# Description of 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 Considered

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.** | |

# Participants

  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.

# Diagrams

## Class Diagram

Before



Figure 1: Class Diagram for Template Pattern

## After



## Sequence Diagram

StoreController has tight dependency with both Cash and Drink related classes

Before



Figure 2: Sequence Diagram for old design

After

CashStoreController has dependency with both Cash related classes only



Figure 3: Sequence Diagram for new design (Template Pattern)

# 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. |