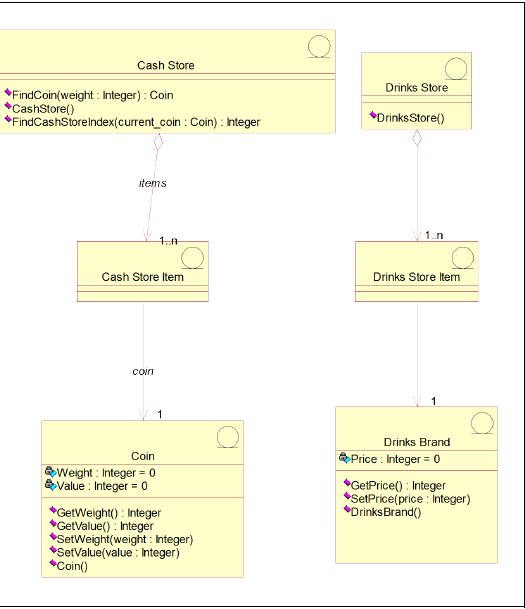
# Design Problem

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.

# Current Design (Class Diagram)



Potentially need to extend for every add-on feature.

# Current Design (Sequence Diagram)



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

# Candidate Design Pattern Considered

The identified design problem is a Structural issue therefore the following Candidate Design Pattern are from Structural:

|  |  |  |
| --- | --- | --- |
|  | **Decorator** | **Bridge** |
| 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. |
| 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 * For responsibilities that can be withdrawn.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. |
| Potential Design |  |  |
| Aspect | Responsibilities of an object without subclassing | Implementation of an object |
| Decision | After study the two candidate patterns, **the decision is to use Decorator pattern**. The reason for choosing Decorator pattern over Bridge pattern 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 Bridge pattern. | |

# 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
* ConcreterDecorator (LessSugarDecorator, LessMilkDecorator)
* Extending **Decorator** class functionality

