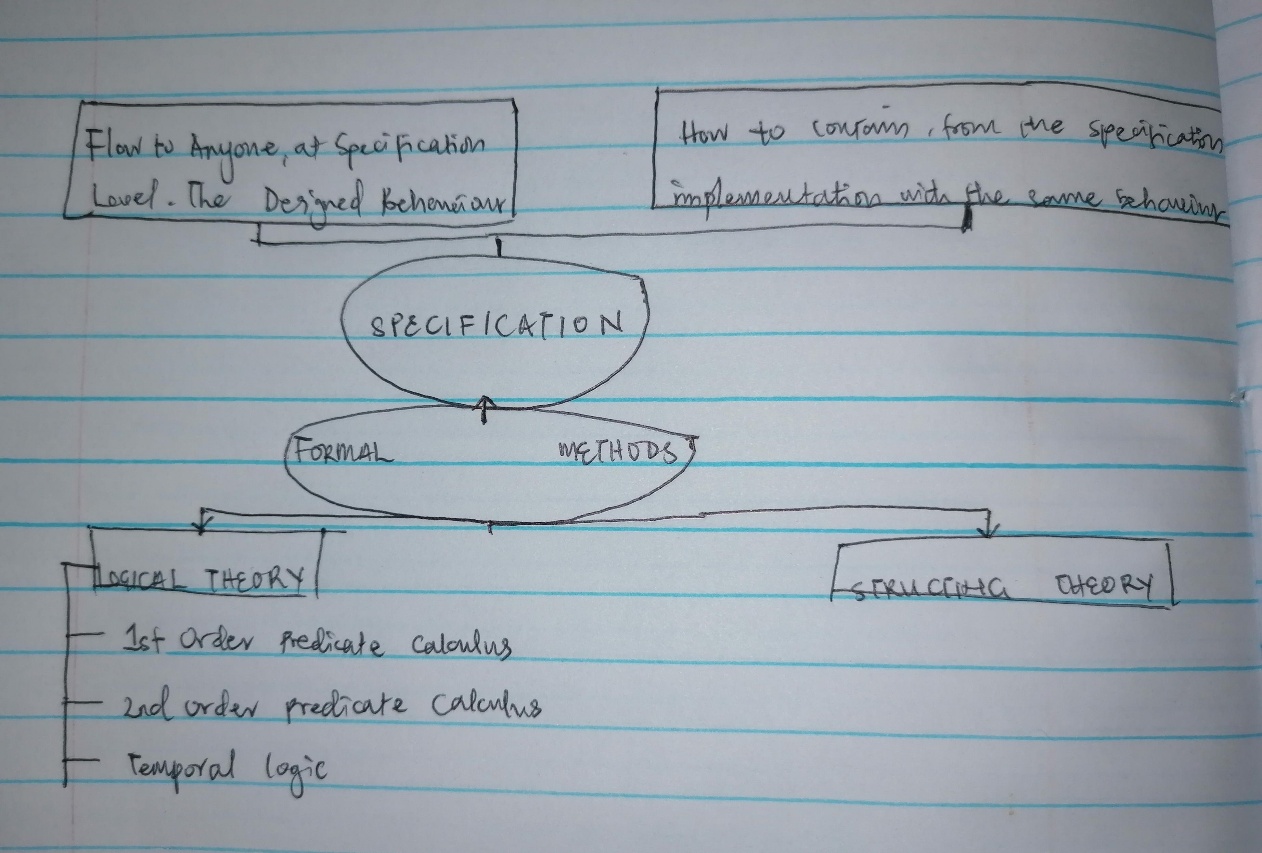
**NAME: GABRIEL-OHANU VICTOR**

**MATRIC NUMBER: 17/SCI01/035**

**COURSE CODE: CSC308**

1a)

1a (i) **Concept of Formal Methods**



ii. **REASONS FOR CONSIDERING FORMAL METHODS**

• Systems are becoming increasingly dependent on software components;

• Complexity of Systems with embedded software has increased - A modern mid-sized sedan car is equipped with at least 150 processors;

• Maintaining reliability in software-intensive systems is very difficult. Quality problems with software may cause minor irritations or major damage to a customer’s business including loss of life.

