Faculty of Technology – Course work Specification 2019/20

Module name:	Object Oriented Programming		
Module code:	IMAT5101		
Title of the Assignment:	Assignment 3		
This coursework item is:	Summative		
This summative coursework will be marked anonymously		No	
The learning outcomes that are assessed by this coursework are: 1. Have a comprehensive understanding of OO programming concepts: abstraction, encapsulation, inheritance, etc. 2. Be able to select appropriate API facilities in the design, implementation and testing of OO applications and applets.			
This coursework is:	Individual		
This coursework constitutes 50% **to the overall module mark.**			
Date Set:	Monday 9th December 2019.		
Date & Time Due:	Monday 13th January 2020 at 14:00		
Your marked coursework and feedback will be available to you on: If for any reason this is not forthcoming by the due date your module leader will let you know why and when it can be expected. The Head of Studies (headofstudies-tec@dmu.ac.uk) should be informed of any issues relating to the return of marked coursework and feedback. Note that you should normally receive feedback on your coursework by **no later than four working weeks after the formal hand-in date,** provided that you met the submission deadline.	Within 4 weeks of week 16. Your tutor will always aim to provide you with feedback as soon as is possible.		
When completed you are required to submit your coursework to: 1. Blackboard VLE through a submission portal.			
Late submission of coursework policy: Late submissions will be processed in accordance with current University regulations which state: *"the time period during which a student may submit a piece of work late without authorisation and have the work capped at 40% [50% at PG level] if passed is **14 calendar days**. Work submitted unauthorised more than 14 calendar days after the original submission date will receive a mark of 0%. These regulations apply to a student's first attempt at coursework. Work submitted late without authorisation which constitutes reassessment of a previously failed piece of coursework will always receive a mark of 0%."*			
Academic Offences and Bad Academic Practices: These include plagiarism, cheating, collusion, copying work and reuse of your own work, poor referencing or the passing off of somebody else's ideas as your own. If you are in any doubt about what constitutes an academic offence or bad academic practice you must check with your tutor. Further information and details of how DSU can support you, if needed, is available at: http://www.dmu.ac.uk/dmu-students/the-student-gateway/academic-support-office/academic-offences.aspx and http://www.dmu.ac.uk/dmu-students/the-student-gateway/academic-support-office/bad-academic-practice.aspx			
Tasks to be undertaken: See (following) attached document.			
Deliverables to be submitted for assessment: See (following) attached document.			
How the work will be marked: See (following) attached document.			
Module leader/tutor name:	**Please see Blackboard Staff Contacts**		
Contact details:	**Please see Blackboard Staff Contacts**		

Assessment 3

About this assessment

This individual summative assessment counts 50% towards your module mark. You are given a scenario for your assignment. This takes the form of a design description and a specification by way of a class diagram. You should gain an understanding of the scenario, implement the class diagram and produce the functionality listed within the use cases.

You should implement each class to the expected standards as discussed and practiced during the module.

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 and a client program. In particular:

1. To design and implement classes with suitable fields, constructors, accessor methods, and modifier methods.
2. To conform to the standard conventions of Java.
3. To implement classes that are associated by inheritance, delegation, composition and aggregation.
4. To write a client application that uses your classes to show that they function correctly.

Submission

Submit one .zip Archive File called **`imat5101LabTest.zip`** via Blackboard consisting of your (Eclipse) project folder. Make sure all of your source code is in there.

