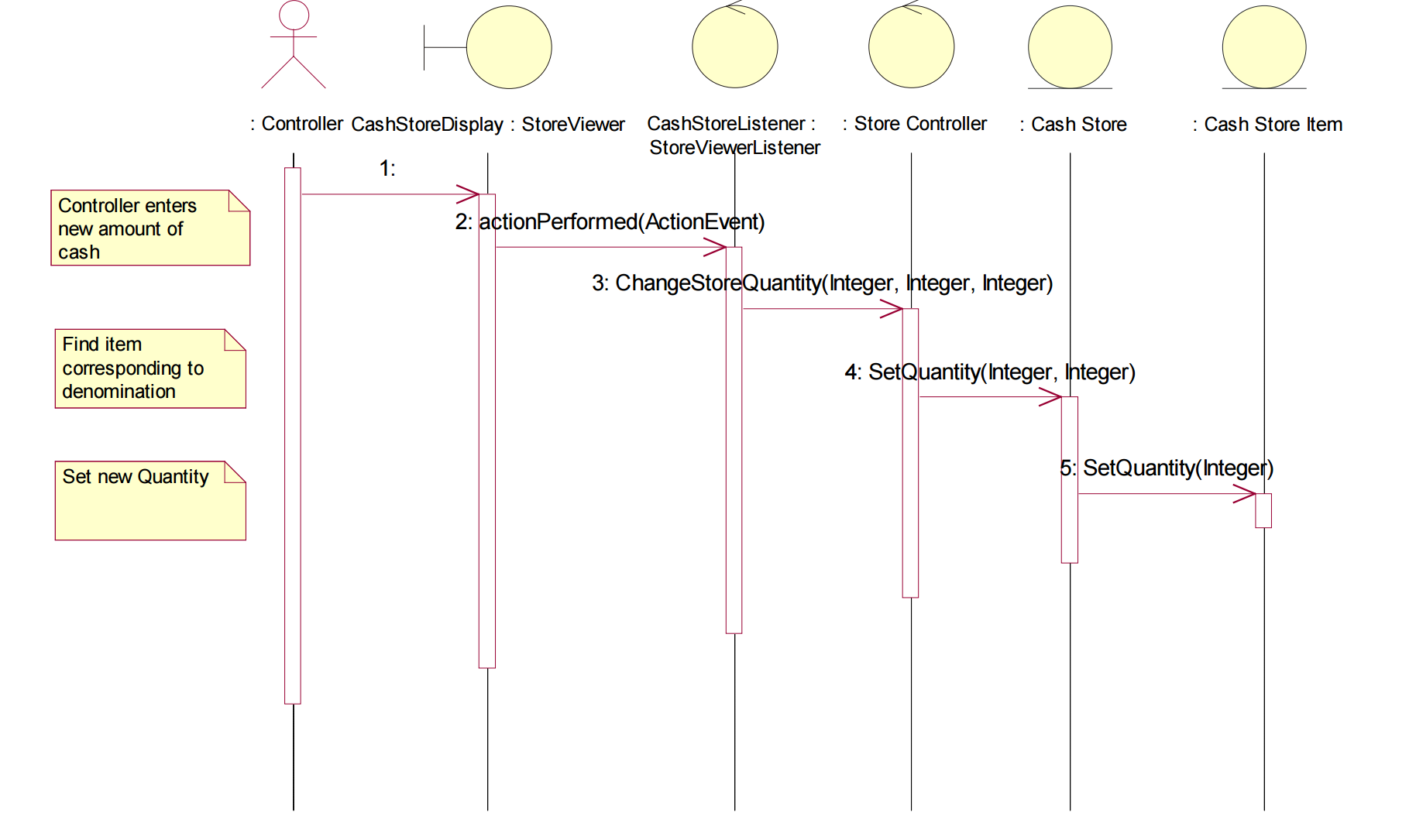


# 3. Sequence Diagram (Before Applying Command Pattern)

## 3.1 Drink Selection



## 3.2 Change Cash Store



# 4. Candidate Design Patterns:

1. Command:

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queues or log requests and support undoable operations.

