# Design

Within this chapter I will be discussion the design choice I have made whilst implementing this project and the reasons for making them choices. This chapter will also include details on the user interface design.

## Overall Architecture

I intended to design the system in a modular format thus the system contains four main packages with each package containing a tests package. Keeping code modulated enhances the reusability of the code and also makes the code easier to maintain. In this section I will be listing each package and explaining their purpose within the application.

## Activities package

Activities are Androids views; an activity provides a method of displaying information on screen for the user. The activities package contains the implemented classes for the activities.

## Models package

The models package contains the model classes for each of the database tables. A model represents an item created from the database, for example a drug or a piece of information about a drug. Keeping all models within one package allows the models to be reused in other applications.

## Data download package

The data download package contains the Robospice requests classes, which are responsible for downloading the individual API URLs and then parsing the data obtained. Each class within this package downloads a specific API XML file. Having all the data download classes in one package makes it easy to add new classes to download additional data in the future.

## Database package

The database package contains the class that is responsible for the interacting with the SQLite database. This class is responsible for fetching, creating and deleting data from the database. Having this class within its own package will allow the database and database code to be reused within other applications. If the code within the database helper class becomes to large using a package will also allow the code to be broken into smaller classes.

## Tests packages

Each package contains a package called tests. Within each test package is classes testing each class of the original package through both JUnit tests and Android instrument tests

## XML edibility

As the application will be given to a team of developers who may not have extensive experience in Android development it was essential that I make the basics of the application editable by an inexperienced developer, to achieve this all string values have been set in one XML file (strings.xml). Having done this, any text within the application can be changed by editing this XML file. As this application is produced for NHS Wales they may in the future want to translate the application to Welsh, to adding Welsh language support would be relatively easy due to this design choice.

To further improve the customisability of the application all API URL’s and the structure of the XML files is set within a single XML file (data\_download.xml). This makes the task of changing API URL’s and the format of the data simple.

## Data storage

One of main decisions when designing the application was deciding how the data for the application would be stored, within this section I will discuss the possible methods of storing the data and why I chose the method I chose.

## Store the data as XML files

Downloading the XML from the XML API URLs and storing the XML file on the device was one of the methods I considered. This method would allow for a quicker download task, as the task would not have to parse the XML. This method would also ensure the data was exactly as it was downloaded.

This is not a great method of storing the data on the device, as the device would have to parse all the XML files every time the data was needed be used. This method would also be slow at searching and ordering the data, as this method provided no indexing functionality.

## Store the data as a serialised object

For this method the application would parse the XML file into Java objects and then save the object onto the device through serialisation. This method would be quicker at runtime than storing the as XML files as the data will already be parsed when opening the data.

There are several issues with this method, the first issue being that this method does not allow indexing of the results and therefore searching the data will be slow. Another issue with this method is the entire dataset will need to be loaded into the devices memory when using the data, which could cause some devices to run out of memory.

## Store the data within an SQLite database

The Android SDK comes with a set of libraries for creating and interacting with SQLite databases. SQLite database can be indexed and are therefore quicker than the above-mentioned methods for searching the data.

I decided to use an SQLite database to store the data for the application. The data is parsed from XML into usable objects and then stored in the SQLite database. Using an SQLite database allowed me to only load needed data into memory by using the WHERE operator when searching for data. Using an SQLite database also helped keep the data in organised structure, which would allow the data to be used by other applications if needed.

The main issue with using SQLite database is that the stored data may be larger than if the data was stored as a serialised object, but if the size of the database became an issue the database could be compressed.

## Downloading of data

The next part of the design process was to decide how the data for the database would be downloaded. The download process had to be able to run in the background on the device as the download can take several minutes, therefore a user is likely to minimise the application. In this section I will discuss the methods of downloading the data that I considered.

## Download the data using an AsyncTask

It would have been possible to download the data using an AsyncTask as they AsyncTasks create a new thread meaning they can be used for HTTP requests. Within the Android documentation it states that AsyncTasks should not be used for long running tasks (more than a few seconds), due to this reason I decided not to use AsyncTasks for downloading the data. Another reason against using AsyncTasks is that they are attached to the activity that created them, so if the user minimises the application the task will also be killed, thus not allowing background downloading.

## Download the data using a service

The Android documentation suggests using a service for any long running task. A service runs within its own thread therefore allowing the download to continue working whilst in the background. A service will also place an item in the devices notification panel letting the user know that the service is still running. Using a service to download the data would have worked and been a good solution if implemented correctly.

A service is only a basic class for running a process in the background, it provides no implementation for alerting the view of its progress, nor does it contain any error handling should the tasks fail. The reason I chose not to download the data using a basic service is that I would need to implement a large amount additional code to handle all areas of failure and to notify the view of the downloads progress.

## Download the data using a Robospice service

Robospice is an open source library released under the Apache licence that simplifies the implementation of long running tasks. Robospice runs on top of the Android service providing implementation for frequently used features needed when implementing services. Robospice makes notifying the view of progress simple and also allows for multi-threading meaning that simultaneous downloads can occur. Another useful feature that Robospice contains is its error handling features; Robospice retries a task a set amount of times before finally alerting the main thread of its failure.

I decided that using a Robospice service to handle the downloading of data would be the best solution. Using a Robospice service meant I would be able to focus on the body of the task instead of worrying about implementing the various features needed for the service. Robospice does allow for caching, but as the data will only be downloaded once every few months I decided using an un-cached service would be acceptable.

## Design pattern used

When planning the application it was planned to create the application using a Model-View-Controller (MVC) design pattern. Once I began developing within Android it became apparent that a true MVC approach was not possible within Android. Although a true MVC approach was not possible I attempt to follow the MVC pattern throughout my design, thus creating a package for models, a package for activities (views) and separate packages for everything else (controllers). I believe that by modulating the code, the maintainability and reusability of the code has been increased.

## Download data singleton

It had been decided that the downloading of data would be executed within a Robospice service. Robospice services like Android services run within there own thread and are therefore are not attached to a specific activity. Although Robospice services allow for the passing of progress from the service to the activity, they do not natively support the passing of data.

To pass data between the download services and the download activity I designed a singleton class. A singleton class is a class that can only have one object of itself instantiated; a method is created within the class for the purpose of retrieving the single instantiated object. Using a singleton for the download progress meant that both the service and the activity would be using the same object, thus allowing data to be passed between them.

## Calculation class

The calculation class is the class responsible for handling the dosage and infusion rate calculations. I intended to create a powerful class that could handle both the validation and calculation of the calculations.

A method called validate was designed, this method checks that all data used for the calculation is present and correct. The method then returns an integer for the result of the validation; this integer can either represent an error (which the user must change), a warning (which the user has the option to change or continue) or success.

Once the validate method has been called, if the result is a success or a warning and the user opted to continue the calculation will be performed.

Keeping the validation and calculation within the same class means that the calculation code be reused within other projects. Using this approach also allowed for more rigorous testing as the validation and calculations could be tested together whilst Unit testing.