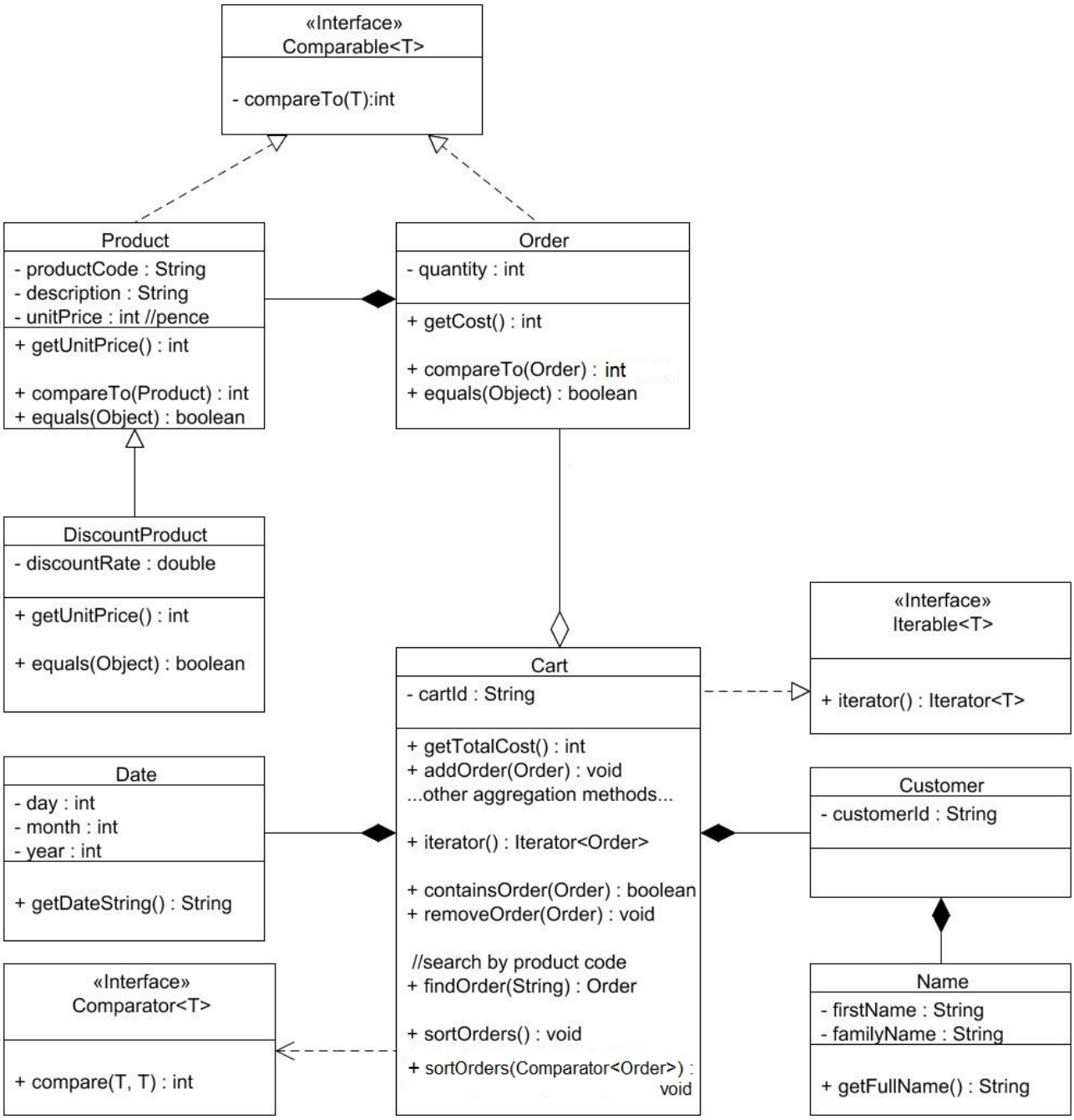
P.T.O.

Your Scenario: `Shopping Cart`

[STAGE 1 – 80%] 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 contents of the shopping cart can be sorted in various ways. Orders within a cart can be iterated over. To support further features of the cart, orders and associated products can be tested for equality.

The **partial** UML class diagram below shows the underlying data model you should use to achieve this scenario:



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 unique product code, description, and price per unit. The price is given in pence. A product can be tested for equality by providing an appropriate overridden equals(...) method that assesses the equality of each field. Additionally, products are comparable, so they can be sorted into a natural order. They should firstly be compared by product code, if these are the same, then by description, and if these are the same, then by unit price.
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. 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 comparable and should firstly be compared by their associated product, and if these are the same, then by their quantity.
Cart	A cart is for a customer and has a delivery date and a unique id. It consists of a list of product orders, and a set 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 should also allow its orders to be sorted into their natural order via a `sortOrders()` method. It should provide an additional overloaded method `sortOrders(Comparator<Order>)` that accepts a custom comparator and applies this to the list of orders. The cart requires two further methods: `containsOrder` and `removeOrder`, as shown in the UML diagram, that will make use of an equality test. A cart should also provide an iterator over its orders.

