AUTHTORS:

SIVASANKARI D/O SUBRAMANIAM

TEAM MEMBER 2

TEAM MEMBER 3

TEAM MEMBER 4

TEAM MEMBER 5

TEAM MEMBER 6

VMCS: A New Perspective

Object Oriented Design pattern

Version 1.0

October 26, 2016

# Overview

Based on the design document, we have noticed areas in which Object Oriented Design Patterns can be applied.

Design Patterns help us in the following ways:

1. Extensibility
2. Reusability
3. Maintainability

# Design patters chosen

1. Observer Design Pattern
2. Factory Design Pattern
3. Chain of Responsibility Design Pattern

The next sections will focus on the following items for every Design Pattern chosen,

1. Design Problem
2. Current Design
3. Candidate Design Pattern(s) considered
4. Participants
5. Revised Design
6. Implementation

# Observer Design Pattern

## Design Problem

* The vending machine has multiple user panels. They are,
  + Customer panel
  + Maintenance panel
  + Machinery panel
* They depend on the same data object (e.g. StoreItem).
* Cash and Drink 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 updates in other panels change the StoreItem.

## Current Design

### Sequence Diagram

No update to MachineryController and MaintenanceController

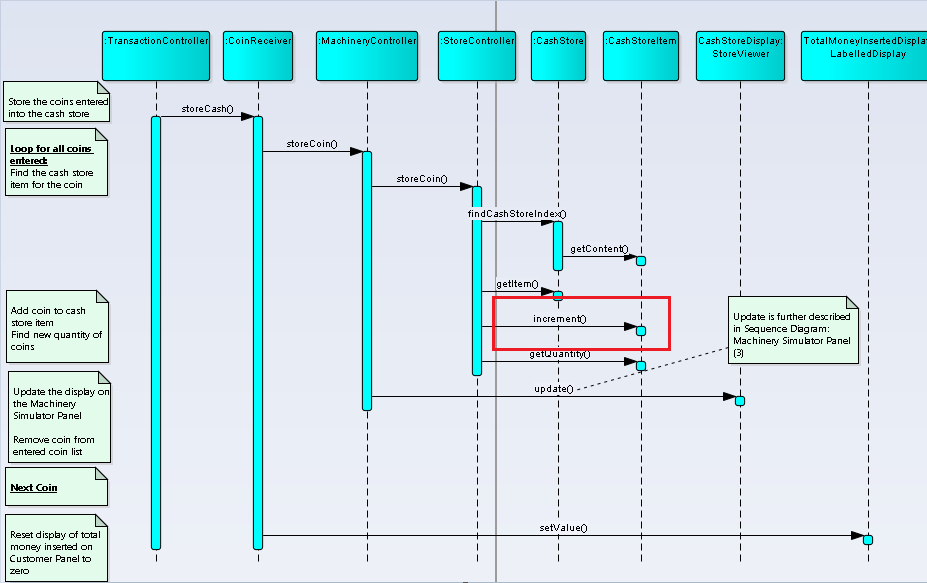
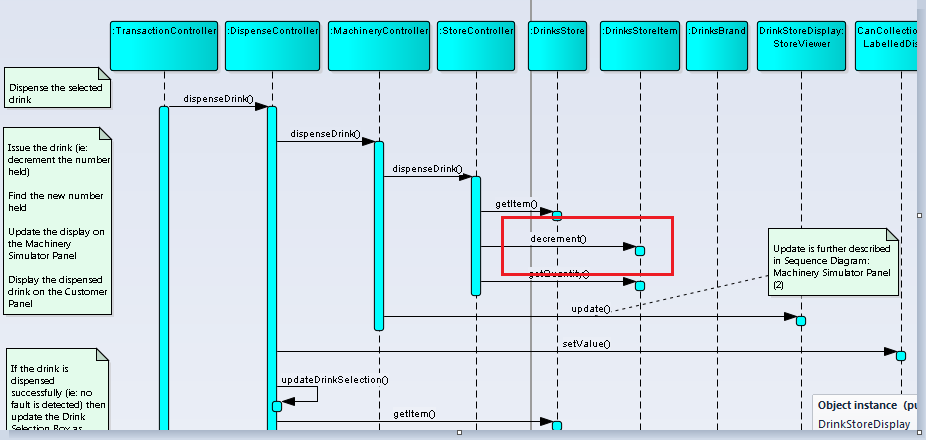


Figure 1: Store Coins

Figure 2: Dispense Drink



No update to MachineryController and MaintenanceController

## Candidate Design Pattern(s) Considered

The identified design problem is a Behavioural issue. Therefore, the following Candidate Design Patterns were considered:

|  |  |  |
| --- | --- | --- |
| Deciding Factors | 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 studying both the two candidate patterns, the decision was to use the Observer pattern. The reason for choosing Observer pattern over Mediator pattern is because, here, change in one object requires change in others. And dependent objects need to be up to date.** | |