1. Mediator:

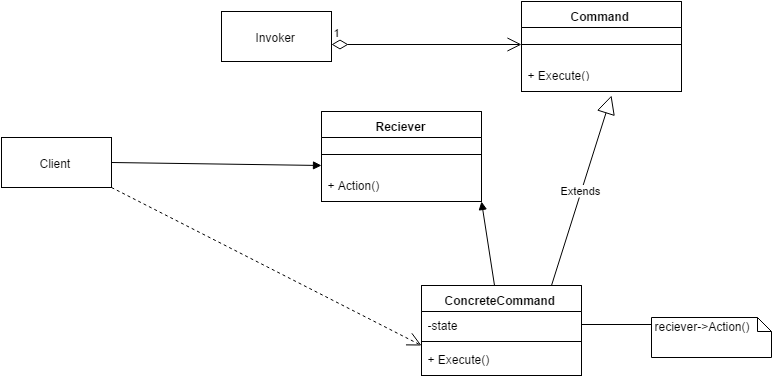
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interactions independently.

Choosed Pattern: Command Pattern

# 5. Motivation to choose Command Pattern:

Using command design pattern we can decouple the objects that invokes the operation from the one that knows how to perform it and it’s easy to add new features without changing existing class much and adding new commands require minimal and manageable changes. Since we need one way communication not multiple way of communication no need to go for Mediator pattern.

# 6. Structure of command Pattern



**Participants in the command pattern implementation:**

**Command:** Command Interface acting as command in the implementation

* Declares an interface for executing an operation.

**Concrete Command:** DrinkSelectionCommand concrete Implementation class and ChangeStoreQtyCommand concrete implementation class are acting as concrete commands

* Defines a binding between a Receiver object and an action.
* Implements Execute by invoking the corresponding operation(s) on Receiver.

**Client:** DrinkSelectionBox class and Store Viewer class acting as Client

* Creates a Concrete Command object and sets its receiver.

**Invoker:** Invoker class has been acting as Invoker and this class been implemented as singleton.

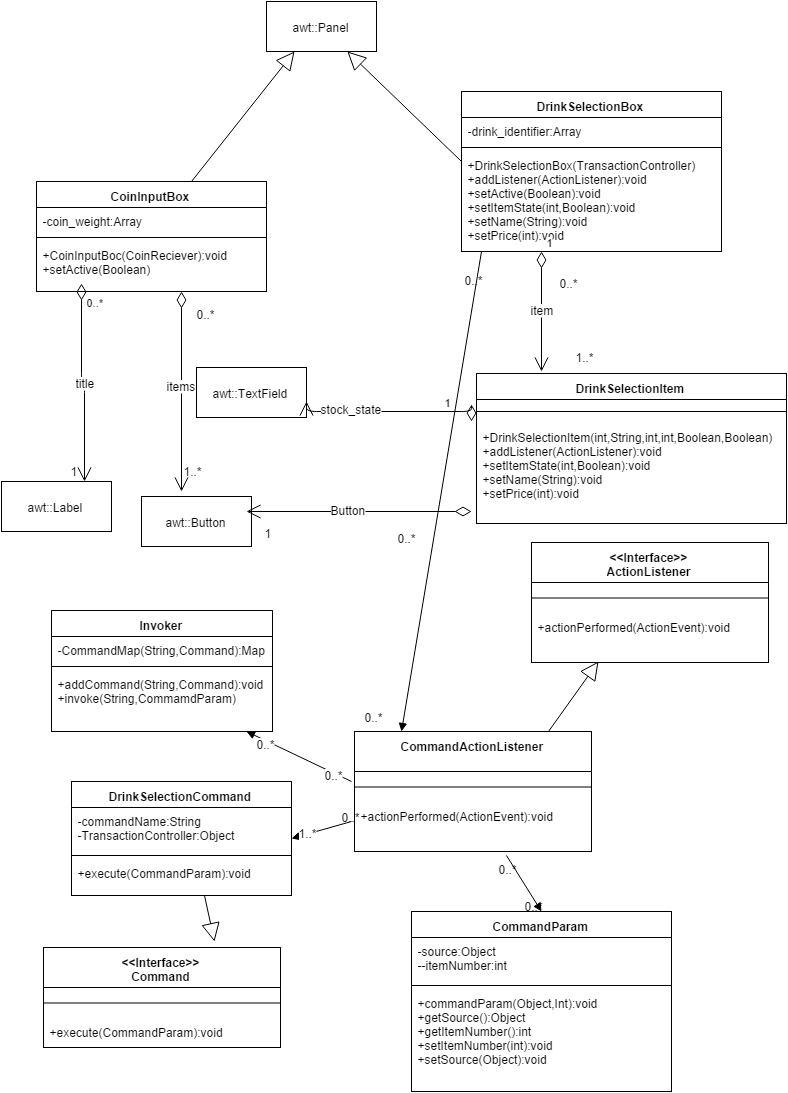
* Asks the command to carry out the request.

