Heat Production Management Project for Semester Project 2

Kacper Grzyb Sebestyen Deak Ignad Bozhinov
Leonardo Gianola Levente Sohar

03-06-2024

Contents

Chapter 1

Introduction

Introduction chapter goes here

Chapter 2

Release Planning

Release Planning chapter goes here

Chapter 3

Sprint Materials

Sprint Planning Chapter goes here

Chapter 4

Technical Details

4a Software Architecture and Design

Tools

The team decided to use the ASP.NET Core framework's Razor Pages for building the project for reasons such as:

- The most important argument for using ASP.NET Core is it's cross-platform functionality. The developers in the team use both Macintosh and Windows based systems, therefore a framework that could switch seamlessly between them was crucial.

- As the name suggests, the framework runs in the .NET ecosystem, which is what the team has been taught in the course so far therefore it is what the team is most experienced and most comfortable working in.

- With Razor Pages being a web-page based solution, the UI is mostly composed of HTML and CSS, which some team members already had experience in and the rest was eager to learn. A light-weight, web-based solution allowed the team to be more flexible, and develop the app at a more rapid pace compared to if the team chose a Model-View-Controler (or a Model-View-ViewModel) solution. This freedom allowed for more features and made the process of adjusting to changing the project requirements easier.

- Additionally, Razor Pages allows the project to be published and deployed straight away, which is a nice bonus.

As for other tools, the team used:

- Github: Source and Version Control, Collaborative Development of the App
- Jira: Task Management and Planning as well as adhearing to Agile which was one of the requirements for the project
- Discord: Communication and Resource Sharing
- diagrams.net—draw.io: Creating UML Diagrams
- Figma: Prototyping and General UI Design
- Visual Studio and Visual Studio Code: Code Editors

Application Flow

The user journey begins in the Homepage, which is the project requirements' Asset Manager component. The user inside of this component has two possible routes:

- Load Heat Demand Data
- Configure Production Units

Let's explore unit configuration first. When the user clicks on that option, they get redirected to the unit configuration page, where they can change the properties of the pre-loaded production units provided by the project requirements (Gas Boiler, Oil Boiler, Gas Motor and Electric Boiler), while also being able to create and configure thier own custom production units. From this page the only option is to go back to the Homepage. Next, loading the heat demand data can be done in one of three ways: choosing the preloaded data provided by Danfoss Deliveries (Winter Period Heat Demand 08.02.2023-14.02.2023 and Summer Period Heat Demand 08.07.2023-14.07.2023), loading the data from a custom CSV file or loading the data from a custom Excel Workbook file. Once heat demand data has been loaded, more options open up for the user in the navbar:

- Database Preview
- Result Data Manager

The Database Preview component was not initially intended to stay in the final product. At first it was used solely for debugging purposes, when the team was implementing the data loading options. But due to supervisor and project presentation feedback, we decided to keep this component to also allow the user to double-check if their data was correctly loaded and recognized by our application. From this page the user has the option to either go back to the Homepage or go to the Result Data Manager.

The Result Data Manager is where the magic happens. It hosts the main functionality of the program which is the Optimizer Component, Data Visualization and saving optimized data. By default, the Optimizer runs automatically whenever the user get redirected to this page with the settings: Use all production units, optimize for costs, use the standard optimizer. We made this choice so that the user is immediately greeted by the results, and not just a blank page. Once on the page, the user can tune the optimizer to produce data that suits their individual needs. Here are the checkbox options given:

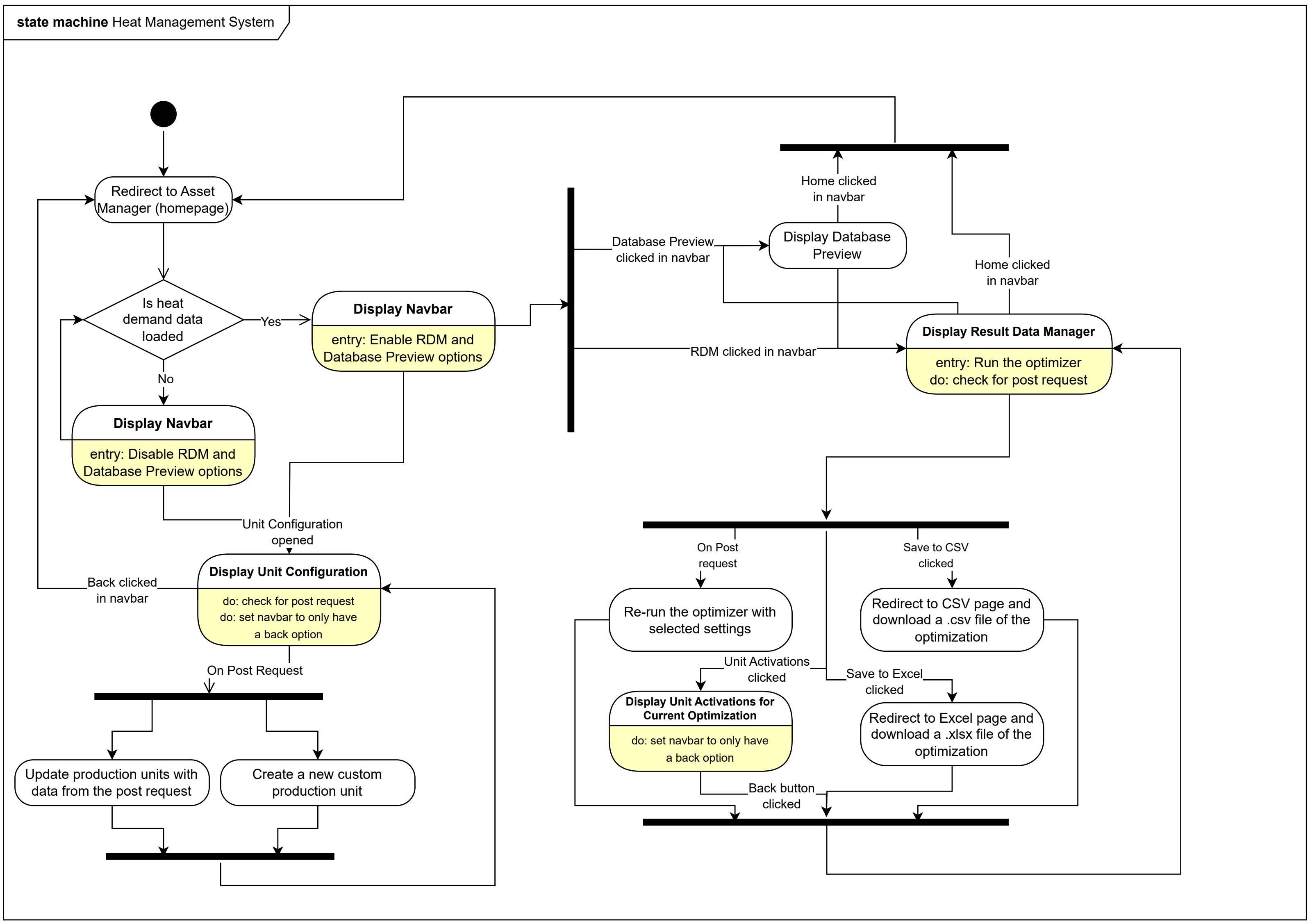
- Choose which production units to include in the optimization process
- Choose which parameters to optimize for (Costs, CO2 Emissions)
- Choose which optimizer to use (Standard or Neural Network)

