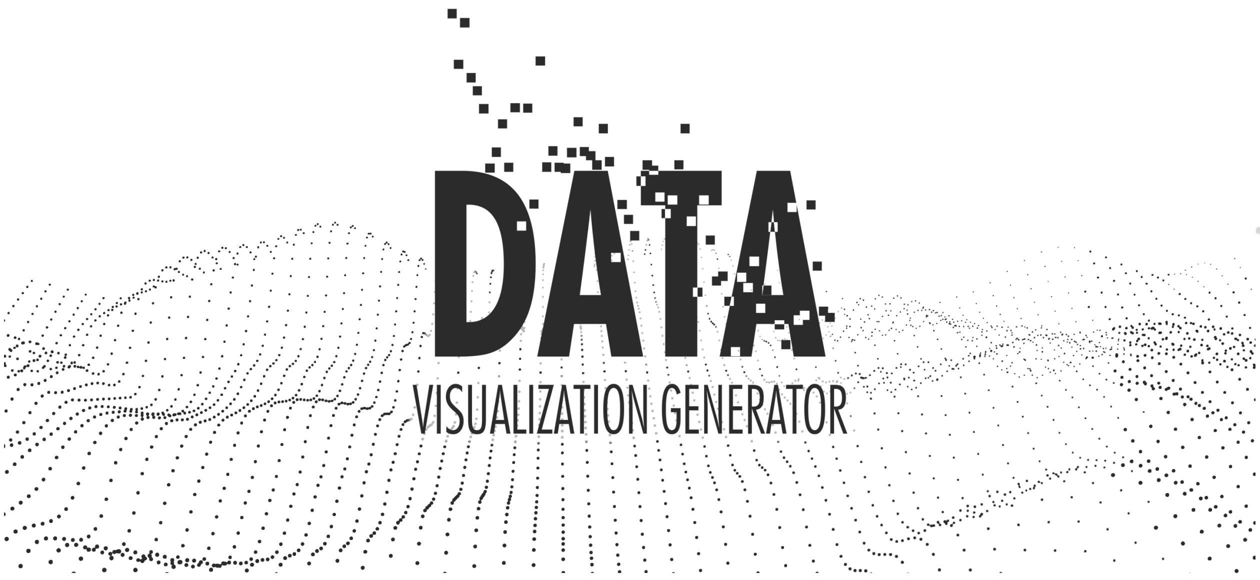
# Data Visualization Generator

Coding Standards Document

Doofenshmirtz Evil Inc

COS 301 - 2020

Marco Lombaard u18026975
Elna Pistorius u18010319
Phillip Schulze u18171185
Byron Tomkinson u18042717
Gian Uys u18052569



## Contents

## 1 Introduction

Coding Standards are rules that serve as requirements and guidelines for writing programs within an organisation or for a project. This document defines Coding Conventions and Requirements while designing and implementing the Data Visualisation project.
Documentation will occur as headers in files as multiline comments, the symbols for which are language dependent. For example, documentation in JavaScript will occur as:
/**
* Some documentation lines here
*/
Note that '/**' is used even if '/*' is sufficient. This is to keep our comments Doxygen-compliant.

## 2 File Headers

As mentioned earlier, File Headers will be documented as multiline comments and will contain the following information:

| **Item** | **Description** |
|---|---|
| File Name | The name of the file. |
| Project name | Name of the project for which the file was written. |
| Copyright | Information on copyright. |
| Organisation | Name of the organisation. |
| List of Modules | List of modules contained in the file. |
| Related Documents | A list of related documents. |
| Update History | A record of updates, each specifying the date, author and what was updated. |
| Test Cases | A list of test scripts along with their paths. |
| Functional Description | A description of what the program does and its interactions. |
| Error Messages | A list of possible error messages generated by the program along with a description of each. |
| Assumptions | A list of conditions that must be satisfied or that may affect the operation of the program. |
| Constraints | A list of restrictions on the use of the program, as well as input, environment, and other variables. |

## 3 Description of Classes

### 3.1 Class Description

A description of each class will be added as a multi-line comment before the declaration of the class.

| **Item** | **Description** |
|---|---|
| Purpose | The purpose/function of the class. |
| Usage Instructions | Instructions on how the class would be used. |
| Author | The original author of the class. |

### 3.2 Method Description

Additionally, before each method declaration, a short description on the method should be given.

| Item | Description |
|---|---|
| Purpose | The purpose/function of the method. This should be simple text and not have a heading. |
| Parameters | Description of the parameters accepted by the method, listed seperately and specified by @param. |
| Return values | Description of the value(s) that will be returned by the method, listed seperately and specified by @return. |
| Other Input/Output | Description of other input/output such as files or database tables that may be affected by this method. |

### 3.3 Component Description

REACT Components need a combination of both Class- and Method Descriptions, and will have the following in their headers:

| Item | Description |
|---|---|
| Purpose | The purpose/function of the component. This should be simple text and not have a heading. |
| Parameters | Description of the parameters accepted by the component, listed seperately and specified by @param. |
| Return values | Description of the value(s) that will be returned by the component, listed seperately and specified by @return. |
| Author | The original author of the class, specified by @author. |

## 4 Naming Conventions

### 4.1 Folders

Folders are named using camel casing with the exception being folder which contain a React component definition. In which case starts with a capital letter.

