

# Syddansk Universitet



## Subdivision of price areas in the Power Market Simulator

DET TEKNISKE FAKULTET

MÆRSK MC-KINNEY MØLLER INSTITUTTET

SOFTWARE ENGINEERING

T510012101

---

### Bachelor Projekt

Start date: 1. February, 2023

Hand in date: 1. June, 2023

---

### Group members

**Patrick Andersen**

patan14@student.sdu.dk

**Danny Hoang-Nguyen Ly**

daly19@student.sdu.dk

**Dan Nguyen**

dangu19@student.sdu.dk

---

### Supervisors

**Bo Nørregaard Jørgensen**

SDU supervisor

bnj@mmmi.sdu.dk

**Flemming Nissen**

SDU co-supervisor

fln@mmmi.sdu.dk



University of  
Southern Denmark

# Abstract

**Background:** Strategirummet is a small Danish company owned by Flemming Nissen. The company has developed a tool, the Power Market Simulator, which can simulate the electricity market all over the world. Currently the PMS has no functionality to divide countries into multiple areas which is a problem, because some countries have their hourly prices calculated as one area, but in reality the areas of the country are not connected electrically which gives inaccurate calculations. The task is therefore to expand the existing PMS with the functionality of being able to divide the country into multiple areas.

**Method:** The method used in the project was Scrum. A sprint was decided to last two weeks. A time schedule was created with the estimated times to complete the features derived from the objectives. The features of the system were prioritized using MoSCoW and added to the Product Backlog. Meetings with the owner were held at the end of every sprint to ensure that the project was progressing in the desired direction.

**Result:** The end result was an extension to the PMS with the functionality to subdivide countries into smaller areas and being able to export the data of the subdivision. A new PMS version was created which can import the subdivision data which allows the user to be able to calculate the prices of electricity of a subdivided area. The analysis, design and implementation of the new system has been documented. Unit tests have been performed and documented. Performance and reliability testing was conducted using JMeter and acceptance testing using scenarios was completed to verify the requirements.

**Conclusion:** The goal of implementing a feature that subdivides an area into multiple areas has been fulfilled. The success criteria, which were defined as objectives and then refined into requirements have mostly been satisfied. The functionality to set transmission capacity and calculating multiple area's prices were deprioritized and not implemented in this iteration of the project because of time constraints. The project can therefore be seen as a medium success, since not all success criteria have been met.

# Glossary

Table 1: Glossary.

| Acronym                  | Definition  |
|--------------------------|---|
| MVC                      | Model View Controller, a design pattern often used in web development.  |
| OOP                      | Object Oriented Programming, a paradigm of how to develop software.   |
| ORM                      | Object Relational Mapping, technique to work with database tables represented as OOP classes.                                     |
| DTO                      | Data Transfer Objects, simple object used for transporting data between components or layers within an application.               |
| UX                       | User Experience, the overall experience the user has with a product.  |
| UI                       | User Interface, the visual elements the user can interact with.   |
| MosCoW                   | Must, Should, Could and Have, prioritizing analysis technique.  |
| UML                      | Unified Modeling Language, a standardized modeling language for representing software systems.                                    |
| CMS                      | Content Management System, software application that allows creation, managing and publishing of digital content on the internet. |
| MVP                      | Minimal Viable Product, includes the core features for a minimum functional working product.                                      |
| Fork                     | The action of creating an independent branch of a software project.   |
| PMS                      | Power Market Simulator, This refers to the existing power market simulator site.  |
| Subdivision editing view | The view for the subdivision site, where it is possible to subdivide areas.   |
| PMS subdivision version  | This version takes input from the subdivision editing view.<br>A separate version of the power market simulator.                  |
| GIS                      | Geographic Information System, computer system that is used for analyzing and displaying geographic information.                  |

Table 2: Glossary.

| Acronym | Definition  |
|---------|---|
| API     | Application Programming Interface, a set of protocols and rules that allows different software applications to communicate with each other. |
| HTTP    | Hypertext Transfer Protocol, application protocol that is used for communication between web browsers and web servers.                      |
| JSON    | JavaScript Object Notation, a data format used for representing structured data.  |
| LINQ    | Language-Integrated Query, a feature from .NET to query and manipulate data from data sources such as databases.                            |
| SQL     | Structured Query Language, programming language for managing and manipulating relational databases.   |
| SOA     | Service-oriented Architecture, an architectural structure that emphasizes the use of services.  |
| Razor   | The view element from Microsoft's ASP.NET framework.  |
| WKT     | Well-known text, is a format used to represent geometry objects in text.  |

# Editorial table

Table 3: Editorial table.

| Section                                       | Responsible | Contributor |
|---|-------------|-------------|
| Introduction                                  | Danny       | All         |
| 1.1 Project goal                              | Dan         | All         |
| 1.2 Methodology, technologies and tools       | Danny       | All         |
| Requirements                                  |             |             |
| 2.1 MoSCoW                                    | Danny       | All         |
| 2.2 Functional requirements                   | All         |             |
| 2.3 Non-functional requirements               | All         |             |
| Analysis                                      |             |             |
| 3.1 The current system                        | Dan         |             |
| 3.2 Layered architecture diagram              | Dan         |             |
| 3.3 Analysis of requirements                  | All         |             |
| 3.4 Use case diagram                          | Patrick     |             |
| 3.5 Detailed Use Case diagrams                | Dan         | Patrick     |
| 3.6 Project Boundary                          | Danny       | All         |
| 3.7 Sprint Planning                           | Danny       | All         |
| 3.8 Summary                                   | Dan         |             |
| Design  |             |             |
| 4.1 Designing for non-functional requirements | Dan         | All         |
| 4.2 The Architecture of the System            | Danny       | Dan         |
| 4.3 System Communication                      | Danny       |             |
| 4.4 Area Subdivision Algorithm                | Patrick     |             |
| 4.5 UX and UI                                 | Dan         |             |
| 4.6 Summary                                   | Dan         |             |

Table 4: Editorial table.

| Section  | Responsible | Contributor |
|--|-------------|-------------|
| Implementation   |             |             |
| 5.1 Display areas in a separate map with division function | Dan         |             |
| 5.2 Divide an existing area into two or more subareas      | Patrick     | Danny       |
| 5.3 Export generated subdivision data                      | Dan         |             |
| 5.4 Import generated subdivision                           | Danny       |             |
| 5.5 A separate version of the PMS                          | Danny       |             |
| Verification   |             |             |
| 6.1 Validation and review                                  | Danny       | All         |
| 6.2 Manual acceptance testing                              | Danny       | All         |
| 6.3 Validation of Performance and reliability              | Danny       |             |
| 6.4 Validation of Modifiability and Maintainability        | Dan         |             |
| 6.5 Validation of Functional requirements                  | Patrick     | Danny       |
| Discussion   |             |             |
| 7.1 The Product Backlog                                    | Danny       |             |
| 7.2 UX and usability                                       | Dan         |             |
| 7.3 Manual testing   | Dan         |             |
| 7.4 Evaluation of Requirements                             | Patrick     | Danny       |
| 7.5 Process evaluation                                     | Danny       | All         |
| 7.6 Future work  | Danny       | Patrick     |
| Conclusion   | Dan         | All         |

# Table of Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                           | <b>10</b> |
| 1.1      | Project goal . . . . .                        | 10        |
| 1.1.1    | Objectives . . . . .                          | 11        |
| 1.2      | Methodology, technologies and tools . . . . . | 11        |
| 1.2.1    | Methods . . . . .                             | 12        |
| 1.2.1.1  | Scrum . . . . .                               | 12        |
| 1.2.1.2  | UML . . . . .                                 | 12        |

---

|          |   |           |
|----------|---|-----------|
| 1.2.1.3  | GitFlow . . . . .   | 12        |
| 1.2.2    | Tools . . . . .   | 13        |
| 1.2.2.1  | Git and Github . . . . .  | 13        |
| 1.2.2.2  | GitHub Projects . . . . .   | 13        |
| 1.2.3    | Technologies . . . . .  | 13        |
| 1.2.3.1  | GeoJSON and Leaflet . . . . .   | 13        |
| 1.2.3.2  | Umbraco . . . . .   | 14        |
| 1.2.3.3  | NET and ASP.NET . . . . .   | 14        |
| 1.2.3.4  | Entity Framework . . . . .  | 14        |
| 1.2.3.5  | Microsoft SQL Server . . . . .  | 14        |
| 1.2.3.6  | JMeter . . . . .  | 15        |
| 1.2.3.7  | Visual Studio . . . . .   | 15        |
| <b>2</b> | <b>Requirements</b>   | <b>16</b> |
| 2.1      | MoSCoW . . . . .  | 16        |
| 2.2      | Functional requirements . . . . .                                     | 17        |
| 2.3      | Non-functional requirements . . . . .                                 | 19        |
| <b>3</b> | <b>Analysis</b>   | <b>20</b> |
| 3.1      | The current system . . . . .  | 20        |
| 3.1.1    | Service-oriented Architecture . . . . .                               | 20        |
| 3.1.2    | Model-View-Controller . . . . .                                       | 20        |
| 3.1.3    | System analysis . . . . .   | 21        |
| 3.1.3.1  | umbracoInSero . . . . .   | 22        |
| 3.1.3.2  | E2G.WebAPI . . . . .  | 22        |
| 3.1.3.3  | E2G.Business . . . . .  | 22        |
| 3.1.3.4  | E2G.Services.EntityFramework . . . . .                                | 22        |
| 3.1.3.5  | E2G.Model . . . . .   | 22        |
| 3.1.3.6  | E2G.Services . . . . .  | 23        |
| 3.2      | Layered architecture diagram . . . . .                                | 23        |
| 3.3      | Analysis of requirements . . . . .                                    | 24        |
| 3.3.1    | Display areas in a separate map with division functionality . . . . . | 24        |
| 3.3.2    | Divide an existing area into two or more subareas. . . . .            | 25        |
| 3.3.2.1  | JSON, DTO and Models . . . . .  | 25        |

---

|          |   |           |
|----------|---|-----------|
| 3.3.2.2  | The database . . . . .  | 25        |
| 3.3.2.3  | GIS Coordinates and GeoJSON . . . . .                                 | 26        |
| 3.3.2.4  | User division cases . . . . .   | 27        |
| 3.3.2.5  | The subdivision module . . . . .                                      | 29        |
| 3.3.3    | Export generated subdivision data . . . . .                           | 29        |
| 3.3.4    | Import generated subdivision data . . . . .                           | 30        |
| 3.3.5    | A separate version of the PMS . . . . .                               | 30        |
| 3.3.5.1  | Integrating data . . . . .  | 30        |
| 3.3.5.2  | Use of existing data . . . . .  | 30        |
| 3.4      | Use case diagram . . . . .  | 31        |
| 3.5      | Detailed Use Case diagrams . . . . .                                  | 32        |
| 3.6      | Project Boundary . . . . .  | 35        |
| 3.7      | Sprint Planning . . . . .   | 35        |
| 3.8      | Summary . . . . .   | 36        |
| <b>4</b> | <b>Design</b>   | <b>37</b> |
| 4.1      | Designing for non-functional requirements . . . . .                   | 37        |
| 4.2      | The Architecture of the System . . . . .                              | 38        |
| 4.2.1    | umbracoInSero . . . . .   | 39        |
| 4.2.2    | E2G.WebAPI . . . . .  | 40        |
| 4.2.3    | E2G.Business . . . . .  | 41        |
| 4.2.4    | PMS.Subdivide . . . . .   | 42        |
| 4.2.5    | E2G.Services.EntityFramework . . . . .                                | 43        |
| 4.2.6    | The Common library . . . . .  | 43        |
| 4.3      | System Communication . . . . .  | 44        |
| 4.4      | Area Subdivision Algorithm . . . . .                                  | 45        |
| 4.5      | UX and UI . . . . .   | 51        |
| 4.6      | Summary . . . . .   | 52        |
| <b>5</b> | <b>Implementation</b>   | <b>53</b> |
| 5.1      | Display areas in a separate map with division functionality . . . . . | 53        |
| 5.1.1    | The system must have a new empty map that can load one or more areas  | 53        |
| 5.1.2    | The user must have the ability to draw 1 polygon on the map . . . . . | 54        |
| 5.1.3    | The user must have the ability to select 1 area in the map . . . . .  | 55        |



---

|          |   |           |
|----------|---|-----------|
| 5.2      | Divide an existing area into two or more subareas. . . . .                  | 56        |
| 5.2.1    | The user must be able to divide an existing area into two or more subareas. | 56        |
| 5.2.1.1  | SubdivideAreas . . . . .  | 56        |
| 5.2.1.2  | multiSolver . . . . .   | 57        |
| 5.2.1.3  | allCutDataFinder . . . . .  | 58        |
| 5.2.1.4  | lineIntersection . . . . .  | 59        |
| 5.2.1.5  | intersectionAdders . . . . .  | 60        |
| 5.2.1.6  | fixCutData . . . . .  | 63        |
| 5.2.1.7  | pointInShape . . . . .  | 63        |
| 5.2.1.8  | noIntersectionSolver . . . . .  | 64        |
| 5.2.1.9  | cutArea . . . . .   | 66        |
| 5.2.1.10 | borderAdders . . . . .  | 68        |
| 5.2.1.11 | loopAdders . . . . .  | 69        |
| 5.2.1.12 | unchangedAdder . . . . .  | 69        |
| 5.2.2    | Subdivided areas should have new labels . . . . .                           | 70        |
| 5.2.3    | Power plants must be distributed to subdivided areas . . . . .              | 71        |
| 5.3      | Export generated subdivision data . . . . .                                 | 76        |
| 5.4      | Import generated subdivision data . . . . .                                 | 77        |
| 5.4.1    | Button to import data . . . . .   | 77        |
| 5.4.2    | Importing data using button . . . . .                                       | 78        |
| 5.5      | A separate version of the PMS . . . . .                                     | 80        |
| 5.5.1    | Integrate with GetAreaPrice . . . . .                                       | 81        |
| 5.5.2    | Integrate with GetAreaChartResult . . . . .                                 | 84        |
| <b>6</b> | <b>Verification</b>   | <b>88</b> |
| 6.1      | Validation and review . . . . .   | 88        |
| 6.2      | Manual acceptance testing . . . . .   | 88        |
| 6.3      | Validation of Performance and reliability . . . . .                         | 89        |
| 6.3.1    | Performance . . . . .   | 90        |
| 6.3.2    | Reliability . . . . .   | 92        |
| 6.4      | Validation of Modifiability and Maintainability . . . . .                   | 93        |
| 6.5      | Validation of Functional requirements . . . . .                             | 94        |
| 6.5.1    | Unit testing . . . . .  | 94        |
| 6.5.1.1  | lineIntersectionTest and distOf2PointsTest . . . . .                        | 95        |

---

|           |  |            |
|-----------|--|------------|
| 6.5.1.2   | noIntersectionSolverTest . . . . .                           | 95         |
| 6.5.1.3   | allCutDataFinderTest . . . . .                               | 96         |
| 6.5.1.4   | allIntersectionAdderMainTest and allIntersectionAdderNewTest | 96         |
| 6.5.1.5   | fixCutDataTest . . . . .                                     | 97         |
| 6.5.1.6   | cutAreaTest and unchangedAdderTest . . . . .                 | 98         |
| 6.5.1.7   | mapPowerPlantsToCorrectCountryTest() . . . . .               | 100        |
| <b>7</b>  | <b>Discussion</b>  | <b>101</b> |
| 7.1       | The Product Backlog . . . . .                                | 101        |
| 7.2       | UX and usability . . . . .                                   | 101        |
| 7.3       | Manual testing . . . . .                                     | 102        |
| 7.4       | Evaluation of Requirements . . . . .                         | 102        |
| 7.4.1     | Functional Requirements . . . . .                            | 102        |
| 7.4.1.1   | More than 2 cuts in one polygon . . . . .                    | 102        |
| 7.4.1.2   | Edge case of drilling . . . . .                              | 103        |
| 7.4.1.3   | Additional edge case . . . . .                               | 103        |
| 7.4.1.4   | GIS coordinates of power plants . . . . .                    | 103        |
| 7.4.2     | Non-Functional Requirements . . . . .                        | 104        |
| 7.5       | Process evaluation . . . . .                                 | 105        |
| 7.6       | Future work . . . . .  | 106        |
| <b>8</b>  | <b>Conclusion</b>  | <b>107</b> |
| <b>9</b>  | <b>Literature list</b>                                       | <b>108</b> |
| <b>10</b> | <b>Appendix</b>  | <b>110</b> |
| 10.1      | Detailed use cases . . . . .                                 | 110        |
| 10.2      | Non-functional requirements . . . . .                        | 116        |
| 10.2.1    | Graphs for GetAreaPrice and Subdivide area . . . . .         | 116        |
| 10.2.2    | Diagrams for the performance test scenarios . . . . .        | 117        |
| 10.2.3    | Diagrams for reliability test scenario . . . . .             | 117        |
| 10.3      | Process evaluation figures . . . . .                         | 118        |
| 10.3.1    | Product Backlog . . . . .                                    | 118        |
| 10.3.2    | Time Schedule . . . . .                                      | 119        |

# 1 Introduction

Strategirummet, owned by Flemming Nissen, is a small Danish company focused around developing tools for the energy sector. The Power Market Simulator is one of these tools. The PMS is a model, which simulates the electricity market all over the world. It bases its data from the French company Enerdata, which contains information on 130.000 power production plants. One of the main functions of the current version of the PMS is the functionality to calculate the hourly prices of the different countries of the world.