Then the user can rerun the optimization process with the selection options to obtain their results. The results themselves are displayed in the same page, both in text and graph format, which covers the data visualization requirement. Having the optimization outcomes opens up some more paths to take:

- Unit Activations Page
- Save Optimized Data to CSV
- Save Optimized Data to an Excel Workbook

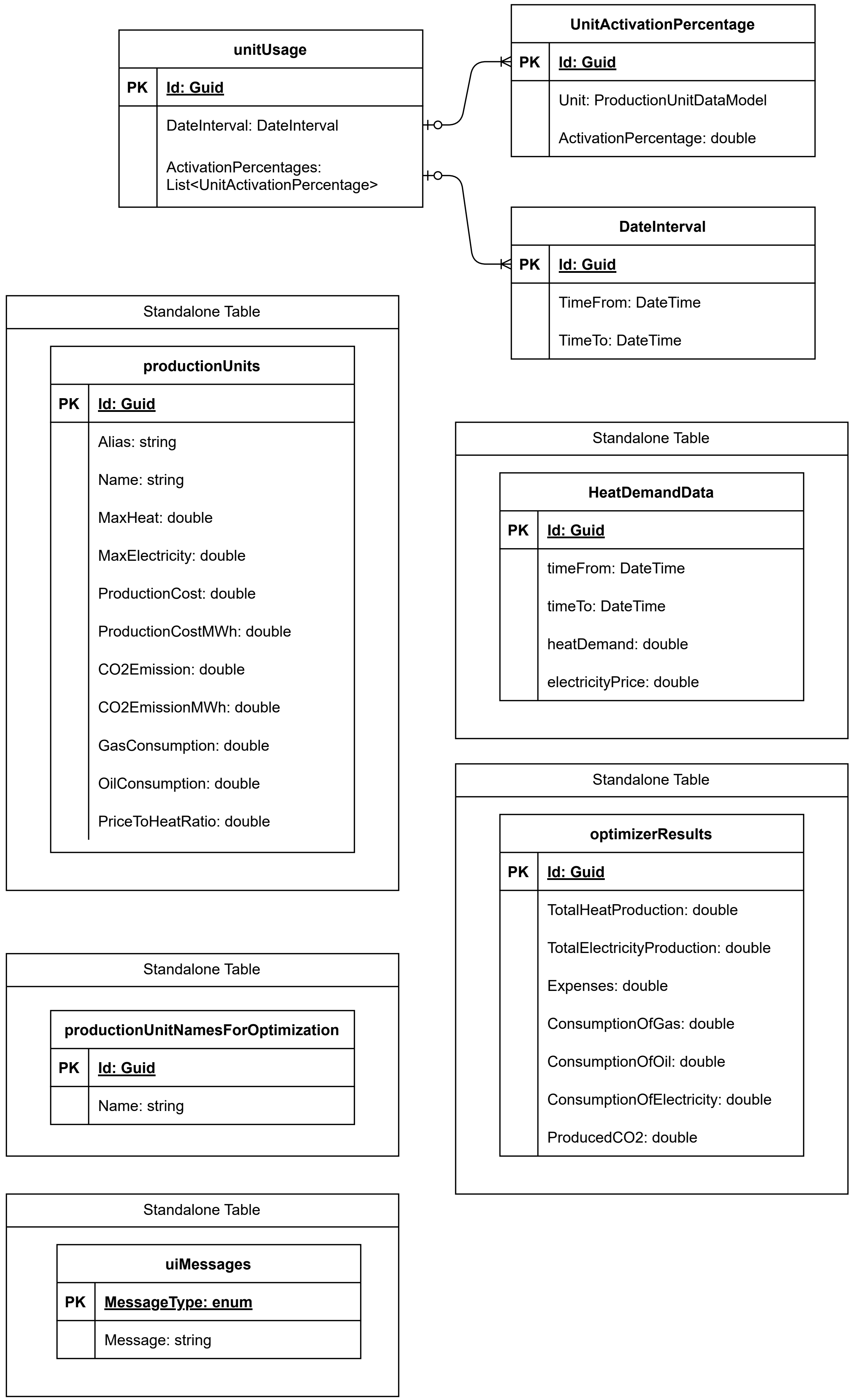
The Unit Activations Page is the last component the user gets access to in their journey. It shows the activation percentages for each production unit at each timeframe. They can be treated as instructions for the user on how to manage the boilers to achieve the optimal Produced Heat, Electricity Consumption, Production Costs, Fuel Consumption and CO2 Emissions proposed by the current optimizer configuration. The last options allow the user to save all of the optimizer output data mentioned previously into a single CSV or Excel Workbook file, which gets downloaded to the users Downloads folder.

The project in itself does not operate on many states, therefore it does not need nor have a complex state machine diagram. Most of the application's functionality happens automatically and/or instantly, therefore creating custom states and switching between them would have created unnecessary complexity and slowed down the program. With that in mind, during the creation of the state machine diagram, we decided to focus on how Razor Pages functions in general, where the main states are waiting for post requests which is, in other words, user input. Despite that, we still thought that a state machine diagram was one of the best ways to represent the application flow and user journey therefore it can be seen below.


state machine Heat Management System
Redirect to Asset Manager (homepage)
Is heat demand data loaded
Yes
No
Display Navbar
entry: Enable RDM and Database Preview options
Display Navbar
entry: Disable RDM and Database Preview options
Unit Configuration opened
Display Unit Configuration
do: check for post request
do: set navbar to only have a back option
Back clicked in navbar
On Post Request
Update production units with data from the post request
Create a new custom production unit
Database Preview clicked in navbar
Display Database Preview
Home clicked in navbar
Home clicked in navbar
RDM clicked in navbar
Display Result Data Manager
entry: Run the optimizer
do: check for post request
On Post request
Save to CSV clicked
Re-run the optimizer with selected settings
Redirect to CSV page and download a .csv file of the optimization
Unit Activations clicked
Save to Excel clicked
Display Unit Activations for Current Optimization
do: set navbar to only have a back option
Redirect to Excel page and download a .xlsx file of the optimization
Back button clicked

