

COM1027 Programming Fundamentals

Coursework Descriptor for Summative Assessment

Exploring Object-Oriented Programming

Released	[REDACTED]
Deadline	[REDACTED]
Feedback	Individual feedback on SurreyLearn
Weighting	80% of final module grade
Academic Misconduct	Coursework will be routinely checked for academic misconduct. Your submission must be your own work. Please read the Student Handbook to ensure that you know what this means. Do not give your code to anyone else, either before or after the coursework submission deadline. Coursework will be routinely checked for academic misconduct.

Contents

1 Overview	2
2 Problem Definition	4
3 Setting Up Your Workspace	6
4 Commenting and Code Style	7
5 Question 1 (48%): Defining and Unit Testing a Simple Car Auction	8
6 Question 2 (5%): Introducing More Complexity	12
7 Question 3 (26%): Introducing Modularity	14
8 Question 4 (21%): Completing the Functionality	16
9 Submission Instructions and Checking Your Work	19

Please read this descriptor thoroughly and carefully. Many issues encountered in labs came from misreading UML diagrams or missing out steps in the instructions. You can save a lot of hassle by digesting everything before you start to code.

OVERVIEW

1 Overview

The purpose of this coursework is for you to demonstrate your ability to:

- define classes, fields, constructors and methods
- manipulate collections of objects, lists and maps
- implement an inheritance structure
- use JUnit to define your own test classes.

The questions in the coursework brief will also allow you to explore and implement the following:

- exceptions and error handling
- use of generic classes
- file reading/writing.

This coursework should help you develop your understanding of inheritance in object-oriented techniques and develop your skills and use of algorithms to manipulate nested data structures.

You are required to capture the problem definition given in Section 2 using Java. There are four *mandatory* questions and each question builds on the previous one. The bulk of the deliverables in this descriptor may be found in Sections 4 to 8. The rest of the sections outline the problem (Section 2) and how to upload your solution (Section 9).

We have provided a set of test classes as part of the skeleton code package, but you will also be required to write some tests of your own.

You will receive a mark and individual feedback via SurreyLearn. 80% of your final mark for this module is apportioned to this assessment. The marking scheme will be made available in a separate document.

Referencing

Writing code is similar to academic writing in that when you use or adapt code developed by someone else as part of your project, you must cite your source. If any external resources were used to support the implementation of the project, these should be specifically identified in the project. This includes any material that is not available on SurreyLearn. Instead of quoting or paraphrasing a source (as in academic writing papers), you should include an inline comment in the code. These comments not only ensure you are giving proper credit but also help with code understanding and debugging.

Generally, the URL and the date of retrieval are sufficient. If this is not an online source, use the name instead (e.g. name of the textbook, article, etc.). You should *never* copy code from other students. Your peers are not considered authorised sources—please see your Student Handbook for more information on academic misconduct.

An example of referencing in your Java code:

```
/**  
 * Reference:  
 * URL: https://www.w3schools.com/java/java_data_types.asp  
 * Date: 05/10/2023  
 */
```

Output Formats

Pay close attention to the required outputs as given in this descriptor and the JUnit tests. In the below example, the symbol `\t` means the tab character (`\t`), the symbol `\s` means a space character, and the symbol `\n` means a linefeed (`\n`). Note that lines 5–7 consist of linefeed (newline) characters only and line 8 is completely blank.

```
1 Spaces \s text \s and \s linefeed\n2     \t1 \s tab \n\n3         \t2 \s tabs \n\n4 \s \s \s 4 \s spaces \s at \s start \n\n5 \n\n6 \n\n7 \n\n8
```

2 Problem Definition

You must develop Java code that allows a car trading system to be modelled. This is based loosely on AutoTrader, which is a British automotive classified advertising business. The coursework has some simplifications, and focuses on the algorithms rather than the graphical user interface.

The proposed system manages the purchase of different types of automotive (cars). These can either be purchased through an auction or through a trade. In order for a car to be bid for or bought directly, they have to be registered in the system first. When the sale has ended for a particular car, the system records the cars that have been sold successfully, as well as those that are still unsold.

You should base your code on the specification below and the test cases provided, facilitating test-driven development. By the end of the coursework the system will have the following functionality:

- A. A user has a name and a surname (known as full name).
- B. A user can place an offer on an existing car advert.
- C. A user can be a seller or a buyer. A seller can provide cars to the dealership, whereas buyers can only place offers on the available cars.
- D. An advert consists of a car and a list of offers that have been placed by users.
- E. Each offer consists of a user and an associated value which must be greater than 0.
- F. The system will be able to determine the highest offer/price for any car advert. This will determine whether a car was successfully purchased at the end of the sale.
- G. Each car has its own specification stored including its unique ID, name, colour, reserved price, type of gearbox, type of car body, a set number of seats, and its condition.
- H. The reserve price for each car needs to be greater than or equal to 0.
- I. Some properties of the car are fixed, and can only take the form of one of the values in the group of constants (unchangeable variables, like final variables). This is particularly the case with the gearbox, the body of the car and the car's condition.
- J. Each car has to be registered in the system before a user can place an offer for or buy it.
- K. A car cannot be registered if it is already being processed or auctioned.
- L. An auctioneer is part of the system, and keeps track of the cars (that are for auction) that are currently being sold, and a history of the cars that have been sold and/or remain unsold at the end of a sale.
- M. The system shall be able to control the end of the sale for a particular car advert.
- N. The system shall allow a buyer to place an offer provided the 'Biddable' car (meaning the car for auction) is registered as being for sale and that the new offer value is higher than the current highest value for that advert.
- O. The system shall allow a buyer to buy a car directly (if not in the auction), provided that the car is available. Upon a successful purchase, the sale of the car should end, causing the car advert to also be removed.
- P. The system will be able to display all the sold cars for any car dealership; auctioneer or trader.
- Q. The system will be able to display all the unsold cars for any dealership.
- R. The system will be able to display some key statistics depending on the type of the dealership.

