CSC8404 - Advanced Programming in Java

Assessed Coursework

The coursework, described below, involves development of interfaces, classes and tests to be used in the development of system for processing orders for a PC retailer.

Please check the submission deadline for this coursework on NESS.

---

1. Aim

The aim of this coursework is to practice the design principles covered in lectures. You will develop interfaces and classes to demonstrate that you have learned and understood module material, including:

- appropriate overriding of `Object` class methods, including overriding `toString` and providing a static `valueOf` method when appropriate
- design of interface-based hierarchies, programming through interfaces and dynamic/late binding
- the use of factories to control instantiation of objects, including guaranteeing the instantiation of unique instances
- defensive programming including the use of immutability
- the use of appropriate interfaces and classes from the Java Collections framework
- appropriate use of Javadocs to document your interfaces and classes
- the use of unit testing via JUnit framework

The coursework is **not** algorithmically challenging. The focus is on good design and good practice.

The coursework is **not** about development of an end-user application. You are developing interfaces and classes that could be used for the development of an application. **You should not develop a graphical user interface or a command line interface.** They are not necessary, and **you will be given no credit for doing so. You must test all your interfaces and classes using unit tests**.

Note: the specification provided below is a deliberate simplification of a real-world system. Your solution should correspond to the simplicity of the specification. You risk losing marks if you attempt to provide a more realistic model of the system or provide a solution that is more complicated than necessary.

---

2. System overview

A PC retailer company needs a set of interfaces and classes to manage their online ordering system.

The company offers PC models of two different types: preset models and custom models. Preset models are provided by third-party manufacturers and have a fixed list of parts that cannot be modified by the customer. Custom models are created by customers themselves from the parts available in the warehouse (see Section 3 for more details).

Customers can place orders by providing a list of PC models, their customer details and the credit card information. Once the order has been placed it can be cancelled. Each placed order must contain the ordered PC models, the customer information, payment details and a date and time it has been placed. One customer can place multiple orders. Each placed order can be fulfilled or cancelled. Once an order is fulfilled, it cannot be cancelled.

In order to fulfil an order, the company needs to generate the following information: 1) all third-party manufactures (together with the preset model names and their quantities) from whom they need to order the preset models, and 2) all computer parts for the custom models (i.e., names and quantities) that will need to be collected from the company warehouse.

The company also needs to keep track of the history of orders to see who their largest customer is (the one with most fulfilled orders), the most ordered preset model (the model that has been ordered the most by all customers from a single third-party manufacturer across all fulfilled orders) and the most ordered part for custom models (the part that appears most frequently in all custom models across all fulfilled orders). You should use alphabetical order for tie-breaking (e.g., if two largest customers have the same number of fulfilled orders, you need to return the one whose full name comes first alphabetically; similarly, for models and parts).

The following provides more detail of the required functionality. Please note that the pseudo-code below is just for your assistance. You may change the methods signature the way you see necessary.

placeOrder(PCModels, customer, creditCard)
: This method places an order for the given customer. You can assume that an order can be successfully placed if the customer's credit card is valid.

cancelOrder(order)
: This method cancels the given order. The order cannot be cancelled if it has been fulfilled.

fulfillOrder(order)
: This method fulfils the given order. It must provide the details about all the preset models and their quantities that need to be ordered from each third-party manufacturer and a list of parts and their quantities for custom models that need to be collected from the company warehouse. If the given order has been cancelled before, it cannot be fulfilled. Otherwise, you can assume that any placed order can be successfully fulfilled by the company: i.e., the company possesses an infinite number of all possible computer parts, and all third-party manufacturers have an infinite stock of all possible models.

getLargestCustomer()
: This method returns the customer with the most fulfilled orders together with the number of their orders.

getMostOrderedModel()
: This method returns the most popular preset model supplied by a single third-party manufacturer across all fulfilled orders in the company together with the number of times this model has been ordered.

getMostOrderedPart()
: This method returns the most ordered part across all custom models supplied by the company together with the number of times this part has been ordered.

