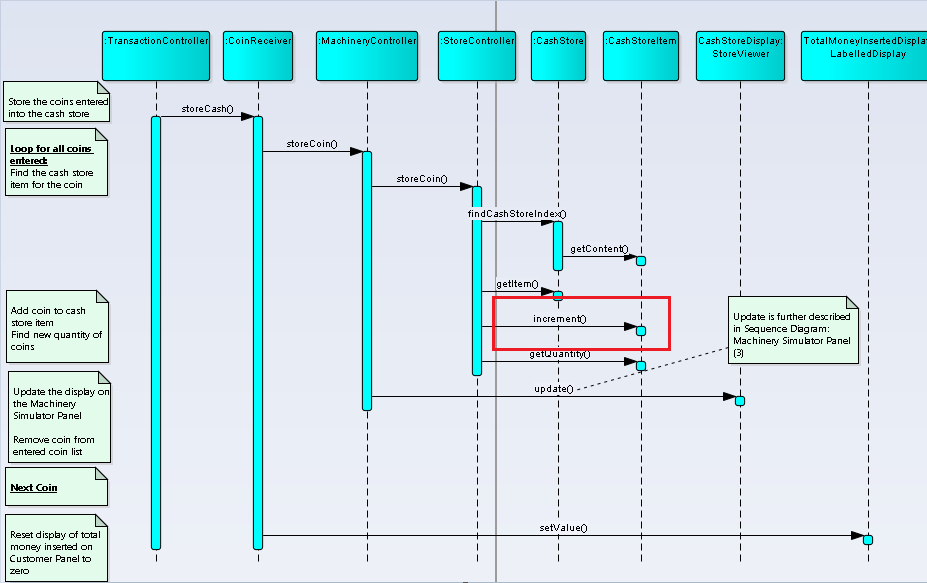
# Design Problem

* In the Vending machine multiple user panels for example Customer panel, Maintenance panel and Machinery panel depend on same data object (e.g. StoreItem). Cash and Drinks Quantities are displayed on Maintenance panel and Machinery panel but further changes in StoreItem are not propagated to these panels.
* The Vending machine does not allow auto refresh mechanism in all user panels when one panel cause some change in StoreItem.

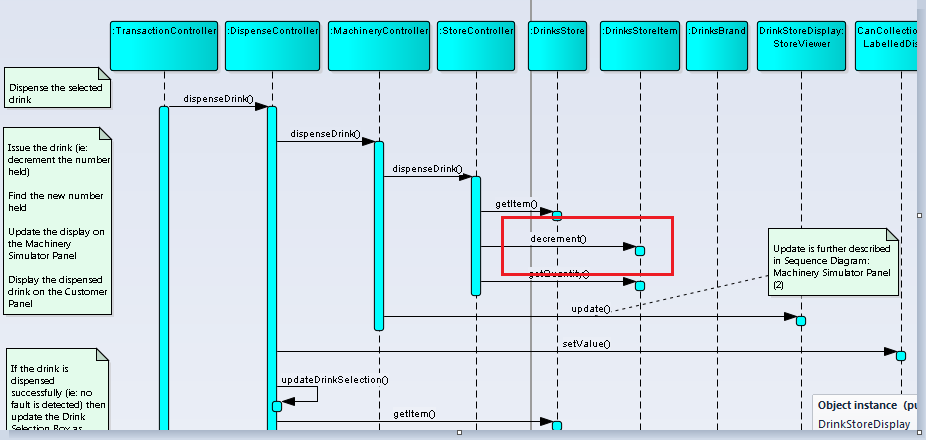
# Current Design (Sequence Diagram)

**Store Coins**



No update to MachineryController and MaintenanceController

**Dispense Drink**



No update to MachineryController and MaintenanceController

# Candidate Design Pattern Considered

The identified design problem is a Behavioural issue therefore the following Candidate Design Pattern can be considered:

|  |  |  |
| --- | --- | --- |
|  | **Observer** | **Mediator** |
| Intent | Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically | 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 interaction independently. |
| Applicability | * When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently   + Different views of the same object can be separated and encapsulated into different classes, so that the object can be reused independent of the views * When a change in the object requires change in the others, and you do not know how many of them need changes * Need loose coupling | * A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand. * Reusing an object is difficult because it refers to and communicates with many other objects. * A behaviour that's distributed between several classes should be customizable without a lot of sub classing. |
| Aspect | Number of objects that depend on another object; how the dependent objects stay up to date | How and which objects interact with each  other |
| **Decision** | **After study the two candidate patterns, the decision is to use Observer pattern. The reason for choosing Observer pattern over Mediator pattern is because here change in one object requires change in others and dependant object need to stay up to date.** | |

