Supermarket Billing Software

Most of the supermarkets use software for preparing the final bill before the goods are packed. This software itself can very easily be designed by someone starting in Java (Note that most of the actual software used for billing in supermarkets use Java itself). This billing software sums the amount for individual items and then adds them up to get the final sum that the customer has to pay. The number of individual items, the price of each item should be editable for the user. Also, there should be an option to remove the item from the list altogether.

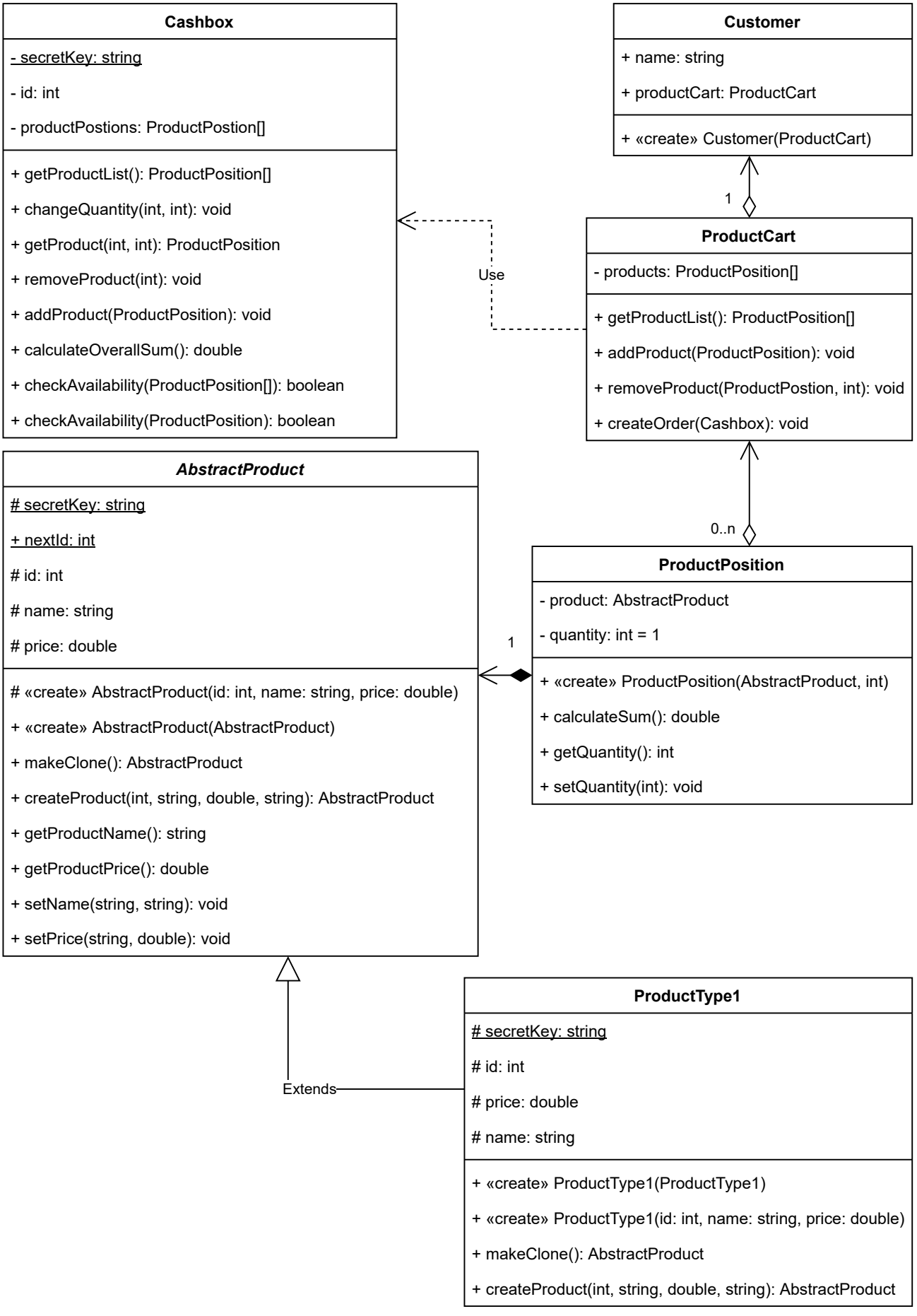
We should choose the only design pattern:

Factory Method | Abstract Factory | **Prototype** | Singleton

Requirements:

- For the UML class diagram: the fields / methods are implemented logically according to the feature (you don't have to define a lot of fields/methods, define only those that are necessarily obvious). And when you are defining primitive types, the types should be consistent with: https://www.uml-diagrams.org/data-type.html
- For the UML class diagram: the relationships between classes should also be defined (if there is an association then define it). Look at here: https://www.uml-diagrams.org/class.html. When there is a relationship between classes, you should also indicate it in your class fields/methods (for example, if class A has the instance of class B, then indicate class B as a field in the class A).
- When you implement the UML diagram in code, follow OOP principles. And your code should be consistent with your UML diagram, say all the methods/fields in your UML diagram should also be implemented in your code. Also, show how your classes can be used. (For example, if you created classes for the Factory design pattern, also write a code snippet that implements, creates objects etc).
- Try to follow the Java code conventions, so that we could judge the cleanness of your code: https://google.github.io/styleguide/javaguide.html
- Your report clearly and shortly justifies why you chose the particular pattern and describes the job you have done in UML and your code.
- Programming language restrictions: you can only use Java in your code

UML Diagram of Supermarket Billing Software



Report

Vladimir Makharev | Kirill Kuznetsov | Murat Shaikhutdinov | Artem Batalov | Pavel Baharuev

Why do we use Prototype design pattern?

We use the Prototype design pattern for creating product copies. It is useful for us because we have an unlimited warehouse for products of each type. So, each product is the clone of the sample in a warehouse with the same name, price, and auto-incrementing "id" field. Therefore, each copy of the product is unique as in real life, and it can be useful in further development for refund implementation, for example.

Code semantics explanation

Our code imitates the work of an offline supermarket. We have a customer, cashier (cashbox), and cart. Customers can put products from the product list into their shopping cart or take them out of it.
When the customer is done with filling the shopping cart, he creates an order and pushes it to the cashier. The cashier can edit the parameters of the order (change product names, prices, quantity of them), and can calculate the bill. For doing it, the cashier uses the key as in a real supermarket.

Code class relationships explanation

We have classes of customer, product cart, cashbox, and products. For each customer, we have one shopping cart. In the product cart, we have methods to add and remove products. It also has the list containing all the products in it and the method that pushes orders to the cashbox. We use the cart only through operating it within the customer (only the customer sends the request to it). When the order is ready, we push it to the cashbox. The cashier can edit the parameters of the product list using a special key. This key is checked in the product class. Customers can interact with products only by cloning them to a cart (that is the reason why we used a Prototype design pattern). However, the cashier can edit the parameters of the products using a key to change protected fields of the product class.