SEM5640 Group Project
Report

Go!Aber

Submitted in partial fulfillment of
the requirements for the award of the degree of

MEng
in
Software Engineering

Submitted by

Student No.	Connor Goddard
Student No.	Helen Harman
Student No.	Craig Heptinstall
Student No. 110036072	Samuel Jackson
Student No.	Daniel McGuckin

Department of Computer Science
Aberystwyth University

December 2015

Abstract

Contents

List of Figures

Chapter 1

Project Overview

Following from the brief provided at the start of this project[7], we are able to clarify a detailed overview of the application requested by the client. A new system to be named 'GoAber' would allow members or 'participants' from a university to interact with other participants enabling them to record activity data such as step counts, distances travelled and heart rates.

The System means to allow a number of methods for entering activity data from sources such as Fitbit[4] and Jawbone[6] devices, alongside manual and external entry via an application API. The external API would then be able to cope with users who wish to use other devices such as smart watches or health tracking mobile apps.

Users should be able to access the system either via a vanilla log in or through their university emails, though the different sign-on systems of universities may have to be considered. With the data entered by users, they would then be able to interact with other participants on a deeper level, through challenges. With challenges, the overall purpose of the application is to enable staff and students of a wide set of universities to keep active, and give them incentive while comparing their efforts to others. The application will give each participant the opportunity to work as a team when performing challenges, by allowing them to become a part of a 'group'. For instance, a certain department or group of individuals at a university may want to be represented, and therefore compete with other groups.

Following on from groups and interacting with users, it is important that different levels of authorisation is followed for anyone logged into the system. There is a total of three level of authorisation required of the system:

- Participants- as mentioned previously, will be part of a group and university(community).
- Coordinator- A user with privileges to create challenges, and add users to a group that they created.
- Administrator- A higher level user, with all the abilities of the previous,

alongside the abilities to edit and remove participants activity data, groups, and communities.

The final high level features that the system should be able to perform is to use the activity data entered by participants to display some form of league table in order to rank both groups, institutions and individuals. Alongside this, a scheduled email system will be in place for users to receive updates on their performance in challenges, and produce emails when a user is inactive for a certain length of time.

The client has asked the system to be implemented in both JavaEE and .NET, allowing a more flexible choice of installation for universities that may want either a Linux or windows machine to run their server. Each running instance of the application should be self dependent, and only send or request data from the others when interaction such as challenges occurs. A thorough set of testing should be performed before the release of the application to ensure that the user interface functions correctly and that the bilingualism the user has asked for works, while the logic in the back of the application should be tested through unit tests and stress testing to ensure the application can withstand a number of concurrent users.

Finally, this report outlines the process of work completed during the project, detailing each stage of the project. Any modifications or clarifications to to the original client brief will be described in the following section.

1.1 Detailed requirements

The requirements spoken of here are based off the initial requirements specification document [7] and some of the key requirements are mentioned alongside their requirement codes. Before the design of the project began, a few clarifications was made for some of the requirements in a QA session with the client, all of which are described in the group project meeting minutes. These will be available in the 'docs/minutes' folder in the hand-in. To best detail the requirements of the application, parts of the application can be grouped by functionality.

1.1.1 Management of users, authentication and authorisation

The first of the requirements that should be available to the user is the registration and signing into the application. As mentioned in the requirements specification under D-FR1, the system should allow users to be added to the system and to a group within a community. As discussed with the client in the first meeting, signing up for the site should be both available through the university credentials (SSO), or a vanilla log in, with a safe means of storage.

Alongside the registration of users, there should also be the functionality to edit users details and remove them. All levels of users should be able to perform these actions, though it was clarified in the QA session that only admin-authorised can edit and remove others users. Other administrator level activities only applicable to those users include renaming groups, removing groups, renaming their community, and deleting any other users' activity data. The auditing of other users' data is described in the auditing section of this chapter.

To clarify how users will connect through groups and communities, the example below highlights an example scenario:
A groups such as 'Computer science' would be contained within a community such as 'Aberystwyth University', and a participant would be added to Computer science by a coordinator that created that group. Then if for instance another group challenged Computer science the user's activity data would combine with others in their own group to compare their totals with the second groups. This is a similar story for communities, where totals of all participants of a given community could be compared with another community.

1.1.2 Activity data

The next subset of requirements required of the system is the integral need for activity data to be saved. As mentioned in the overview, the user will should be able to link Fitbit and Jawbone devices (of which data will be grabbed on a daily basis, or by syncing manually at any time), alongside manual entry of data. This has been outlined in D-FR2, which also mentions that there should be some means of saving all types of activity mutually.

This was another part of the requirements specification that was clarified with the client, where it was decided that categories of activity data could be expected as always a numerical format and should be dynamic to allow for more categories to be added. A final clarification here was that although dynamic addition of categories of data should be possible, three types (walking, cycling and running) will suffice at this point.

1.1.3 Data auditing

Alongside the entry of data, D-FR8 was a feature of the application that the client was intent on being implemented. The auditing of data should be possible for administrators and for participants (where the participants are only changing or removing their own data), and notes should be left by the remover to allow for a reason to be left to a user wanting to know why data was modified or deleted.

An extension of the auditing data feature that was clarified again with the client is the auditing of all actions by the administrator. Unlike the data auditing, this would mean an auditing record should be created for any administrator level user that performs an action only available to them. Another highlighted clarification of this requirement is that all auditing should be final, and that no rollback functionality is required. For instance, if an administrator deleted data from a user, to undo this action would not be possible.

1.1.4 Challenges

Linking back to the way users interact in the system, challenges is one of the largest and most important aspect of the proposed system. C-FR1 to C-FR4 outline the challenges requirements, though again clarifications have been made since starting this project. Rather than assuming a challenge will be sent to another community or group for them to accept, instead the flow of challenges should be as follows:

- Coordinator of a group creates a challenge
- A coordinator from another group joins the challenge
- Both groups users can compare the competitions progress and statistics

In addition to the information about overall progress, users in a challenge should be able to see a league table within their own group. It was clarified in the QA session that summaries for each user should be available to all others in their groups, meanwhile outside their own group, only a very basic stat should be visible. This goes also for privacy of names and personal information, which should only be visible by people in their own group, although users will need control of their user name, which could be made anonymous if requested. As for higher level users, administrators will see all users details within challenges, and coordinators will see the same information but at a restriction of only overall data.

Upon completion of a challenge which will be decided by length of time of the event, it will be the responsibility of the coordinator that created the challenge to publish results and inform any participants taking part. Although this should be automated, initially sending out results should be completed by the coordinator to give time for validation of the data. Alongside the emailed out results, data from completed challenges should be visible on any participants of the challenges' dashboard upon next log in to the application.