Database Design

Before diving into the program architecture the in-memory database solution offered by Razor Pages must first be mentioned. In order to achieve data persistance while switching in between pages in a Razor Pages project, a database must be used. Since we did not want to go too far out of the scope of the project, we decided not to use a dedicated database solution like a MySQL or MSSQL Server for this project, expecially because the team has not had any database modeling courses yet. Instead we chose a middle-ground, which is the before mentioned in-memory database. This soultion offers similar functionality to a real database, with the comfort of running in the program's memory, which eliminates potential connection, authentication privilege and/or security risks and issues connected with using databases. To represent the applications data storage sytstem the group devised an improvised custom uml diagram inspired by other standardized database diagrams. It can be seen below, together with explanation for each of the tables.


unitUsage
PK
Id: Guid
DateInterval: DateInterval
ActivationPercentages:
List<UnitActivationPercentage>
UnitActivationPercentage
PK
Id: Guid
Unit: ProductionUnitDataModel
ActivationPercentage: double
DateInterval
PK
Id: Guid
TimeFrom: DateTime
TimeTo: DateTime
Standalone Table
productionUnits
PK
Id: Guid
Alias: string
Name: string
MaxHeat: double
MaxElectricity: double
ProductionCost: double
ProductionCostMWh: double
CO2Emission: double
CO2EmissionMWh: double
GasConsumption: double
OilConsumption: double
PriceToHeatRatio: double
Standalone Table
HeatDemandData
PK
Id: Guid
timeFrom: DateTime
timeTo: DateTime
heatDemand: double
electricityPrice: double
Standalone Table
optimizerResults
PK
Id: Guid
TotalHeatProduction: double
TotalElectricityProduction: double
Expenses: double
ConsumptionOfGas: double
ConsumptionOfOil: double
ConsumptionOfElectricity: double
ProducedCO2: double
Standalone Table
productionUnitNamesForOptimization
PK
Id: Guid
Name: string
Standalone Table
uiMessages
PK
MessageType: enum
Message: string

The database does not fully comply to database system standards. We do see the limitations and flaws of this design choice, such as the lack of connections between tables and the lack of foreign keys inside of each table. Instead some fields in the tables use custom class data types for ease of use. Despite the flaws of using DbSets, we found that their functionality was sufficient for the scope of the project. It is also worth to mention, that due to our lack of experience with Razor Pages, we were unsure of how DbSets and in-memory data structures function. Because of that in the middle of development the database had to be refactored from a more json-like data storage structure to one that complies with standards enforced by DbSets, such as using Primary Keys. It is because of this process that we ended up having a mixture of both structures. The most crucial tables found within the programs database are the *productionUnits* and *HeatDemandData*. As the names suggests they contain data that is needed for the application's main functionality - calculating optimized production unit usage results for the inputted heat demand for each timeframe. Some of the properties inside of *productionUnits* are additions from the team, such as *ProductionCostMWh* and *CO2EmissionMWh*, which made it easier to remember how to calculate final optimization results, and *PriceToHeatRatio*, which also was a helper variable in optimization calculations. When it comes to storing the results from the optimizer for later display or download, the *unitUsage* and *optimizerResults* tables were used. We think their names explain their functionality sufficiently. Lastly, the *uiMessages*, as the name suggests stores messages for the user that may appear in the user interface under certain conditions, such as confirmation of a successful data upload or an error during the optimization process. These messages have to be stored in a database, because of how user input works in Razor Pages. The only way the team found to obtain user input is through post requests, which only make impact on the interface after a page redirect, which would reset the pages memory and lose the message intended for display. That is why for these messages to persist it is important to have a place to save them before the redirect.

Code Architecture

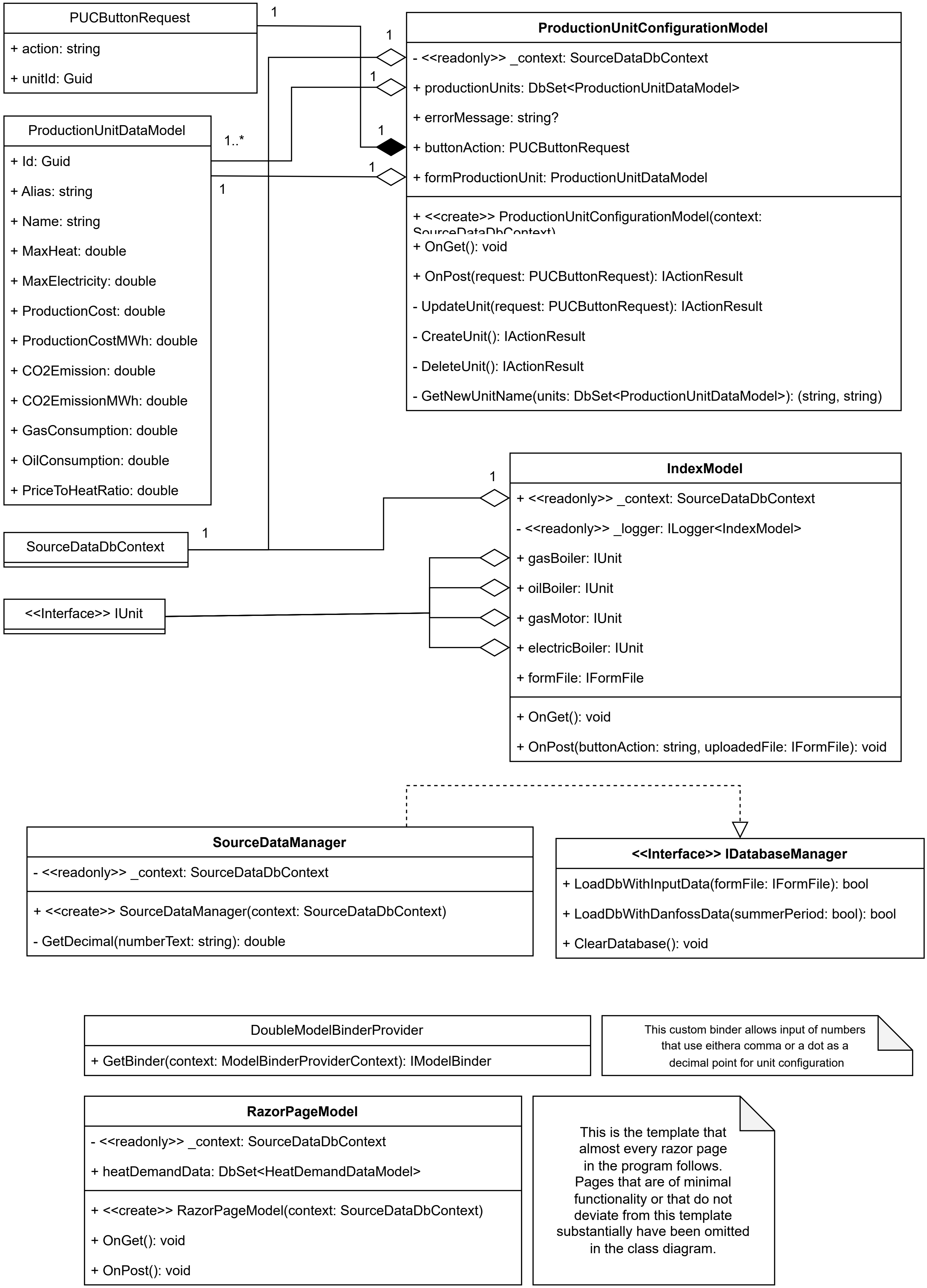
With the database covered, we can shift our focus towards the rest of the codebase. Any issues, bugs or problems encountered by the group during the development/implementation phase of the project will be <u>underlined</u> in this subsection of the report. We shall start in the same place as the program's structure: *Program.cs*. This file mostly contains automatically generated Razor Pages setup code for every built-in feature to work correctly, but it is also where one of the first problems started. Culture Info was a common problem across all groups working on this project in this semester, and it definitely caused some issues for us as well, like incorrect DateTime variable parsing, incorrect decimal separator detection, errors with reading data from files and incorrect data display. Fortunately manually setting the user's DefaultThreadCurrentCulture and DefaultThreadCurrentUICulture to the Danish CultureInfo was an easy fix that solved most of these problems. From this code file, the Razor Pages applications gets built and the user gets redirected to the *Index.cshtml* page with the Navbar and DataUpload ViewComponents overlayed on top of it, which we will come back to shortly. The group chose to create both component and class diagrams to explain how the code is designed. The component diagram helps us break down the program into the components from the project's requirements, and clarify in which location each component is implemented. This diagram also plays a part in explaining the interactions between each component, and the class diagram is able to complete this explanation, as well as go more in depth for each component, breaking them down into individual classes. The component diagram can be seen below.

