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
```

You may encounter some probles with running `docfx` command

Sometimes `docfx` is not automatically added to your `PATH`. In order to fix that find the location of your dotnet tools. For UNIX based systems it should be `~/.dotnet/tools/docfx`. Then either add your `dotnet/tools` directory to global `PATH` or just run `docfx` from explicitly specified path.

e.g

```
~/.dotnet/tools/docfx docfx.json --serve
```

How to add Markdown files?

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

```yaml
- 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

```yaml
- name: <Name of the directory>
  href: <PathToTheDirectory>/toc.yml
```

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.

OperationStatus Enum

Namespace: `Heatington.Controllers.Enums`

Description

Enumeration representing the status of an operation.

Members

- `SUCCESS` = 0: Indicates a successful operation.
- `LOADING` = 1: Indicates that the operation is in progress or loading.
- `FAILURE` = 2: Indicates a failed operation.

`IDataSource` Interface

Overview

The `IDataSource` interface serves as a protocol for managing data sources in the Heatington application. It stipulates the provision of methods to perform read and write operations on data.

Methods

`Task<List<DataPoint>?> GetDataAsync(string filePath)`

The `GetDataAsync` method signifies an asynchronous operation for retrieving data from a specified data source. The input `string filePath` represents the path to the file containing the pertinent data.

The method is expected to return a `List<DataPoint>` object, which contains the data of interest - specifically the heat demand and electricity price details. In cases where the data retrieval is unsuccessful or if there is no data present at the given file location, the method may return `null`.

`void SaveData(List<DataPoint> data, string filePath)`

The `SaveData` method is purposed towards storing `DataPoint` objects into a CSV file located at a specific file path. The `List<DataPoint> data` argument contains the data points that are set to be saved. The `string filePath` is an argument that provides the location at which the CSV file will be written to or overwritten.

Considering that its return type is `void`, all complications that arise during the data-saving process should be conveyed via exceptions.

This method is currently not implemented

Implementations

Any classes that function as Csv data sources within the Heatington application should implement this interface. This allows for consistency in the management of data across varying data sources and enables smoother transitions between different data sources. Examples of such classes could include `CsvDataSource`, `XmlDataSource`, and the like.

IReadWriteController Interface

Namespace: `Heatington.Controllers.Interfaces`

Description

Generic Interface for all controllers performing I/O operations.

Members

`Task<T?> ReadData<T>()`

Function for reading data out of a model.

Returns

Task to wait for Data.

`Task<OperationStatus> WriteData<T>(T content)`

Function for Writing data into the model.

Parameters

- `content`: Content to write into the model, generic type.

Returns

Task to wait for status of the operation.

NOT IMPLEMENTED

`CsvController` Class

Overview

The `CsvController` class provides a concrete implementation of the `IDataSource` interface specific to data in CSV format, allowing for the reading of such data within the Heatington application.

Methods

`Task<List<DataPoint>?> GetDataAsync(string filePath)`

The `GetDataAsync` method is a function for asynchronously fetching data from a CSV file located at a provided file path.

The method initiates by asynchronously reading the file's entire content into a string (`rawData`). The `rawData` is then deserialized by the `CsvController` utility class into a `CsvData` object – a controller specifically designed to handle and manipulate data in CSV format.

Following the successful deserialization of the `rawData`, the `CsvData` object is converted into a `List` of `DataPoint` objects. Each `DataPoint` object encapsulates heat demand and electricity price data.

Should the method fail to retrieve data from the file or if no data exists at the provided file path, then the method will return `null`.

Upon the occurrence of an exception, the exception's message is displayed using the `Utilities.DisplayException(e.Message)` method and the exception is then re-thrown.

`void SaveData(List<DataPoint> data, string filePath)`

The `SaveData` method remains unimplemented and consequently triggers a `NotImplementedException` when called.

It is projected that in the future, the method will save a `List` of `DataPoint` objects into a CSV file located at a specified file path. The `List<DataPoint> data` parameter represents the data points to be stored. The `string filePath` parameter provides the location of where the CSV file will be created or rewritten.

The utilization of this method remains dependent upon the needs of the Heatington project.

Remarks

While the `CsvController` class is intended to offer a concrete manner for handling CSV data in the application, it's worth to mention that its ability to write data is not implemented. Depending on future development decisions, this feature could potentially remain so. The future of object creation and modification may also leverage design patterns such as the Factory or Builder.

FileController Class

Namespace: `Heatington.Controllers`

Example

```
string pathToFile = Utilities.GeneratePathToFileInAssetsDirectory("testFile.json");
IReadWriteController fileController = new FileController(pathToFile);
await fileController.WriteData("1");
string? data = await fileController.ReadData();
```

Description

Class for performing read/write actions on local files.

Constructor

`FileController(string pathToFile)`

Class constructor. Gets a path to the file to control.

Parameters

- `pathToFile` (string): Path to the location of a file. Follows the pattern `file.json`.

