CS993 – Time Booking System

Team Lewis – Jonathan Bowen, Boakya Dankwa, Alistair Hill, Jane Howell, Calum Mortimer

# Introduction and Background

Target – 300 words

This report details the software development cycle of a Time Booking System requested by the client: John McGuire, owner of [Pulsion Technology Ltd](https://www.pulsion.co.uk/). The client addressed the class with his requirements for the project in late January of this year and gave a broad outline of the software he envisioned for us to create. We were given a short opportunity to clarify any of the initial requirements or ask about additional ones, with an additional follow-up opportunity to clarify requirements some weeks later via e-mail.

This report will guide the reader through the various stages of the software development cycle that our team undertook to produce a system that would meet the client’s requirements.

# Requirements

Target – 1000 words

For the successful development of the proposed software according to the client’s needs, it was critical that from the offset the requirements of the software were clearly understood and documented. This offers certainty to both parties. For us, the developers, this means that the product we are designing would be clearly understood. For the client, they would understand what product was going to be delivered at the end of the project.

A good, clear set of requirements would remove ambiguity and minimise the risk of conflict between both parties. It should be obvious to both the software engineering team and the client what represents a contractual change which requires further scope and what simply represents the software engineering team abiding by the previously agreed scope.

The project team’s approach to formalising the requirements with the client consisted of the following process:

1. The client produced a user requirements specification (Appendix A.1.) which detailed the requirements of the software system

2. A formal meeting was held with the client and the software development team to discuss the initial requirements and follow up with any questions in-person.

3. A numerated requirements list was developed by the software team from all of the available information (Appendix A.2.) which served as the initial set of requirements.

4. A numerated set of assumptions (Appendix A.3.) was developed by the software team and sent to the client for clarification.

Following step 4, no further communication was received from the client to the software team. Had the team managed to clarify the assumptions with the client and formalise the requirements list, this would have potentially allowed us to produce a formal quotation for the works required.

The following sections detail the workflow we pursued for requirements capture, the information which we obtained, and the steps we would take in the future to bring the project to eventual completion.

## Client User Requirements Specification

The client User Requirements Specification (URS) (Appendix A.1) lists the requirements of the software system to be designed by the project team.

Following review of the URS, the team concluded that the client was describing a standard software package for tracking the time employees spend on projects and tasks within a consultancy framework. This functionality is already available in several “off the shelf” software packages. One such example of a widely used package would be Oracle’s JD Edwards project accounting software, which allows employees to record time against projects to recognise project profit and revenue. One member of our project team had experience of using this software as part of their previous career.

One difference which stands out versus “old school” timesheet recording is the client’s request for a disconnected mobile application. An example of a piece of software which provides this functionality is the Harvest package of time booking software. It is highly likely that there are also many more packages available which offer this.

Given that the client’s requirements are likely to be met with existing packages. As a team, we missed an opportunity to clarify with the client *why* he is looking to develop this software from scratch. Performing our due diligence, and pointing out that there may be existing solutions, could improve our relationship with the client and lead to valuable future work. Furthermore, it could have been more beneficial to be paid a modest amount of money to do very little software development and a lot of market research, than it would be to be paid handsomely to attempt to reinvent the wheel.

Nevertheless, the following sections assume that a scope of work was agreed with the client to produce time booking software from scratch.

## Requirements Capture

The requirements list (Appendix A.2) was inferred from the User Requirements Specification produced by the client and the meeting which was held.

The methodology for gathering requirements was reading through the documentation and forming User Requirements Stories using flashcards. The following structure was used for each user requirement story: “As a \_\_\_\_\_, I need to \_\_\_\_\_, so that I can \_\_\_\_\_”. This ensures that there is consistency between each requirement, and that requirements are documented with a specific purpose. Each requirement should target a specific user group, provide functionality to the application, and result in some benefit for the user. If there is no user group, or no functionality to be added, or no purpose for the functionality – it should not be captured in the requirements list.

All the requirements are numbered, in order that they can be referred to throughout the project to maximise accountability.

Further to the requirements list, a set of assumptions (Appendix A.3) was also compiled which could be clarified with the client and each marked as valid, or invalid assumptions. These assumptions were also enumerated.

Both the requirements list and assumptions list were uploaded to GitHub so that they could be referred to and adjusted throughout the course of the project – maintaining their status as live documentation.

## Next Steps

The requirements list (Appendix A.2) and the assumptions list (Appendix A.3) should be formed into a formal document for the client to review – this will form the principal document for tendering for the works required.

