SG5208:

Object Oriented Design Patterns – Team PT01

# VMCS redesign

**Participants (Team PT01):**

METRANI NIKHIL DILIP – A0135935

BARLA MARUTHI DATTATREYA – A0120530

JING DONG – A0135901

SITHIVINAYAGAM KOKULAKUMARAN – A0120490

SIVANANTHAMOORTHY JANAHAN – A0120591

Table of Contents

[VMCS redesign 1](#_Toc433557727)

[1. Simplification of complex user interface mechanisms (by Metrani Nikhil Dilip – A0135935) 5](#_Toc433557728)

[**1.1 Design Problem** 5](#_Toc433557729)

[**1.2 Design Pattern Selection** 6](#_Toc433557730)

[1.2.1 Candidate design patterns 6](#_Toc433557731)

[1.2.1 Motivation to choose Observer pattern 7](#_Toc433557732)

[**1.3 Design changes** 8](#_Toc433557733)

[1.3.1 Initial design 8](#_Toc433557734)

[1.3.2 Modified design 12](#_Toc433557735)

[**1.5 Implementation** 16](#_Toc433557737)

[1.5.1 Participants 16](#_Toc433557738)

[1.5.2 Implementation considerations 16](#_Toc433557739)

[2. Multiple Store controller instance causes inconsistent of store data (by BARLA MARUTHI DATTATREYA – A0120530) 17](#_Toc433557740)

[**2.1 Design Problem** 17](#_Toc433557741)

[**2.2 Pattern Selection** 17](#_Toc433557742)

[2.2.1 Candidate Design Patterns 17](#_Toc433557743)

[2.2.2 Motivations to Choose 17](#_Toc433557744)

[**2.3 Design Changes** 18](#_Toc433557745)

[2.3.1 Initial Design 18](#_Toc433557746)

[2.3.2 Modified Design 20](#_Toc433557747)

[**2.4 Implementation** 22](#_Toc433557748)

[3. Unexpected error causes inconsistency of data (by BARLA MARUTHI DATTATREYA – A0120530) 23](#_Toc433557749)

[**3.1 Design Problem** 23](#_Toc433557750)

[**3.2 Pattern Selection** 23](#_Toc433557751)

[3.2.1 Candidate Design Patterns 23](#_Toc433557752)

[3.2.2 Motivations to Choose 23](#_Toc433557753)

[**3.3 Design Changes** 24](#_Toc433557754)

[3.3.1 Initial Design 24](#_Toc433557755)

[3.3.2 Modified Design 26](#_Toc433557756)

[**3.4 Implementation** 28](#_Toc433557757)

[4. Create a general purpose GUI component (by JING DONG – A0135901) 29](#_Toc433557758)

[**4.1 Design Problem** 29](#_Toc433557759)

[**4.2 Candidate design patterns** 29](#_Toc433557760)

[**Selected pattern:** 29](#_Toc433557761)

[**4.2 Motivation** 30](#_Toc433557762)

[**4.3 Class diagram** 31](#_Toc433557763)

[**4.4 Sequence Diagram** 32](#_Toc433557764)

[**4.5 Implementation** 33](#_Toc433557765)

[4.5.1 External vs. internal iterators. 33](#_Toc433557766)

[4.5.2 We use External Iterators in the Implementation 34](#_Toc433557767)

[5. Bridge Pattern Design (by SITHIVINAYAGAM KOKULAKUMARAN - A0120490U) 37](#_Toc433557768)

[**5.1 Design Problem** 37](#_Toc433557769)

[**5.2 Candidate Patterns** 39](#_Toc433557770)

[**5.3 Motivation to select bridge pattern** 39](#_Toc433557771)

[**5.4 Class Diagram after applying the design pattern** 40](#_Toc433557772)

[**5.5 Sequence Diagrams** 41](#_Toc433557773)

[**5.5 Implementation Decisions** 43](#_Toc433557774)

[6. Command Pattern (by SIVANANTHAMOORTHY JANAHAN – A0120591) 44](#_Toc433557775)

[**6.1 Design Problem** 44](#_Toc433557776)

[**6.2. Class Diagrams (Before Applying Command Pattern)** 44](#_Toc433557777)

[6.2.1 Drink Selection 44](#_Toc433557778)

[6.2.2 Change Cash Store 45](#_Toc433557779)

[**6.3. Sequence Diagram (Before Applying Command Pattern)** 46](#_Toc433557780)

[6.3.1 Drink Selection 46](#_Toc433557781)

[6.3.2 Change Cash Store 47](#_Toc433557782)

[**6.4. Candidate Design Patterns** 47](#_Toc433557783)

[**6.5. Motivation to choose Command Pattern** 47](#_Toc433557784)

[**6.6. Structure of command Pattern** 48](#_Toc433557785)

[**6.7. Class Diagram (After Applying Command Pattern)** 49](#_Toc433557786)

[6.7.1 Drink Selection 49](#_Toc433557787)

[6.7.2 Change Cash Store 50](#_Toc433557788)

[**6.8. Sequence Diagram (After applying Command Pattern)** 51](#_Toc433557789)

[6.8.1 Drink Selection 51](#_Toc433557790)

[6.8.2 Change Cash Store 52](#_Toc433557791)

[**6.9. Implementation Decisions** 52](#_Toc433557792)

# 1. Simplification of complex user interface mechanisms (by Metrani Nikhil Dilip – A0135935)

## **1.1 Design Problem**

The VMCS design, is targeted for three types of users, namely customer, controller, and maintainer. The design achieves to satisfy its targeted users with the help of rich user interface panels. Customer accesses the system via Customer Panel, controller makes use of the Machinery Panel, and maintainer uses the Maintenance Panel.

All three user interface panels display the name, quantity, and/or price of all store items. Store items currently consist if various type of drinks and various denominations of coins.

Consider following use cases –

1. Maintainer:
   1. Updates price of any drink
   2. Transfer all cash from the store
2. Controller:
   1. Update quantity of any denomination of coins
   2. Update quantity of any drinks

All these above activities require to display updated quantities on both Maintenance and Machinery panels. Which means that changes done in maintenance panel must be visible in machinery panel and vice-versa.

With the current design, the update mechanism requires that the Maintenance and Machinery panel know each other and are able to notify whenever there is any change in store items. In other words, they are tightly coupled.

The panels/controllers also makes use of update and refresh mechanisms for the complete panel instead of only updating the changed item. This complicates things future if any requirement changes like, introduction of new store item, say snacks, or addition of a new user interface panel, are to be implemented.

## **1.2 Design Pattern Selection**

### 1.2.1 Candidate design patterns

1. Command PatternIntent:
   1. Encapsulate a request as an object.
   2. Parameterize clients with different requests, queue or log requests.
   3. Support undoable operations.
2. Mediator PatternIntent:
3. Define an object that encapsulates how a set of objects interact.
4. Promote loose coupling by keeping objects from referring to each other explicitly.
5. Vary the interaction independently.

1. Observer PatternIntent:
   1. Define one-to-many dependency between objects.
   2. Notify all dependents when object state changes.

**Selected pattern:** *Observer*

### 1.2.1 Motivation to choose Observer pattern

There are some panels sharing the quantity of coins and drinks, so if the changes take place in the quantity of coins and drinks, those panels / displays are needed to be informed. The important of the consistent of viewing the data has become obvious. The observer pattern helps the views to be notified when a change occurs in the data object. So there is a change in the quantity of coins or drinks, any panels which are attached with the data will be notified. In the observer pattern, the key objects are subject and observer. In VMCS, the coin item and drink item which contain the information of the quantity will become Subject. So any panels / displays which are interested on those items will become observer. One clear advantage is that there are no modifications on the objects when new panel is introduced or changes in the view. Those store items will send out the notifications without having to know who its observers.

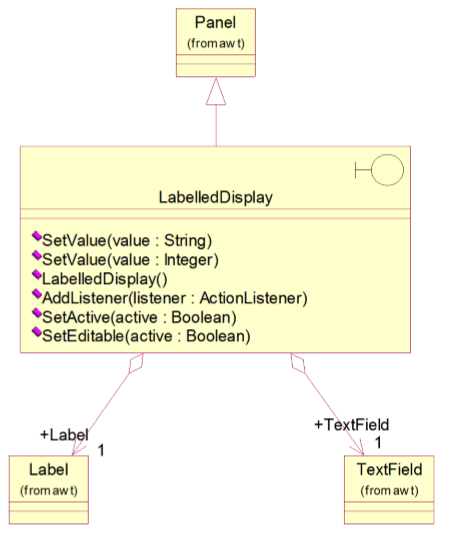
## 

## **1.3 Design changes**

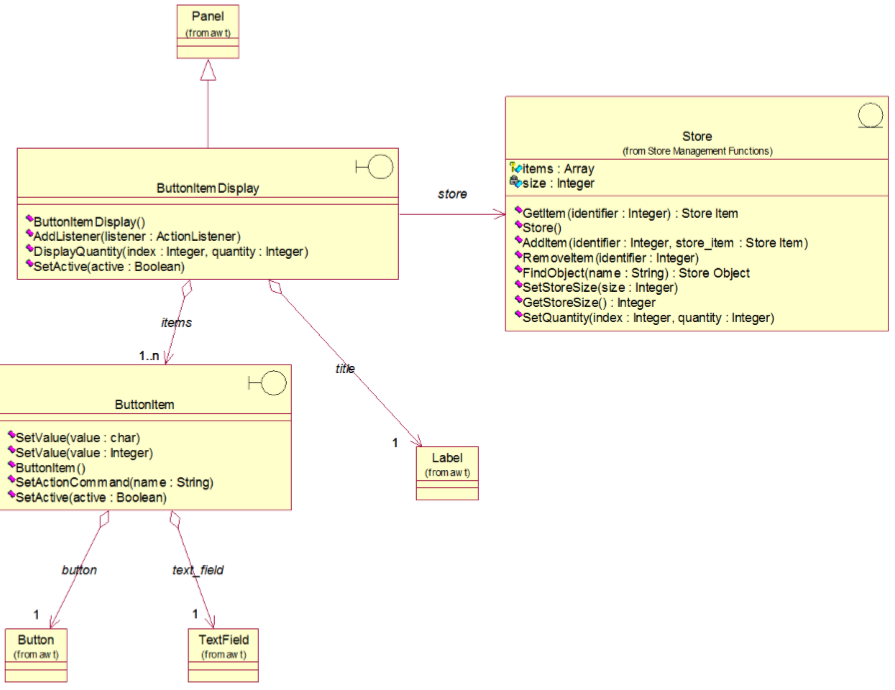
### 1.3.1 Initial design

a. Class diagrams

Labelled Display:

