# Observer Pattern

## 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.

## 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

## 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 reduce 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.

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



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

### Establish Observation



### Notify



## 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.