FIT2099 S1 2022

Online Forum (23/5/2022)

Exam Revision guidelines:

- Revision Learning materials (EdLessons & bootcamps & assignments) based on the learning outcomes.
- Listen to the "video recording", if needed.
- Focus on understanding and applying the concepts.
- Try out the sample exam questions available in Scheduled Final Assessments tab, Moodle. (The sample solutions and marking rubric are given)

Consultation hour : **Week 13 : 30 May 2022, 12 noon – 1pm.**

Exam Method Hints:

- Do easiest questions (for you) first.
- Focus on answering the questions.
- Clearly label which question & sub-question you are answering in the examination answer sheets.
- Write clearly, precisely, and concisely.
- DON'T PANIC.

Revision

Bootcamp 2

There are 5 tasks in this bootcamp. Task 1 and 5 have instructions related to UML diagram and Git. The following images are captured from the Moodle.

Task 2. Start the implementation

Even when you have a complete design, it is usually a good idea to implement and test your program bit by bit.

Create a new project called **CarAuction** using your preferred IDE (as mentioned last week, we recommend IntelliJ IDEA) and add the following class.

```
public class CarAuction{
    public void printStatus(){
        System.out.println("Welcome to FIT2099 Car Auction Systems");
        System.out.println("Thank you for visiting FIT2099 Car Auction System");
    }
}
```

As you can see, all the class can currently do is print a welcome message and goodbye message.

Once you have added the class to your project, create a new **CarAuctionDriver** class. This class should contain only a **main(...)** method that creates a **CarAuction** object and calls its **printStatus** method. Run your program and confirm that it works as expected. If you have questions or difficulties then, make an EDiscussion form post, attend a consultation, or ask your lab TA (tutor).

Task 3. Extending the implementation - adding a Car class

The system needs a **Car** class. Each Car object must know its make (e.g. "Audi"), its model (e.g. "TT") and the model year (e.g. 2014). Make sure the model year is of type Integer. The Car class should also have a getCarDescription() method, that returns a string of the concatenated model year, car maker and model, e.g. "MY2014 Audi TT". Make sure the acronym "MY" (model year) appears just before the year. To test your Car class, add some code to CarAuction printStatus() that

- creates a Car object and
- prints its description using the getCarDescription() method.

Confirm that your program is producing the correct output before proceeding to the next task.

Task 4. Arrays of Cars

Now, we would like the system to be able to handle an arbitrary number of Cars. Modify the CarAuction class so that instead of containing a single Car, it contains an array of Cars (Java array not ArrayList).

Add two new methods, createCars() and displayCars(), to CarAuction.

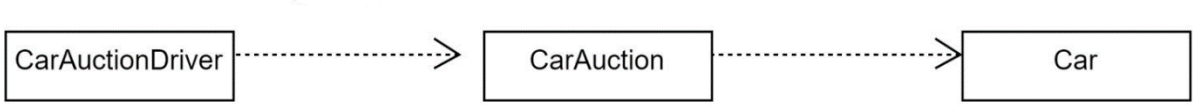
- createCars() must create three new Car objects, add car maker, model and model year to them, and store them in the array.
- displayCars() must display the descriptions for the cars, which you can generate by invoking getCarsDescription() on each of the cars in the array. Try to use a loop rather than repeated code, and try to avoid hardcoding the array length into displayCars.

Modify the printStatus() method to display the welcome message, call the createCars() method, call the displayCars() method, and then display the goodbye message.

Expected output:

```
Welcome to FIT2099 Car Auction System
Car (1) MY2017 BMW X7
Car (2) MY2014 Audi TT
Car (3) MY2020 Chevrolet Corvette
Thank you for visiting FIT2099 Car Auction System
```

