**Lessons Learned**

Through Java Project 1, I learned many key concepts and ideas that I will keep in my handbook for designing software. Most importantly, I learned how to build a n-tier system and understood what constitutes each layer. I would use the following approach to build such a client-server application from scratch:

1. Starting bottom-up, I would first identify all the entity classes based on the requirements. These entity classes would be nouns that represent concrete objects that almost translate 1-1 to database tables (not always and not exactly). These classes would become the fundamental model package that all other classes operate on to maintain its state. This would be the bottom layer of the application layer and is best encapsulated from higher levels of the application layer.
2. Then introduce internal interfaces that connect the business logic/activity layer to the models. The business logic would be all the operations that are executed on the model. In our project ProxyAuto could be described as this layer. This layer is more like an adapter that wraps all the underlying models and IO operations. All incoming requests go through this layer and are outsourced to FileIO or Automobile class as appropriate. Since this layer is the central bridge between all functionality it also stores a data structure to keep track of all the instances of the models it operates on.
3. Then introduce external interfaces that act as an API that external clients can use to access the underlying business logic. In our project, this could said to be BuildAuto and all the interfaces such as CreateAuto, UpdateAuto that clearly publish the functional categories of services available to any client.
4. Figure out what operations need to be performed on the client and what operations can be performed on the server.
5. Establish connection between client and server. The biggest news to me when executing this part was that all the models and functionality could be replicated on the client and could be used. The automobile model was also made available on the client side. This made sense once done as it really helped facilitate manipulating and working with the automobile object on the client side.
6. Make available client functionality in plain vanilla console format initially and then incorporate servlets and JSP to make the client functionality available through the web.
7. Ensure client server interaction works smoothly so that objects are created, sent and stored properly in the datasource. Eventually, incorporate a DB to persist the data storage.
8. Ensure separation of concerns so that business logic, data access code and user interface code are separated.

There are many smaller steps in between involving creating interfaces between servlets and business logic layer on the webserver. But at a high level these 7 steps help a novice like me organize a n-tiered application from the bottom-up. In this document, I have described some more lessons I have learned by topic.

**How encapsulation helps?** - In Unit 1, we created the option class within optionset. We fully encapsulated the option class and optionset classes within Automobile by using containment. Containment facilitates encapsulation. Encapsulation allowed us to hide the complexity of the model from the rest of the packages as everything was accessed through the public methods exposed by Automobile and nothing from optionset and option was directly accessible. Encapsulation also creates additional layers of interaction by needing adapter or router methods that basically act as a shell and reroute to other methods (i.e: ProxyAuto methods and many Automobile methods wrapped access to OptionSet values). Based on encapsulation techniques used in class, I have learned the following:

*Takeaways:*

* Encapsulation can be achieved using inner classes, access restrictors (getter/setters for private variables) and composition. It helps hide complexity.
* A composition creates a strong has-a relationship which depends on the life of the encapsulating object. In the case of our project, the life of the optionset and option depend strongly on the life of the automobile. Neither can exist on their own therefore it makes sense to contain them.
* Encapsulation ensures restricted access of or limited visibility to contained data (optionset) by neighboring components such as FileIO and Exception class.
* Encapsulation may add additional layers of functionality that act as wrappers to hide the encapsulated objects or data. For example: ProxyAuto and Automobile have methods that basically call other methods they have direct access to get the job done, they themselves are not always performing the logic.
* In our project, we stored the choice variable in optionset class. The benefit of doing this was we could simply point to the option object and if the option name or price was updated it would simply reflect in the choice variable without having to update choice value explicitly. However, because option is encapsulated we had to store choice in optionset and expose it through public methods in automobile class.
* If code is encapsulated and hidden then client will not rely on its implementation details. Encapsulation makes the code more robust by not allowing others to change variables. Enforcing the usage of getters/setters makes the code more extensible as well as maintainable.

**How to design for Self-Containment?** - Objects should be designed to be self-contained and independent. Therefore, the Object should simulate a whole entity in terms of its properties and behavior and should not have to rely on other objects in order to exist.

*Takeaways:*

* Inherited objects are not entirely self-contained and therefore create strong dependency.
* Think of all the attributes and behavior that describe an entity to design the class.
* Add methods that operate on internal state of object.

