Assignment 3: JavaFX Shopping Cart GUI

**About this assessment**

This summative coursework counts **50%** towards your module mark. You are required to work **in groups of three**. The deadline for submitting work via Blackboard is **23:59 Sunday 11th March 2018**.

**You must develop the GUI using JavaFX (and not any other Java framework such as Swing or AWT). Furthermore, you cannot use a GUI builder of any kind. Failure to meet either of these two requirements will result in an automatic mark of zero for this assignment.**

**Objectives**

The objective of this assessment is for you to demonstrate your ability to design and implement an OO system consisting of a set of Java classes, using advanced libraries within the Java SDK:

1. To study and correctly make use of a prebuilt shopping cart data model.
2. To build a suitable user interface using JavaFX 8 libraries.
3. To implement event handling procedures that provide a basis for an interactive and user-friendly system.
4. To adhere to standard principles of the Model-View-Controller (MVC) design pattern and appropriately decompose classes through abstraction and encapsulation.

**Submission**

Each group should submit two UML class diagrams, for the respective administrator and customer applications. The UML class diagrams should clearly show the classes (and their associations) that form the respective ‘view’ implementations for these two applications. This is not assessed but may be discussed during your viva. Ideally submit this in a common readable format, e.g. pdf or jpg. You may draw UML neatly on paper and scan it.

You must include documentation in your submission which enables us to understand the concepts (technical, user interface, architecture) behind your work. Ideally this documentation should be good enough to enable for somebody who was not involved in the project to continue your project. We leave the format of the documentation to you. A presentation is an option, but you could also do audio materials or anything else you consider useful. In any case it must be possible to view the materials on a standard PC using free or standard software (including Microsoft Office).

Submit the completed Shopping Cart GUI through the submission portal available on Blackboard. You should create a new folder called “CTEC2201Assignment3”, copy your Eclipse project(s) into this folder, ensuring all of the Java source files (and any other associated files, including the UML class diagrams) and your documentation are in it. Then compress the folder into a .zip file, which you can submit through Blackboard. Please can all students in a group submit the same work to Blackboard.

*Please turn over…*

**Eclipse Java Project / Package structure**

Please place the model, controller and view classes into appropriate **packages** within your Eclipse project, as shown in the MVC projects provided for the associated lab work on the module. The ApplicationLoader class that runs your GUI may go in a main package. This will help you logically separate them during development, and will help your tutor to mark these different components more easily.

As you need to work on two separate applications for an Administrator and a Customer (*see further detail later in this document*) these will initially be in different Java Projects. Whilst it is suitable to submit two separate projects within a .zip file, you should ideally end up with a single project containing both applications. To achieve this you could have a package structure such as: admin.view, admin.controller, etc, and then customer.view, customer.controller, etc. In having both applications in the same project you can more seamlessly save data from one application and load it into another. The alternative (and likely requirement during development) may involve manually copying a file from one project to another

**Marking and Feedback**

Your work will be marked using a predefined Blackboard VLE Rubric grid, which will display your level of achievement and feedback for each of the assessment criteria for this assignment (*shown later in this document*). The categories within the rubric act as a guide and the 'closest fit' will be selected for each individual piece of work.

**Anonymity**

The University has introduced the requirement to anonymously mark all assessment work. However, it recognises that, for some forms of assessment, anonymous marking is neither possible nor appropriate. Due to limitations with marking anonymously using Blackboard VLE Rubric grids, it has been agreed that this assessment will not be marked anonymously.