A smaller additionally desired but not essential feature discussed during the QA meeting was extra results for categories such as 'best walkers' etc.

1.1.5 Updates and emails

This brings this part of the requirements specification to how emailing will be handled. Emails are an additional though highly desired part of the brief responsible for both sending mail out for completed challenges and for reminding inactive users to register data. Where reminders and challenge news is sent out, the amount of time between each automated email should be modifiable by an administrator.

As per discussed with the client, the default times for each email type should be as follows:

- Participant progress emails- Weekly
- Missing readings emails- Daily
- Results of users- Manual

By combining the automatic tasks of sending emails and receiving data from external services, it should be possible to make use of a general scheduler to help perform a range of actions when the user decides.

1.1.6 User interface and internationalisation

Because the application will be a web-application, it is key that the application has an easy to use appearance that is preferably mobile friendly. The client has agreed that the default appearance using the Bootstrap[9] is allowed here. In addition to a friendly user interface, internationalisation is a required feature of the system, and should provide the user the ability to switch between at least the English and Welsh languages with opportunity for more.

1.1.7 External endpoint

Although part of D-FR5 which covers the requirement of receiving data from a variety of means, a special mention to the requirement that the application should have an API endpoint. The API (to be SOAP) will be required to allow interaction between different servers running the application in order for different communities to communicate and send challenges.

Alongside this, the API endpoint should also be able to receive generic activity data to be inserted into the system (though the client has clarified that this part of the API will only require POST, as to only receive data).

1.2 Desired and required libraries

In addition to the requirements of the application, the client has requested that third party software should be made use of to encourage a faster implementation of the product. Uses of third party software was discussed during initial meetings, with some group members already having a good knowledge of possible email libraries and OAuth libraries to help with the connectivity to Fitbit and Jawbone services. Both of which will be discussed further in this report, with reasoning for why such libraries were chosen.

Chapter 2

Development Methodology

Chapter 3

Design

This section outlines the initial designs for the system to be produced. As this project is being developed using a Scrum based development methodology, there is no concrete up front design for the system. Much of the design presented here was created during the initial stages of the project and therefore has been subject to change throughout the implementation stages of the project. However, we were not going to start development on the project without any design whatsoever, especially since nearly all team members were unfamiliar the technologies we were going to be working with. Unless otherwise stated all designs are applicable to both the .NET and Java EE systems.

3.1 Use Case Diagrams

The first real piece of design to be undertaken was a detailed analysis of the requirements specification. This initial step aimed to tease out what was likely to be the most challenging, unintuitive elements of the project. Through this discussion we were able to draw out what we thought were the major use cases for the different system users.

Figure 3.1 shows the use cases for registering and logging in a user to the system. Note that all types of the system user can perform these actions regardless of their role. By "vanilla" login/registration we mean a custom login system specific to site that does not interact with another site via SSO etc.

Figure 3.2 shows the different actions that can be performed to administer a user account with the GoAber system. These are broadly pretty common sense and would be the sort of actions normally expected of system such as this. Note that in figure 3.2 administrators can do everything a participant can plus modifying their privileges. Changing user privileges is the only action that cannot be performed by anyone other than an administrator. "Deactivating" a user account will leave an audit record in the system but will remove all activity data as well.

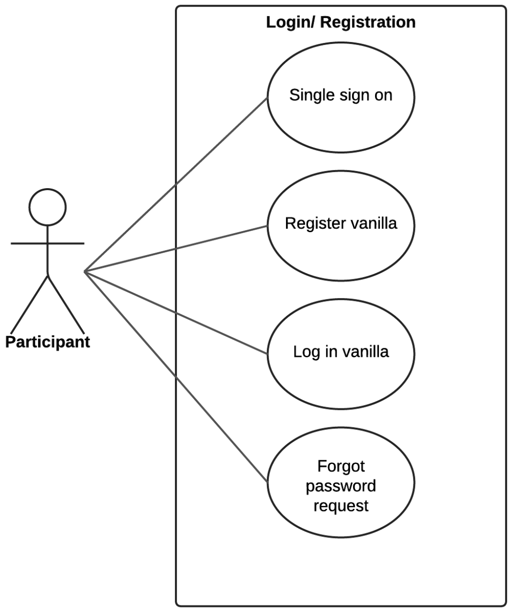

Figure 3.1: Shows the different login & registration actions that system user (participant, coordinator or administrator) can take. These actions are associated with D-FR11.

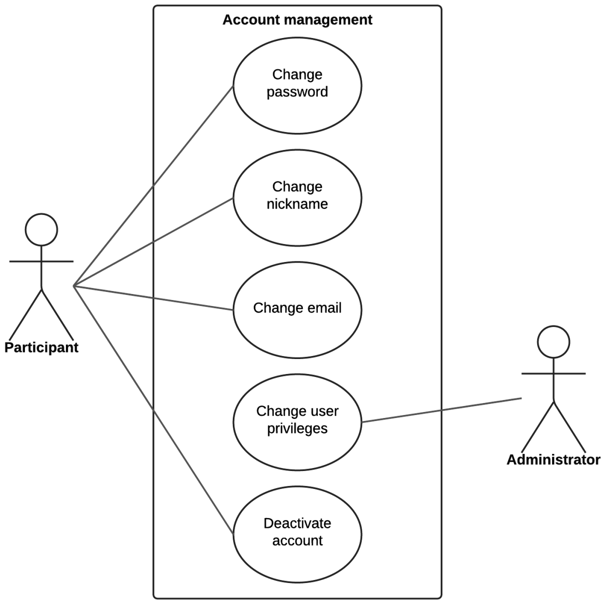

Figure 3.2: Shows the different account administration actions that can be performed by a participant of the system. This references requirements D-FR1 and D-FR10.

The next use case diagram (figure 3.3) is arguably the most important in the series. This gives a loose summary of how the users will interact with their activity data in the system. It also shows which actors have permission to carry out particular actions. As mentioned before, administrators can carry out all actions that co-ordinators and participants can. Co-ordinators can only view information about themselves and others, much in the same way as users, but can obviously perform CRUD actions on their own data.

The diagram shown in figure 3.4 shows the interactions that can be carried out on groups of users. Administrators have the permission to perform CRUD operations on a group and have the ability to add participants to a group. Participants are only able to view a summary of other people in the group.

