|  |
| --- |
| Edinburgh Napier University |
| SET10101 – Software Architecture |
| Michael Suttie - 40541559 |

|  |
| --- |
| Suttie, Michael  11-24-2022 |

Table of Contents

[1. Introduction 3](#_Toc120754220)

[1.1. Problem Description 3](#_Toc120754221)

[2. Considered Architectures 4](#_Toc120754222)

[2.1. Three-Layered Client-Server 4](#_Toc120754223)

[2.1.1. Three-Layered Client-Server Advantages and Disadvantages 5](#_Toc120754224)

[2.1.2. Summary of Three-Layered Client-Server 5](#_Toc120754225)

[2.2. Aspect-Oriented 6](#_Toc120754226)

[2.2.1. Aspect-Oriented Software Architecture - Advantages and Disadvantages 7](#_Toc120754227)

[2.2.2. Summary of Aspect-Oriented Software Architecture 7](#_Toc120754228)

[3. Chosen Architecture 8](#_Toc120754229)

[4. Design and Development 9](#_Toc120754230)

[4.1. Requirements 9](#_Toc120754231)

[4.2. MoSCoW 10](#_Toc120754232)

[4.3. Design 11](#_Toc120754233)

[4.3.1. Wireframes 11](#_Toc120754234)

[4.3.2. Class Diagrams 12](#_Toc120754235)

[4.3.3. Activity Diagrams 14](#_Toc120754236)

[4.3.4. Languages & Tools 15](#_Toc120754237)

[4.3.5. Database Design 16](#_Toc120754238)

[5. Evaluation of Design and Implementation 18](#_Toc120754239)

[5.1. MoSCoW Review 18](#_Toc120754240)

[5.2. Adherence to Chosen Architecture 19](#_Toc120754241)

[5.3. Future Development 20](#_Toc120754242)

[References 21](#_Toc120754243)

Table of Figures

[Figure 1‑1: Proposed KwikMedical Application 4](#_Toc120752920)

[Figure 2‑1 Three Tiered Client Server Architecture (Raj, Raman, & Subramanian, 2017) 5](#_Toc120752921)

[Figure 2‑2: Example of Aspect Oriented Architecture (Liu, 2022) 7](#_Toc120752922)

[Figure 4‑1: Initial MoSCoW table 11](#_Toc120752923)

[Figure 4‑2: Presentation Mockups 12](#_Toc120752924)

[Figure 4‑3: HQ Operator Class Diagram 13](#_Toc120752925)

[Figure 4‑4: Ambulance Class Diagram 14](#_Toc120752926)

[Figure 4‑5: Hospital Class Diagram 14](#_Toc120752927)

[Figure 4‑6: HQ Operator Activity Diagram 15](#_Toc120752928)

[Figure 4‑7: Hospital Update / Delete activity Diagram 15](#_Toc120752929)

[Figure 4‑8: Ambulance Rescue Activity Diagram 16](#_Toc120752930)

[Figure 4‑9: KwikMedical ERD 17](#_Toc120752931)

[Figure 4‑10: Implemented KwikMedical Relationships 17](#_Toc120752932)

[Figure 4‑11: Patient Table in MS SQL Server Management Studio 17](#_Toc120752933)

[Figure 4‑12: Partial incident report in MS SQL Server Management Studio 18](#_Toc120752934)

[Figure 4‑13: Full incident report in MS SQL Server Management Studio 18](#_Toc120752935)

[Figure 4‑14: Hospital Table in MS SQL Server Management Studio 18](#_Toc120752936)

[Figure 5‑1: Completed and Evaluated MoSCoW 19](#_Toc120752937)

# Introduction

## Problem Description

An Ambulance service based in the UK requires an updated system for their command and control centers which has been dubbed as “KwikMedical” – this new system should be able to connect to an existing database of medical records and the prototype should support at least 20 hospitals. The current system is accessed by SQL, this could be changed in the future update.

The system should be distributed, and the entry point is typically that of the HQ operator that will take in a call from a patient – this information will be entered into their system and sent to a hospital and if necessary, generate an ambulance request form and send the information to an ambulance. The ambulance crew will then receive the patient information and be required to update the hospital on the condition of the patient while on call.

It is hoped that eventually, the KwikMedical system will implement a GPS system that will enable the KwikMedical system to use advanced location based services.

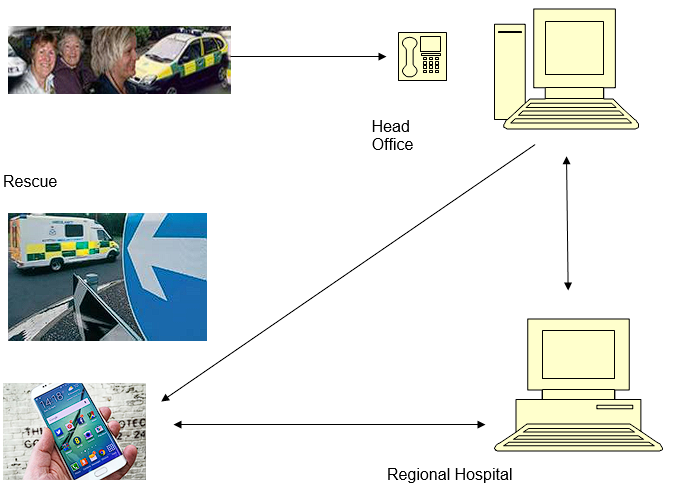


Figure ‑: Proposed KwikMedical Application

# Considered Architectures

## Three-Layered Client-Server

One of the most common multi-tier architectures is that of a three-tiered client-server. The basic premise is that the entire application is spread and organised into three tiers (GeeksForGeeks, 2022):

**The Presentation Tier** – This tier is the user interface and the highest tier in the architecture. It takes requests from the client and will display information to the client through the use of a GUI. The presentation tier can be accessed through any type of device such as a desktop, laptop, mobile phone and so on. It is common for the presentation tier to be web based, where the relevant pages will be fetched and presented to the user. It is crucial that this tier communicated with the other thiers that precede it in order to present the the client with the correct information.

**The Application Tier** – This is the middle tier, here, the business logic of the application resides and runs. The business logic in this case is the rules that are required in order for the application to run properly as set out by the organisation. This tier is typically run over one or more application servers.