In the PMS there is currently no functionality to divide every country into different subdivisions. Only specific countries such as Norway, where the market is prominently divided, have subdivisions.

Not having this functionality is an issue, since some countries such as Indonesia are split into different islands and are not electrically connected. This indirectly results in the electricity facilities in one part of the country supplying power to another part of the country, which should not be possible. This makes the calculations of the hourly prices in PMS, to be imprecise as they show the same price.

Furthermore, in countries such as Germany, there are internal bottlenecks since the transmission connections between certain areas of Germany have a certain capacity which is not taken into account. This means that the calculation of hourly prices of Germany are the same, which is not the actual case because the power facilities in one part of Germany cannot transfer enough electricity to the other part of Germany, thus it should result in different prices for each area. The motivation is to give the user the functionality to divide countries into subdivisions in PMS. This makes it possible for the user to perform calculations such as calculating the hourly prices separately for the subdivided areas, to gain a more precise result. Subdividing the areas makes it possible for the user to set the transmission connections between the different subdivided areas to account for the capacity of the transmission connections.

## 1.1 Project goal

This project will make it possible for users to create subdivisions of areas by letting the user select areas to subdivide. They will then open them in a new map where they can draw polygons on top of the area. This should result in two subareas, one area inside the drawn polygon, and one area outside the polygon. The PMS expansion will find the corresponding units of the subdivided areas by looking at the GIS coordinates of the polygonal borders and

the GIS coordinates of the powerplant units. The subdivided areas and their corresponding power plants can be exported into a file which can be imported into a new version of the PMS in which the user can perform some of the existing features.

Following is a section of success criteria that must be met for the project to be recognized as successful. These criteria have been discussed with the product owner.

### **1.1.1 Objectives**

1. Build a new geographic map that can load selected areas from the existing PMS map.
2. Make it possible for the user to draw polygons in the new map.
3. Implement a feature that divides the area into two subareas, given the original areas GIS coordinates and the coordinates of the user drawn polygon.
4. Implement functionality to distribute power plants to the corresponding subdivided area.
5. Implement a feature to export data from subdivided areas and distributed power plants.
6. Build a new version of the PMS which can take subdivided data and perform existing features of the PMS.
7. Prepare the new version to enable imported new data, which have been exported from objective 5, to be loaded in and integrate with existing data.
8. Prepare the new version to enable users to set transmission capacity between subdivided areas.
9. Prepare the new version for users to be able to calculate hourly prices of subdivided areas using imported data and existing data.
10. Prepare the new version for users to be able to view the charts of subdivided areas using imported data and existing data.

## **1.2 Methodology, technologies and tools**

The following section introduces and describes the methodology, technologies and tools used in the project. The technologies and tools used for the project are the same as the ones already integrated in the existing project for consistency. The methodologies are chosen based on previous experiences by the team members.

### 1.2.1 Methods

#### 1.2.1.1 Scrum

Scrum is an agile framework, which enables a team to deliver value incrementally. It is an empirical process where each iteration during the process increases the quality of the next iteration. Scrum consists of a scrum team that includes a product owner, scrum master and developers. The tasks to be completed within the project span are kept in a product backlog. These tasks consist of the requirements defined in section 2. The schedule is divided into sprints, where a sprint backlog contains the tasks worked on in the current sprint[11]. The team chose to use this framework because it enables agile workflow, and we have previous experience using this framework.

The framework allows the team to work iteratively with the development of the product, which is more flexible in case requirements change.

The duration of each sprint in our project is decided to be 2 weeks. A sprint meeting is held at the end of each sprint, to review the sprint and to discuss and update the sprint backlog. The scrum artifacts for this project can be seen in appendix 10.3.

#### 1.2.1.2 UML

UML is a language syntax used for making models which visualize, specify, construct and document software. UML is tightly connected to OOP concepts such as classes, objects, interfaces, associations and are therefore often used as a basis for implementation.

There are mainly two categories of UML diagrams, dynamic and static.

Dynamic view encapsulates the behavior of the system. It shows the interaction between different entities, flow of data and the passage of time in the system. It is a look into the system during runtime. Sequence diagrams, a type of dynamic view, can be seen in the design chapter 4.

Static diagram model encapsulates the structural aspect of the system. It represents the entities as classes, objects, packages and modules. It shows the relations between these different entities. These types of diagrams are often used to document the software structure. Structural diagrams can be seen as a package diagram 1, layer diagram 2 and a class diagram 8.

#### 1.2.1.3 GitFlow

Gitflow is a branching strategy which involves the use of feature branches and multiple primary branches. Gitflow is the branching strategy that is used throughout the project. The master branch reflects the production environment. The develop branch reflects the staging

environment which is for testing and bug fixing. Feature branches are merged into the develop branch when ready. Once the feature has been tested on the develop branch, they are merged into the master branch and the feature branch is then deleted. The point with Gitflow is having a master branch always ready for production and keeping it up to date [2].

## **1.2.2 Tools**

### **1.2.2.1 Git and Github**

Git is a version control system, which provides the functionality for the team to work together on local copies of the same project in a repository and later integrate the changes together. A server is needed to host the repository.

The team will be using GitHub which will host the repositories for free. GitHub allows users to upload the project to their servers, so the team can get a local version of the source code and make multiple branches to work from the original branch. This allows each of the team members to work on different feature branches, which are later integrated with the original branch by merging. Git keeps track of every change and allows us to revise and revert these changes. The branching strategy we use is described in 1.2.1.3 GitFlow [8].

### **1.2.2.2 GitHub Projects**

GitHub Projects is a tool which integrates with the issues of the project. GitHub Projects is used to create boards which keeps track of the project progress. Multiple boards are created for the Scrum methodology where we have a product backlog, which keeps track of all the requirements and issues to be addressed. The other boards are Sprint Backlogs which are created every 2. week and keeps tracks of the todos for the sprint [10].

## **1.2.3 Technologies**

### **1.2.3.1 GeoJSON and Leaflet**

GeoJSON is a format for working with geometrical data structures. It has a feature object that allows for more complex geometrical structures, which can represent country borders. Leaflet is an open-source JavaScript library for working with interactive maps. It has features that allow the user to add different layers onto the map, and drawing capabilities to draw geometrical objects.

Combining Leaflets features with the formatting of GeoJSON will work as the basis for the frontend and developing an interactive map with subdivision functionality.

### 1.2.3.2 Umbraco

Umbraco is a free and open source content management system that is built on the Microsoft .NET framework with the use of ASP.NET. The CMS is designed to simplify the management of client websites by providing an intuitive and user-friendly interface. One of the key features of Umbraco is its use of Document Type, a container that can be considered as a class where properties can be added to input data. Each property has a data type, such as a string or a number, which makes it easy to manage and manipulate data within the CMS. Umbraco also employs Templates, which are used to output input data. These templates are essentially view pages that can reference the properties defined in the Document Type to retrieve data from the CMS. This allows developers to easily create new websites or modify existing ones, without having to worry about the underlying code [13].

### 1.2.3.3 .NET and ASP.NET

.NET is a software framework by Microsoft that provides a platform for building applications and running them. It includes a set of libraries, tools and runtime environments. One of the frameworks on top of .NET is ASP.NET which helps developers create dynamic web applications, web services, and web APIs using a variety of programming languages that .NET support. We will be using the tool Visual Studio 2022 to develop the new features using .NET [9].

### 1.2.3.4 Entity Framework

Entity Framework is an ORM (Object-Relational Mapping) framework developed by .NET applications. It provides a higher level of abstraction when working with databases, allowing developers to interact with databases using object-oriented concepts. Entity Framework allows developers to create classes that represent database tables and relationships, and to use LINQ (Language-Integrated Query) to query the databases using the classes. Entity Framework also supports database migrations which allow developers to update their database schemas as their application evolves. Entity Framework can reduce the amount of boilerplate code that is needed when working with databases. Entity Framework is used since it is already employed within the existing project [6].

### 1.2.3.5 Microsoft SQL Server

Microsoft SQL server is a relational database system used in this project, since the existing system's persistence is based on Microsoft SQL Server. We will be using the tool Microsoft SQL Server Management Studio to handle the database outside of the system such as setting

up the existing database tables for the environment setup.

#### **1.2.3.6 JMeter**

Apache JMeter is an open-source Java application which offers a set of tools designed to load-test the functional behavior and measure the performance. It can be used to simulate different scenarios and output data in different ways for analysis. It supports various protocols including HTTP which will be utilized to test the new features implemented in the backend. It offers a variety of different graphical records to allow for assessment of the results in terms of listeners [1].

#### **1.2.3.7 Visual Studio**

The existing system is developed with Visual Studio as the IDE and is written in the programming language C#. Microsoft has developed both of these. Visual Studio is therefore the premier IDE for working with C#, since it has a lot of inbuilt support such as code refactoring and debugging tools tailored to C#.



## 2 Requirements

The requirements will encapsulate the major features and functions for the project to meet its objectives. In this following chapter we will go through the functional requirements, non-functional requirements and a MoSCoW analysis. The requirements have been given different levels of priority, such that the group can establish a realistic project plan to complete the most important features and functions.

A project description was given to us, with certain criterias for the project, which was used as a basis for a discussion with the product owner about the requirements. Then internally the team brainstormed the requirements list, which was refined with further discussions with the product owner.

### 2.1 MoSCoW

To prioritize requirements, the project team used the MoSCoW analysis technique, which is an acronym for "Must have", "Should have", "Could have", and "Won't have". This technique employs a hierarchical structure to determine which requirements carry the most weight. The "Must have" priority level represents the minimum viable product, which receives the most attention. As the priority level decreases, the likelihood of a requirement being implemented later on or not at all increases. MoSCoW analysis is an effective tool for prioritizing requirements in a project.

## 2.2 Functional requirements

Table 5: Functional requirements.

| ID       | Description  | MoSCoW      |
|----------|--|-------------|
| R1       | The system must be able to display a user-selected group of areas in a separate map with division functionality.                     | Must have   |
| R1.1     | The user must be able to select 1 or multiple areas in the PMS.  | Must have   |
| R1.2     | The user must have a button to open the selected areas in a new map.   | Must have   |
| R1.3     | The system must have a new empty map that can load one or more areas.  | Must have   |
| R1.3.1   | The user must have the ability to select 1 area in the new map.  | Must have   |
| R1.3.1.1 | When the user selects a second area, the system must unselect the first area.  | Must have   |
| R1.3.2   | The user must have the ability to draw 1 polygon on the new map.   | Must have   |
| R1.3.2.1 | The user could be able to delete the drawn polygon.  | Could have  |
| R1.3.3   | The new map must have a button that calls the division feature.  | Must have   |
| R2       | The user must be able to divide an existing area into two or more subareas.  | Must have   |
| R2.1     | The division feature must only be called when the user has selected 1 area and has drawn 1 polygon.                                  | Must have   |
| R2.2     | The division feature must result in 2 subareas. 1 area with everything inside the drawn polygon, and 1 area with everything outside. | Must have   |
| R2.3     | Subdivided areas should have new labels example: “A1” and “A2” or “A1.1” and “A1.2”  | Should have |
| R2.4     | Power plant units must be distributed to their corresponding subdivided area.  | Must have   |

Table 6: Functional requirements 2.

| ID     | Description  | MoSCoW      |
|--------|--|-------------|
| R3     | The user must be able to export generated subdivision data.  | Must have   |
| R3.1   | The user must have a button to export data.  | Must have   |
| R3.1.1 | The user must be able to download subdivision data using the button.   | Must have   |
| R4     | The user should be able to import generated subdivision data.  | Should have |
| R4.1   | The user should have a button to import data.  | Should have |
| R4.1.1 | The user should be able to import subdivision data using the button.   | Should have |
| R5     | A separate version of the PMS should be created to handle the new subdivided areas.  | Should have |
| R5.1   | The separate PMS should be able to integrate the imported data within the existing system.   | Should have |
| R5.2   | The separate PMS should be able to do the established core features on the subdivided features.                                      | Should have |
| R5.2.1 | The user should be able to set the transmission capacity between a subdivided area to another area (which could also be subdivided). | Should have |
| R5.2.2 | The user should be able to calculate hourly prices on the subdivided areas using imported data and existing data.                    | Should have |
| R5.2.3 | The user should be able to view charts of the subdivided areas using imported data and existing data.                                | Should have |

## 2.3 Non-functional requirements

Non-functional requirements have been established to assure that the system works optimally, does not interfere with the existing system and gives the users a nice experience of using the system. The requirements are defined in table 7 and have been given a MosCoW analysis.

Table 7: Non-functional requirements.

| ID  | Description  | MoSCoW      |
|-----|--|-------------|
| NF1 | Reliability: The subdivision should fulfill 99,5% of its requests based on 10000 requests.   | Must have   |
| NF2 | Performance: The subdivision feature, on average, should run around (5% deviation) as fast as the existing feature for calculating area price (in ms). | Should have |
| NF3 | Modifiability and maintainability: The subdivision feature should have a cyclomatic complexity of <10, depth of inheritance <6 and class coupling <9.  | Should have |

## 3 Analysis

In this chapter, we conduct a system analysis of the existing application to get a better understanding of the pre existing constraints that need to be upheld. A layered architecture diagram is then made to get an overview of the structure.

Afterwards, when the structure and constraints are better understood, use case diagrams and detailed use cases are made to illustrate the user flow for our feature. The end of the chapter will then go over the project boundary and project plan.

### 3.1 The current system

The project is a brownfield software project. This means we are developing an application built on existing technology and infrastructure. Working on a brownfield project means there are gonna be constraints and limitations that have to be considered, before adding a new feature. These constraints are defined in the following section.

#### 3.1.1 Service-oriented Architecture

The current system utilizes a service-oriented structure in combination with a model-view-controller pattern. The SOA lays the foundation for the high-level architecture while the MVC defines how the low-level components are defined through ORM.

SOA is an architecture that uses loosely coupled and reusable services to form an application. These services communicate through interfaces. These interfaces serve as contractual obligations that the specific implementations must uphold. These services are exposed through network protocols [7]. The PMS uses HTTP api to expose these services.

#### 3.1.2 Model-View-Controller

The PMS uses an MVC architectural pattern, which means the application is separated into three components: model, view and controller. This framework is often used for web applications, since it offers a well-defined structure, promotes separations of concerns and enables reusability and modularity.

The model component handles the interaction with the database. Whenever the controller wants data from the database it has to make a request to the model. The model will then retrieve the data from the database and send it back to the controller. Another part of the model is the object relational mapping part, that maps classes to tables in the database [4].

The view component can be simplified as the visual part of what the user interacts with. It generates the UI.

The controller component is the most important part of MVC. It acts as an intermediary between the model and view component. It handles the routing and returns the correct views based on the route. When returning views, the controller can also return data to that view, which it gets by making a request to the model component.

How the MVC is adopted in the PMS will be shown through the following section.

### 3.1.3 System analysis

This chapter is about getting a general high-level overview of how the current system functions, and not a deep dive into implementation details. This package diagram is meant for documentation for someone with no initial knowledge about the application.

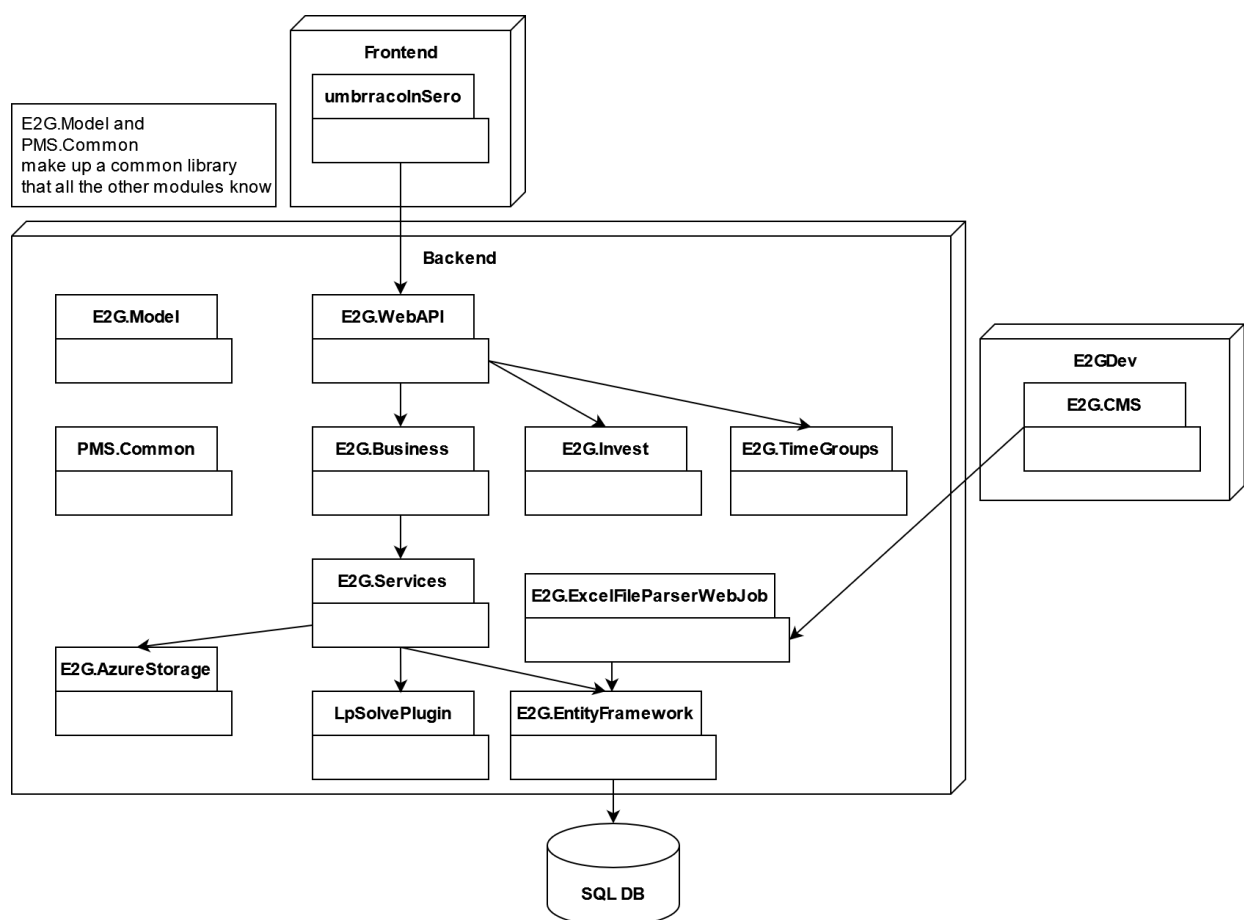


Figure 1: Package diagram.