**Guidance on working in groups**

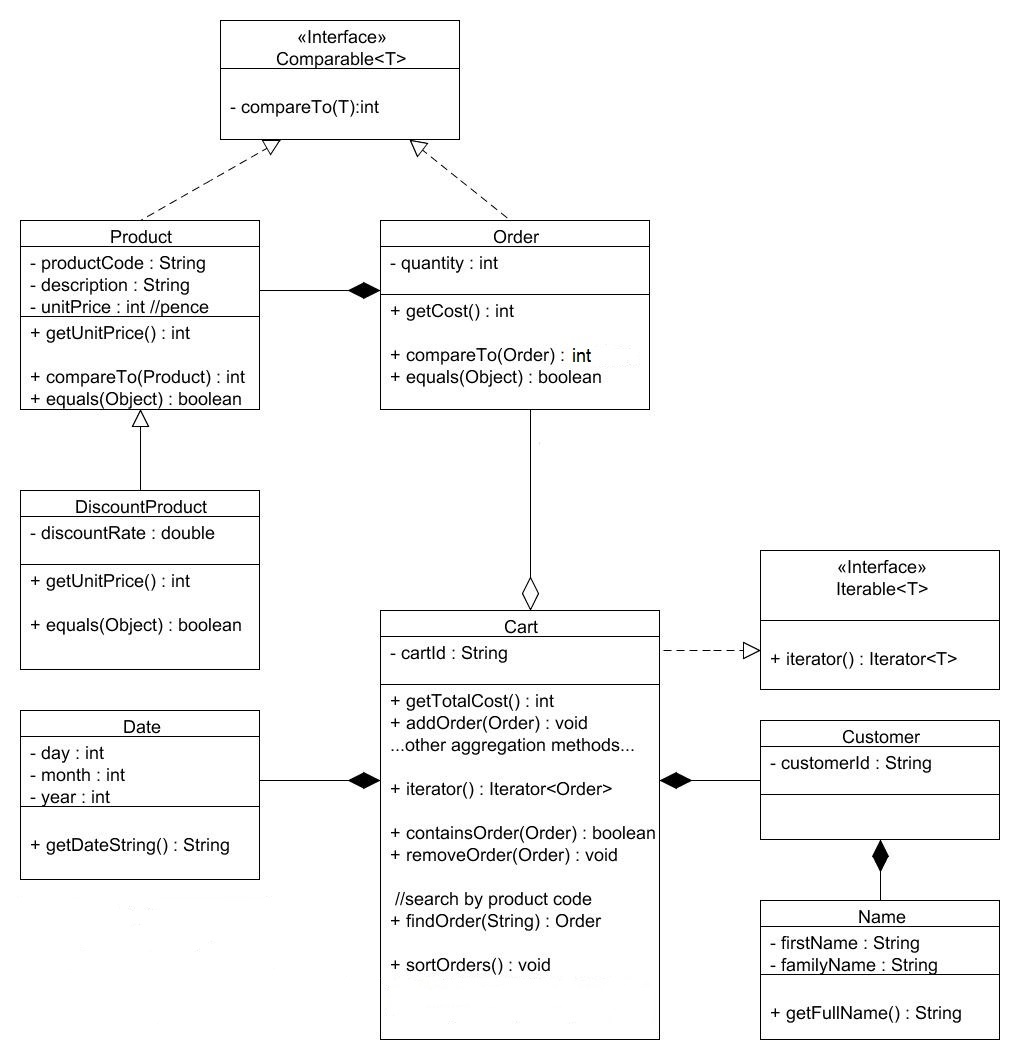
As previously mentioned, you need to work on two separate applications for an Administrator and a Customer (*see further detail later in this document*). You need to decide who does what in your group.

You should of course collaborate on the application in person or via email for example. Crucially, one of the requirements is that the customer application reads data from a file that the administrator application has produced. This won’t immediately be possible and so you can initially hardcode some relevant data into the customer system.

**Data Model: Shopping Cart**

When shopping for groceries online a **Customer** chooses a **Product** and places an **Order** for some quantity of it. The order is added to a **Cart**. A **Date** is set for the delivery. More orders can be added to the cart, and they can be changed up until checking out. It is possible for products to have a discount rate applied, i.e. a **DiscountProduct**.

The **partial** UML class diagram below shows the general underlying data model that you have been provided for this assignment (excluding a further extension shown later in this document).



