AUTHORS:

SIVASANKARI D/O SUBRAMANIAM

NG SHEN SOON BENJAMIN

LU WENSHUO

KOO SHENG KIAT

SMITA RANI BISOYI

ZHAO YUHENG

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
2. Factory
3. Chain of Responsibility
4. Bridge
5. Decorator
6. Template Method

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.

## Motivation

* To provide loose coupling between data object and views.
* Data object can be reused without any impact on view.
* To provide the ability, when there is a change in the date object then pass the change to the other panels, and without any need to know how many of them need changes.

## Candidate Design Pattern(s) Considered

The identified design problem is a Behavioral 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

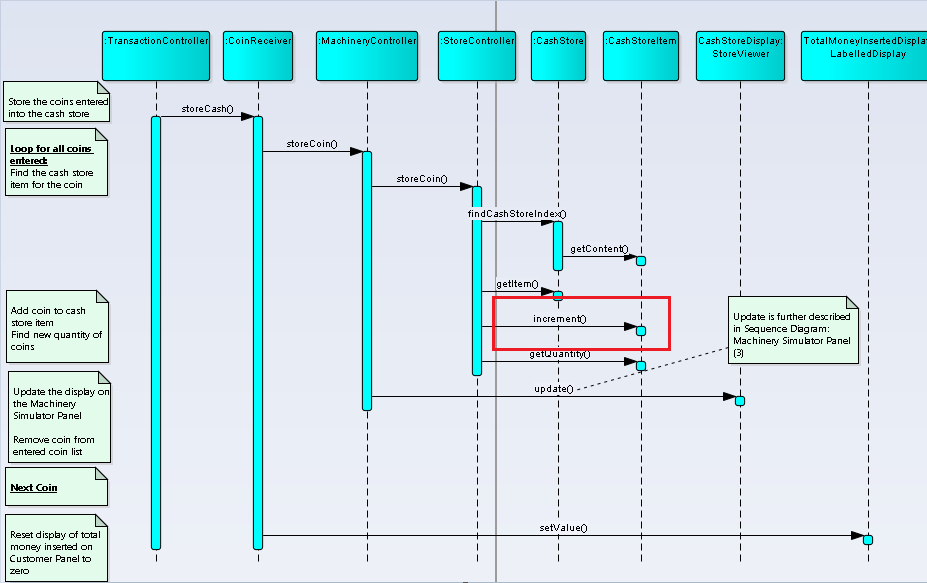
The classes and objects participating in this Observer pattern are:

* 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.