PROBLEM DEFINITION

- S. A dealership can be an auctioneer or a trader. An auctioneer is responsible for managing all the car adverts that are available for auction, on which buyers can place an offer. A trader is responsible for managing the car adverts that are available for direct purchase.
- T. The system differentiates its functionality for different types of users (buyers, sellers).
- U. Buyers must be at least 18 years old. Anyone requesting to purchase a car below the age of 18 should automatically be rejected by the system.

3 Setting Up Your Workspace

Download the skeleton code of the coursework from SurreyLearn. It contains the test classes and a package structure for each question.

Extract the contents of the project in your Downloads folder. Copy the uncompressed (unzipped) ‘Coursework’ folder to your local git repository. Note: do not copy the ‘COM1027_Coursework’ folder. Navigate into the folder, and only copy the ‘Coursework’ folder to your local git repository.

If you are using the computer labs, then your default local git directory would look like this:

/user/HS***/[username]/git/com1027[username]/

For example: /user/HS***/[username]/git/com1027sk0041/ *where sk0041 is a sample username*

To import the skeleton code, we will follow the same steps as the labs:

1. Start Eclipse and open a workspace.
2. Import the project in Eclipse, by clicking on File > Import > General > Project from Folder or Archive.
3. Select Directory to locate Coursework from your local git repository.
4. Once the files are loaded, select the Eclipse project and click Finish.

For submission instructions see Section 9.

Important

Instructions on updating your GitLab repository configuration files for the coursework will be communicated separately. This information will be released here:

<https://surreylearn.surrey.ac.uk/d2l/le/lessons/239743/topics/2591953>

4 Commenting and Code Style

In this coursework we want you to spend time writing code and understanding how to develop algorithms for nested data structures.

Therefore, you do not need to comment every class. However, we do want to see that you understand how to write useful comments, so you must include them in the Auctioneer class for Question 1.

We require neat code that is well-formatted and readable. In each of the questions, well formatted code means adhering to the Java coding style used in the module. Note that the style and comments of your code will be assessed and marked.

If the code does not work, then you should attempt to comment on what you have done so that we can see what you were trying to do. You can of course include comments in any of the classes if they help you to develop the solution.

Important

10% of the total marks are allocated for well-formatted code. This applies to all tasks you attempt.

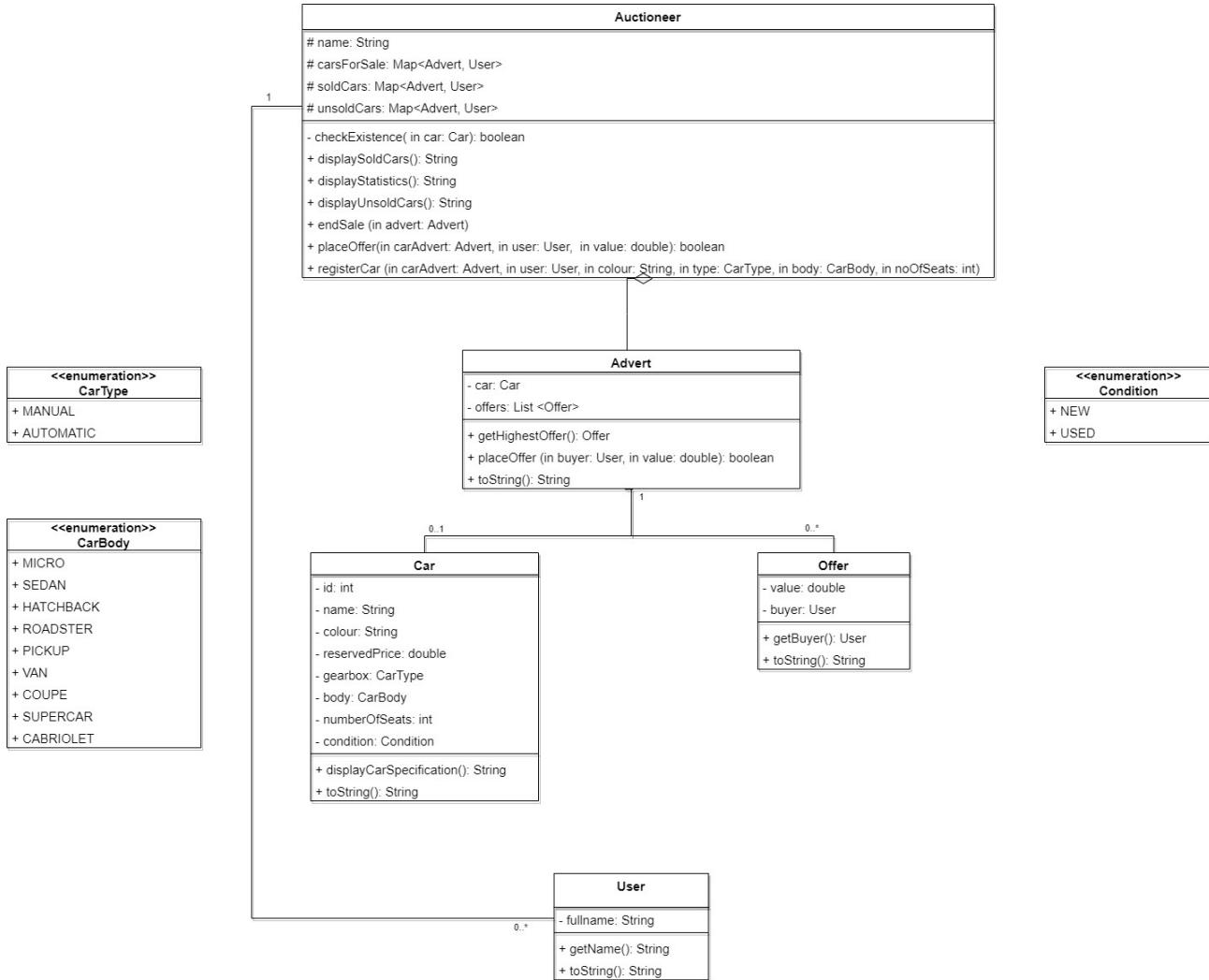
2% of the total marks are given for commenting, which is only required for Auctioneer in Question 1.

