3. Methodology

3.1 Introduction

This chapter discusses several methodologies used in the course of this project. The chapter also discusses methodologies I intend to implement in order to develop a reliable and quality software.

3.2 Volere Requirements Specification

Requirements are important! One of the many factors contributing to the failure of software development is inadequate requirements specification or lack of client ‘s input in the specification of requirements. Volere’s approach to requirements gathering intends to provide a robust means of capturing software requirements.

The first step is to download Volere’s requirements specification template from <http://www.volere.co.uk>. The template guides you in specifying the requirements of software. The first eight items determine the feasibility of a project and are as follows:

1. The Purpose of the Product
2. Client, Customer, Stakeholders
3. Users of the Product
4. Mandated Constraints
5. Naming Conventions and Definitions
6. Relevant Facts and Assumptions
7. The Scope of the Work
8. The Scope of the Product

Sufficient details needed to complete the first eight sections are to be obtained from the client. A clear understanding of these eight sections is paramount before progressing any further.

The next steps are:

* Identifying events and use cases
* Select a vital event and start gathering requirements for the event
* Design the vital business use case either by low fidelity prototyping or with any technique that animates the use case
* Gather and define atomic requirements of each product use case
* Test the atomic requirements of each business use case for completeness, measurability, consistency and relevancy

A comprehensive description of the Volere Requirements Specification has been attached as Appendix D.

3.3 Agile Software Development Lifecycle

The chosen software development lifecycle is agile. Agile software development lifecycle involves continuous iterations of requirements and solutions, including regular team meetings and interactions with the client. The client is kept up-to-date by these interactions, which ensures the initial requirements are still valid. If the initial requirements are no longer valid, changes can be incorporated immediately rather than at the end of the software development lifecycle. The ultimate goal is to produce software that achieves the desired requirements of the client.

The client prioritises the requirements and milestones are determined and marked. The next step is to undertake several series of sprints in order to achieve a milestone. These sprints are iterations of researching, planning, designing, coding, debugging, testing -- they have no sequential order.

Once a milestone is met, the feature is demonstrated to the client to get client’s feedback. These steps are repeated till all requirements are met. This type of software development promotes collaboration between teams; rapid software development; incremental software features update; continuous improvement; and it encourages a rapid response to changes in requirements. A detailed agile software lifecycle has been attached as Appendix E.

3.4 User-centered Design (UCD)

User-centered design (UCD) is a framework of activities geared towards the end users of a software product. The end users’ needs, wants and limitations are considered at each stage of the design process.

A feature of UCD is the multi-stage problem-solving processes that force designers to not only conceptualise how users will possibly use a software product, but to also test the authenticity of their assumptions of users’ interactions with actual users’ interactions of the software product in the real world. These problem-solving processes are applied from the requirements gathering, planning, designing and testing stages.

Such testing is needed because, sometimes, it is hard for designers of software products to understand intuitively how first-time users interact with their product and the challenges these users encounter..

The main difference from other design principles is that UCD designs the software product around the end users’ wants and needs rather than forcing the users to alter their behaviour in order to accommodate the software product.

3.5 Unit Testing

Unit testing is a software development process that involves the testing of individual units of an application. These individual units are tested independently of each other to ensure each unit works as expected -- method stubs and mocked objects are used in the testing of these individual units. Unit testing is often automated but it can also be done manually. Test-driven development is an example of an automated unit test.

3.6 Integration Testing

Integration testing is usually carried out after unit testing. The Individual units are combined and tested as a group: the individual units are grouped into larger modules and tested according to the tests outlined in an integration test plan.

The aim of integration testing is to verify functional and non-functional requirements are met. This can be achieved using black-box tests. In black-box testing, the internal workings of the software are unknown or not considered for testing.

Some types of integration testing methods are top-down and bottom-up integration tests. Other integration patterns are client-server integration and distributed services integration.

3.7 Acceptance Testing

In software development, acceptance testing is defined as: formal testing with respect to the client’s needs and requirements, conducted to determine whether the final product satisfies the business requirements and goals.

The client specifies scenarios that need to be met in order for the software to be accepted or deemed successful. The software development team makes use of these scenarios or user stories to generate test cases.

These test cases are written in a language that’s easily understood by the client. Acceptance tests are black-box tests. Each test expects a specific behaviour or some results from the software. Clients are responsible for verifying the software works as expected. Acceptance tests can also be used as regression tests before and after changes to the source code. An example of an acceptance test case can be found below:

*Given a user is logged in*

*When the user visits the “Products” page*

*Then the user expects to see a list of products*

3.8 Refactoring

Refactoring is the process of restructuring the source code without changing its external behaviour or functions. Refactoring improves the non-functional characteristics of the software. Benefits of refactoring include improved code readability, reduced complexity, removal of redundant source codes and improved quality of source codes. Refactoring results in source codes that are easier to maintain.

It’s important to state the importance of having integration or regression tests before refactoring so changes to the source code don’t result to changes in system behaviours or functions.

3.9 Dependency Injection

Dependency Injection is a software design pattern in which components are given their dependencies instead of hard coding them within the component. This relieves a component from locating the dependency and makes dependencies configurable. This helps in making components reusable, maintainable and testable (TutorialsPoint, *AngularJS - Dependency Injection*).