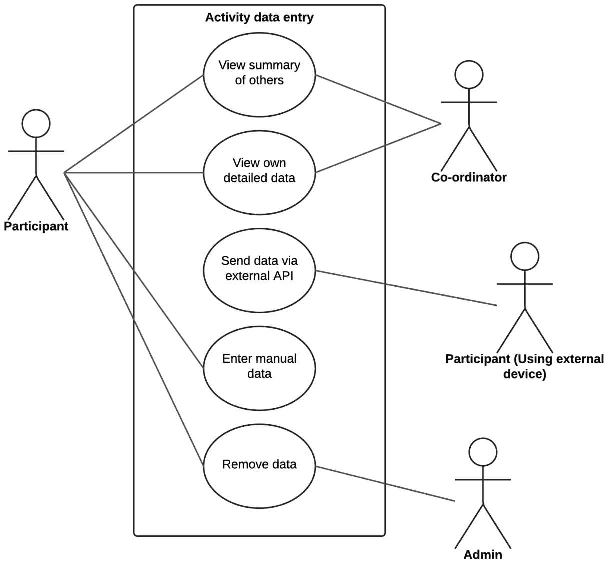

Figure 3.3: Shows the actions that can be performed by the three types on activity data. These were taken from requirements D-FR5, 7, 8, 9, 10

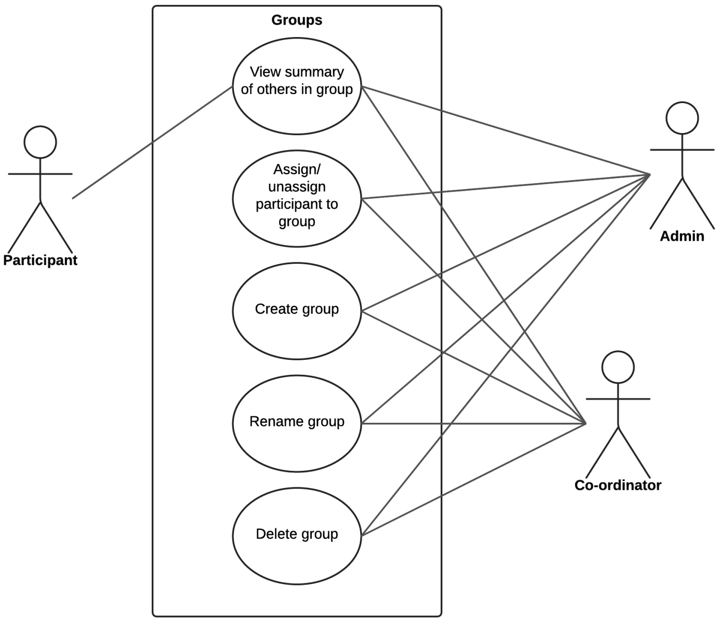

Figure 3.4: Shows how actors will interact with groups of users. This is in relation to requirements D-FR1, D-FR10

Use cases for challenges are shown in figure 3.5. Here, the major differences to be aware of is the difference between what actions a user and coordinator can perform. Users can only view information about challenges while coordinators can setup and edit challenges between groups and communities. Administrators will be able to perform all of these actions.

Figure 3.6 shows a couple of additional participant use cases which will be required in order to allow users to authorise their devices with our system. Participants should be able to authorise their devices another system via OAuth.

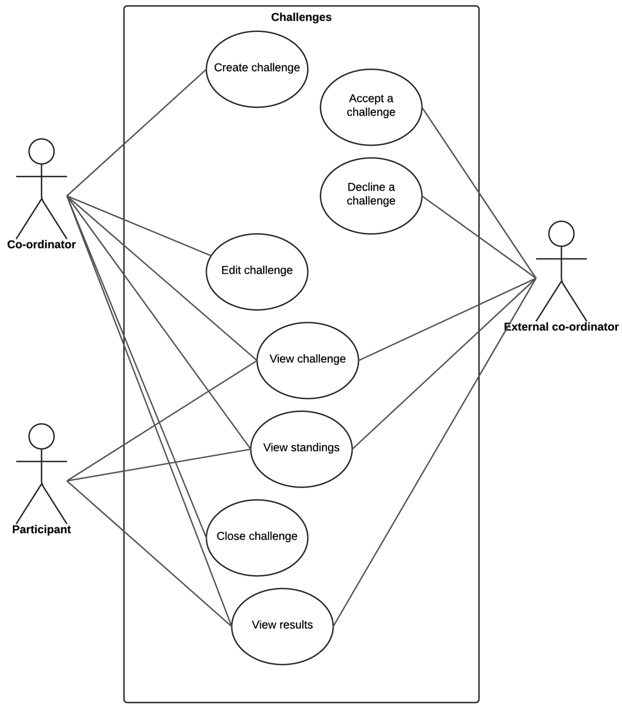

Figure 3.5: Shows how actors will interact with the system in terms of challenges. These reference requirements C-FR1-4 and E-FR1

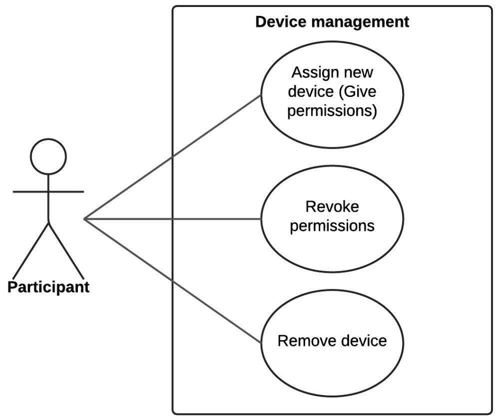

Figure 3.6: Shows how a user can authorise a device with an external system (e.g. Jawbone/Fitbit). This relates to requirement D-FR3.

Finally, figure 3.7 shows some additional use cases for system administrators. This figure simply states some additional actions which were not included in the other use case diagrams. Administrators should be able to schedule there emails sent out from the system as well as edit and delete users.

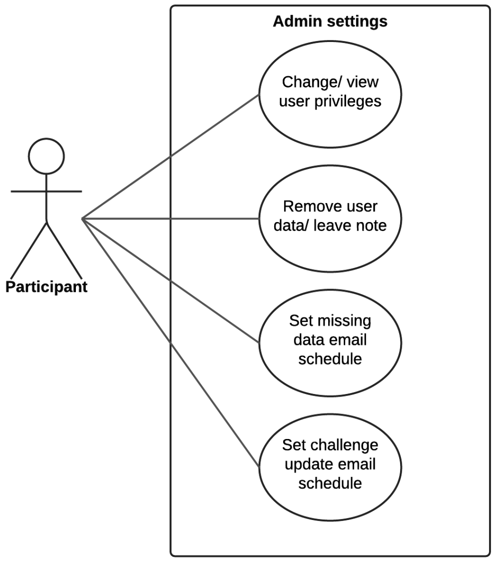

Figure 3.7: Shows some additional uses cases that can be performed by a system administrator. These action are all taken from requirements D-FR1, D-FR8, E-FR3 and E-FR4.