Figure 1 shows a general overview of the different packages and a high level depiction of how the different packages are structured.

The following section will explain each affected package and their responsibility within the system. Some modules are left out, because they will not affect the subdivision feature and vice versa.

#### **3.1.3.1 umbracoInSero**

This is the front-end portion of the application. The project is set up using ASP.NET framework and using the Umbraco CMS. It contains the views and partial views from Umbraco. Furthermore, it also consists of the JavaScript files that make requests to the Web APIs. In context to MVC, this can be seen as the view portion of the design pattern.

#### **3.1.3.2 E2G.WebAPI**

This project contains the web API controllers. Whenever the client makes HTTP requests, they hit an endpoint within this project. In the context of MVC, this is a portion of the controller component. The Web API controllers use business logic from the E2G.Business package to get data that it needs to return to the views.

#### **3.1.3.3 E2G.Business**

This project contains the business logic that the WebAPI component needs to handle data. It contains a manager class which makes requests to the EntityFramework to get data. It then handles this data and returns it back to the WebAPI component. The separation of server side logic from the web APIs, encapsulates the responsibilities, such that a change in one package will not break the other.

#### **3.1.3.4 E2G.Services.EntityFramework**

Entity Framework is Microsoft's answer to ORM (Object Relational Mapping). This project contains classes that set up the implementation of the DbContext, an interface from the Entity Framework, which is used to interact with the database directly.

#### **3.1.3.5 E2G.Model**

Instead of making SQL statements, the Entity Framework enables us to use C# classes as database table representations that we can interact with instead. The relational modeling happens in the E2G.Model project. In context to MVC, this is the model component together with the E2G.EntityFramework project.

### 3.1.3.6 E2G.Services

This package contains the interfaces of the services. As mentioned in section 3.1.3.1, the services are hidden behind the interfaces, such that whenever a component of the system or another service wants to use the service, they go through the interface.

## 3.2 Layered architecture diagram

Figure 2 outlines the architecture of the affected modules and the relations between them.

The architecture of the system as mentioned in 3.1.3.1 and 3.1.3.2 is a mix between SOA and MVC.

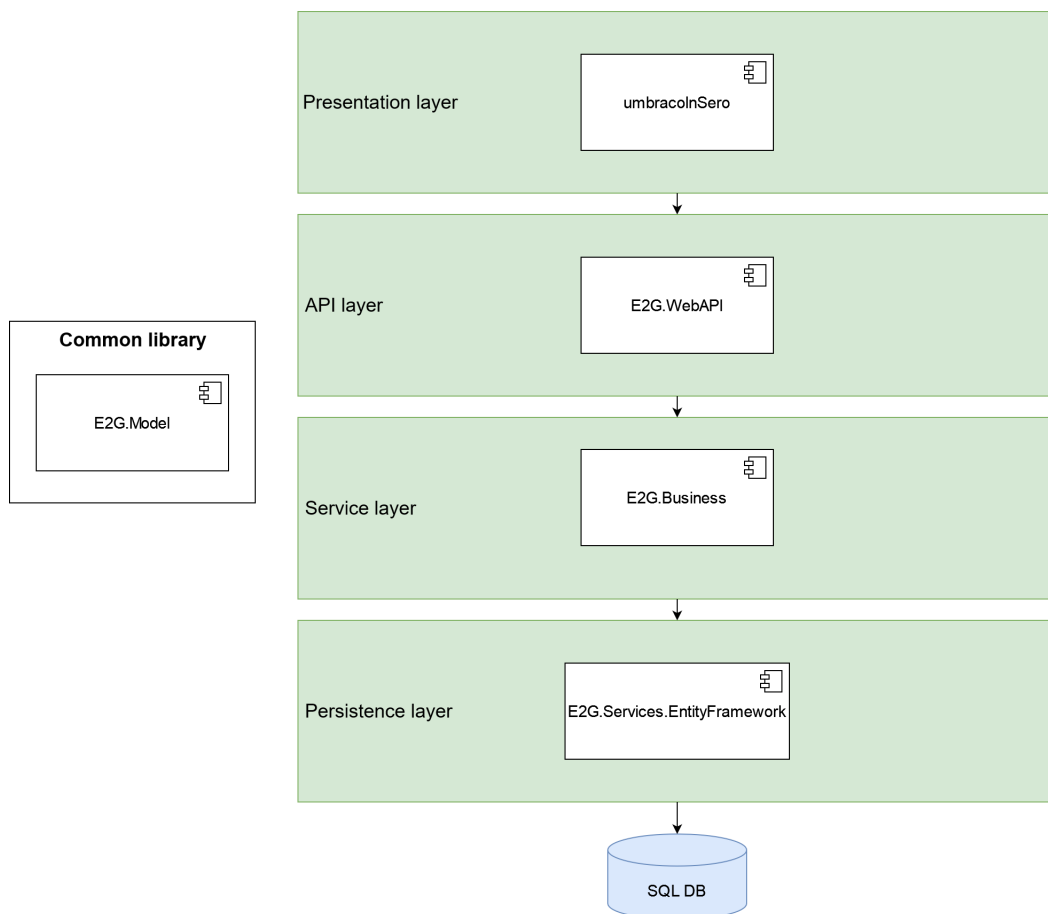


Figure 2: Architecture layer overview.

The SOA is represented in different layers. The architecture is divided into four layers which is the presentation layer, the api layer, the service layer and the persistence layer. The presentation layer shows the UI, which consists of all the views and JavaScript files. The API layer functions as the controller, and manages the services. The service layer contains all the services that the API layer can access. The persistence layer handles the interaction with the database.



The MVC can be seen when using ORM to map classes to database tables, and then using the controller from the E2G.WebAPI module to delegate data between model and view. A more in depth analysis of the structure and how our feature will affect it, can be seen in section 4.2.

### **3.3 Analysis of requirements**

The following sections will go through each main requirement and analyze how to fulfill them. Points such as the current structure of the system and the current technologies used are taken into consideration, together with possible alternatives.

#### **3.3.1 Display areas in a separate map with division functionality**

Fulfilling this requirement means using an interactive map to create the subdivision editing site. When researching different interactive map libraries for JavaScript, there are many different alternatives. The most popular interactive maps all have many of the same functionality with some slight differences. Leaflet is open-source and lightweight, easy to use with a lot of documentation. MapBox, which is built using Leaflet as a foundation, is the frontrunner when working with vector based maps and generally has more functionality. OpenLayers can be used to visualize and analyze geographic data, but is however a bit more complicated without experience working with GIS technologies.

The subdivision map is going to be created using the Leaflet library using the GeoJSON format. This project does not concern vector tiles and geographic data analysis. Leaflet, which is easy to use and lightweight, makes the most sense for this project. Leaflet is also the format that the current PMS is using and therefore it is natural to continue using the same format, rather than using a new map and converting between two different libraries.

The view of the subdivision map should be added through the Umbraco CMS interface. The view has to be made through the Umbraco website, because the current PMS system utilizes Umbraco variables to separate languages. The PMS offers both a Danish and English version. The Umbraco variables can be set to different values depending on which language is selected. The user needs to select areas from the PMS site before redirecting to the subdivision editing site. The GeoJSON, which represents the areas, needs to be sent with the redirection to fulfill R1.3. A more detailed description of the GeoJSON structure can be seen in section 3.3.2.3.

### **3.3.2 Divide an existing area into two or more subareas.**

This section is concerned with the functionality of dividing an existing area into two or more subareas.

Understanding the structure of the system is important before thinking of how to add our features. The link between the frontend and the backend is an API layer named E2G.WebAPI. All the calls from the frontend goes through the API layer which calls the business components E2G.Business, E2G.Invest and E2G.TimeGroup in the backend. Separating our new feature from the rest of the system, a new module PMS.Subdivide will be made in the backend to handle the subdivision logic. The data received from the frontend is in JSON format which is translated to models. In the E2G.WebAPI module, a new controller E2G.SubdivisionController will be created to handle the link between the new business logic in the PMS.Subdivide and the frontend.

#### **3.3.2.1 JSON, DTO and Models**

JSON is the data format used for communication across the system. DTO is an object that is used to encapsulate data and pass it between the layers of the system which in this case is from the backend to the frontend. The DTO does not possess any logic. It simply holds value. In the existing system, the translation from frontend to the backend happens automatically by .NET in the process of serializing and deserializing the data from JSON to models and reversely.

In the existing system, DTOs are used to hold the data that is sent to the frontend by defining a set of properties needed by the frontend. In addition to using DTOs, models are also used. Models have multiple purposes. They are used to reflect the data that is sent from the frontend to the backend and are also used to represent domain specific entities used for business logic. DTOs and models are used throughout the system to cut down the data properties of the models and to restrict the data sent back to the client side.

New DTOs and models might need to be created when implementing the new features of the subdivision to pass and hold important information for the new features in the backend.

#### **3.3.2.2 The database**

The E2G.Services.EntityFramework module handles the database related functionality. E2G.Services.EntityFramework uses the .NET Entity Framework, discussed in section 1.2.3.4, to handle the interaction with the database. In the same module, a class MarketSimRepository.cs is defined. The methods inside the MarketSimRepository.cs all have different data retrieved from the database. One useful method is ListOperatingPowerPlantsByPriceArea()

which will get the power plant units based on the area names and the current slider value year. This method will be used to retrieve the necessary power plant units to distribute them across the subdivided areas which is concerned with requirement R2.4.

### 3.3.2.3 GIS Coordinates and GeoJSON

The PMS currently works using GIS coordinates in a GeoJSON file to represent the countries as polygons. It is therefore a given that the project also uses GIS and GeoJSON.

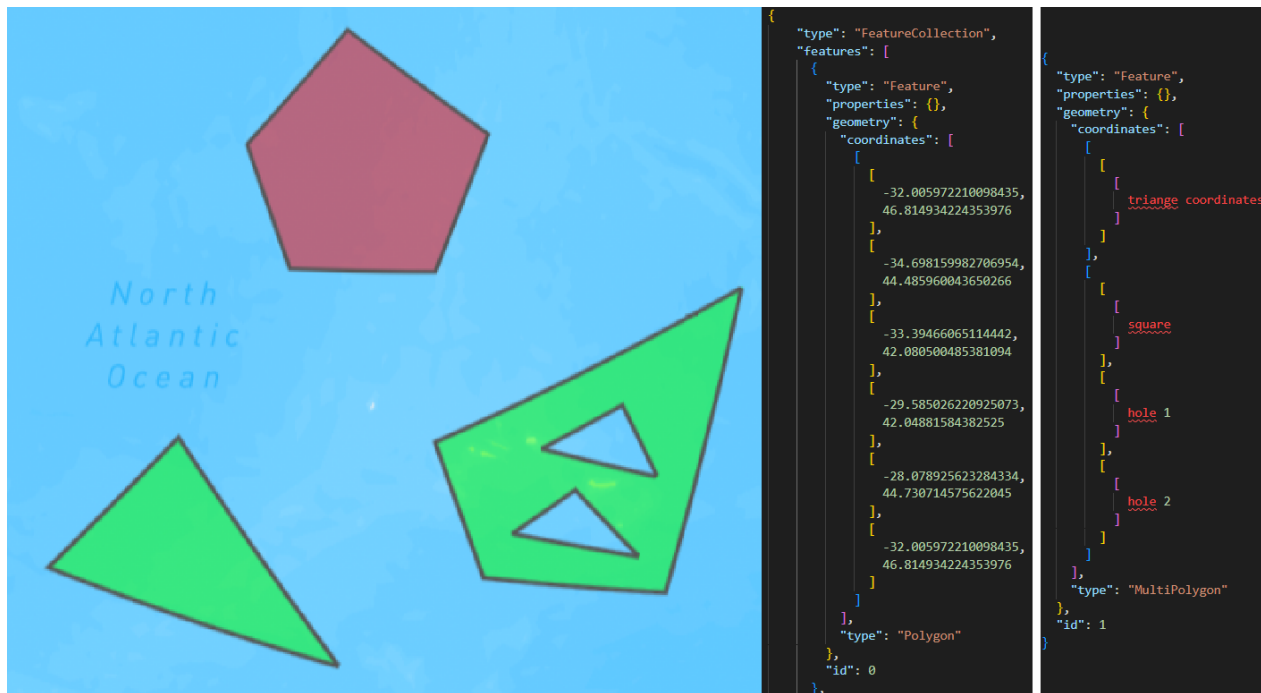


Figure 3: two polygons and coordinates.

There are two types of polygons: Polygon and MultiPolygon. Figure 3 shows a visualization of the two types. The red is a Polygon and green is a MultiPolygon. The left coordinate list creates the Polygon and the right is the simplified version that creates the MultiPolygon.

The Polygon is very simple, it is a single list of coordinates. When the user places a point while drawing, the point is appended to the list. The first and the last coordinate have to be identical to close the Polygon.

The MultiPolygon is similar, but consists of multiple nested lists: a list of areas with a list of Polygons containing a list of coordinates. A MultiPolygon can be one or multiple unconnected “islands”, each island can have zero or multiple “holes” in the island. The list of islands are completely separated, but the inner list consists of first, a list of the outline of the island, followed by a list of coordinates for the holes inside of the first list. This means that the order of islands is not important, but the order of inner lists is very important.

### 3.3.2.4 User division cases

The user needs to be able to freely draw any way they want. It is therefore necessary to prepare the system so that it will work correctly in all cases.

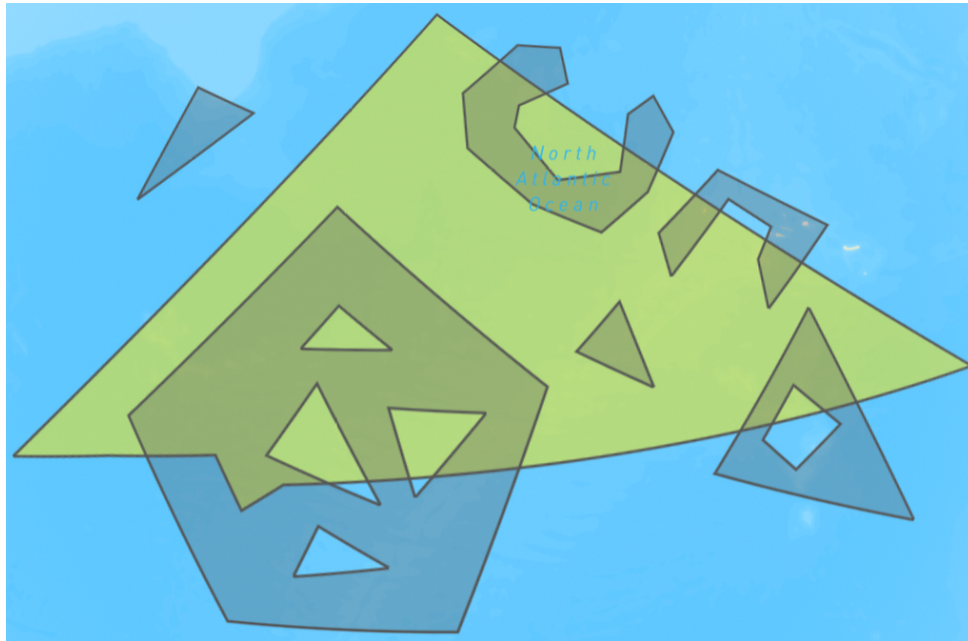


Figure 4: One MultiPolygon and a yellow drawn area.