Instructions on checking your code style will be published by Week 7 on the following page:

<https://surreylearn.surrey.ac.uk/d2l/le/lessons/239743/topics/2591953>

QUESTION 1 (48%): DEFINING AND UNIT TESTING A SIMPLE CAR AUCTION

5 Question 1 (48%): Defining and Unit Testing a Simple Car Auction



UML diagram for Question 1

The classes for this question are shown in the UML diagram. You should write the code that implements this structure.

Place your code in the ‘coursework_question1’ package within the ‘src/main/java’ folder. The question provides the learning experience of:

- defining simple and complex classes
- using enumerations
- iterating through a map of car adverts and their associated sellers
- iterating through a list of offers.

This question addresses the following functionality from Section 2: A, B, D-N, P (partly), Q, R (partly).

Note: This question deals only with cars that can be bid for. You are expected to throw exceptions when validations fail on input parameters in order for all the tests to pass.

QUESTION 1 (48%): DEFINING AND UNIT TESTING A SIMPLE CAR AUCTION

Writing Java Code

- Answer this question in the ‘coursework_question1’ package. Some accessor and mutator methods have been omitted from the UML diagram. You can add such methods, including private methods, to support the functionality of your project.
- Define three enumerated types: CarType, CarBody and Condition as per the UML diagram.
- Define a well-formatted User class. A user object consists of a fullname field which holds the user’s first and last name, each starting with a capital letter. If this condition is not met, then an exception is raised. The class also consists of two methods; getName and toString. The getName method returns only the first name of the user, and the toString method returns the user’s first and last name without being formatted in any way:

```
1 Stella Kazamia
```

- Define a well-formatted Offer class. An offer object consists of the buyer who places an offer on a car advert, and the value placed. If an invalid value is passed, then an appropriate exception should be raised. The Offer class also includes a simple accessor method, and a toString method. For example:

```
1 Bob Ross offered £2000.00
```

- Define a well-formatted Car class. This class makes use of three enumerations: CarType, CarBody and Condition. Define each enumeration separately in its own Java file.

The Car class also consists of two methods. The displayCarSpecification returns the formatted output, for example:

```
1 1234 - Mazda MX5 (£20000.00)
2   ↴Type: MANUAL
3   ↴Style: SUPERCAR
4   ↴Colour: White
5   ↴No. of Seats: 2
6   ↴Condition: NEW
```

And the toString method returns the following:

```
1 1234 - Mazda MX5
```

- Define a well-formatted Advert class that handles all the offers placed on a given car. Two fields are required, one of which is a list of all the offers. The Advert class provides three methods: getHighestOffer, placeOffer and toString.

The toString method is required to return the formatted advert as:

```
1 Ad: 2345 - Toyota Corolla (£15000.00)
2   ↴Type: MANUAL
3   ↴Style: MICRO
4   ↴Colour: Red
5   ↴No. of Seats: 4
6   ↴Condition: NEW
7
```

QUESTION 1 (48%): DEFINING AND UNIT TESTING A SIMPLE CAR AUCTION

Make sure that appropriate exceptions are raised when invalid values are entered.

