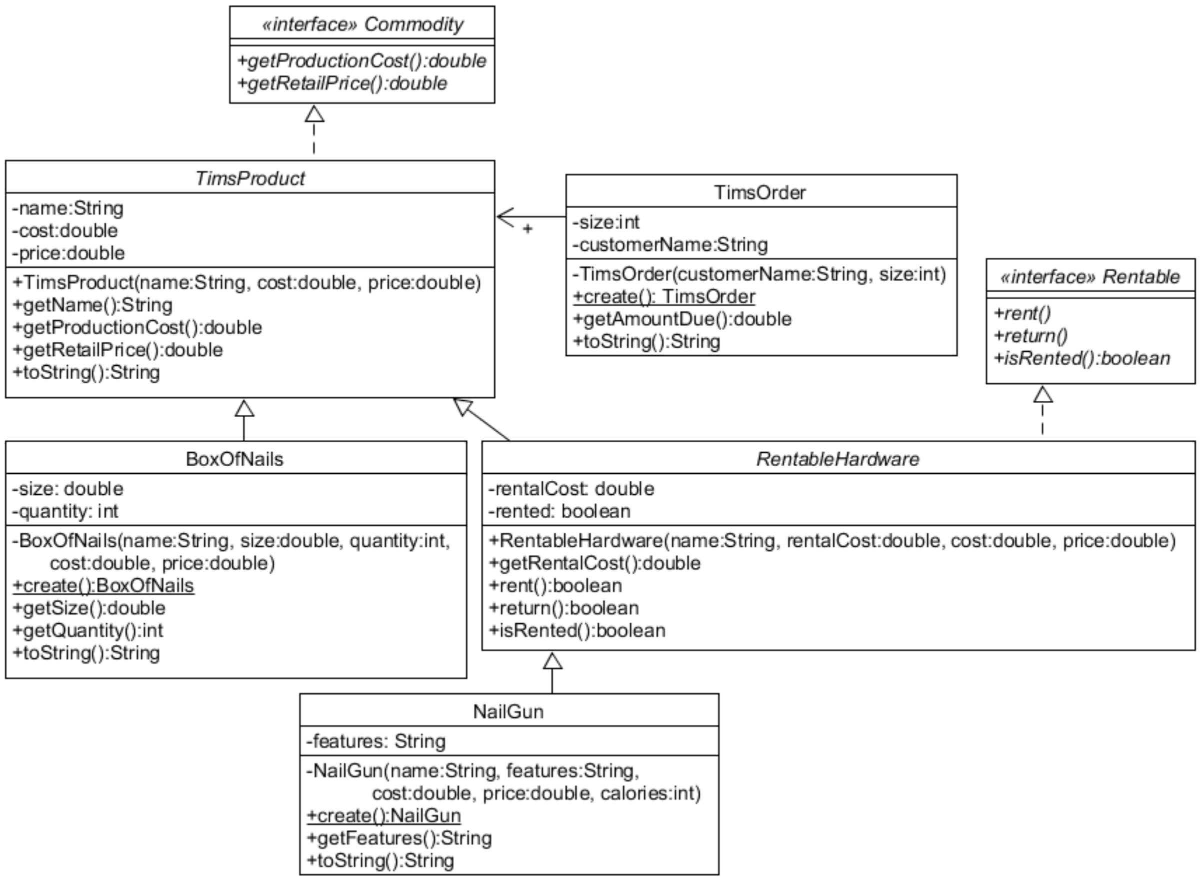
COMP10062: Assignment 7

The Assignment

This assignment is about polymorphism, abstract classes, and interfaces. The UML class diagram below shows an arrangement of classes with inheritance, association, and implementation relationships between them. Your job is to implement the classes and relationships shown here.



Description and Notes

The classes in the diagram represent products that are available at Tim's Hardware Store. Some of them are **Rentable** and some are not. Each one is a **TimsProduct** and a **Commodity**. You must add one **Rentable** product and one that is not **Rentable.**

Also represented is **TimsOrder**. This class has an array that contains 1 or more **TimsProducts**, representing a named customer's order. The **+** on the association arrow means "1 or more".