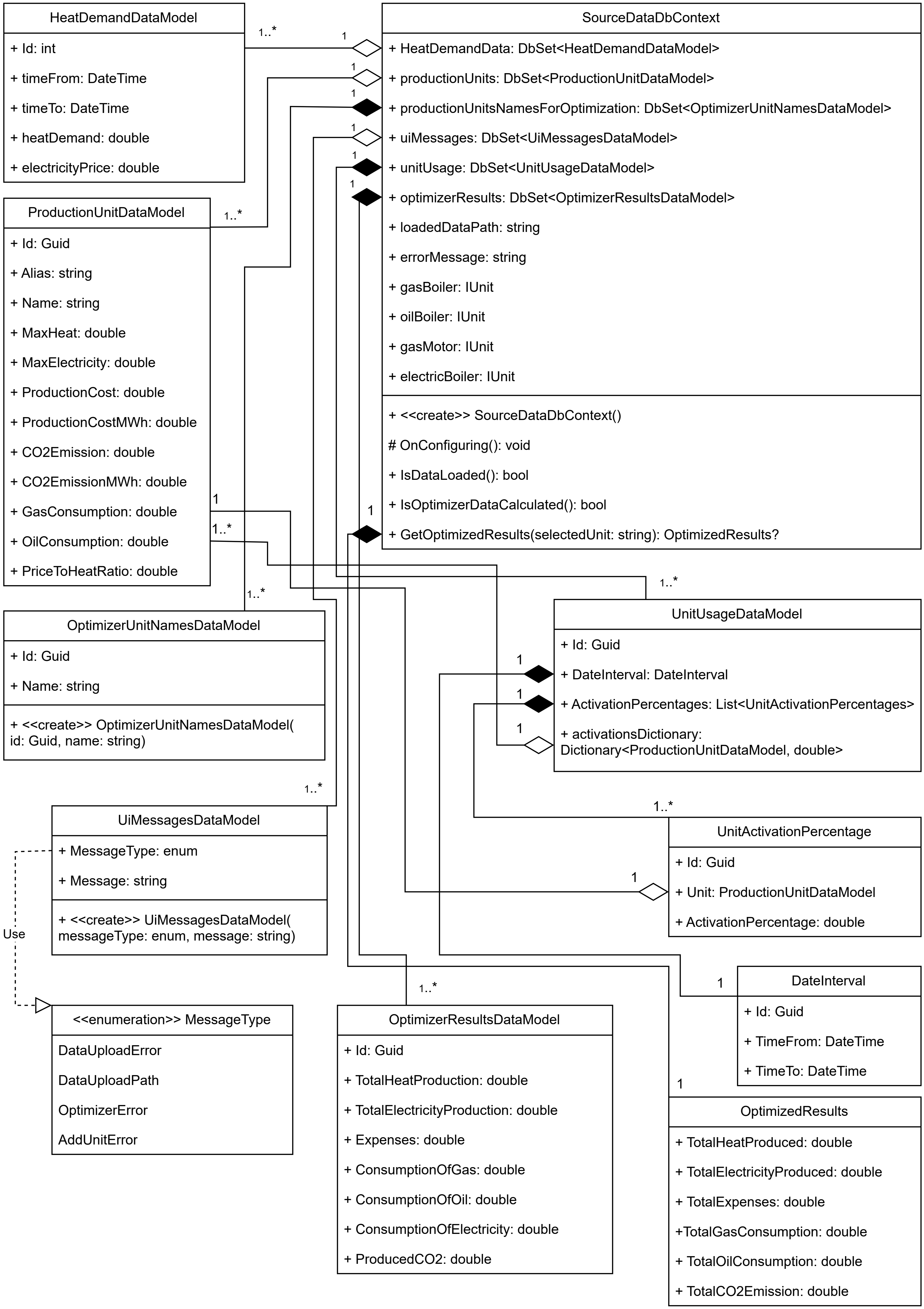
Following the component hierarchy we will cover the **Source Data Manager** next. The class diagrams facilitating this explanation can be seen below this paragraph. Our implementation of this component makes it handle the most important interaction between the program and the database, which is loading user input or pre-loaded data and deleting that data. The component itself is an implementation of the *IDatabaseManager* interface, which was used to reinforce the SOLID principles within our codebase, decouple dependencies, make the code less brittle and more expandable. The main property of **Source Data Manager** is *_context*, which will appear in most other components, and is essentially a reference to the in-memory database. *SourceDataDbContext*, which implements the *DbContext* Interface is the class that contains the in-memory database. It uses class data models such as *HeatDemandDataModel*, *ProductionUnitDataModel*, *OptimizerUnitNamesDataModel*, *UnitUsageDataModel*, *UiMessagesDataModel* and *OptimizerResultsDataModel* for data storage. The other properties it contains have custom get and set methods defined which interact with their respective DbSets, and serve as quick references to specific items from the database that are widely used throughout the program. This is a result of major database refactor, before which these properties were preset in the **Source Data Manager** component. In order to avoid merge conflicts and errors these properties were kept but puerly as references to the database, which is where C#'s feature of custom property getters and setters came in really handy. Moving onto the **Asset Manager** component, which uses the *IndexModel.cshtml.cs* class as it's code-behind. As mentioned previously, this is the Homepage of the program, it contains a reference to the database, OnGet and OnPost methods for handling user input. In this component, we also made use of Razor Pages' View Component feature, which essentially allowed us to display a page within another page, which helped break up the code into more manageable chunks and work on different parts of the page without interfering with each other. The only two View Components in the program are:

- DataUploadView - a page that either calls **Source Data Manager** to load in pre-provided heat demand data, or accepts a file uploaded by the user, while checking if its format and extension is correct.