**When is Inheritance appropriate?** - Although inheritance causes tight dependencies between classes, it is a powerful tool to abstract the details and to create layering in the application. We used this minimally in the application to minimize interdependencies. However, its usage between BuildAuto and ProxyAuto worked well in creating an interface between the interfaces and the detailed access methods for Automobile. BuildAuto hides the details of ProxyAuto and enforces a standard communication gateway by implementing the interfaces. As we look at the application we see multiple layers of encapsulation, where each hides the complexities of the layers underneath.

*Takeaways:*

* While inheritance should be used cautiously to minimize dependencies, it is a powerful tool in adding layers of abstraction.
* Just like BuildAuto, it can be used inversely to create an empty class that although implements interfaces has no implementation of its own. It inherits all the implementation.
* Abstract classes are often used as superclasses in inheritance cases so that common implementation can be abstracted away while behaviors/methods can be enforced upon child classes.

**How to design an Interface?** - There are internal and external APIs, building of which can be facilitated by java interfaces. The purpose of APIs is to hide the implementation details from the end user and provide a standard format that can be used consistently by several different clients. Due to the interface, if the implementation changes there will be no impact on the client service as the input, output and approach to access the service remains the same. As long as the inputs and outputs are clearly defined to the interface, the interface can act as a standard communication bridge and add layers of security in between. In this project, we created several interfaces for different functionalities that users need and then we added a consistent way of accessing the functionality defined by each interface by merging it all in BuildAuto class. The benefit of the BuildAuto class is that it created layers in between hiding all the implementation details from the driver class. While designing interfaces, the following should be kept in mind:

*Takeaways:*

* The types of access that the interface needs to publish should be thought through in terms of use cases that could be satisfied in a standard way. If I had done this prior to the project then I could have minimized random interaction between classes.
* Input and Output for the interfaces should be clearly defined.
* Minimal input that is known to the client that gets maximum job done should be requested of an interface.
* Input should be taken preferably in a pass by value or read only format. Manipulated input (passed by reference) gives little choice to client in terms of how they would have wanted the object modified.
* Avoid returning null values in APIs
* Internal interfaces between components helps standardize the way all other components communicate with each other. For example: If all DB CRUD operations are exposed through an interface then all other packages can access it in a standardized way instead of accessing the DB methods in a random fashion and creating spaghetti code. This way low-coupling between components can be achieved as well as maintainability and readability of code.
* Two things are important to keep in mind when publishing an interface:
  + Who are the clients of the interface?
  + What is the minimum functionality required?
* Interface should be standalone and not dependent on specific implementations.
* Separate functionality into smaller interfaces rather than one monster interface that is hard to maintain or reuse. In this project, we divided BuildAuto abilities into CreateAuto, UpdateAuto, FixAuto, etc.
* Polymorphism plays a big role in the usage of interfaces. It makes the usage of functionality extensible. A single interface type can be used to instantiate any class that implements that interface. The generality achieved by polymorphism helps in creating a bridge between a class and many different behaviors. This is typically used through dependency injection. In our case, all CRUD operations on the automobile were achieved through interfaces tied to a BuildAuto instance.

**How to organize and connect to a Database?** - While creating the DB in Unit 6, we used a bridge table to tie the various table keys together as foreign keys. The foreign keys help maintain referential integrity between two or more tables. This means that if a value in one table depends on a value in another table, through foreign keys they can be associated together and if one is changed the effects can be cascaded to the associated table. While we made a bridge table associating Automobile, OptionSet and Option, in hindsight this may not have been the best possible way to design the database. While writing database update statements, I realized that there were many use case scenarios that could not be handled by current design. For example: if an optionset name is updated, the resulting effect will be applied to all automobiles that share that optionset but in the linkedhashmap not automobile object will be updated. Furthermore, the optionset and option values are closely tied together and it might have been better to have the optionset ID as a foreign key in the option table. The current design ties both the optionset and option to the automobile and not to each other and could end up making nonsensical associations between optionset and option values such as a blue transmission. The best possible way to design a DB would be to list down all the CRUD operation scenarios beforehand and then see how much sense the DB design makes in each use case. Also, a balance needs to be struck between normalizing the data so it is not redundant and creating many tables. Too many tables require complex joins thus slowing down the DB operations. While redundant data is hard to keep consistent.