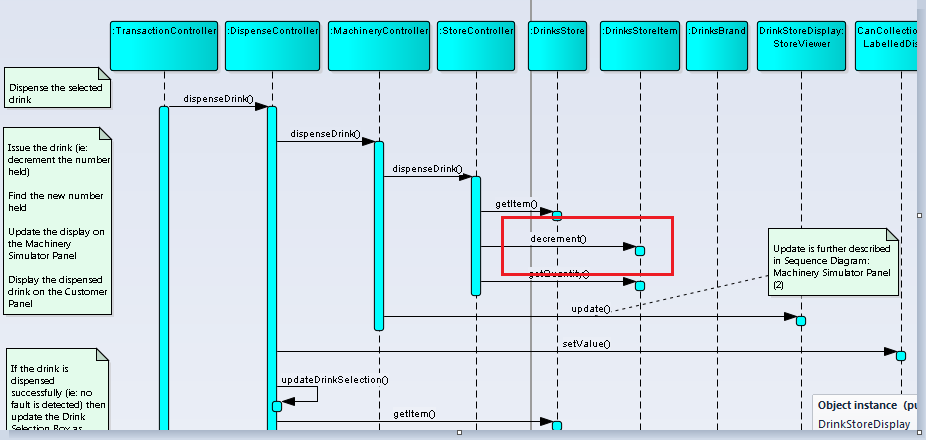
## Current Design

### Sequence Diagram : Store COINS

No update to MachineryController and MaintenanceController



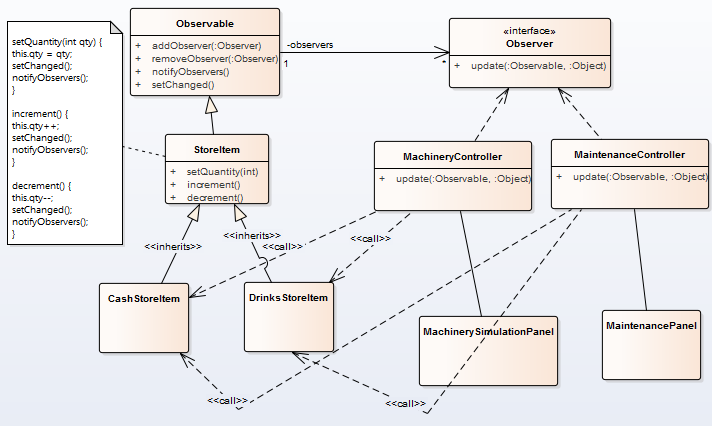
### Sequence Diagram : DISPENSE DRINK



No update to MachineryController and MaintenanceController

## Revised Design

### Class Diagram



Concrete Observer

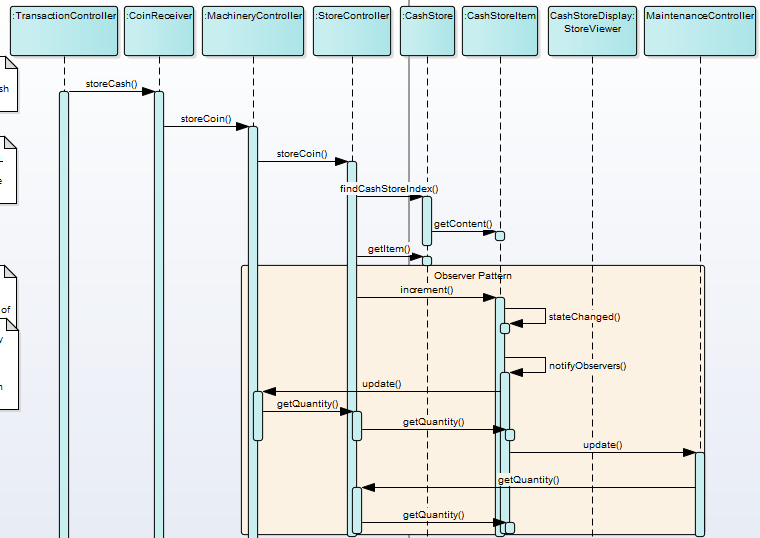
Concrete Observer

Concrete Subject

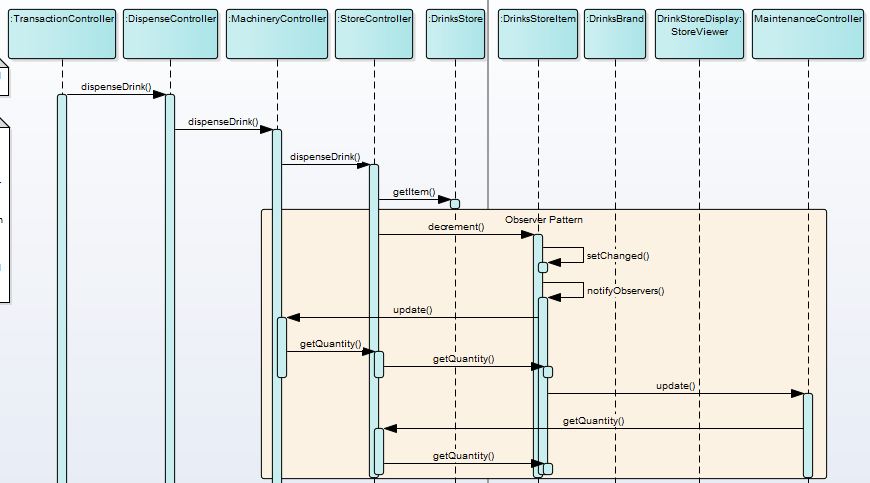
Observer

Subject

### Sequence Diagram : Store COINS



### Sequence Diagram : DISPENSE DRINK



## 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.

|  |  |  |
| --- | --- | --- |
| Deciding Factors | Factory Method | Singleton |
| Intent | Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. | Ensure a class has only one instance, and provide a global point of access to it. |
| Applicability | * A class cannot anticipate the class of object that it must create. * A class may want it's subclasses to specify the objects to be created. * A class may delegate responsibility to one of several helper subclasses so that knowledge can be localized to specific helper subclasses. | * There must be exactly one instance of a class, and when it must be accessible to clients from a global access point. * When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code. |
| Aspect | Subclass of object that is instantiated. | Sole instance of a class. |
| Decision | 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

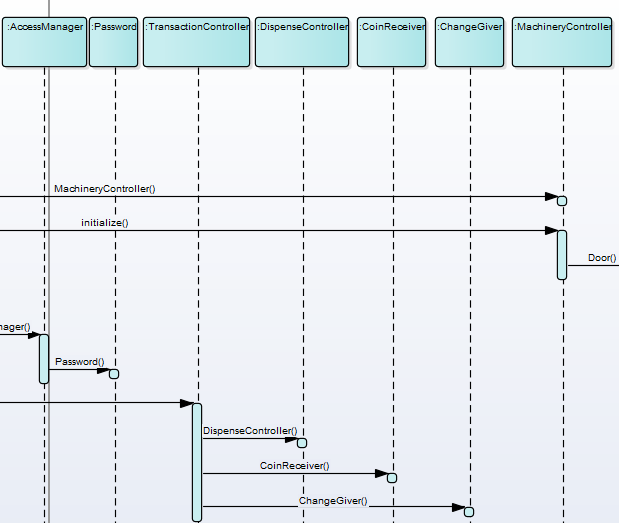
  The classes and objects participating in this pattern are:

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

## Current Design

### Sequence Diagram

Sequence diagram for Build Application (2)

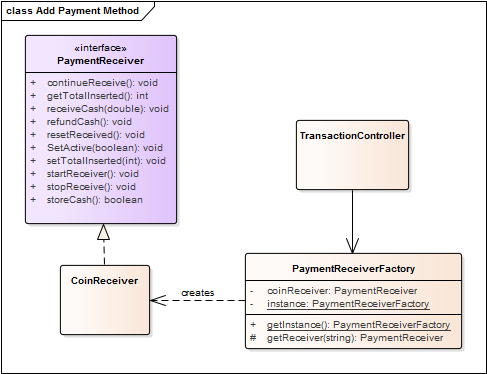


Directly initialize the CoinReceiver class.