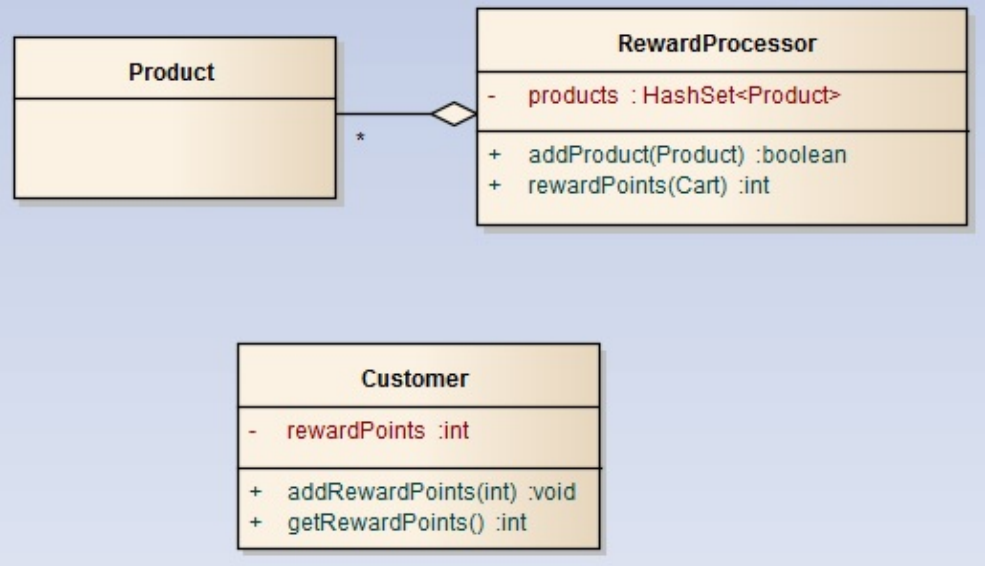
Use Cases: You should have a test class, with a main method, that should complete the following use cases for the "Shopping Cart System" to show it works as expected. Please **clearly highlight** where each use case is exercised in your test program with comments:

CartTest	**UC1**: A test program that creates an instance of Cart, populates it with several orders (for both products and discount products), and uses a for-each loop to produce a formatted listing of the orders with their price, the number of items in the cart and the overall total price. *Note*: You can either use a data file to dynamically read orders, or hardcode the order instances to add to the cart. **UC2**: Test each of the sort methods work as expected, as evidenced by appropriate output. • For the `sortOrders(Comparator<Order>)` method, you should pass in a custom comparator that sorts by cost (asc) and if these are the same then by product (desc). **UC3**: Additionally, test the use of equality by using the contains and remove methods. Also, you should use a PrintWriter to produce a receipt for all of the orders and associated details of the cart, and output these into a text file Cart.txt. **UC4**: Add further code to show how each method of Cart can be used. In particular, increase and/or decrease the quantity of one or more of the products in the cart, and then re-print the updated cart listing.

Please turn over…

[STAGE 2 – 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. **Please note:** The Customer and Product classes should remain as before, but should include any additional implementation details as shown in the UML diagram.



The table below gives further guidance:

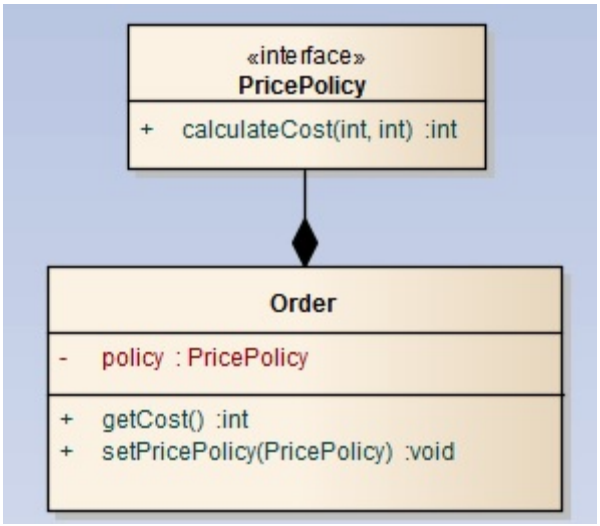
Class	Comment
Customer	A customer now has a field to hold their reward points, a means of adding these, and retrieving their current quota of points.
RewardProcessor	A reward processor should store products, which can gain rewards for the customer if they are in their shopping cart. A hash set collection should be 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. **Note**: you may need to do some research into how to use a HashSet collection in Java.