The PMS currently contains 115 Polygons, like Switzerland and 117 MultiPolygons, like Germany. Polygons should be simple to divide, but figure 4 showcases how it can be much more difficult to divide a MultiPolygon.

In both Multi and Polygons it is necessary to find the intersection between the lines of the drawn area and the original area. While a polygon only needs to find the intersections in one list, the Multipolygon will have to look through all the separate areas list and all of their hole lists. A MultiPolygons area, that does not have any holes, can be treated as a normal polygon. A MultiPolygon with holes will be more complicated, since it will also have to find the intersections in the holes. When the drawn area line cuts through both the outline of a MultiPolygon and one or more holes, this should result in the holes' GIS coordinates being added to the outline list and the hole list disappearing. The holes that do not intersect should remain, but they need to be added to the correct side of the drawn area.

A polygon that has more than 2 intersections will result in at least one MultiPolygon, the two polygons in the north east of figure 4 show how a polygon can be separated into more than one separation.

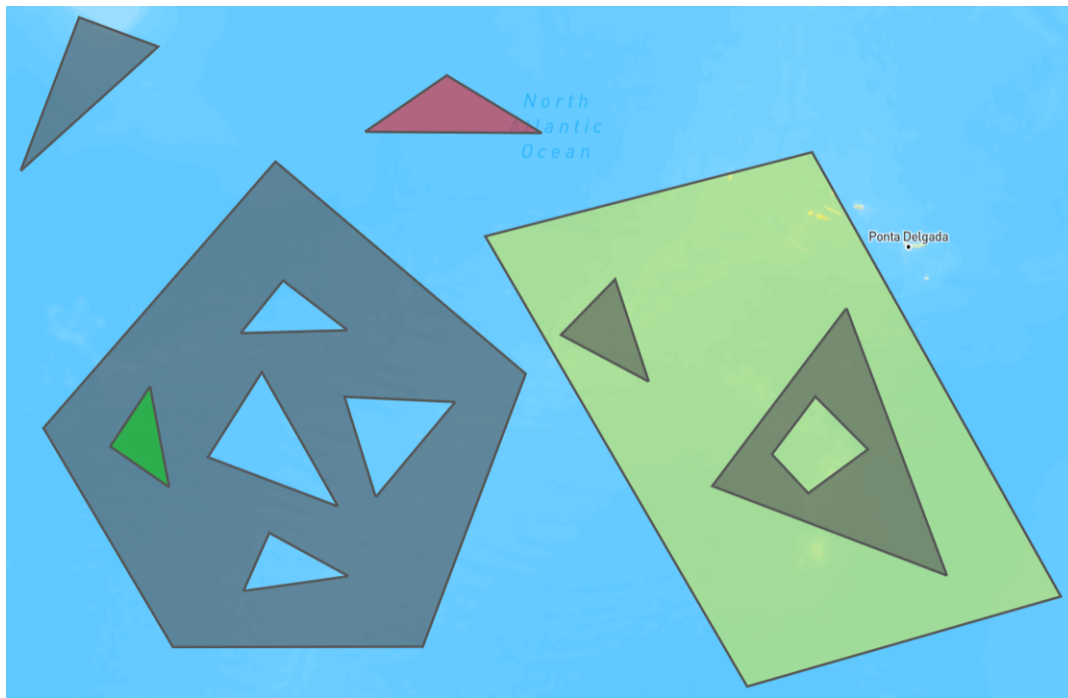


Figure 5: One MultiPolygon and three drawn areas.

The user will be able to draw a drawn area that does not intersect with anything.

The green drawn area, seen in figure 5, will result in a new hole in the correct MultiPolygon area, and a new area that is the same as the drawn area.

The yellow drawn area will split the MultiPolygon into two, one with everything inside the yellow area and one with everything outside.

The red drawn area should do nothing, since nothing is inside of the drawn area. Similarly if the user draws a polygon, where everything is inside the drawn area, the same should happen, since nothing is outside of the drawn area.

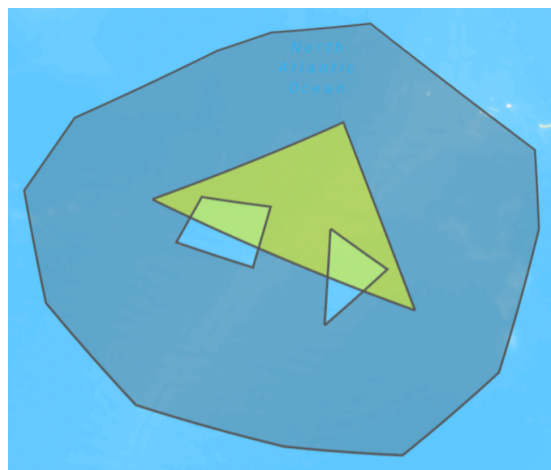


Figure 6: Drawn area only affects holes.

Another specific case can be seen in figure 6. Here there will be no cuts with the outline of the MultiPolygon, however the drawn area cuts both holes, meaning the holes should be combined into one hole, instead of being added to the original MultiPolygon.

### **3.3.2.5 The subdivision module**

Based on the analysis of R2, a module called PMS.Subdivide will be created which will contain 3 of the functional requirements R2, R2.3 and R2.4. The communication between the layers will be in JSON format using DTOs and models. To work with the GeoJSON data in the backend, the NuGet package GeoJSON.Net can be used. GeoJSON.Net is a .NET library for GeoJSON types which makes it easier to serialize and deserialize GeoJSON formats in C#. To satisfy requirement R2, an algorithm will be developed to subdivide the selected area with a polygon since no suitable library could be found for doing so.

The functional requirement R2.3 is concerned with the naming of the new divided areas, which will take the output list from the subdividing algorithm and name the newly divided areas, based on the order of the output list from 1 to 2. Either “.1” or “.2” will be added to the end of the existing name for the subdivided area. Such as DK1 becomes DK1.1 or DE becomes DE.1 and DE.2. The requirement R2.4 is concerned with finding the correct power plant units in the correct subdivided area. All power plants of the system have a set of GIS coordinates which describes where the power plant unit is located. Instead of writing the algorithm from scratch to find collisions between the subdivided features and powerplant, the .NET library GeoLibrary can be used. The library allows one to check whether a point intersects with a Polygon or Multipolygon which can be used to check for collision and distribute the power plant.

### **3.3.3 Export generated subdivision data**

When fulfilling this requirement, the important part is understanding exactly what data to export and what element is gonna receive the data.

The PMS subdivision version is the view that is gonna use the data. It needs two types of data to function. It needs the GeoJSON data of the areas, and also the power plant information corresponding to each area. The GeoJSON data is used to make the Leaflet map, and the powerplant information is needed to use the core functionality of the PMS such as R5.2.1, R5.2.2 and R5.2.3.

### **3.3.4 Import generated subdivision data**

The system currently has functionality to import data, however the data imported is for the configurations of sliders and input values that affect the calculations of the PMS. The new import functionality will import the data mentioned in previous section 3.3.3. Instead of expanding the current import function to be able to take the subdivision data generated from the export, another button and import feature method is implemented to avoid giving too many responsibilities to one button and functionality. Furthermore, by creating a separate button, it can easily be hidden depending on the version of the PMS that the user is currently accessing. The existing functionality of importing data could be reused with some changes when implementing the new import functionality for map data and power plant data.

### **3.3.5 A separate version of the PMS**

The reason the PMS subdivision version has to be made, is because the feature implemented in this project, will become an optional add-on in the future release of the application. The PMS subdivision version is going to use the same Umbraco template of the PMS. An Umbraco template is a blueprint of which multiple Razor views can be made out of. As the two PMS sites use the same Umbraco template, and therefore also the same JavaScript, there needs to be a way to distinguish between the two sites. An Umbraco variable, that is gonna be called “isSubdivisionMode” can be set to true or false depending on which site the user is on. The current application also has a demo mode, in which it uses the same method of distinguishing between different sites.

#### **3.3.5.1 Integrating data**

The imported data section 3.3.4 which consists of map data and power plants will need to be integrated with the system in order to use the data. The map data will be used to replace the map when importing the import file and the power plants will be saved in a list. Using the Umbraco variable “isSubdivisionMode”, it can be checked against which version of the PMS that the user is currently accessing. If the user is on the PMS subdivision version, the imported data will be used if they press the import button.

#### **3.3.5.2 Use of existing data**

The core features R5.2.1, R5.2.2 and R5.3 of the PMS need to be adapted to the PMS subdivision version, since the areas are subdivided. After importing the data, the user should be able to select the imported subdivided area to calculate the area price of the subdivided area and its

delegated power plant units. The user should be able to see the area chart to see the capacity of each power plant in the area and the users can set the transmission capacity between the imported areas. These existing features will need to be adapted to work with the imported data.

### **3.4 Use case diagram**

Figure 7 is a use case diagram showcasing the user, and the use cases that the user can perform. The user is able to select the areas which he wishes to subdivide. The user can draw polygons on the map to create a new subdivision after having selected the areas. When creating a subdivision, the system finds the corresponding power plants across the subdivided areas. The user is then able to import the subdivision data which contains the GeoJSON of the subdivided areas and the corresponding power plants. In the PMS subdivision version, the user can import the subdivision data and perform the three existing features, set the transmission capacity between the uploaded areas, calculate the area prices of the uploaded areas and show the bubble area charts of the uploaded map.





Figure 7: System use case diagram.

### 3.5 Detailed Use Case diagrams

This section contains detailed use case descriptions of the most important use cases. The chosen use cases are “Draw polygons” and “Create subdivisions”. These use cases have been chosen, because without them the use case diagram would not make sense. “Create subdivisions” is the main goal of our project and “Draw polygons” is needed for creating these subdivisions. The rest of the detailed use cases can be seen in appendix 10.1.

Table 8: Detailed use case description for draw polygons.

| Use case             | Draw polygons  |
|----------------------|--|
| Brief description    | The user must be able to draw polygons on a map.   |
| Actors               | User   |
| Secondary actors     |  |
| Preconditions        | <ol style="list-style-type: none"> <li>1. The user must have selected a country from the PMS map.</li> <li>2. The user must have pressed subdivide area button. <ol style="list-style-type: none"> <li>a. This routes the user to the next site.</li> </ol> </li> </ol>  |
| Flow of events       | <ol style="list-style-type: none"> <li>1. The user presses a button that symbolizes a polygon. The user is now in a draw state. While in the drawing state they can no longer select areas, but instead every click adds a point.</li> <li>2. The user presses somewhere on the map to add a point.</li> <li>3. The user adds two or more points.</li> <li>4. The user clicks a finish button. <ol style="list-style-type: none"> <li>a. The points connect and form a polygon.</li> </ol> </li> <li>5. The user is no longer in a drawing state.</li> </ol> |
| Postconditions       | A new GeoJSON feature has been added to the map.   |
| Alternative scenario | <ol style="list-style-type: none"> <li>3a. The user clicks a finish button. <ol style="list-style-type: none"> <li>a. Nothing happens. The finish can not connect points since there are less than 3 points.</li> </ol> </li> <li>3b. The user adds one more point.</li> <li>4b. The user clicks a finish button. <ol style="list-style-type: none"> <li>a. Nothing happens. The finish can not connect points since there are less than 3 points.</li> </ol> </li> </ol>  |

Table 9: Detailed use case description for create subdivision.

| Use case             | Create subdivision   |
|----------------------|--|
| Brief description    | The user must be able to divide a country into one or more subareas.   |
| Actors               | User   |
| Secondary actors     |  |
| Preconditions        | <ol style="list-style-type: none"> <li>1. The user must have selected a country from the PMS map.</li> <li>2. The user must have pressed subdivide area. <ol style="list-style-type: none"> <li>a. This routes the user to the next site.</li> </ol> </li> </ol>   |
| Flow of events       | <ol style="list-style-type: none"> <li>1. The user has drawn an area (reference use case: “Draw polygons”)</li> <li>2. The user selected an area (reference use case: “Select areas”)</li> <li>3. The user presses the subdivide button.</li> <li>4. The system creates 2 new subareas, everything inside the polygonal area and everything outside.</li> <li>5. The system finds all power plants inside the areas.</li> <li>6. Labels each area as mentioned.</li> </ol>   |
| Postconditions       | One area has become two subdivided areas.  |
| Alternative scenario | <ol style="list-style-type: none"> <li>1a. The user has drawn nothing.</li> <li>2a. The user selected an area.</li> <li>3a. The user clicks the subdivision button. <ol style="list-style-type: none"> <li>a. Nothing occurs since no polygon has been drawn, the same map will be loaded in.</li> </ol> </li> <li>2b. The user has not selected an area.</li> <li>3b. The user presses the subdivision button. <ol style="list-style-type: none"> <li>a. Nothing occurs since no area has been selected for subdivision.</li> </ol> </li> </ol> |

### 3.6 Project Boundary

The primary focus of the project is to extend the existing PMS by introducing a new map component to subdivide selected areas and additional features. The important requirements for the features were elicited in the requirements section and are prioritized with “Must have”, using the MoSCoW analysis as they constitute the minimal viable product, seen in section 2.2. The MVP consists of the user being able to select countries to subdivide, and afterwards being able to export the subdivided areas and the power plants data to a local file.

The other requirements that are marked with “Should have” involve modifications to the existing system of the PMS to integrate the subdivision data and are implemented if time allows it.

The group will add the new features iteratively to a fork of the current version of the application. When the product is finished the forked project can be merged back into the original repository.

The deadline of the product and the report must be completed within the 1. June, 2023. The group has considered this time constraint while prioritizing the requirements, since some requirements may not be met before the final deadline. These considerations can be seen in section 2.1.

The group consists of three team members and the methodology used is Scrum, where each sprint is decided to last two weeks. Each sprint ends with a stakeholder meeting with the product owner and an internal sprint review between the group members. The sprints will mainly focus on development of the product while also writing notes to the report alongside. In the last month, the focus is on verifying and documenting the implementations and finishing the report.

### 3.7 Sprint Planning

Based on the analysis, we have created a gantt diagram and estimated the number of sprints needed to complete the MVP from the requirements in the product backlog refer to product backlog 10.3.1.

The schedule is structured with two sprint weeks after each other and the official work days are every Wednesday and Friday as the team has other activities running in parallel with the project. However, the members of the team are free to work on the project whenever it is desired.

The requirements are implemented in order based on their MoSCoW priority while considering

the use case diagram. For example, it makes sense to implement selecting a country to subdivide before being able to subdivide.

On the official work days we conduct sprint meetings to discuss and catch up on the progress, and use the time to update the sprint backlog and the product backlog. At the end of a two week sprint, we discuss the results of the sprint and take notes and discuss which items of the product backlog should be implemented in the following sprint. If we are behind on schedule, we add more official work days in the following sprint.

The gantt diagram can be seen in appendix 10.3.2

### **3.8 Summary**

The project is brownfield software development, meaning the development is about developing on top of an existing system.

Therefore, the analysis for this project focused mostly on understanding how the existing works, and the structure of the system. The structure of the system can be seen in the system analysis 3.1 and layered architecture diagram 2. The functionality of subdivision must coexist with the existing functionality without breaking the system. Afterwards the requirements were analyzed in depth to understand how to fulfill them within the constraints. Use case diagrams and detailed use cases were made to understand the user journey better.

## 4 Design

In this chapter, we define the architecture for our software solution extending the existing system architecture. The analysis chapter defined an overview of the structure and some constraints, such as how data flowed through the system and which technologies were used. These considerations are taken into account in the design chapter. This chapter will include a more detailed diagram of the architecture design and a detailed description of the components. Lastly, it will contain sequence diagrams describing the flow of the different features.

### 4.1 Designing for non-functional requirements

Considering the design's direct influence on the system's quality, it becomes crucial to bear in mind the non-functional requirements during the system's design phase.

NF1 is about the reliability of the system. The system was designed for reliability by considering the design principle, separation of concerns. The principle helps keep the system modular, which results in isolation of possible errors in the system and thereby minimizing the impact of the errors.

NF2 is about the performance of the system. When designing for performance, it was considered how the data flows through the current system and tried to maintain this flow with our feature. It goes from a JavaScript controller that makes an API call, which hits an endpoint from the web API. The web API leverages the business logic of a component within the business layer and subsequently invokes the repository. The repository makes direct contact with the database and returns the data. When we keep the same data flow, it means if a performance issue occurs, it will most likely stem from the business component, where all of the logic is encapsulated. Using tools such as JMeter helps identify if there are performance issues. JMeter data can be seen in verification section 6.3.

NF3 is about the modifiability and maintainability of the system. Changes occur all the time when working with software. Adding new features, removing old ones and updating libraries are some examples of the evolving nature of software. Therefore it is important we consider tactics for increasing the modifiability of our added feature. We need to encapsulate the subdivision functionality inside its own module to reduce the coupling and increase the cohesion of the system. Having all the functionality regarding subdivision in its own module, means whenever a change is being made to the subdivision feature, there will be a lower chance of it affecting multiple modules. It also makes it easier to detect possible faults for subdivision, since the



the `umbracoInSero` component.

`PMS.Subdivide` is going to contain our business logic on how to subdivide the areas. The `SubdivideController.cs` is gonna use this business logic, and the established repositories, which are implementations that build upon the `DbContext` from the `EntityFramework`, to retrieve power plant data.