Adding new Receiver class require the change in TransactionController

## Revised Design

### Class Diagram



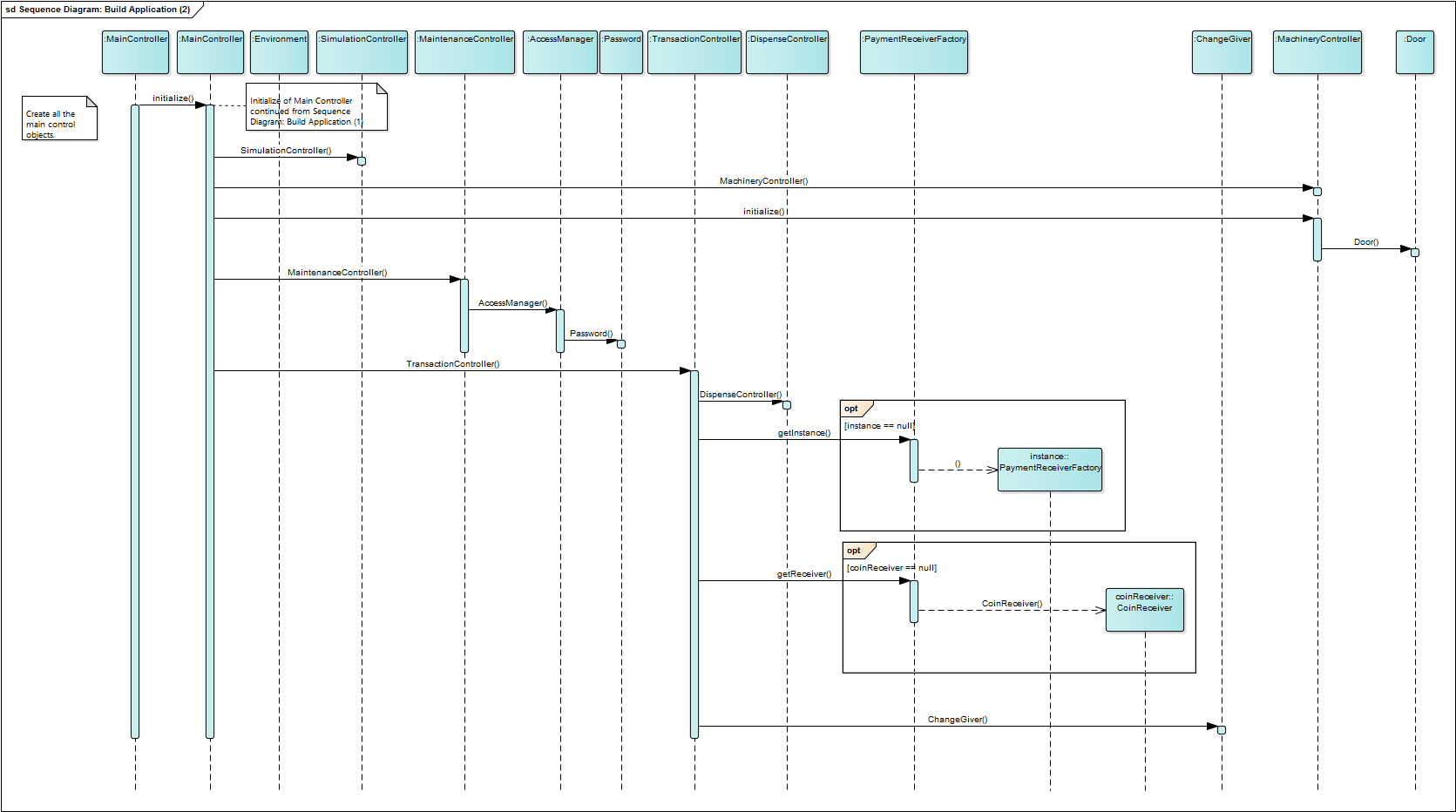
ConcreteProduct

Product

ConcreteCreator

Client

### Sequence Diagram



## Implementation Issues

|  |  |
| --- | --- |
| Issue | Rationale |
| Two Major Varieties | When the creator class is a concrete class as in our case, allows for flexibility and supports our Singleton pattern. |
| Parameterize Factory Methods | The factory can create multiple types of classes based on the parameter. This allows the creation of other receiver in the future. |
| Ensuring a unique instance | The PaymentReceiverFactory must ensure only 1 instance of the factory exists and only 1 instance of receiver is returned to the TransactionController. |
| Subclassing the Singleton Class | The PaymentReceiverFactory has no plans to be subclass and must be mindful of the consequences if it is to be subclass in the future. |
| Thread-safe implementation for multi-threading use | To ensure our PaymentReceiverFactory is thread-safe, it is instantiated when the class is loaded and not when it is first used. |

# 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.

## 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

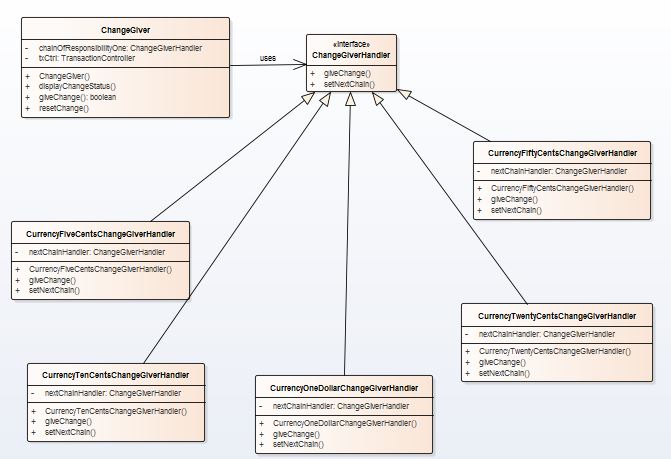
## Current Design

### Sequence Diagram

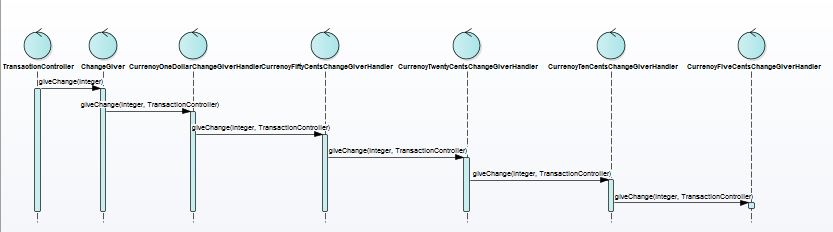


## Revised Design

### 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. |

# Bridge

## Design Problem

The vending machine currently use File Property Loader to load and initialize Cash and Drink, it does not support other loaders. For example, if the vending machine application is to be reused in another project which store all Cash and Drink information in Database instead of file.