---

3. Implementation

To complete the PC ordering system outlined in Section 2, you will need to provide interfaces and classes for the functionality described in this section. You must also implement test classes to unit test your solution.

Models

All PC models have the following functionality:

- A name of the model.
- A list of computer parts that this model contains.

All preset PC models also have:

- A name of the third-party manufacturer.
- The model specification (i.e., the list of its computer parts) cannot be changed.
- You can assume that different manufacturers can supply models with the same name.

All custom PC models:

- Have unique model names. You can decide the format of names for custom models.
- Must have methods for adding and removing computer parts.

To simplify real-world complications:

- You can assume that each computer part is represented by a non-empty user-defined string.
- You can also assume that the company and the third-party manufacturers keep an infinite stock of all customer-specified parts and models.

You must provide an appropriate hierarchy for models.

Customers

A customer has a name consisting of the first name and the last name.

You must provide methods to access a customer's first name and last name.

Two customers are considered to be the same if they have the same name.

Credit card

A credit card has three components: a unique 8-digit number, a date of expiry, and the name of the holder.

For simplicity, you **must** use `java.util.Date` for the date of expiry. You must **not** use **deprecated** methods of the `Date` class. So, in your test classes, you should use `java.util.Calendar` to construct the expiry date. You can assume default time zone and locale.

A credit card is considered valid, if the values for all three components are set, and if the card has not yet expired.

You must decide whether to define a separate class for the credit card. Whether or not you do, you must provide methods to access all individual components of the credit card (i.e., card number, expiry date, holder's name) and a method indicating whether the credit card is valid or not.

You must guarantee the uniqueness of credit card numbers.

Orders

Each order must have the following information: a list of ordered models, customer details, credit card details and the date and time the order has been placed (represented by the `java.util.Date` class).

You must also provide a way to indicate the order status (i.e., placed, cancelled or fulfilled).

You must decide on the appropriate collections to represent the history of orders.

---

4. Deliverables

Your solution should include your interfaces and classes that comprise the implementation of the system and types outlined in Sections 2 and 3. You must annotate your code with appropriate Javadocs. In addition, you should provide separate JUnit test classes that demonstrate thorough testing of your solution.

Also, write a document (MS Word, or PDF) that explains the structure of your overall design. This document should have a picture containing UML diagrams of all your classes and interfaces and their relationships depicted by annotated arrows (extends, implements, uses). For each class and interface, the UML diagram should show the class/interface name, fields and methods (where appropriate). Also, each diagram should indicate whether it is a class, abstract class or an interface.

You must submit your solution through NESS as a single zip archive that contains your Java source code files and your document.

Please check the submission deadline for this coursework on NESS.

---

5. Assessment & mark allocation

Marks will be allocated for:

- Overall structure (e.g. interfaces, classes and their relationships), **20%**
- Correct implementation of rules specified in Sections 2 and 3, as well as for choice and use of maps and collections, **35%**.
- Following good practice guidance: maintenance of invariants and defensive programming, use of immutability, appropriate overriding of `Object` methods, use of Javadoc comments, **25%**.
- Evidence of testing by implementation of appropriate test classes that test the normal case, boundary conditions, and exceptional cases. **20%**.

---

6. Style guidelines

Adopt a consistent style, do not violate naming conventions (when to use upper/lower case letters in names) and make appropriate use of whitespace (indentation and other spacing).

---

7. Further notes and hints

Break the coursework down into separate tasks. Start with the simpler classes first (e.g. for customer and credit card). Unit test classes as you work through the coursework.

For each class you implement you should consider:

1. whether to override `Object` methods (`equals`, `toString` etc.),
2. whether to use an interface-based hierarchy, and
3. whether the class should be immutable.

For any questions, please make full use of the practical classes. For any clarifications, you can also email:

Fedor.Shmarov@newcastle.ac.uk

Ellis.Solaiman@newcastle.ac.uk