- Define a well-formatted Auctioneer class. This class manages all the car adverts that are available for auction, and keeps track of those that have been sold and/or remained unsold at the end of a sale. The Auctioneer consists of a name field and three maps representing the cars for sale, those sold and those that remained unsold (due to no successful offers).
 - The checkExistence method checks whether the specified car exists in the sequence of cars for sale.
 - The register method is responsible for registering a specific car if it has not been registered yet. The registration includes the initialisation of the car's specification, for example its colour, body, gearbox, and number of seats.
 - The placeOffer method only allows an offer to be placed, if a) valid parameters are passed, and b) the car is available for sale.
 - The endSale method determines the end of a car advert. Upon a successful purchase of a car, the relevant advert gets removed so no further offers can be placed.
 - This class also has three display methods. Note that the display methods should not contain `System.out.println()` code. Such methods need to return appropriate String representations so that they can be tested against using an assert statement. These display methods are `displaySoldCars`, `displayUnsoldCars` and a simple `displayStatistics`. The `displaySoldCars` method returns its output in the following format:

```
1 SOLD CARS :  
2 1234 - Purchased by Stella with a successful £20000.00 bid.  
3
```

whereas the `displayUnsoldCars` returns the following:

```
1 UNSOLD CARS :  
2 Ad : 1234 - Mazda 3 (£20000.00)  
3   ↗Type : AUTOMATIC  
4   ↗Style : HATCHBACK  
5   ↗Colour : Blue  
6   ↗No. of Seats : 5  
7   ↗Condition : NEW  
8
```

The final display method (`displayStatistics`) simply returns the word 'Statistics' without performing any operations.

Notes

- Be careful when returning a String representation of your objects. Each display method uses a set of special characters to format the String. Revisit the resources on special characters in Strings. See Section 1 for further details.
- The `registerCar` method is responsible for registering a car and adding it to the for-sale map. If the car is already registered then it should already be in the for-sale map; thus this method should not enable the car to be re-registered with the same information. Once a car is registered, buyers can place offers on the car until the seller ends the sale.
- The `endSale` method is responsible for checking whether the car in question is available in the for-sale map and if so, it will assess whether the last offer placed (if any) is above the reserved price of the car in order to make the decision on whether the car will be listed at the end of the sale as an unsold or a sold car. Upon a successful purchase of a car, the relevant car advert gets removed so no further offers can be placed.

QUESTION 1 (48%): DEFINING AND UNIT TESTING A SIMPLE CAR AUCTION

Perform Correct Validation

Example: Look at the code in the UserJTest class and ensure your User class throws the necessary exceptions, otherwise the tests will fail. In particular, this means a user cannot be created with invalid parameters. In this example, invalid parameters include:

- a null value
- a String representation that does not consist of two words (first and last name)
- a String representation that does not have each word starting with a capital letter (i.e. ‘joe Bloggs’ instead of Joe Bloggs’).

JUnit Tests

You have been provided with the unit test classes for the User, Car, Offer and Auctioneer classes.

Define a new test class called AdvertTest with appropriate test methods and place this in the correct folder ‘test/main/java’ within the coursework_question1 test package to unit test the Advert class.

Important

The test classes are to be defined in the ‘test/main/java’ folder. The Java project is separated into two main source folders; the main folder that consists of your source code (i.e. Auctioneer, Advert, Car, etc.) and the test folder that consists of the test classes.

Test classes like the AdvertJTest class, should be defined in the test folder with the rest of the JUnit files. Make sure that this class is defined as a JUnit test case with appropriate test methods and the use of the ‘assertEquals’ method.

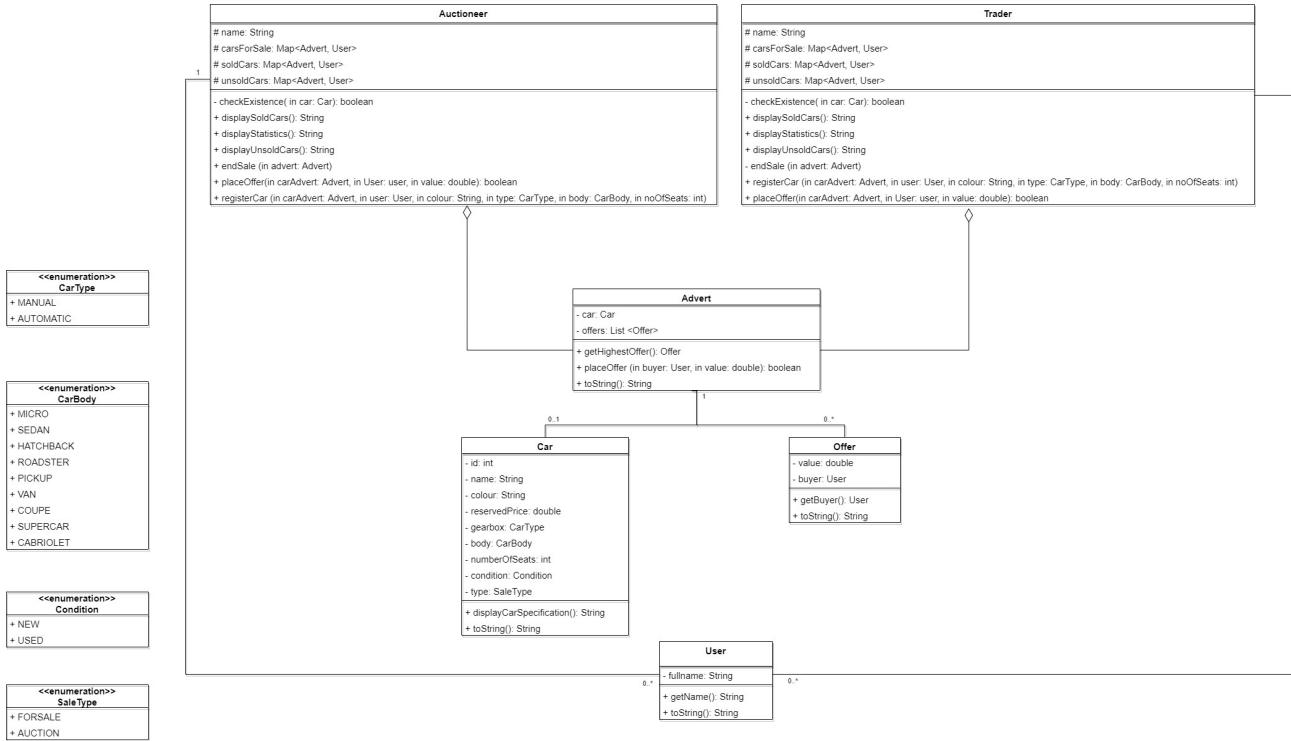
After completing the code for each class, test it against the test classes provided for this question. You don’t have to wait until you have completed all the classes before running the tests. Test as you write the code. To do this, right-click on the test files for coursework_question1 and select Run As > JUnit Test.