Based on the current design structure, it is not easy to extend to support the additional ways of Property Loader feature.

## Motivation

* The ability to reuse the vending machine system to store the Cash and Drink information in various ways including file without affecting existing design.
* The Cash/Drink Property Loader implementation is closely coupled with File Property Loader implementation. Whenever a client creates a Cash/Drink Property Loader, it instantiates a concrete class that has one specific implementation (File Property Loader). This, in turn, makes it harder to port the system using other Loader implementations.

## Candidate Design Pattern(s)

The identified design problem is Structural issues therefore the following Candidate Design Patterns are from Structural:

|  |  |  |
| --- | --- | --- |
|  | Bridge | Strategy |
| Intent | Decouple an abstraction from its implementation so that the two can vary independently. | Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. |
| Applicability | * You want to avoid a permanent binding between an abstraction and its implementation. * This might be the case, for example, when the implementation must be selected or switched at run-time. * Both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently. * Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled. * You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client. | * Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors. * You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms. * An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures. * A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class. |
| Aspect | Implementation of an object | An algorithm |
| Decision | After study the candidate patterns, **the decision is to use Bridge pattern**. The reasons for choosing Bridge pattern over other patterns is because:   * It can avoid a permanent binding between an abstraction and its implementation. Property Loader should not permanent bind with File Loader Implementation. * It can hide the implementation of an abstraction completely from clients. Cash Property Loader must not know what Loader implementation is used to load Cash in system. * Changes in the implementation of an abstraction should have no impact on clients. That means adding a new Loader Implementation should don’t require any code change on Cash Property Loader. | |

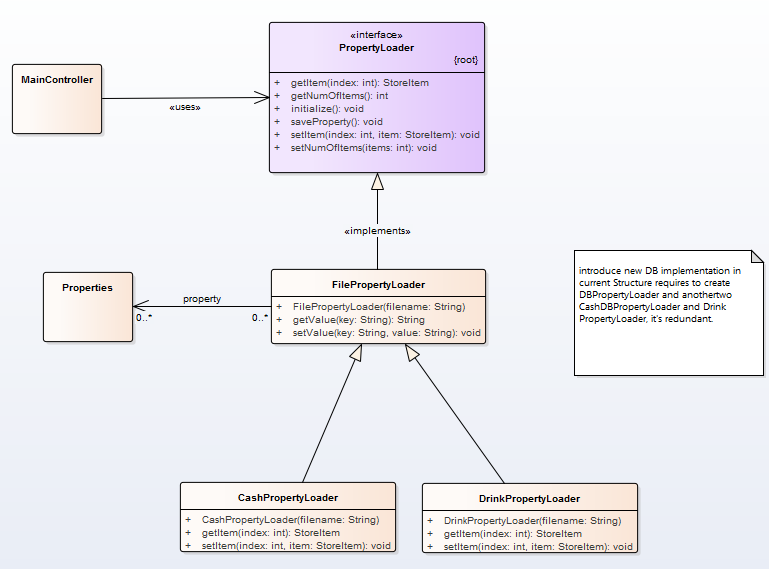
## Participants

The classes and objects participating in this pattern are:

* Client (MainController)
* It use the Abstractions
* Abstraction (PropertyLoader)
* Defines the abstraction’s interface
* Maintains a reference to an object of type Implementor
* Refined Abstraction (CashPropertyLoader, DrinkPropertyLoader)
* Extends the interface defined by PropertyLoader
* Implementor (PropertyLoaderImpl)
* Defines the interface for implementation classes. This interface doesn’t have to correspond exactly to Abstraction’s interface; in fact the two interfaces can be quite different. Typically the PropertyLoaderImpl interface provides only primitive operations (such as getValue or setValue), and PropertyLoader defines higher-level operations based on these primitives.
* ConcreteImplementor (FilePropertyLoaderImpl, DBPropertyLoaderImpl)
* Implements the PropertyLoaderImpl interface and define its concrete implementation.

## Current Design

### Class Diagram



## Revised Design

### Class Diagram

Please note that the Concrete Implementor for DB is based on future enhancement and **are only added in for illustration purpose**, it will not store or load any data from/to data base.