**Receiver:** StoreController class has been acting as Receiver.

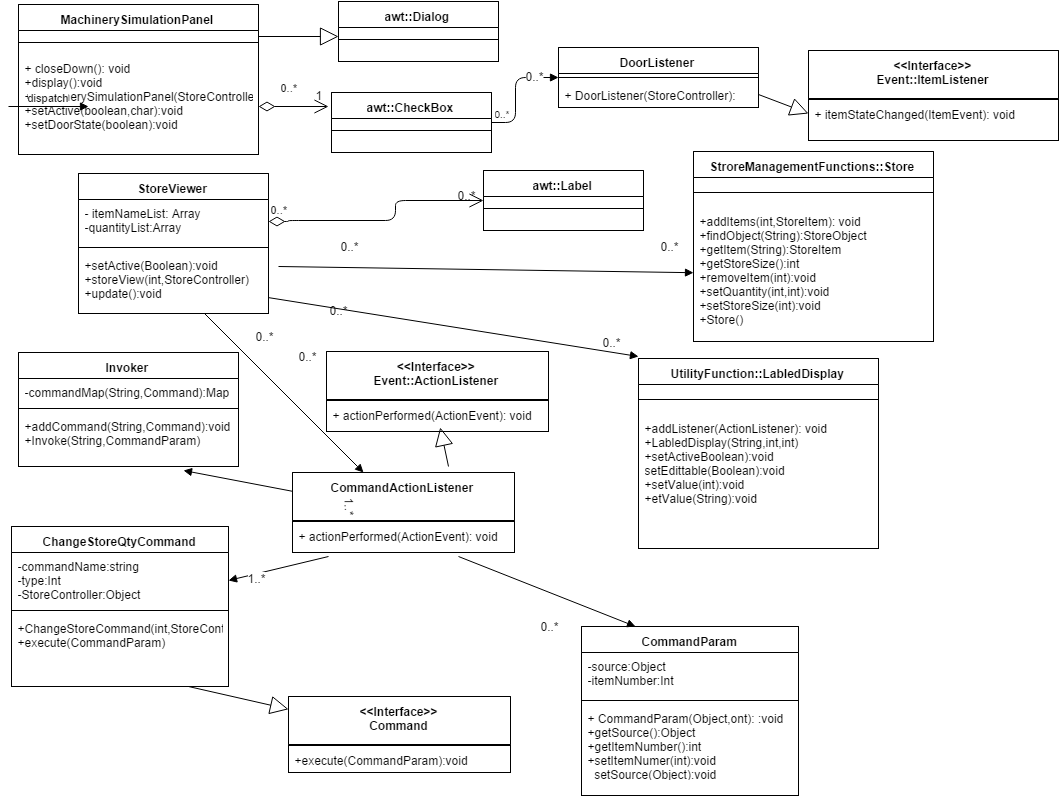
* Knows how to perform the operations associated with carrying out a request