1b)

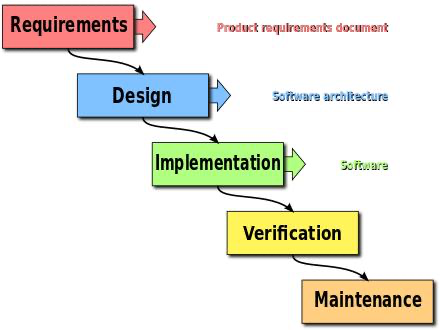
|  |  |  |
| --- | --- | --- |
|  | **Functional** | **Non-Functional** |
| **Requirement** | It is mandatory | It is Non-Mandatory |
| **Capturing** **Type** | It is captured in Use case | It is Captured as a Quality Attribute |
| **End-Result** | Product Feature | Product Properties |
| **Capturing** | Easy to Capture | Hard to Capture |
| **Objective** | Helps you to verify the functionality of the Software | Helps you to verify the Performance of the software |
| **Area** **Of** **Focus** | Focus on User Requirement | Concentrates on the User Expectation |
| **Documentation** | Describe what the product does | Describes how the product works |
| **Type** **Of** **Testing** | Functional Testing like System, Integration, End to End, API testing, etc. | Non-Functional Testing like Performance, Stress, Usability, Security testing, etc. |
| **Test** **Execution** | Test Execution is done before non-functional testing. | After the functional testing |
| **Product** **Info** | Product Features | Product Properties |

2a) Most modern development processes can be vaguely described as agile. Other methodologies include waterfall, prototyping, iterative and incremental development, spiral development, rapid application development, and extreme programming.

**I. Agile software development**

Uses iterative development as a basis but advocates a lighter and more people-centric viewpoint than traditional approaches. Agile processes fundamentally incorporate iteration and the continuous feedback that it provides to successively refine and deliver a software system. There are many agile methodologies namely: dynamic system development method(DSDM), kanban, scrum

**II. Waterfall development**



The waterfall model is a sequential development approach, in which development is seen as flowing steadily downwards (like a waterfall) through several phases, typically:

Requirements analysis resulting in a software requirements specification

Software design

Implementation

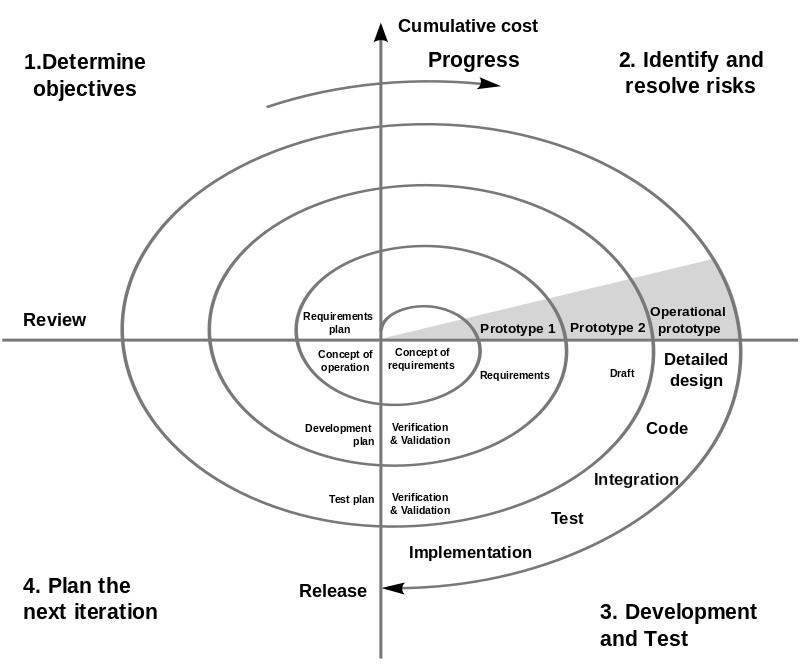
Testing

Integration, if there are multiple subsystems

Deployment (or Installation)