# 

Concrete Implementor

Concrete Implementor

Refined Abstraction

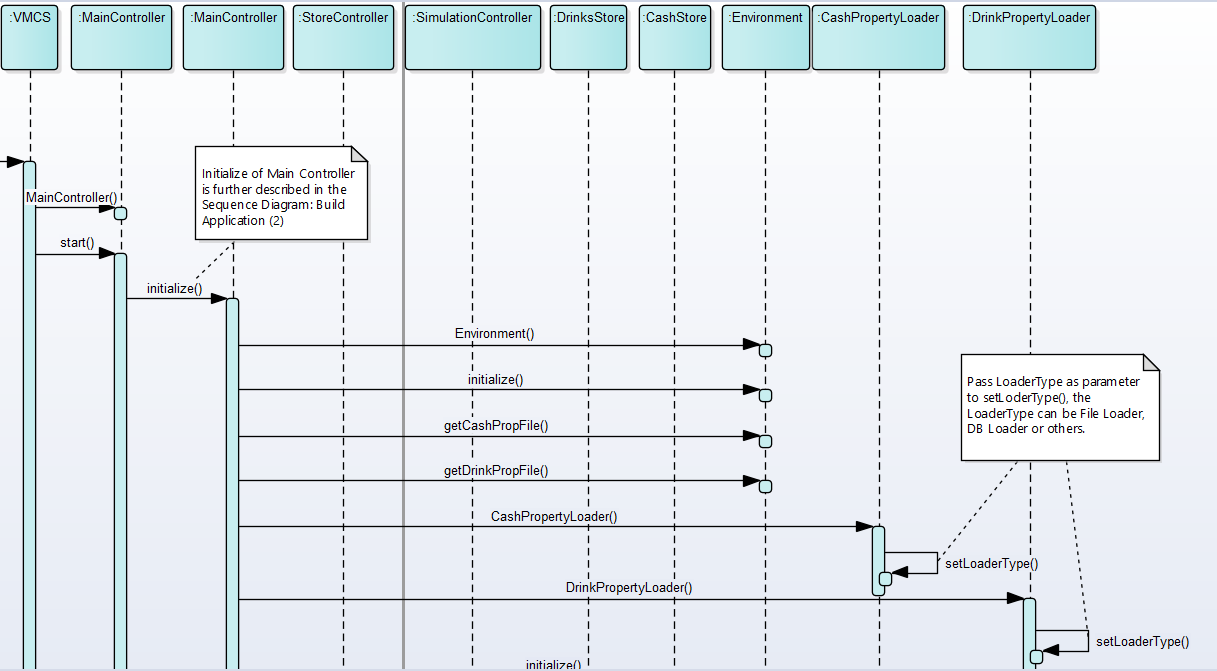
Refined Abstraction

Implementor

Abstraction

Client

### Sequence Diagram



## Consequences

Using Bridge pattern has the following consequences (positive and negative).

|  |  |
| --- | --- |
| Consequence | Rationale |
| Decoupling interface and implementation. | Decoupling abstraction and Implementor also eliminates compile-time dependencies on the implementation. Changing an implementation class doesn’t require recompiling the Abstraction class and its clients. This property is essential when you must ensure binary compatibility between different versions of a class library. |
| Improved extensibility | Can extend the Abstraction and Implementor hierarchies independently. |
| Hiding implementation details from clients. | Can shield clients from implementation details, like Cash Property Loader do not need to know where the data is stored. |

## Implementation Issues

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

|  |  |
| --- | --- |
| Issue | Rationale |
| Implementor switching at runtime cause loss of changed data | At the same running time, vending machine system is storing data in to either DB, File or other storage. Switching the Implementor at runtime will lose the changed data in prior implementor. |
| Creating the right Implementor object | How, when and where do you decide which Implementor class to instantiate when there’s more than one?  Although various implementors are provided, each vending machine system only need to instantiate one specific Implementor according to the chosen data storing method. |

# Decorator

## Design problem

The vending machine does not allow customization add-on to the drink. For example, a customer is not able to choose less sugar, less milk or less ice when buying drink. Allowing customize add-on make the vending machine more flexible and introduce more choices for the customer or even promotion like choose less sugar or milk can reduce the price of the drink.

Based on the current design structure, it does not easily extendable to support the add-on feature.

## motivation

* The ability to create new add-on without affecting existing design.
* Able to add one or more add-on after selecting a Drink before paying yet. Some add-on may have own functionality like deduct 5 cent for total price.

## Candidate Design Pattern(s)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Decorator | Bridge | Composite |
| Intent | Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. | Decouple an abstraction from its implementation so that the two can vary independently. | Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. |
| Applicability | * To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects. * For responsibilities that can be withdrawn. * When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support ever y combination. Or a class definition maybe hidden or otherwise unavailable for subclassing. | * You want to avoid a permanent binding between an abstraction and its implementation. * This might be the case, for example, when the implementation must be selected or switched at run-time. * Both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently. * Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled. * You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client. | * You want to represent part-whole hierarchies of objects. * You want clients to be able to ignore the difference between compositions of objects and individual object s. Clients will treat all objects in the composite structure uniformly. |
| Potential Design (Decorator) |  | | |
| Potential Design (Bridge) |  | | |
| Potential Design (Composite) |  | | |
| Aspect | Responsibilities of an object without subclassing | Implementation of an object | Structure and composition of an object |
| Decision | After study the candidate patterns, **the decision is to use Decorator pattern**. The reasons for choosing Decorator pattern over other patterns is because of add/remove one or more responsibilities (in this case, the add-on) to individual objects (drink) dynamically and transparently, that is, without affecting other objects. The aspect of the Decorator pattern is more closely related to the add-on feature than the rest of the candidate patterns. | | |

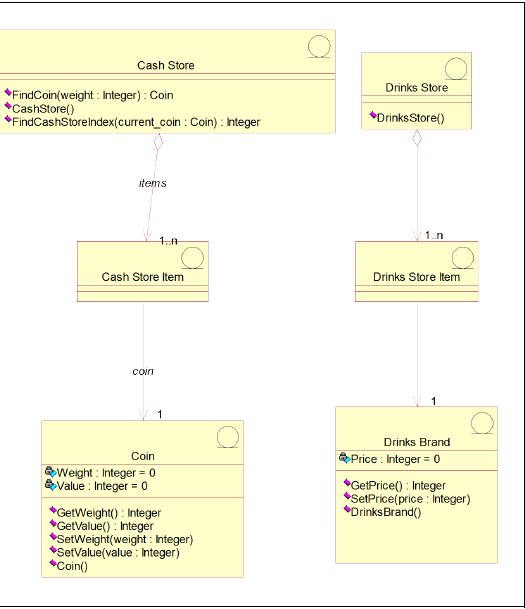
## Participants

The classes and objects participating in this pattern are:

* Component (DrinksProduct)
* The Interface defining the methods that will be implement
* Newly created Interface to support Decorator pattern
* ConcreteComponent (DrinksBrand)
* Basic implementation of the **Component**
* Decorator (DrinksDecorator)
* Decorator class implements the **Component** and it has a HAS-A relation with the Component
* ConcreteDecorator (LessSugarDecorator, LessMilkDecorator)
* Extending **Decorator** class functionality like giving discount or add-on

## Current Design

### Class Diagram



Potentially need to extend for every add-on feature.