Use Cases: Add the following use case at the end of your existing code in your test program – you may need to update other code that has been affected by the additional changes.

CartTest	**UC5**: Create an instance of RewardProcessor, add a selection of products to it (but not all the same as those in your cart), and then test the `rewardPoints` method appropriately.

[STAGE 3 – 10%] When shopping, each **Order** has an associated **PricePolicy**. An order may have a default standard price policy, which does not affect the cost, or other price policies, such as buy one, get one free (B1G1F), or buy two, get one free (B2G1F), etc. The cost of the order would be reduced as a result of the price policy being applied.

The **partial** UML class diagram below shows some updates to the underlying data model. **Please note:** The Order class should remain as before, but should include any additional implementation details as shown in the UML diagram.



The table below gives further guidance:

Class / Interface	Comment
PricePolicy	The functional interface PricePolicy, contains a single abstract method called `calculateCost`, which accepts two integers representing quantity and unit price, and returns an integer representing the calculated cost, for the given price policy.
Order	As already stated, the standard cost of an order is calculated by multiplying a product's unit price by the quantity ordered. A default price policy would do just this. Other price policies can be set, resulting in the `getCost()` method applying that policy.

Use Cases: Add the following use case at the end of your existing code in your test program – you may need to update other code that has been affected by the additional changes.

CartTest	**UC6**: Create four different Order object instances and ensure each has a different price policy – default, buy 1 get 1 free, buy 2 get 1 free, and buy 5 get 1 free. Calculate the cost of each to show the price policies are being applied correctly, and test each policy with different quantities.

IMAT5101 Assignment 3 Assessment Indicators

Grade indicators

Giving a general idea of how to achieve a pass, merit, and distinction for this assessment.

Pass (50 - 59%): Classes without dependencies have been implemented meeting standard basic conventions. At least one Inheritance, one Composition and one Aggregation association have each been attempted. The data model has been populated (probably by hard-coding). At least two use-case has been attempted. Basic tasks have been carried out independently during the lab test.

Merit (60 - 69%): As for Pass above. Most classes in the design have been implemented to meet standard conventions and documentation standards. Test program has identified the limitations of a class. The data model has been suitably populated. Classes implement appropriate interfaces (e.g. Comparable, Iterable). Use-cases have been simulated with moderate success. Most tasks have been completed in the lab test and successfully demonstrate the essential functionality of the system.

Distinction(70%+): As for Merit above. Additionally: a coherent data model has been designed and implemented, with appropriate levels of abstraction. Any additional specified functionality has been added, with creativity beings shown. The integrity of fields has been a consideration and mostly maintained. All use-cases have been implemented with high levels of success. Documentation is at the appropriate level to support maintainable code. All the lab test activities have been completed successfully and demonstrate the correctness and quality of the system.

Please Turn Over for a Checklist of Assessment Indicators.

Checklist

These indicators will be used as a basis for the assessment mark and will form your feedback.

(70% Model) Class documentation, design and implementation.

- Javadoc class headers, method and constructor descriptions, and use of @tags. *[Stage 1]*
- Fields, constructors and standard methods – (adhering to standard Java conventions). *[Stage 1]*
- Correct implementation of associations (Composition, Aggregation and Inheritance). *[Stage 1]*
- Appropriate overridden implementations of the equals(…) method. *[Stage 1]*
- Use of common Interface types, e.g. Iterable, Comparable, Comparator. *[Stage 1]*
- Implementation of reward points, and a hash-based collection. *[Stage 2]*
- Implementation of price policies, using a functional interface. *[Stage 3]*

(30% Testing) Use Case success.

- Use Case 1 has been achieved appropriately. *[Stage 1]*
- Use Case 2 has been achieved appropriately. *[Stage 1]*
- Use Case 3 has been achieved appropriately. *[Stage 1]*
- Use Case 4 has been achieved appropriately. *[Stage 1]*
- Use Case 5 has been achieved appropriately. *[Stage 2]*
- Use Case 6 has been achieved appropriately. *[Stage 3]*

For each use case the overall quality of test data and output will be considered.