Following from the client review, a meeting should be held between the client and members of the development team to sign off on the requirements documentation. Completing this step with close attention to detail will ensure that the project is kicked off on the correct footing and accountability for all parties is maintained.

# Design

Target – 1000 words

Using the requirements that were gathered in the previous design stage, the team began to think about the code we would have to write to fulfil the client’s requirements. The first approach the team employed was to make use of a design technique called (physical) Class-Responsibility-Collaboration (CRC) cards as a method of visualising and realising the application’s architecture and infrastructure.

This brainstorming method is useful for establishing the classes and data that would be necessary to develop the application before starting to code. Each data structure we designate as a class is written on an index card, then it’s responsibilities (i.e. the things it can do) are listed. From this we can complete-by-inference the final section, by listing the other classes that we could consider each class’ collaborator.

Other advantages of this high-level design include that it is easily communicable to the client, regardless of their technological knowledge. Being able to communicate effectively with the client at this crucial, early stage can prevent costly setbacks later in the development process. Furthermore, the CRC cards serve a dual-purpose, by also providing an unambiguous, strong springboard from which the development team can begin to code the system.

The use of CRC cards uncovered design challenges that we had not foreseen until that point. Our newfound ability to more easily visualise the intricacies of interactions between various ‘objects’ within the software led to many of the assumptions listed in Appendix A.3. Moreover,

The team also made use of another conceptual modelling process that is similar to CRC cards - a class diagram. The main purpose of such diagrams is to describe the architecture of our software in terms of its classes, data, and functions. Using information gathered in the requirements phase, a class diagram was created using UML (Unified Modelling Language) (Appendix A.4) which listed each class we intended to create, the data it would manipulate, and the methods contained in each class.

Another design method we employed in this stage was the use of design patterns. which bring a plethora of benefits to the project by providing reliable, known solutions to common problems encountered in any software design stage. Not only are time and money saved by not having to write and test the software/procedure anew, but time is also saved in the design stage by not having to create, from scratch, a certain procedure nor do we have to consider too deeply the implications of its implementation. Only a relatively small amount of effort will be needed to integrate any relevant frameworks with our code.

The team identified at least one design pattern that would prove useful in our design - the singleton - which ensures that only one instance of a class can exist. In this case, our system would only want one ‘master’ timetable to exist to prevent the time-booking conflicts that would very quickly occur if multiple timetables where instantiated. Furthermore, it was discussed that a singleton could be beneficial in instantiating only one database connection that could be shared my multiple users, as each user opening a new connection could result in excessive server loads – dependent on the number of users.

A further use of design patterns that we intended to use was an ‘Observer’. This pattern allows an object (the subject) – in our case, the timetable, a slot within the timetable, or an activity booked in a slot – to maintain a list of dependent objects (the observers). When a change is detected in the subject, the observers are notified of that change. As an example, this pattern would be especially useful to communicate booking changes to all users (observers) that have been invited to the meeting (subject).

Similar to design patterns, the team also considered various Java frameworks and common architectures. Such frameworks can range from pre-designed code ‘skeletons’ that solve a common design problem, with space left to add your own code, to virtually complete applications that require very little customisation. As established in section 2.1 of this report, the client’s requirements (at face value) can be met by several off-the-shelf solutions. Indeed, this could be to our advantage as it suggests that frameworks already exist for this kind of application. Should the client still require a custom solution, such frameworks could save much development time and lead to a more timeous delivery of the product to the client.

The common architecture that we would likely employ is the client/server architecture. The server controls clients’ access to the timetable and provides easy maintenance of a single timetable that is updated by many client requests. Linking this architecture with a relevant framework would give us a very strong foundation from which we can customise until all of the client’s requirements are met. One example of a framework we considered was Spring. Initially introduced to us during our laboratory sessions, the spring framework was found to be especially advantageous for java and web application development.

# Construction

Target – 1000 words

In order to construct the program, the team elected to use Java to construct the business logic. Java is comprehensive enough to cover all of the use-cases and commonly used for this kind of application. Most importantly, the team already has experience with programming in Java.

Starting from the classes and methods identified during design, we set out method signatures and unit tests (using JUnit) to drive development. We use test-driven development like this in order to ensure that our code meets the API specifications necessary to integrate it with other systems and code generated by other team members and, more generally, in order to ensure that we generate working code to begin with. These unit tests generated for development can also be used as unit tests for the specific functions for testing and maintenance.

Another aspect that's important for integrating code from different team members is code standards. Choosing and adhering to a code style saves time and effort spent reading and working with other people's code. It also helps to prevent errors being introduced from misunderstanding how code functions when attempting to extend it. Since the team is primarily trained in Java, for that part of the program we didn't introduce any conventions other than the standard Oracle Java conventions\*.