**The Data Tier –** This is the lowest tier and is concerned with the storage and retrieval of the application data that is required by the application and presentation tiers. This usually comes in the dorm of a database server or any other any other device that supports and provices the steps to ensure that the data is exposed without providing any access to the storage and retrieval mechanisms. The data tier handles this by prociding an API to the application tier which ensures the transparency of the data operations. Thus, changes to this tier should not dramatically impact the applications server (Raj, Raman, & Subramanian, 2017).

A diagram depicting the architecture can be seen below:

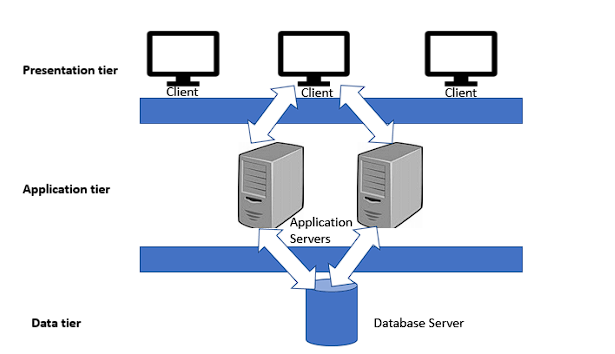


Figure ‑ Three Tiered Client Server Architecture (Raj, Raman, & Subramanian, 2017)

### Three-Layered Client-Server Advantages and Disadvantages

**Advantages of Three-Layered Client-Server**

• Logical separation is maintained between Presentation Tier, Application Tier, and Database Tier.

• Enhancement of Performance as the task is divided on multiple machines in distributed machines and moreover, each tier is independent of other tiers.

• Increasing demand for adding more servers can also be handled in the architecture as tiers can be scaled independently.

• Developers are independent to update the technology of one tier as it would not impact the other tiers.

• Reliability is improved with the independence of the tiers as issues of one tier would not affect the other ones.

• Programmers can easily maintain the database, presentation code, and business/application logic separately. If any change is required in business/application logic then it does not impact the presentation code and codebase.

• Security is improved as the client cannot communicate directly with Database Tier. Moreover, the data is validated at Application Tier before passing to Database Tier.

**Disadvantages of Three-Layered Client-Server**

• The Presentation Tier cannot communicate directly with Database Tier.

• Complexity also increases with the increase in tiers in architecture.

• There is an increase in the number of resources as codebase, presentation code, and application code need to be maintained separately.

### Summary of Three-Layered Client-Server

Upon comparing the benefits and drawbacks of the architecture, it’s clear that the benefits far outweigh the drawbacks in the hands of the right development team – and, when paired with the correct problem, can prove to be a very effective and efficient architecture. The splitting of tiers allows for quick development with little interference from other teams, thus increasing the reliability of the application. The security benefits are also notable given that the architecture restricts the client from accessing the database tier directly.

However, not having the ability for the presentation tier to communicate directly with the database adds additional complexity which only increases with the number of tiers. The upside of having multiple teams can also be a downside, as it requires more company resources to develop and maintain each tier separately.

## Aspect-Oriented

A large issue in modern programming is that of interconnected code. This issue is most prominent in the implementation of things such as logging, error handling and standards enforcement and feature variations which are notoriously difficult to implement in a modular way. This leads to tangled code across the system which then has an impact on quality, productivity and maintenance (Eclipse Foundation, N/A). An architecture that aims to solve these issues is Aspect-Oriented Software Architecture (AOSA).

Aspect-Oriented Software Architecture (AOSA) is a relatively new and up-and-coming architecture that is beginning to gain wide use. It attempts to address the problem where a piece of software has many interconnected parts which can often make modification to the codebase without breaking or impacting another part of the system difficult. ASOA addreses this by implementing a form of abstraction known as an aspect which is used alongside other abstractions such as objects and methods (Liu, 2022). An example of this can be seen below:



Figure ‑: Example of Aspect Oriented Architecture (Liu, 2022)

### Aspect-Oriented Software Architecture - Advantages and Disadvantages

**Advantages of AOSA**

• AOSA is gaining a lot of popularity and is therefore widely used – thus, many developers are familiar with the architecture.

• AOSA promotes reusability in that is affects code through abstraction and does not interfere with other parts of a codebase. Additionally, aspects can be called multiple times.

• AOSA is simple and maintainable as each aspect extends to the central core – it is easy to edit or change a single aspect without having an impact on other aspects. This usually makes it easy to identify issues and fix them.

• AOSA complements other styles due to its nature as an auxiliary architectural style.

**Disadvantages of AOSA**

• AOSA has no standard testing method

• The use of aspects makes the program resemble a web rather than a linear style and as such requires knowledge of how to read and interpret the code.

• Can be very difficult to read as a result of the web-like structure

### Summary of Aspect-Oriented Software Architecture

AOSA attempts to solve the issue of code tangling and improving reliability, usability and modifiability. As it is a new, but popular architecture, it will be well known by many developers. The act of increasing reusability and maintainability are key points to note too – as is the fact that it works very well alongside other architectures.

On the other hand, with it being a new architecture, there isn’t yet a standard method of testing which could make testing the entire system difficult. Additionally, those that haven’t used the architecture may struggle to learn it due to its non-linear and web like structure which can be hard to read and get used to.

Therefore, if the team developing the system is familiar with the architecture then it is a solid choice for the benefits it brings. If, however, the team have never used it, it may be better to reconsider and chose another architecture.

# Chosen Architecture

**The Three-Layered-Client-Server** architecture is best suited for the specification and the timeframe given to implement and design a solution. The drawbacks of the architecture can be managed with a better understanding of the architecture, care when implementing its core principles, and a developer / team that have worked with it previously. The benefits of the three-tier architecture discussed previously work extremely well with the requirements for the KwikMedical Application.

When developing the KwikMedical application using a team of developers, the speed of the development should be quick and very efficient – this is because the team can be split so that each tier can be worked on and developed simultaneously by those that specialise in their respective areas. This is because of the modularised nature of the architecture, which also comes with the benefit of each individual tier having little impact on the development of the other tiers through updates or development.