3.2 System Architecture

This section discussed the high level system architecture that we conceived in the very early stages of the project. Figure 3.8 shows a formalised version of the early system architecture. We in no way expect the final design of the implemented system to accurately reflect this diagram. This diagram was a useful by product of early discussions to try and understand how the final system might hang to together. This was a key discussion that lead to us identifying parts of the system that we did not readily understand or that we found to be ambiguous.

This design was originally produced in rough on a white board, allowing us to shuffle key elements around until we were in agreement on how each part should work. Afterwards the diagram was formalised into figure 3.8 to be preserved as a design artefact. This overview is meant to be independent of the technology used (either .NET or Java EE). This, combined which the fact that a Scrum based methodology can and will allow us to be flexible with the design are the major reasons why the final system will almost inevitably differ from this diagram. However, as mentioned, it played an important role in getting all team members on the same page before diving into implementation.

The system in diagram 3.8 can be split into two parts: “our system” on the left hand side and the “outside world” on the right. The outside world consists of regular human users (participants, coordinators, and administrators) but also includes other computer systems. For example, other GoAber systems need to communicate information about users and challenges between one another. Other systems that need to be communicated with are the Fitbit/Jawbone servers and with generic SOAP input.

The diagram shows several connections from our system to the outside world. The most obvious one, in the top centre of the diagram, shows that users can connect to a front end website. Conceptually this component is broken down into two parts: the views (what the page looks like) and the controllers (how the views get built displayed). The models sit further back in the system and are accessed by the controllers.

Below this component is the SOAP web API. This provides an access point for remote systems (i.e. non-human users) to communicate with our system. This includes both other community servers and potential other devices.

The third point of access in the bottom centre of the diagram is the most complicated part of the external communication systems. This shows a scheduling component, inside of which is nested a data collection component. The scheduling component will be responsible for firing off events both internally and externally. For example, internally this will be responsible for closing a challenge on time. Externally it will be used to periodically request data from the Fitbit and Jawbone APIs and as a timer to send out emails.

Sitting behind these front three layers is the business logic and data models for the system. These are shared by all three of the components described above. This part of the diagram is deliberately left more vague than the other parts of the diagram. As mentioned in the preceding chapter, we are using a Scrum based methodology and producing a big up front design would be going against its guidelines. Additionally this section of the system is highly likely to be specific to .NET and Java EE. For this reason we will choose to keep the low level design decisions of this part of the system up to the developer. Broadly speaking our approach in this project will be to keep the .NET and Java EE systems structured similarly. Both systems should share the same model structures, unless the specific implementation forces us to change for some reason.

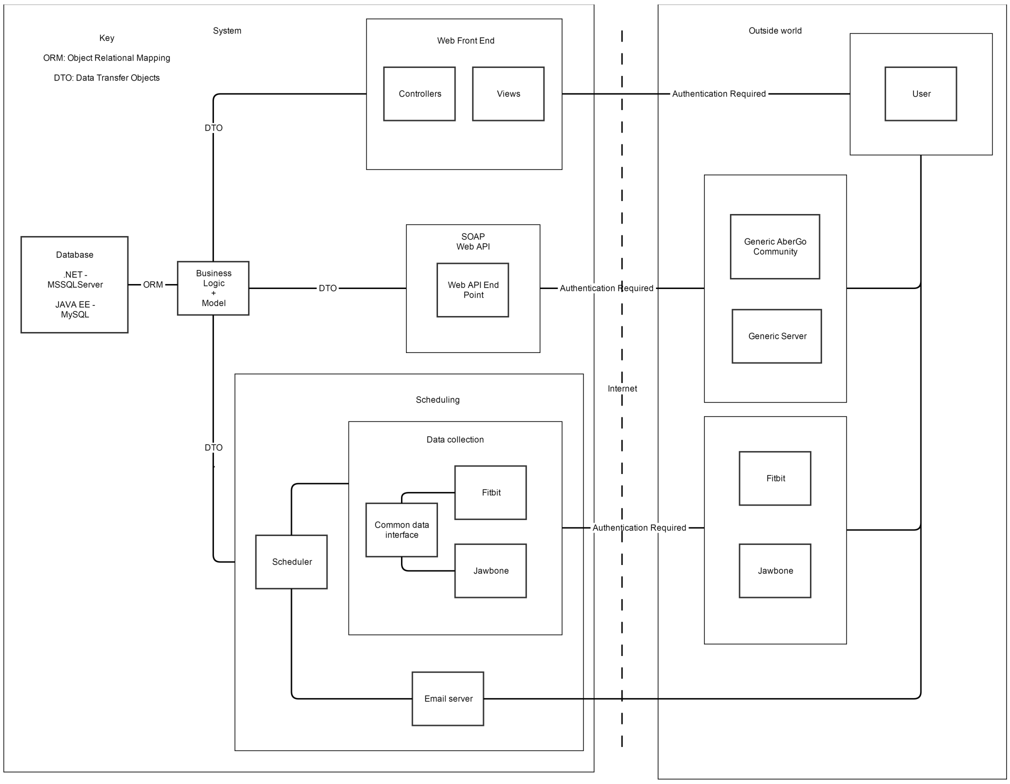

Figure 3.8: High level conceptual overview of the proposed system.

3.3 Database Design

The second major piece of design that we undertook in preparation for the implementation of this project was to come up with a entity relationship diagram which would form the basis for the model code in both versions of the application. Once again, in practice it is likely that this design will need to change once we become more familiar with the two different technologies we are using.

This, like the high system architecture in the previous section was very important to do early on in the project. It helped to solidify how we were going to represent objects on data in our system and gave good starting guidelines for the team members who implement these this design.

Starting to the right of the diagram is the *ActivityData* entity. This is perhaps the most important entity in the entire system. An *ActivityData* item is a single piece of activity data for a particular user. An *ActivityData* entity is associated with a particular system *User* and also has a reference to a *CategoryUnit* entity. A *CategoryUnit* is simply a linking table between the categories (such as “running” and “swimming”) and units (such as “steps” or “strokes”).

The system is designed in this way so that there is decoupling between the value of the data stored in the system the category or unit it belongs to. Every activity data item will simply be stored as a numerical value. The interpretation of that value is determined by what the associated category or unit is. This allows the database model to be flexible to the number of different types of activity data that the client may wish to store in the system.

