University of Pretoria
Software Engineering - COS 301

General Coding Standards and Guidelines

Team Singularity
24 May 2019

Team Members:

Richard McFadden	**u17026662**
Adrian le Grange	**u17056782**
Jarrod Goschen	**u17112631**
Alessio Rossi	**u14137934**
Kyle Olivier	**u15001319**

Contents

1 Introduction

The goal of this document is to guarantee that all code written for the project is consistent. This makes the code easier to read, explain to other team members and to maintain the software. Coding standards help a project run smoothly.

Well written software has many benefits and advantages; less bugs, more efficiency. Software has a life cycle, most of which is consists of maintanence, which will be much easier for the original author, team members as well as future developers responsible for maintaining the code, or modifying it. This directly boosts the productivity of the developers, and in turn decreasing the development time of the project.

This document and the examples given are general and can be applied to most programming languages (Javascript is used as examples). Note that these styles and standards leave some details up to the developer to provide more freedom. It is important that if some style is used for a file, then that style has to be consistently used throughout that file; It is easier to maintain consistent code.

2 Documentation Standards

Proper and well documented software modules are easier to read and understand. The following standards are mandatory.

2.1 File Headers

Every file containing source code must have a header. The header contains all maintenance information for that file. Note that none of the headers contains a history as this is tracked by the project git repository.

2.1.1 Format of header for file containing no class declaration:

```
/**
* File Name    : someFile.someExtention
* Version      : V1.0
* Author       : authorName authorSurname
* Project Name : A-Recognition (Advance)
* Organisation : Singularity
* Functional Description: A description of what the code does or what it
*                         implements.
**/
```

2.1.2 Format of header containing a class definition

```
/**
* File Name    : someFile.someExtention
* Version      : V1.0
* Author       : authorName authorSurname
* Project Name : A-Recognition (Advance)
* Organisation : Singularity
* Class Name   : SomeClass
* Related Requirements: REQ1, REQ3 (All requirements that this class
                        implements)
* Functional Description: A brief description of what the class does.
**/
```

2.2 Descriptions of classes and members

Each class must have a comment block directly above it. This block will contain all neccesary documentation for that class as a whole. In addition, each member function definition should have a comment block above it, providing a short description of what the function's purpose is as well as details about it's parameters and output values. Lastly all member variables should have a corresponding single line description above it. Example of a class with correct headers:

```
/**
* The Rectangle class is used to represent a specific rectangle.
* The contructor creates a rectangle with the provided width and length.
* (Some detailed description of the class.)
**/
class Rectangle
{
   constructor(height, width)
   {
      ///The height of the rectangle
      this.height = height;

      ///The width of the rectangle
      this.width = width;
   }

   /**
   * A member function that sets the the length and width of the
       rectangle
   * should the given width be greater than the given height and greater
       than 0
   * @param w an integer argument representing the width
   * @param h an integer argument representing the height
   * @return Whether the values was set.
   */
   res setAttribs(w, h)
   {
     if((w > h) && (w > 0))
     {
        this.width = w;
        this.height = h;
        return true;
     }
     else
     {
        return false;
     }
   }
}}
```

3 Coding Standards

These standards are obligatory and all written code must adhere to these standards. The standards are necessary to increase reusablity, improve maintainability and provide clarity to the purpose of each unit of code.

3.1 Project File Structure

All source code is saved to a single folder. This folder is the root of the project and contains the makefile and git repository. Inside this folder, every subsystem has their respective folder. All code related to a subsystem must be kept within it's folder. Other folders may be added as neccesary inside the subsystem folders.

