Coding Standards

Golf Course Mapper

Team Recursive Recursion

August 12, 2018

Contents

1 Introduction

1.1 Purpose

This *Coding Standards Document* is intended to be a guide on the policies, standards and practices that are followed when working on the *Golf Course Mapper* project. The remainder of this section describes the different repositories used within the project and the remainder of this document then continues to describe the coding standards applied to each individual repository.

1.2 Repositories Structure

There are four (4) repositories that are used for the project: `web-app`, `mobile-app`, `mapper-api` and `documentation`. A short description of the contents of each repository is listed below.

- `web-app`

  Contains web application subsystem used for mapping and managing golf courses. The web application is built using *Angular 6*.

- `mobile-app`

  Contains the *Android* app used for viewing golf courses. The mobile app is built using native Java code.

- `mapper-api`

  Contains the *Entity Framework Core* API used to manage and provide access to the database. The database used is *PostGIS*, an extention of *PostgreSQL*.

- `documentation`

  Contains all documentation relating to the project, including this file. Documents are written in LaTeX. The repository also contains all additional images and published PDF versions of the documents.

2 Global Standards

The following standards are applied to all files across all repositories, with the exclusion of automatically generated files.

2.1 File Format

- Files are encoded using the **UTF-8** character set.
- Lines should not be longer than **80 columns**.
- **Soft tabs** expanded to **4 spaces** should be used.
- Each level of indentation uses **1 tab**.
- Line continuation indentation uses **2 tabs**.
- Every file contains a **file header**.

2.2 File Header Layout

Headers are always on the first line of a file and are placed in comments. The following information must always be present where applicable:

- Name of the file
- Original author of the file
- Name of the class(es) contained within the file
- Short description of the file

The file header layout is illustrated below. The `(start)` and `(end)` tags indicate the block-comment start and end symbols. As these are language specific, they are described in the sections dealing with per-language standards.

```
(start)
    Filename: File.ext
    Author  : John Doe
    Class   : SampleClass

        The SampleClass contains many different sample
        methods.
(end)
```

3 Web App Standards

This section describes the coding standards applied to the `web-app` repository. The system design of the subsystem is illustrated in Section 3.1 and the file structure of the repository is explained in Section 3.2. The web application makes use of TypeScript, HTML and CSS. The standards for these languages are described in Sections 3.3 and 3.4.

3.1 System Design

UML class diagram coming soon...

3.2 File Structure

The repository contains a `WebApplication/` folder at the root which contains the source files of the subsystem. This folder was created through the Angular CLI, and are therefore organized according to the standard Angular layout [??].

The most important folder is the `src/` folder, which contains the following notable locations:

- `app/`

  This folder contains all of the source code of the web application. The most important files here are `app.component.ts` and `app.module.ts` which describe the root Angular components.

- `app/components/`

  This folder contains different subfolders. Each subfolder represents one of the components as represented in Section 3.1. Each subfolder contains the `.html` and `.ts` files associated with each component.

- `app/interfaces/`

  This folder contains the `.ts` files that describe the enums and interfaces used in the web app.

- `app/services/`

  This folder contains different subfolders. Each subfolder represents one of the services as represented in Section 3.1. Each subfolder contains the `.ts` files associated with each service.

- `assets/`

  This folder contains all the assets such as background images and icons used within the web app.

3.3 Typescript Standards

3.3.1 Naming Conventions

- **Variables** are named using camel casing. Descriptive names should be used with the exception of counters in loops.
- **Classes** start with a *capital* letter and use camel casing.
- **Components** must have the word `Component` as the last word in the class name. Similarly, **Services** must have `Service` as the last word.
- **Functions** are also named similar to regular variables and should be descriptive.

```
export class SampleComponent {

    sampleVariable : string;

    public myFunction() : void {
        ...
    }
}
```

3.3.2 Style

Braces will be styled in the following manner:

- Opening braces are placed on the same line as the header.
- Closing braces are placed on a separate line at the same indentation level as the header.
- Else clauses are placed on the same line as the closing brace.
- While clauses of a do-while are placed on the same line as the closing brace.
- Braces are never left out for one-line loops or conditions.

```
if (condition) {
    statement;
} else {
    statement;
}

while (condition) {
    statement;
}
```

```
do {
    statement;
} while (condition);
```

Continuation lines should end on the operator as to indicate that the line is not complete and has a continuation.

```
var result = example + of + (a * very) /
        long - equation;
```

3.3.3 Comments

File headers are structured according to section 2.2 and are styled in the following way:

```
/***
 * Filename: sample.component.ts
 * Author  : John Doe
 * Class   : SampleComponent
 *
 *     The SampleComponent contains many different sample
 *     methods.
 ***/
```

Function headers are provided for every function and take the following form (the description can be omitted in the case of simple functions, such as mutators & accessors).

```
/***
 * function(ParType1, ParType2) : ReturnType
 *
 *     Description of the function.
 ***/
function(p1 : ParType1, p2 : ParType2) : ReturnType {
    ...
}
```

Inline comments should be kept to a minimum, since code should be self-documenting. Only use inline comments in the case of code that may be difficult to understand.

3.4 HTML & CSS Standards

3.4.1 Style

CSS files should be styled in the following way:

```
selectors {
    some-attribute: style;
```

```
    another-attribute: style;
}

more {
    some-attribute: style;
}
```

HTML files should be styled according to the following rules:

- Opening and closing tags of **block** elements should be kept on their own lines, with the content indented.
- Opening and closing tags of **inline** elements should be kept on the same line, with the content between the tags.

```
<div>
    <p>
        Here is some <span class="red">red</span> text!
    </p>
</div>
```

3.4.2 Comments

File headers are structured according to section 2.2 and are styled for CSS and HTML, respectively, in the following ways:

```
/***
 * Filename: style.css
 * Author  : John Doe
 *
 *     The styling for some example page is contained
 *     here and applies a material style.
 ***/
```

```
<!--
    Filename: page.html
    Author  : John Doe

        The page displays some content.
-->
```

4 Mobile App Standards

This section describes the coding standards applied to the `mobile-app` repository. The system design of the subsystem is illustrated in Section 4.1 and the file structure of the repository is explained in Section 4.2. The mobile application makes use of Java. The standards for the Java language are described in Section 4.3.

4.1 System Design

UML class diagram to come here...

4.2 File Structure

The root of the repository contains the *Android Studio* project folder and is therefore organized according to the standard *Android Studio* layout [??].

4.3 Java Standards

4.3.1 Naming Conventions

- **Variables** are named using camel casing. Descriptive names should be used with the exception of counters in loops.
- **Member variables** are named similarly to regular variables with the addition of an *underscore* prefix.
- **Classes** start with a *capital* letter and use camel casing.
- **Functions** are also named similar to regular variables and should be descriptive.

```
class SampleClass {

    int _someMember;

    public void myFunction() {
        int someInteger;
    }
}
```

4.3.2 Style

Braces will be styled in the following manner:

- Opening braces are placed on the same line as the header.

- Closing braces are placed on a separate line at the same indentation level as the header.
- Else clauses are placed on the same line as the closing brace.
- While clauses of a do-while are placed on the same line as the closing brace.
- Braces are never left out for one-line loops or conditions.

```
if (condition) {
    statement;
} else {
    statement;
}

while (condition) {
    statement;
}

do {
    statement;
} while (condition);
```

Continuation lines should end on the operator as to indicate that the line is not complete and has a continuation.

```
double result = example + of + (a * very) /
        long - equation;
```

4.3.3 Comments

File headers are structured according to section 2.2 and are styled in the following way:

```
/***
 * Filename: SampleClass.java
 * Author  : John Doe
 * Class   : SampleClass
 *
 *      The SampleClass contains many different sample
 *      methods.
 ***/
```

Function headers are provided for every function and take the following form (the description can be omitted in the case of simple functions, such as mutators & accessors).

```
/***
 * function(ParType1, ParType2) : ReturnType
 *
 *      Description of the function.
 ***/
ReturnType function(ParType1 p1, ParType2 p2) {
    ...
}
```

Inline comments should be kept to a minimum, since code should be self-documenting. Only use inline comments in the case of code that may be difficult to understand.