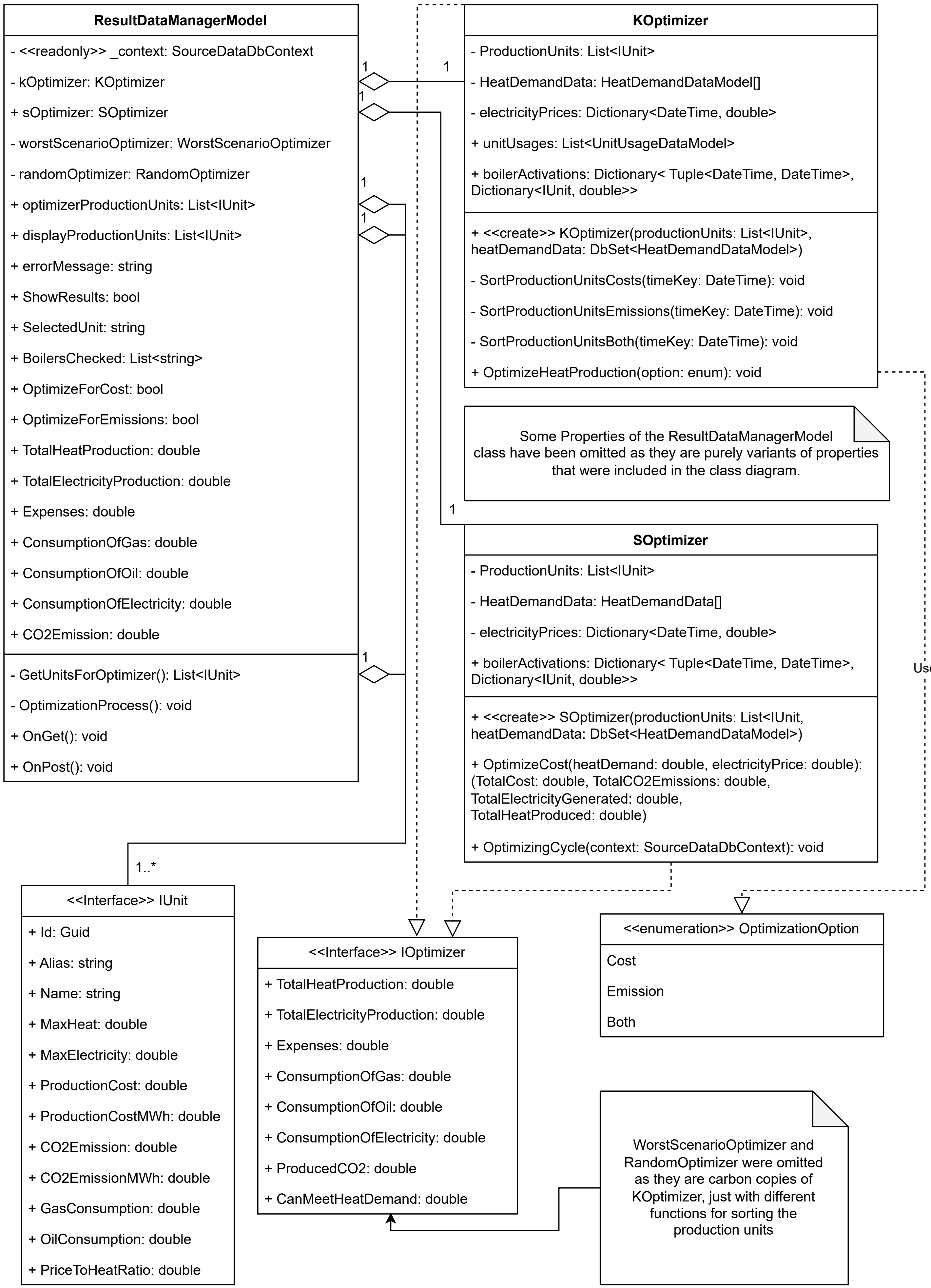
- PageNavigationView - as the name suggests, it is an overlay that allows the user to navigate through the program, namely to three components: **Asset Manager**, **Result Data Manager** and **Database Preview**. The navigation bar is displayed in each mentioned component.

The **Asset Manager** also gives access to the **Production Unit Configuration** component. This is where the user can configure properties of pre-existing or custom added production units. All of the user input is again handled in the OnGet and OnPost methods with the aid of the *PUCButtonRequest* class, which allows the OnPost request to recognize which button in the UI was pressed and to react accordingly. Unit creation, updating and deletion processes use unit IDs from the *ProductionUnitDataModel* class to make sure changes occur to the right unit, thanks to Guid ensuring each Production Unit record in the database has a unique ID. Obtaining the new parameters for chaning a picked production unit is handled thorugh binding the *formProductionUnit* property, which is a technique used also in other pages when handling user input. This is also where an issue came up connected with Culture Info, that was not solved in *Program.cs*, which were the decimal separators for number inputs. This is why a custom *DoubleModelBinder* was written, to convert HTML form input's values to variables of type *double* no matter if they used a dot or a comma as their decimal separator. A safety check was also implemented, so that a production unit's Alias property has to be unique, to eliminate potential errors with the optimization process, which will be covered soon.

