Friday, June 23, 2017

Lessons Learned — Nina Maller

Java

- Lab 1
- Serialization: when serializing an object, I have to write serializable in the child classes as well. The Automobile object cannot get serialized if OptionSet and Option classes do not each “implements Serializable”.
- It is a good habit to have to write a toString for every object. This way, coding is easier later on.
- Encapsulation is about making the object private, hiding its contents. Containment is when one object contains the other, but the contained object is not necessarily hidden. In my project, OptionSet and Option are hidden by making them private, so only Automobile class has access to them.
- To make the code readable, it is important to write many comments, keep clear spacings, and write code that is effect while it is also clear to the reader (sometimes there is no reason to make the code much more complicated to understanding if it is only a little more efficient; it is a tradeoff).
- Serialization makes sending and accepting objects very easy and comfortable. By using serialization, I can restart my last program from the point I left it, without the need to add all the objects over again.
- Reading from a text file is limited, but it is a good way of organizing all the data for each car. In my project, I made the files very engineer-oriented, because it is easier for the engineer to read, while the client has to enter not necessary information, like number of options for each option set.
- Designing objects that are self-contained and independent is the goal of object oriented design. The Automobile class is independent because it has all it needs - it has the OptionSet and Option classes in it.
- Encapsulation is used in every class, where data is being hidden in order to protect it. Data is then accessible by getters and setters. Association is when there is some relationship between classes through their objects. An example is the Driver and the Automobile class. Containment is when a class contains another class, like Automobile that contains OptionSet. This is a good way to encapsulate OptionSet, by making it private (only accessible through Automobile). Inheritance is used when a

class is reusing code from another (parent) class. While it reuses, it also adds new code or information. Polymorphism is the concept of a parent class that can have many child classes. So, many classes can inherit from one class (but they can inherit from only one class, there is no multiple inheritance).

- Lab 2
- We use abstract class to provide a base template for a class. It is a contract for implementation - abstract methods. It also provides reusability. class definition is incomplete so it can exist on its own.
- I cannot instantiate an abstract class. But it can have constructor, to initialize its own properties. This way it is self contained.
- Interfaces are very important. They allow implementation across different families, polymorphism, reusability. They can restrict genericity by specifying. which data type has to be accepted, and not leaving it as a template. They are used to recuse global constants.
- Since java does not allow multiple inheritance (one class cannot extend more than one parent class), the only way to implement multiple inheritance is by interfaces. A class can extends as many classes as it wants. This way, there is a contract that the class has to implement the method (but it is really not multiple inheritance).
- Another important goal of interfaces is controlled access to methods. When we use ProxyAuto, we don't want all the classes to have access to all the methods there. So, by enabling the access only by interfaces, we can control which client does what. For example, Ford can only add cars to the list. While on the other side, a client of KBB can only edit a car, but not add a new car. This is role based access to functionality. RBA - role based access, different interfaces for services.
- Abstract classes form a bridge between internal and external worlds of a programming context.
- The return type of methods in an interface can be void so it will be overridden. We don't always know what object will be used.
- BuildAuto class is empty in order to encapsulate ProxyAuto.
- Proxy Auto will have implementation of all methods in all interfaces and it is called spaghetti code. It will use code from internal components, integrating internal to external. Acts as a bridge between internal and external

- The purpose of ProxyAuto is - **SSOT** - single source of truth.
- Self healing software is about handling expetions. There are two types for exception handling: errors (no recovery), and exceptions (recovery). Exceptions are ones you know about and ones that you will make unknowingly.
- We implement self healing software by try and catch blocks. This way, we can predict an exception and fix it, so the program can continue and not stop every time there is a little mistake.
- If the exception will not cause the program to crush, we might ignore it (although it is better to handle all exceptions).
- Whenever we get an exception, it is important to log it. Then, we make a report of all the exceptions - writing the exception number, the name, and the time. Every time something stops working we have some data to analyze (real world application).
- Some exceptions I handled is missing data (I checked if name is missing, or if space are entered instead of the name), a base price that cannot be negative, data mismatch, and more.

- Lab 3
- In lab 3 it was important to learn how to use the right data type. One way to decide is to look at the ratio of create and read. How many times to I enter the list vs how many times I add to the list? It is a good idea to write a program that counts inserting and reading elements from lists and prints the time takes. It is very obvious which are efficient and which are not.
- Another consideration is whether the list is sorted or not. Should it be sorted? If we only add, and not read a lot, then not sorted is fine. Also, if there are not a lot of elements, then it doesn't matter. However, if we read a lot, the list must be sorted in order to save time.
- When the rate of insertion is high, as well as the rate of reading, a hash map is a good idea because it is very fast.
- In order to work with the linked hash map we have chosen to store the cars, we have to add an iterator to go through all the elements.
- The linked hash map has to be static. Because we enable the access to proxy auto through interfaces, different Automobile objects will change only their stance of

Automobile. however, when we make the list Static, there is only one copy of each car that exists in the whole program. This way, we can create a car with one interface, edit it with another, and the same car will be changed.

