Super Market Checkout

Application Program – Technical Document

Contents

[1. Introduction 3](#_Toc398468405)

[2. Requirement 3](#_Toc398468406)

[3. Component Structure 3](#_Toc398468407)

[4. How it works 4](#_Toc398468408)

[5. Flow Diagram 5](#_Toc398468409)

[6. Implementation Details 5](#_Toc398468410)

[7. Junit Test Program 6](#_Toc398468411)

[8. Checkout – Output – Command line 7](#_Toc398468412)

[9. Extract zip file & Run the program 7](#_Toc398468413)

[10. Enhancements 7](#_Toc398468414)

[11. Environment 7](#_Toc398468415)

# Introduction

Objective of this document is to detail the classes & objects and technical logic deployed in implementing the requirements and accomplishing the goal. This document also details some generic way of completing the requirements as well.

# Requirement

A Supermarket sells 3 products listed below:

Product A = $20,

Product B = $50 (or 5 for the price of 3)

Product C = $30

Implement the code for a checkout register that calculates the price of a given sequence of items. The input is a product list as a String, e.g "ABBACBBAB" : for which the output should be the integer 240. Please consider testability, documentation, and other good coding practices in your solution. As an additional challenge, consider how new pricing rules might be provided programmatically.

Implement the following:

public interface Supermarket {

public int checkout(String items);

}

# Component Structure

Following components are created. These are packed under the name **‘shop’**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| Product | Class | Independent entity type of class having *name & price* as properties |
| ProductOrder | Class | Class for wrap-up properties for the *Product* including *quantity, offer, baseQuantity, offerQuantity* |
| ProductFactory | Class | Class for creating new Product object for the given values – *name and price.* And also, it can keep track of total number of Product objects created, irrespective of the type (that is, *name)*  Included only for the separation of creation of Product object |
| SuperMarket | Interface | Single interface with method for final checkout – calculates & returns the total amount for the items (products) selected |
| SuperMarketImpl | Class | Implements *SuperMarket* interface with required methods and properties/fields  Core class for the functionality implementation – applying pricing, considering product offers if any, calculating final price for the check-out |
| ShoppingHandler | Class | Interceptor class from *ShoppingClient* to the *SuperMarket interface.* It creates *SuperMarketImpl object and*  to just call methods in *SuperMarketImpl* and pass back the results to the Client.  Gets the values for *items, prices, rules* from the Client and initializes *items,* injects *prices & rules* into *SuperMarketImpl* |
| ShoppingClient | Class | Main class for running the program. It accepts command line inputs for the *prices, rules* and *item sequence* |

# How it works

*ShoppingClient* is the starting point as it is the Main class. In the class, by default, the *prices, rules & items string* are defined. However, user is given an ability to enter his/her own string of items, prices and rules.

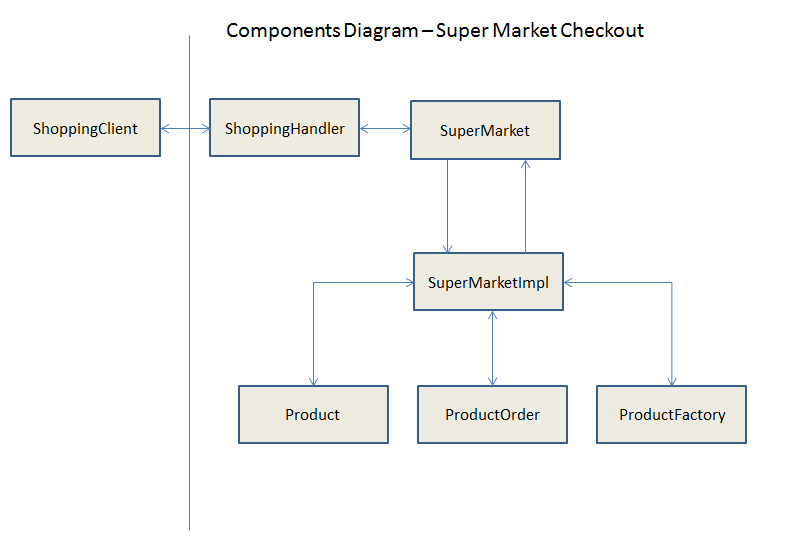
Assumptions:

1. Items: The items will always have the combination of three alphabets – A, B & C. This means, *items* could be ‘ABCAB’, or ‘ABBCABBCC’ and so on. The default *items* string is: ‘ABBACBBAB’ (this is the items in the requirement)
2. Prices: By default the prices are defined in this order: A 🡪 $20, B 🡪 $50, C 🡪$30. However, user can enter new pricing as well in the command line input
3. Rules: This means, if there is any ‘offer’ for any particular item type, then the rule will be applied in calculating the total price for the particular item. That is, if for ‘B’, it is specified as *5 for the price of 3’,* then, for the total price of 5 ‘B’ items, only 3 ‘B’ items is considered; however, the total quantity would be 5. The offer is considered only in calculating the prices. User can also give choice for which particular item would have the offer, and how it needs to be included.