This formulation has several distinct advantages. Firstly with this approach there is no need to introduce null entires into the table as you would have to do the type of the data was store alongside the value itself. Secondly, it means that it would be easy to add the ability to introduce new categories and units should the customer require. Thirdly this makes the system almost completely independent of the type of data that a user might want to store. As long as the activity data item is a numerical value, no changes to the database structure are required to introduce the new type. The only exception to this is if the new type of activity data to be introduced happened to be categorical instead of numerical. But even in this circumstance another table could be easily created (e.g. *CategoricalActivityData*) in order to support it.

Moving onto other parts of the model; in the centre of the diagram is the *User* model. This will store almost all of the information about a user (name, email etc.). A user record also has a link to a user credentials table, which contains their password for the system and other authentication data. Additionally users also have a user role associated with them. The user role specifies what type of user they are and permissions they have (e.g. participant, coordinator or administrator).

Along side this user data a user may also register several devices. The devices entity stores the data about a specific device that a user has connected with the site. Each device has a device type. The device type stores the details for connecting to a specific third party site that can be polled by our code for activity data. In this project those sites will be limited to Fitbit and Jawbone.

The top of the diagram shows the tables that will be required to implement the challenges portion of the system. Each user can be associated with a group. A group conceptually a list of users. Each group belongs to a single community. Groups can have challenges associated with them. A challenge entity contains information such as the start and end date and the type of activity data that is associated with it. Participants in a challenge are linked using the *UserChallenge* entity.

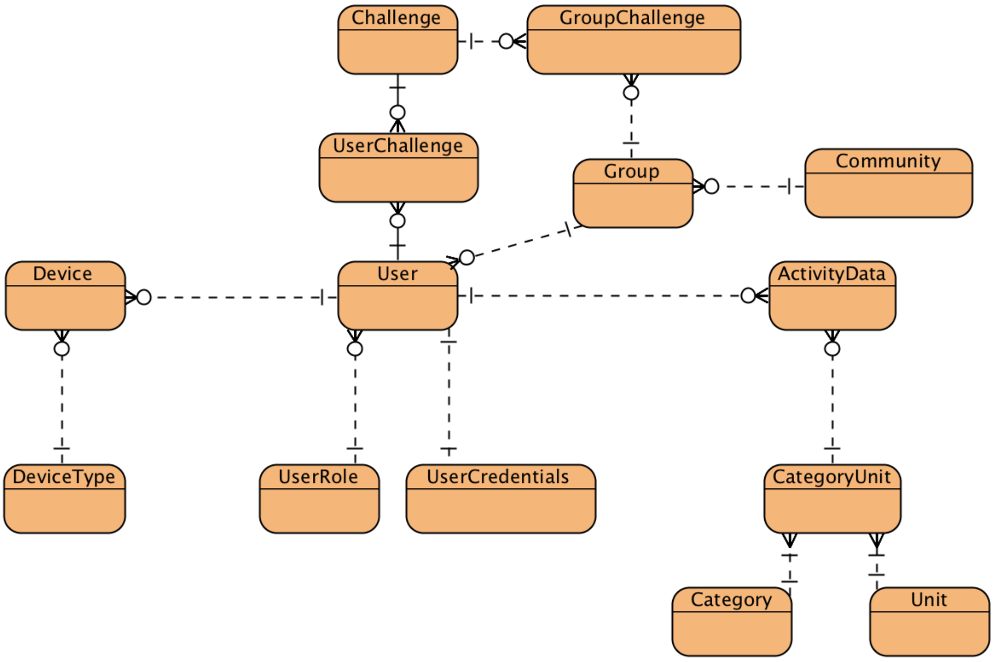

Figure 3.9: A entity relationship diagram showing how the data model in both of the applications interact with one another.

3.4 Activity Diagrams

Alongside the other diagrams presented in this section we also found it useful to produce some state diagrams to try and get a better idea of how a user will transition from one state to another around the site.

Figure 3.10 shows a activity diagram for authenticating a user in the GoAber system. There are several different paths that a usr should be able to follow though the login procedure. If they are already registered they can directly login. If not they my first register with the system before proceeding. In either of these cases if their details fail to validate they are redirected to the appropriate page.

Figure 3.11 shows the workflow for both a participant and an administrator needs to pass through in order to delete activity data from the system. In the case of a participant they should be asked to confirm the deletion. If the user is an administrator they should be also asked to provide a reason why the data is being removed.

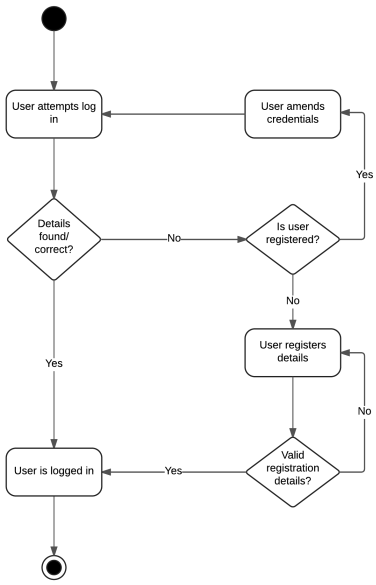

Figure 3.10: An activity diagram showing the states that a user can transition between when authenticating with the system.

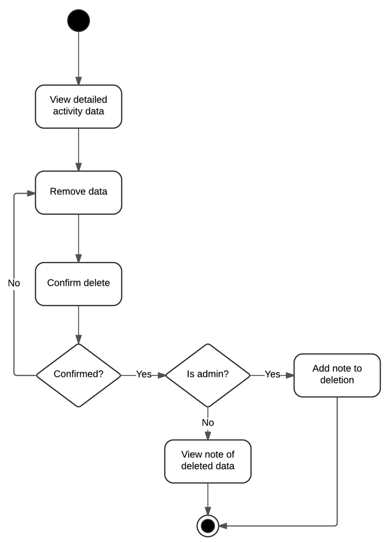

Figure 3.11: An activity diagram showing the states a participant/administrator passes through in order to delete activity data.

The next figure (3.12) shows the states the workflow for authorising a users device (e.g. a Fitbit or Jawbone device) for use with our system. Users should be able to view which devices are connected to the system and revoke access is desired. They should also be able to add a new device and any errors in the case that the external site returns a failure.

Group management activities are shown in figure 3.13. Most of these actions are fairly self explanatory but there is a slightly non-trivial case where we wish to add a user to the group. In this case we must check make sure that they are removed from their old group as they can only ever be affiliated with one group at a time.