### sequence diagram

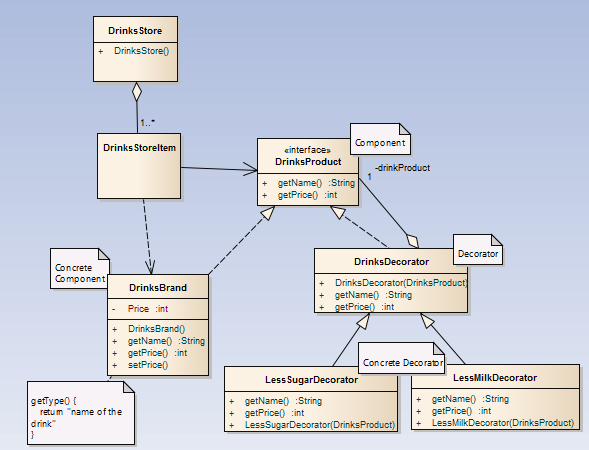


The price is retrieved so that TransactionController knows when to stop. This prevent add-on that may reduce the price

## Revised Design

### Class Diagram

Please note that the Concrete Decorator is based on future enhancement and **are only added in the test package for illustration purpose** so that it will not add to the existing design.



### sequence diagram

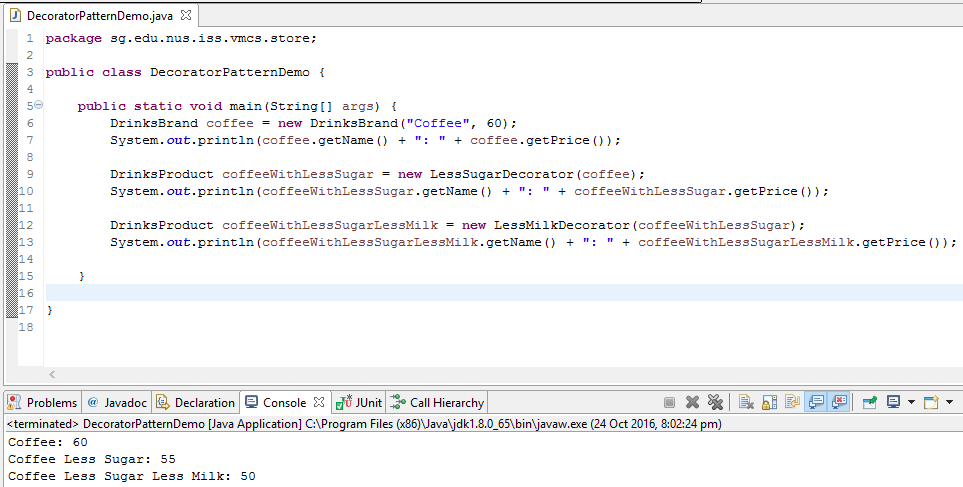
There is no change to the current diagram as adding a DrinksProduct Interface and DrinksDecorator does not change how the other object is calling DrinksBrand since there are no concrete Decorator classes in main source package yet.

No change to existing implementation.