*Takeaways:*

* Strike a balance between data redundancy and complex joins
* Preempt DB design with thorough use case scenario analysis
* Use foreign keys to maintain referential integrity and cascade changes as appropriate
* Use bridge tables to associate multiple tables together. Typically use a unique row id of each table as the foreign key.
* Make sure the DB transactions comply with ACID properties
* Use logical transactions to make several associated database updates atomic. Turn off autocommit, commit once all associated updates are done and if there is an error rollback the changes.
* Load DB connection properties (URL, username, password) from a properties file so that if DB needs to be changed, the properties file can be easily replaced.
* Store SQL queries separately in a file so that it is easier to maintain and reuse them and all of them can be retrieved from one place based on their key name.
* Typically, DB connections should not be made for every DB transaction as the connecting and disconnecting process is expensive. A connection pool is provided by the server typically so connections can be reused.
* Have an interface that facilitates interaction with DB.

**When to use Serialization & Data Streams?** - A lot of the inter-communication happens through data or object streams. All the IO classes, use file streams, object streams, etc to communicate. Sockets also use object streams. To read and write data to files, sockets, etc, the information has to be made into a stream. Serialization entails creating a byte stream to store information in-memory. The benefit of serialization is that it helps with prototyping persistence (saving the state of the object) and short term storage of anything that is serializable.

*Takeaways:*

* It is best to avoid using buffers for temporary storage when reading from a stream.
* Close the streams explicitly after usage to avoid memory leaks.
* Minimize scope of streams and passing them around different classes.
* Serialization is brittle, as objects change serialized data would be hard to work with. However, it is convenient for short-term storage in files especially in java.

**Why and when to multi-thread?** - Having only one thing happening at a time in a large enterprise application is wasteful of resources and time. Many clients can be served at a time through multi-threading. However, to multi-thread proper synchronization should be done to access shared resources. Multi-threading is used in servers to spawn a new thread to handle each client request.

*Takeaways:*

* Each thread should be self-sufficient and have the run method that calls various synchronized methods.
* If a class implements runnable, it is best not to instantiate the thread in the constructor of the class using “this”. The constructor is creating the object “this” and so referring to the object while it is being constructed and passing it to a thread instance as a parameter can lead to unpredictable behavior.
* A race condition can be tested by slowing down the thread execution using random wait in the thread life cycle.
* In our project, we exposed functionality that supported both thread safe and thread unsafe version. We did this by having separate interfaces for accessing each and overriding the non-thread safe version of methods.
* The main thread should be kept alive till the end (can be achieved through join).
* Servers typically use multi-threading to support many clients together. The driver spawns new threads for each client.

**Why Synchronize usage of resources?** - Synchronization should be done to ensure that data does not get corrupted by multiple threads trying to access the same resources. Synchronization essentially implements wait and notify in the background and does not mean that a resource is thread safe. All methods that access that resource would need to be synchronized. It is best to synchronize at a lower level so that large resources are not blocked from usage by concurrent threads. For example: In this project it would be better to synchronize methods that access specific automobile objects versus synchronizing access to the entire linkedhashmap. Furthermore, not all data structures are thread safe. Linkedhashmap does not support synchronization.

*Takeaways:*

* Synchronize resources at a lower level as much as possible so that entire resources are not blocked from usage. At the same time, don’t need to go to a very low level either like option or optionset as it increases complexity and needs to be maintained.
* Be mindful of data structures and other resource objects that do not support synchronization.
* Synchronizing an object explicitly in curly brackets is usually better than synchronizing an entire method as it gives more control over when to lock and release the resource.
* Synchronization impacts performance as shared resources have to be waited upon but it is needed to avoid data corruption. Multithreading on the other hand improves performance by making the application scalable and allowing parallel tasks.

**How to limit scope & access of variables?** - The scope of variables should be minimized. There should be as few global variables as possible. Also data access restrictions should be placed to provide minimum visibility so that clients only know what they need to and the internal complexity can be hidden and made secure.