Remember that if you see any errors in the test classes, this is because you have probably misread the UML diagram or instructions, or forgotten to write the code in the correct package. Missing import statements could also affect the functionality of your code.

Looking at the format of the test classes will give you an insight into how to write correct code, including the expected output or String representations of the `toString` or `display` methods.

QUESTION 2 (5%): INTRODUCING MORE COMPLEXITY

6 Question 2 (5%): Introducing More Complexity



UML diagram for Question 2

The purpose of this question is to differentiate between the two different kinds of cars and to modify the implementation of methods where appropriate.

Copy your source code from the ‘coursework_question1’ folder to the ‘coursework_question2’ folder. Make sure all package names refer to the correct version at the beginning of each class.

This question addresses the following functionality: A, B, D-P, R (partly). This question will provide a learning experience in manipulating multiple classes, and conditional statements that enable the program to function differently depending on its input.

Writing Java Code

- Answer this question in the ‘coursework_question2’ package.
- Copy the required classes across from the previous question as your starting point.
- Define an enumerated type called `SaleType` with its required range of values.
- Define a new well formatted `Trader` class. The `Trader` class consists of the same fields as the `Auctioneer` class; three maps representing the cars for sale, those sold and those that remained unsold. Unlike the `Auctioneer` class, this class is responsible for managing the car adverts that are available for a direct purchase, and are not for auction. It consists of seven user-defined methods, and the following might function differently from the `Auctioneer` class:
 - The `register` method is responsible for registering a specific ‘biddable’ car (i.e. `AUCTION`) if it has not yet been registered. The registration includes the initialisation of the car’s specification, for example its colour, body, gearbox, and number of seats.

QUESTION 2 (5%): INTRODUCING MORE COMPLEXITY

- The placeOffer method only allows an offer to be placed, if valid parameters are parsed, and the car is available for sale. If a successful offer is made on the car, then the car gets removed from the list of available cars.
- The endSale method determines the end of a car advert. Upon a successful purchase of a car, the relevant advert gets removed so no further offers can be placed. The endSale method only gets called in the placeOffer method in the Trader class, unlike the Auctioneer class.

Note: This class may result in having duplicate code – for example having the same or similar fields (or method functionality) between the Trader and Auctioneer classes.

- Revise the Car class as required, to reflect the structure of the class diagram.
- Revise the Auctioneer class as required, to reflect the structure of the class diagram.

Perform Correct Validation

Example: Look at the code in the TraderJTest class and ensure your Trader class throws the necessary exceptions, otherwise the tests will fail. In particular, this means that the endSale cannot accept a null car advert.

JUnit Tests

All the tests for this question have been written for you. So you can run your code against the tests.