# 7. Class Diagram (After Applying Command Pattern)

## 7.1 Drink Selection

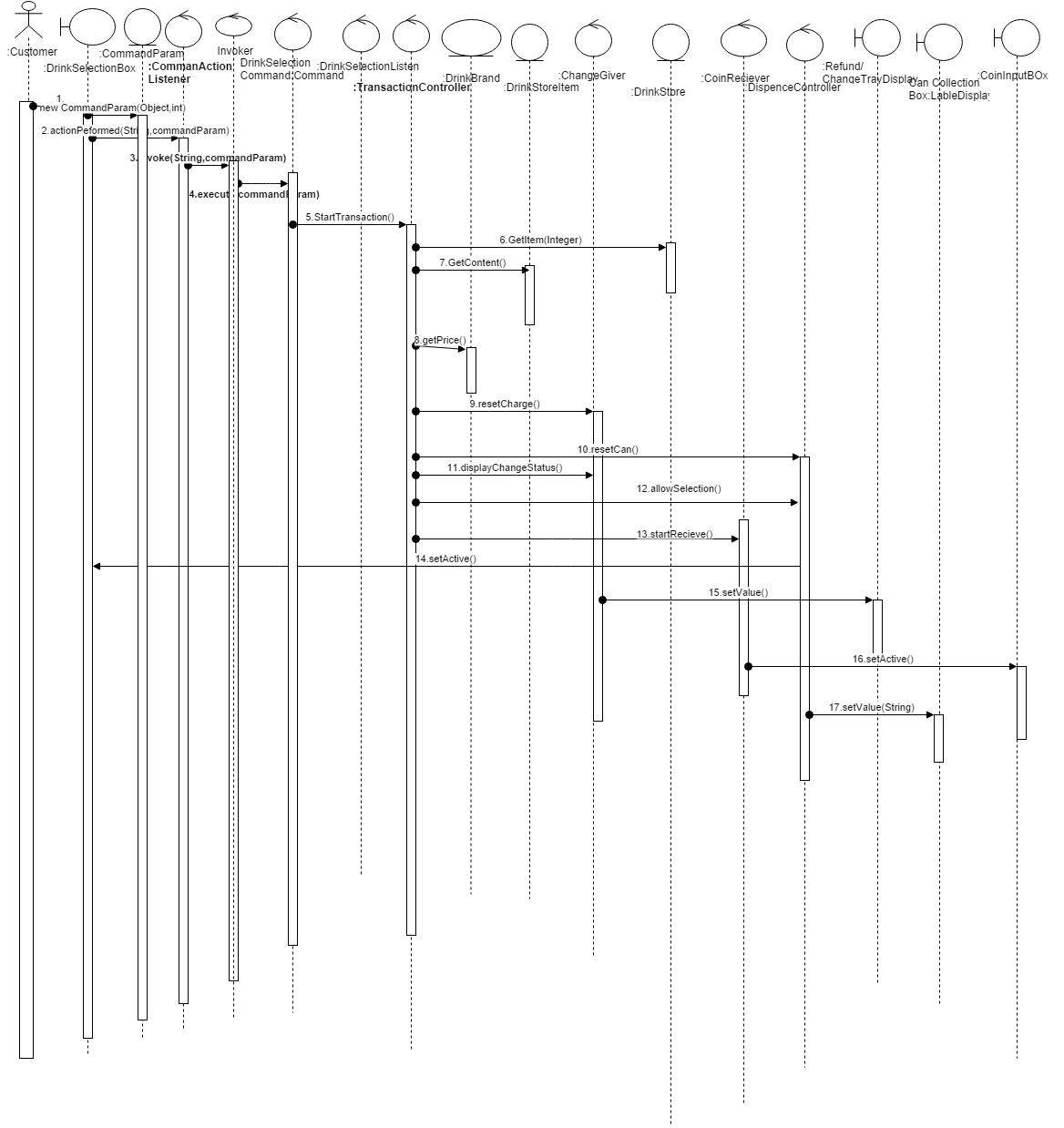


## 7.2 Change Cash Store

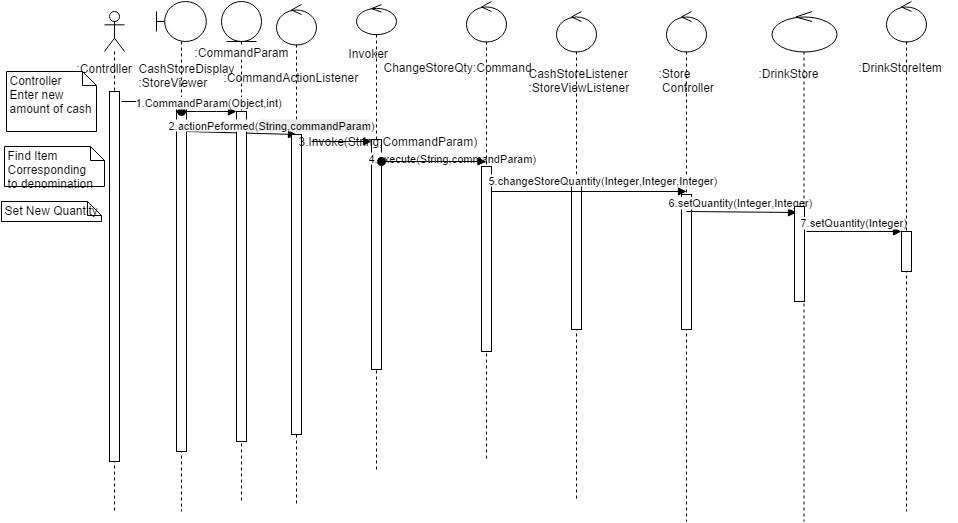


# 8. Sequence Diagram (After applying Command Pattern)

## 8.1 Drink Selection



## 8.2 Change Cash Store



# 9. Implementation Decisions:

1. How Intelligent should a command be?

It defines binding between the receiver (Business Component) and UI system by parameterizing the action that carryout the request. Command implementation classes chose the method to invoke on receiver object. It works as a bridge between receiver and action methods.

1. Supporting Undo and Redo?

Undo – Not supported

1. Avoiding Error Accumulation in the undo process

Undo – Not supported

1. Using C++ template.

Command require arguments. So creating command subclass for every kind of actions and receiver required.