Car Configuration Application

Reflection Questions on Unit 1

You should review following questions, to make sure you understand the outcomes from Unit 1. You should document lessons learnt for submission (with final unit of Car Configuration Application). You do not submit these questions for grading:

1. What is the relationship between containment and encapsulation (as applied in this project), when building components?
2. What are some ways to analyze data (presented in requirements) to design Objects?
3. What strategies can be used to design core classes, for future requirements, so that they are reusable, extensible and easily modifiable?
4. What are good conventions for making a Java class readable?
5. What are the advantages and disadvantages of reading data from sources such as text files or databases in a single pass and not use intermediary buffering?
6. What is the advantage of using Serialization? What issues can occur, when using Serialization with Inner classes?
7. Where can following object relationships be used: encapsulation, association, containment, inheritance and polymorphism?
8. How can you design objects, which are self-contained and independent?

Requirements

Part A

In this assignment, you will continue to build the application, for configuring the car:

I would like you to expand your proof of concept, by building API's for car configuration classes, using interfaces and abstract classes. You will also add a custom exception handler, to enhance your design.

For expanding proof of concept, please consider the following requirements:

- Define a set of methods in an interface (as API), to exercise the functionality of the existing class set.
- Create an exception handler, which handles at least 5 exceptions.
- Enhance your design and code to create any abstract classes for extensibility and reusability.

Deliverable:

Design and code classes for above requirements and write a driver program, to exercise API and test the exception handler. Test your code adequately.

Concepts you will need to know:

- Object Theory
- Exception Handling
- Abstract Classes
- Interfaces

Plan of Attack - Part A

Refactoring

All class names should be declared as nouns. In the last unit, we created Automotive, which needs to be renamed to Automobile. Please complete this step before approaching this unit. Please do this with extreme care using IDE's refactor feature.

Step 1:

Writing API – Design and Construction

1. Analyze what you are being asked to do.
    a. You are being asked to build API, for car configuration classes.
    b. Develop a custom exception handler to make your module more reliable.
2. From the information provided in the lab description, following are the considerations for developing the API:
    a. Use Interfaces.
    b. You have Model package (a component), which has Automobile class. Each Automobile class contains an instance of OptionSet and respective Options.
    c. In first unit, you were asked to make the methods in OptionSet and Option class protected. Methods of Automobile class only are set as public.
    d. We are now trying to provide a set of methods, which act as a Programming Interfaces for accessing functionality, in Model package.
    e. Before we decide on how to create and implement the methods, let's read through and understand the meaning of a Java Interface. Please review the article at - http://docs.oracle.com/javase/tutorial/java/concepts/interface.html, to learn about the importance of Interfaces.
    f. Now let's decide on methods, which should be exposed in the Interface.

```
public void BuildAuto(String filename);
```

//Given a text file name a method called BuildAuto can be written to build an instance of //Automobile. This method does not have to return the Auto instance.

```
public void printAuto(String  Modelname);
```

//This function searches and prints the properties of a given Automodel.

```
public void updateOptionSetName(String Modelname, String OptionSetname, String newName);
```

//This function searches the Model for a given OptionSet and sets the name of OptionSet to //newName.

```
public void updateOptionPrice(String Modelname, String Optionname, String Option, float newprice);
```

//This function searches the Model for a given OptionSet and Option name, and sets the price to //newPrice.

For our implementation we will implement these four methods.

g. Few considerations for structuring the code:

i. It is not important for us to expose the Automobile instance. We can create an Automobile instance and provide all the required functionality in the chosen API in the last step.

ii. The code should be organized to encapsulate Automobile:

- Create a new package called Adapter.
- Add two interfaces in package called Adapter:
  - CreateAuto and add BuildAuto and printAuto methods in it.
  - UpdateAuto and add updateOptionSetName and updateOptionPrice in it.
- Add a new abstract class called proxyAutomobile in Adapter package. This class will contain all the implementation of any method declared in the interface.

```
package Adapter;

import Model.*;

public abstract class proxyAutomobile {

      private Automobile a1;

}
```