*Please turn over…*

**The table below gives further guidance on the implementation of each class:**

|  |  |
| --- | --- |
| **Class** | **Comment** |
| **Name** | Simple name comprising a first name and family name. |
| **Date** | Simple date record with day/month/year fields. No validation. |
| **Customer** | A customer has a name, and a customer number. |
| **Product** | A product item has a product code, description, and price per unit. The price is given in pence.  A product can be tested for equality, and products are comparable, so they can be sorted into a natural order. |
| **DiscountProduct** | A discount product is a product with a discount rate applied. The discount rate should be between 0 and 1.0, with 0.1 being a 10% discount, 0.2 being a 20% discount, etc.  When retrieving the unit price of a discount product it should return the standard unit price with any discount applied. A discount amount should always drop to the nearest whole number, e.g. a calculated discount of 7.9 pence would apply as a discount of 7 pence on the unit price. |
| **Order** | An order is for a quantity of product. It is possible to increase and decrease the quantity. The cost of the order (i.e. Product’s unitPrice \* quantity) can be retrieved with the getCost() method.  An order can be tested for equality - an order is equal to another if its associated product is the same, i.e. quantity is ignored. Orders are also comparable. |
| **Cart** | A cart is for a customer and has a delivery date and an id. It consists of a list of product orders, and a variety of methods to add or remove an order, retrieve an order so that it can be modified or printed on a receipt, for example. The total cost of the Cart is given by the getTotalCost() method. An order can be searched for by product code.  A cart also allows its orders to be sorted into their natural order via a sortOrders() method.  The cart requires further methods: containsOrder and removeOrder, that will make use of an equality test. It also provides an iterator over its orders. |

*Please turn over…*

**Guidance on building the application**

*You are advised to thoroughly read this guidance and to continually refer to it as a means of helping you design and implement the JavaFX Shopping Cart GUI application.*

**Administrator application overview**

An administrator can create a variety of products, which should accumulate inside of the GUI, e.g. in a ListView control. You should also have the ability to add a discount to some products.

These products then need to be transferred to the Customer application, which can either be achieved by writing the product data to a file in text form, or more simply by saving a collection of products to a file using an ObjectOutputStream. If the latter approach is chosen then you could create a class in your model that aggregates a collection of products and save an object instance of this to a file.

**Customer application overview**

A customer can firstly enter their details, upon which, they will be able to view a list of all of the available products – these products should be read in from the file created by the Administrator application. Discount products should be clearly shown, with their discounted price.

The customer should be able to order one or more of these products and provide a quantity they wish to order, and in doing so populate their shopping cart. The quantity should be adjustable and products should be able to be removed.

Once the customer is happy with their selection, they should be able to see an overview of the cart’s contents (ideally sorted) and a total cost of the cart’s orders, along with any other important details, e.g. the delivery date. They should be able to write a receipt to a text file.

**Reloading applications** – ideally both the administrator and customer applications should be able to be saved in their current state and reloaded at a later point in time. This should ideally be done by saving the relevant objects in the data model to a file, and upon reloading them, repopulating the view.

**Model-View-Controller (MVC) design pattern**

You should clearly showcase the MVC design pattern throughout your implementation and attempt to separate the concerns of the model, view and controller. You will be assessed on your ability to sensibly decouple these entities to make a maintainable and reusable solution.

**The Model**

You have been provided with the data model for this application - see the file **model.zip** on Blackboard. You do not necessarily need to use every method within the model, and so you should pick and choose appropriate methods to aid your interpretation of the task.

The model classes purposely have no javadoc comments, so you are required to study the source code in order to understand how to use them as the data model for your MVC application.

For the majority of this assignment, you should not need to add any further code to the model and are advised to make use of existing methods wherever possible. If you wish to update the data model then you should ideally discuss this with your tutor beforehand.

**The View**

You can choose how you wish your GUI to look and feel, however, one suggestion would be to have separate forms on different tabs by using a TabPane, or you could legitimately choose another approach, e.g. an Accordion.

The desktop shopping cart system should allow a single user to become a customer and then populate a shopping cart with orders. You do not need to worry about multiple users using this system.