The following shows the folder structure of the complete React application:

```
├── public
└── src
    ├── assets
    ├── components
    │   ├── App
    │   ├── Dashboard
    │   └── Panel
    ├── globals
    ├── helpers
    └── tests
```

The `components` folder only contains React components. Therefore, each folder within that folder has a capital letter.

### 4.2 Files

All files that exist within the project can only be of the following type:

- Source code file.
  Any source code files can either consist of a React component class or helper functions. In the case of it being a React component, pascal case is used. Otherwise, camel case is used.
- Non-source code related file.
  These files are considered as *assets* files and are named using only lowercase letters with words separated by a hyphen character.

### 4.3 Functions Parameters

Functions and parameters uses camel casing. The only exception being functions that define React components which should start with a capital letter.

### 4.4 Constants

All constants are written with uppercase characters with words separated with an underscore character.

### 4.5 CSS

All CSS class and id names throughout the project uses the Block, Element, Modifier (BEM) naming convention. In which the block specifies the encapsulated outermost element of the target entity that is being styled and is placed first. The element of the name refers to the child of the block, in which case, the block and element parts of the name is separated with double underscore characters. For example, `button__confirm`, wherein `button` being the block and `confirm` the element part of the name. If the element is a standalone entity, a double underscore prefix is not required. The modifier part specifies the appearance, behaviour or state of the entity. In between the block or element and modifier, double hyphens is used. In the case of the previous example with the addition of the modifier `color`, `button__confirm--color`. The block, element and modifier's individual parts are written using a camel casing.

### 4.6 SASS Variables

All variables follow the general model below:

```
$component-element_state__property__modifier
```

If the individual parts require more than one word, camel casing is used.

## 5 Formatting Conventions

Formatting conventions specify formatting rules used to arrange program statements. The following formatting conventions should be adhered to, in order to keep the project neat and legible:

| Item | Description |
|---|---|
| Line Break | A line break should be used after every function for clear separation between functions.<br>A line break should be used after a group of declarations of new variables.<br>Code within condition statements/loops should start on the line after the condition. |
| Indentation | Tabs should be used for indentation. Coding blocks within other coding blocks have one more tab than the coding block they occur within. |
| Alignment | The '{' starting a coding block should start on the same line as the start of the coding block, and the '}' should be aligned with the character that started the coding block.<br>If the body of an if-statement consists of only one line, then the entire if-statement should occur on the same line |
| Spacing | A space should occur before and after every sign used in an operation or condition, i.e. '+', '=', etc.<br>A space should occur before and after closing and opening braces( '(', '{', '[', etc. ), excluding nested braces and empty braces. |

## 6 In-code Comment Conventions

In-code comments facilitate program understanding and maintenance. In-code comments that occur close together should be aligned to appear neater. In-code comments that become long enough to require scrolling should wrap around into a new line. This new line should be aligned with the start of the comment.

## 7 Example

```
/**
*   @file DashboardButton.js
*   Project: Data Visualisation Generator
*   Copyright: Open Source
*   Organisation: Doofenshmirtz Evil Incorporated
*   Modules: App.js
*   Related Documents: SRS Document - www.example.com
*
*   Update History:
*   Date       Author            Changes
*   -------------------------------------------------------
*   13/4/2020  Phillip Schulze   Original
*   16/4/2020  Byron Tomkinson   Added display Function
*
*   Test Cases: data-visualisation-app/tests/App/App.js
*
*   Functional Description: This file implements the frontend of the app. It generates the
*                           display by calling the construction of the component objects that
*                           need to be displayed.
*
*   Error Messages: "Error"
*   Assumptions: None
*   Constraints: None
*/

import React from 'react';
import './DashboardButton.css';

/**
*   This class is a React component and is used for rendering a button, representing a
*   dashboard, to the screen.
*   @param colour a hexadecimal string indicating the background colour of the component.
*   @param action a callback function that is called when the component is clicked.
*   @param isAddButton a boolean indicating whether the component adds a new dashboard.
*   @param panel a object that contains the name and description of the dashboard.
*   @return a React Component
*   @author Gian Uys
*/
function DashboardButton( { colour, action, isAddButton, id, panel } ) {
    const sizeStyles = { width: 300, height: 300 };

    /**
    *   creates an object that specifies the positional style of the component.
    *   @returns a style object.
    */
    function getContentPositionStyle() {
        return { marginTop: sizeStyles.height / 5 - 10 };
    }

    /**
    *   creates an object that specifies the size style of the component.
    *   @returns a style object.
    */
    function getSizeStyle() {
        return {
            width: sizeStyles.width + 'px',
            height: sizeStyles.height + 'px'
        };
    }

    /**
    *   Determines the structure of the component based on whether it adds a dashboard or not.
    *   @returns a React component.
    */
    function comp() {
        if (isAddButton) {
            return (
                <div className='panelLayout panelStyling' style={{...getSizeStyle(),
                backgroundColor : colour}} onClick={() => action()}>
                    <div onClick={() => action()}>
                    <div style={getContentPositionStyle()}>+</div>
                </div>
            </div>
            );
        } else {
            return (
            <div className='panelLayout panelStyling' style={{ ...getSizeStyle(),
            backgroundColor : colour }}
                onClick={() => action( id )}>
                <div>
                    <div style={ getContentPositionStyle() }>{ panel.name }</div>
                    <div>{ panel.description }</div>
                </div>
            </div>
            );
        }
    }

    return <div>{ comp() }</div>;
}

export default DashboardButton;
```