Referring to the UML class diagram below (a sample of incorrect solution),

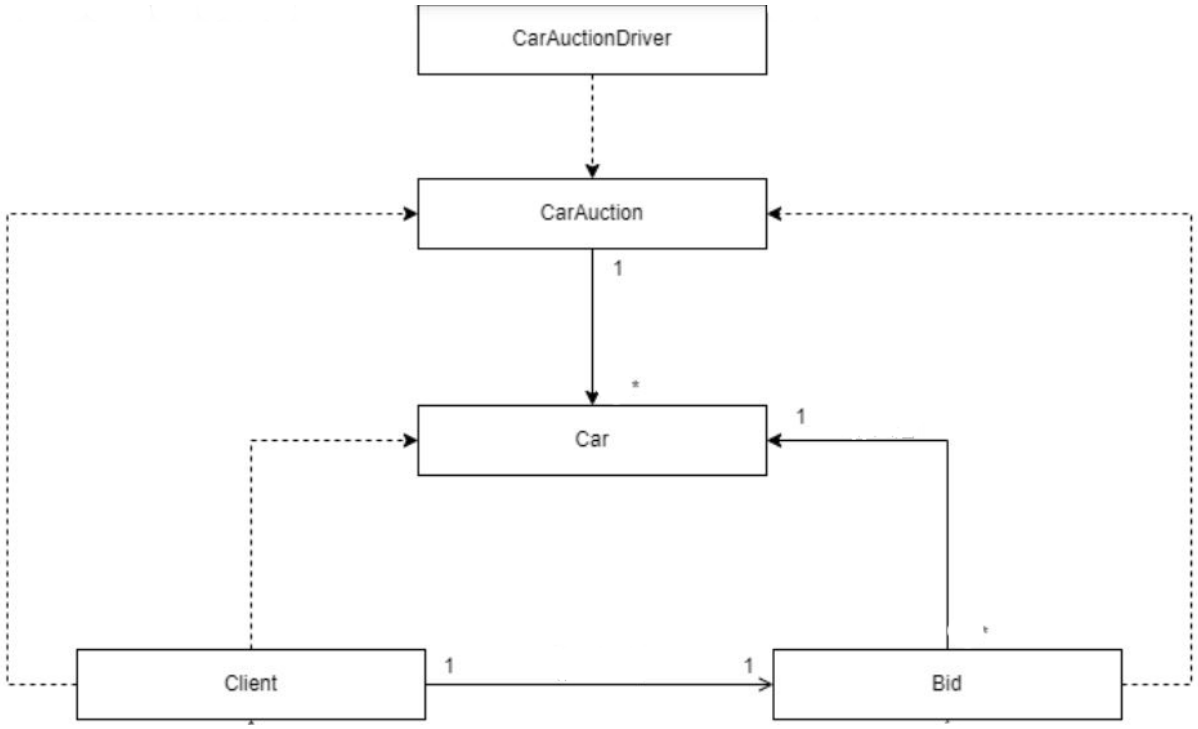


Mistake(s): Incorrect relationship between CarAuction and Car (Dependency).

Correction(s): Relationship between CarAuction and Car should be association (Solid line instead of dotted line). [CarAuction] -> [Car] association.

Bootcamp 3

There are 9 tasks in this bootcamp. Task 1 and 9 have instructions related to UML diagram and Git. Referring to the UML class diagram below (a sample of incorrect solution),



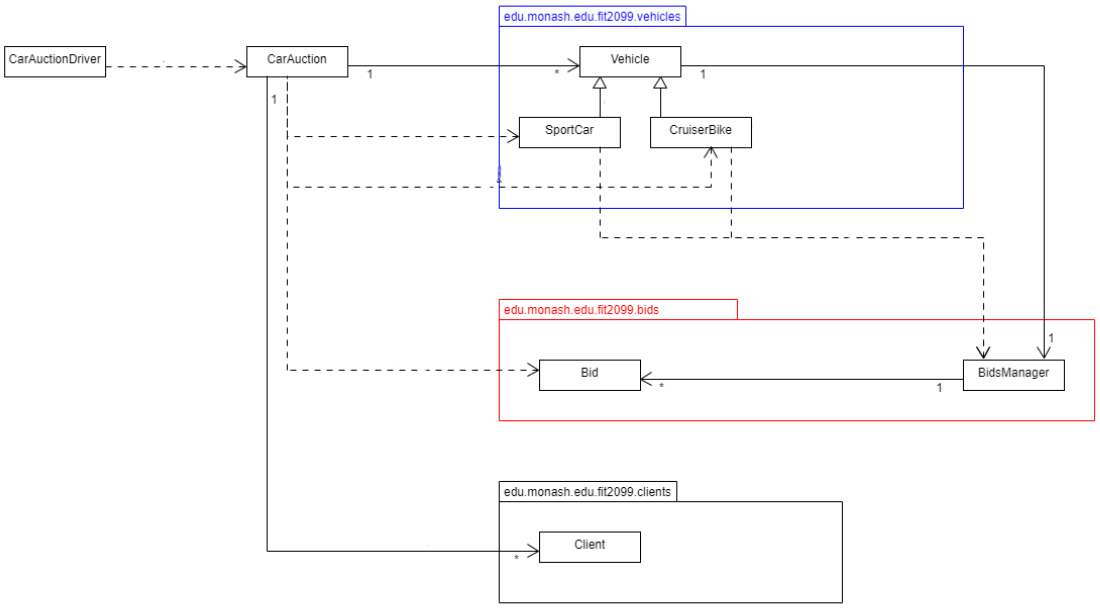
Mistake(s): Incorrect relationship (arrow direction) between Car and Bid, Client and Bid, CarAuction and Client, CarAuction and Bid.