The subdivide component handles the responsibility of the points made in section 3.3.2.4. The controller will use the business logic from the subdivision component to subdivide the areas and delegate the corresponding power plants to each area. Afterwards the controller sends the data back to the views. The JavaScript controllers for the views, as mentioned in section 3.3.1 and 3.3.5, will use the data to instantiate the Leaflet map.

The design diagram has given an overview of the structural aspect of the system, which gives us enough detail for implementation. In the next section, a further description of each component is going to be detailed to better understand how our feature affects them.

#### 4.2.1 `umbracoInSero`

This component is inside the presentation layer and contains all the front-end of the project. Meaning all the JavaScript and Razor views are defined here. The structure of the JavaScript is object-oriented. The PMS site has a main JavaScript controller called `MarketSimController.js`, which instantiates the other JavaScript objects such as `MarketSimMenuController.js`, that handles instanting the menu part of the view, and `MarketSimMapController.js`, that manages and instantiates the map.

This structure makes sense, since it encapsulates the responsibilities to each class, instead of having one big JavaScript file that handles all the client side logic.

A new JavaScript, called `MarketSimSubdivideController.js`, needs to be made for the subdivision editing site. This script is not made in the same way as the others. The `MarketSimSubdivideController.js` controller is only made for the subdivision editing site. Therefore, it does not need to be object-oriented since the main controller `MarketSimController.js` does not need it. The responsibility of the `MarketSimSubdivideController.js` deals with all of the map requirements about the separate subdivision site. These requirements are from R1.3 to R1.3.3. It also handles the export function mentioned in R3.

The views are established, but there needs to be more analysis to understand how to get from the PMS to the subdivision editing site. This redirection will be shown in the following



sequence diagram.

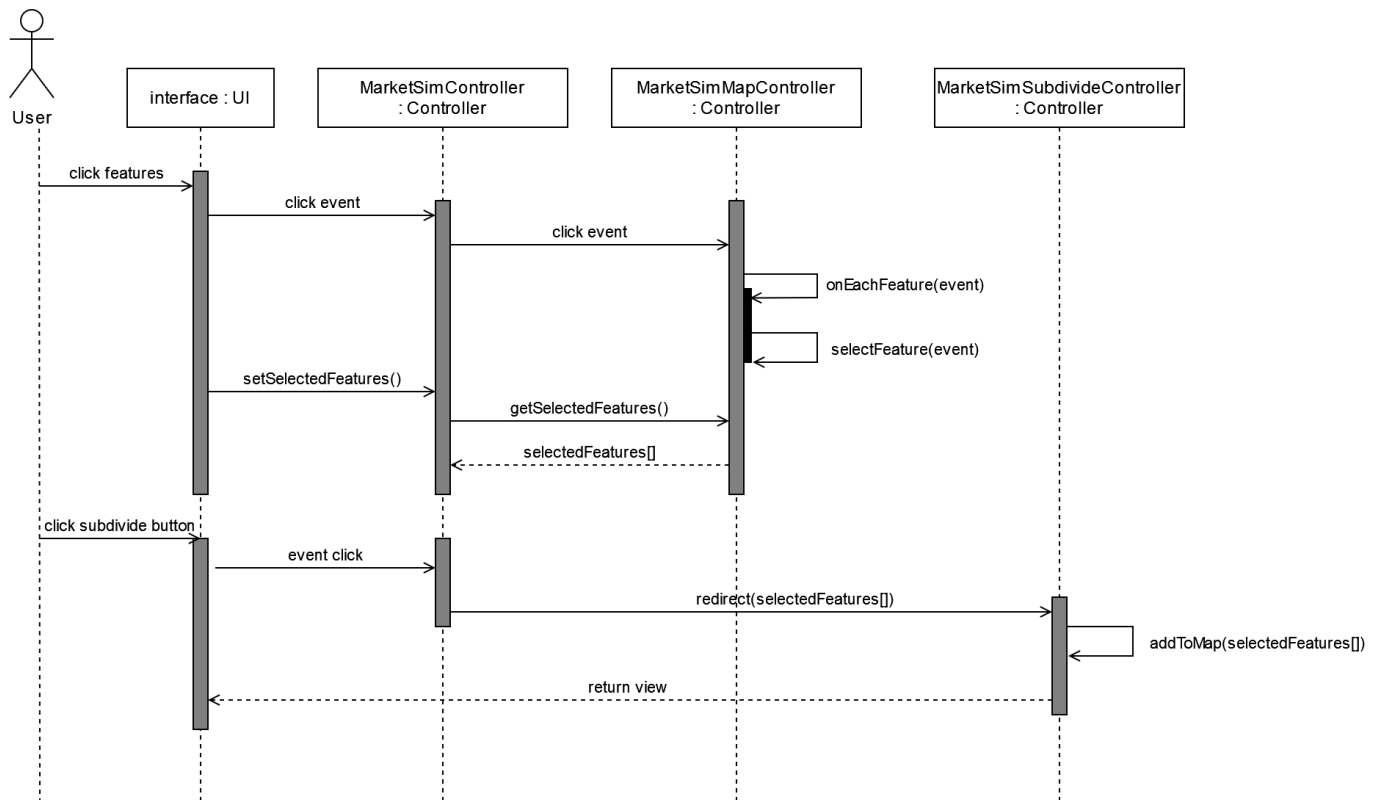


Figure 9: Sequence diagram.

Fig 9 shows the interactions that lead to a separate map with an user-selected group of areas. As mentioned in analysis 3.3.1, the user must select areas from the PMS site and then click on a button to redirect to the subdivision editing site. When selecting areas, a click event occurs that calls a method in the MarketSimMapController. This method marks a feature as being selected. When the user then clicks the redirect button, they will be redirected to the subdivision editing site together with the selected area data.

#### 4.2.2 E2G.WebAPI

This component resides in the API layer of the system. It contains controllers, which handle the requests from the frontend. E2G.WebAPI is responsible for handling the incoming request and generating an appropriate response back to the frontend. There are currently four controllers which are AdminController.cs, EnergyStreamSimulatorController.cs, MarketSimulatorController.cs and PMSController. Each API controller has different responsibilities in terms of features and each is dependent on the different modules of the backend containing business logic.

Another controller will be added for the implementation of the requirements to keep the sepa-

ration of concerns. The controller will be named SubdivisionController and is only concerned with the features implemented in the project which is subdividing polygons R2, naming subdivided areas R2.3 and distributing power plants R2.4.

The frontend will send requests to the SubdivisionController's endpoints with the necessary data to perform the business functionality implemented in this project. More about the communication in the system and a sequence diagram is presented in section 4.2.

### **4.2.3 E2G.Business**

This component resides in the service layer. This module contains the majority of the business logic in the system. The features that will affect this module is the main requirement R5 and its sub-requirements. Requirement R5 is concerned with integrating the data imported and being able to perform existing features. The imported data will be integrated by checking if it is the PMS subdivision version that is currently being accessed. If it is the PMS subdivision version, the imported data should be used when performing some of the existing features such as requirement R5.2.2. This is the same way the system currently checks what functionality of the backend should be activated based on whether the calls are from the demo version or not. This flow is described by the sequence diagram on figure 10.

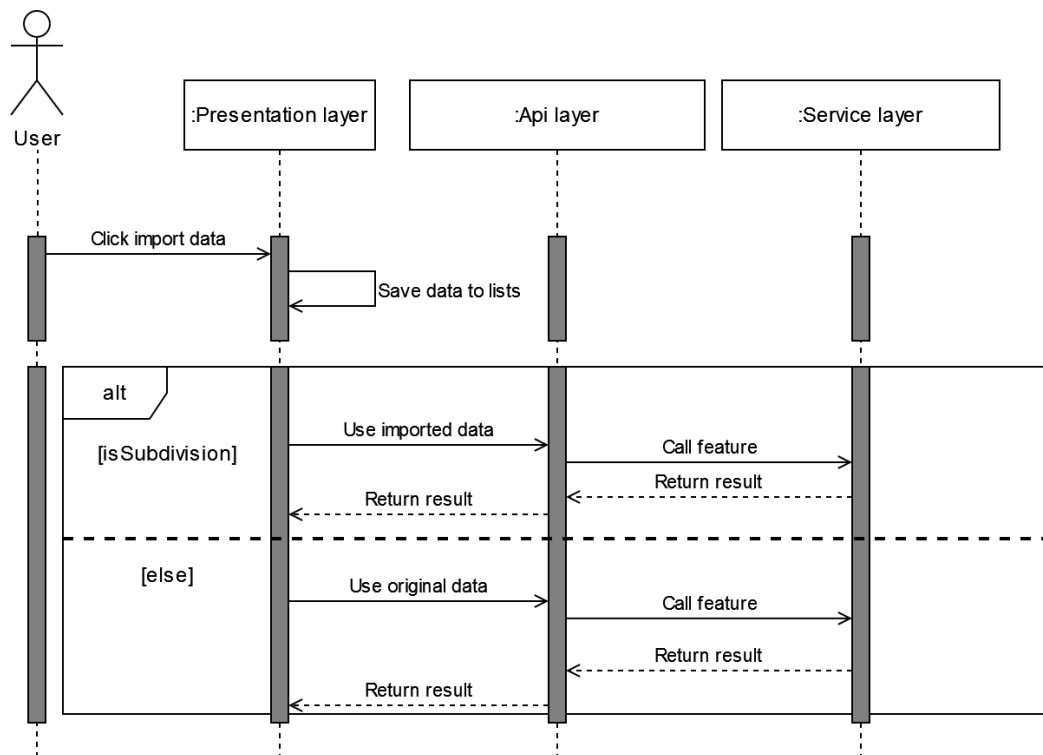


Figure 10: Sequence diagram 2.

Figure 10 shows that the user interacts with the click import data and imports the data. The imported data is saved to a list in the presentation layer.

Then the next lifeline shows if the user is on the PMS subdivision version which is indicated by a boolean “IsSubdivision”, then the API controller in the API layer will use the saved data in the presentation layer which was imported from an subdivision configuration file. The API layer will make calls to the service layer, where the existing feature resides and pass the imported data. If the user is accessing the original version, it will use the original implementation and data.

#### 4.2.4 PMS.Subdivide

This module will be an extension to the system. It will contain the algorithm for dividing the areas using polygons, the logic for distributing plants across the subdivided areas and the naming of the subdivided areas. It will need to have a dependency on the E2G.Model component which contains business entity objects. Furthermore, the PMS.Subdivide will import the two NuGet libraries GeonJSON.Net and GeoLibrary, which will be used for the implementation of the requirements. The communication between the layers is described in section 4.2.

#### 4.2.5 E2G.Services.EntityFramework

This module resides on the persistence layer as it handles the communication with the database. This database will need to be queried in order to get the power plant units to distribute them across the subdivided areas.

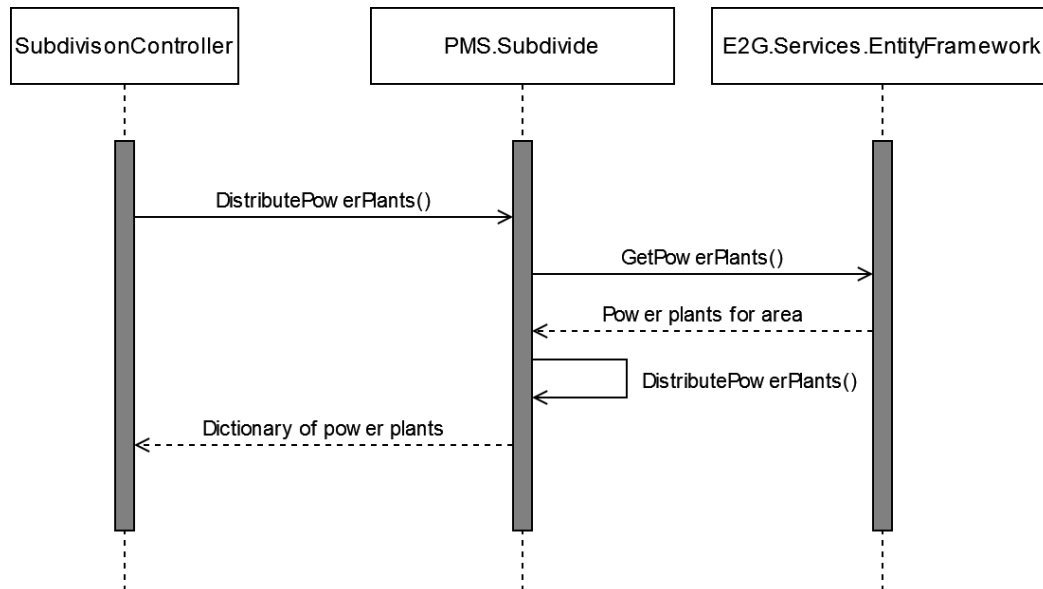


Figure 11: Sequence diagram 3.

Figure 11 shows how the controller uses the business logic from the Subdivide component. The subdivided component needs to make a call to the EntityFramework which is the only component that has direct interaction with the database. Afterwards, the data is eventually returned to the controller.

#### 4.2.6 The Common library

The common library layer consists of the modules E2G.Common and E2G.Model. Both of these components contain business entities which are used to perform the business logic in the different services of the backend. The business model that will be used is SystemPowerPlant.cs and PriceAreasDictionary.cs which are both found in the E2G.Model module. SystemPowerPlant.cs contains all the necessary data properties needed to perform the business logic and is needed when handling the distribution of the power plants across the subdivided areas.

PriceAreaDictionary.cs is a singleton instance of all the known areas of the PMS. This dictionary will have to be expanded, when introducing the subdivided areas as it contains the name of the areas with their ID as the values.

### 4.3 System Communication

The communication between the client side and server side in the existing system happens through sending JSON with HTTP POST requests. When the user performs an action that requires processing in the backend, the frontend will make a request to the appropriate endpoint in one of the controllers in the backend. Since the HTTP method is POST a message body in JSON format is sent together with the request to the backend. When the endpoint is hit, an appropriate controller method receives the data in JSON format which is deserialized into a server side object. This happens automatically using the inbuilt namespace “System.Net.Http” in ASP.NET, if the controller methods use models that match with the JSON data that is expected to be received from the frontend.

The API layer processes the data from the request and makes calls to the corresponding services in the backend. After processing the data in the services, the API layer generates a response and sends back relevant data to the client side with a HTTP status code. As of section 4.2.2, the subdivision API controller will receive HTTP POST requests when the user interacts with a UI component in the frontend. The frontend will transmit the data to the backend through the message body, to perform business domain logic with the data, which is sent back as a response to the frontend for the user to see.

The following sequence diagram describes how the general flow of data traces through the 4 layers of the system.

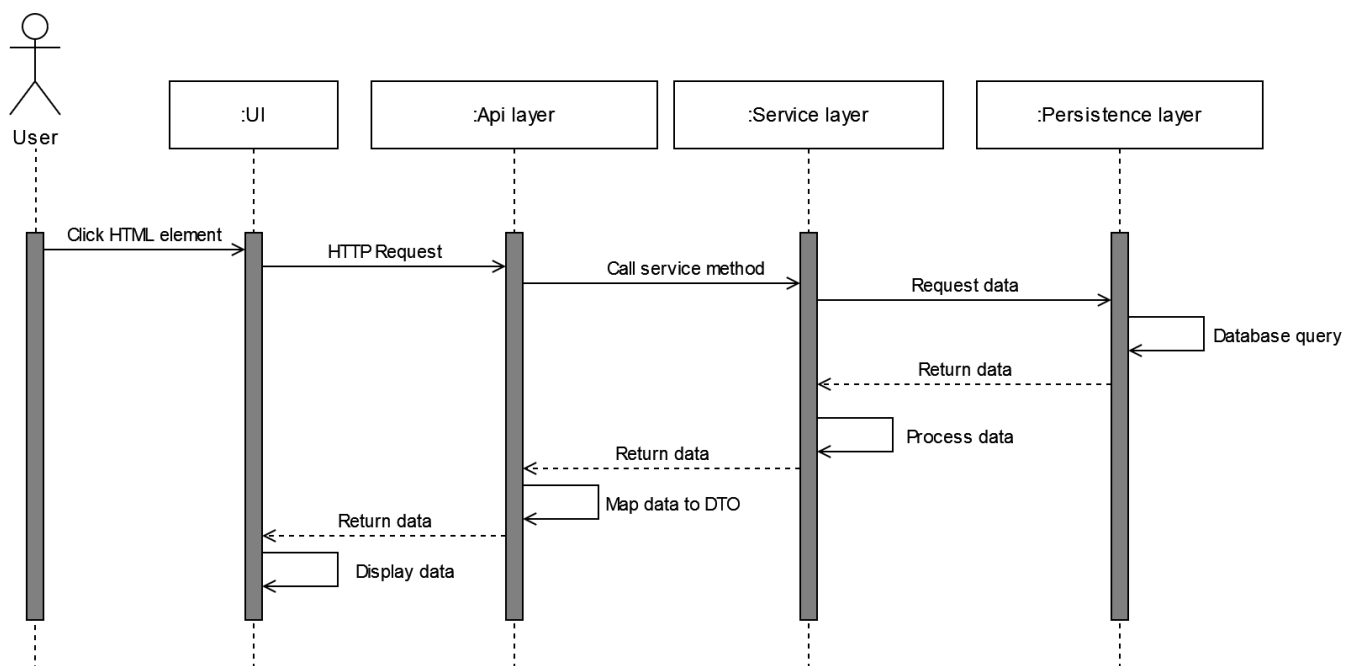


Figure 12: Sequence diagram 4.

