Critical Reflection

The design we received to implement was by no means perfect – no software design ever is, and there is no such thing as bug free software. We had to make several changes to architecture and overall approach in our final product, some differing wildly from the design we received and some not.

The first and most obvious thing that was incorrect with the design we received was that no members in any of the classes on the class diagram had an access control level of private assigned to them. This makes everything public. This is bad engineering practice and a very big oversight – it means that anything that has a reference to an instance to a class also has full read and write access to all of its members, this is a code smell. Here is an example of where it occurs in the code base (some code is omitted for brevity).

public class User {

String userID;

String name;

String userType;

String password;

All these fields are public when they definitely should not be, for example it should not be possible to change the userID of a user after construction; its supposed to be unique. Changing a user’s name and type should possibly be editable, but setting a password is rarely as simple as updating that user object. You will probably want some validation to check that the password is strong enough, or that the new password isn't a previous password of that user. It would make sense to have something like a setPassword method on the user class to perform these kind of actions.

As mentioned above, this was an issue across the entire design and this example is just a single instance of it. In some situations, we altered access control levels, but mostly we left them as per the design. The time cost associated with locking up all the access control levels to what they should be was something we could not afford to pay. What actually took longer to resolve was the fact that no members had types either. This took a substantial amount of time to derive, using the rest of their design documentation at some times to ensure they had not defined it in the sequence diagrams, but we mostly had to guess The occasional types that did exist seemed to be incorrect, such as having a variable called ‘transportType’ with type ‘Transport’, even though there is a ‘TransportType’ type.

Another large problem we encountered was something that broke the S in the SOLID object orientated design principles; the S stands for Single Responsibility principle. The ManagementUI detailed in the design we received that was in the class diagram and had its behaviour detailed in one of the sequence diagrams is in charge of authenticating users **and** re rendering itself and displaying a list of fares. This gives it 2 responsibilities, breaking SOLID. You can always tell if a class breaks SOLID by trying to put what its responsibility is into a sentence. If that sentence contains the word ‘and’, it breaks S in SOLID.

This is easily remedied by refactoring the login behaviour of ManagementUI out into another UI – which is just a class that extends Jframe. We called this ManagementLoginUI. Now the responsibilities for those UI's are as follows:

\* ManagementUI displays a list of Fares

\* ManagementLoginUI handles authenticated of users.

Both of these technically also link to the “next” UI in the sequence, but that is unavoidable in this kind of system. For example, the ManagementLoginUI tells the ManagementUI to render when a user is successfully logged in.

Something else we had to add regarding user interfaces relates to that point. We needed to give each UI class a reference to the UI that rendered it – so if the user needs to go back (a very common action), we can set the current UI to not show and show the previous one. Again, seems messy but unavoidable. This can be easily done like so:

public class ManagementUI extends javax.swing.JFrame {

private final ManagementLoginUI loginUi;

private final Server server;

Now ManagementUI can re-render the login UI when a user logs out.

There are numerous smaller problems that are not really worth explaining in great depth here, so here is a list of some. The full document explaining problems we found with our design is at docs/Problems.md in the files we have submitted for this assignment.

* Account has an address, an occupation and a date of birth. These belong in the user class.
* Account didn't have access to TokenList. We added so we can validate a Token.
* Server didn't have an account list.
* Account class – check balance method – parameter should be float
* Account class – updateBalance – missing from UML.

**Conclusion**

We did as much changing of the design at the start of the implementation stage as possible, but it was unavoidable to change it throughout. After we had finished doing any initial alterations, we first split the class diagram up into sub systems (Interfaces, Pricing System, Authentication etc.) and each took one or two of those to implement. We then used an Agile approach to developing the software, having sprints a week long, with reviews at the end to iterate on what we had done (We assume the reader knows what Agile is).

Version control helped massively for us to keep on track time wise with this assignment. We used Git and GitHub with the GitFlow model (splitting everything up into feature or bug branches) and then pull requesting those changes to master and getting them quality assurance tested by other members of the group. We had a checklist of items that must be met before something was merged into the master branch. You can find this in our documentation.

This was the first time any members of our group had implemented a piece of software based on a design provided by another group of students. While this was a complete pain and we hated it the entire time, it did display the irritation of working without a fully fledged design to go from.