# Participants

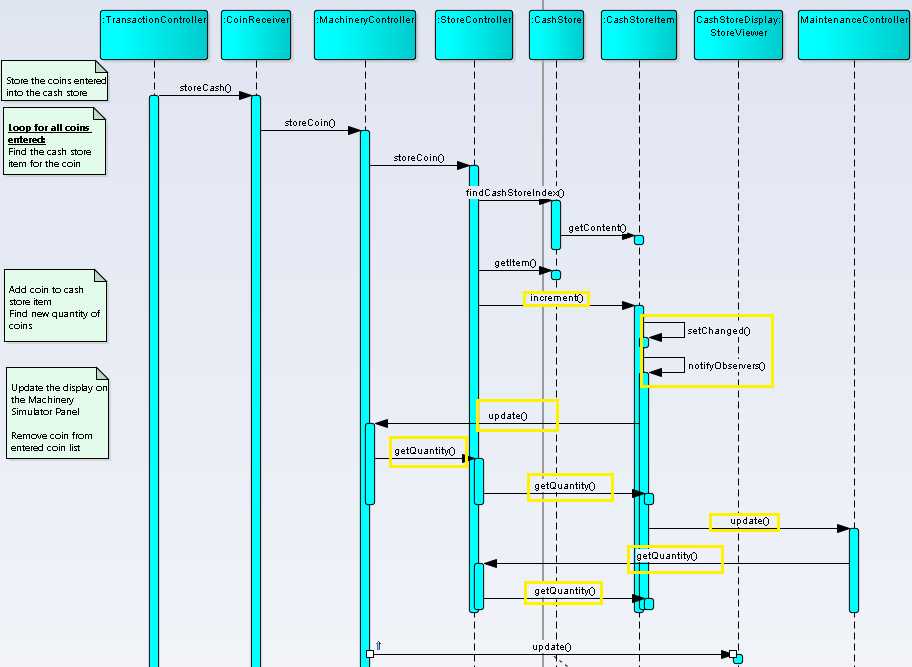
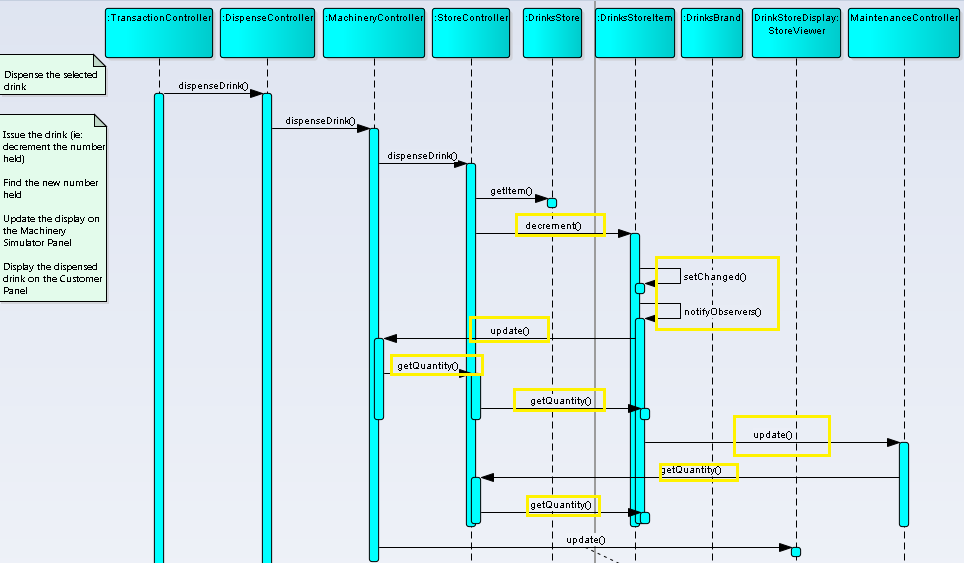
The classes and objects participating in this Observer pattern are:

* Subject (Java observable)  
  + Java built-in observable
  + Knows its observers. Any number of Observer objects may observe a subject.
  + Provides an interface for attaching and detaching Observer objects.
* ConcreteSubject (StoreItem)  
  + Stores state of interest to ConcreteObserver objects.
  + Sends a notification to its observers when quantity changes.
* Observer (Java Observer)
  + Java built-in observer interface.
  + Defines an updating interface for objects that should be notified of changes in a subject.
* ConcreterObserver (MachinaryController, MaintenanceController)
  + Maintains a reference to a ConcreteSubject object.
  + Implements the Observer updating interface to keep its state consistent with the subject's.

# Revised Design (Class Diagram)

# 

# Revised Design (Sequence Diagram)

**Store Coins:****Dispense Drink:**

# Implementation

There are several issues that were considered when applying the Observer pattern.

|  |  |
| --- | --- |
| **Issue** | **Rationale** |
| Who triggers the update? | Either subject or client can be made to trigger the update.  If client is made to trigger, then the disadvantage is that the clients have an added responsibility to trigger the update. This makes errors more likely, since clients might forget to call Notify.  Here we cannot afford missing few intermediate notifications, so the Subject call Notify after they change the subject's state. |
| Dangling references to deleted subjects. | Deleting a subject should not produce dangling references in its observers. One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset/detach their reference to it. Here in Java, dangling references cannot occur. |
| Avoiding observer-specific update protocols: the pushpull models. | In the 'push' model, the subject (i.e. the Observable) sends the observer on notification all the data it will need. The observer doesn't need to query the subject for information. In the 'pull' model, the subject merely notifies the observer that something happened, and the observer queries the subject based to get the information it needs.  Here push model has been used.  The main advantage of the 'push' model is lower coupling between the observer and the subject. The observer doesn't need to know anything about the subject in order to query it. |