## 4.4 Area Subdivision Algorithm

The main part of the business logic consists of dividing one area into two areas. This has multiple cases with multiple solutions. It is therefore necessary to divide these solutions into separate methods that will have separate sequence diagrams.

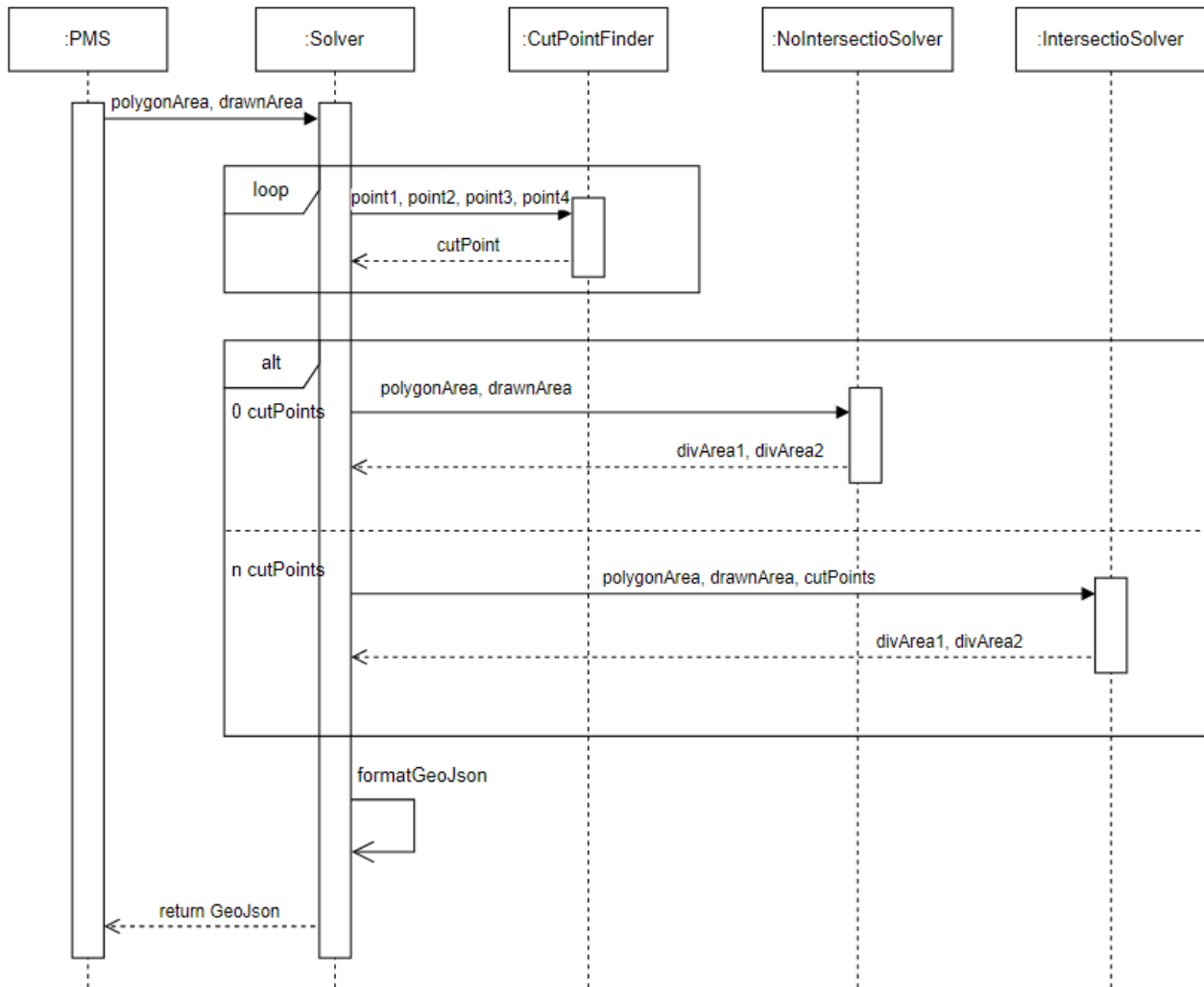


Figure 13: Sequence diagram 5.

The diagram of figure 13 gives a quick overview of how the division solver operates. The PMS delivers the polygons to be used in the division. The solver then looks for cut points and delegates the work to the solver that is best suited for the given case.

As the analysis 3.3.2.4 describes, the user will be able to create division with or without having intersections between the original polygon and the drawn area polygon. It should be easy to determine which of the two cases is being used, by finding all the intersections and saving them. If the intersection count is zero or more, it will determine the case.

An intersection can be found by calculating the Line-Line intersection. For that it will be necessary to calculate the equation of the lines. This means that for every two adjacent GIS coordinates in the area's list of coordinates, it will be necessary to calculate the line intersection with every two adjacent GIS coordinates in the drawn polygons list of coordinates. The Line-Line intersection calculation will always find an intersection, unless the lines are parallel, so it is necessary to check if the intersection lies within the two given lines.

As the analysis 3.3.2.4 describes, there are three cases where a user can draw a subdivision without creating any intersections: 1. an error, 2. separating islands, 3. cutting a hole. Once it is determined that there are no intersections, it will be necessary to determine the case. This will be doable by using collision detection. Given that there are no intersections, it is known that every polygon will be fully inside or outside the cutting polygon, it is therefore only necessary to check collision of one point for each polygon. If a point is colliding with the drawn area, it will not be the 3. case. The polygons that collide will be stored in a list, and those that do not collide will be stored in another list. If one of the lists is empty, it will be the 1. case, since there is nothing to separate. If there is something in each list, the 2. case will be processed. Since the areas already are in separated lists, it is only necessary to format the two lists into a GeoJSON file. If none of the polygons appear inside the drawn area, that means the drawn area might collide inside a polygon, which is the 3. case. To process the case, the drawn area is added to a new list and the polygons hole list. The two lists are then formatet into a GeoJSON file.

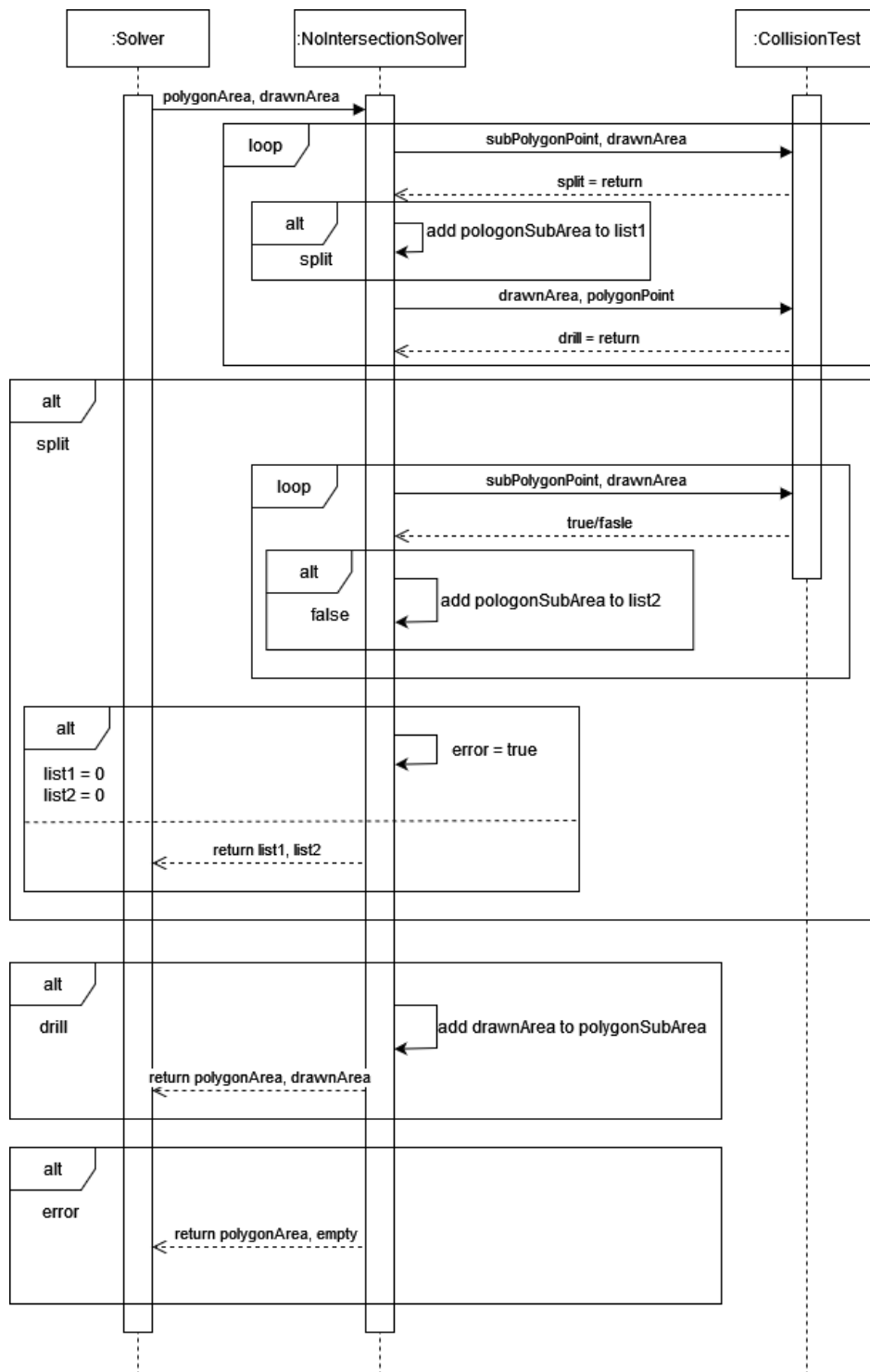


Figure 14: Sequence diagram 6.

The diagram in figure 14 shows how the three cases from figure 5 in analysis 3.3.2.4 would be solved. These different cases are shown as alternative scenarios named split, drill and error. At first, the collision test is utilized to determine how the polygon area and the drawn area are placed in relation to each other. Something to note is that both split and drill can be true at the same time. If that is the case, it will not reach the drill case, given that the return in split halts the method. In the split case, the output will be stored in two separate lists. If one of



them is empty, the case is changed to the error case. The drill case adds the drawn area to the original polygon list, and returns both separately. The error case returns the polygon list and an empty list to indicate that nothing has changed.

If intersections are found, it will be necessary to split a polygon into two parts. To do so, it will be beneficial to save a lot of data about the intersections.

The intersections are found by using the equation of the lines. The issue here is that all lines that are not parallel have an intersection since the math assumes that the lines are infinitely long.

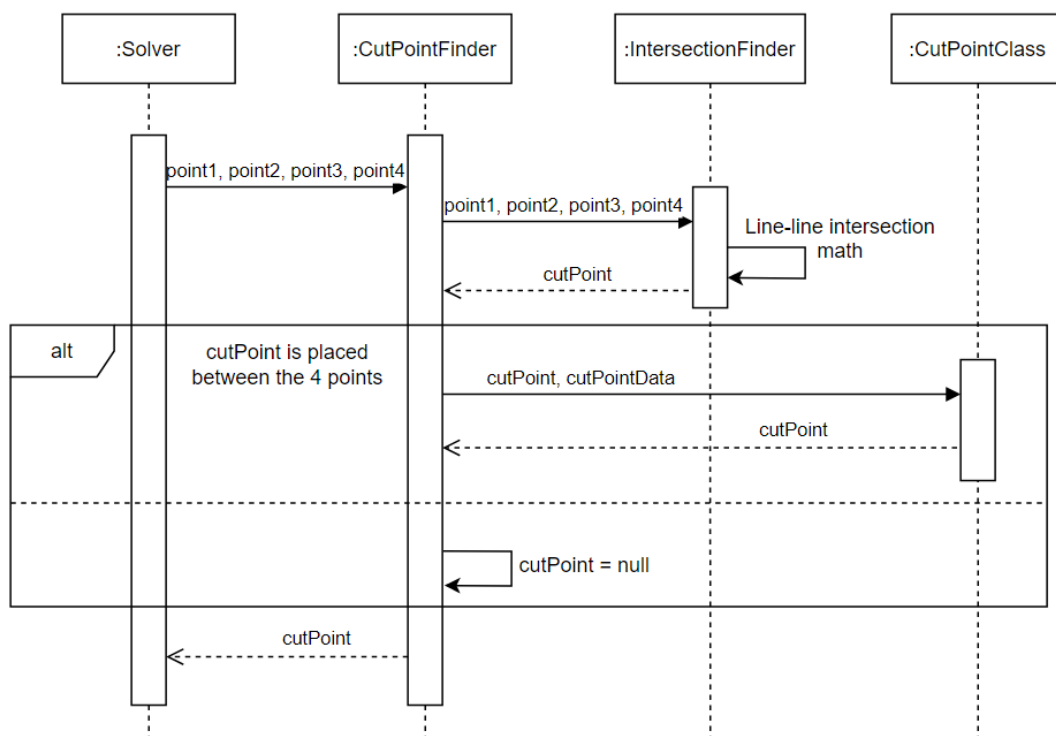


Figure 15: Sequence diagram 7.

The diagram in figure 15 shows how the `CutPointFinder` finds the cut point and checks if the point falls within the given points. If it does, it creates a new `CutPoint` using the data class, if not the cut point is set to null. The cut point is then returned to the solver.

The `CutPoint` class is designed so that the system does not need to traverse the polygon lists multiple times to find where the cut points are and how they are associated. Once the cut points are found, they will be added to the polygons lists. The points will be added using their found pre coordinate. This is a problem, if 2 cut points have the same pre coordinate. In figure

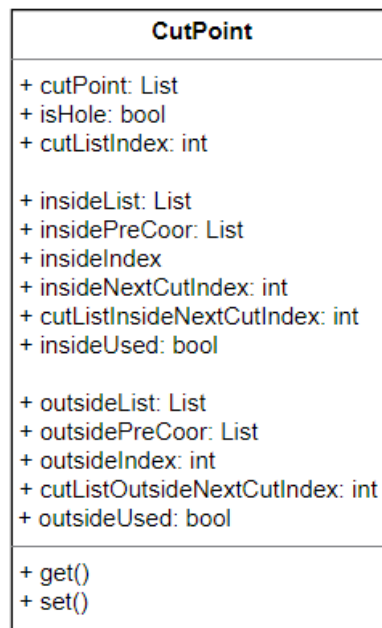


Figure 16: Cut point class diagram.

17 the points 2, 3, 4, 5, 6, 7, 8, 9 and 10 will all have the same pre coordinate for the drawn polygon. When searching for the points, the algorithm will first find all points in the polygon outline and find cut 1 and 6, before moving to the holes. There is no way to determine what hole is the first to be traversed or where the starting point is, so the outcome might be that the first point found is 5 and then 4, before moving to 2 and 3. The result would be 1, 6, 5, 4, 2 and 3, which is not correct. To solve the problem, all points with the same pre coordinate will have their distance to the pre coordinate calculated using the distance formula. Then they will be sorted after so that the closest comes first and the furthest away comes last.

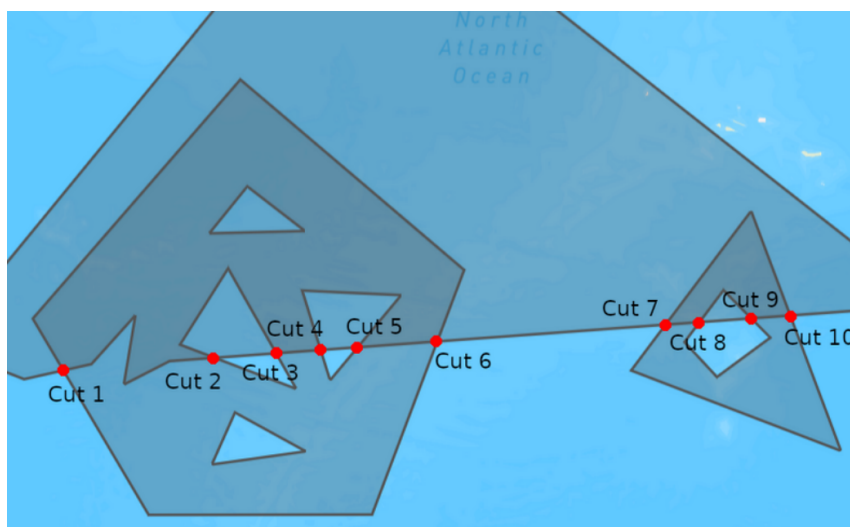


Figure 17: Cut point example.

In the example in figure 17, the polygons will be cut in two. This will be done by traversing a polygon list and adding the points into two new lists. The points will be distributed to each list by determining whether they are inside or outside the drawn polygon. The points that appear on the drawn cut point list should be added to both lists. After all points in the polygon have been used, the algorithm will move to the next polygon.