# 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.

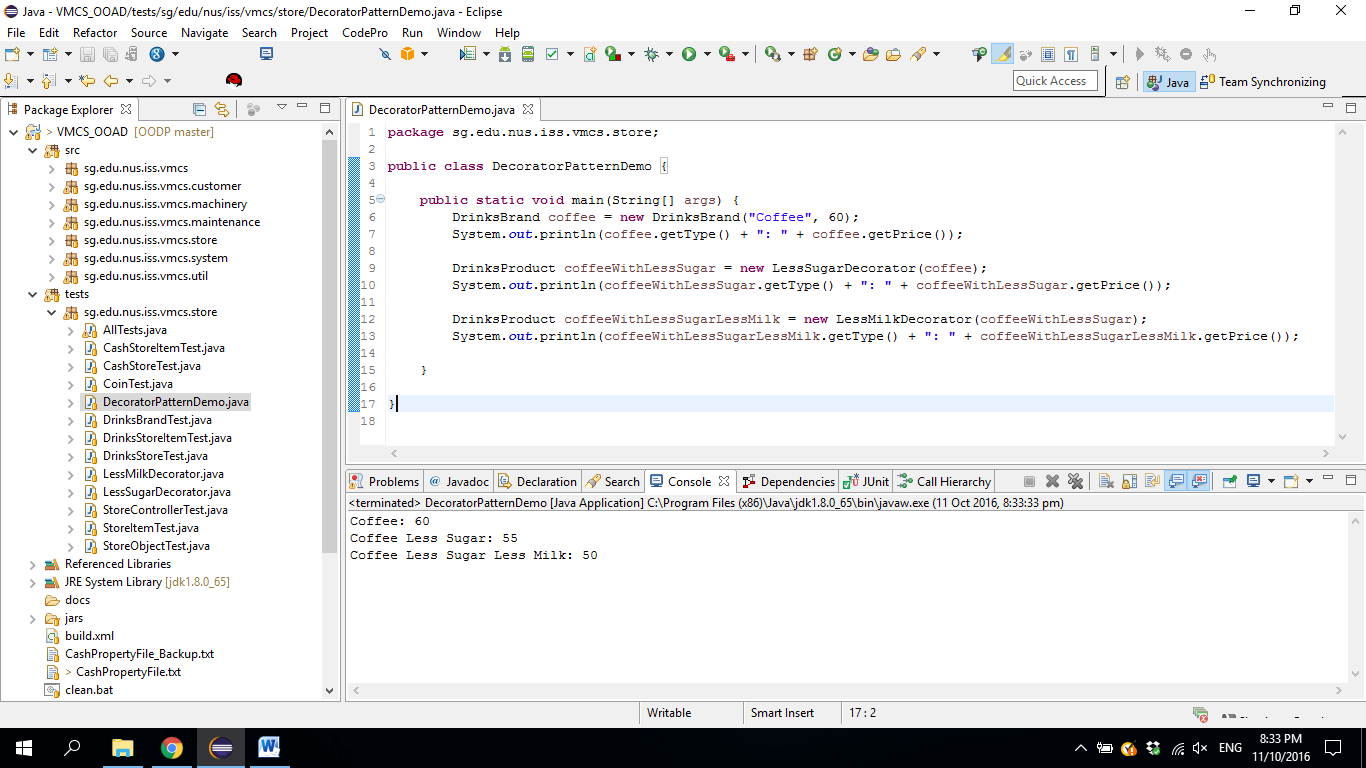
# 

# Revised Design (Sequence Diagram)

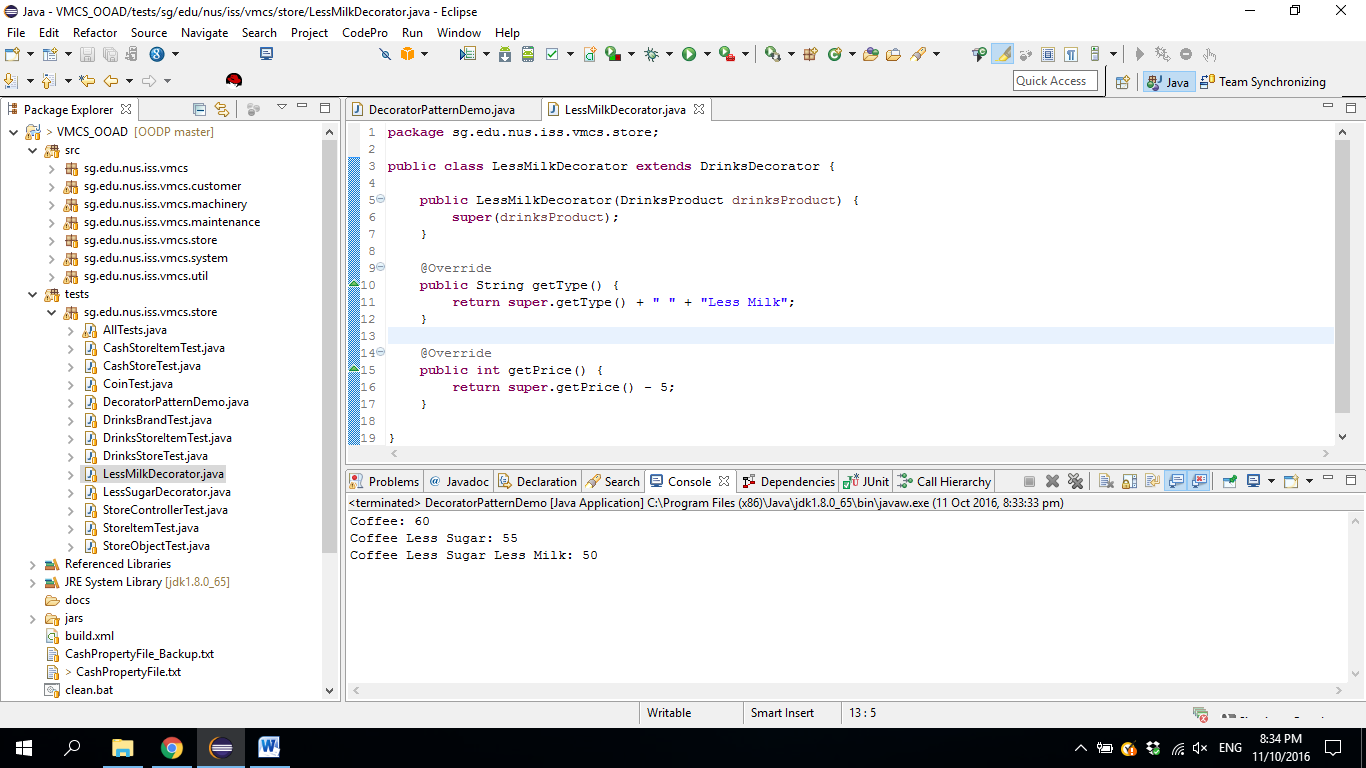
There is no change to the current diagram as adding a DrinksProduct Interface and DrinksDecorator does not change how the other object is calling DrinksBrand since there are no concrete Decorator classes in main source package yet.

