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 CO_2 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.

`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.

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.");

        }
    }
}
```