Future updates to the code, functions, or user interface within the application can be done without greatly affecting any other tiers which helps greatly with the maintenance of the application.

For example, say that a new “client” is being added. This will likely result in the addition of another user interface that allows the user to add, manage or update patients within the system. This new UI will have to connect to the Business Tier so the patient data can be stored as an object that can be inserted into the database. Afterwards, the Hospital and HQ should be able to access the information added by this new client with no problems – and if something where to go wrong, in theory, it shouldn’t have much if any impact on the existing system.

Due to the sensitive nature of the application handling patient and emergency services data, the fact that the architectures security comes with a boost as the client doesn’t have direct access to the database and it is therefore more difficult for a client to obtain any data that it is unauthorised to obtain – additionally, the business logic is more secure as it is stored on a central secure server. Additionally, this approach will allow multiple clients to access the data at the same time – for example, while the HQ is searching for a patient, the Hospital can do the same thing. The Ambulance crew will also be able to view their current patients data and update it as and when is needed.

# Design and Development

## Requirements

*The System must:*

* Receive Patient Details from the system operator:
* Patient Name
* NHS Registration Number
* Address
* Location
* Medical Condition
* Check the information entered by the operator against existing records held within the database (SQL Database).
* Work out the best way of helping the patient and generate an ambulance rescue request to one of 20 regional hospitals.
* Dropdown for Locations for Operator to choose from
* System will use the location, calculate mileage to nearest hospital and choose which hospital to dispatch ambulance from
* Send the patient information, as well as the ambulance request, to one of the 20 hospitals.
* Hospital will be determined by the last section.
* Extract the patient’s information and send the medical records to a device located in the ambulance (Likely a smartphone)
* Simulate a phone to receive requests and to perform the tasks mentioned next
* Regional Hospital will receive updates from the ambulance team which will update the patient record with call-out details such as:
* Who is involved
* What is the problem
* When / Where
* Action taken by team
* Length of call-out

*The System should later:*

* Interface with the GPS fitted to the rescue vehicles to facilitate location-based functions.

## MoSCoW

The MoSCoW system is used to decide the significance of each specification using the following criteria, ranked most important to least important: **Must Have, Should Have, Could Have, Won’t Have.**

|  |  |  |
| --- | --- | --- |
| Number | Classification | Requirement |
| 1 | Must Have | Receive input from system Operator containing: Patient Name, NHS Number, etc. |
| 2 | Must Have | Verify the existence of existing patients in the database. |
| 3 | Must Have | Assign patients to hospital and send records to the hospital. |
| 4 | Must Have | Assign an ambulance for the patient if needed. |
| 5 | Must Have | Send patient details to the ambulance. |
| 6 | Must Have | Ambulance can enter call-out details and send back to hospital |
|  |  |  |
| 7 | Should Have | Update existing patient records |
| 8 | Should Have | Work out the hospital closest to the incident |
| 9 | Should Have | Have a working GUI |
| 10 | Should Have | Full input validation |
| 11 | Should Have | Store multiple incidents per person |
| 12 | Should Have | Link between patient details and incidents |
| 13 | Should Have | Function on an actual mobile phone |
|  |  |  |
| 14 | Could Have | Offer temporary treatment as opposed to sending out an ambulance |
| 15 | Could Have | Delete patients |
|  |  |  |
| 16 | Won’t Have | Use of GPS to optimise vehicle assignment |
| 17 | Won’t Have | Login system |

Figure ‑: Initial MoSCoW table

This MoSCoW table will be referred to in the evaluation section to determine whether or not the project was a success or not.

## Design

### Wireframes

Mockups of the three presentation layers for the KwikMedical System are shown below – two for desktop and one designed for mobile first:

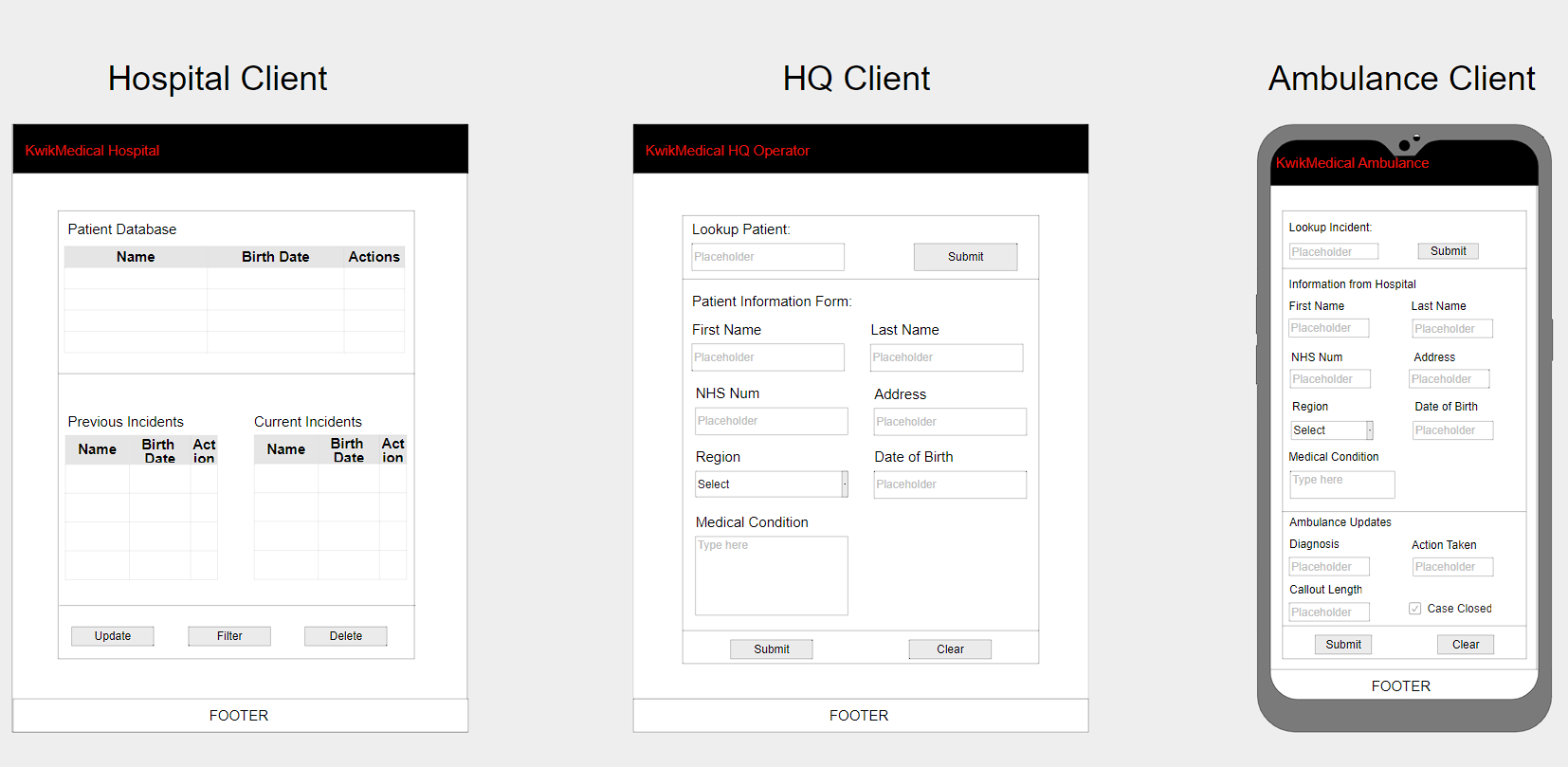


Figure ‑: Presentation Mockups

### Class Diagrams

The following section shows the class diagrams generated and assembled from the multiple clients in Visual Studio. Each \_Default class is the backend code of the presentation layer which communicates with the appropriate Logic and Data Access layers when needed – The object refers to the Database Model created using ADO.Net Entity Model Framework.

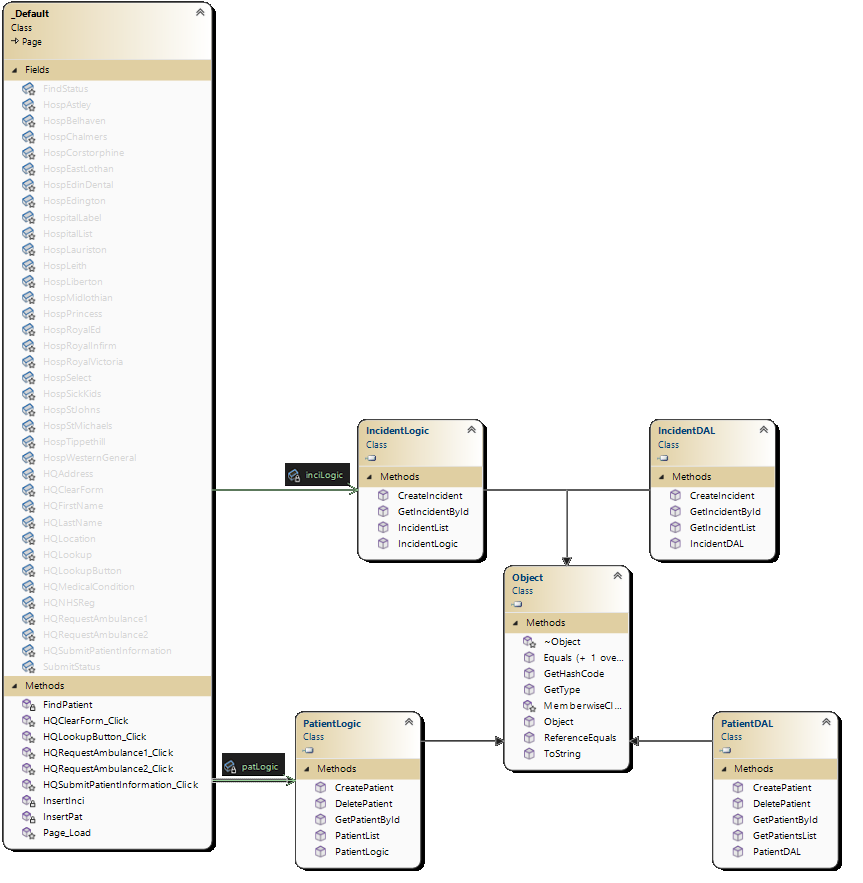


Figure ‑: HQ Operator Class Diagram

The HQ client shown above in figure 4-2 only accesses the data in both the Patient and Incident tabes, hence there only being two associations in comparison to the three associations seen in the Ambulance and Hospital diagrams shown on the following page which access the Patient, Incident and Hospital tables.