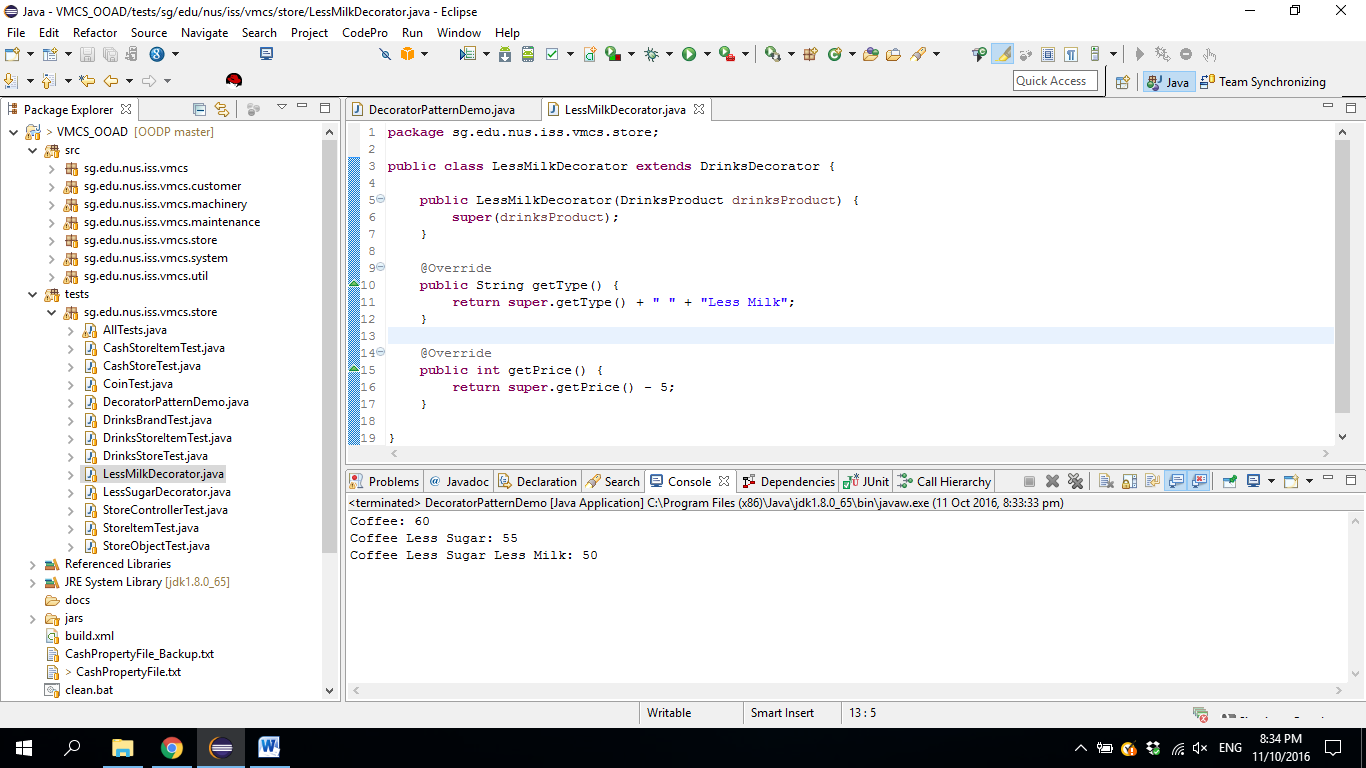


## sample code (test package)

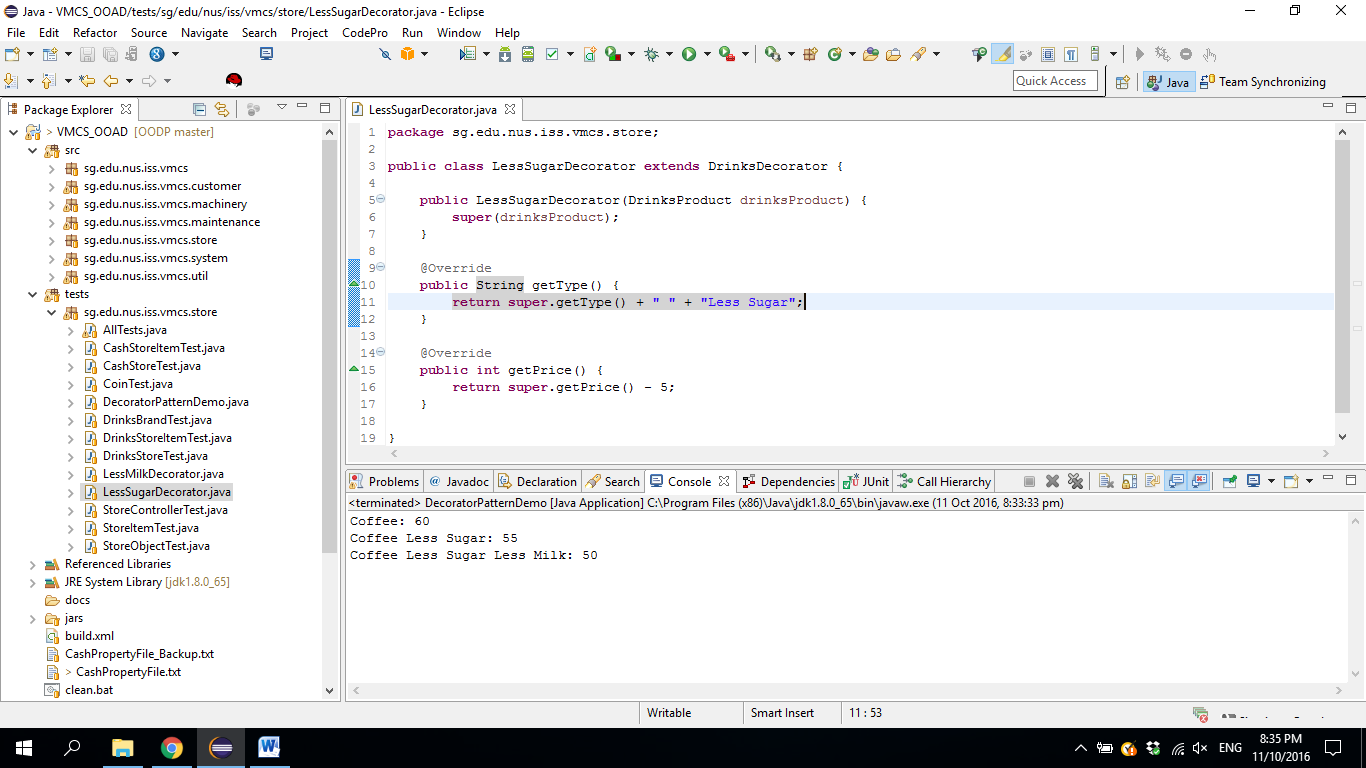
The sample code inside the test package shows how the decorator pattern is implement and the result of the pattern.