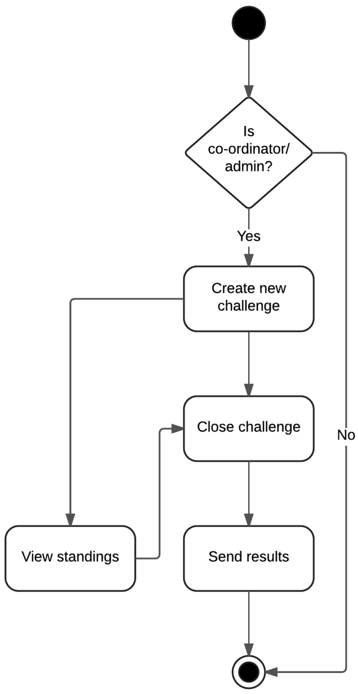

Figure 3.12: An activity diagram showing the workflow for device authorisation.

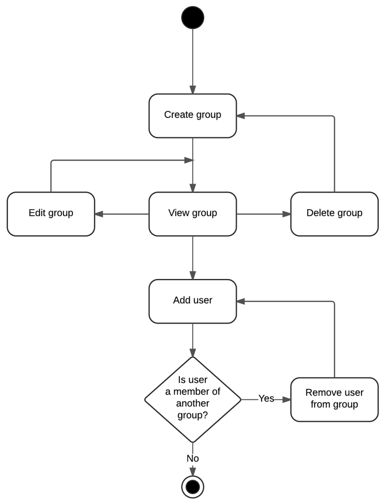

Figure 3.13: An activity diagram for the management of user groups in GoAber.

In figure 3.14 the activity flow for entering data into the system is shown. Activity data can be entered manually by participants in which case there data is validated on submission. With external systems there is the potential possibility of a connection failure and that we get bad data back. This could be either from a device API or manual input for a third party device via the SOAP API.

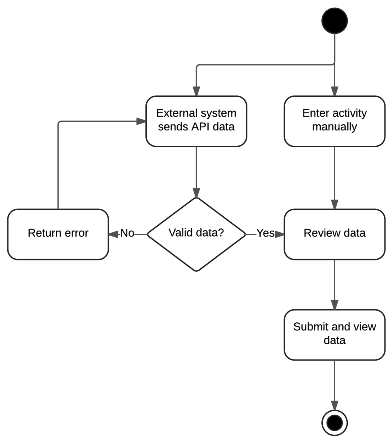

Figure 3.14: An activity diagram for manually inputting user data into the sytem.

Chapter 4

Implementation

Chapter 5

Testing

During this project many different testing methods were performed. This section will discus the testing that has been carried during the different stages of the project.

5.1 Unit Testing

5.1.1 JavaEE

While developing we placed all back-end code within service classes. This would allow unit tests to be performed. However in some cases the use of mock class would also be required to mock the classes accessing the database.

In order to mock the behaviour of the facade classes Mockito[3] has been used. All facade methods which the class under test calls are mocked.

images/testing/UnitTestsJavaEE.png

Figure 5.1: JavaEE unit tests.

5.1.2 .NET

This initial plan was to perform unit testing in the same manor as JavaEE. Unfortunately the .NET service class are not as easily testable. Nearly all of the code in these service classes contain SQL. When running the unit tests the database is inaccessible, therefore only code that does not access the database can be tested.

To get around the database access issues we considered using of Moq [1]. However this would cause considerable change to the code as interfaces would be required for all classes being mocked.

Using unit testing the controllers was also considered, but again mocking classes would have to be used to stop these from accessing database code. As cucumber testing was already being used, little would be gain by adding controller tests. The cucumber tests would allow the checkout that the correct views and database is presented to the user.

images/testing/unitTestsNET.jpg

Figure 5.2: .NET unit tests

5.2 Continuous Integration

5.2.1 Automated Continuous Integration

.NET

Visual Studio Online allows continuous integration to be performed automatically. This allowed us to test that our project builds correct after a developer has integrated their work with the main development branch.

Due a the limited time of build time available, during the first 4 sprints the automatic build was only triggered when a merge to master occurred. This merge happened at the end of every sprint. During the latter stages of the project the automatic build was set to the "develop" branch. This allowed us to discover bugs more quickly as the final parts of the project came together.

As shown in figure 5.3 some builds failed. When a build failed an email was automatically sent out to all developers. This allowed the bug to be quickly resolved and the build to be re-ran.

images/testing/builds.png

Figure 5.3: Selection of builds that have been run during the project.

The summary information on the build (figure 5.4) allowed us to view how long the build look at if there are any issues. For example warnings about unused variables.

images/testing/ExampleBuild.png

Figure 5.4: Example of a build that has been performed.

JavaEE

Consideration was given to changing the JavaEE project into a Maven project. This would allow Visual Studio Online to build the project. However it was not discovered until a considerable amount of work had gone into the project that Visual Studio Online used Maven. A decision was made that changing to Maven at this stage would take too much time away from development, and the manual integration testing was providing a sufficient method of integration testing.

5.2.2 Manual Integration Testing

Throughout the first 4 sprints of the project no developer was allow to push to the develop branch unless two other group members had tested and reviewed their code. This was enforced using Visual Studio Online. Due to the number of these pull request being made this changed to one developer in order to speed up the process.

images/testing/PullRequest.png

Figure 5.5: Example of a pull request

For a pull request to be approved the project must build and the tasks stated by the creator should work as stated within the request. Within each pull request steps on how to test the functionality have been provided. An example of this is shown in figure 5.5.

As well as checking the functionality the reviewer also performed a small code review. This would encourage the use of clean commented code, which follows the MVC design pattern.

As well as finding many bugs, this also allow multiple members of the group to learn about how a requirement of the system had been implemented.

5.3 User Interface Testing

To test the user interface SpecFlow [2] was used. SpecFlow allows the creation of behaviour driven test that are written in plain English scenarios contained within feature files. These scenarios (figure 5.6), along with some step definitions (figure 5.7) allow interactions with the user interface to be performed.

images/testing/cucumberScenarios.jpg

Figure 5.6: Cucumber test scenarios

images/testing/cucumberCodeSteps.jpg

Figure 5.7: Cucumber steps

PhantomJS [5] allowed us to run the tests without opening an internet browser. This will allow the tests to be ran by our continuous integration server. Selenium web driver is used to select and control the web browser.

Tests to check that no invalid data can be entered into the forms are performed. This includes boundary checks, valid type checks and valid format (e.g. email format) checks. The results of the tests are shown in figure 5.8.

The steps (shown in figure 5.7) only have to be written once. These tests interact with the HTML in the web browser, therefore, they do not care about the underlying technologies. By just changing a URL these test can be used for both the JavaEE and .NET projects.

images/testing/cucumberPassedTests.jpg

Figure 5.8: Cucumber test results

5.4 SOAP Communication

To test that 3rd-party applications can send our application activity data a second application was created. This second application just sends activity data to our main application.

