Introduction

This is Group's number 16, Second Semester Project Documentation

Project Name: Heatington

Installation

following the official documentation

> Documentation Link

To install using .NET SDK (it has to be installed previously)

```
dotnet tool update -g docfx
```

How does it work?

DocFx automatically creates documentation based on the contents of a file. This Documentation is available in the `API` tab on server.

If you want to add your own, more detailed documentation. Simply add a Markdown (`.md`) file to `/docs` directory in the right folder coresponding to the real location of a class in a project.

It's available in the `Docs` tab.

How to run it?

Run in the same directory as `docfx.json`

```
docfx docfx.json --serve
```

Now you can preview the website on http://localhost:8080.

If you want to rebuild your documentation run this command in a *new* terminal window

```
docfx docfx.json
```

How to add Markdown files?

in `toc.yml` file inside your directory add a desired markdown file, then add this to the YAML file

```
- name: <Name of the component>
- href: <NameOfTheFile>.md
```

How to add new directories?

in `toc.yml` file inside your directory add a new directory then add this to the YAML file

```
- name: <Name of the directory>
- href: <PathToTheDirectory>/
```

REMEMBER to add the `toc.yml` file later inside of the new directory, pointing onto the `.md` files

Useful Resources

DocFx runs their own (of course) version of Markdown

> Brief Introduction to a DocFx (15min Video)

> Documentation about docfx markdown

SP2 Coding Guidelines

Semester Project: Development of Software Systems, (F24) - Group 16

March 2024

Run code formatter

```
dotnet format
```

Coding style

This document defines the coding style used throughout the project. The coding style is essentialy the one outlined by Microsoft for C# projects It is a slightly modified version of the following document.

1. We use *Allman* style braces, where each brace begins on a new line. A single line statement block can go without braces but the block must be properly indented on its own line and must not be nested in other statement blocks that use braces (See rule 18 for more details). One exception is that a `using` statement is permitted to be nested within another `using` statement by starting on the following line at the same indentation level, even if the nested `using` contains a controlled block.

2. We use **four spaces** of indentation (no tabs).

3. We use `_camelCase` for `internal` and `private` fields and use `readonly` where possible. Prefix `internal` and `private` instance fields with `_`, `static` fields with `s_` and thread `static` fields with `t_` . When used on `static` fields, `readonly` should come after `static` (e.g. `static readonly` not `readonly static`). **Public** fields should be used sparingly and should use **PascalCasing** with no prefix when used.

4. We avoid `this.` unless absolutely necessary.

5. We always specify the visibility, even if it's the default (e.g. `private string foo` not `string foo`). Visibility should be the first modifier (e.g. `public abstract` not `abstract public`).

6. Namespace imports should be specified at the *top* of the file, *outside* of `namespace` declarations, and should be sorted alphabetically, with the exception of `System.*` namespaces, which are to be placed on top of all others.

7. Avoid more than one empty line at any time. For example, **do not have two** blank lines between members of a type.

8. Avoid spurious free spaces. For example avoid `if (someVar = 0). . . =`, where the dots mark the spurious free spaces.

9. If a file happens to differ in style from these guidelines (e.g. `private` members are named `m_member` rather than member), the existing style in that file takes precedence.

10. We only use `var` when the type is **explicitly** named on the right-hand side, typically due to either new or an explicit cast, e.g. `var stream = new FileStream(...)` not `var stream = OpenStandardInput()`.

11. We use language keywords instead of BCL types (e.g. `int`, `string`, `float` instead of `Int32`, `String`, `Single`, etc) for both type references as well as method calls (e.g. `int.Parse` instead of `Int32.Parse`).

12. We use **PascalCasing** to name all our constant local variables and fields. *The only exception* is for interop code where the constant value should exactly match the name and value of the code you are calling via interop.

13. We use **PascalCasing** for all method names, including local functions.

14. We use `nameof(...)` instead of `"..."` whenever possible and relevant.

15. Fields should be specified at the top within type declarations.

16. When including non-ASCII characters in the source code use **Unicode** escape sequences instead of literal characters. Literal non-ASCII characters occasionally get garbled by a tool or editor.

17. When using labels (for goto), indent the label *one less* than the current indentation. 1

18. When using a single-statement `if`, we follow these conventions

    - Never use single-line form (for example: `if (source = null) throw new ArgumentNullException(‘‘source’’);`
    - Using braces is always accepted, and required `if` any block of an `if=/=else if=/.../=else` compound statement uses braces or if a single statement body spans multiple lines.
    - Braces may be omitted only if the body of every block associated with an `if=/=else if=/.../=else`.compound statement is placed on a single line.

19. Make **all**`internal` and `private` types `static` or `sealed` unless derivation from them is **required**. As with any implementation detail, they can be changed if/when derivation is required in the future.

Asset Manager

The AM is a repository for static system information and is responsible for making this information available for other modules. The AM can hold its data in local files.

Through the AM, other modules can retrieve heating grid information such as name and a graphical representation of the grid.

The AM provides all configuration data for the production units such as name, possible operation points and a graphical representation of the units itself.

An operation point defines how a device can be operated. Parameters are produced heat, produced / consumed electricity, production costs, consumption of primary energy, produced CO2 emissions. There is typically a minimum and a maximum operation point but, in this case, we assume that there is a maximum operation point for each unit which can be regulated to zero, meaning the device is switched off.

NOT IMPLEMENTED YET