Maintenance

**III. Spiral development**



This is a methodology which combines some key aspect of the waterfall model and rapid prototyping methodologies, in an effort to combine advantages of top-down and bottom-up concepts. It provided emphasis in a key area many felt had been neglected by other methodologies: deliberate iterative risk analysis, particularly suited to large-scale complex systems.

**IV. Incremental funding**

Methodology is an ROI-informed approach to software development in which software is developed and delivered in carefully prioritized chunks of customer valued functionality. These chunks are known as Minimum Marketable Features (MMFs).

Other software development methodologies include

1. Prototyping
2. Behavior-driven development and business process management
3. Lightweight methodology - a general term for methods that only have a few rules and practices

And so on.

2b) Formal methods are techniques used to model complex systems as mathematical entities. By building a mathematically rigorous model of a complex system, it is possible to verify the system's properties in a more thorough fashion than empirical testing.

3(a)

i

Predicate logic is opposed to propositional logic, which simply uses symbols without the ability to do predication. For example: means "p and q" or "p and q are both true", where p and q are propositions. ... In first-order predicate logic, variables can appear only inside a predicate.

3(a)

ii

**Logical operations in propositional logic**

**A. Negation**: The simplest logical operation is negation. It is unary. The logical negation of the proposition A, is !A. The operator ! is sometimes represented by the symbol ¬, a minus sign (−), a tilde (˜), or the word "not." The negation of A is sometimes called the inverse of A. Logical negation is like a negative sign in arithmetic (a negative sign, not a minus sign, which operates on a pair of numbers. Truth table below

A B C !A !B !C

0 0 0 1 1 1

0 0 1 1 1 0

0 1 0 1 0 1

0 1 1 1 0 0

1 0 0 0 1 1

1 0 1 0 1 0

1 1 0 0 0 1

1 1 1 0 0 0

**B. Disjunction (|):** It is an operation on two propositions (a binary operation) that results in another proposition: the proposition (A | B) is true if p is true or if q is true or if both p and q are true. The operation | is sometimes represented by a vee (∨) or by the word "or." The truth table for disjunction is given below:

A B A|B

0 0 0

0 1 1

1 0 1

1 1 1

**C. Conjunction(&):** It combines two propositions to produce another. The proposition (p & q) is true if both p is true and q is true; it is false if either p is false or q is false (or both). The operation & is sometimes represented by a wedge (∧) or the word "and." Here is the truth table for &:

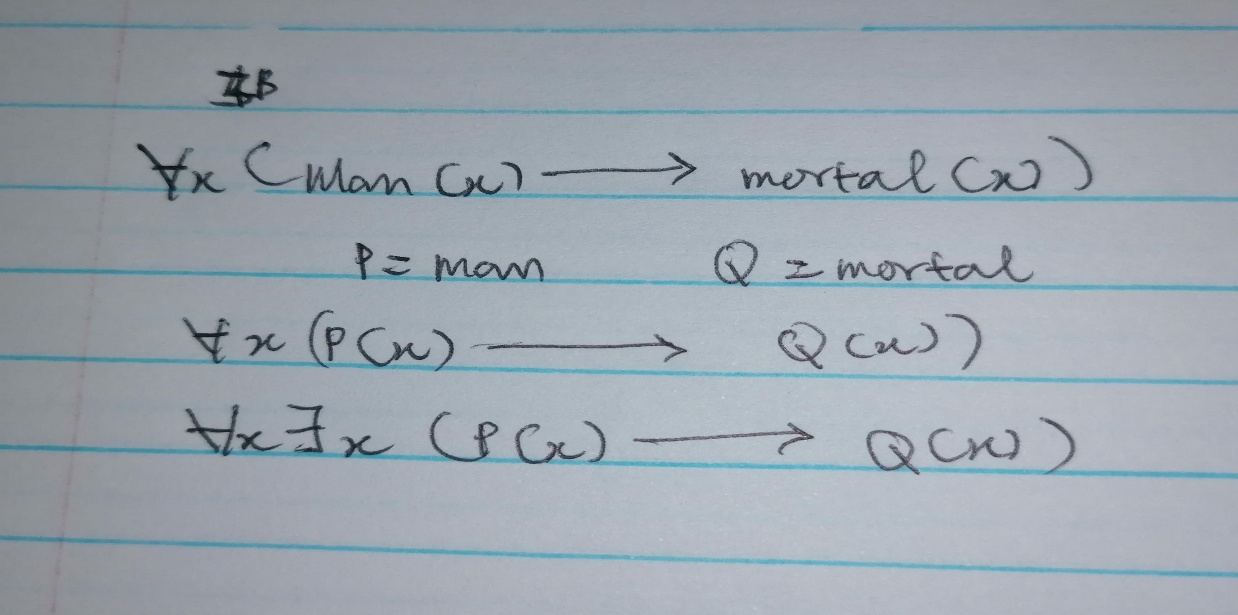
A B A&B

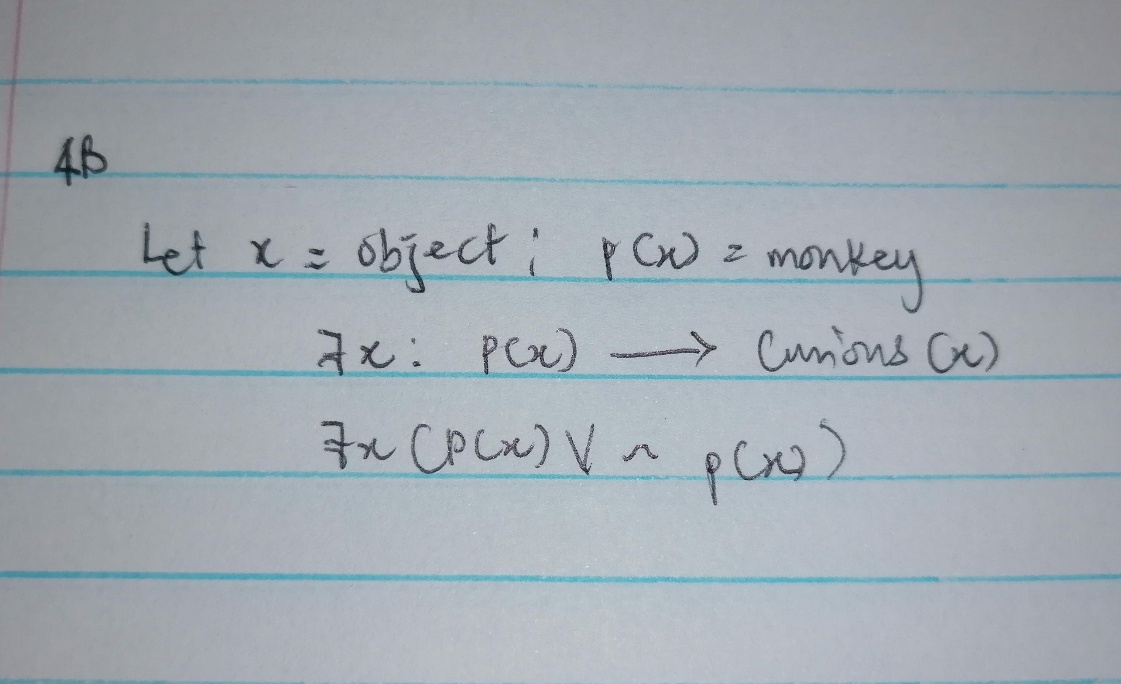
0 0 0

0 1 0

1 0 0

1 1 1





4b)

1. **History-based specification**

This behavior is based on system histories, assertions are interpreted over time.