Correction(s): Change the arrow direction for those identified mistakes. [Car]->[Bid] association, [Bid] ->[Client] association, [CarAuction] ->[Client] dependency, [CarAuction]->[Bid] dependency.

Bootcamp 4

There are 7 tasks in this bootcamp. Task 1 and 7 have instructions related to UML diagram and Git.

Referring to the UML class diagram below (a sample of incorrect solution),



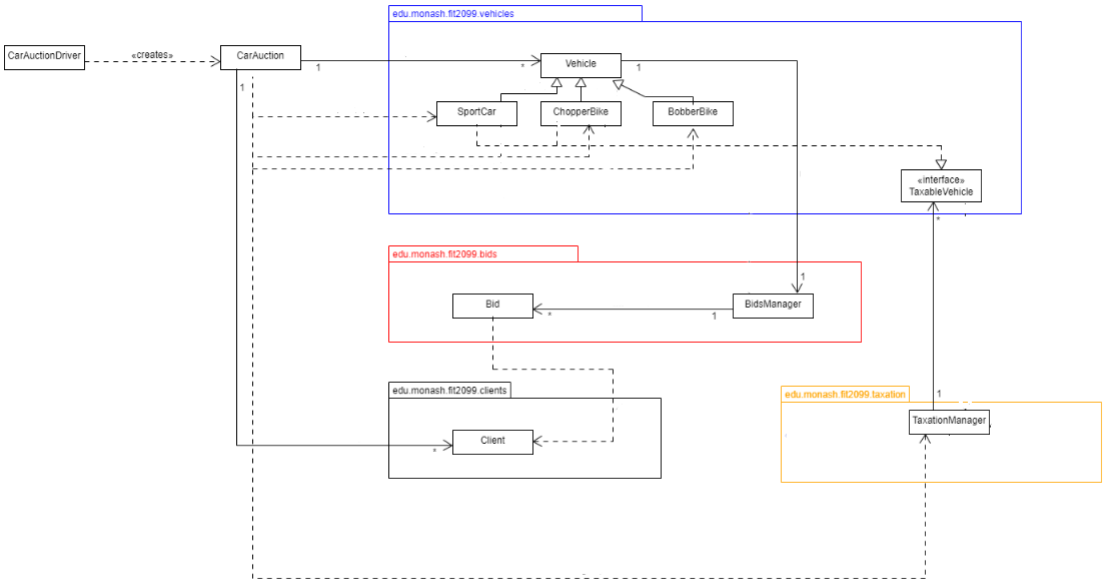
Mistake(s): Incorrect relationship between SportCar and BidsManager, CruiserBike and BidsManager. Missing <<abstract>>.

Correction(s): Remove the relationships. Add the keyword "<<abstract>>" to Vehicle OR Italic the Vehicle.

Bootcamp 5

There are 8 tasks in this bootcamp. Task 1 and 8 have instructions related to UML diagram and Git.

Referring to the UML class diagram below, do you find any mistake(s)?



Mistake(s): Missing <<abstract>>.

Correction(s): Add the keyword "<<abstract>>" to Vehicle OR Italic the Vehicle.

Bootcamp 5.