5 Mapper API Standards

This section describes the coding standards applied to the `mapper-api` repository. The system design of the subsystem is illustrated in Section 5.1 and the file structure of the repository is explained in Section 5.2. The API makes use of C#. The standards for the C# language are described in Section 5.3.

5.1 System Design

UML class diagram to come here...

5.2 File Structure

The root of the repository contains the `Test` and `MapperApi` folders. The `Test` folder contains all the Unit Test files as described further in the *Testing Policy Document* [??]. The `MapperApi` folder is the *Entity Framework Core* project folder and is therefore organized according to the standard *Entity Framework Core* file layout [??].

5.3 C# Standards

5.3.1 Naming Conventions

- **Variables** are named using camel casing. Descriptive names should be used with the exception of counters in loops.
- **Member variables** are named similarly to regular variables with the addition of an *underscore* prefix.
- **Classes** start with a *capital* letter and use camel casing.
- **Functions** are also named similar to regular variables and should be descriptive.

```
class SampleClass {
    int _someMember;

    public void myFunction() {
        int someInteger;
    }
}
```

5.3.2 Style

Braces will be styled in the following manner:

- Opening braces are placed on the same line as the header.

- Closing braces are placed on a separate line at the same indentation level as the header.
- Else clauses are placed on the same line as the closing brace.
- While clauses of a do-while are placed on the same line as the closing brace.
- Braces are never left out for one-line loops or conditions.

```
if (condition) {
    statement;
} else {
    statement;
}

while (condition) {
    statement;
}

do {
    statement;
} while (condition);
```

Continuation lines should end on the operator as to indicate that the line is not complete and has a continuation.

```
double result = example + of + (a * very) /
        long - equation;
```

5.3.3 Comments

File headers are structured according to section 2.2 and are styled in the following way:

```
/***
 * Filename: SampleClass.cs
 * Author  : John Doe
 * Class   : SampleClass
 *
 *      The SampleClass contains many different sample
 *      methods.
 ***/
```

Function headers are provided for every function and take the following form (the description can be omitted in the case of simple functions, such as mutators & accessors).

```
/***
 * function(ParType1, ParType2) : ReturnType
 *
 *      Description of the function.
 ***/
ReturnType function(ParType1 p1, ParType2 p2) {
    ...
}
```

Inline comments should be kept to a minimum, since code should be self-documenting. Only use inline comments in the case of code that may be difficult to understand.

6 Documentation Standards

This section describes the standards of the `documentation` repository. As there are no strict standards for the layout of the LaTeXfiles, only the file structure of the repository is described in Section 6.1.

6.1 File Structure

The repository contains a folder for each of the four documents, namely `coding-standards/`, `requirements/`, `testing-policy/` and `user-manual/`. Each folder contains the `.tex` source file of the document as well as any additional files required for the document, such as images.

The `publish` folder contains publish-ready versions of the documents in *PDF* format.

Finaly, the `other` folder contains extra documents that do not form part of the main documents, but are used elsewhere such as on the landing page.

7 Code Review

This section describes how and when code is reviewed. It further describes who is responsible for reviewing code.

7.1 Review Process

Before a pull request to the `master-dev` branch is accepted, the reviewer responsible for that repository must review the code. Code is only reviewed before mergin to `master-dev`, as that will ensure that `master` is also always up to standard.

The code reviewer will review the code through inspection by looking at the changes proposed in the pull request. If the code complies to the repository's standards as described in this document and the build succeeds, the pull request is accepted.

If the code does not comply to the repository's standards, the person responsible for the code is notified and must ammend the pull request to change the code. Once the code complies to the standards, the pull request is then accepted.

7.2 Responsible Party

A single person is assigned to a specific repository during the weekly scrum meetings. The person is assigned for a period of two weeks after which a new person (or possibly the same person again) is assigned to the repository. This person is solely responsible for performing the code review process (as described in Section 7.1) on the assigned repository.