1. **State-based specification**

This behavior is based on system states, series of sequential steps, (e.g. a financial transaction). Languages such as Z, VDM or B rely on this paradigm

1. **Transition-based specification**

This behavior is based on transitions from state-to-state of the system. It is best used with a reactive system. Languages such as Statecharts, RSML or SCR rely on this paradigm.

5a)

1. **WELL FORMED FORMULA & PREDICATE**

In mathematical logic, a predicate is commonly understood to be a Boolean-valued function P: X→ {true, false}, called a predicate on X. However, predicates have many different uses and interpretations in mathematics and logic, and their precise definition, meaning and use will vary from theory to theory. So, for example, when a theory defines the concept of a relation, then a predicate is simply the characteristic function (otherwise known as the indicator function) of a relation. However, not all theories have relations, or are founded on set theory, and so one must be careful with the proper definition and semantic interpretation of a predicate.

1. **QUANTIFIER**

A quantifier is a language element that helps in generation of a quantification, which is a construct that mentions the number of specimens in the given domain of discourse satisfying a given open formula.

Usually, the term refers to computer-generated image data created with the help of specialized graphical hardware and software. It is a vast and recently developed area of computer science.

In mathematical logic, propositional logic and predicate logic, a well-formed formula, abbreviated Well formed formula or wff, often simply formula, is a finite sequence of symbols from a given alphabet that is part of a formal language.

5bi) if "a" is greater than "b" then it is true otherwise false

6a(I)

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple examples of software design. They include:

## **Structured Design**

Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.

Structured design is mostly based on ‘divide and conquer’ strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.

The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.

These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -

**Cohesion** - grouping of all functionally related elements.

**Coupling** - communication between different modules.

A good structured design has high cohesion and low coupling arrangements.

## **Function Oriented Design**

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation.. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

## **Object Oriented Design**

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategies focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object Oriented Design:

* **Objects -**All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
* **Classes -**A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

* **Encapsulation -**In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
* **Inheritance -**OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
* **Polymorphism -**OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

6b)

A product requirements document (PRD) is a document containing all the requirements to a certain product. It is written to allow people to understand what a product should do. A PRD should, however, generally avoid anticipating or defining how the product will do it in order to later allow interface designers and engineers to use their expertise to provide the optimal solution to the requirements.

PRDs are most frequently written for software products, but can be used for any type of product and also for services.

7a)

**Structure of a Requirements Document**

Requirements have to be specified using some specification language. Though formal notations exist for specifying specific properties of the system, natural languages are now most often used for specifying requirements. When formal languages are employed, they are often used to specify particular properties or for specific parts of the system, as part of the overall SRS.

All the requirements for a system, stated using a formal notation or natural language, have to be included in a document that is clear and concise. For this, it is necessary to properly organize the requirements document.

The IEEE standards recognize the fact that different projects may require their requirements to be organized differently, that is, there is no one method that is suitable for all projects. It provides different ways of structuring the SRS. The first two sections of the SRS are the same in all of them.

**The introduction section** contains the purpose, scope, overview, etc., of the requirements document. The key aspect here is to clarify the motivation and business objectives that are driving this project, and the scope of the project.

The next section gives an overall perspective of the system—how it fits into the larger system, and an overview of all the requirements of this system.

Detailed requirements are not mentioned. Product perspective is essentially the relationship of the product to other products; defining if the product is independent or is a part of a larger product, and what the principal interfaces of the product are. A general abstract description of the functions to be performed by the product is given. Schematic diagrams showing a general view of different functions and their relationships with each other can often be useful. Similarly, typical characteristics of the eventual end user and general constraints are also specified.

