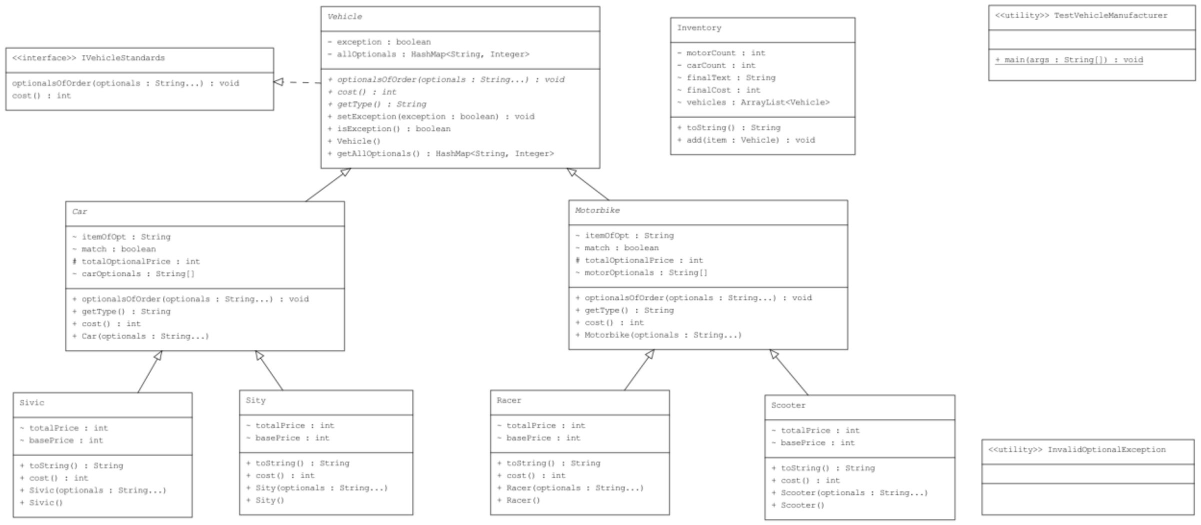
CENG 1004 Spring 2021 Final Exam Documentation

a)



<<interface>> IVehicleStandards
optionalsOfOrder(optionals : String...) : void
cost() : int
Vehicle
- exception : boolean
- allOptionals : HashMap<String, Integer>
+ optionalsOfOrder(optionals : String...) : void
+ cost() : int
+ getType() : String
+ setException(exception : boolean) : void
+ isException() : boolean
+ Vehicle()
+ getAllOptionals() : HashMap<String, Integer>
Inventory
- motorCount : int
- carCount : int
~ finalText : String
~ finalCost : int
~ vehicles : ArrayList<Vehicle>
+ toString() : String
+ add(item : Vehicle) : void
<<utility>> TestVehicleManufacturer
+ main(args : String[]) : void
Car
~ itemOfOpt : String
~ match : boolean
# totalOptionalPrice : int
~ carOptionals : String[]
+ optionalsOfOrder(optionals : String...) : void
+ getType() : String
+ cost() : int
+ Car(optionals : String...)
Motorbike
~ itemOfOpt : String
~ match : boolean
# totalOptionalPrice : int
~ motorOptionals : String[]
+ optionalsOfOrder(optionals : String...) : void
+ getType() : String
+ cost() : int
+ Motorbike(optionals : String...)
Sivic
~ totalPrice : int
~ basePrice : int
+ toString() : String
+ cost() : int
+ Sivic(optionals : String...)
+ Sivic()
Sity
~ totalPrice : int
~ basePrice : int
+ toString() : String
+ cost() : int
+ Sity(optionals : String...)
+ Sity()
Racer
~ totalPrice : int
~ basePrice : int
+ toString() : String
+ cost() : int
+ Racer(optionals : String...)
+ Racer()
Scooter
~ totalPrice : int
~ basePrice : int
+ toString() : String
+ cost() : int
+ Scooter(optionals : String...)
+ Scooter()
<<utility>> InvalidOptionalException

b)

I think the most important difference between an abstract class and a concrete class is that a concrete class is a subclass of an abstract class, which implements all its abstract methods. Therefore, it's easier for us to implement new sub-classes (in our case, vehicle types) under an abstract class. Vehicle, Car and Motorbike classes are general types and there might be a new vehicle type added under the Vehicle class, a new car model under the Car class or a new motorbike model under the Motorbike class. Therefore, they can be created as abstract classes, just like I did.

c)

Encapsulation is keeping precious data safe and private in short. I've used encapsulation in Inventory and Vehicle classes via using "private" keyword. To use those data, I've implemented getter and setter methods.

d)

Inheritance can be used to benefit from similarities and differences of objects. Similarities are defined in an upper class in hierarchy and it reduces the code repetition. I've used Vehicle class as the top class and Car/Motorbike classes as mid classes. All upper classes are used whether to check or assign similar parts of our objects.

E)

Final keyword can be used to assign a constant. I don't think I could assign any variable as final. I could've assigned vehicles' base prices as final but in case of a raise situation, that could be a problem.

f)

```
Vehicle carOrder = new Sivic( "ABS", "Music System", "Airbag");
order.add(carOrder);

carOrder = new Sivic("ABS", "Sun Roof");
order.add(carOrder);

carOrder = new Sity("Music System", "Sun Roof");
order.add(carOrder);

Vehicle motorbikeOrder = new Racer("ABS", "Seat Heating");
order.add(motorbikeOrder);

motorbikeOrder = new Scooter("Seat Heating");
order.add(motorbikeOrder);
```

carOrder and motorbikeOrder are polymorphic objects because they keep changing their classes in between different sub-classes of Car and Motorbike respectively.