Since the design calls for at least three differenct systems co-operating (the front-end, business logic layer and database), one of the first goals for construction is to build a spike or single feature that uses all layers. Rather than developing all of the business logic first, choosing one feature to develop through every layer allows us to do any necessary experimentation with unfamiliar systems up-front and develop the necessary interfaces for the layers to interact that will be used for other features. If there turns out to be a problem with the design, working on a spike like this will allow us to go back and revise the design quickly rather than developing an entire layer and then having to redevelop or discard it because it can't be integrated with the others.

Having a functional piece of software quickly is helpful for showing clients what the development team is doing and ensuring it meets the requirements, or finding if some piece of the design or implementation needs to be changed. This is the strength of Agile methodology and rapid prototyping. Even if this isn't necessary, having a piece of software that works to build from can be good for team morale, which is a factor in any team activity including software development.

In order to create a deployable web application prototype, the team elected to develop for the Heroku platform. Heroku provides a platform for small-scale web applications that can be used for deployment or testing and, importantly for a student development group, provides a (limited) free service. Even if the application would ultimately be run from a server owned by the client, using Heroku allows for prototyping and demonstration of the application in a web context and without the development group setting up a server (which could be a barrier for student teams in particular).

Heroku deployments come with a small Postgresql database, so the initial prototypes of the application would be set up using this kind of database. Postgresql is open-source, making it convenient for use in a student project. If the client has their own database that they want the application to interact with (possibly Oracle or Microsoft Server SQL), provided the database interface classes are written sensibly, it should be straightforward to redeploy it using that database.

# Testing

Target – 1000 words

Towards the goal of designing and implementing valuable tests on the application, it was necessary for the team to fully understand the requirements of the system being tested. Hence, on achieving a User Acceptance Testing (UAT) methodology, evaluation techniques were applied right after requirement gathering and continued through all stages of the development process.

This section discusses verification and validation techniques as well as unit testing t and the integration of testing frameworks that were applied. These ensured Software Quality Control within the stages of the development process and effect, ensured that the applications being developed meets the functionality and specifications outlined in the user requirements.

# Methodologies and Tools for the Software Development Lifecycle

Target – 800 words

The plan for approaching the development lifecycle of this project was using a scrum workflow, which is part of the toolbox of agile methodologies within the software engineering industry. The following section details how this methodology is practically applied to this project, and why the use of the scrum approach is justified in these project circumstances.

# Concluding Remarks

Target – few words

# Appendices

## Client User Requirements Specification

# 

## Project Requirements List

1. As a 'user', I need to 'log in', so that 'I am given appropriate information from the system.'

2. As a 'user', I need 'to be restricted to my own projects/tasks' so that I cannot interfere with irrelevant projects/tasks

3. As an 'administrator', I need to make new tasks/projects, so that users can book time against them.

4. As an administrator, I need elevated system privileges, so that I can administer the system.

5. As an administrator, I need to assign users to projects, so that the users have access/can use the system

6. As an administrator, I need projects to be composed of tasks, so that tasks can be easily organised and users can be assigned more flexibly.

7. As an administrator, I need the ability to assign users to specific tasks within a project, so that I can have control over what users may book time towards.

8. As a 'user', I need to submit project\_code, task\_code, time details & comments when booking time against a task. So that my work can be accurately tracked.

9. As a user, I need to be locked to one task at a time, so that there can be no time conflicts between tasks.

10. As a 'user', I need to be able to edit/recover/reset my account details, so that I am not prevented from logging in.

11. As an 'administrator', I need to be able to delete user accounts, so that old users can be removed from the system.

## Project Assumptions List

1. An admin can create projects and tasks - a regular user (time booker) cannot.

2. A task code is a sub-code of a project code.

3. A time booker can be assigned to entire projects or specific task codes within a project.

4. Admins are also time bookers.

5. A time booker cannot book time which conflicts with another of their time bookings (booking the same time twice).

6. An administrator has administrative access to all projects and time bookings - not a subset of projects which they administrate.

7. If the application is disconnected within the mobile version, how should login authentication work without a connection to the server?

8. An administrator can add and remove user accounts.

9. What is the hosting preference for the application? Cloud-computing or a standalone server? We will assume a Heroku cloud-based solution as deployment will be quick and require few man-hours to complete. However, this could be more expensive than other cloud based solutions at scale.

## Class Diagram

../Booking%20System%20Class%20Diagram.pdf