SE 3XA3: Software Requirements Specification
Title of Project

Team #, Team 3
Erin Varey, Vareye
Nik Novak, Novakn
Joel Straatman, Straatjc

November 13, 2016

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 (Parnas et al., 1984). We advocate a decomposition based on the principle of information hiding (Parnas, 1972). 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 Parnas et al. (1984), 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 (Parnas et al., 1984). 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.

The rest of the document is organized as follows. Section 2 lists the anticipated and unlikely changes of the software requirements. Section 3 summarizes the module decomposition that was constructed according to the likely changes. Section 4 specifies the connections between the software requirements and the modules. Section 5 gives a detailed description of the modules. Section 6 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 7 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 2.1, and unlikely changes are listed in Section 2.2.

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.

Due to API compatability issues any code that relies on an outside API is maintained in its own module. This means particular social media API's need to be kept hidden from one another so if there is modifications that affect the original it will not affect the application as a whole. This means there will be three replacement modules:

Facebook feed replacer. Twitter feed replacer. Replacer for all other websites.

Additionally the image replacement software will need to be kept seperate due to relying on API's. The image replacer relies on instagram RS feeds to obtain images. Image recognition software for identification of memes also replies on an API and will be needed to be kept modular from the rest of the modules.

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

There is minimal hardware required for Rather to run. It requires a laptop that is able to access Google Chrome. There are minimal input devices required. Some form of text communicator is required. This could be done through a keyboard or a speech to text convertor. Additionally some sort of selction input device is required such as a mouse. These are unlikely to change drastically anytime soon and will not accounted for in the design

AC2: The format of the initial input data.

The input data is sourced from the internet and user input.

User input: The user is required to input items to filter, and a desired replacement. This is done through strings that are communicated textually.

Internet: There is a variety of input required from the webpages the user interacts with. There is the source code that will be obtained from every single webpage used. There is also input that is communicated from webpages that do not contain all the information in the source code. Additionally due to webpages being loaded dynamically, frequent reloading of the source code is required so the application will continue to interact with webpage. Some webpages have unique streamlet like information that will replace the entire webpage rather than the individual feed. The actual streamlet information will need to be uniquely obtained. This will be done for Facebook, and Twitter. These each have API's that have been previously developed and supported to give this information.

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).

The input will always be done textually. This hardware input method is unlikely to change anytime soon.

The output will be done through a screen and internet. These are pieces of hardware that are unlikely to be modified anytime soon. It is assumed there will always be a visual output for the user, and that webpages will always contain source code.

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 1. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

Due to the nature of javascript code, modules are not often used. We used modules by having seperate javascript files that performed each action. These wll be treated as modules in the tables

M1: Hardware-Hiding Module

There are no hardware hiding modules on this project. The hardware will not affect how the code operates provided that there is something that matches the input and output style.

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 3.

Level 1	Level 2
Hardware-Hiding Module	
Behaviour-Hiding Module	? Twitter-API.js Facebook-API.js Filter-Feature.js Detect-Website.js page.js popup.js content.js
Non Java-script Modules and Design modules	? manifest.json popup.html

Table 2: Module Hierarchy

5 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). 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 (M1)

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.

Level 1	**Level 2**
Module and requirements	
Functional	? Twitter-API.js —– Allow the user to block specific items in their twitter feed Facebook-API.js —– Allow the user to block specific items in their newsfeed Filter-Feature.js —– Allow the user to filter individual post they do not want censored Detect-Website.js —– Allow the user to block items on specific web feeds. (Will allow functionality for Twitter and Facebook) Image.js ——If a keyword is found images containing or tagged with it will be filtered page.js —– Main Controller for direction of information flow and program state popup.js —– Allow the user to filter content on websites that are not twitter and facebook content.js —– Allow the user to detect both content and tags related to a word popup.html —– Allow the user to input a "kill list" of words they want to remove popup.html —– Allow the user to input a "replacementl list" of words they want to remove popup.html —– Allow the user an option to mute or replace content
Non-Functional	? manifest.json —– performance requirements and file naming index file popup.html —– Allow the program to have a sleek look and feel

Table 3: The modules that satisfies each requirement is listed

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: Facebook-API.js, Twitter-API.js, Filter-Feauture.js, content.js, popup.html

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: popup.html, content.js

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: Detect-Website.js, page.js

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	popup.html
R2	popup.html
R3	popup.html
R4	Facebook-API.js, Twttier-API.js, Detect-Website.js
R5	Image.js, Content.js
R6	Image.js
R7	Image.js, Content.js, Popup.js
R8	Popup.html

Table 4: Trace Between Requirements and Modules

AC	Modules
AC??witter-API changes	Twitter-API.js
AC??acebook-API changes	Facebook-API.js
AC??nput style changes	popup.html
AC??ebpage does not load completely initially	Detect-Website.js

Table 5: Trace Between Anticipated Changes and Modules

7 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) 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 1 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.

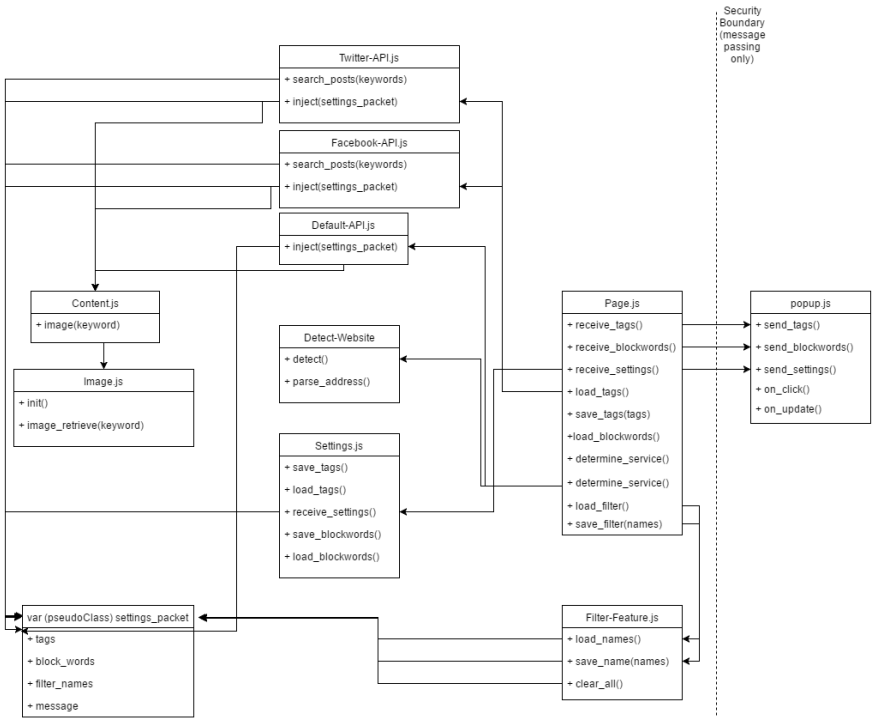

Figure 1: Use hierarchy among modules

References

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.