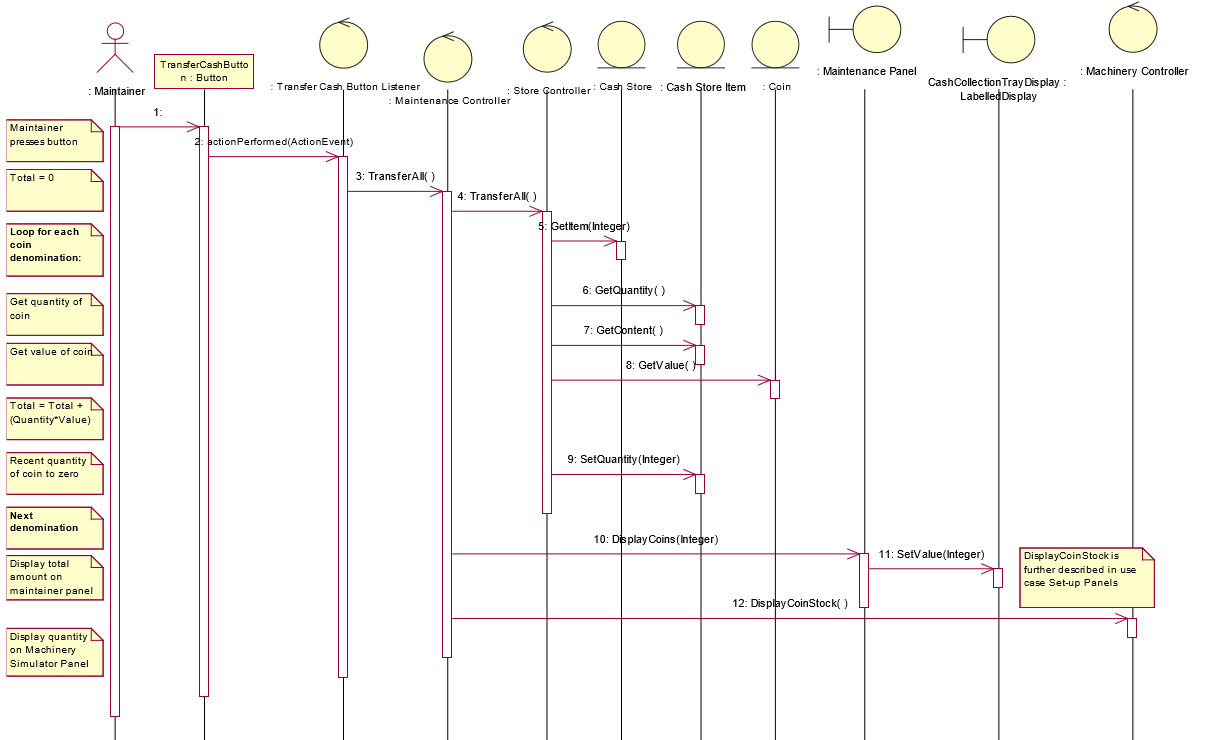


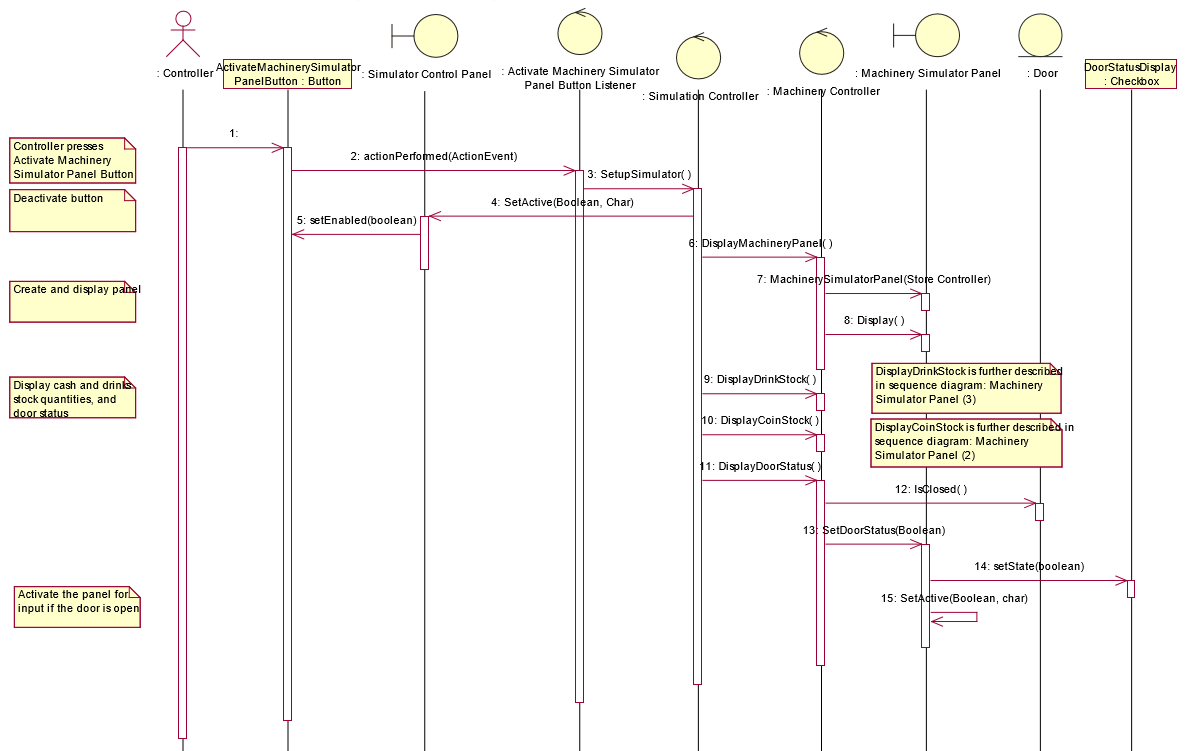
ButtonItemDisplay and ButtonItem:



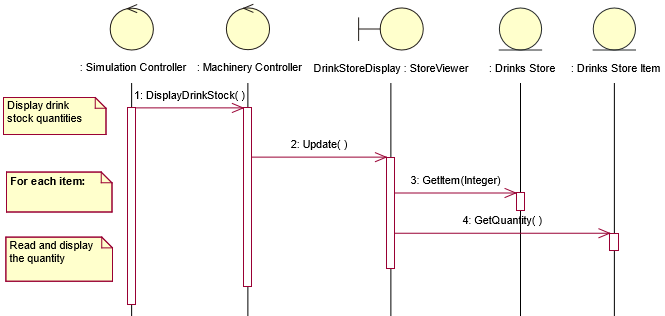
b. Sequence diagrams

Transfer All Cash

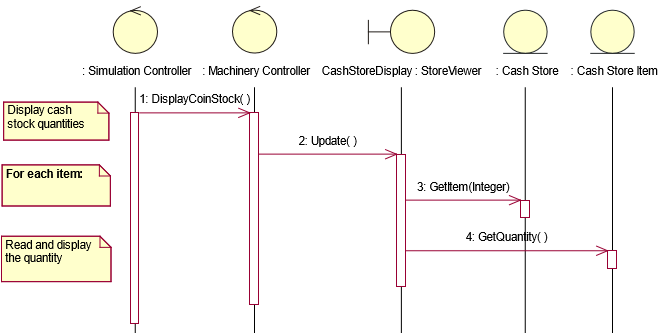


Display Machinery Simulation panel 1

Machinery Simulation Panel 2



Machinery Simulation Panel 3



### 1.3.2 Modified design

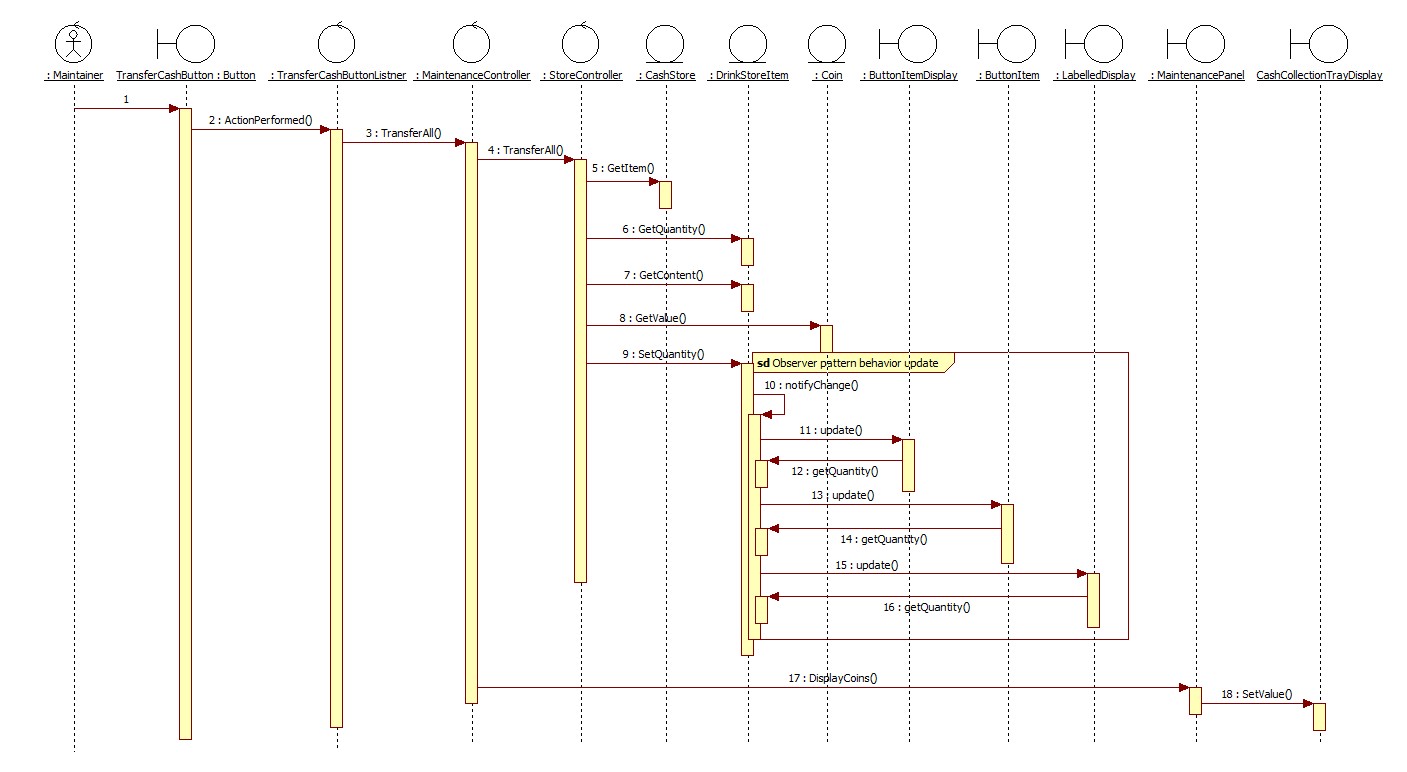
a. Class diagrams

Combined class diagram

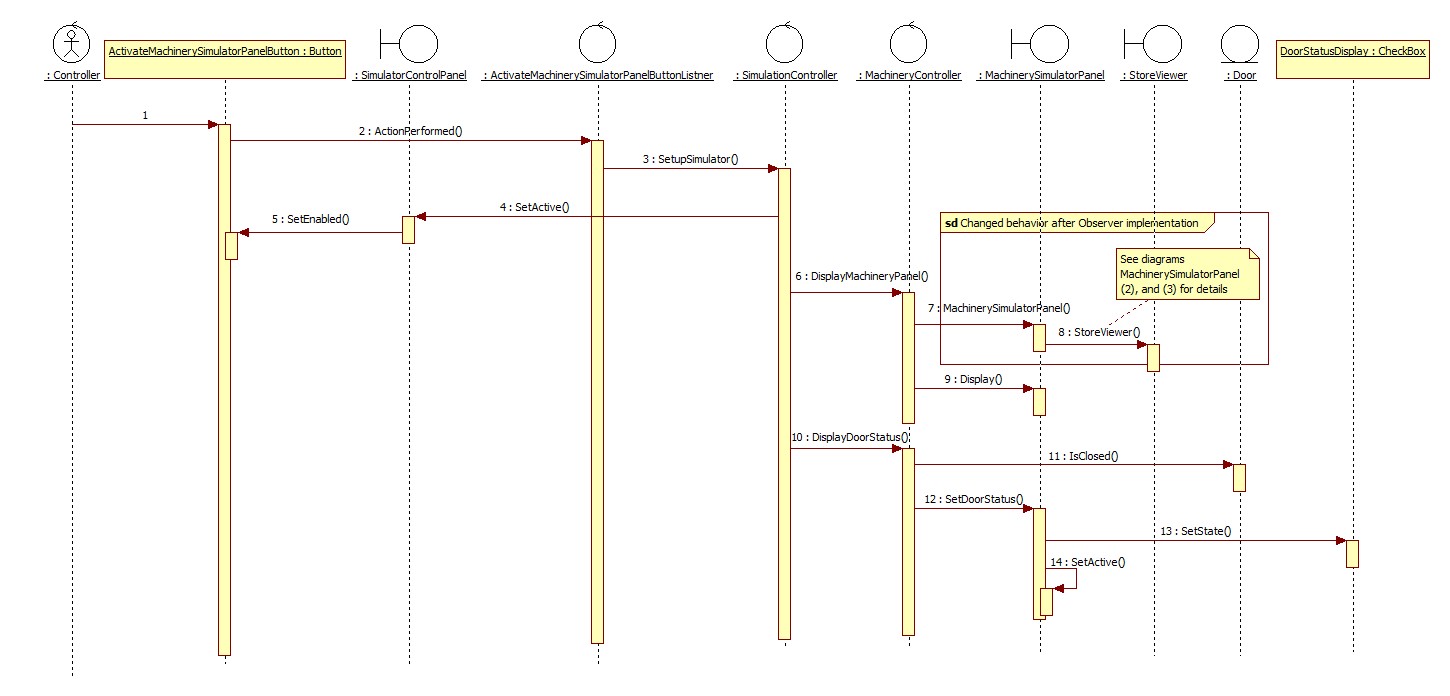
## D:\develop\BitBucket\vmcs_oodp\VMCS\documents\Main.jpg

b. Sequence diagrams

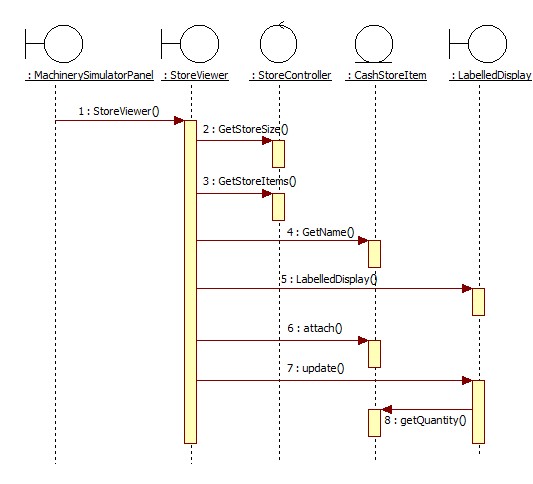
Transfer all cash



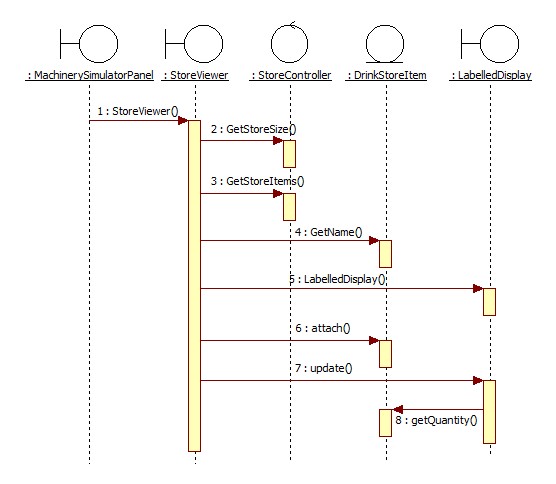
Machinery Simulation Panel 1



Machinery Simulator Panel 2:



