Supermarket Pricing

TABLE OF CONTENTS

Test & Build the Application

To test and build the application, run the following command:

```
mvn clean test package
```

Modeling

Given the instructions from Dave Thomas, this kata is greatly focused on modeling, so first of all follows a class diagram of my solution:

I would like to emphasize that I didn't come up with this diagram before I implemented the code. I applied TDD and iteratively refined the model, which has been shaped by tests into its final form.

Follows also my comments on the requirements and questions raised by Dave:

> three for a dollar (so what's the price if I buy 4, or 5?)

My model uses the strategy pattern to implement different promotional offers. The concrete implementation of `PromotionalOffer` named `XForPriceOffer` addresses this requirement.

I believe that my design respect all S.O.L.I.D principles:

- All my methods are small and only do one thing. Cyclomatic Complexity in my code is low (if one runs SonarQube, which I did using my own local SonarQube instance, more on that in the next section).
- Using the strategy pattern and dependency inversion created an open-close design.

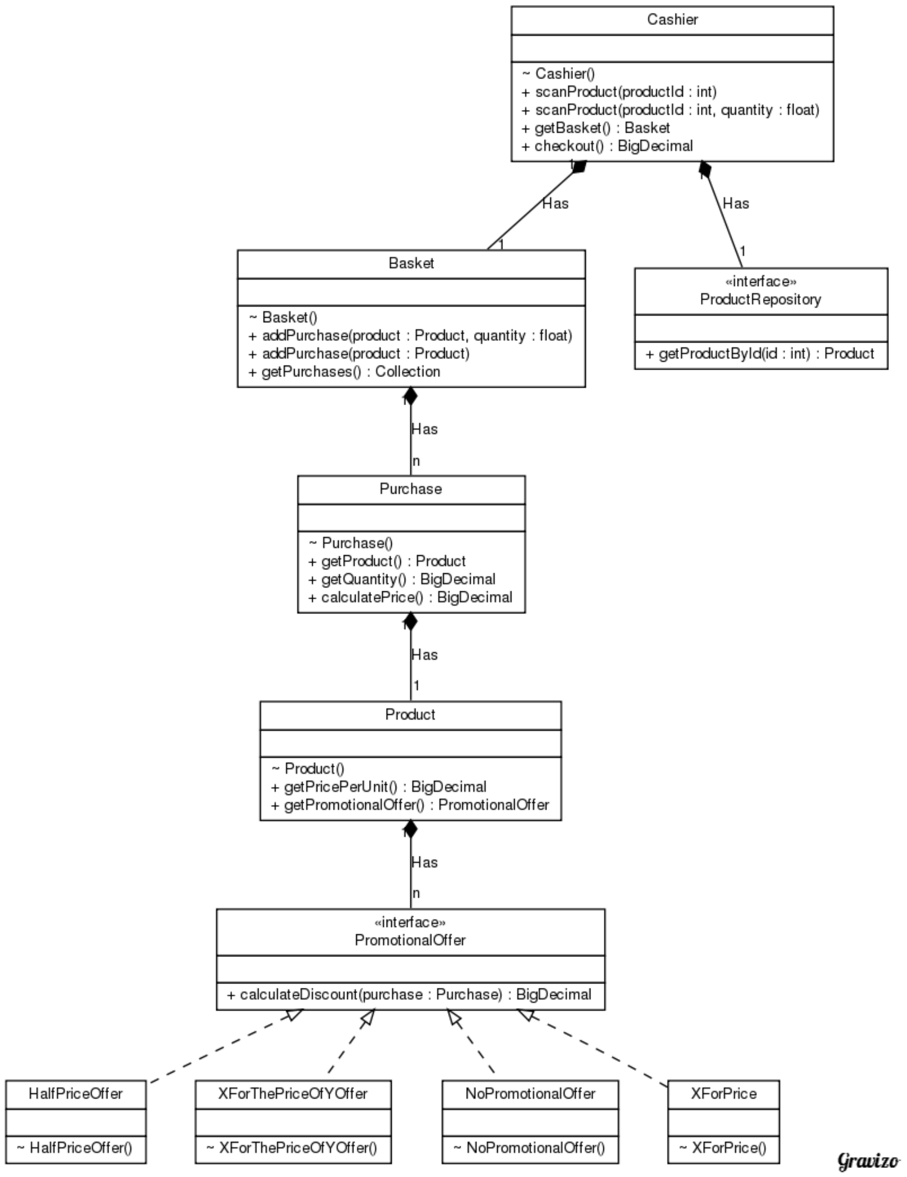
Cashier
~ Cashier()
+ scanProduct(productId : int)
+ scanProduct(productId : int, quantity : float)
+ getBasket() : Basket
+ checkout() : BigDecimal
Has
1
1
Has
1
1
Basket
~ Basket()
+ addPurchase(product : Product, quantity : float)
+ addPurchase(product : Product)
+ getPurchases() : Collection
«interface»
ProductRepository
+ getProductById(id : int) : Product
1
Has
n
Purchase
~ Purchase()
+ getProduct() : Product
+ getQuantity() : BigDecimal
+ calculatePrice() : BigDecimal
1
Has
1
Product
~ Product()
+ getPricePerUnit() : BigDecimal
+ getPromotionalOffer() : PromotionalOffer
1
Has
n
«interface»
PromotionalOffer
+ calculateDiscount(purchase : Purchase) : BigDecimal
HalfPriceOffer
~ HalfPriceOffer()
XForThePriceOfYOffer
~ XForThePriceOfYOffer()
NoPromotionalOffer
~ NoPromotionalOffer()
XForPrice
~ XForPrice()
Gravizo

Figure 1: Supermarket Pricing Class Diagram

- I'm not using inheritance (one should favor composition instead for low decoupling).
- My interfaces are small and my concrete implementations actually implement all their methods (no dummy concrete implementations for any method).
- I have used Dependency Inversion for the product repository, which needs to adhere to an Interface (`Product Repository`) defined by `Cashier`. For a great article on dependency inversion, refer to this link. Nice for tests as well. I have a simple implementation named `TestProductRepository` which is implemented using a `HashMap` and used only for test purposes.