If agile methods are being used, this may be sufficient for the initial requirements phase, as these approaches prefer to do the detailed requirements when the requirement is to be implemented.

**The external interface requirements section** specifies all the interfaces of the software: to people, other software, hardware, and other systems. User interfaces are clearly a very important component; they specify each human interface the system plans to have, including screen formats, contents of menus, and command structure. In hardware interfaces, the logical characteristics of each interface between the software and hardware on which the software can run are specified. Essentially, any assumptions the software is making about the hardware are listed here. In software interfaces, all other software that is needed for this software to run is specified, along with the interfaces. Communication interfaces need to be specified if the software communicates with other entities in other machines.

In **the functional requirements section,** the functional capabilities of the system are described. In this organization, the functional capabilities for all the modes of operation of the software are given. For each functional requirement, the required inputs, desired outputs, and processing requirements will have to be specified. For the inputs, the source of the inputs, the units of measure, valid ranges, accuracies, etc., have to be specified. For specifying the processing, all operations that need to be performed on the input data and any intermediate data produced should be specified. This includes validity checks on inputs, sequence of operations, responses to abnormal situations, and methods that must be used in processing to transform the inputs into corresponding outputs.

**The performance section** should specify both static and dynamic performance requirements. All factors that constrain the system design are described in the performance constraints section. **The attributes section** specifies some of the overall attributes that the system should have. Any requirement not covered under these is listed under other requirements. Design constraints specify all the constraints imposed on design

7b)

**1. Create an outline**

An outline is a plan of your document that contains key points about the product and its scope. You can use it for generating new ideas and organize them in a logical structure. Here’s a sample of an outline.

1. Introduction

1.1. Purpose

1.2. Document Conventions

1.3. Intended Audience

1.4. Product Scope

1.5. References

2. Overall Description

2.1. Product Perspective

2.2. Product Functions

2.3. User Classes and Characteristics

2.4. Operating Environment

2.5. Design and Implementation Constraints

2.6. User Documentation

2.7. Assumptions and Dependencies

3. External Interface Requirements

3.1. User Interfaces

3.2. Hardware Interfaces

3.3. Software Interfaces

3.4. Communications Interfaces

4. System Features

4.1. System Feature 1

4.2. System Feature 2 (and so on)

5. Other Non-Functional Requirements

5.1. Performance Requirements

5.2. Safety Requirements

5.3. Security Requirements

5.4. Software Quality Attributes

5.5. Business Rules

6. Other Requirements

Appendix.

**2. Define the goals**

To define the goals of your project, answer the following questions:

What is a new product developed for?

What kind of value does it bring?

Who are the major competitors? Can you make a better product?

What are the essential requirements for a future project?

You can sum up the answers and formulate the objectives. It’s good to make a list of people that will have access to the document and how they will use it. You can add BAs, project managers, coders, QAs, people from the client’s side and your sales (marketing) teams.

**Use and Audience**

During this stage of the SRS writing process, you should decide on who will be able to read and use the SRS documentation. The intended group of people may include test engineers, developers, PMs, and other stakeholders.

**Scope and definitions**

The scope should include all the benefits the stakeholders can get with the software under development. It must also describe the general business goals. Add the definitions of the terms you use throughout the document.

**3. Add details to the requirements**

There are a few types of requirements that you should add to the document. We can roughly divide them into high-level and low-level ones. Here are the high-level requirements:

Business requirements – goals, mission, problems to solve, etc.

User needs – the interaction of a user with a new system, functionalities they need.

System requirements – functional and non-functional.

Here are the essential low-level requirements:

a. User interface

b.Market specs

c.Stakeholder specs

d.Let’s take a closer look at system specifications.

**4. Get an SRS document approved**

You should show the document to the people that are participating in the development process. You can make a short presentation for the stakeholders if it’s required. You may have the need to update the document or add any other important issues.

This SRS example will help you structure your own specifications. Besides, we’ve collected useful tips for writing specs.