Machinery Simulator Panel 3:



## **1.5 Implementation**

### 1.5.1 Participants

* Subject

- Subject - abstract class

- Knows its Observers. Any number of observers can observe the subject.

- Provides a mechanism to attach and detach observers and to notify changes.

* Concrete Subject

- StoreItem (CashStoreItem, and DrinkStoreItem inherit this role)

- Stores state of interest to ConcreteObserver objects.

- Notifies observers when state changes.

* Observer

- Observer - interface

- Defines an updating interface for objects that should be notified of changes in a subject.

* Concrete Observer

- LabelledDisplay, ButtonItem, and ButtonItemDisplay

- Maintains a consistent reference to a ConcreteSubjects state.

- Implements the Observer updating interface to keep its state consistent with the Subject’s.

### 1.5.2 Implementation considerations

**Mapping of Subjects to their Observers:** By making LabelledDisplay, ButtonItem, and ButtonItemDisplay the concrete observer, VMCS can be freed of the complex methods required to refresh the display. Example – DisplayDrinkStock, and DisplayCoinStock. The StoreViewer attached appropriate LabelledDisplay to necessary StoreItem, thus handing over the responsibility of update.

**Who triggers the update:** The StoreItem, who is the concrete subject, triggers the notifyChange() method whenever there is a change in quantity. This can be seen in property accessor – setQuantity(in quantity) and methods increment(), decrement(), and store().

# 2. Multiple Store controller instance causes inconsistent of store data (by BARLA MARUTHI DATTATREYA – A0120530)

## **2.1 Design Problem**

In existing design, the Store Controller is used for retrieval and update of Drink store and Coin store. Currently, Store Controller is only created in Main Controller and always accessible from Main Controller. If some other class create Store Controller by itself, then more than one store controllers exists in the system, which will cause inconsistent of Drink store and coin store item information.

## **2.2 Pattern Selection**

### 2.2.1 Candidate Design Patterns

* Singleton
  + Ensure a class only has one instance and provide a global point of access to it.
* Facade
  + Provide a unified interface to a set of interfaces in a subsystem.
  + Façade defines a higher-level interface that makes the subsystem easier to use.

## 2.2.2 Motivations to Choose

Singleton pattern is selected to solve this problem.

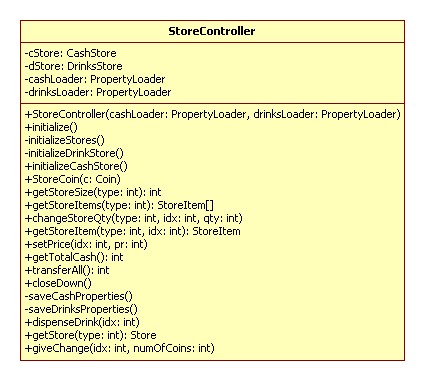
Reasons for choosing:

* Drink Store Item and Coin Store Item are shared global resources and should be controlled by only one instance class of Store Controller. Having single instance provides consistency of data in the system.

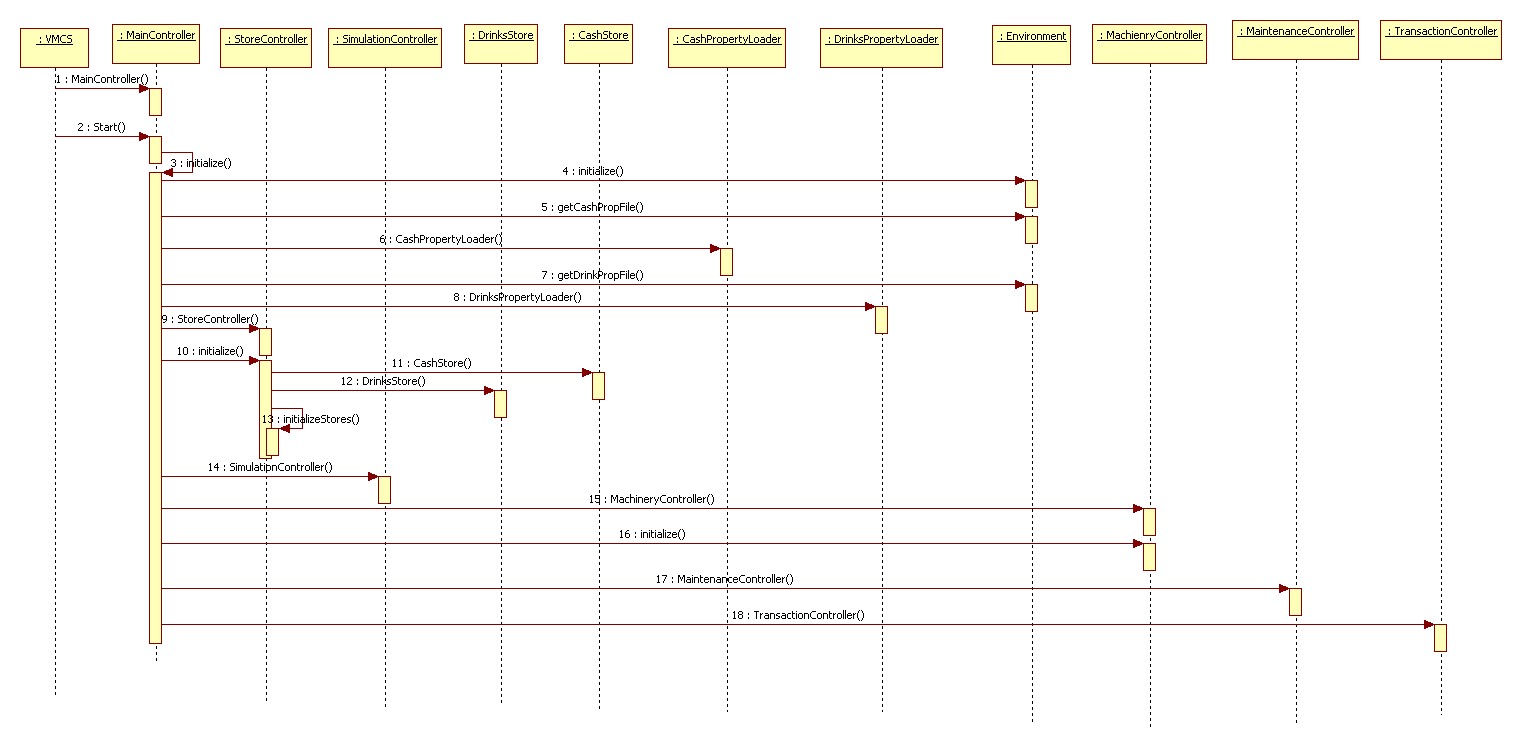
## **2.3 Design Changes**

### 2.3.1 Initial Design

Store Controller Class Diagram

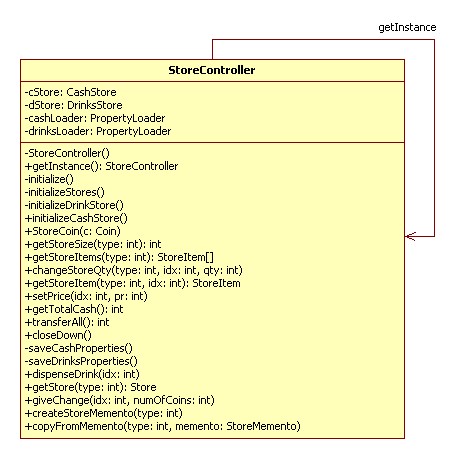


Main Controller initiation Sequence Diagram

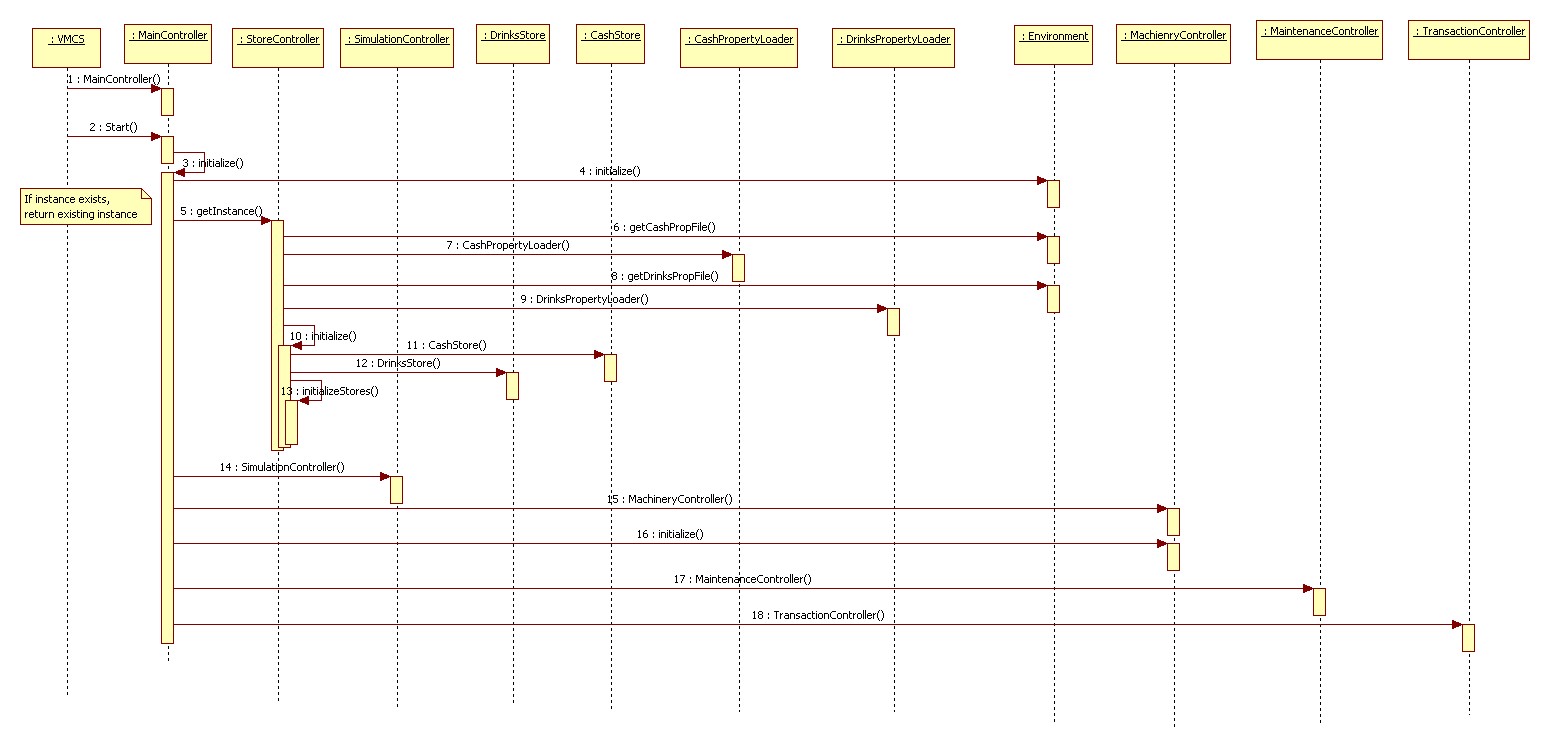


### 2.3.2 Modified Design

Store Controller Class Diagram



Main Controller initiation Sequence Diagram



## **2.4 Implementation**

* Ensuring a unique instance
  + Changed public constructor to private constructor in Store Controller class and introduced getInstance public static method to return the global instance.
  + Static field is used for holding store controller instance. If instance is not created earlier then only new instance is creating otherwise existing created returning to the caller.