Members

`TryFileOperationRunner<T>(Func<Task<T>> funcToTry)`

Helper method performing try-catch clauses in file-oriented manner.

Parameters

- `funcToTry` (Func<Task>): Function to run inside of try-catch block.

Type Parameters

- `T`: Return type of a function.

Returns

Return the outcome of the run function. If the function returns void, the lambda function should return 0.

`ReadFileFromPath()`

Function for reading the contents of a local file. Reads from the path passed during object initialization.

Returns

Content of the file as a string.

Exceptions

- `FileNotFoundException`: If the file does not exist or the path is not correct, the exception is thrown.

`WriteToFileFromPath(string content)`

Function for writing string data into a local file from a path.

Parameters

- `content` (string): Content to be written to a file as a string.

`ReadData<T>()`

Function for reading the data of a local file. Reads from the path passed during object initialization.

Returns

Data of the file as a string.

`WriteData<T>(T content)`

Function for writing data into a local file.

Parameters

- `content` (T): Content to be written to a file as a string.

Utilities Class

Namespace: `Heatington.Helpers`

Description

Utility class providing various helper methods.

Members

`DisplayException(string message)`

Displays an exception message in red color in the console.

Parameters

- `message` (string): The exception message to display.

`ConvertObject<T>(object? obj)`

Converts an object to the specified type `T`.

Parameters

- `obj` (object): The object to convert.

Returns

The converted object of type `T`.

`GetAbsolutePathToAssetsDirectory()`

Gets the absolute path to the assets directory of the project.

Returns

The absolute path to the assets directory.

`GeneratePathToFileInAssetsDirectory(string fileName)`

Generates the path to a file in the assets directory based on the provided file name.

Parameters

- `fileName` (string): The name of the file.

Returns

The full path to the file in the assets directory.

`DataPoint` Class

Introduction

The `DataPoint` class in the Heatington application represents a single piece of data. Each instance of this class indicates the amount of heat used and the cost of electricity at a particular time.

Structure

`DataPoint(string startTime, string endTime, string heatDemand, string electricityPrice)`

To create a new data point, we have a constructor that accepts four `string` parameters.

- `startTime` and `endTime`: These represent the start and end of a time frame. They must follow the "M/d/yy H:mm" pattern. To keep things standardized, we use `DateTime.ParseExact` and `CultureInfo.InvariantCulture`, which convert the strings into `DateTime` objects.
- `heatDemand` and `electricityPrice`: These values tell us how much heat was used and how much the electricity cost at that time. They are converted into `double` data types using `double.Parse` and `CultureInfo.InvariantCulture`.

Properties

`StartTime`

This `DateTime` property marks the start of a data point's time frame.

`EndTime`

This `DateTime` property marks the end of a time frame for the data point.

`HeatDemand`

This `double` property shows the amount of heat used in the given time frame.

`ElectricityPrice`

This `double` reflects the cost of electricity during that time period.

Future Considerations

Adding a factory method to this class would allow us to manage specific object creation scenarios better, like parameter validation or offering more ways to create a `DataPoint`.

Note

We are not sure about the formatting of the HeatDemand and ElectricityPrice. Because of the different cultures, we might need to consider different decimal separators. We will need to test this in the future.

Documentation for optimizer

- Optimizer module for the first scenario

Properties

- _dataPoints -> will hold the hourly information loaded from the csv from the sdm
- _productionUnits -> hold the production units loaded
- Results -> public get, will hold the computed results

Public methods

- LoadData() -> Loads data from the SDM to the OPT which are needed to compute the optimization
- OptimizeScenario1() -> Once the data is loaded, it will optimize the activation of heat boilers and make the results accessible
- CalculateNetProductionCost() -> Once the optimizer was run, this method will calculate the Net production costs for heat only boilers
- LogResults() -> Will display the computed results
- LogDataPoints() -> Will display the dataPoints if they are loaded from the SDM
- LogProductionUnits() -> Will display the loaded production units which will be used for the optimization

Private methods

- SetOperationPoint() -> will be called to set the operation point on each boiler
- CalculateHeatUnitsRequired() -> is the main part of the optimizer as it organises the data and sets the appropriate computed values
- GetDataPoints() -> Loads data from the SDM
- GetProductionUnits() -> Loads production units (eventually from AM)

How the logic works Once the optimizer starts optimizing:

- each dataPoint loaded needs to be optimized
- production Units need to be stored sorted by efficiency

1. Determines how many boilers need to be activated
2. Based on that number the correct boilers are selected
3. Once the boilers are selected their operation point is set
4. All the information is added to an object ResultHolder which is added to a local list
5. After the foreach loop is completed the results are added to the public property
6. Now the netProductionCosts can be computed

First iteration of the OPT module

Everything under here is left as history, nothing applies anymore.

Scenario 1

The optimizer class has three private fields.