5.4.1 .NET

For .NET a GUI application was created to allow the entry of an authorization token. (Shown in figure 5.9) This token is send to the main application along with the activity data. The main application will check the validity of the token before saving the activity data to the database. The activity data will be associated with the user that is currently logged into the application.

images/testing/clientAuthTokenNET.png

Figure 5.9: The .NET SOAP client application.

Unit tests have also been added to the project to test that the data can be sent and the valid return value is received. (Shown in figure 5.10.)

images/testing/SoapUnitTestsNET.png

Figure 5.10: Unit tests for the SOAP operations in the .NET project.

5.4.2 JavaEE

The JavaEE SOAP test application is a command line application which, like the .NET application, allows the user to enter an authorization token. When running this application the user does not need to be logged in. Which users account to added the activity data too is specified in the request. Unit tests have also been added to this project.

images/testing/SoapUnitTestsJavaEE.png

Figure 5.11: Unit tests for the SOAP operations in the JavaEE project.

5.5 Cross Browser Compatibility

Throughout the project the applications were tested on multiple browsers this includes Chrome, Safari, Internet Explore 11, Edge and Opera. Both application should work on all modern browsers.

5.6 Community Communication

To test that a community could send a challenge to a different community, we ran instances of the JavaEE and .NET projects on different computers. These tests involved checking that both instances could send and receive challenges, and that the community which started the challenge sent the results to all other communities.

Chapter 6

Project Status

During the project, meetings with the client took place to discuss which features would be given a low priority in the backlog. This section will discuss features that remain on the backlog, and the reasons for this.

6.1 Incomplete Features

6.1.1 Administrator setup of Categories (D-FR2)

The functional requirement D-FR2 states that "The system will provide a mechanism, for an administrator to setup the categories of data that can be stored for a community". This functionality has not been provided. Users can store any category of data, and community league tables are available for all categories.

6.1.2 Fitbit (D-FR3)

Due to technical difficulties the JavaEE system is unable to obtain data from Fitbit. For authorisation to devices we use a library called scribe [8]. Unfortunately, this library is unable to retrieve the access tokens from Fitbit. After manual attempts to put together the authorisation header, an agreement with the client was reached, that this feature would be put at the bottom of the backlog. Jawbone communication, manual entry and SOAP have been implemented to allow activity data to be entered into the JavaEE system.

6.1.3 Email notifications (E-FR3, E-FR4 and C-FR4)

As agreed with the client email notifications were given a lower priority on the backlog. Due to time constrains this feature was unable to be implemented. The means that users do not get notified when a challenge has been completed, and do not receive reminds when they are inactive.

6.1.4 Authentication and authorisation (D-FR11)

The requirements state that Single Sign On (SSO) will be used. Our initial plan was to start with having our own vanilla login systems and add SSO at a later stage. However, SSO has remained unimplemented due to time constraints.

6.1.5 Individual Progress (E-FR4)

It was requested that a total distance in miles/kilometres would be provided. This has not been included in the final product, again due to time pressure to deliver the project, but is a feature that could easily be added in future development.

Chapter 7

Critical Evaluation

We found that there are many differences between JavaEE and .NET. Throughout the project many things have gone well while developing the two applications, and there are many things we would have done differently. In this chapter we will evaluate these.

7.1 Platform Comparison

7.1.1 Implementation

JavaEE splits the code into two projects : the WAR containing the web pages; and the EJB which should contain everything else. This allows the UI to be clearly separated from the rest of the code. Within .NET all code is contained within a single project, which can lead to parts of the view being intertwined with the controllers and models.

Despite everyone having previous experience in using Java, and only a few having previous C# experience, we felt that the learning curve for JavaEE was greater than that of .NET. The .NET MVC framework hides a lot of the complexities from the user. JavaEE is more flexible, which leads to more complications. Therefore a lot more time was spent on the JavaEE project than the .NET project.

An example of this is the implementation for the user's activity data pages. The AJAX used to display these graphs requires a JSON representation of the activity data. In .NET this done through requesting the model as JSON object. In JavaEE a model can not be converted into JSON, this meant adding an extra class that convert the ActivityData object into a ActivityDataDTO object which could then be converted to JSON.

The JavaEE application kept caching data. After entering data via the forms, the data displayed in the view pages would not contain the update; until the page was refreshed. An annotation was added to the class to prevent it from being cached. This problem was not experienced when developing the .NET application.

Implementing the Jawbone and Fitbit connection using OAuth was more successful in the .NET than in JavaEE. The library used for OAuth in .NET was a lot more stable and documented than the libraries used for JavaEE. The OAuth version for both devices has also been recently updated, which meant that was very few examples.

In JavaEE there are many options for the scheduling of jobs, whereas in .NET Hangfire appears to be the only option. Having multiple options allowed us to pick the scheduling method which suited our situation.

Often after pulling changes into our local copy of the git repository the JavaEE project would not run. Even when no changes to the database had been performed, this would require the re-creation of the database. However, before recreating the database we needed to make sure no users are logged into the application, as those users would no longer exist in the database.

7.1.2 Internationalisation

Bundles were automatically generated in JavaEE to allow the internationalisation of the application. Other than adding these to a configuration file no extra work was required.

In comparison to JavaEE the .NET application's internationalisation was more difficult to setup. This involved creating a controller which grabbed the user's language from a cookie. All other controllers inherit from this controller.

7.1.3 Database

JavaEE's named queries versus .NET's LINQ library had mixed opinions within the group. Some like the abstracted view of SQL that LINQ provides. However, this leads to the SQL being found within views, models and controllers. The named queries allow the SQL to be kept separate from the code and are closer to standard SQL syntax. Some members of the group found this less intuitive than LINQ.

When an updated to the database was required the .NET project's models were changed and the "add-migration" command was ran. We found it a lot more difficult to updated the JavaEE project's database. These changes were performed through editing the database and then updating the code. This often required manual modifications to the code. Using the code-first entity framework, in .NET, worked more smoothly.

7.1.4 Configuration

A positive for using glassfish was that a script could be produced to automatically setup realms for user authentication. Configuring the project to allow authentication is done on the server which extracts the configurations from the code. In .NET the authentication is within the project itself. Though it was easier to configure the .NET authentication, we found the JavaEE way of separating this out is cleaner.

.NET has a vast amount of libraries available that are installable by the NuGet package manager. This allows aspects like selecting which OAuth library to make use of a lot simpler. When looking for a JavaEE OAuth library many different developers had different recommendations, and it was unclear to which would be the best one to use.

7.1.5 Testing

When testing the projects we made use of mocking libraries to mock the database interactions. In JavaEE this can be done through just mocking the facade the class under test is accessing. Only the functions that it is calling need to be mocked.