# 3. Unexpected error causes inconsistency of data (by BARLA MARUTHI DATTATREYA – A0120530)

## **3.1 Design Problem**

Current design doesn’t have proper rollback mechanism of data. if any unexpected error happens after updating store item information, then store data will be inconsistent state. We might want to revert the changes if any unexpected exception. Example in case of Give change scenario, if any exception occur need to restore the cash store to it previous state.

## **3.2 Pattern Selection**

### 3.2.1 Candidate Design Patterns

* State
  + Allow object to alternate its internal behavior when it’s internal state changes.
* Memento
  + Allow object to record an object internal state without violating encapsulation and reclaim it later.

### 3.2.2 Motivations to Choose

Memento pattern is selected to solve this problem.

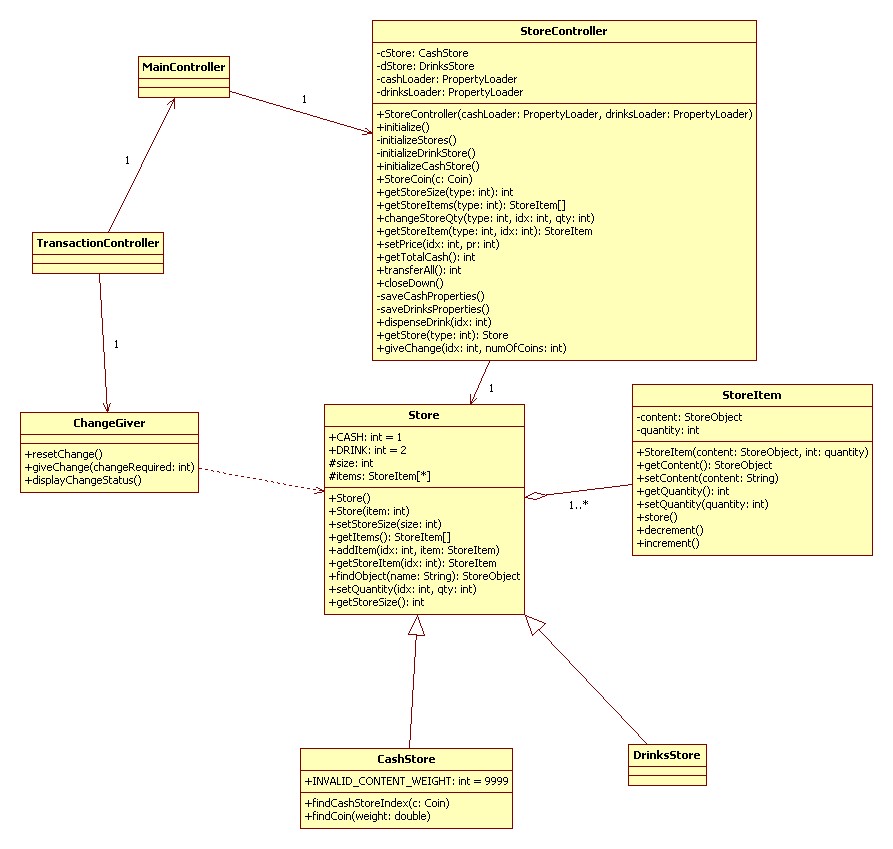
Reasons for choosing:

* Give change stores coin store information in memento. In case of VMCS Exception it will restore cash store from memento.
* Store Item is light weight objects. So creating copy of cash store item will not impact system performance.

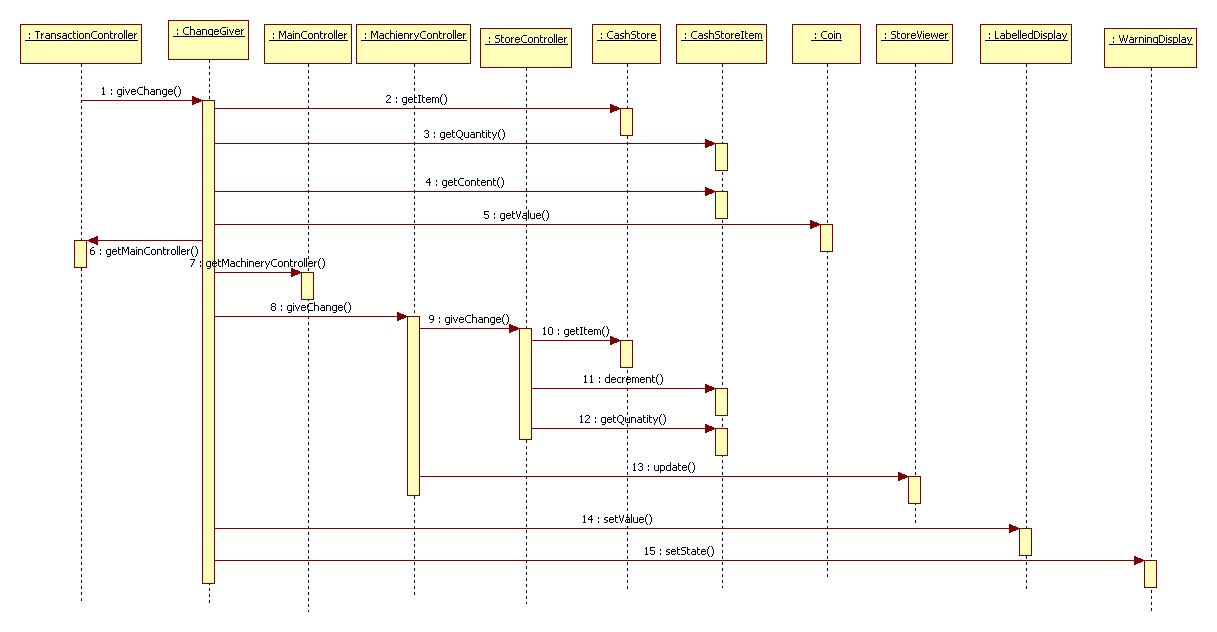
## **3.3 Design Changes**

### 3.3.1 Initial Design

Give Change Class Diagram

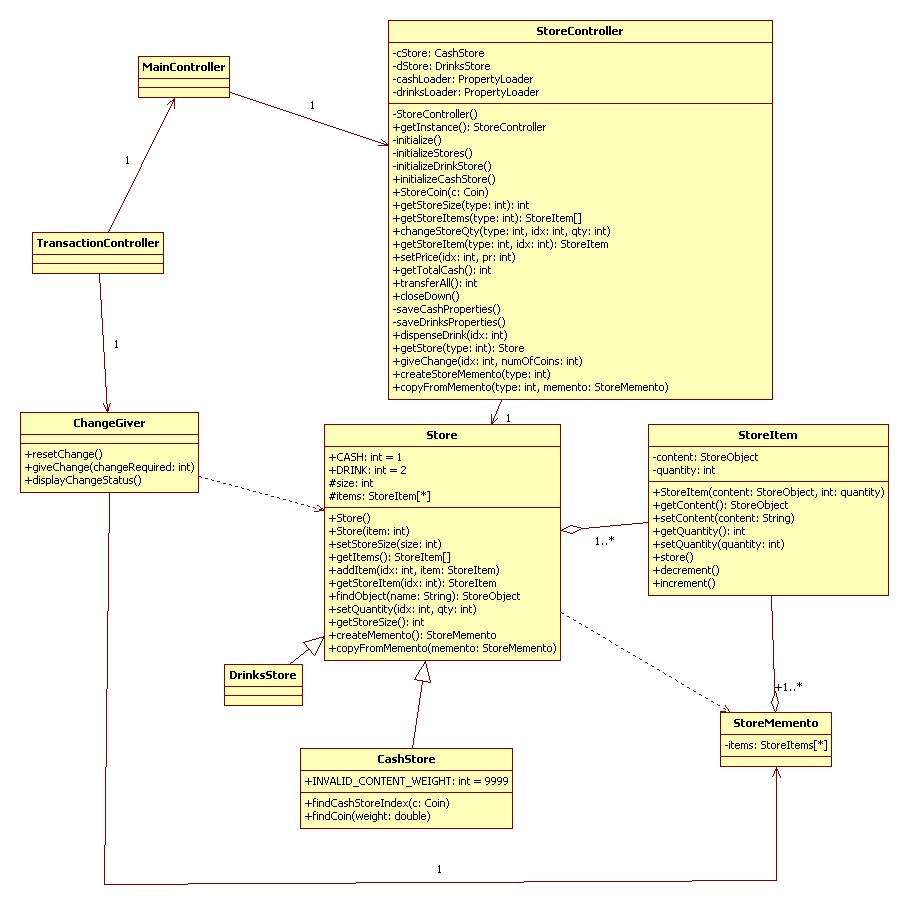


Give Change sequence Diagram

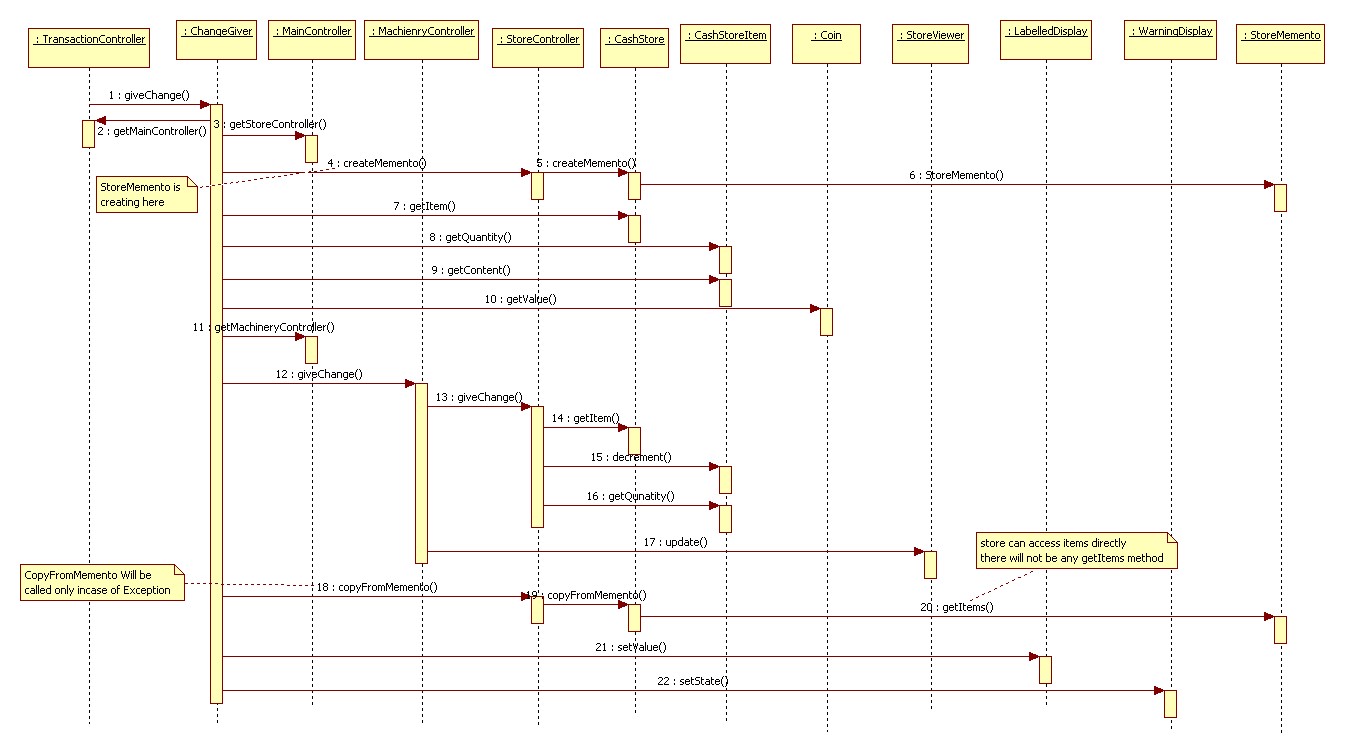


### 3.3.2 Modified Design

Give Change Class Diagram



Give Change sequence Diagram



## **3.4 Implementation**

* Language support
  + Using java inner class to support the narrow interface. Memento internal state can only be accessible from Store.
  + Cash store having few store items. Creating full copy of cash store object every time.