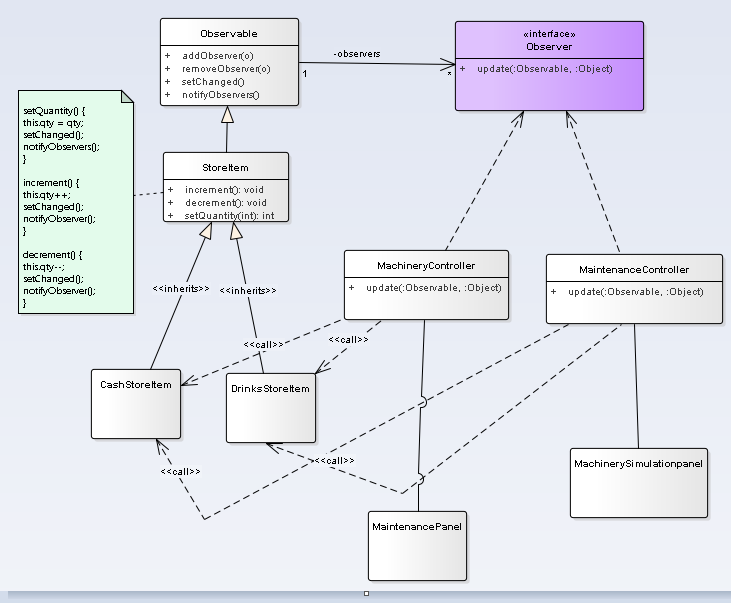
## Participants

The classes and objects participating in this Observer pattern are:

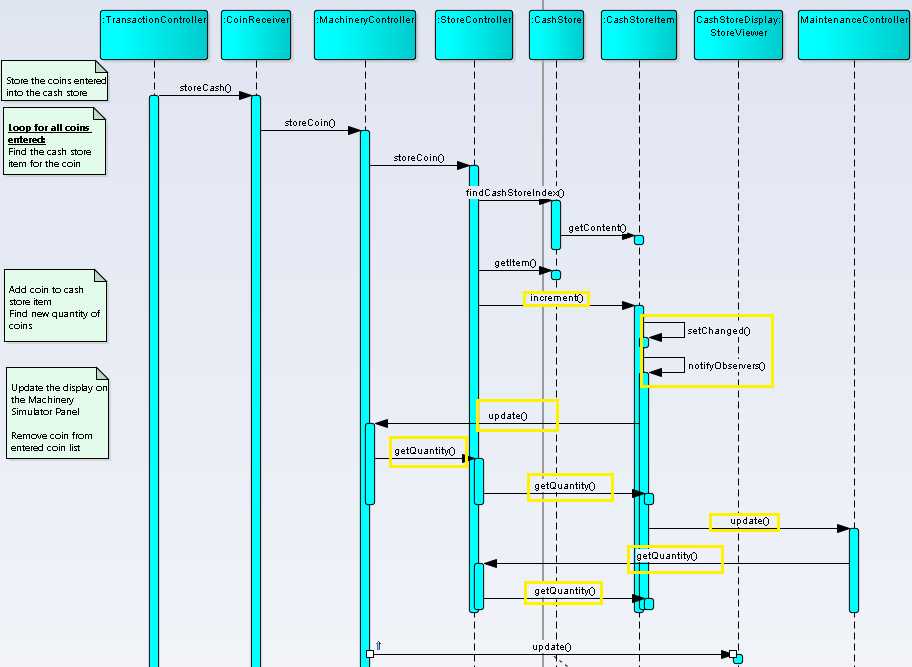
* Subject (Java Observable)
  + Java built-in observable
  + Knows its observers. Any number of Observer objects may observe a subject.
  + Provides an interface for attaching and detaching Observer objects.
* ConcreteSubject (StoreItem)
  + Stores state of interest to ConcreteObserver objects.
  + Sends 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



### Sequence Diagram



## Implementation issues

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. |

# Factory Design Pattern

## Design Problem

* The system must be able to accept other payment methods
  + Example, cash.
* Currently, the transaction controller is creating the coin receiver class explicitly.

## Current Design

## Candidate Design Pattern(s) Considered

A creational design pattern is required.

1. Factory Method.
   * Using an interface to create objects and let subclasses decide which class to instantiate.
2. Singleton
   * Ensure a class has only one instance, and provide a global point of access to it.
3. Prototype
   * A fully initialised instance to be copied or cloned.

We have chosen Parameterised Factory design pattern implemented with Singleton.

Factory design pattern allows new payment methods to be added easily. Singleton ensures that there is only one instance of the payment method.