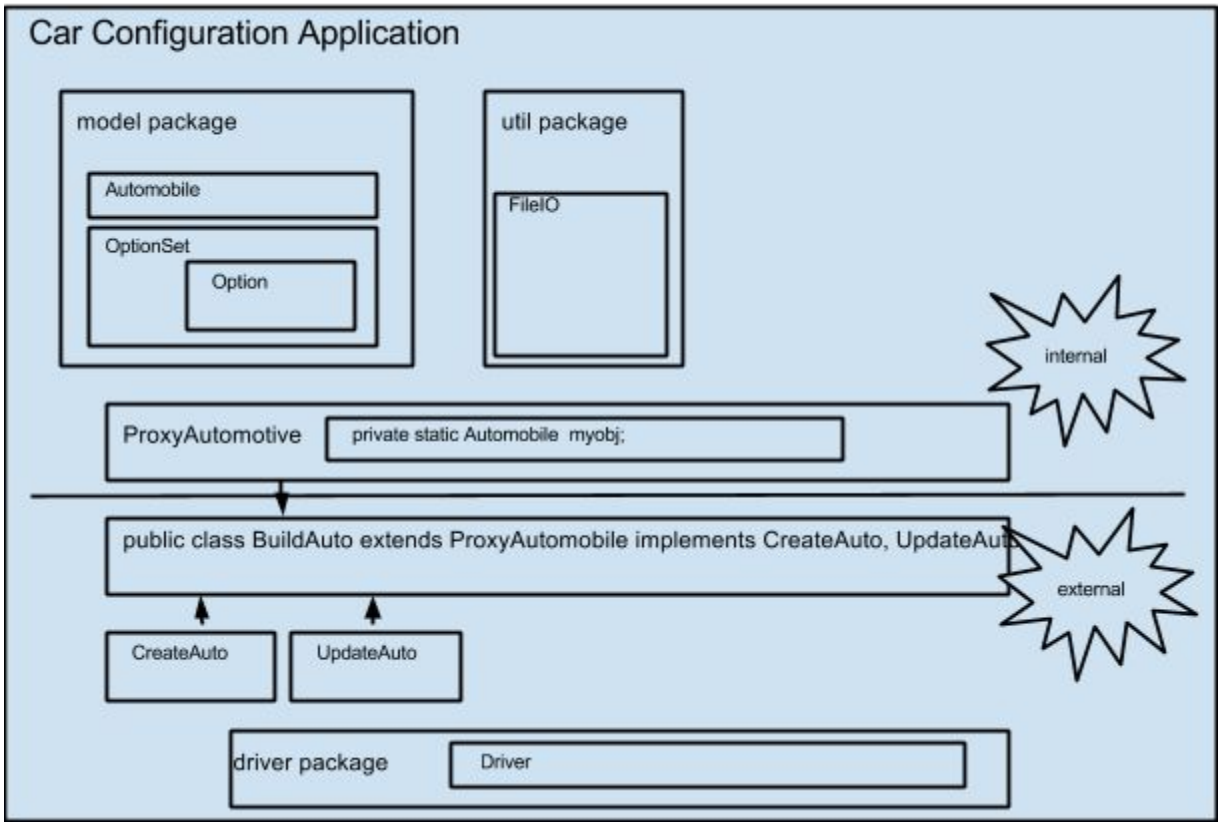
This class contains an instance variable of type Automobile. Variable a1 can be used for handling all operations on Automobile as needed by the interfaces.

So this raises a question – if we are defining methods, declared in interfaces in the abstract class proxyAutomobile, then why is proxyAutomobile not implementing the interfaces. We will use inheritance for this purpose. We want proxyAutomobile to be “hidden” containing all the functionality and expose an empty class, for usage with Interfaces. In other words, we will add a new class called BuildAuto in Adapter package that will have no lines of code, but will always look like this.

```
package Adapter;

public class BuildAuto

     extends proxyAutomobile implements createAuto, updateAuto{

}
```

Whenever a new interface has to be added, you can simply update BuildAuto declaration and write all methods in Abstract class called proxyAutomobile.

In a nutshell, we have encapsulated access to Automobile instance, from the API and also hidden the code (artificially) in the abstract class.



Next write a driver to test each of the methods:

1. Are you able to create and print an Auto instance through CreateAuto interface?
2. Are you able to update one of OptionSet's name or Option Price for the Auto instance, created in the previous step.
3. You will not be able to update, as the object in proxyAutomobile is not declared as a "static" Object. In other words, when you create an instance for BuildAuto (child of proxyAutomobile), a new Automobile is created. This requires variable a1 to be static, so it can be shared between objects.