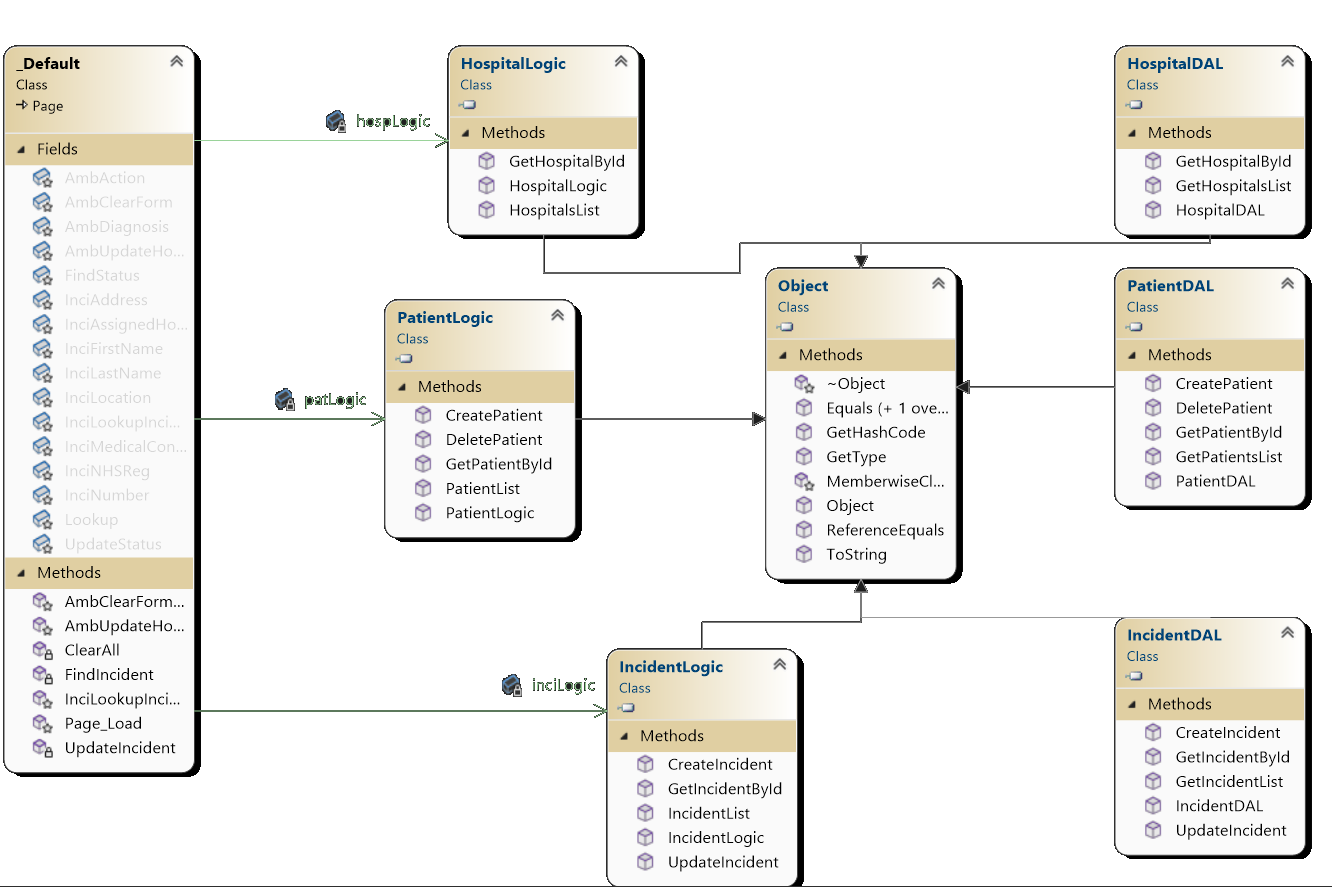


Figure ‑: Ambulance Class Diagram

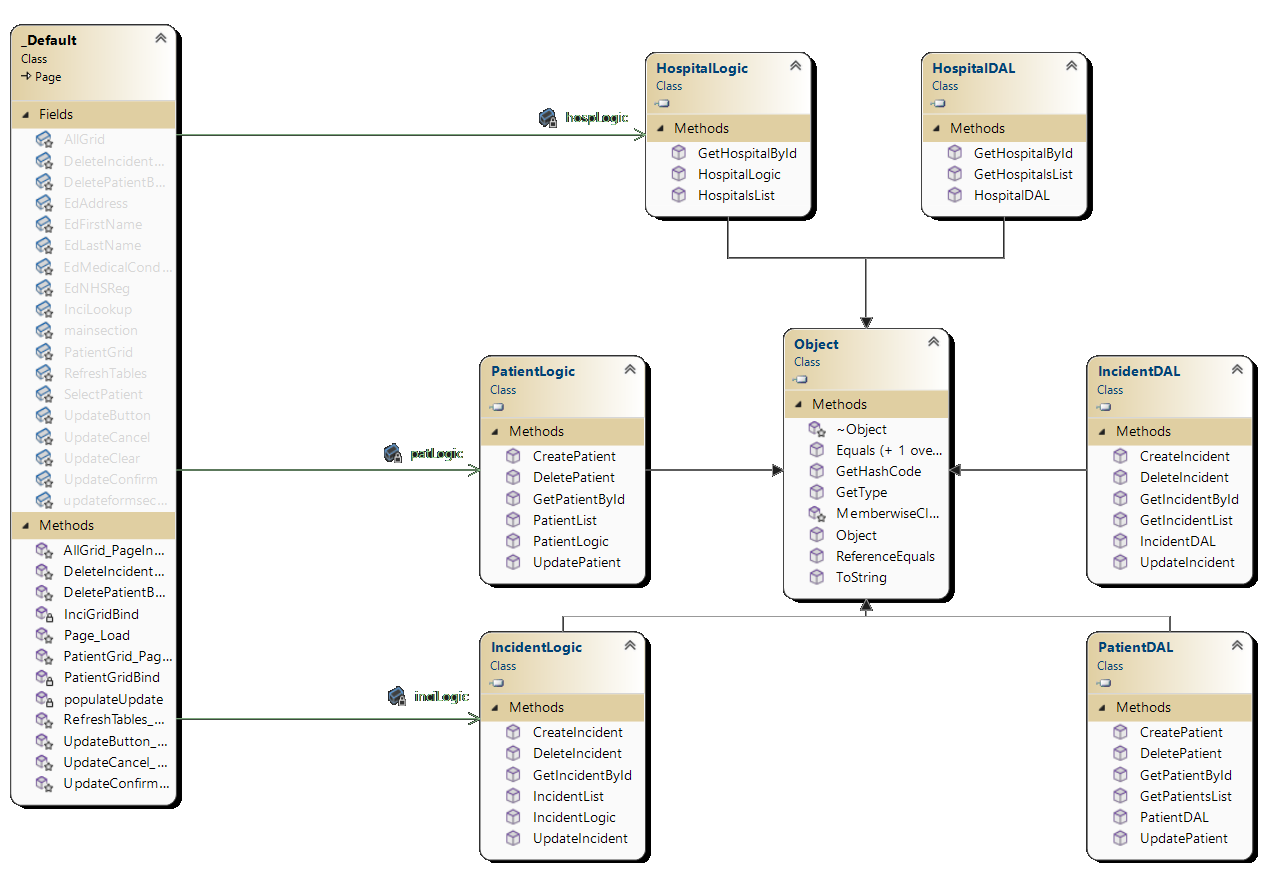


Figure ‑: Hospital Class Diagram

### Activity Diagrams

Displayed here are activity diagrams for each of the three KwikMedical clients; HQ, Ambulance and Hospital.