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Interpretation** |
| Items: ABBACBBAB  Prices: 20 50 30  Rules: B 5 3 | 240 (3\*20 + 3\*50 + 1\*30) | A’s price is $20, B’s price is $50 and C’s price is $30 with offer on B – 5 for the price of 3 |
| Items: ABBACCBA  Prices: 20 50 30  Rules: A 3 2 | 250 (2\*20 + 3\*50 + 2\*30) | A’s price is $20, B’s price is $50 and C’s price is $30 with offer on C – 3 for the price of 2 |

Please note that, the Rule is applied only for a SINGLE item type – either ‘A’ or ‘B’ or ‘C’, not for all and not for two either. And further, only these three are the allowed type for creating the combination of input *items* string.

# Flow Diagram



# Implementation Details

The class *SuperMarketImpl* is the core for this task. We describe it’s details here as it implements the required business logic.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Input** | **Output** |
| SuperMarketImpl() | Public Constructor | String price[],  String rules[] | Void |
| Used to initialize the prices and rules for the subsequent operations. This constructor is invoked when an interface type of *SuperMarket* is created in *ShoppingHandler* class | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Input** | **Output** |
| createOrders() | Private Method | String prodSeq | Void |
| 1. This method accepts input item sequence in the name of ‘prodSeq’. It creates three separate ‘Product’ objects of specific type. 2. It loops through the given item sequence string. It checks if a character is ‘A’. If so, create a corresponding ProductOrder object to hold only the type of Product object with the name ‘A’. Similarly, for ‘B’ and for ‘C’. In this way, we can have a dedicated ProductOrder for every Product, keeping track of number of such Products, it’s base quantity, it’s offered quantity (via rules) | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Input** | **Output** |
| calculateOfferCount() | Private Method | ProductOrder | Integer value |
| This method is used to calculate the quantity of a particular item, based on the rules (offers) and returns the count that would be in-turn used for the calculation of total price for this product. It does not alter the total number of such products created. The return value is used only for the calculation of the price | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Input** | **Output** |
| calculateOrderPrice() | Private Method | Void | Void |
| This method internally filters the calculation based on the quantity condition such that, the base quantity should be less than or equal to the total number of products quantity for a particular type. And it should qualify for getting the offer.  For an instance, for the type ‘A’,  **int** quantityA = orderA.getProducts().size();  **if**(orderA.isHasOffer() && (orderA.getBaseQuantity() <= quantityA)) | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Input** | **Output** |
| Checkout() | Public Method | String items | Integer – total price |
| This is the method implemented (declared in the base interface) with @Override annotation. This returns the final amount that the client has to pay for the shopping he/she did with the given input items sequence | | | |

# Junit Test Program

Created Junit Test suite with a class to test and assert the method return value. This is packaged in the name **‘test’**

**package** test;

**import** shop.ShoppingHandler;

**import** org.junit.Test;

**import** **static** org.junit.Assert.*assertEquals*;

**public** **class** SuperMarketTest {

String items = "ABBACBBAB";

String price[] = {"20", "50", "30"};

String rules[] = {"B", "5", "3"};

ShoppingHandler shoppingHandler = **new** ShoppingHandler(items, price, rules);

@Test

**public** **void** testCheckOut()

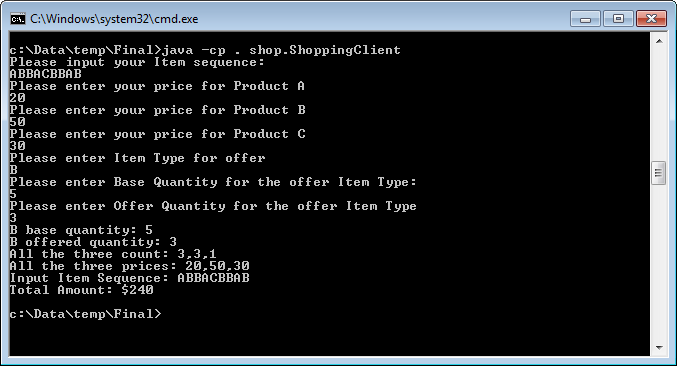
{

*assertEquals*(240,shoppingHandler.checkout());

}

}

# Checkout – Output – Command line



# Extract zip file & Run the program

Unzip the Checkout.zip file. The unzipped folder would have the following structure

*Shop –* directory having required java sources

*Checkout.*bat – Windows batch file – double click on this to run the program

*SuperMarketCheckout\_TechnicalDocument.doc -* This document as well

# Enhancements

The input pricing, rules details can also be retrieved from a file. An input file containing this information can be created and the details can be modified later when needed.

# Environment

This task has been implemented using : JDK1.7, Junit 4.4.jar, Eclipse Helios, Windows XP