Step 2

Defensive mechanisms to make software Self-Healing

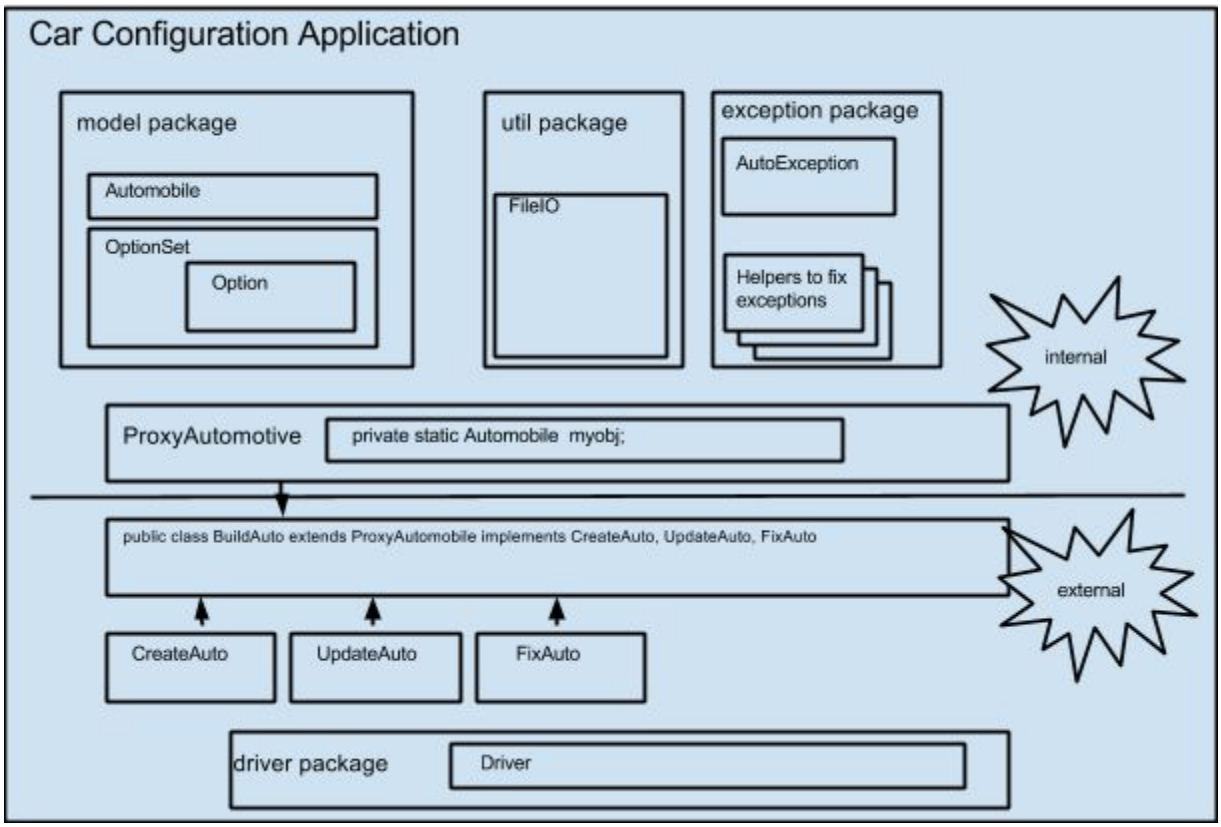
Our next step is to add a custom exception handler to deal with issues in runtime. For doing this, you will first need to review the custom exception handling classes provided. After you have learnt how to write a custom exception class, throw an exception and how to catch and fix, you can implement it in this project.

Think of five possible exceptions to fix - Here are some possible examples:

1. Missing price for Automobile in text file.

2. Missing OptionSet data (or part of it).
3. Missing Option data.
4. Missing filename or wrong filename.

Your Exception class at a minimum should handle and fix at least one exception.



AutoException should have following features:

1. Ability to track error no and error message.
2. Contain an enumeration of all possible error numbers and messages, which can be used, when AutoException is instantiated.
3. Ability to log AutoException with timestamps into a log file (you do not need to implement any complex logging mechanism).
4. Write helper classes to delegate fixes for each method. For example, if exception number 1 to 100 is assigned to model package, you might author a class called Fix1to100 as a helper class for AutoException, which contains fix methods for exceptions raised in the model package.
5. AutoException should have following implementation of fix method, which can be used for fixing any exception in entire application.

```
public void fix(int errno)

{
```

```
Fix1to100 f1 = new Fix1to100();

switch(errno)

{

case 1: f1.fix1(errno);break;

…

}

}
```

6. Next, make the fix method accessible through FixAuto interface.

Submitting your work

Please review your work against this checklist before submission:

1. Program Specifications / Correctness
    a. No errors; program always works correctly and meets the specification(s).
    b. The code is reusable - as a whole or each routine.
    c. Custom Exception Handler has been implemented with the usage of five minimum exceptions.
    d. Abstract Classes constructs are utilized to add extensibility.
    e. Interfaces utilized to expose Model's (Auto, OptionSet and Option) functionality is meaningful (means the methods in interface are well thought out/useful in context of application).
2. Readability
    a. No errors; code is clean, understandable and well-organized.
    b. Code has been packaged and authored based on Java coding standards.
    c. Text input file is readable and easy to follow.
3. Documentation
    a. The documentation is well written and clearly explains what the code is accomplishing and how.
    b. Class Diagram is provided.
4. Code Efficiency
    a. No errors; code uses the best approach in every case. The code is extremely efficient, readable and understandable.
    b. Ability to correct exceptions is added in exception handling class. Ability to log exceptions is added.