# 4. Create a general purpose GUI component (by JING DONG – A0135901)

## **4.1 Design Problem**

Class ButtonItemDispaly has to know the item collection mechanism for traversal. If the item collection type is changed, the client (Class ButtonItemDispaly) has to change the source code to traverse it. Current design didn’t provide a uniform interface for traversing different collection, so we cannot create a general purpose GUI component that will be able to iterate through any collection of the application.

## **4.2 Candidate design patterns**

1. CompositeIntent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
2. FactoryIntent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
3. IteratorIntent:

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Provide a uniform interface for traversing different collection.

Allows to modify the collection implementation without making any changes outside of collection

Enables to create a general purpose GUI component that will be able to iterate through any collection of the application

**Selected pattern:** Iterator

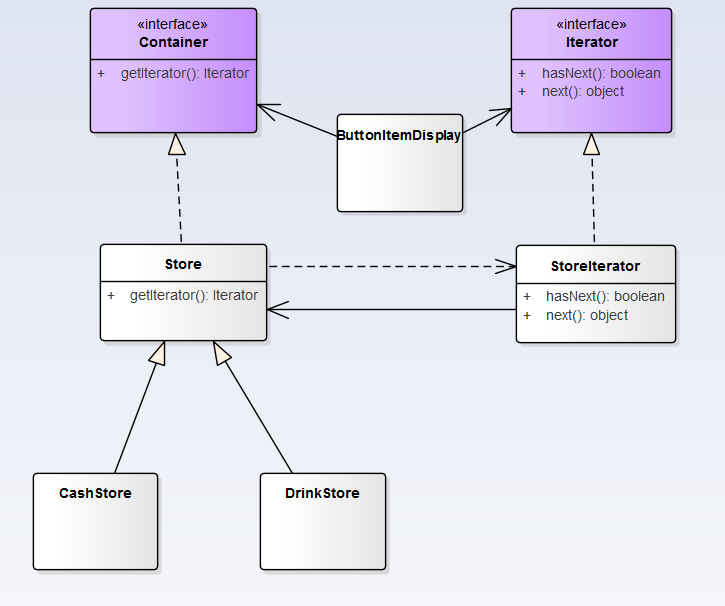
## **4.2 Motivation**

One of the most common data structures in software development is what is generic called a collection. A collection is just a grouping of some objects. They can have the same type or they can be all cast to a base type like object. A collection can be a list, an array, a tree and the examples can continue.

But what is more important is that a collection should provide a way to access its elements without exposing its internal structure. We should have a mechanism to traverse in the same way a list or an array. It doesn't matter how they are internally represented.

The idea of the iterator pattern is to take the responsibility of accessing and passing through the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

## **4.3 Class diagram**



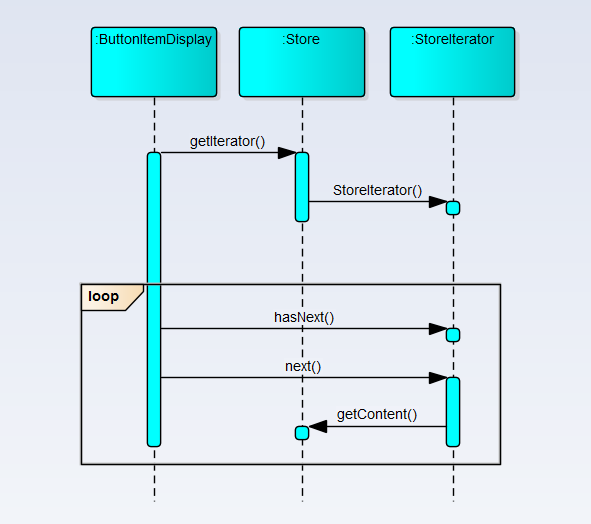
Iterator - This interface represent the Abstract Iterator, defining the iterator

StoreIterator - This is the implementation of Iterator (implements the Iterator interface)

Container - This is an interface defining the Aggregate

Store- An implementation of the Container

## **4.4 Sequence Diagram**



## **4.5 Implementation**

### 4.5.1 External vs. internal iterators.

External Iterators - when the iteration is controlled by the collection object we say that we have an external Iterator.

In our implementation an external iterator is implemented. In the following code an external iterator is used:

*while(iterator.hasNext())*

*{*

*StoreItem storeItem =(StoreItem) iterator.next();*

*StoreObject ob = storeItem.getContent();*

*}*

Internal Iterators - When the iterator controls it we have an internal iterator

When an internal iterator is used it means that the code is be run is delegated to the aggregate object.

The main idea is to pass the code to be executed to the collection. Then the collection will call internally the doSomething method on each of the components. In C++ it's possible to send the doMethod method as a pointer. In C# .NET or VB.NET it is possible to send the method as a delegate. In java the Functor design pattern has to be used. The main idea is to create a base Interface with only one method (doSomething). Then the method will be implemented in a class which implements the interface and the class will be passed to the collection to iterate.

### 4.5.2 We use External Iterators in the Implementation

**Who defines the traversal algorithm?**

The algorithm for traversing the aggregate can be implemented in the iterator or in the aggregate itself. When the traversal algorithm is defined in the aggregate, the iterator is used only to store the state of the iterator. This kind of iterator is called a cursor because it points to the current position in the aggregate.

The other option is to implement the traversal algorithm in the iterator. This option offers certain advantages and some disadvantages. For example it is easier to implement different algorithms to reuse the same iterators on different aggregates and to subclass the iterator in order to change its behaviour. The main disadvantage is that the iterator will have to access internal members of the aggregate. In Java and .NET this can be done, without violating the encapsulation principle, by making the iterator an inner class of the aggregate class.

In our implementation, we use inner class for the iterator, and define traversal algorithm in the iterator.

*private class StoreIterator implements Iterator {*

*int index;*

*@Override*

*public boolean hasNext() {*

*if(index < items.length){*

*return true;*

*}*

*return false;*

*}*

*public Object next() {*

*if(this.hasNext()){*

*return items[index++];*

*}*

*return null;*

*}*

*}*

**Robust Iterators**

A robust iterator is that elements are not accessed twice or ignored when elements are added or removed from an aggregate during iteration.

In our implementation, we don't need a robust iterator. We can just throw an exception when an element is added to the collection stating that the operation has failed.

**Iterator and multithreading**

In our implementation, the collection return a new iterator (using in our example the getIterator method). Usually this step is not affected when it is used in multithreading environments because it returns a new iterator object.

Step two: The iterator is used for iterating through the objects. Since the iterators are different objects this step is not a problematic one in multithreading environments.

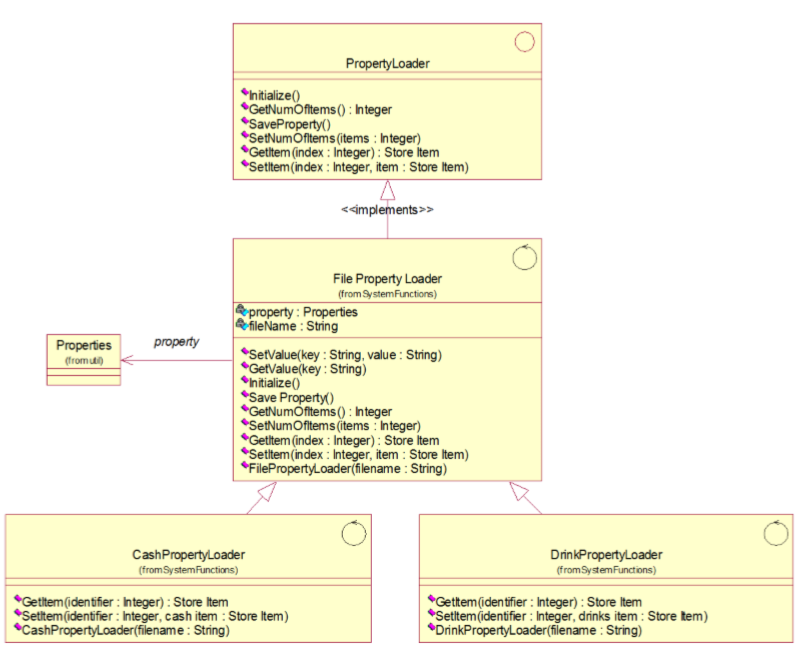
It seems that the iterator does not raise special problems when a collection is used from different threads.

# 5. Bridge Pattern Design (by SITHIVINAYAGAM KOKULAKUMARAN - A0120490U)

## **5.1 Design Problem**

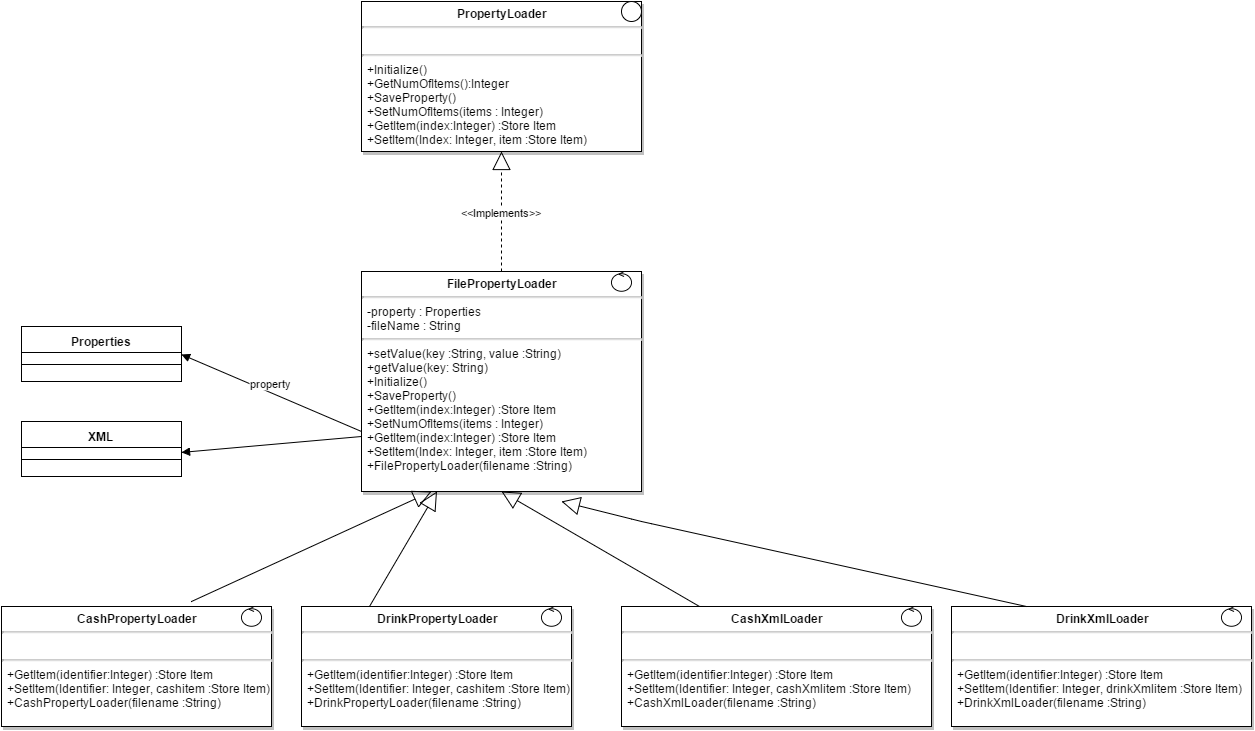
It is difficult to add or change the persistence mechanism in the existing system of VMCS. Currently it is done only using properties file. There is no possibility given to user to change the storing mechanism. If we need to add XML based data store mechanism, we need to add 3 classes. Hard to extend Business Objects are having persistence logic.

The following is the class diagram of existing design of the persistence mechanism.



Current design doesn’t provide the expandability, easy maintainability which should have been expected in the design stage. Though this design is well satisfied the current requirement, there is no room to extend, if requirement for changes comes in future to collaborate different kind of storing mechanism.

The following is the brute forced class diagram of implementing XML persistence mechanism to current design. Each additional mechanism inclusion will add more classes and code repetitions.



## **5.2 Candidate Patterns**

The following design patterns have been analyzed to solve this design problem.