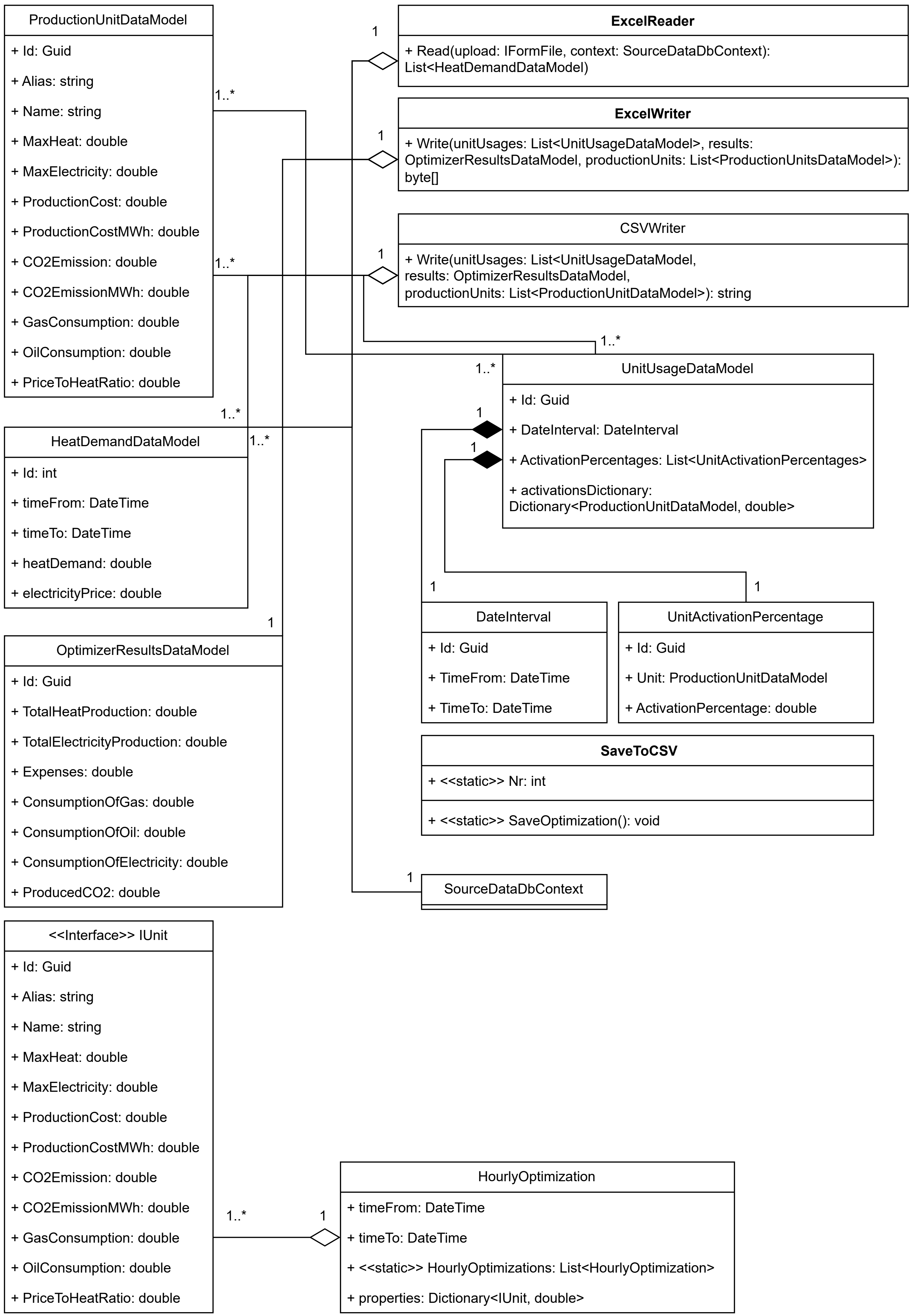

PUCButtonRequest
+ action: string
+ unitId: Guid
ProductionUnitConfigurationModel
- <<readonly>> _context: SourceDataDbContext
+ productionUnits: DbSet<ProductionUnitDataModel>
+ errorMessage: string?
+ buttonAction: PUCButtonRequest
+ formProductionUnit: ProductionUnitDataModel
+ <<create>> ProductionUnitConfigurationModel(context: SourceDataDbContext)
+ OnGet(): void
+ OnPost(request: PUCButtonRequest): IActionResult
- UpdateUnit(request: PUCButtonRequest): IActionResult
- CreateUnit(): IActionResult
- DeleteUnit(): IActionResult
- GetNewUnitName(units: DbSet<ProductionUnitDataModel>): (string, string)
ProductionUnitDataModel
+ Id: Guid
+ Alias: string
+ Name: string
+ MaxHeat: double
+ MaxElectricity: double
+ ProductionCost: double
+ ProductionCostMWh: double
+ CO2Emission: double
+ CO2EmissionMWh: double
+ GasConsumption: double
+ OilConsumption: double
+ PriceToHeatRatio: double
1
1
1
1
1
1..*
1
SourceDataDbContext
1
<<Interface>> IUnit
IndexModel
1
+ <<readonly>> _context: SourceDataDbContext
- <<readonly>> _logger: ILogger<IndexModel>
+ gasBoiler: IUnit
+ oilBoiler: IUnit
+ gasMotor: IUnit
+ electricBoiler: IUnit
+ formFile: IFormFile
+ OnGet(): void
+ OnPost(buttonAction: string, uploadedFile: IFormFile): void
SourceDataManager
- <<readonly>> _context: SourceDataDbContext
+ <<create>> SourceDataManager(context: SourceDataDbContext)
- GetDecimal(numberText: string): double
<<Interface>> IDatabaseManager
+ LoadDbWithInputData(formFile: IFormFile): bool
+ LoadDbWithDanfossData(summerPeriod: bool): bool
+ ClearDatabase(): void
DoubleModelBinderProvider
+ GetBinder(context: ModelBinderProviderContext): IModelBinder
This custom binder allows input of numbers that use eithera comma or a dot as a decimal point for unit configuration
RazorPageModel
- <<readonly>> _context: SourceDataDbContext
+ heatDemandData: DbSet<HeatDemandDataModel>
+ <<create>> RazorPageModel(context: SourceDataDbContext)
+ OnGet(): void
+ OnPost(): void
This is the template that almost every razor page in the program follows. Pages that are of minimal functionality or that do not deviate from this template substantially have been omitted in the class diagram.