3.2 Indentation

Proper indentation makes code easy to read and maintainable. Tabs are allowed for indentation or a minimum of three spaces. Please note that if for example four spaces was used for indentation then that should be used throughout the document. What should be indented:

- Loop bodies
- Conditional bodies
- Function bodies

Example:

```
//Indentation for body of function
function findByName(searchTerm)
{
   //Indentation for body of loop
   for(index = 0; index < length; index++)
   {
      //Indentation for conditional body
      if(members[index].name == searchTerm)
      {
         return members[index];
      }
   }

   alert( Person        + searchTerm +      was not found! );
   return null;
}
```

3.3 Comments

Block and inline comments must be used as neccesary to explain the flow of code. Care should be taken to avoid writing comments that is obvious or states what a single line of code does. Comments should explain the idea of the code and not each line. For example:

3.3.1 Good comments

```
function findByName(searchTerm)
{
   //Linear search for user with given name
   for(index = 0; index < length; index++)
   {
      if(members[index].name == searchTerm)
      {
         //Return the matched member
      return members[index];
      }
   }

   //Member was not found
   alert( Person        + searchTerm +     was not found! );
   return null;
}
```

3.3.2 Bad comments

```
function findByName(searchTerm)
{
   //Loop through members array
   for(index = 0; index < length; index++)
   {
      //Check if the member at current index matches the searched name
   if(members[index].name == searchTerm)
      {
         //Return the member
      return members[index];
      }
   }

   //Show message that member was not found
   alert( Person        + searchTerm +     was not found! );

   //Return nothing as member was not found
   return null;
}
```

3.4 Naming

A good naming scheme helps everyone looking at code to easier understand what is happening. To improve readability and make code more clear, camel-case must be used for all names.

3.4.1 Classes

All class names should start with an uppercase letter. Class names should be nouns, such as "Button" or "Thread". Avoid using acronyms, for example don't use "DBController", instead use "databaseController".

3.4.2 Functions

All function names should start with a lowercase letter. The function name should be a verb, e.g. getResult(), print().

3.4.3 Variables

Variables should always start with a lowercase letter. Any name given to a variable should reflect it's purpose, even to somebody that has never seen the code before. One letter variables should be avoided.

3.4.4 Constants

All constants must be in all uppercase letters. Underscores are used between words if a constant contains multiple words. Examples: 'PI', 'MAX_ACTIVE_SESSIONS'.

3.4.5 Source Files

A source file's name should indicate it's function. All code in a file, such as a class and functions should have a common purpose relating to the filename.

3.5 Braces

Braces should always be used as follows:

```
function findByName(searchTerm)
{
   for(index = 0; index < length; index++)
   {
      if(members[index].name == searchTerm)
      {
         return members[index];
      }
   }

   alert( Person        + searchTerm +      was not found! );
   return null;
}
```

The only exception being callback functions, in that case any style is acceptable, as long as the style is consitent throughout the document.

Braces shall be used even when optional such as when there is only one statement in a control block, for example:

3.5.1 Good

```
if(members[index].name == searchTerm)
{
   return members[index];
}
```

3.5.2 Bad

```
if(members[index].name == searchTerm)
   return members[index];
```

4 Coding guidelines

These are general good practices that should be incorporated into every team member's own code styles and practices.

4.1 Code lines

- A line of code should be kept to 80 characters or less. If a line has to be split, it must be split after an operator or comma.
- If an expression is split, allign the split parts to the start of the expression, for example:

```
attendanceRatio = numberOfPeoplePresent + numberOfUnexpectedPeople
                  / numberOfPeopleBooked;
```

4.2 Use of spaces in statements

- A space should be used before and after operators (Except for unary operators, where no space should be used).
- A keyword next to parenthesis should be seperated by a space.

4.3 Compiler Warnings

Compiler warnings should be regarded as errors and fixed before moving on. These might not be a problem when they first occur but later in the software lifecycle they can cause some unpredicatable bug.

4.4 Error messages

All error messages done as part of error handling should have a meaningful description as well as include the file or function name where the error has occurred.

4.5 Size of functions

Functions should not be excessively long. This could indicate that the function has more than one responsibility which is a problem. If some code is repeated somewhere it could also indicate that it should be contained in a function. A good size for functions is more or less 200 lines.