SOLID principles

To adhere to Object-oriented design principles,

The Vehicle class is an abstract class, which has subclasses SportCar, CopperBike and BobberBike. This design follows the DRY principle since SportCar, CopperBike and BobberBike share the similar attributes (maker, model, makeYear) and behaviours (setters) of superclass (Vehicle). SportCar, CopperBike and BobberBike have specific responsibility and implemented as classes separately as per single responsibility principle (SRP). (*should be reflected in UML diagram and implementation.)

```
public abstract class Vehicle {
   private String maker;
   private String model;
   private Integer makeYear;
   //state other attributes

public Vehicle(String maker, String model, Integer makeYear)
public boolean setMaker(String maker)
// state other setter
//state getter
//state toString()
```

Open-closed principle (OCP) can be applied to abstract class Vehicle. More classes (SportCar, CopperBike,…) can be extended from abstract class Vehicle and the extension doesn't change the way the existing code used by the client(s) or doesn't change the existing source code. SportCar and CopperBike can add new method(s) by implementing TaxableVehicle interface. (*should be reflected in UML diagram and implementation.)

```
public class SportCar extends Vehicle implements TaxableVehicle
public class ChopperBike extends Vehicle implements TaxableVehicle
```

SportCar, BobberBike, and CopperBike share some characteristics and are designed as child class of abstract class Vehicle. Replacing SportCar, BobberBike and CopperBike with abstract class Vehicle should not result in any unexpected behavior. This implements the Liskov substitution principle (LSP).

Proof with implementation/code.

```
//toString() method
```

TaxableVehicle interface is an interface contains only the methods for taxable vehicle. SportCar and CopperBike (implement TaxableVehicle) do not depend upon unnecessary interfaces that they do not use. This adhere to Interface Segregation Principle(ISP). (*should be reflected in UML diagram and implementation.)

```
public class SportCar extends Vehicle implements TaxableVehicle
public class ChopperBike extends Vehicle implements TaxableVehicle
```

```
public interface TaxableVehicle {

    public double calculateTaxRate(double price);

    default void addToTaxationManager(){
        TaxationManager.getInstance().appendTaxableItem(this);
    }
}
```

Assignment 1, 2
Sample
(**Analyze the scenario)

Separation of concern/Single responsibility principle
(have a specific, well defined responsibility)
The new classes/interfaces

- Item: Coin, PowerStar, SuperMushroom, Wrench,
- Actor: Toad, Goomba, Koopa,
- Grounds: Dirt, Floor, Wall, sprout, sapling, mature
- Actions: Attack, Consume, Jump, KillDormant, PickCoin, Purchase, Reset, Talk
- Monologue
- Interface: Speakable (such as generateMonologue method), Purchaseable/Buyable (such as addInto method), Resettable (such as resetInstance method), consumable (instead of using ConsumableItem – abstract class).
- Inheritance: Enemy(abstract class), Goomba, Koopa. Tree(abstract), sprout, sapling, mature. ConsumableItem(abstract class), PowerStar, SuperMushroom.

The Enemy class is an abstract class, which has subclasses Goomba and Koopa. This design follows the DRY principle since Goomba and Koopa share the similar attributes and behaviours of superclass (Enemy). Goomba and Koopa have specific responsibility and implemented as classes separately as per single responsibility principle (SRP). (*should be reflected in UML diagram and implementation, State the pros and cons.)
public abstract class Enemy extends Actor implements Resettable
public class Goomba extends Enemy
public class Koopa extends Enemy

OCP is implemented via– abstract classes(inheritance) and interfaces.
Open-closed principle (OCP) can be applied to abstract class Enemy. More classes (Bowser,…) can be extended from abstract class Enemy and the extension doesn't change the way the existing code used by the client(s) or doesn't change the existing source code. Enemy can add new method(s) by implementing Resettable interface. (*should be reflected in UML diagram and implementation, State the pros and cons.)

LSP is implemented via – inheritance.
// Enemy abstract class. Replacing Goomba, and Koopa with abstract class Enemy should not result in any unexpected behavior - playTurn()

ISP is applied through interfaces.
Purchaseable interface. Purchaseable interface is an interface contains only the methods for player to add the item(s) to inventory.
public abstract class ConsumableItem extends Item implements Purchaseable

By using interfaces (adhere to DIP), for instance, purchaseable interface is implemented by all the classes that Player can purchase via the PurchaseAction class.