1. Adapter Design Pattern

Adapter pattern converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

1. Bridge Design Pattern

Bridge patterns decouples an abstraction from its implementation so that the two can vary independently

1. Abstract Factory

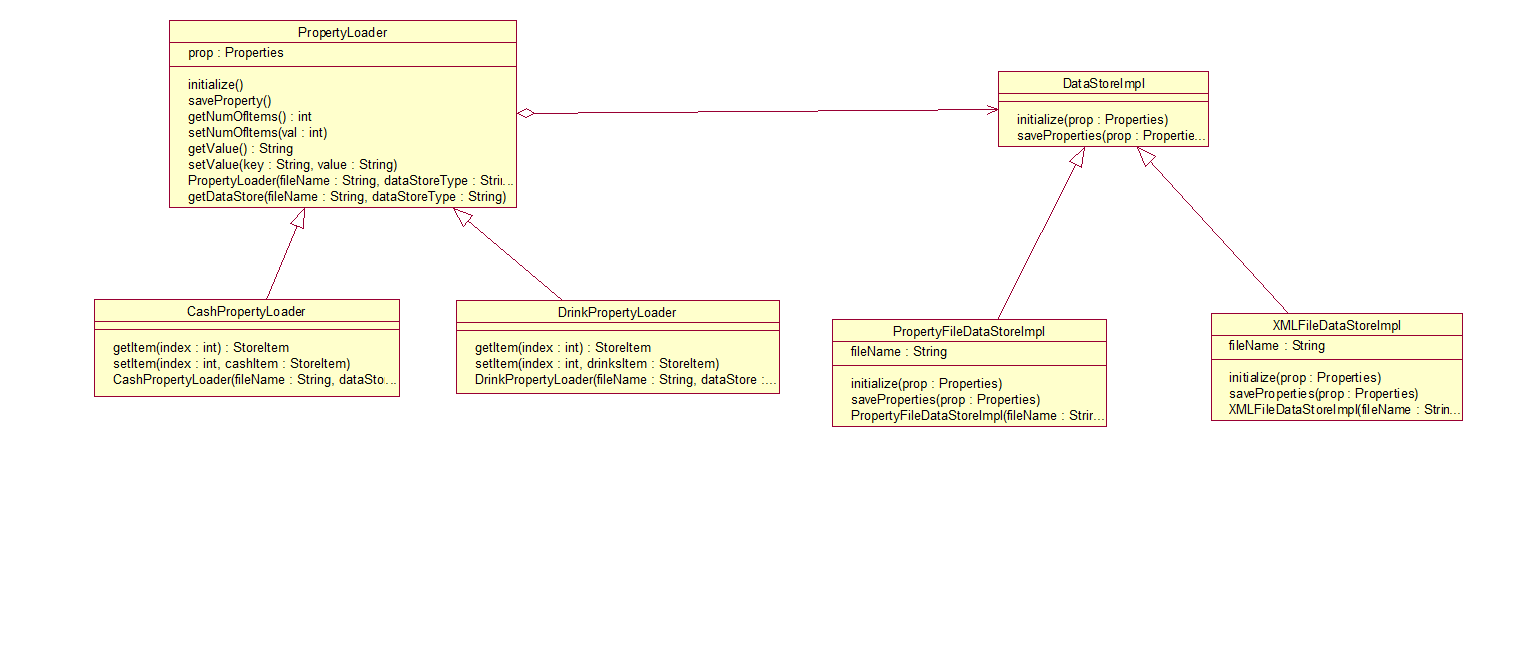
Abstract Factory pattern provides an interface for creating families of related objects without specifying their concrete class. For current design issue, will need only one storage mechanism at a time.

## **5.3 Motivation to select bridge pattern**

Bridge pattern has been selected to solve this design issue out of three candidate design. According to current design, the implementation related functionalities has to be removed from the CashPropertyLoader and DrinkPropertyLoader classes. So that, they don’t need to stick with other abstract functionalities. So implementation detail can be decided in run time. Further it is not required to expose the implementation to clients. And only one class is needed to be added when adding different storage mechanism.

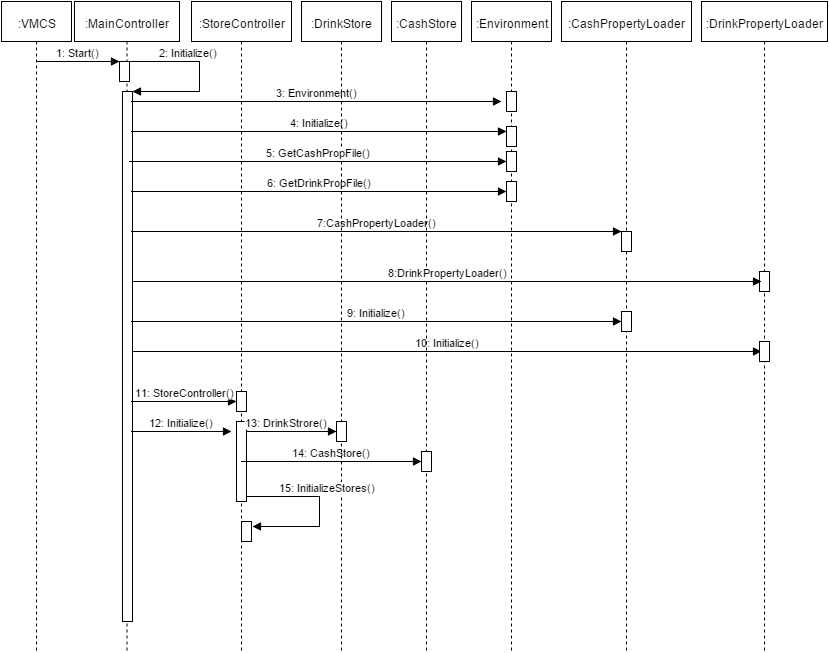
## **5.4 Class Diagram after applying the design pattern**

The following is the class diagram after applying the bridge design pattern with two persistence mechanisms such as properties file and XML file.

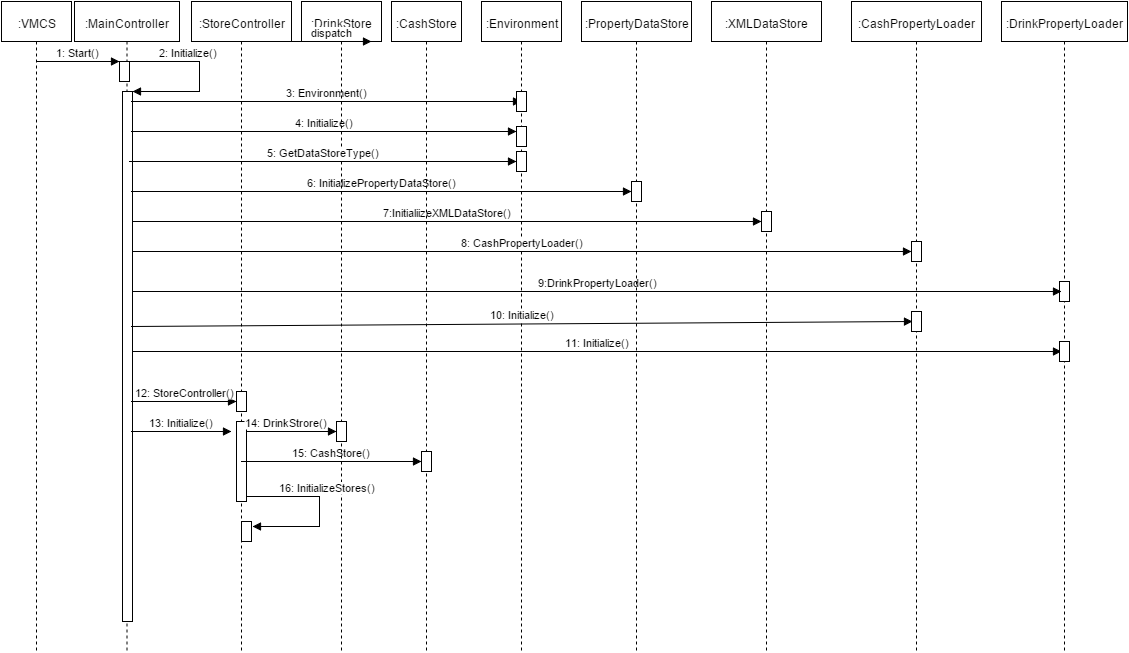
****

## **5.5 Sequence Diagrams**

Following sequence diagram illustrates system start up before applying the bridge design pattern



Following sequence diagram illustrates system start up after applying the bridge design pattern



## **5.5 Implementation Decisions**

1. Only one Implementer

In some scenarios, only one Implementer class should be needed, for those cases no need to define abstract implementer, concrete implementer is enough. But in this case, currently we decided to have two concrete implementers and this might get increased based on the future needs.

1. Creating the implementer object

Though abstraction knows the concrete implementer classes, to enable the desired restrictions (not to use some implementer in concrete abstraction) within concrete abstraction classes, we decided to put the concrete implementer creation in the subclasses of Loader those are CashLoader, DrinkLoader. Factory pattern can be used for externalization of implementation class creation. DataStore type will be retrieved from vcms properties file

1. Sharing implementers

Implementers are not shared in this context.

1. Using multiple Inheritance

Multiple inheritance are not supported directly in Java. So it is not relevant in this context.

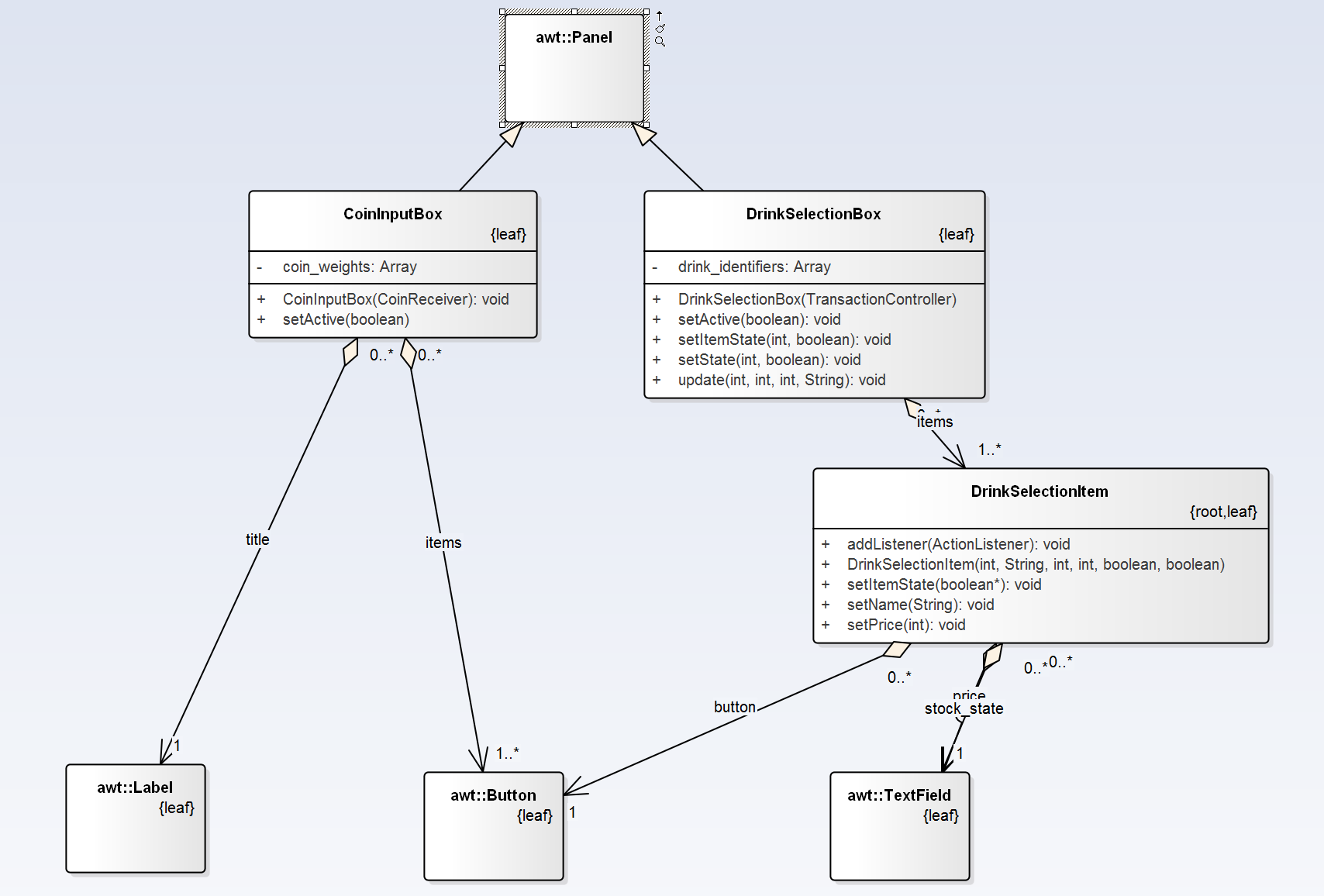
# 6. Command Pattern (by SIVANANTHAMOORTHY JANAHAN – A0120591)