# Sample code (test package)

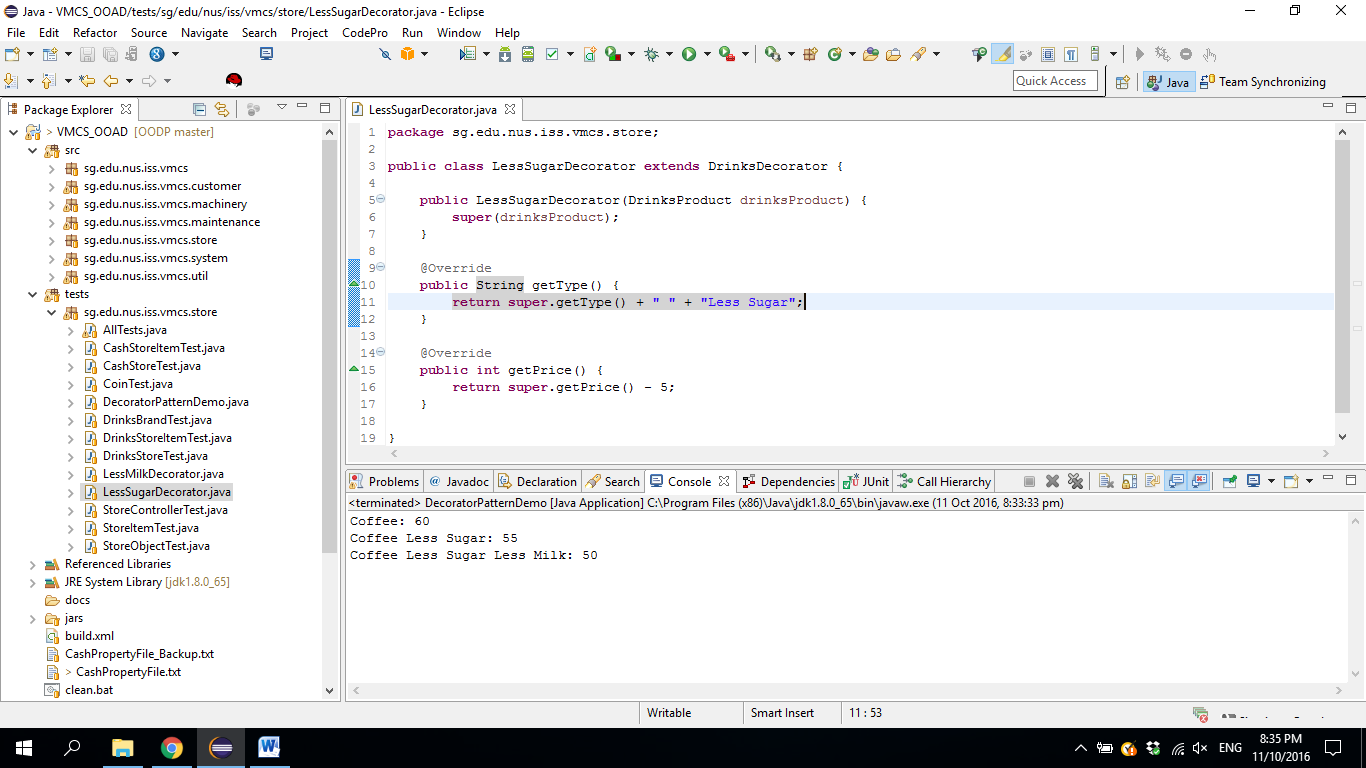
The sample code inside the test package shows how the decorator pattern is implement and the result of the pattern.

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 |
| 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. |

# 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. |