# Hospital Management System

# Reflective Report

## Design Process

On starting this project, I struggled to grasp the scale of the program. I generated one or two initial sketches of UML diagrams, but these barely covered basic functionality before I got confused. As such I started a NetBeans project following these rudimentary designs to give me an idea of how I could flesh out the program and expand it. Once I was content with my design (which at this time just covered my plans for the request factory, which remained unchanged but expanded in my final design) I started a new project and coded to this. Once this functionality was complete, I started a process of planning and diagramming the next portion of functionality, and then coding to meet that.

In the long run, this led to some unfortunate design choices; for example, there was a period of several days where I tried to implement the design with a doctor keeping track entirely of their own schedule and appointments, which made sense to me. However, this seemed to cause issues with serialisation and despite countless hours debugging I couldn’t solve this, so I switched to store all appointments in a singleton that was essentially manually serialised as opposed to relying on the cascade function. I designed using this, and then somewhere along the way the problems with serialisation just seemed to… Disappear, meaning I could have used my earlier (and in my opinion, better) design. However, at this point time pressure along with the amount of work I would have to redo meant that the appointment singleton persisted in the final design.

There were several areas of functionality that I was not confident in, for example handling dates in java and serialising multiple items to one file. For these I created small additional projects wherein I could safely test the function and understand it before implementing it in my design.

## Design Choices

Given the limitations that are bound to come with coursework, I am largely happy with my design. I feel like I have used several appropriate design patterns without using them unnecessarily; for example, using a factory pattern to create requests meant that as I developed the program and added new request types (for example I added prescription and stock requests very late in development) I could simply follow the pre-existing pipeline to create them.

Additionally, I found the adapter pattern to be invaluable in designing this program. Once an adapter has been created and tested and proven to work, it becomes unbelievably convenient to use it to retrieve a certain piece of information.

I have used the singleton pattern a lot in my program; all the controllers are singletons to keep a persistent authorising account, alongside the standard singletons for storing data. This is discussed later as I believe they are used enough the be detrimental to my design.

My rudimentary observer pattern was implemented with the sole purpose of cleanly deleting a patient account and removing their floating requests. This is a useful piece of functionality, and would be useful in a real-world scenario, but could have been expanded much further – once more, discussed later.

For processing the outcome of an appointment, my final design uses an abstract factory pattern. This was not my ideal situation, but it works and provides full functionality to the end user.

## To Which Degree Does My Design Meet Good Design Criteria?

I particularly struggled with processing the outcome of an appointment, as there was so much information to be processed and so many variants. I initially planned to use the decorator pattern (ie you have on AppointmentResult class with just the notes, as these are required, and then this can be decorated with an additional appointment or prescription request) but in implementation I could not make this work. Therefore, I eventually settled on an abstract factory pattern, which I believe works well, if less elegantly. It is still open for expansion, but it would be more awkward to implement an expansion as you would need to cover all possibilities.

Additionally, I believe I could have used the observer pattern to much greater effect – for example, in having each doctor account observed by their active appointments so that when a doctor account is deleted it signals each observer to cleanly delete and remove references to itself. However, given my issues with cascade serialisation mentioned earlier, I wanted to avoid this for fear of triggering the same problems again. This is largely true for most of my program, where I believe I have an overreliance on singletons where I should have been using the observer pattern instead. This would then have reduced the complexity of my Compilation class, with (eventually, ideally) as little as just the accounts, stock, and id generator needing to be stored here for serialisation.

Largely, I believe that my design follows the single responsibility principle. Classes command only the functionality required of them, and other classes tap into this – for example, an account is responsible for generating a summary of its information and other classes are enabled to draw upon this. In a lot of places, I have been using subclasses and superclasses interchangeably, with the exception being when I need to explicitly access information only from the subclass. Therefore, I believe that my program follows the Liskov Substitution Principle. The Interface-Segregation Principle I believe I have followed correctly. I have several interfaces, each with a very specific function (ie viewing availability, or for viewing the information relevant to a prescription).

## Shortcomings in my Design

During testing of my program, I discovered one major flaw: at times, requesting an account would overwrite all saved data and replace it with only the request. While this is a shocking flaw, and while I believe it is fixed, as I have no way of knowing what caused the issue, I have no way of properly testing if it is fixed.

Besides this, an eventual plan would include extending the Doctor class to be observable and have that observed by its active requests. Therefore, this could follow the same process as deleting a patient account, wherein all floating requests are deleted as well.

## Repository Address

https://bitbucket.org/Ctrlaltdelete44/coursework/src/master/