*Takeaways:*

* Variable scope should be minimized by using more local variables within the method and fewer global instance variables.
* Variables pointing to objects should be made null when done so GC can pick it up.
* Variables, Methods and Classes should be restricted thoughtfully to create encapsulation. In this project, we used private and protected methods inside option and optionset and public methods in Automobile to encapsulate the former and expose the latter to ProxyAuto.

**How to improve Code Readability?** - In order to make code more maintainable and readable the following conventions should be used:

*Takeaways:*

* Package names start with small letters, class names start with capitals and variables are usually camel case. Naming conventions for all three should be descriptive and consistent so that developer can easily understand what the code is doing.
* Add no-arg and other constructors. Don’t rely on default constructors.
* Add comments against each method and class and groups of lines that form a task.
* Declare and initialize variables at the top of the class/method.
* Avoid repeating the same code in various different methods and classes. Modularize such code into separate methods.

**Why Modularize?** - This I think is the most fundamental piece to remember in order to avoid redundant code. If this is not done right, it becomes difficult to maintain and reuse code. When building the CarModelOptionsIO class to parse automobile list or automobile object and display it to the front-end, I captured user input from the console in the same method as processing the input. For this reason, when it came to providing the same functionality through servlets, I was not able to reuse the methods well.

*Takeaways:*

* When creating a method, think about the smallest unit of independent functionality in the method that has the potential of being reused.
* Methods should not correspond one to one with the use case or activity being performed. It can be broken down way more. When broken down, think of input/output to these methods so that they can be reused as much as possible.
* Functions/ Classes should do one thing only. Separation of responsibility.
* In addition to method modularization, the classes also need to be divided so that they are self-contained and describe an entity on their own.
* All methods should not be calling all others. Think about using sort of “factory/parent methods” that delegate to other methods and coordinate all the various tasks in a use case.
* DB queries for CRUD operations should be reused by having separate methods for each type of operation and passing different variable values.

**When to use Static Variables?** - Use of static variables should be avoided as much as possible. Static variables are useful when dealing with shared resources across many classes.

*Takeaways:*

* Data Sources such as Automobile LinkedHashMap needs to be static because it is a single source of all automobiles for all users. This source needs to be consistent across the platform at all times.
* Other cases where static variables may be used are creating singleton class for serving DB connections from a pool. In my project I used a single static method to serve DB connection to all the classes that needed to connect to the DB. This needs to be static so that DB connection instances are only created and accessed at one place.

**Why read properties from files?** - In our project, we used property files to read car configurations, DB connection details and DB sql queries. This enabled readability and maintainability.

*Takeaways:*

* Configurations should be read from separate property file as to enable reuse, maintainability and change.
* The interface that takes in the file should be made flexible to accept different file formats through a single interface. Although the interface accepts any file format, in the background the implementation may have different parsers to parse the file differently based on file type.

**When to Overload and Override? -** Overriding should be used along with inheritance, abstract classes and interfaces to provide/define basic functionality in super classes/interfaces and overriding that for specific implementations. Overloading is recommended for constructors and other methods to keep consistent signatures with options of different input and output.

*Takeaways:*

* Use overriding with inheritance to provide the basic common functionality in superclass and overriding those methods in subclass. We used this with the EditThread class, which extended ProxyAuto’s basic methods and overrided them to make them synchronized. Then through different interfaces either the plain vanilla method or synchronized method could be accessed as appropriate.
* Use overloading to keep consistent signature with varying input/output. This is useful in APIs to give client the option to access the same functionality with different forms of input and output.

**How to make programs self-healing?** - Exception Handling doesn’t have to be limited to the classes provided by java API. Custom exception must be added as appropriate to handle all sorts of error scenarios. Furthermore, exceptions can be made self-healing by throwing the error and delegating the fixing to helper classes.

*Takeaways:*

* To organize custom exceptions, use error numbers to identify each kind of error and try to categorize them into buckets that may be addressed by some helper fixer class.
* Use an overarching exception class that sort of acts like a factory class to delegate the fixing to other helper classes based on the error code.
* Use enumerations to standardize error codes and messages.
* When error occurs in an API, show client appropriate error messages and don’t return null values or vague messages. Give client the opportunity to re-try so that the error can be self-healed.
* If issues are not known at time of coding then writing custom exceptions allows to handle such runtime issues without having to restart the program. For example: incorrect user input, etc.