**Note**: You are strongly advised to decompose the view by separating logical containers into their own top-level classes. You will also need to provide suitable methods that allow relevant data and controls to be accessed and modified within the view.

**The Controller**

You should decouple the model and view by deploying a MVC architecture with a separate controller (as taught in this module), which will primarily contain the event handlers for this application.

**Application Loader**

Each application will have a main method program that extends Application (commonly called ApplicationLoader).

In the Customer application this could simply instantiate a default copy of the model (i.e. Cart) and view (i.e. your root pane) and pass these to the controller. The cart could then have its details set as the application is used.

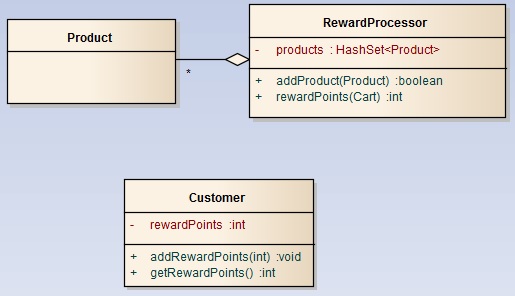
In the Administrator application this may instead just instantiate a copy of the view (i.e. the associated root pane) and pass this to the controller. The administrator has no access to the cart, but does need to work with products and discount products, so your controller / view should still interact with these classes. You can choose how to approach this.

*Please turn over…*

**EXTENSION to the system (weighted at 10%)**

When shopping for groceries online a **Customer** can gain reward points. A reward processor holds a collection of each **Product** for which points should be awarded, and can process a shopping **Cart** to ascertain its contents and apply any necessary awards.

The **partial** UML class diagram below shows some updates to the underlying data model.



**The table below gives further guidance on the implementation:**

|  |  |
| --- | --- |
| **Class** | **Comment** |
| **Customer** | A customer now has a field to hold their reward points, a means of adding these, and retrieving their current quota of points. |
| **RewardProcessor** | A reward processor should store products, which can gain rewards for the customer if they are in their shopping cart. A hash set collection is used for this. The rewardPoints(Cart) method checks if a rewardable product is in the cart, and if so adds points to the cart's customer. The number of points to be added should be the quantity of that particular order. The method should return the total number of reward points that have been awarded to the customer for this cart. |

Once the administrator has input a variety of products, they should be able to select some of these to be entitled to reward points, and therefore add them to the reward processor. The reward processor object in the model should then be saved to a file.

When a customer has populated their shopping cart, the cart should be passed to the reward processor (read in from a file) and an overview of their reward points shown.

*Please turn over..*

**Assessment Criteria**

The following criteria show how you will be assessed:

* **Documentation (20%):** The documentation helps with understanding the application and how it is built. It shows the fundamental concepts of the application.
* **User Interface (20%):** The user interface appropriately displays and captures relevant data, with a suitable layout and consideration of resizing.
* **Event Handling & Fitness for purpose (20%):** The user interface is responsive to interaction and can be used to achieve all associated tasks. There is appropriate validation throughout the application to ensure operations behave correctly, and features cannot be misused.
* **MVC Design & View Decomposition (20%):** The Model-View-Controller (MVC) design pattern has been applied to separate concerns and reduce coupling. The view has had logical containers separated into their own top-level classes. Abstraction has been applied throughout the design to reduce duplicate functionality.
* **Saving & Loading (10%):** Products can be transferred between the administrator and customer applications via saving and loading logical data. A customer can save a ‘receipt’ overviewing their cart to a file. Both the administrator and customer applications can be saved in their current state and reloaded at a later point in time.
* **Reward Processor extension (10%):** The extended data model has been used to allow an administrator to attach reward points to certain products. These details can then be saved and reloaded by the customer application, which correctly processes and displays reward points.

**Important note**: You must develop the GUI using JavaFX (and not any other Java framework such as Swing or AWT). Furthermore, you cannot use a GUI builder of any kind. Failure to meet either of these two requirements will result in an automatic mark of zero for this assignment.