The **rent**, **isRented**, and **return** methods

These methods are used for checking a rentable tool out, checking whether it is currently rented, and returning it. In the **RentableHardware** abstract class, these methods all manipulate the boolean variable **rented**.

Constructors vs. Create Methods

Notice that many of the constructors are private. The classes with private constructors have static **create** methods. Instead of calling a constructor, call the **create** method, like this:

```
BoxOfNails b = BoxOfNails.create();
```

This method will have a dialog with the user to collect information about the object (in this case then name, cost, and price of the product as well as the size of the nails and the quantity of nails in the box), and will then create and return an object by calling the private constructor. When you add your two new products, you should follow this same design pattern.[1]

For rentable products, you should ask the user whether the customer will be renting or buying the product. If they are renting, you should call the **rent** method.

The **TimsOrder** class also uses a **create** method. This method asks the user their name and how many products they want, then creates a **TimsOrder** object by calling its private constructor. Then for each item in the array, it asks what type the customer wants (**BoxOfNails**, **NailGun** or one of the other two types you will add), then calls the appropriate **create** method and stores the result in the array.

Model vs. View

Note that because the **create** methods are talking to the user, this assignment does not maintain the standard model/view separation. However, you can think of the static methods as implementing the **view**, and the instance methods and variables as implementing the **model**.

The **getAmountDue** Method

The **getAmountDue** method adds up all the retail prices of the products in the order (or rental prices if **rented** is true for a product) and returns the sum. The **toString** method returns a **String** with the name of the customer and the **toString** values of all the products in the order, something like this:

```
Order for: Sam Scott
TimsProduct{name="Finishing Nails", cost=2.34, price=5.99}
   Type... BoxOfNails{size=1.5, quantity=50}
TimsProduct{name="Pneumatic Nail Gun", cost=98.32, price=199.99}
   Type... RentableHardware{rentalCost=19.99, rented=true}
   Type... NailGun{features="8 nails per second"}
```

Notice that the **BoxOfNails** and **NailGun toString** methods are incorporating the **TimsProduct** and **RentableHardware toString** methods. You should do this as well, but feel free to make your output look nicer than this!

[1] This is known in software engineering as the **Factory Design Pattern**. The create methods are called **static factory methods**. The Java FX **Color** class uses static factory methods for creating colors (**rgb**, **web**, etc.). Each of these methods interprets its arguments and then calls a **Color** constructor and returns the result.

TestClass.java

Use the code below to test your classes. This short program should initiate an elaborate dialog with the customer through calls to the **create** methods, then output the order and the total, rounded to two decimal places. You should be able to use this code without changing it.

```
public class TestClass {
    public static void main(String[] args) {
        TimsOrder t = TimsOrder.create();
        System.out.println(t);
        System.out.printf("Total Price: $%.2f\n", t.getAmountDue());
    }
}
```

Advice

1. Go top down. Start with the highest interfaces and classes and work downwards.
2. Make instance variables, and easy methods. For the more complicated methods, just write “stubs” for now.
3. A method stub is an empty method. You make this first and fill in the details later. An example of a method stub for **TimsOrder.create** is shown below.
4. Once you have the entire structure created like this, you can start filling in the details of each method.

```
public static TimsOrder create() {
    // TODO: Fill in this stub to have a dialog with the user
    //and create a TimsOrder.
    return null;
}
```

Hint

In the **TimsOrder.create** method, create the **TimsOrder** object as soon as you know many products are required, then you can access the private array of products directly. This is allowed because the **create** method is inside the **TimsOrder** class.

Handing In

See instructions on MyCanvas for submission.

Make sure you follow the **Documentation Standards** for the course.

Evaluation

Your assignment will be evaluated for performance (40%), structure (40%), and documentation (20%) using the rubric in the Canvas.