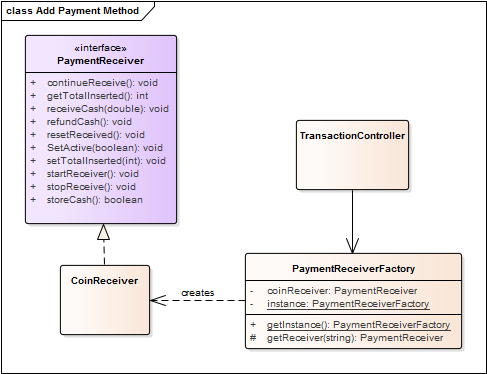
## Participants

  The classes and objects participating in this pattern are:

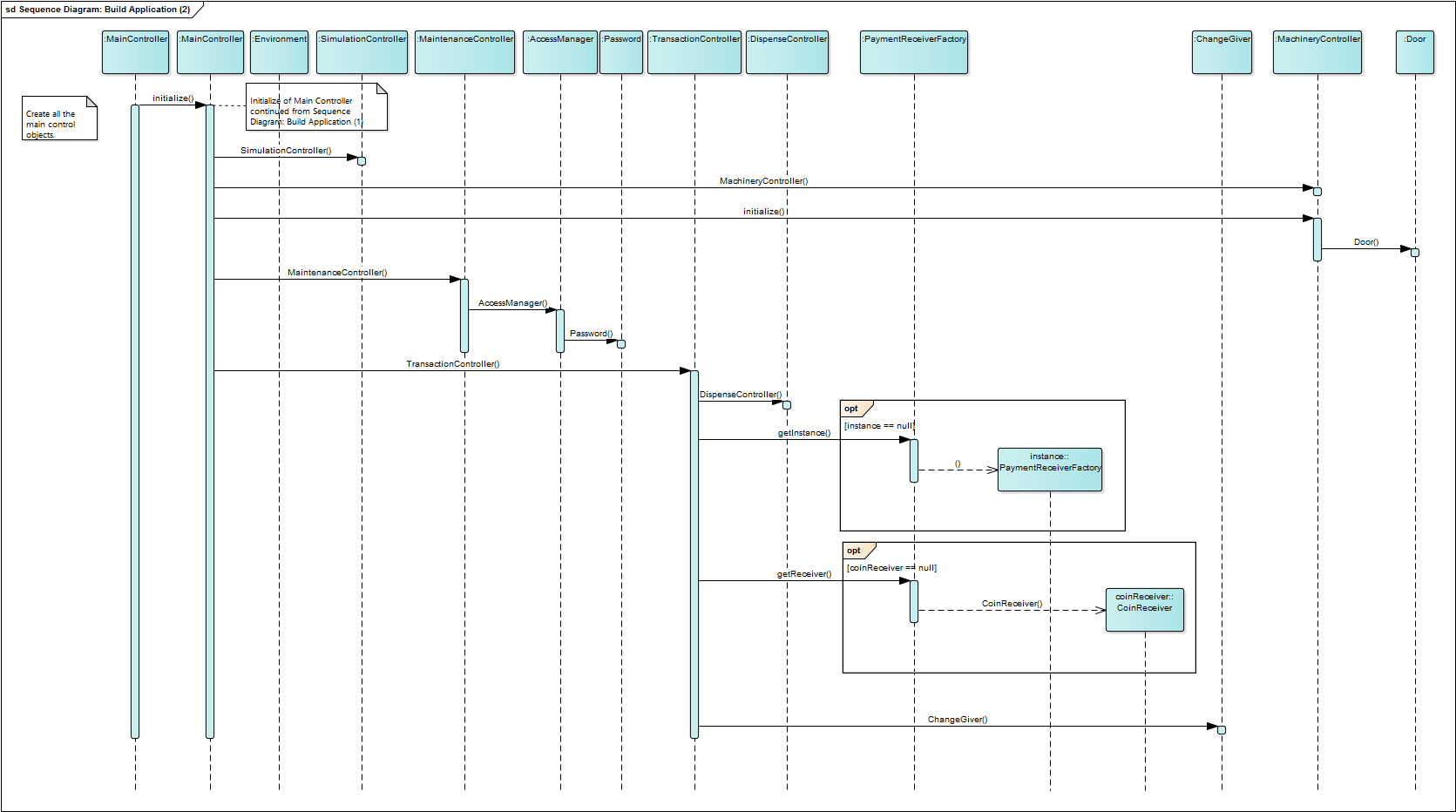
* PaymentReceiver
  + defines the interface of objects the factory method creates
* ConcreteReceiver  (CashReceiver)
  + implements the PaymentReceiver interface
* PaymentReceiverFactory
  + It declares the factory method, which returns only an instance of an object of type PaymentReceiverFactory.
* Client (TransactionController)
  + Initialized the PaymentReceiverFactory

Revised Design

Class Diagram



Sequence Diagram



## Implementation Issues

There are 2 main variations of the Factory Method design pattern. To enforce that only 1 instance of the factory class, a parameterized factory pattern would be the most suitable.