- `_productionUnits` holds the production units objects
- `_energyDataEntries` hold the data eventually provided by the SDM, which reads them from the CSV
- `_resultEntries` currently holds how many boilers need to be activated for any given time period

Optimize scenario 1

`OptimzeScenario1`does what it sounds like. It optimizes the first scenario, there are only heat boilers, no electricity has to be taken into consideration

1. After getting the production units and the time series data which holds how many Mwh are to be produced it orders the production units by their production cost
2. For each "hour" it calls `CalculateHeatingUnitsRequired` which calculates how many heating units have to be activated to satisfy the heating demand

BEWARE `CalculateHeatingUnitsRequired` returns how many heating units need to be activated. This number references the index of the productionUnit list, meaning that 0 means only the production unit with index 0. 2 would mean all the production units with index 0, 1, 2

Net Production cost

Since heat only boilers have no further income or expenses it useful to know what the cost of production is. Once the `OptimzeScenario1()` has run `NetProductionCost()` can be executed. Currently it just calculates it and displays it on screen. Once RDM is implemented it will correctly format the data and make it available to the RDM.

General expedients

- The class is not completed, this is only the initial implementation
- Most classes with which it works with, are going to be changed calling for major refactoring
- It tried to adhere to the SOLID principle as much as possible, refactoring should work ok

`SourceDataManager` Class

Brief Overview

The `SourceDataManager` class is the static repository for the Heatington application's core data management. It gets data from and saves data to a specific data source, which is an form of `IDataSource` interface.

Important Bits

- `_dataSource`: This is a private copy of the the `IDataSource` interface. It has all the methods needed to work with

- a specific data source.

- `_filePath`: This is a private copy of the path to the file where the data operations happen.

- `TimeSeriesData`: This public `List<DataPoint>` works like a holding area for the data. Each item represents a data point at a specific time.

Constructor

`SourceDataManager(IDataSource dataSource, string filePath)`

Constructing a new `SourceDataManager` needs an `IDataSource` instance and a file path string. This gives it the flexibility to work with any type of data source and file location. We use constructor injection to make sure that `SourceDataManager` is not tightly coupled to any specific data source.

Core Methods

- `ConvertApiToCsv(List<DataPoint> dataFromApi)`: This takes a list of `DataPoint` items and saves them using the `SaveData` method from `IDataSource`. **This method is for future use when we implement the API-driven iteration.**

- `FetchTimeSeriesData()`: This method fetches the data from the `_dataSource` using the `GetData` method and stores it in the `TimeSeriesData` property.

- `LogTimeSeriesData()`: This method logs the data in `TimeSeriesData`. Each log message contains the index, formatted start and end times, heat demand, and electricity price for each data point. **This method will be removed once we move to the GUI-driven iteration.**

Quick Note

The `SourceDataManager` implementation simplifies testing because `IDataSource` can be easily mocked. It also adds flexibility, as different implementations of `IDataSource` can be used without major code changes.

An Example Of How To Use It

```
using Heatington.Data;

namespace Heatington
{
    internal static class Program
    {

        public static async Task Main(string[] args) // async Task -> if we want to
implement async operation
                                                     // especially for IO
        {
            // Define the file path
            const string filePath = "../Assets/winter_period.csv";

            // Create a new CsvDataSource
            IDataSource dataSource = new CsvDataSource();

            SourceDataManager.SourceDataManager sdm = new(dataSource, filePath);

            // Fetch data from dataSource
            await sdm.FetchTimeSeriesDataAsync().ConfigureAwait(true);

            // Log the loaded data to the console
            sdm.LogTimeSeriesData();

            Console.WriteLine("Data loaded successfully.");

        }
    }
}
```

NOT IMPLEMENTED YET

NOT IMPLEMENTED YET

NOT IMPLEMENTED YET

FileControllerTests Class

Namespace: `Heatington.Tests.Controllers`

Description

Unit tests for the `FileController` class.

Constructor

`FileControllerTests()`

Creates a temporary test folder for testing purposes.

Members

`ReadFileFromPath_ReadFile_ReadsCorrectContent()`

Unit test to verify that reading a file from a specified path reads the correct content.

`WriteFileFromPath_WriteFile_WritesCorrectContent()`

Unit test to verify that writing to a file from a specified path writes the correct content.

`WriteFileFromPath_WriteEmptyStringToFile_CreatesFile()`

Unit test to verify that writing an empty string to a file creates the file.

`WriteToFileFromPath_WriteToTheSameFileTwice_CreatesTwo()`

Unit test to verify that writing to the same file twice creates two files with the same name.

`ReadFileFromPath_ReadNotExistingFile_FileNotFound(string wrongFileName)`

Unit test to verify that reading from a path to a file that does not exist throws a `FileNotFoundException`.

IDisposable Implementation

`Dispose()`

Clears the temporary test directory after all tests have been run.

NOT IMPLEMENTED YET

NOT IMPLEMENTED YET

NOT IMPLEMENTED YET

NOT IMPLEMENTED YET