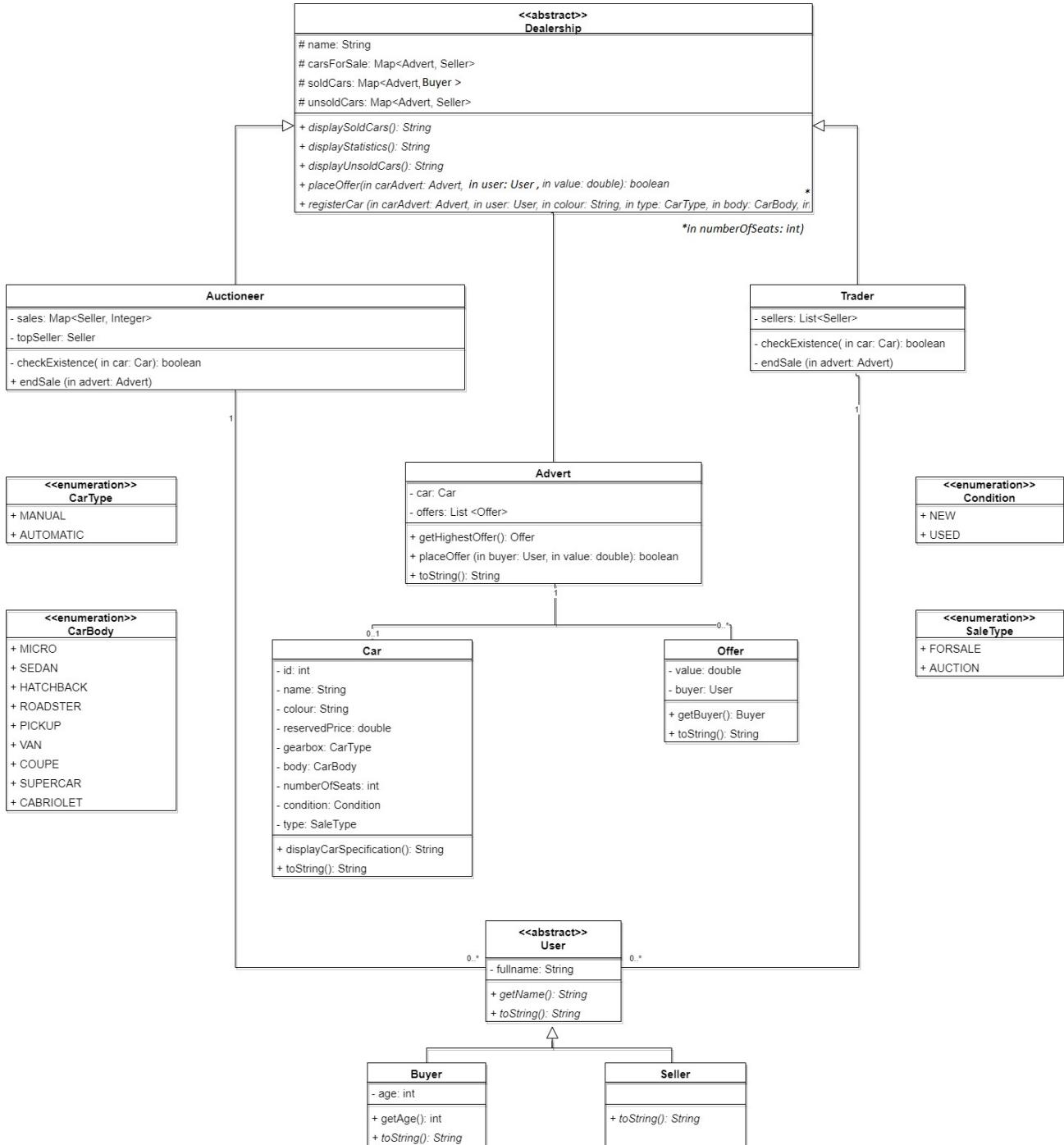
After completing the code for each class, test it against the test classes provided for this question. You don't have to wait until you have completed all the classes before running the tests. Test as you write the code. To do this, right-click on the test files for coursework_question2 and select Run As > JUnit Test.

Remember that if you see any errors in the test classes, this is because you have probably misread the UML diagram or instructions, or forgotten to write the code in the correct package. Missing import statements can also affect the functionality of your code.

Looking at the format of the test classes will give you an insight into how to write correct code, including the expected output or String representation of the toString or display methods.

QUESTION 3 (26%): INTRODUCING MODULARITY

7 Question 3 (26%): Introducing Modularity



UML diagram for Question 3

This question addresses the following functionality: A-P, R (partly), T and U. We are abstracting the code to an abstract **Dealership** class so that it becomes a superclass and then the other two will become subclasses; **Auctioneer** and **Trader**.

QUESTION 3 (26%): INTRODUCING MODULARITY

Copy your source code from the ‘coursework_question2’ folder to the ‘coursework_question3’ folder. Make sure all package names refer to the correct version at the beginning of each class.

At no point will we want to create objects of the Dealership class so it is appropriate to make it an abstract class. In the UML diagram, abstract methods are written in Italics.

Note that some of the methods have already been defined in the subclasses. These need to be modified as required to override the original functionality of the superclass Dealership.

The notion of recording the statistics for each type of dealership is not captured fully in this question either (as it will be covered as an exercise later in Question 4).

Writing Java Code

- Answer this question in the ‘coursework_question3’ package.
- Copy the required classes across from the previous question as your starting point.
- Revise the User class to reflect the structure of the class diagram. This is a new abstract class that defines the structure of its subclasses; Buyer and Seller.
- Define two subclasses Buyer and Seller with the required class definition, fields and methods. The `toString` method in the Buyer class returns the output in the format:

1 S *** a

which represents the initial and last letter of the buyer’s name, and three asterisks.

Whereas the `toString` method in the Seller class uses its fields to structure the output as follows:

1 Bob R. ()

which represents the seller’s name, the initial of their surname, and a set of empty brackets. Note: The `toString` method will later be modified further in Question 4.

- Define a new well-formatted abstract Java class called Dealership which groups together the functionality of the two different kinds of dealership; Auctioneer and Trader. A Dearlership object consists of the common fields and functionality that the subclasses have. This Dealership class provides a parameterised constructor and 5 abstract methods.
 - Revise the Auctioneer class to reflect the structure of the class diagram. The changes include new fields, and a modification to some of the methods. The new map of sales needs to be initialised within the constructor. The topSeller also needs to be defined. (Note: These will only be used in the next question.)
 - Revise the Trader class to reflect the structure of the class diagram. This also includes the update of some methods to reflect the new modularisation. The new list of sellers needs to be initialised within the constructor. (Note: This field will only be used in the next question.)
 - Revise the Offer class to reflect the structure of the class diagram. This includes a change to the `getBuyer` method.
 - Update the Advert class to ensure that the `placeOffer` method only allows a buyer to place an offer.