DecoratorPatternDemo.java with result

LessMilkDecorator.java



LessSugarDecorator.java



## consequences

Using Decorator pattern has the following consequences (positive and negative).

|  |  |
| --- | --- |
| Consequence | Rationale |
| More flexibility than static inheritance | Decorator pattern allows add-on objects to be added or removed at run time based on the customer selection whereas static inheritance requires creating a new class for each additional add-on. |
| Avoids feature-laden classes high up in the hierarchy | Decorator pattern allows adding functionalities in the Decorator object easily compared with extending complex classes. Potential functionalities can be added such as discount for less sugar/milk or customer with membership. |
| Potential lots of little objects | If there is many different type of add-on, it could result in lots of little objects. The potential add-on objects are:   * Less sugar * More sugar * Less milk * More milk * Less ice * Less coffee * More coffee * No ice |

## implementation

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

|  |  |
| --- | --- |
| Issue | Rationale |
| New Interface to represent drink object and extends by DrinksDecorator class versus DrinksDecorator just extends DrinksBrand. | One of the considerations to implement Decorator pattern is that “a Decorator object’s interface must conform to the interface of the component it decorates”. Existing design has DrinkBrand class that can be represented as “component it decorates” for DrinksDecorator class to implement. Another option is to create a new Interface that implement by both DrinksBrand and DrinksDecorator.  After consider the two options, the decision is to create a new Interface (named DrinksProduct) because in the event current DrinkaBrand class has many methods that the Decorator class is not interested to know. |
| DrinksDecorator created as abstract class. | DrinksDecorator is implemented as abstract class even though potentially only have two methods, getPrice() and getType(), because need to prevent a null object is passed into DrinkDecorator constructor. |
| Keeping DrinksBrand (component) classes lightweight. | Because both DrinksBrand (component) and DrinksDecorator (Decorator) must inherit from a common Component class (named DrinksProduct), it can potentially make DrinksBrand contains lot of functionality that does not require because of implementing the Decorator pattern. In order to make it as lightweight as possible, it is considered to make subclass of DrinksBrand but for now, the design will remain as it is. |
| Create DrinksDecorator (or add-on) immediately (runtime) post-effect | Each time a customer select an add-on, the system should create a new DrinksDecorator's implementation (based on the selected add-on) for dynamic/run-time reduced price and its content (lesser milk, suger and etc) but this can lead to issue about desselect the add-on before paying.  One way to overcome this issue could be by using memento pattern and on the UI in order to deselect the add-on, the customer can only choose "undo" that the system will know how to deselect the previous add-on. However this design is not friendly as a customer can first select "Less Sugar" then "Less Milk" but decided to deselect "Less Sugar" and the proposed design (memento) can only undo/deselect the last add-on ("Less Milk") then undo/deselect to desire add-on ("Less Sugar"). |
| Drink that does not need decorator (or add-on functionality) | Drink like soft drink, milk does not need to have any add-on and using Decorator pattern might accidentally make the “Milk with less coffee”. To prevent this scenario to happen, it is decide to add more validation on the GUI to prevent it from happening. |

# Template method

## design problem

* StoreController has dependency to DrinkPropertyLoader, CashPropertyLoader,CashStore and DrinkStore
* StoreControlelr implemented both Cash and Drink related methods:
  + Cash:
    - getTotalCash()
    - giveChange()
    - storeCoins()
    - transferall()
  + Drink
    - dispenseDrink()
    - setPrice()
  + Common methods
    - saveProperties()
    - initializeStores()

## motivation

Anything changed in Store/Cash store, there will be impact to StoreController.

## Candidate Design Pattern(s)

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

|  |  |  |
| --- | --- | --- |
|  | **Template** | **Strategy** |
| Intent | Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. | Define a family of algorithms, encapsulate each one, and make them interchangeable.  Strategy lets the algorithm vary independently from clients that use it. |
| Applicability | • To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.  • When common behavior among subclasses should be factored and localized in a common class to avoid code duplication. This is a good example of "refactoring to generalize" as described by Opdyke and Johnson [OJ93].  You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.  • To control subclasses extensions. You ca n define a template method that calls  "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points. | Many algorithms exist for breaking a stream of text in to lines. Hard-wiring all such  algorithms into the classes that require them isn't desirable for several reasons:  • Clients that need line breaking get more complex if they include the line breaking code. That makes clients bigger and harder to maintain, especially if they support multiple line breaking algorithms.  • Different algorithms will be appropriate at different times. We don't want to support multiple line breaking algorithms if we don't use them all.  • It's difficult to add new algorithms and vary existing ones when line breaking is an integral part of a client.  We can avoid these problems by defining cl asses that encapsulate different line breaking algorithms. An algorithm that's encapsulated in this way is called a strategy. |
| Aspect | steps of an algorithm | an algorithm |
| **Decision** | **After study the two candidate patterns, the decision is to use Template pattern. The reason for choosing Template pattern over Strategy pattern is because strategy pattern is more like to solve if else scenario, in this case both CashStoreController and DrinkStoreController to need to be instantiated.** | |

## particpants

  The classes and objects participating in this pattern are:

* **AbstractClass (StoreController)**

This class has been made as abstract class. Implements only common method across both Cash and Drink store

* **ConcreteClass (CashStoreController)**

This is concert controller to handle cash store specific methods.

* **ConcreteClass (DrinkStoreController**)

This is concert controller to handle drink store specific methods.

## Current Design

### Class Diagram

## 

### sequence diagram

StoreController has tight dependency with both Cash and Drink related classes

### 

## Revised Design

### Class Diagram



### sequence diagram

CashStoreController has dependency with both Cash related classes only

### 

## implementation

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

|  |  |
| --- | --- |
| Issue | Rationale |
| Minimizing primitive operations | An important goal in designing template methods is to minimize the number of primitive operations that a subclass must override to flesh out the algorithm. The more operations that need overriding, the more tedious things get for clients.  After consider the two options, the decision is to make StoreController as abstract class, because in the event current StoreController class has many methods DrinkStoreController/CashStoreController do not need to customize. |
| Single responsibility | Moved Drink related dependencies into DrinkStoreController, and Cash related dependencies into CashStoreController. Therefore, any changes in DrinkStore will not have impact to CashStore and vis-à-vis. |

# individual contribution

|  |  |
| --- | --- |
| Name | Design Pattern |
| Ng Shen Soon Benjamin | Decorator Pattern |
| Koo Sheng Kiat | Factory Method |
|  |  |
|  |  |
|  |  |
|  |  |

# Reference List

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