## **6.1 Design Problem**

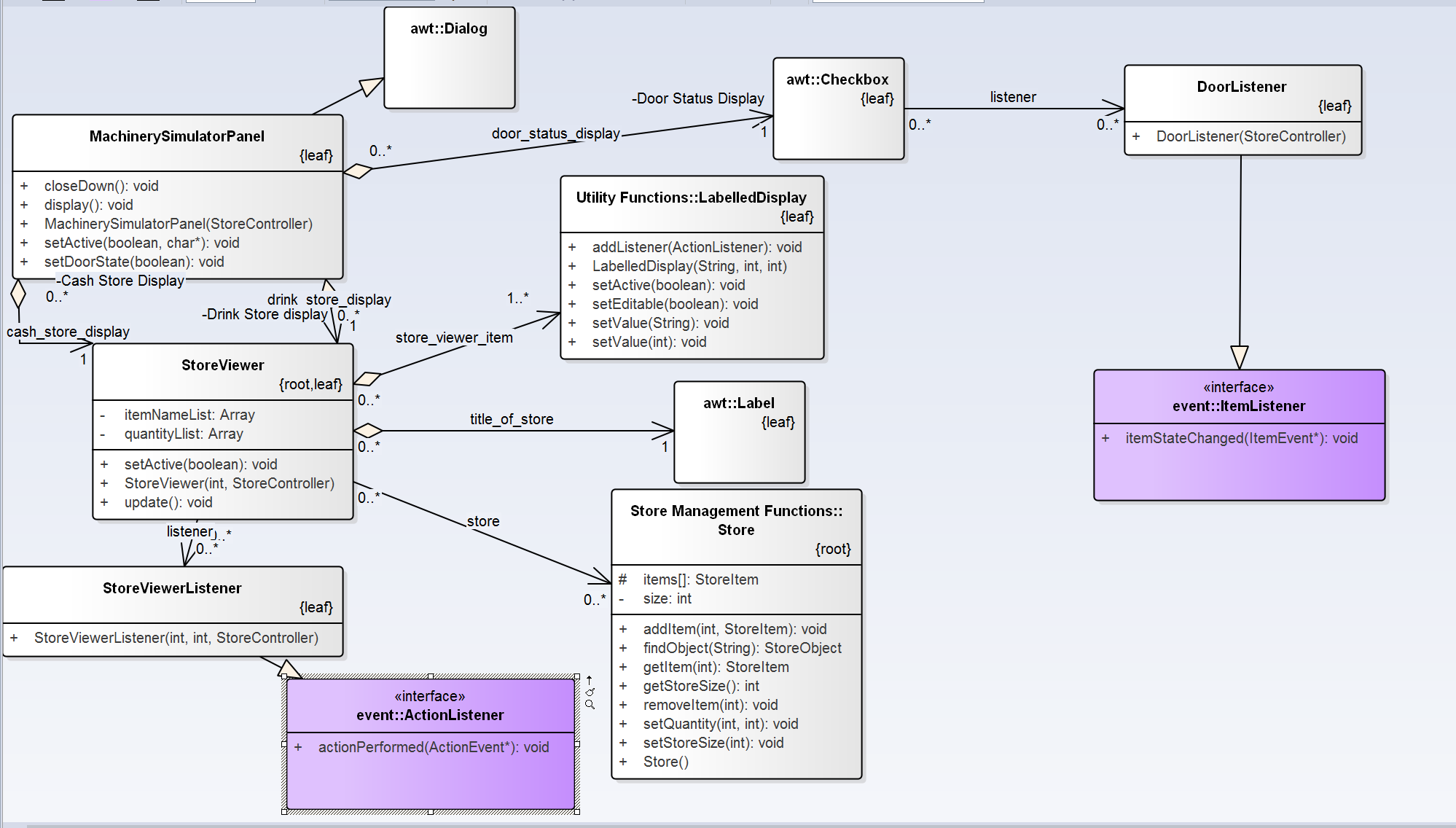
In this design GUI components need to know which operation in the respective controller to invoke. In here every listener is initializing controllers. Here client and service rendering receiver objects are coupled together. For example: According the current design DrinkSelectionListener class initializing 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.

## **6.2. Class Diagrams (Before Applying Command Pattern)**

### 6.2.1 Drink Selection

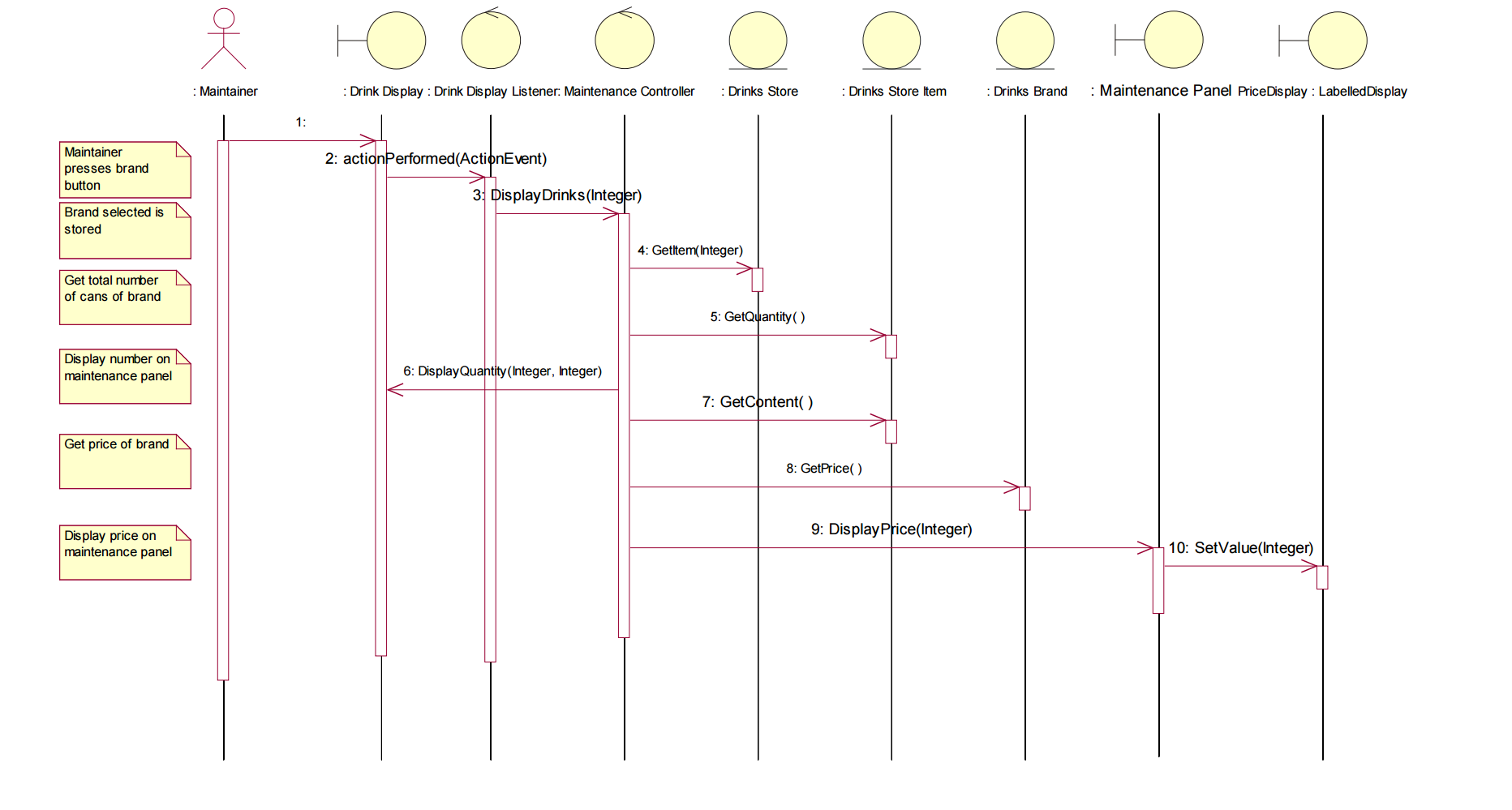


### 6.2.2 Change Cash Store

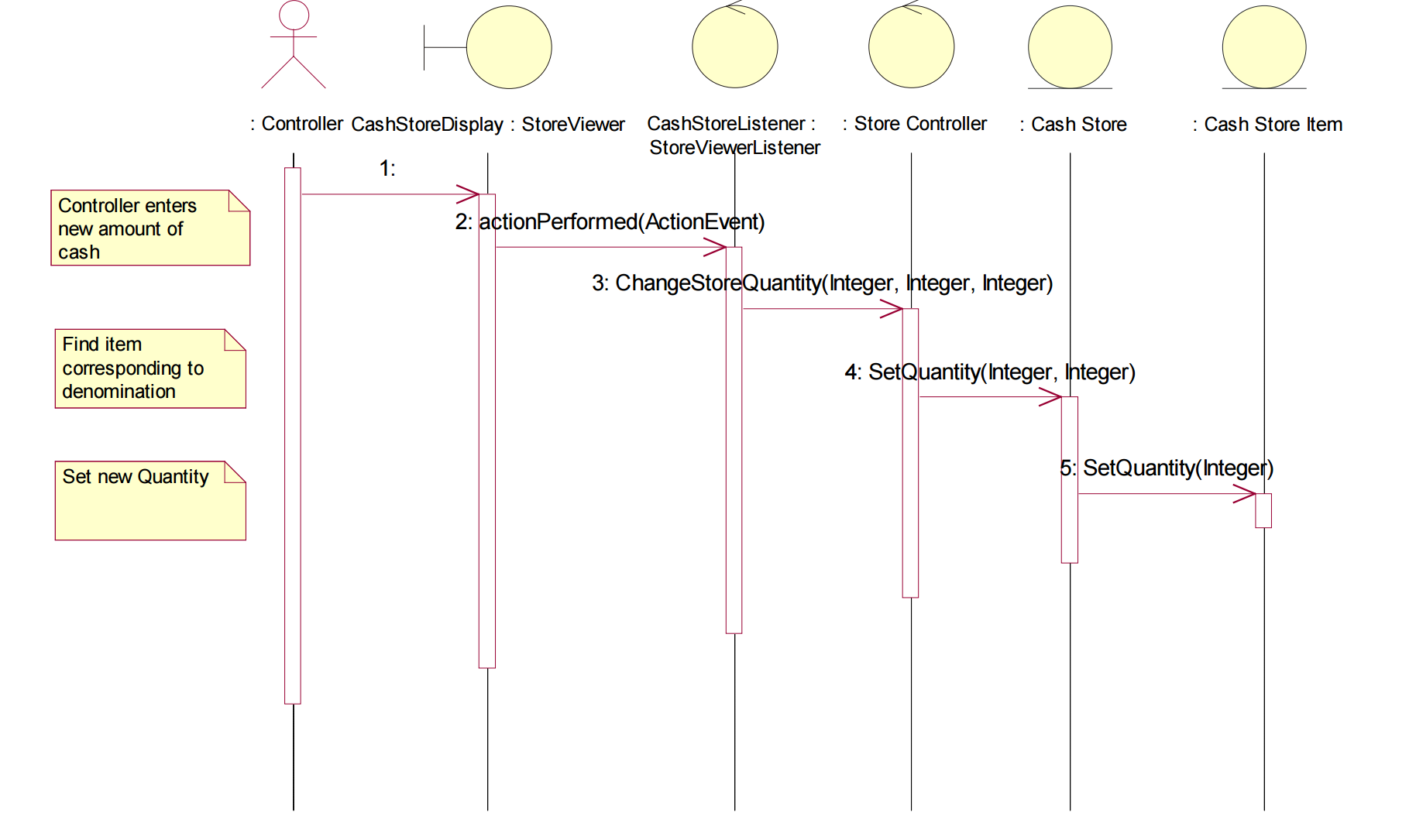


## **6.3. Sequence Diagram (Before Applying Command Pattern)**

### 6.3.1 Drink Selection



### 6.3.2 Change Cash Store



## **6.4. 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:

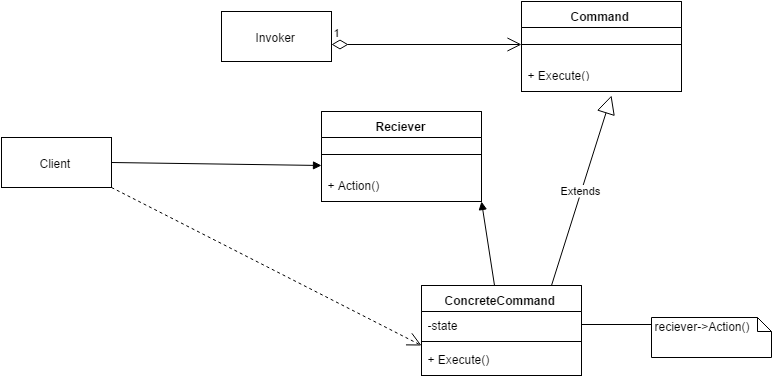
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interactions independently.