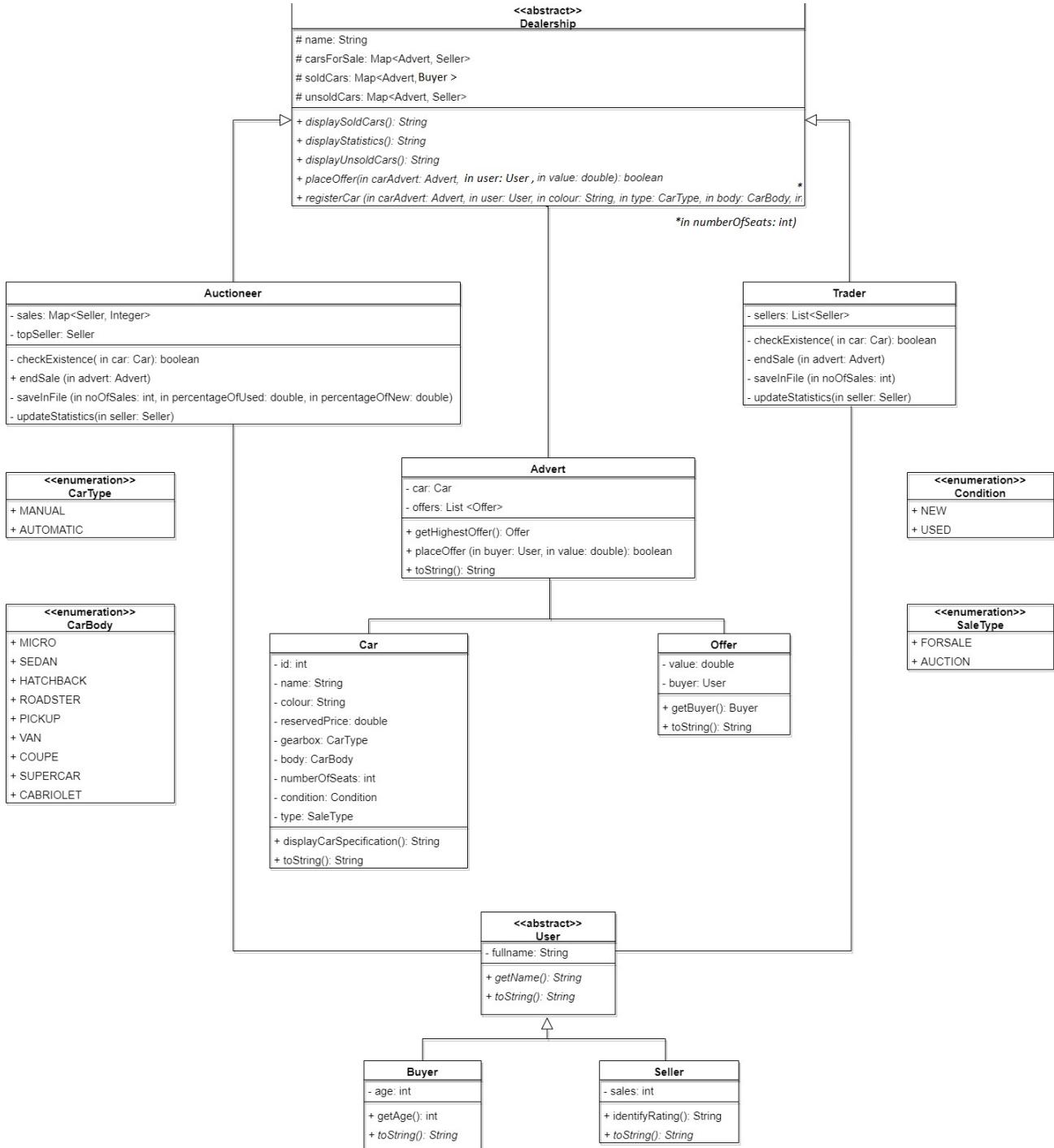
JUnit Tests

All the tests for this question have been written for you. So you can run your code against the tests.

Note that the Dealership class cannot be tested until you have written the Auctioneer and Trader classes. This is also the case with the User class. The Buyer and Seller classes have to be correctly defined first.

QUESTION 4 (21%): COMPLETING THE FUNCTIONALITY

8 Question 4 (21%): Completing the Functionality



UML diagram for Question 4

This question addresses the complete functionality of the proposed system. This includes the following requirements: A-U. Copy your source code from the ‘coursework_question3’ folder to the ‘coursework_question4’ folder. Make sure all package names refer to the correct version at the beginning of each class.

QUESTION 4 (21%): COMPLETING THE FUNCTIONALITY

Writing Java Code

- Answer this question in the ‘coursework_question4’ package.
- Copy the required classes across from the previous question as your starting point.
- Revise the Seller class to reflect the structure of the class diagram. This class now consists of a new field called sales which is responsible of keeping a record of all the sales that a seller has. With each successful sale, this field gets updated. The Seller class also consists of a new identifyRating method that returns the rating of a seller; Level 0, 1, 2 or 3. This is calculated based on the number of sales.

For example:

- 0 sales: Level 0
- Up to 5 sales: Level 1
- Between 6 and 10 sales (inclusive): Level 2
- More than 10 sales: Level 3.

The Seller class also has a `toString` method that needs to be revised to return the following output format:

```
1 Bob R. (Sales: 0, Rating: Level 0)
```

- Revise the Dealership, Auctioneer and Trader classes accordingly, to ensure that the following methods function as expected.