There is no way to determine where a polygon list starts, and what direction it has been drawn in. Therefore, a cut point will be used as the starting point when traversing the lists. Once the starting point is re-found, all points should have been traversed. All the cut points will be saved in a list, to make them easily accessible. If the first cut is cut 1, the algorithm will bypass cut 2, 3, 4, 5 and 6, when doing so, they should be marked as used, that way it will be possible to use the cut point list to only find points that have not been used yet and chose cut 7 as the next starting cut point.

When the algorithm starts in cut 1, it will have to traverse the list 2 times, once to create the inside area and once for the outside. It will therefore have to traverse the list in 2 directions.

There is no way to determine where a polygon list starts, and what direction it has been drawn in, so multiple checks will be taken to determine what to do. The first cut might be cut 6, then a variable from the CutPoint called `insideNextCutIndex` will here point to cut 7, this would not be the right path since we have to leave the polygon to another polygon. However if the first cut is cut 1, the `insideNextCutIndex` will be cut 2, which is the correct path, even though we are leaving the polygon list. To indicate if the CutPoint is allowed to change the polygon list, a boolean called `isHole` will be checked whether it is true or false, since a hole should become a part of the new polygon.

When points from the drawn area are added, they should be added to both new polygons, but points that fall inside the drawn area should be added to only one list, and points that fall outside should be added to the other list. To determine what direction the traversal should take, a collision check can be used to see if the next point is inside or outside the drawn area. Everytime a point has been used, it will be marked, so that a point will not be used more than necessary.

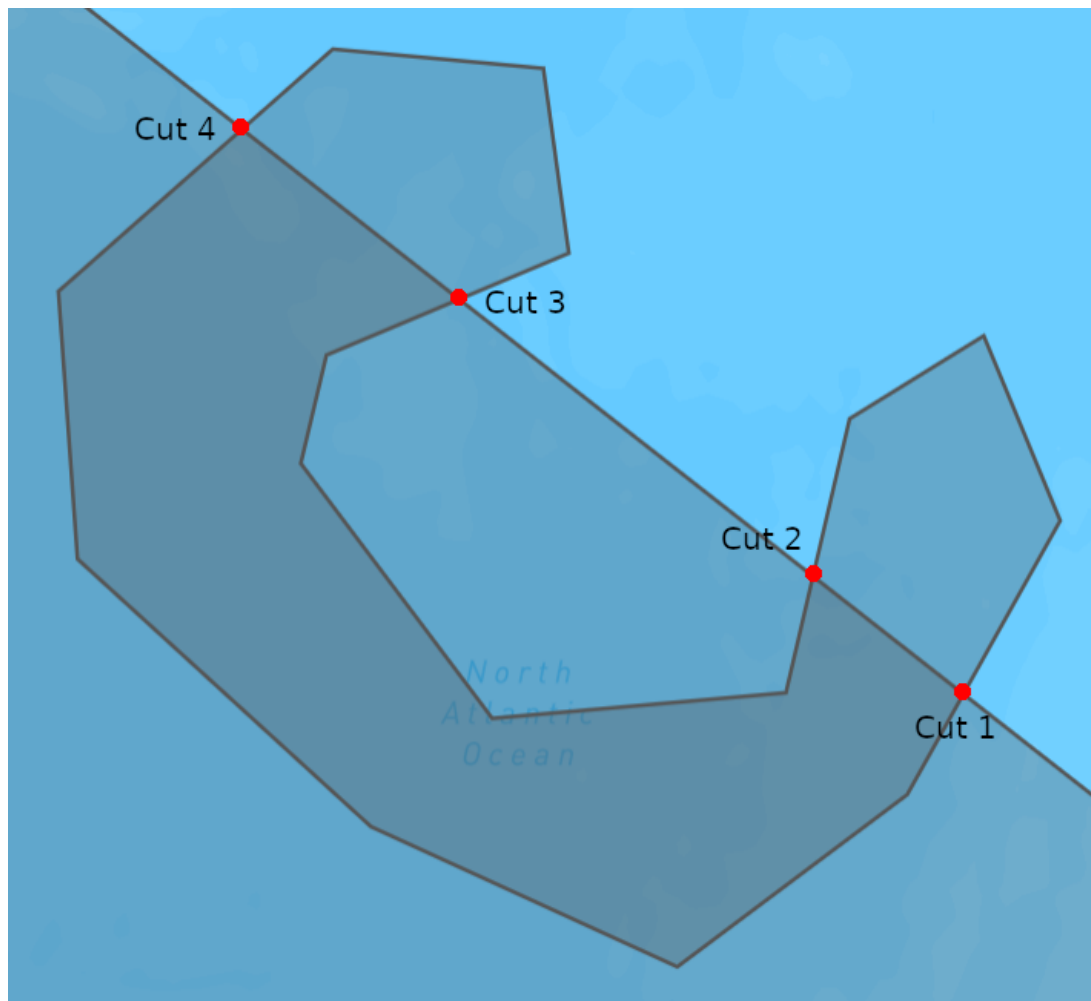


Figure 18: Cut point example 2.

As figure 16 shows, each cut point has an `insideUsed` and `outsideUsed` value. Figure 18 shows cases why this is necessary. If the first point is cut point 1, the outside will use cut 1 and cut 2 and the inside will use cut 1, 2, 3 and 4, this means that without the division of `outsideUsed` and `insideUsed` the outside area of cut 3 and 4 would be ignored. Another thing figure 18 shows cases, is that a Polygon might be turned into a MultiPolygon, similarly a MultiPolygon can be turned into a Polygon. A simple way to handle this would be to always return a MultiPolygon in all cases. Otherwise multiple checks and multiple return methods would be required.

## 4.5 UX and UI

The UX aspect of the project has not been prioritized. The project is a proof of concept to show the feature working with the rest of the system. UX has been pushed to the next iterations of the project. More about this in discussion section 7.2.

The UI for the subdivision site has been made to look like other popular interactive maps. Google Maps, Bing Maps, MapQuest et cetera, all have one thing in common. They put their

UI elements all the way to either the left or right side, and sometimes at the very top of the page. This is important since the user should be able to see as much of the map as possible.

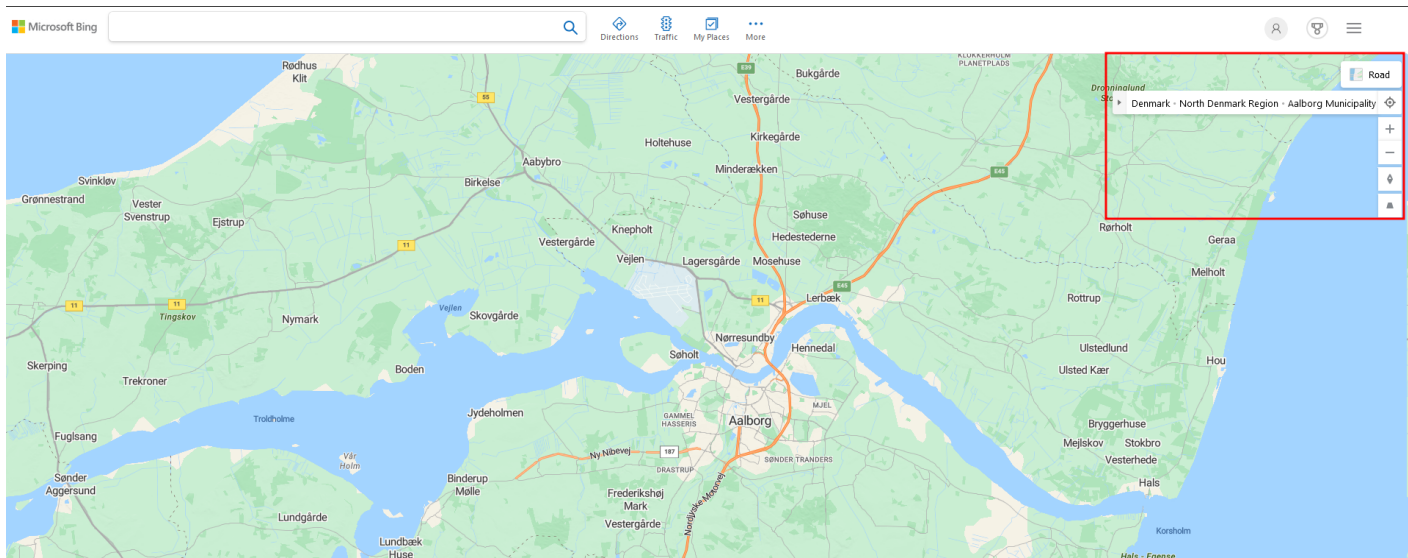


Figure 19: Bing Maps UI.

## 4.6 Summary

The design phase of this project focused on maintaining the current architecture. The analysis section gave an overview of the components that are gonna be affected by our feature. These components have been looked at in greater detail in the design section, to understand how our own component will interact with them. Sequence diagrams have been made to get a better understanding of these interactions and the flow of control in the system.

## 5 Implementation

The design class diagram in figure 8 has given an overview of how to implement our features within the existing system. The analysis section describing how our features might impact the affected components, has given the team a better idea on how to remain within the system constraints when implementing the feature.

This chapter presents the implementation of features that are based on the requirements in section 2.2, and provides an explanation for the reason behind the chosen solutions.

### 5.1 Display areas in a separate map with division functionality

#### 5.1.1 The system must have a new empty map that can load one or more areas

```
41 function initializeMap(_selectedFeatures) {  
42     map = L.map('map');  
43  
44     //the selected feataures are initiated as a leaflet layer  
45     selectedFeatures = _selectedFeatures;  
46     selectedFeaturesArray = JSON.parse(selectedFeatures);  
47     geojsonLayer = L.geoJSON(JSON.parse(selectedFeatures), {  
48         onEachFeature: onEachFeature  
49     });  
50  
51     //layer group to add to map. layers inside the layer group.  
52     geojsonLayerGroup = new L.LayerGroup();  
53     geojsonLayerGroup.addTo(map);  
54     geojsonLayerGroup.addLayer(geojsonLayer);  
55  
56     //editable layers are for the drawn polygons  
57     editableLayers = new L.FeatureGroup();  
58     editableLayers.addTo(map);  
59  
60     centerMap();  
61     setLabels(selectedFeaturesArray, geojsonLayer);  
62 }
```

Figure 20: MarketSimSubdivideController.js initialize map method.

Figure 20 shows how to initialize the map for the subdivision site. As seen in sequence diagram 9 the user has to select one or more areas from the PMS site, before redirecting to the subdivision site. These areas are then sent to the subdivision site, represented as GeoJSON data. The GeoJSON is used in the initializeMap method and is then converted into Leaflet layers called geojsonLayer in the snippet. These layers are then inserted into the map. A couple of utility methods are used for centering the map around the selected areas, and also labeling them with the correct country codes.

### 5.1.2 The user must have the ability to draw 1 polygon on the map

The method from figure 20 also adds another layer called `editableLayers` to the map. This layer is made for the drawing of polygons. The reason why a separate layer has to be made for the drawings, is because the area layer has to be reset every time a subdivision occurs, therefore it is easier to maintain the drawing functionality when it is encapsulated within its own layer. Leaflet has an inbuilt drawing functionality for their maps. First a configuration has to be made for the draw control.

```
306 //The drawing and editing options.
307 var options = {
308   position: 'topright',
309   draw: {
310     polyline: true,
311     polygon: {
312       allowIntersection: false, // Restricts shapes to simple polygons
313       drawError: {
314         color: '#e100', // Color the shape will turn when intersects
315         message: '<strong>Oh snap!<strong> you can\'t draw that!' // Message that will s
316       }
317     },
318     circle: true, // Turns off this drawing tool
319     rectangle: true,
320     marker: true
321   },
322   edit: {
323     featureGroup: editableLayers, //REQUIRED!!
324     remove: true
325   }
326 };
```

Figure 21: MarketSimSubdivideController.js drawing configuration.

The configuration is formatted as JSON data. The position variable puts the drawing to the top-right of the map. Which as mentioned in the UI section 4.5 is the ideal spot to place the UI. The draw variable sets the different shapes the user is able to draw, and the edit variable sets which layers are gonna have the draw UI, which is the `editableLayers`. The remove variable sets a delete button in the UI for deleting polygons which fulfills R1.3.2.1.

```
328     var drawControl = new L.Control.Draw(options);
329     map.addControl(drawControl);
330
331     map.on(L.Draw.Event.CREATED, function (e) {
332         var type = e.layerType,
333             layer = e.layer;
334
335         if (type === 'marker') {
336             layer.bindPopup('A popup!');
337         }
338         editableLayers.addLayer(layer);
339
340         //Log GIS coordinates from drawn objects
341         console.log("A drawing has been created:");
342         console.log(JSON.stringify(layer.toGeoJSON()));
343     });
```

Figure 22: MarketSimSubdivideController.js leaflet draw functionality.

The configurations are used in the Leaflet inbuilt draw functionality as seen in line 328. The following function from line 331 adds the layer with the polygon to the `editableLayers` whenever a drawing event is created.

### 5.1.3 The user must have the ability to select 1 area in the map

In code snippet figure 20 , when initializing the layer that contains the areas, an “oneachfeature” attribute was added. This function can define what happens when different types of events occur for each feature on that layer.

```
248
249     //Function for each feature on the layer
250     function onEachFeature(feature, layer) {
251         layer.on({
252             mouseover: highlightFeature,
253             mouseout: resetHighlight,
254             click: selectFeature
255         });
256     }
```

Figure 23: MarketSimSubdivideController.js function that runs on each feature.

In snippet 23 the `mouseover` and `mouseout` events are for when hovering over an area, then it should highlight another color. The `click` event is for selecting features to be subdivided.



```
278 function selectFeature(e) {  
279     //If the current feature has not already been selected. reset the style of the selected fea  
280     if (e.target !== _selected) {  
281         geojsonLayer.resetStyle(_selected)  
282         currentPageSelectedFeatures.length = 0;  
283         currentPageSelectedFeatures.push(e.target);  
284  
285         var layer = e.target;  
286         layer.setStyle({  
287             weight: 3,  
288             color: 'white',  
289             fillOpacity: 1,  
290             fillColor: '#fbec55'  
291         });  
292  
293         _selected = layer;  
294     } else {  
295         geojsonLayer.resetStyle(_selected)  
296         _selected = e.target;  
297     }  
298  
299  
300     for (let i = 0; i < currentPageSelectedFeatures.length; i++) {  
301         alert(currentPageSelectedFeatures[i].feature.properties.area);  
302     }  
303 }
```

Figure 24: MarketSimSubdivideController.js select function.

The select function is a simple if-else function. If the feature that is clicked on is already selected, then we deselect it, else in every other scenario, we select that feature and highlight it. If a new feature is selected while there is already a feature selected. The already selected feature gets deselected and the new feature becomes selected.

## 5.2 Divide an existing area into two or more subareas.

### 5.2.1 The user must be able to divide an existing area into two or more subareas.

#### 5.2.1.1 SubdivideAreas

The division starts in the SubdivideAreas() method, which receives the two polygons as strings. The strings are turned into FeatureCollections.

```

1 public String SubdivideAreas(string addedPolygon, string existingFeatures, List<string> areaNames)
2 {
3     NameSubdividedPriceAreas(areaNames);
4
5     var mainAreaOpen = JsonConvert.DeserializeObject<FeatureCollection>(existingFeatures);
6     var newAreaOpen = JsonConvert.DeserializeObject<FeatureCollection>(addedPolygon);
7
8     Feature mainArea = mainAreaOpen.Features[0];
9     Feature newArea = newAreaOpen.Features[0];
10
11     List<List<List<List<float>>>>> mainAreaList = new List<List<List<List<float>>>>>();
12     List<List<float>> newAreaList = new List<List<float>>>();
13     if (mainArea.Geometry.GetType().ToString().Equals("GeoJSON.Net.Geometry.Polygon"))
14     {
15         List<List<List<float>>>> temp = new List<List<List<float>>>>();
16         temp.Add(unpackPolygon(mainArea));
17         mainAreaList.Add(temp);
18     }
19     else if (mainArea.Geometry.GetType().ToString().Equals("GeoJSON.Net.Geometry.MultiPolygon"))
20     {
21         mainAreaList = unpackMultiPolygon(mainArea);
22     }
23
24     newAreaList = unpackPolygon(newArea);
25
26     List<List<List<List<List<float>>>>> div = multiSolver(mainAreaList, newAreaList);
27
28     string s1 = stringify(div[0]);
29     string s2 = stringify(div[1]);
30
31     string result = quickGEOJSON(s1, s2, areaNames);
32
33     return result;
34 }

```

Figure 25: SubdivideAreas code.

The FeatureCollections are used to find out if the existingFeatures is a Polygon or a MultiPolygon. The Polygon and MultiPolygon have different list structures, so they need different methods to be turned into lists. If the existingFeatures is a Polygon, it is added to an extra list so that it has the same format as a MultiPolygon list. The addedPolygon, which is the polygon the user has drawn, will always be a Polygon. It is thereby possible to use the same solver for both cases. Most of the division functionality is within the multiSolver method. The resulting division is turned from lists into strings and the strings are then formatet to match the syntax of GeoJson.