The PaymentReceiverFactory will not be subclassing because its main functionality is to create instances of PaymentReceiver. Enforcing singleton ensures that only 1 instance of PaymentReceiverFactory and CoinReceiver are created.

# Chain of Responsibility

## Design Problem

* The ChangeGiver class is implementing giveChange method.
* The method does an explicit for loop to go over each domination of coins.
* There is a tight coupling between the requester and handler.
* It is not open for extensibility due to the tight coupling.

## Current Design

### Sequence Diagram



## Candidate Design Pattern(s) Considered

|  |  |  |
| --- | --- | --- |
| Deciding Factors | Chain of Responsibility | Mediator |
| Intent | A handler is used for processing the message. Different handlers process different requests. | A mediator object is used to send messages from requesters to the appropriate classes. |
| Applicability | * When there is a need for loose coupling, we can use this pattern to decouple the requester and receiver classes. | * When there is a need for loose coupling, we can use this pattern by removing direct dependencies between classes. |
| Decision | **We have decided to use the Chain of Responsibility design pattern. Each process will be handled by the respective handler. We decided not to have a mediator object to send messages. Chain of Responsibility design pattern allows loose coupling between the client and handler, in this case ChangeGiver class, and the handlers processing the requests. Each domination of change will be handled by the respective class.** | |

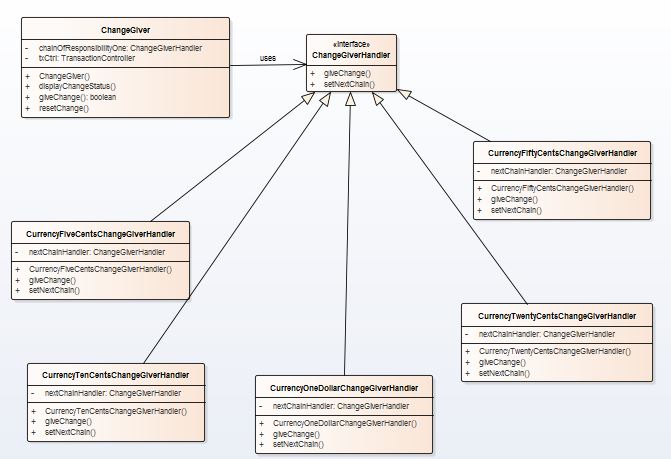
## Participants

  The classes and objects participating in this pattern are:

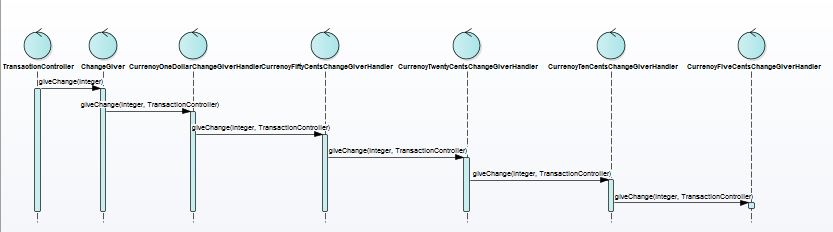
* Handler
  + “defines the interface for handling requests” (Erich Gamma, 1994)
  + ChangeGiverHandler.java
* ConcreteHandler
  + Implements the respective requests
  + Total of five concrete handler classes (1 dollar, 50, 20, 10, 5 cents)
  + CurrencyFiftyCentsChangeGiverHandler.java
  + CurrencyTwentyCentsChangeGiverHandler.java
  + CurrencyTenCentsChangeGiverHandler.java
  + CurrencyFiveCentsChangeGiverHandler.java
  + CurrencyOneDollaeChangeGiverHandler.java
* Client
  + This will initiate the call to handler to start processing the request
  + ChangeGiver.java
  + The successor chains will be defined here

## Revised Design

### Class Diagram



### Sequence Diagram



## Implementation Issues

|  |  |
| --- | --- |
| Issue | Rationale |
| Implementing the successor chain | The new links can be defined either in the Handler or the ConcreteHandler.  For our implementation, the new links are defined in the ConcreteHandler itself.  The following will list the sequence of calling the different handlers (based on dominations).   1. One dollar 2. 50 cents 3. 20 cents 4. 10 cents 5. 5 cents |
| Connecting Successors | The handler does not maintain the successor for forwarding the requests.  It is handled solely by the ConcreteHandlers. |
| Representing Requests | Request parameter is taken as request code.  The amount to be returned and the transaction controller are the parameters in this case.  Since the change can be any amount, taking in a parameter will allow more flexibility on how the change is processed.  But we are not using any Request class to bundle the request parameters. |

# Reference List

Erich Gamma, 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 Edition. Addison-Wesley Professional.