Auctioneer:

- The `updateStatistics` method should update the statistics of the sales for an auctioneer. This includes the total number of AUTOMATIC and MANUAL cars sold, as well as the top seller. The top seller is defined as the one that has the most successful sales, despite the number of adverts. Note: You may define more private methods to support the implementation of this method.
- The `saveInFile` method is responsible for capturing the statistics in a text file which is called ‘`auction_statistics.txt`’. These are stored in the following format:

```
1 Total Auction Sales: 1
2 Automatic Cars: 100.0%
3 Manual Cars: 0.0%
4 Top Seller: Stella K. (Sales: 2, Rating: Level 1)
```

The `saveInFile` method is an example of a method that is only used to aid the functionality of a different method within the same class (thus is set to private). It is introduced to support the functionality of the overall project. This includes writing information in the relevant text file. It can be used to support the implementation of the `displayStatistics` method and/or the `updateStatistics` method.

- The `displayStatistics` method should return the recorded statistics in the following format:

```
1 ** Auctioneer - Auto Auction**
2 Total Auction Sales: 1
3 Automatic Cars: 100.0%
4 Manual Cars: 0.0%
5 Top Seller: Stella K. (Sales: 2, Rating: Level 1)
```

where ‘Auto Auction’ is the name of the auctioneer.

QUESTION 4 (21%): COMPLETING THE FUNCTIONALITY

Trader:

- The updateStatistics method should update the statistics of the sales for a trader. This includes the total number of car sales, as well as the list of all the sellers with their corresponding number of sales and rating (in an alphabetical order). Note: You may define more private methods to support the implementation of this method.
- The saveInFile method is responsible for capturing the statistics in the ‘trade_statistics.txt’ text file. These are stored in the following format:

```
1 Total Sales: 7
2 All Sellers:
3     ↳Bob R. (Sales: 6, Rating: Level 2)
4     ↳Stella K. (Sales: 1, Rating: Level 1)
```

- The displayStatistics method should return the recorded statistics in the following format:

```
1 ** Trader - AutoTrader**
2 Total Sales: 7
3 All Sellers:
4     ↳Bob R. (Sales: 6, Rating: Level 2)
5     ↳Stella K. (Sales: 1, Rating: Level 1)
```

where AutoTrader is the name of the trader.

Notes

Each Auctioneer is responsible for selling cars through an auction. However, the Auctioneer does not own the cars, as the Sellers are the ones that own them.

The statistics are related to the total successful sales that the Auctioneer has had, and they indicate how many of those sales include Automatic/Manual cars. For example, if an auctioneer sells 2 cars (both of which are Manual), then the method will return that they sold 100% Manual cars and 0% Automatic cars.

In terms of the Seller, the method should also indicate the top seller. This is done by identifying who owned the cars of those successful sales. Using the example above, if both Manual cars that were sold belonged to Bob (seller), then he would be named the top seller.

JUnit Tests

All the tests for this question have been written for you. So you can run your code against the tests.

Note that the Dealership class cannot be tested until you have correctly defined the method called displayStatistics in the Auctioneer and Trader classes.

9 Submission Instructions and Checking Your Work

Important

Submit your work as you progress through the coursework. Do not leave it all to the last minute.

For this module, we will be using automated assessment tools for the assessment of the coursework. To achieve this, we will be using the Faculty of Engineering and Physical Sciences (FEPS) GitLab platform to upload code.

In each lab activity so far, you will have completed the following:

1. Downloaded a project from SurreyLearn and imported it into your workspace.
2. Made changes to the project by creating new classes.
3. Used the UML diagrams to convert the structured English language to Java code.
4. Committed the changes to your local repository.
5. Pushed those changes onto your remote repository on GitLab.

In this coursework, you will be expected to take the same steps, in order to download a Java project, make changes to it, and push those changes to your remote git repository. You can access your GitLab repository via <https://gitlab.surrey.ac.uk>, using your university username and password.

Checking Your Work

To check whether the code is functional and correctly structured:

1. Firstly, check the functionality of the project by running the JUnit tests.
2. Right-click on the project name and select Run As > JUnit Test. If any errors occur, select the relevant file and check the error and identify a way to resolve it.
3. Should there be any questions about the coursework itself, please use the relevant MS Teams channel. Only use the 'Assessment' channel for such queries (without sharing any of your code).
4. Then test your entire project. To do this, right-click on the project name and select Run As > Maven Test. This should return the message 'BUILD SUCCESS' on the console.

Submitting Your Work

Your final submission must be completed through GitLab:

1. Save all your changes.
2. Commit and push all the changes made to your remote repository. To do this in Eclipse, select the Git Staging view via the menu Window > Show View > Other... and then Git > Git Staging.
3. Add all the unstaged changes, and commit and push the changes to the remote repository.

SUBMISSION INSTRUCTIONS AND CHECKING YOUR WORK

Note: Additional submission advice and help will be released on SurreyLearn under the coursework section—keep an eye on this.

Important

It is your responsibility to submit your work on time. Normal lateness penalties apply. Internet or personal git problems are not valid excuses for late submissions.

Do not leave pushing your first coursework commit to the last minute. Commit as you work through the coursework, so you can identify any issues early on. Additional materials on git and GitLab will be released, although how extensively and well you use git and its functionality will not be assessed. Submission via git is a requirement of the assignment and has been covered in taught content.

Please be sure to make good use of support provided in the labs, tutorials, drop-in support sessions, and office hours for this module, as well as the many resources provided on SurreyLearn.