In .NET not only does the whole model class which the class under test is using have to be mocked, so do all models accessed via its foreign key constraints. This meant that even if we wanted to unit test a small section of the software the majority of the models would have to be mocked. The class under test also had to be modified so that the mock class could be passed to it.

To allow 3rd party application to send our application data a web service which enabled communication via SOAP was created. To test this a separate application in each language was created and the service was imported into the application. In .NET this service could be updated through a right-click menu option. This was more awkward to update in JavaEE, where the web service had to be deleted and re-imported.

7.1.6 Conclusion

We found that JavaEE was far more flexible in what it allows us to do, but took longer to develop. With added flexibility came added complexity. Whereas .NET does everything it can for you and forces you stick to a common way of implementing.

7.2 Development Methodology

At the start of the project we struggled to get going. None of us had any previous experience of how to perform the initial setup of a project. For example we had to work out how long the sprints should be, and which tasks to assign each sprint. We decided to allow two for the first sprint and one week for all other sprints. This turned out to work well.

During the first sprint, according to the burndown, we became about 30 hours behind schedule. This slowly improved as the project progressed. As we became use to the technologies our development started to speed up and our predictions for the length of time tasks will take became more accurate. We also planed for delays and had included a empty sprint. This sprint was used to test, document, tidy up the project and complete outstanding tasks.

7.2.1 Design

We all agreed that we should have done some design at the start of each sprint. At the start of the sprint we handed out the tasks, and implemented them individually with little group discussion. The design discussions would have allowed those with previous experience in the technology to give ideas and advise on the best practices. It would have also given everyone a clearer idea of what everyone else was doing.

However, due to the lack of experience within the group this would not have been possible for all tasks. Many of the tasks involved researching about the technologies. In these cases it would have been useful to feedback into the group the design decisions that had been made.

We also wanted to strike the balance between too much design and too little design. Having addition design meetings would have taken away from the time spent implementing the applications.

7.2.2 Setup

At the start of the project we were not aware that Visual Studio Online would require our JavaEE project to be buildable using Maven. We should have looked into continuous integration testing at the start of the project. To change part way through the project would have required us to create a new Maven project and then copy across the source files.

We kept on getting git conflicts due to configuration files, especially within the JavaEE project. These files are needed so that the correct dependencies and files can be loaded into the project, however these caused merge conflicts throughout the project. We also had merge conflicts in the .NET project's database migration files. These were added to the git ignore, but due to them still being referenced with the .csproj file caused the build to constantly fail.

7.2.3 Implementation

Three out of the five developers used Apple Macs to develop on, however Visual Studio will not run on OS X. This meant working from virtual machines. Due to hardware capabilities these were given low amounts of RAM. When MS windows detects that it is low on memory it starts to close programs. This caused development to take longer that it should have.

Through the majority of the project two developers were required to check through the work that had been performed before it could be pushed to the develop branch. This was a good way of working and many bugs were found through doing this. It also made sure that two other developers knew what features had been implemented.

However, having two people test every feature slowed down the development process as it took awhile to get these approved. This delayed dependent features from getting started. For the last two weeks this was changed to one developer to speed up this process.

Little commenting of the code was performed, and the coding standards were not discussed at the started of the project. We tried to stick to a commonly used coding standard for the languages. However, switching between methods starting with uppercase letters in .NET, and lowercase letters in JavaEE, often caused standards to merge.

7.2.4 Agile Practices

During the project some pair programming was performed. This was a good way of solving bugs and overcoming technical challenges. With more time it would have been good to fit in more pair programming. This would help standardise the coding style and allow multiple develops to learn about aspects of the system.

We planed to perform code reviews and perform static code analysis. In the initial meeting about development methodology we planned to have a few group code reviews to check that everyone is following the same coding standards; then code reviews would be performed in smaller groups throughout the project. Unfortunately due to time constraints this did not happen.

At the end of every sprint we performed retrospectives, giving us the opportunity to review the work done for a sprint. This also gave the group a specific date to complete tasks by. Rather than reviewing work this meeting was often used to complete pull requests and perform the merge to the master branch.

Unit testing was not started until a few weeks before the deadline. It would have been useful to having looked into unit testing at the start of the project, and setup the tools required to do this. Unit tests have been added to the JavaEE project, however the .NET project has very poor unit test coverage.

7.2.5 Development Tools

Communication was primarily done via Facebook. This was a good way of arrange meetings and sharing setup instructions. However, we believe that the project would have ran more smoothly if we all worked in the same office and had the same working hours. This would have allowed questions about the code others have written to be answer straight away, and ideas to be shared more easily.

Making use of Visual Studio Online allowed us to easily track the progress of the project. Each week we reviewed the burndown graph to see how fair behind schedule we were. We could then plan the following weeks working taking into consideration what tasks were carrying over into the new sprint.

Due to the limited amount of users allowed to contribute to a Visual Studio Online we also made use of GitHub to allow our customer and project manger to view the code being produced. Are local versions of the repositories were then setup to push to both remote repositories. This was useful as when Visual Studio Online was down we could use GitHub and vis-versa.

Everyone made good use of version control allowing us to keep track of changes. New branches were created for every feature. This allowed the braking up of the development. When a develop was waiting for a feature to be reviewed they could get started on a new feature using a different branch. Though we had some git issues through lack of experience, overall this worked well.

7.2.6 Conclusion

Throughout the project all developers worked hard to get the two programs working. We stuck to the development methodology throughout the project, and by the end the process had become much smoother at following it. Though not all features were completed, we have been able to deliver a comprehensive product on time.

References

[1] Daniel Cazzulino. Moq. `https://github.com/Moq/moq4`, 2015. [Online; accessed 01-December-2015].

[2] SpecFlow developers. Specflow. `http://specflow.org`, 2015. [Online; accessed 01-December-2015].

[3] Szczepan Faber. Mockito. `http://mockito.org`, 2015. [Online; accessed 01-December-2015].

[4] Fitbit. Developer api. `http://dev.fitbit.com/uk/`, 2015. [Online; accessed 20-October-2015].

[5] Ariya Hidayat. Phantomjs. `http://phantomjs.org`, 2015. [Online; accessed 01-December-2015].

[6] Jawbone. Jawbone up api. `http://jawbone.com/up/developer`, 2015. [Online; accessed 20-October-2015].

[7] Nigel Hardy Neil Taylor. Go!aber requirements specification. *SEM5640 Group Project*, 2015.

[8] scribejava. Scribejava. `https://github.com/scribejava/scribejava`, year =.

[9] Twitter. Get bootstrap. `http://getbootstrap.com/`, 2015. [Online; accessed 21-October-2015].