> $1.99/pound (so what does 4 ounces cost?)

I didn't feel like this requirement needed special treatment. A product has a price per unit, which can be any, e.g., unit free (count), pounds, ounces, etc. Note also that quantity in a purchase is a floating-point number, therefore half a pound would be a valid quantity, which not necessarily needs to be a integer (one bar of chocolate, one kilogram of flour, etc).

The units of measurement can be addressed in the description field. I also didn't feel the need for units of measurement having a separated field. Following the Agile way, I tried to come up with the simplest design that addresses the current requirements.

> buy two, get one free (so does the third item have a price?)

The concrete implementation of `PromotionalOffer` named `XForThePriceOfYOffer` addresses this requirement. One could have 3 for the price of 2, 5 for the price of 4, etc.

> does fractional money exist?

In principle one could express any price in the smallest unit of a given currency, e.g., cents, and all prices could be represented by integers.

But given that you could have a product priced with 1 Euro per pound and the customer could want to buy a third of a pound, the amount to pay would be a fraction that would have to be rounded in some manner to an integer (cents in this example).

I could use integers in my model (representing cents for instance), but I believe that is much easier and intuitive to use `BigDecimal` which is Java native.

> when (if ever) does rounding take place?

I believe my previous example addresses this scenario. For practical reasons, I'm rounding when prices and quantities are passed to create products and purchases and when the final total is calculated on checkout.

If one doesn't round up when creating a product, `BigDecimal` will round-up numbers that can't be quite represented well in floating-point manner and generate discrepancies (try to create an `BigDecimal` with the value `0.4f`, for instance, and multiply it by `4`, it will result in `1.64f`).

> how do you keep an audit trail of pricing decisions (and do you need to)?

This would be useful for auditing stock and sales along a period of time, probably a must for taxing purposes.

I believe that this is separated from the main feature (purchases), but a concrete `ProductRepository` implementation that would be fit for production could save the history of purchases and price (even the whole product) changes in a database under the hood in a manner that is transparent to `Cashier`. I'm not going to implement this here (a kata is supposed to be short).

> are costs and prices the same class of thing?

A little confused here given the context. From the point of view of a customer, given that the requirements don't mention tax calculation (the customer don't have to pay tax on purchase), I would say that cost and price are the same thing.

> if a shelf of 100 cans is priced using "buy two, get one free", how do you value the stock?

That's a bit tricky question. You would need to know the distribution of number of cans bought by customer.

One could guess that is binomial, centered in some number of cans N and then estimate the value of the stock.

I would say that finding the best distribution would take some experiment. One could probably use the purchase history for this product to estimate the shape of the distribution. I believe that the actual value of the present stock cannot be deterministically determined though.

Code Coverage

I have added the JaCoCo plugin to this project.

JaCoCo reports can be found at target/site/jacoco/index.html if one builds and tests the package.

Note that all code used to implement the kata has 100% test coverage. The exception would be for the POJO's that use Guava to implement `toString()`, `hashCode()` and `equals()`, which are only useful for testing and debugging (they are only called if tests fail at the present moment).

I have also a local instance of SonarQube installed in my local machine. The report follows as a PDF file named `SonarQube_-_supermarket-pricing.pdf` in the project root directory.

Note that has no issues, duplication, etc, and the complexity is low. No package dependency cycles as well.

REST API

CRUD could be implemented using REST to create the product catalog:

```
POST catalog/product
{ "product":
    { "description": "some product description",
      "price": 1.23}
    }
}
```

Response:

```
{ "productId": 123 }
201 (CREATED)
```

`POST` could be used for creating, `PUT` for updating, `GET` for selecting and `DELETE` for deleting products.

We could create a basket history for a given customer, add purchases and then perform a checkout operation over the resulting basket:

```
POST cashier/basket/create
```

Response:

```
{ "basketId": 1234567890 }
201 (CREATED)
```

Add purchases to the basket:

```
POST cashier/basket/1234567890/purchase
{ "productId": 123, "quantity": 1.23 }
```

Remove a purchase from the basket:

```
DELETE cashier/basket/1234567890/purchase/123
```

Checkout the basket:

```
POST cashier/basket/1234567890/checkout
```

Response:

```
{ "total": 12.34 }
200 (OK)
```

Cancel the whole basket:

```
DELETE cashier/basket/1234567890
200 (OK)
```

I guess I could implement this more or less easily using the Play Framework, even though I have only used Play with Scala (sample project available in my GitHub account).

But the spec doesn't allow the use of any frameworks (including ORM and REST).

Final Notes

Note that I'm not checking for null anywhere. According to clean code principles, defensive programming is harmful and creates boilerplate code. Given that this is an internal API, one should trust that the API methods will be called with the proper parameters by members of your team.

I'm validating parameters for `XForThePriceOfYOffer` (`X` needs to be greater than `Y`) for instance, as an incorrect configuration by an end-user could happen.

In case one is unable to visualize this markdown document properly from IntelliJ (a markdown plugin is required, I recommend Gfm), there is a PDF version named `README.pdf` on the root directory of this project. Specially the Class Diagram generated with Gravizo might be useful to understand the design.