**Selected Pattern:** Command Pattern

## **6.5. 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.

## **6.6. 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.

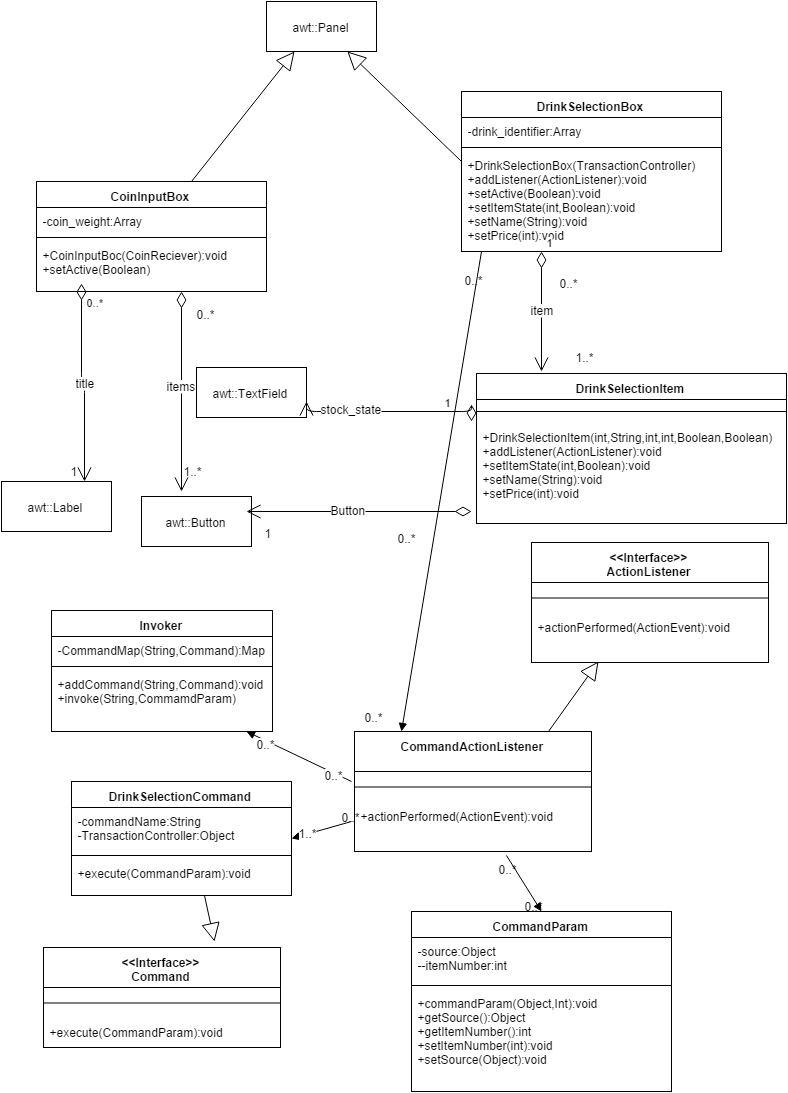
* Asks the command to carry out the request.

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

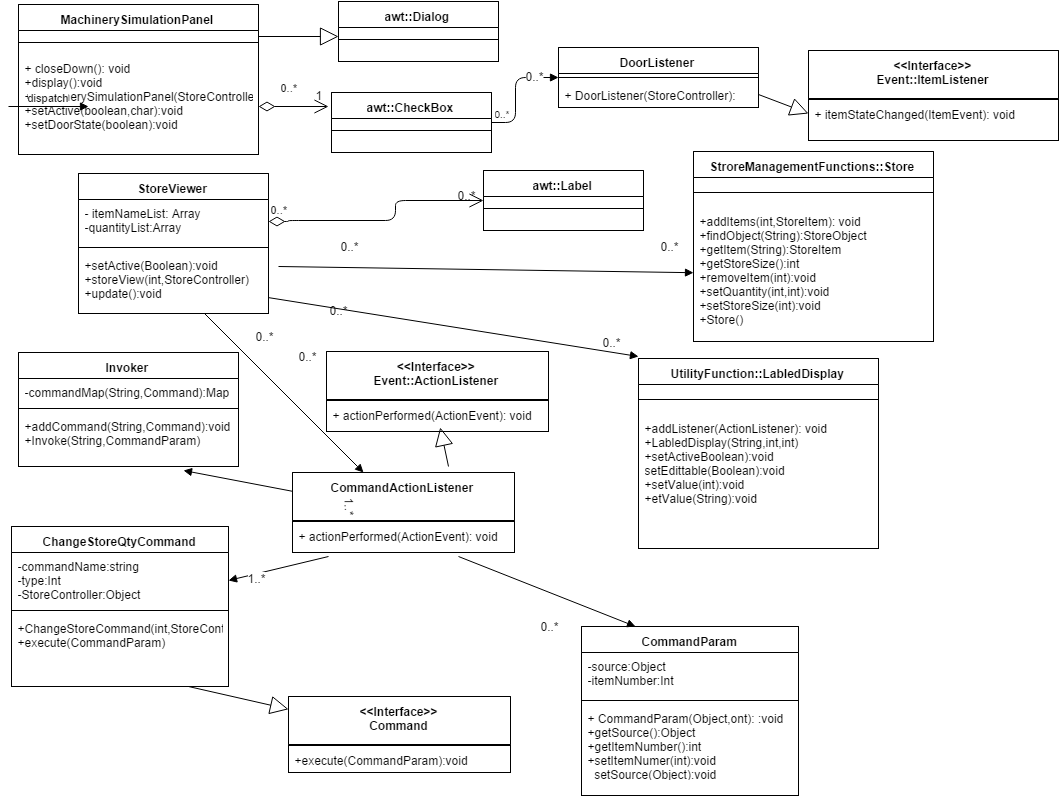
* Knows how to perform the operations associated with carrying out a request

## **6.7. Class Diagram (After Applying Command Pattern)**

### 6.7.1 Drink Selection

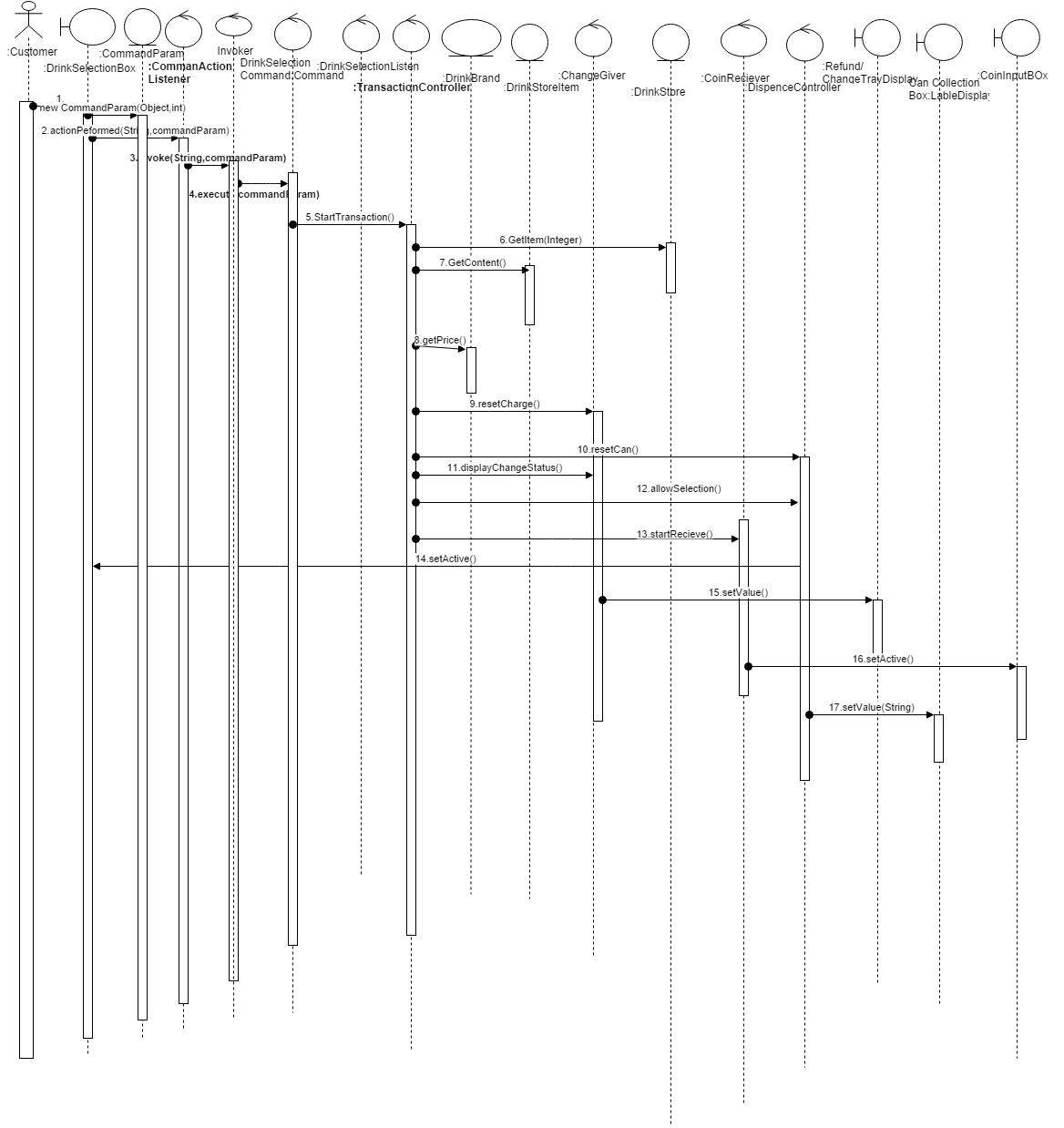


### 6.7.2 Change Cash Store

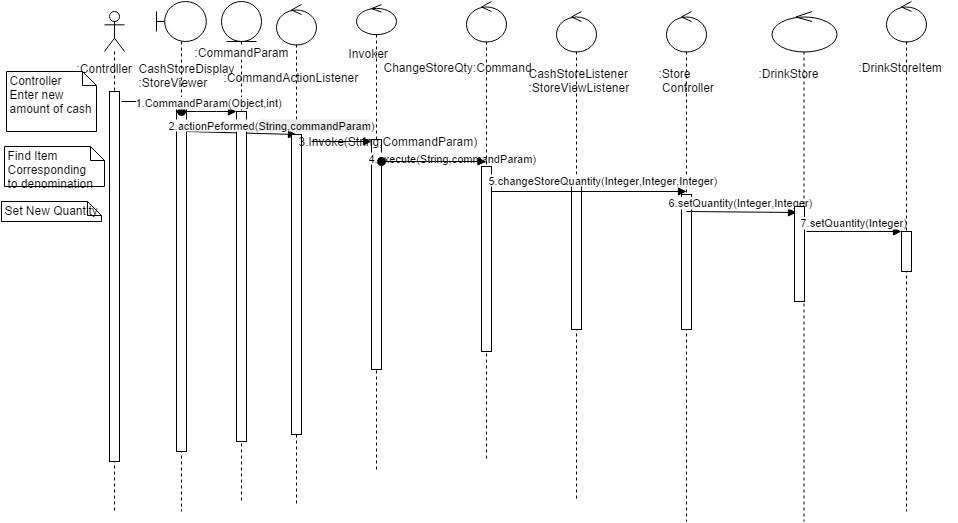


## **6.8. Sequence Diagram (After applying Command Pattern)**

### 6.8.1 Drink Selection



### 6.8.2 Change Cash Store



## **6.9. 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.