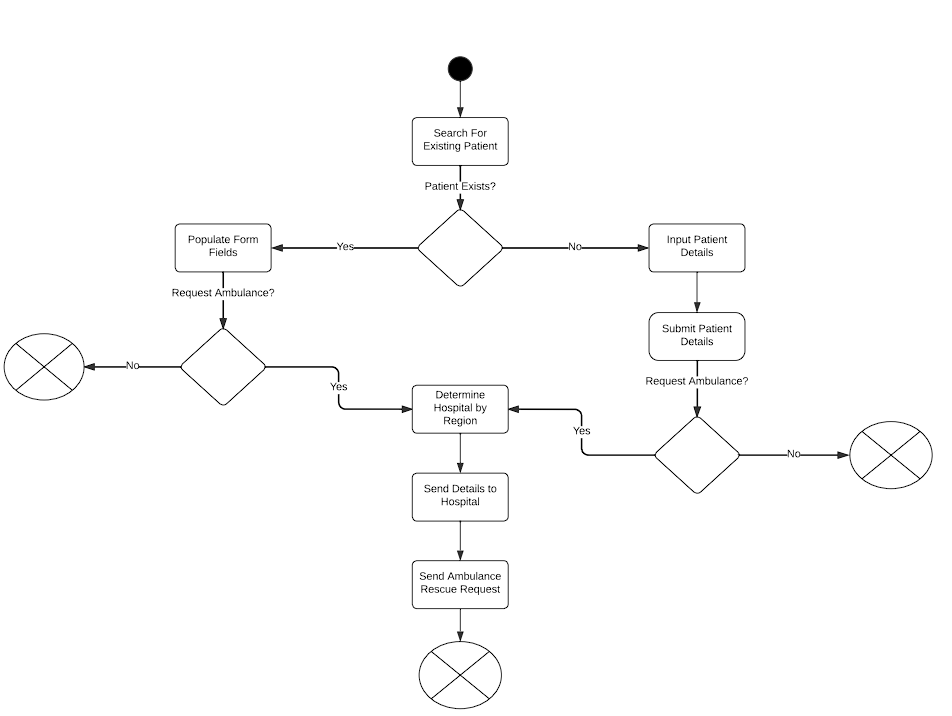


Figure ‑: HQ Operator Activity Diagram

The figure below contains two diagrams for the Hospital Client – One on the left for updating an existing patients records and one on the right for deleting a patient / incident.

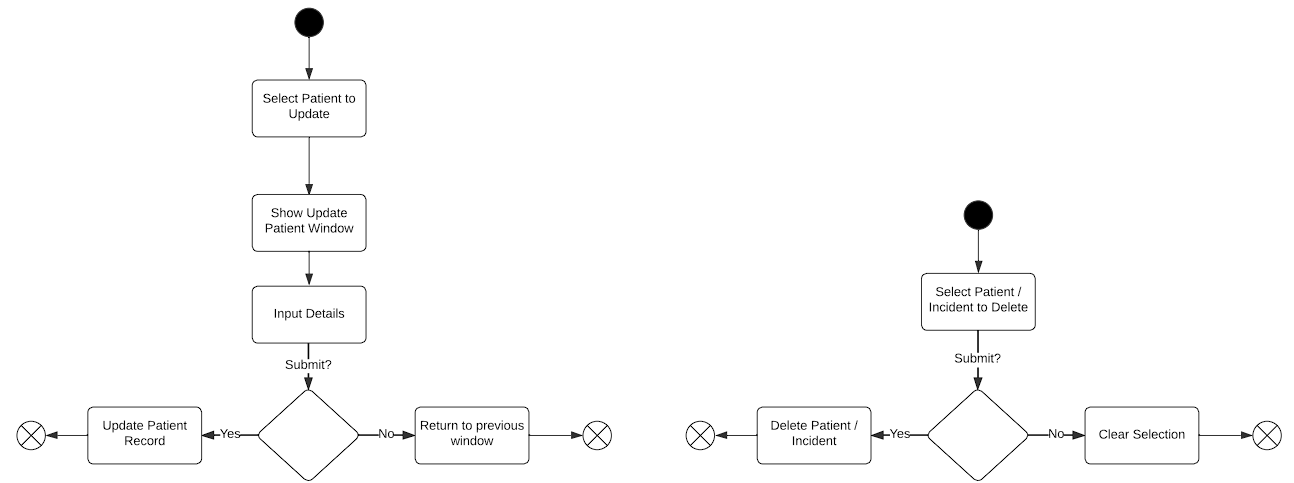


Figure ‑: Hospital Update / Delete activity Diagram

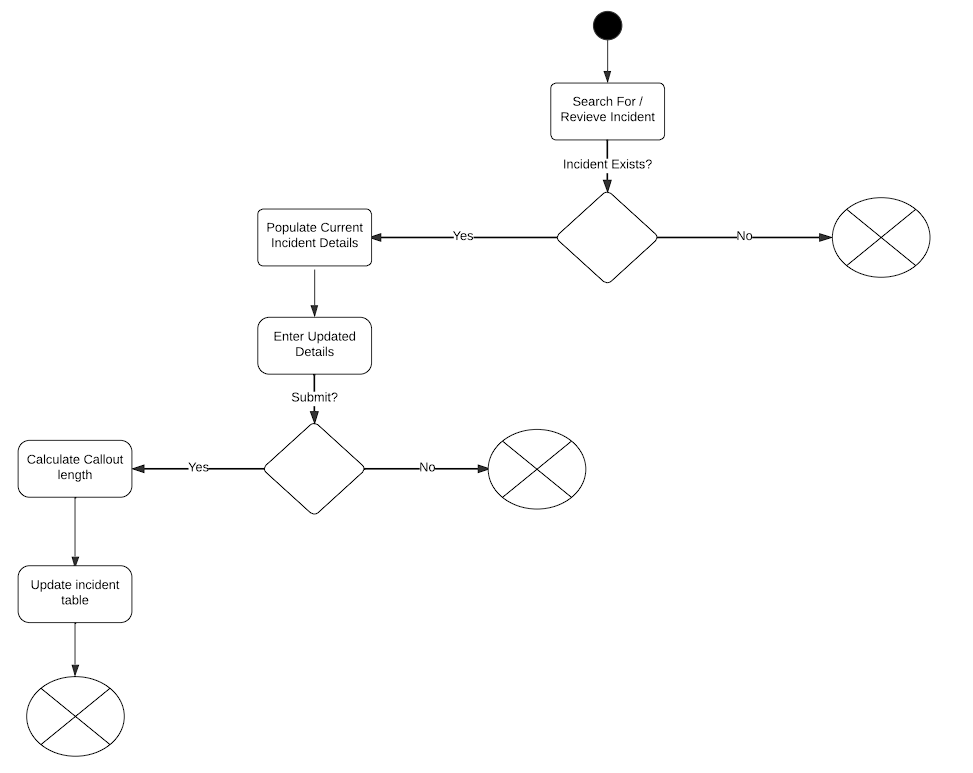


Figure ‑: Ambulance Rescue Activity Diagram

### Languages & Tools

The project will be created using ASP.NET in Visual Studio 2022. As such, the languages used will depend on the tier being developed:

*Note, not all of these languages and techniques will be used – but most of them will*

* Presentation Tier - HTML, JS, CSS and PhP, C#
* Business Logic – C#
* Data Access – SQL (Microsoft SQL, C#)

Additionally, the mySQL database will be hosted on a docker image that will be communicated with and set up using MySQL Workbench.

### Database Design

Seen in the figure below is the ERD created before the database was created – here, most of the information revolved around the incidents as opposed to patients themselves. A patient can have many incidents, but an incident can only have a single patient (at the present). Similarly, there can be many incidents to a hospital, but only one hospital assigned to a specific incident.

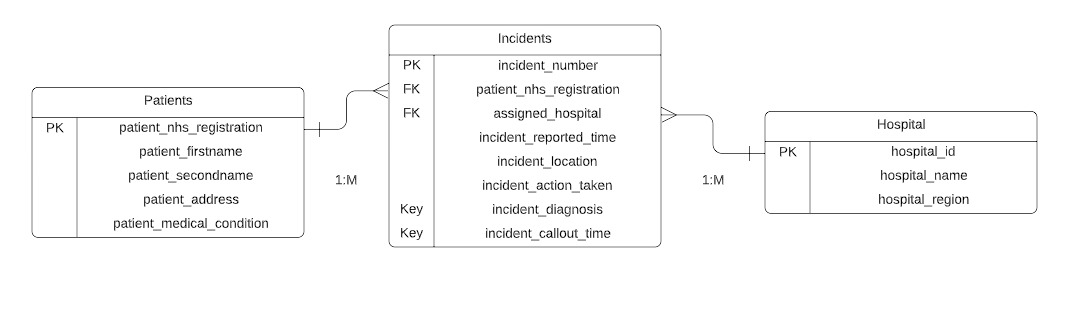


Figure ‑: KwikMedical ERD

This database design worked as intended and translated well into the design you see below in the next figure:

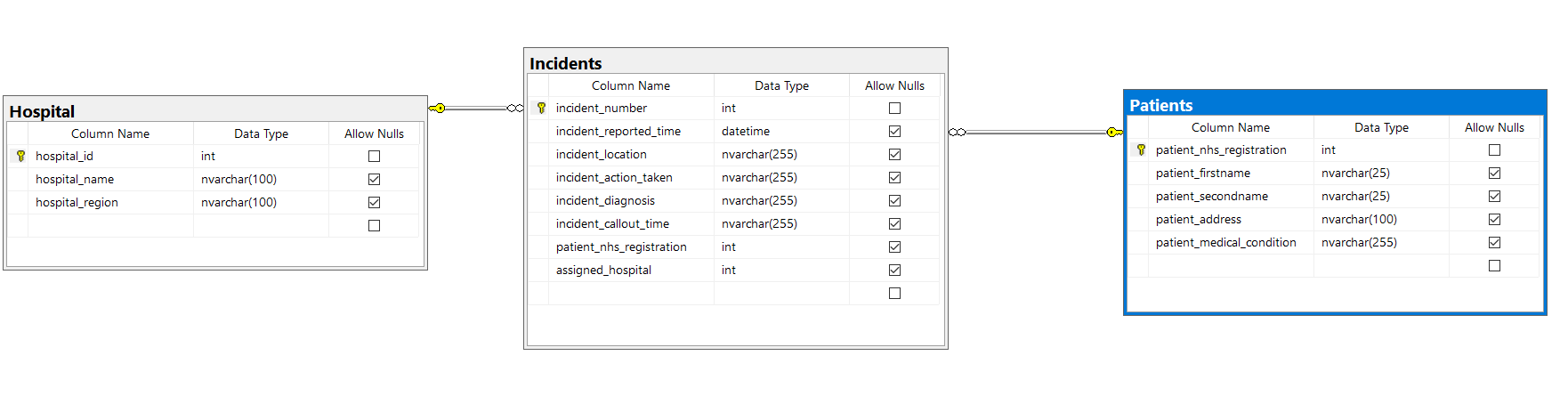


Figure ‑: Implemented KwikMedical Relationships

When a patient is entered by the operator, their information will be input in full into the database as seen below:

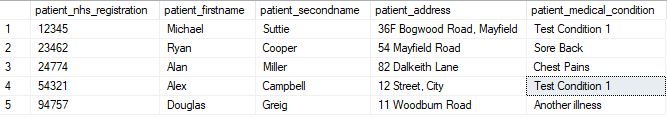


Figure ‑: Patient Table in MS SQL Server Management Studio

However, if an operator decides that an ambulance is required, the database will be populated partially as seen below:

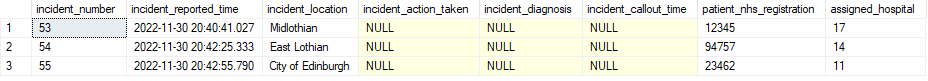


Figure ‑: Partial incident report in MS SQL Server Management Studio

The remaining information is then updated by the ambulance crew when the ambulance arrives to treat the patient, the callout time is calculated by comparing the time the incident was reported at to the time of the update from the ambulance:

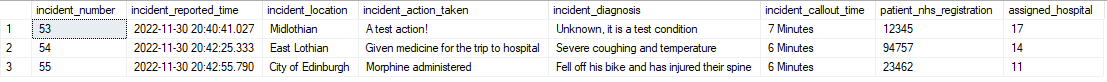


Figure ‑: Full incident report in MS SQL Server Management Studio

The final table in the database implementation is the Hospital table – this is static, and does not change from this state:

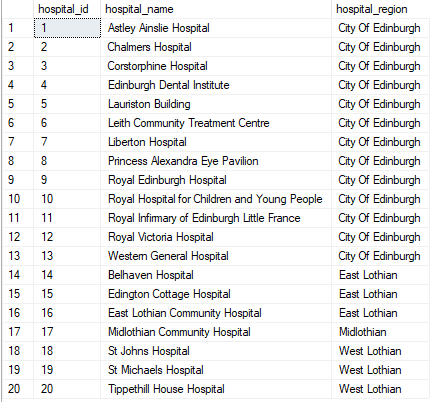


Figure ‑: Hospital Table in MS SQL Server Management Studio

# Evaluation of Design and Implementation

## MoSCoW Review

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Classification | Requirement | Achieved? |
| 1 | Must Have | Receive input from system Operator containing: Patient Name, NHS Number, etc. | **YES** |
| 2 | Must Have | Verify the existence of existing patients in the database. | **YES** |
| 3 | Must Have | Assign patients to a hospital and send records to the hospital. | **YES** |
| 4 | Must Have | Assign an ambulance for the patient if needed. | **YES** |
| 5 | Must Have | Send patient details to the ambulance. | **YES** |
| 6 | Must Have | Ambulance can enter call-out details and send back to hospital | **YES** |
|  |  |  |  |
| 7 | Should Have | Update existing patient records | **YES** |
| 8 | Should Have | Work out the hospital closest to the incident | **MOSTLY** |
| 9 | Should Have | Have a working GUI | **YES** |
| 10 | Should Have | Input validation | **MOSTLY** |
| 11 | Should Have | Store multiple incidents per person | **YES** |
| 12 | Should Have | Link between patient details and incidents | **YES** |
| 13 | Should Have | Function on an actual mobile phone | **YES** |
|  |  |  |  |
| 14 | Could Have | Offer temporary treatment as opposed to sending out an ambulance | **NO** |
| 15 | Could Have | Delete patients | **YES** |
|  |  |  |  |
| 16 | Won’t Have | Use of GPS to optimise vehicle assignment | **NO** |
| 17 | Won’t Have | Login system | **NO** |

Figure ‑: Completed and Evaluated MoSCoW

Evaluating the project using the MoSCoW table, it is clear that all of the “**must have**” requirements were met – as such, the system meets its core functional requirements. Additionally, the system meets 5 out of 7 of its “**should have**” requirements – however, it could be argued that it has met them all. The input validation could use work and the calculation of the closest hospital could be improved, but they are implemented. 1 out of 2 of the “**could have**” requirements have also been met, one of which being the deletion of patients (Incidents can also be deleted as a bonus) – there wasn’t enough time to implement the temporary treatment system, however that can be done at a later date. Finally, the “**won’t have**” requirements have remained as such – there is no GPS API implementation in this version, however, that would prove to be very useful as a future addition. Additionally, the system requires a login system to optimise wide-spread useage if it is to be distributed across the NHS in future – however, for the purpose of a prototype, this will suffice to showcase all of the systems functionality.

## Adherence to Chosen Architecture

As far as sticking to the Three-Layered Client-Server Architecture goes, the prototype adheres to it well. The three main layers are all present with each client having a web form presentation layer, a business logic layer and a data access layer. An unplanned, but welcome addition to the architecture was the addition of the ADO.Net Entity Model Framework which models the tables within the database and makes them queriable with minimal raw SQL writing. These models can be accessed and queried through the Data Access Layer via the Business Logic Layer which is called upon from the backend code of the presentation layer.

As a result of this, future updates should remain easy to implement, even to the database, as updating the model is as simple as either readding a new model or updating the old one – this will not affect the rest of the system in the case of additions to the database, however, it will likely have an impact if a datatype is changed or if an existing field is removed. Even in this case, however, the changes to fix this are minimal and can be traced through the layers. Not only will the updates and fixes be easy, they should also be quick to implement and roll out as teams can focus on their resepective layers.

Additionally, the layering of the application delivers upon the promise of improving security as the HQ operator has no access to sensitive information aside from that which they are authorised to access. Similarly, the Ambulance can only access the sensitive information that is provided to them. The only client with largely unrestricted access is the hospital.

## Future Development

#### Login System

The implementation of a login system for the hospital would allow them to view incidents and patients that have been assigned to them only. This would make more sense than the single hospital client implemented at present that can view every incident and every patient.

#### Ambulance Assignment

In line with the login system, this would allow each hospital to have an ambulance pool that can be drained and filled as ambulance requests are generated and fulfilled.

#### Active and Non Active Cases

Currently the system only allows the Hospital to all cases. A system should be implemented to allow the hospital to view cases that are ongoing as well as past cases.

#### Full Input Validation

While the system has partial input validation, enough to stop it from breaking frequently, it could certainly be improved to allow the system to be more flexible and fool-proof.

#### Add Temporary Treatment

As suggested in the MoSCoW table, a temporary treatment system could be implemented. This could take the form of another table that contains keywords such as “Sore head” which would match with an at home remedy such as “Paracetamol” and suggest this instead of calling an ambulance.

#### Use of GPS Location Services

GPS could be used to assign the hospital to an incident properly.

Currently, the system works based on the “Current Location” dropdown in the HQ Operator input. When an ambulance is needed, this is checked against the regions in the hospital table and fetches all of the hospitals within that region and allows the operator to choose one before submitting the ambulance request.

# References

Eclipse Foundation. (N/A). Retrieved from https://www.eclipse.org/ajdt/

GeeksForGeeks. (2022, 02 24). *Three-Tier Client Server Architecture in Distributed System*. Retrieved from GeeksForGeeks: https://www.geeksforgeeks.org/three-tier-client-server-architecture-in-distributed-system/

Liu, X. (2022). *Unit 5: Heterogeneous Style (3-Tiered) and Aspect-Oriented Architecture.*

Raj, P., Raman, A., & Subramanian, H. (2017). *Architectural Patterns.* Packt Publishing.