HeatDemandDataModel
+ Id: int
+ timeFrom: DateTime
+ timeTo: DateTime
+ heatDemand: double
+ electricityPrice: double
SourceDataDbContext
+ HeatDemandData: DbSet<HeatDemandDataModel>
+ productionUnits: DbSet<ProductionUnitDataModel>
+ productionUnitsNamesForOptimization: DbSet<OptimizerUnitNamesDataModel>
+ uiMessages: DbSet<UiMessagesDataModel>
+ unitUsage: DbSet<UnitUsageDataModel>
+ optimizerResults: DbSet<OptimizerResultsDataModel>
+ loadedDataPath: string
+ errorMessage: string
+ gasBoiler: IUnit
+ oilBoiler: IUnit
+ gasMotor: IUnit
+ electricBoiler: IUnit
+ <<create>> SourceDataDbContext()
# OnConfiguring(): void
+ IsDataLoaded(): bool
+ IsOptimizerDataCalculated(): bool
+ GetOptimizedResults(selectedUnit: string): OptimizedResults?
ProductionUnitDataModel
+ Id: Guid
+ Alias: string
+ Name: string
+ MaxHeat: double
+ MaxElectricity: double
+ ProductionCost: double
+ ProductionCostMWh: double
+ CO2Emission: double
+ CO2EmissionMWh: double
+ GasConsumption: double
+ OilConsumption: double
+ PriceToHeatRatio: double
OptimizerUnitNamesDataModel
+ Id: Guid
+ Name: string
+ <<create>> OptimizerUnitNamesDataModel(
id: Guid, name: string)
UnitUsageDataModel
+ Id: Guid
+ DateInterval: DateInterval
+ ActivationPercentages: List<UnitActivationPercentages>
+ activationsDictionary:
Dictionary<ProductionUnitDataModel, double>
UiMessagesDataModel
+ MessageType: enum
+ Message: string
+ <<create>> UiMessagesDataModel(
messageType: enum, message: string)
Use
UnitActivationPercentage
+ Id: Guid
+ Unit: ProductionUnitDataModel
+ ActivationPercentage: double
DateInterval
+ Id: Guid
+ TimeFrom: DateTime
+ TimeTo: DateTime
<<enumeration>> MessageType
DataUploadError
DataUploadPath
OptimizerError
AddUnitError
OptimizerResultsDataModel
+ Id: Guid
+ TotalHeatProduction: double
+ TotalElectricityProduction: double
+ Expenses: double
+ ConsumptionOfGas: double
+ ConsumptionOfOil: double
+ ConsumptionOfElectricity: double
+ ProducedCO2: double
OptimizedResults
+ TotalHeatProduced: double
+ TotalElectricityProduced: double
+ TotalExpenses: double
+TotalGasConsumption: double
+ TotalOilConsumption: double
+ TotalCO2Emission: double

Back in **Asset Manager**, the navbar allows the user to navigate to the **Result Data Manager** and **Database Preview** components. The latter does not contain any distinguishing code, and was covered entirely in the subsection Application Flow. The **Result Data Manager**, on the other hand, is the most substancial component of the whole application. The code-behind for its page is in *ResultDataManagerModel.cshtml.cs*. It contains the usual reference to the database, but a thing that stands out are the two IUnit Lists and the IUnit Interface itself. Firstly, the program contains both a IUnit Interface and a ProductionUnitDataModel parent class for production units, which are used interchangeably, because of a DbSet issue. At one point in the development when trying to access one of the feature of DbSets, it turned out that a DbSet cannot be of an interface type. At that point, the IUnit interface was used extensively within the codebase and there was not enough time to make the changes necessary, so a band-aid fix was applied, with which the ProductionUnitDataModel class was created. The **Result Data Manager** also hosts the **Optimizer** component, which consists of the KOptimizer, SOptimizer, GeneticAlgorithm (The Neural Network Optimizer) (and WorstScenarioOptimizer with RandomOptimizer, which are used solely as comparison data for some of the graphs). Multiple Optimizer implementations are a result of one of the Sprints the team carried out, where everyone worked on an implementation and at the end the best one was chosen. The main optimizer that the application uses is the KOptimizer, which is also referred to as the Standard Optimizer. It is a simple Greedy Algorithm implementation that judges which production units are the most cost effective for each timeframe, and uses them to produce as much heat as possible until the demand is met. An outlier to the goal of our optimizer sprint was the Neural Network Optimizer, which was too good of an idea to pass up on, while begin to risky to be the only optimization algorithm, so we decided to use both. The explanation on how it is implemented can be found in Chapter 5c. The Optimization Process has 3 main configuration options to consider before re-running it, which were also mentioned in Application Flow. This configuration is handled in the OnPost method of ResultDataManagerModel. It is important to mention, that due to allowing the user to choose their own production unit scenario, each **Optimizer** implementation had to have a built-in check to make sure that the production units provided can even meet the heat demand. If that is not the case, the uiMessages DbSet is used in the database to display an error message to the user. Additionally, this option meant that we needed to store all of the production units and the production units selected for the optimization process by the user at the same time. This is where a second problem with DbSets came up. Razor Pages does not seem to allow for two DbSets of the same type to exist within one database, instead merging them into one singular DbSet if that was the case, resulting in duplicate key value errors. A workaround had to be implemented. This is why the OptimizerUnitNames-DataModel exists, which stores just the production unit aliases, which are then matched with the prodcution units in the productionUnits DbSet and passed into the **Optimizer**. This is also why the **Unit Configuration** component makes sure that each alias is unique. Once the optimization process is finished, the **Result Data Manager** saves the outputs to the database using the UnitUsageDataModel and OptimizerResultsDataModel classes. This is what later allows the ExcelWriter and CSVWriter classes to access this data, convert it to their own respective formats, and redirect the user to a file download page. Lastly, **Result Data Manager** also hosts the **Unit Activations** component, which in similar fashion as the **Database Preview**, was covered in Application Flow.



ResultDataManagerModel
- <<readonly>> _context: SourceDataDbContext
- kOptimizer: KOptimizer
+ sOptimizer: SOptimizer
- worstScenarioOptimizer: WorstScenarioOptimizer
- randomOptimizer: RandomOptimizer
+ optimizerProductionUnits: List<IUnit>
+ displayProductionUnits: List<IUnit>
+ errorMessage: string
+ ShowResults: bool
+ SelectedUnit: string
+ BoilersChecked: List<string>
+ OptimizeForCost: bool
+ OptimizeForEmissions: bool
+ TotalHeatProduction: double
+ TotalElectricityProduction: double
+ Expenses: double
+ ConsumptionOfGas: double
+ ConsumptionOfOil: double
+ ConsumptionOfElectricity: double
+ CO2Emission: double
- GetUnitsForOptimizer(): List<IUnit>
- OptimizationProcess(): void
+ OnGet(): void
+ OnPost(): void
1
1
1
1
1
1
1
1..*
KOptimizer
- ProductionUnits: List<IUnit>
- HeatDemandData: HeatDemandDataModel[]
- electricityPrices: Dictionary<DateTime, double>
+ unitUsages: List<UnitUsageDataModel>
+ boilerActivations: Dictionary< Tuple<DateTime, DateTime>, Dictionary<IUnit, double>>
+ <<create>> KOptimizer(productionUnits: List<IUnit>, heatDemandData: DbSet<HeatDemandDataModel>)
- SortProductionUnitsCosts(timeKey: DateTime): void
- SortProductionUnitsEmissions(timeKey: DateTime): void
- SortProductionUnitsBoth(timeKey: DateTime): void
+ OptimizeHeatProduction(option: enum): void
Some Properties of the ResultDataManagerModel class have been omitted as they are purely variants of properties that were included in the class diagram.
SOptimizer
- ProductionUnits: List<IUnit>
- HeatDemandData: HeatDemandData[]
- electricityPrices: Dictionary<DateTime, double>
+ boilerActivations: Dictionary< Tuple<DateTime, DateTime>, Dictionary<IUnit, double>>
+ <<create>> SOptimizer(productionUnits: List<IUnit, heatDemandData: DbSet<HeatDemandDataModel>)
+ OptimizeCost(heatDemand: double, electricityPrice: double): (TotalCost: double, TotalCO2Emissions: double, TotalElectricityGenerated: double, TotalHeatProduced: double)
+ OptimizingCycle(context: SourceDataDbContext): void
Use
<<Interface>> IUnit
+ Id: Guid
+ Alias: string
+ Name: string
+ MaxHeat: double
+ MaxElectricity: double
+ ProductionCost: double
+ ProductionCostMWh: double
+ CO2Emission: double
+ CO2EmissionMWh: double
+ GasConsumption: double
+ OilConsumption: double
+ PriceToHeatRatio: double
<<Interface>> IOptimizer
+ TotalHeatProduction: double
+ TotalElectricityProduction: double
+ Expenses: double
+ ConsumptionOfGas: double
+ ConsumptionOfOil: double
+ ConsumptionOfElectricity: double
+ ProducedCO2: double
+ CanMeetHeatDemand: double
<<enumeration>> OptimizationOption
Cost
Emission
Both
WorstScenarioOptimizer and RandomOptimizer were omitted as they are carbon copies of KOptimizer, just with different functions for sorting the production units