- Changing from an array to an ArrayList takes time. instead of writing array[i], we have to change it to arraylist.get(i), but that returns an Object so we have to cast it to be OptionSet. Final change: ((OptionSet) optSet.get(i))
  Fix: I shouldn't have forgotten to write ArrayList<OptionSet> = new ArrayList<>();
  This way we don't have to cast.

- Lab 4

- Is it better to lock an object or a method? it depends on which takes a shorter time. For this lab, it is better to lock the method.

- Where does the synchronization happen?
  Not in model package, because that already works and we don't want to start recharging everything.
  Not in proxyAuto (locking the LHM), because everything happens there. The object is constantly alive in the program. It is more efficient to lock the methods, which are, in editOption.

- Getting access to ProxyAuto was difficult. It was either by subclassing, or by using an interface. Subclassing did not work because a subclass does not inherit the private variables of its parent class. So subclassing ProxyAuto will not give access to the LHM.

- What is the relationship between the editOption and proxyAuto? it can be a one way relationship, using extend, or a two way relationship using interfaces. I chose a relationship based on interfaces, which provided me the access to the linked hash map (although I had to add many methods).

- Synchronization allows us to make sure that we will not have dat corruption. However, since changing the color of a car to another color is a very short thread, I had to do something else to test if the synchronization works. I made a method which changes 3 things- make, model, and price. However, between each change I waited for some random amount of time. Without synchronization, the car received many different variables (as Mazda Z4, 35000), while after synchronization, the car always had the right matching values.

- Multithreading was added to the project, but it is only an option. By creating a new package, I made sure that the project still works as is, but it is now extended (instead of replaces) to have the option to use threads.
- Multithreading can provide access to more than one client, which extends the projects.

- Lab 5
- It is important to remember to first write the output stream and then the input stream.
- In order to start the project, I first nee to run the server and then the client.
- At first I thought that I cannot use the same port more than once, and I was constantly changing it and restarting the program. However, you can, I just have to make sure that all of the running projects are terminated (by going through all the open consoles).
- It is a good idea to check if the bufferedReader is ready, using the ready() method which returns true or false.
- the key in the property file cannot have spaces (it took me hours to find this out). I read that the property file divides the key and the value using “=“, so I assumed that spaces would not change anything. I was very wrong.
- Using PrintWriter is a good idea because it automatically flushes, so my program doesn't crash. However, I can use any other output stream and just remember to flush after I write something to the server or the client. Also, PrintWriter flushed automatically after println(), not print().
- When I serialize and send between client and server, classes must be exactly the same (for Automobile). So, if I change a little (adding one method) the Automobile class in the client, it won’t be the same as the Automobile class in the server, and I will not be able to accept an Automobile object through the stream. The exception is type mismatch.
- Lab 5 had another use of interfaces, in order to have a set of constant available at all packages. One place for all constants is very convenient.
- Before doing lab 5, I already had all the code. Lab 5 was about understanding what is happening, and being able to copy the right parts of the instructor’s code and making it work. Although it sounds easy, it was not. Since I wasn't writing a lot of the code by

myself, I found myself very confused. I ended up reopening sockets — by that, I couldn't understand where the mistake in the code is, because all the connections opened correctly and the program was not working.

- Lab 6
- Lab 6 showed the importance of good design in the previous 5 labs. If I designed a good API for lab 5, lab 6 is easier because we already have the methods. However, a bad design costs a lot of time in adding and changing existing methods.
- In a MacBook, all files that start with a dot are hidden (this is why I couldn't find .metadata in the workspace). To fix it, I had to go to the terminal and write defaults write com.apple.finder AppleShowAllFiles YES, and by this unhide all the documents. Since eclipse forgets to transfer some files when working with Tomcat, I had to manually replace ROOT folder.
- In lab 6, the client of lab 5 becomes the server for lab 6. However, it must become a web project. It is possible to easily convert the existing java project to a web project by going to the properties, clicking at “Project Facets” and adding to Java the Dynamic Web Module. This way, java automatically generates all web related projects.
- Many times, index.html file is used as a directory from which the web program starts. Since I first had to start the server, then the web app, and then upload the vehicles from Eclipse, I thought that using the index.html is a good design. This way, the client presses start and is told to go to Eclipse and upload the vehicles. In the future, I can add other options in this page.
- There are few ways to pass input through the URL. One thing I learned is that there cannot be any spaces, so my vehicle options have _ instead of spaces. (I hope to fix it before uploading the project to my resume). Also, there are few ways in sending the data. I used GET because the data is not sensitive.
- For the first page, I used HTML because there wasn't a lot of writing. However, I had to use a JSP file in the last page in order to output the data. This way, I was able to use some java code and reduce the HTML.