

SE 3XA3: Software Requirements Specification  
Title of Project

Team #, Team Name  
Student 1 name and macid  
Student 2 name and macid  
Student 3 name and macid

November 9, 2018

# Contents

## List of Tables

## List of Figures

Table 1: **Revision History**

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

# 1 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (?). We advocate a decomposition based on the principle of information hiding (?). This principle supports design for change, because the “secrets” that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules layed out by ?, as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is used in only one module.
- Any other program that requires information stored in a module’s data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (?). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers’ understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

As the world becomes increasingly more connected through the internet, many common internet users are looking for an easy way to communicate with each other or reach out to distant loved ones. The market for messengers has become saturated with products that put the goal of earning maximum revenue over the needs of the consumer. NodeM’s focus is to capture consumer interest by implementing a free and accessible web application messenger that allows them to personalize their experience and chat with other users in a simple, clean and non-intrusive way. Node messenger will become a haven for users searching for a consumer-friendly product with great functionality and cross-platform support.

The rest of the document is organized as follows. Section ?? lists the anticipated and unlikely changes of the software requirements. Section ?? summarizes the module decomposition that was constructed according to the likely changes. Section ?? specifies the connections between the software requirements and the modules. Section ?? gives a detailed description of the modules. Section ?? includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section ?? describes the use relation between modules.

## 2 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section ??, and unlikely changes are listed in Section ??.

### 2.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

**AC1:** The specific hardware on which the software is running.

**AC2:** The format of the HTML, CSS, and Javascript files.

**AC3:** The color theme of the messaging app.

**AC4:** The format of the message box.

**AC5:** The format of the Login/Register screen.

**AC6:** Domain name.

**AC7:** Addition or removal of Buttons, and their functions.

**AC8:**

...

### 2.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

**UC1:** Input/Output devices (Input: File and/or Keyboard, Output: File, Memory, and/or Screen).

**UC2:** There will always be a source of input data external to the software.

...

### 3 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table ?? . The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

**M1:** Hardware-Hiding Module

...

Level 1	Level 2
Hardware-Hiding Module	?
	?
	?
	?
	?
	?
	?
	?
Software Decision Module	?
	?
	?

Table 2: Module Hierarchy

### 4 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table ?? .

## 5 Module Decomposition

Modules are decomposed according to the principle of “information hiding” proposed by ?. The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. Also indicate if the module will be implemented specifically for the software.

Only the leaf modules in the hierarchy have to be implemented. If a dash (–) is shown, this means that the module is not a leaf and will not have to be implemented. Whether or not this module is implemented depends on the programming language selected.

### 5.1 Hardware Hiding Modules (M??)

**Secrets:** The data structure and algorithm used to implement the virtual hardware.

**Services:** Serves as a virtual hardware used by the rest of the system. This module provides the interface between the hardware and the software. So, the system can use it to display outputs or to accept inputs.

**Implemented By:** OS

### 5.2 Behaviour-Hiding Module

**Secrets:** The contents of the required behaviours.

**Services:** Includes programs that provide externally visible behaviour of the system as specified in the software requirements specification (SRS) documents. This module serves as a communication layer between the hardware-hiding module and the software decision module. The programs in this module will need to change if there are changes in the SRS.

**Implemented By:** –

#### 5.2.1 Input Format Module (M??)

**Secrets:** The format and structure of the input data.

**Services:** Converts the input data into the data structure used by the input parameters module.

**Implemented By:** [Your Program Name Here]

### 5.2.2 Etc.

## 5.3 Software Decision Module

**Secrets:** The design decision based on mathematical theorems, physical facts, or programming considerations. The secrets of this module are *not* described in the SRS.

**Services:** Includes data structure and algorithms used in the system that do not provide direct interaction with the user.

**Implemented By:** –

### 5.3.1 Etc.

## 6 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Req.	Modules
R1	M??, M??, M??, M??
R2	M??, M??
R3	M??
R4	M??, M??
R5	M??, M??, M??, M??, M??, M??
R6	M??, M??, M??, M??, M??, M??
R7	M??, M??, M??, M??, M??
R8	M??, M??, M??, M??, M??
R9	M??
R10	M??, M??, M??
R11	M??, M??, M??, M??

Table 3: Trace Between Requirements and Modules

AC	Modules
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??
AC??	M??

Table 4: Trace Between Anticipated Changes and Modules

## 7 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. ? said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure ?? illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

Figure 1: Use hierarchy among modules