ProductionUnitDataModel
+ Id: Guid
+ Alias: string
+ Name: string
+ MaxHeat: double
+ MaxElectricity: double
+ ProductionCost: double
+ ProductionCostMWh: double
+ CO2Emission: double
+ CO2EmissionMWh: double
+ GasConsumption: double
+ OilConsumption: double
+ PriceToHeatRatio: double
ExcelReader
+ Read(upload: IFormFile, context: SourceDataDbContext): List<HeatDemandDataModel>
ExcelWriter
+ Write(unitUsages: List<UnitUsageDataModel>, results: OptimizerResultsDataModel, productionUnits: List<ProductionUnitsDataModel>): byte[]
CSVWriter
+ Write(unitUsages: List<UnitUsageDataModel, results: OptimizerResultsDataModel, productionUnits: List<ProductionUnitDataModel>): string
UnitUsageDataModel
+ Id: Guid
+ DateInterval: DateInterval
+ ActivationPercentages: List<UnitActivationPercentages>
+ activationsDictionary: Dictionary<ProductionUnitDataModel, double>
HeatDemandDataModel
+ Id: int
+ timeFrom: DateTime
+ timeTo: DateTime
+ heatDemand: double
+ electricityPrice: double
DateInterval
+ Id: Guid
+ TimeFrom: DateTime
+ TimeTo: DateTime
UnitActivationPercentage
+ Id: Guid
+ Unit: ProductionUnitDataModel
+ ActivationPercentage: double
OptimizerResultsDataModel
+ Id: Guid
+ TotalHeatProduction: double
+ TotalElectricityProduction: double
+ Expenses: double
+ ConsumptionOfGas: double
+ ConsumptionOfOil: double
+ ConsumptionOfElectricity: double
+ ProducedCO2: double
SaveToCSV
+ <<static>> Nr: int
+ <<static>> SaveOptimization(): void
SourceDataDbContext
<<Interface>> IUnit
+ Id: Guid
+ Alias: string
+ Name: string
+ MaxHeat: double
+ MaxElectricity: double
+ ProductionCost: double
+ ProductionCostMWh: double
+ CO2Emission: double
+ CO2EmissionMWh: double
+ GasConsumption: double
+ OilConsumption: double
+ PriceToHeatRatio: double
HourlyOptimization
+ timeFrom: DateTime
+ timeTo: DateTime
+ <<static>> HourlyOptimizations: List<HourlyOptimization>
+ properties: Dictionary<IUnit, double>
1
1..*
1
1..*
1
1..*
1..*
1
1..*
1..*
1
1
1
1
1
1..*
1

4b Simple Design

Simple design yapping goes here

4c Incremental Design

Incremental Design yapping goes here

4d Refactoring

Most of the group did not have much experience in this process because most of the assignments we were being given were not on a large enough scale to consider restructuring code throughout their development - the tasks and functionality were finished long before poor code design could become a problem. The scope and timespan of this project was much larger, therefore the we decided that refactoring should play a considerable role in our development. Somewhat to our surprise, refactoring felt quite natural, especially since this was the team's first project with Razor Pages, the more we learned about the framework and the more we understood, the more we saw of what could be implemented better, improved or discarded. Some features of the application even required code refactors, because the foundation of the program could not have facilitated them beforehand due to our initial poor understanding of the framework. It is fair to say that a considerable amount of development time was dedicated solely to code refactoring and it very much paid off in the long run. We found that thanks to continuous code maintenance and refactors, the source code became simpler, cleaner, easier to understand and more expandable. Frequent refactoring also strengthened the code's structure, making it less brittle as well as eliminating a sizeable amount of bugs and vulnerabilities the team was struggling with, which could be called a happy accident. A great example of refactoring done by the group are the database refactors described in Datbase Design and Code Architecture. Storing variables the traditional, which is what we have been doing for all our previous assignments just did not work with Razor Pages, thus we needed to shift more to a database mindset. The first 'code smell' that suggested it was a bug with displaying conditional ui messages for the user, they would show up once but after a refresh disappear, simply because they were not stored in a database. Another instance of refactoring in development was handling the output data from the **Optimizer** component. At first every component that wanted to access it had to get the appropriate properties from the object instance itself. This created and issue when it came time to implement downloading this data to the user's device. Since the user needed to be redirected to a different page, we could not pass over the object reference, so a change was made to store the results in the database. With that also came another refactor to the structure of the **Optimizer's** output data, which was simplified and made compatible with storing in the DbSet data structure. This was a prime example of how refactoring made it easier to work with the code and simplified the underlying logic of a component.

Code refactoring was tremendously benefitial in the implementation phase of our application, and we will most definitely keep this practice in the future, maybe even putting more focus on it.

4e Test-Driven Development

Test-Driven Development yapping goes here

4f Unit Testing

Unit Testing yapping goes here

4g Pair Programming

Pair Programming yapping goes here

4h Code Review

Code Review yapping goes here

Chapter 5

Conclusion and Group's Reflections

Conclusion chapter goes here

5a Working on a common project with other groups

5a yapping goes here

5b What went well and not so well with the group's specific set of tasks

5b yapping goes here

5c Specific contributions of each team member

5c yapping goes here

5d Future actions to prevent problems and difficulties faced during the project

5d yapping goes here