### 5.2.1.2 multiSolver

The multiSolver() method is described in the sequence diagram in figure 13 in analysis 4.4.

```

1 public List<List<List<List<List<float>>>>> multiSolver(List<List<List<List<float>>>> mainArea, List<List<float>> newArea)
2 {
3     List<CutDataClass> cutData = new List<CutDataClass>();
4     for (int i = 0; i < mainArea.Count; i++)
5     {
6         for (int j = 0; j < mainArea[i].Count; j++)
7         {
8             for (int k = 0; k < mainArea[i][j].Count-1; k++)
9             {
10                 for (int l = 0; l < newArea.Count-1; l++)
11                 {
12                     var result = allCutDataFinder(new List<List<float>> { mainArea[i][j][k], mainArea[i][j][k + 1] }, new List<List<float>> { newArea[l], newArea[l + 1] });
13                     if(result != null)
14                     {
15                         cutData.Add(result);
16                     }
17                 }
18             }
19         }
20     }
21
22     if(cutData.Count == 0)
23     {
24         return noIntersectionSolver(mainArea, newArea);
25     }
26
27     List<List<List<List<List<float>>>>> mainWithCuts = allIntersectionAdderMain(mainArea, cutData);
28     List<List<float>> newWithCuts = allIntersectionAdderNew(newArea, cutData);
29     List<CutDataClass> newCutData = fixCutData(cutData, newWithCuts);
30
31     List<List<List<List<List<float>>>>> dividedArea = cutArea(mainWithCuts, newWithCuts, newCutData);
32
33     List<List<List<List<List<List<float>>>>> finalResult = unchangedAdder(mainArea, mainWithCuts, newArea, dividedArea);
34
35     return finalResult;
36 }

```

Figure 26: multiSolver code.

It is necessary to loop through every point in every list of the mainArea and compare them with every point in the newArea to find the cut points.

If no cut points are found, it passes the work to the noIntersectionSolver(). If cuts were found it first adds the points to the lists. It is necessary to use two separate functions for this since the mainArea and the newArea are structured differently, one as a MultiPolygon and one as a Polygon, also the Polygon might have to sort some of the found cut points. Once the points have been added the fixPointData() method is called to add additional data about the cut points. With this additional data it is possible to use the points to cut the area using the cutArea() method. The method only uses the cut points for cutting, meaning it ignores any subarea that does not have a cut point, therefore the unchangedAdder() is used to find these areas and add them to the correct division.

### 5.2.1.3 allCutDataFinder

This method is described in the sequence diagram in figure 15 in analysis 4.4. It is responsible for finding, validating and initiating the cut points. If the point is not valid, it returns null.

```
1 public CutDataClass allCutDataFinder(List<List<float>> firstPoints, List<List<float>> secondPoints, List<List<float>> polygonList, List<List<float>>
2 {
3     CutDataClass cut = null;
4     List<float> point = lineIntersection(secondPoints, firstPoints);
5     float m = 0.0001f;
6     if ((point[0] < firstPoints[0][0]+m && point[0]+m > firstPoints[1][0]) || (point[0]+m > firstPoints[0][0] && point[0] < firstPoints[1][0]+m))
7     {
8         if ((point[0] < secondPoints[0][0] + m && point[0] + m > secondPoints[1][0]) || (point[0] + m > secondPoints[0][0] && point[0] < secondPoints[1][0]+m))
9         {
10             cut = new CutDataClass(point, isHole, polygonList, firstPoints[0], newArea, secondPoints[0]);
11         }
12     }
13
14     return cut;
15 }
```

Figure 27: allCutDataFinder code.

This method proved to be a programmatic issue. The given GIS coordinates consist of decimal numbers, some with more than 15 digits after the comma. These numbers are then performed math on. Most programming languages have issues with precision when it comes to large decimal numbers, particularly when division is performed on them. This means that the found point, that the `lineIntersection()` returns, will not be 100% accurate. In order to check if the point is valid or not, it is checked if the point is on the lines, but because of the margin of error, it likely will not be on the line. The variable `m` is used as a multiplier to create a field around the line, where the point is allowed to be placed and still considered valid. This does raise some concerns about whether or not a cut point still might be able to fall outside of the field, and if two lines that almost intersect, but do not, might create a cut point that should not be there.

#### 5.2.1.4 lineIntersection

This method follows standard math to calculate the line-line intersection given two pairs of points that each create a line.

```
1 public List<float> lineIntersection(List<List<float>> firstPoint, List<List<float>> secondPoint)
2 {
3     float x11 = firstPoint[0][0], y11 = firstPoint[0][1], x12 = firstPoint[1][0], y12 = firstPoint[1][1];
4     float x21 = secondPoint[0][0], y21 = secondPoint[0][1], x22 = secondPoint[1][0], y22 = secondPoint[1][1];
5
6     float a1;
7     if((x12 - x11) == 0)
8     {
9         a1 = (y12 - y11);
10    }
11    else
12    {
13        a1 = (y12 - y11) / (x12 - x11);
14    }
15    float b1 = y12 - a1 * x12;
16
17    float a2;
18    if ((x22 - x21) == 0)
19    {
20        a2 = (y22 - y21);
21    }
22    else
23    {
24        a2 = (y22 - y21) / (x22 - x21);
25    }
26    float b2 = y22 - a2 * x22;
27
28    float x;
29    if ((a1 - a2) == 0)
30    {
31        x = (b2 - b1);
32    }
33    else
34    {
35        x = (b2 - b1) / (a1 - a2);
36    }
37    float y = a2 * x + b2;
38
39    return new List<float> { x, y };
40 }
```

Figure 28: lineIntersection code.

The most notable thing about the method is the constant check for 0 division, as this happened frequently in testing.

#### 5.2.1.5 intersectionAdders

There are two separate functions to add the intersections to the given lists. For one, this is because the lists are structured differently, but also because it might be necessary to go through a sorting phase for the drawn polygon.

```

1 public List<List<List<List<float>>>>> allIntersectionAdderMain(List<List<List<List<float>>>>> area, List<CutDataClass> cutData)
2 {
3     List<List<List<List<float>>>>> result = new List<List<List<List<float>>>>>();
4     for (int i = 0; i < area.Count; i++)
5     {
6         List<List<List<float>>>> tempIsland = new List<List<List<float>>>>();
7         for (int j = 0; j < area[i].Count; j++)
8         {
9             List<List<float>>> tempAreas = new List<List<float>>>();
10            int lastCut = -1;
11            int firstCut = -1;
12            for (int k = 0; k < area[i][j].Count; k++)
13            {
14                tempAreas.Add(area[i][j][k]);
15                for (int l = 0; l < cutData.Count; l++)
16                {
17                    if (cutData[l].insideList == area[i][j] && cutData[l].insidePreCoordinate == area[i][j][k])
18                    {
19                        if (!tempAreas.Contains(cutData[l].cutPoint))
20                        {
21                            tempAreas.Add(cutData[l].cutPoint);
22                            cutData[l].setInsideIndex(tempAreas.Count-1);
23                            if (lastCut != -1)
24                            {
25                                cutData[lastCut].setInsideNextCutIndex(tempAreas.Count - 1);
26                            }
27                            else
28                            {
29                                firstCut = l;
30                            }
31                            lastCut = l;
32                        }
33                    }
34                }
35            }
36            if (firstCut != -1)
37            {
38                cutData[lastCut].setInsideNextCutIndex(tempAreas.IndexOf(cutData[firstCut].cutPoint));
39            }
40            tempIsland.Add(tempAreas);
41            for (int k = 0; k < cutData.Count; k++)
42            {
43                if (area[i][j] == cutData[k].insideList)
44                {
45                    cutData[k].insideList = tempAreas;
46                }
47            }
48        }
49        result.Add(tempIsland);
50    }
51    return result;
52 }
53

```

Figure 29: allIntersectionAdderMain code.

Both methods work by utilizing the pre-coordinated data of the cut point that was found in the allCutDataFinder() method. The snippet in figure x shows how the method collects more data about the cut points while adding them to the new list, this is different from the allIntersectionAdderNew() method, which only adds the points to the list.

As the example in design 4.4 figure 17 showcases, it is necessary to sort the cut points that have the same pre-coordinate for the drawn polygon.

```

1  List<List<float>> cutPointList = new List<List<float>>();
2  for (int i = 0; i < cutData.Count; i++)
3  {
4      cutPointList.Add(cutData[i].cutPoint);
5  }
6  for (int i = 0; i < result.Count-1; i++)
7  {
8      for (int j = 0; j < result.Count - i - 1; j++)
9      {
10         if (cutPointList.Contains(result[j]) && cutPointList.Contains(result[j + 1]))
11         {
12             int p1 = -1;
13             int p2 = -1;
14             for (int k = 0; k < cutData.Count; k++)
15             {
16                 if (result[j] == cutData[k].cutPoint)
17                 {
18                     p1 = k;
19                 }
20                 if (result[j+1] == cutData[k].cutPoint)
21                 {
22                     p2 = k;
23                 }
24             }
25             if (cutData[p1].outsidePreCoordinate == cutData[p2].outsidePreCoordinate && cutData[p1].outsidePreCoordinate == cutData[p2].outsidePreCoordinate)
26             {
27                 if (distOf2Points(cutData[p1].cutPoint, cutData[p1].outsidePreCoordinate) > distOf2Points(cutData[p2].cutPoint, cutData[p2].outsidePreCoordinate))
28                 {
29                     var tempVar = result[j];
30                     result[j] = result[j + 1];
31                     result[j + 1] = tempVar;
32                 }
33             }
34         }
35     }
36 }
37 int lastCut = -1;
38 int firstCut = -1;
39 for (int i = 0; i < cutData.Count; i++)
40 {
41     cutData[i].setOutsideIndex(result.IndexOf(cutData[i].cutPoint));
42     if (lastCut != -1)
43     {
44         cutData[lastCut].setOutsideNextCutIndex(result.IndexOf(cutData[i].cutPoint));
45     }
46     else
47     {
48         firstCut = i;
49     }
50     lastCut = i;
51 }
52 if (firstCut != -1)
53 {
54     cutData[lastCut].setOutsideNextCutIndex(result.IndexOf(cutData[firstCut].cutPoint));
55 }
56 return result;

```

Figure 30: allIntersectionAdderNew code snippet 1.

The sorting algorithm used is a simple bubble sort algorithm. While this is a rather slow algorithm, it should not matter given that it should be rare that it is used in the first place, and likely there will only be a few points to sort. The sorting is done by calculating the distance from the cut point to the pre-coordinate.

```

1 public float distOf2Points(List<float> p1, List<float> p2)
2 {
3     double q1 = Math.Pow((p2[1] - p1[1]), 2);
4     double q2 = Math.Pow((p2[0] - p1[0]), 2);
5     double d = Math.Sqrt(q1 + q2);
6     return (float)d;
7 }

```

Figure 31: distOf2Points code.

A simple helper function is used to calculate the distance. The method also adds data to the cut points, however it only does so after all points have been added and sorted.

### 5.2.1.6 fixCutData

This method finds the last data about the cutpoints. It looks at how the points are placed in relation to each other.

```

1 public List<CutDataClass> fixCutData(List<CutDataClass> cutData, List<List<float>> area)
2 {
3     List<List<float>> cutPointList = new List<List<float>>();
4     for (int i = 0; i < cutData.Count; i++)
5     {
6         cutPointList.Add(cutData[i].cutPoint);
7     }
8     for (int i = 0; i < cutData.Count; i++)
9     {
10        int j = area.IndexOf(cutData[i].cutPoint);
11        cutData[i].setOutsideIndex(j);
12        j = j + 1;
13        while ( !cutPointList.Contains(area[j]))
14        {
15            j++;
16            if (j >= area.Count)
17            {
18                j = 0;
19            }
20        }
21        cutData[i].setOutsideNextCutIndex(j);
22
23        cutData[i].setCutListInsideNextCutIndex(cutPointList.IndexOf(cutData[i].insideList[cutData[i].insideNextCutIndex]));
24        cutData[i].setCutListOutsideNextCutIndex(cutPointList.IndexOf(area[cutData[i].outsideNextCutIndex]));
25        cutData[i].setCutListIndex(i);
26    }
27    return cutData;
28 }

```

Figure 32: fixCutData code.

The data for the lists of cut points makes it possible to jump from one list to another without searching or checking indexes.

### 5.2.1.7 pointInShape

This method is the collision checking method and one of the most important methods, as it



can determine if a point is inside or outside the drawn polygon.

```

1  public bool pointInShape(List<float> p, List<List<float>> s)
2  {
3      bool isInside = false;
4      int j = s.Count-1;
5      for (int i = 0; i < s.Count; i++)
6      {
7          if ((s[i][1] > p[1]) != (s[j][1] > p[1]) && (p[0] < (s[j][0] - s[i][0]) * (p[1] - s[i][1]) / (s[j][1] - s[i][1]))
8          {
9              isInside = !isInside;
10         }
11         j = i;
12     }
13     return isInside;
14 }

```

Figure 33: pointInShape code.

The method takes a point and a polygon and checks if the point exists within the polygons boundaries. If it does, the method returns true, if not the method returns false.

#### 5.2.1.8 noIntersectionSolver

This method is described in the sequence diagram in figure 14 in analysis 4.4. As described, it starts by looking at what case it is dealing with.

```

1  for (int i = 0; i < mainArea.Count; i++)
2  {
3      if (pointInShape(mainArea[i][0][0], newArea))
4      {
5          split = true;
6          firstList.Add(mainArea[i]);
7      }
8      else
9      {
10         secondList.Add(mainArea[i]);
11     }
12
13     if (pointInShape(newArea[0], mainArea[i][0]))
14     {
15         drill = true;
16         drillIndex = i;
17     }
18
19     if (drill)
20     {
21         for (int j = 1; j < mainArea[i].Count; j++)
22         {
23             if (pointInShape(newArea[0], mainArea[i][j])){
24                 drill = false;
25             }
26         }
27     }
28 }

```

Figure 34: noIntersectionSolver code snippet 1.

It heavily relies on the `pointInShape` method to find out what is inside and what is outside of the drawn area, and collects the results in two lists. It simultaneously checks if the drawn polygon appears inside a polygon of the given `MultiPolygon`, if it does it might be a drill case, however it could still be possible that the drawn polygon has been drawn inside a hole within the given polygon, if that is the case, the drill is set back to false.

It is possible for both drill and split to be true at the same time, in that case it should be a split case.

```
1  if (split)
2  {
3      if (firstList.Count == 0 || secondList.Count == 0)
4      {
5          split = false;
6      }
7      else
8      {
9          result.Add(firstList);
10         result.Add(secondList);
11         return result;
12     }
13 }
14 else if (drill)
15 {
16     secondList = new List<List<List<List<float>>>>>();
17     mainArea[drillIndex].Add(newArea);
18
19     secondWithHoles.Add(newArea);
20     secondList.Add(secondWithHoles);
21
22     result.Add(mainArea);
23     result.Add(secondList);
24     return result;
25 }
26
27 // do nothing, the user made an error while drawing
28 result.Add(mainArea);
29 result.Add(new List<List<List<List<float>>>>>());
30 return result;
```

Figure 35: `noIntersectionSolver` code snippet 2.

In the split case, the return lists have already been created, so as long as they both contain something, they get returned. For the drill case, the drawn area has to be added as a hole for the correct polygon. The polygons index has already been saved, so it is simple to add it. The second list only has to consist of the drawn polygon, so it only needs to be formatted into the correct list type before it can be returned.

If it is not a split or a drill case, it must be an error case, meaning that there is nothing to divide. In that case, the `mainArea` returned alongside an empty list.

### 5.2.1.9 cutArea

This is the biggest method and the most complicated. It starts by looking at the first cut point in a list of cut points and saves some data about the point, for example the point itself and the polygon list it appears in.

```

1 List<List<List<List<float>>>> firstList = new List<List<List<List<float>>>>();
2 List<List<List<List<float>>>> secondList = new List<List<List<List<float>>>>();
3 for (int i = 0; i < cutData.Count; i++)
4 {
5     if ((!cutData[i].insideUsed || !cutData[i].outsideUsed) && !cutData[i].isHole )
6     {
7         List<List<List<float>>>> firstWithHoles = new List<List<List<List<float>>>>();
8         List<List<List<List<float>>>> secondWithHoles = new List<List<List<List<float>>>>();
9
10        List<float> firstCut = cutData[i].cutPoint;
11        List<List<float>>> outlineList = cutData[i].insidelist;
12
13
14        CutDataClass current = cutData[i];
15        List<List<float>>> currentList = current.insidelist;
16
17        List<List<float>>> innerList = new List<List<float>>>();
18        List<List<float>>> outerList = new List<List<float>>>();
19
20        List<List<float>>> checkList;
21        bool newBorder = true;
22        //////////////// add inner ////////////////
23        if (!cutData[i].insideUsed)
24        {
25            // first run 1 time so current isn't first cut anymore
26            cutData[i].useInside();
27            checkList = outlineList;
28
29            if (cutData[current.cutListOutsideNextCutIndex].isHole || currentList.Contains(cutArea[current.outsideNextCutIndex]))
30            {
31                currentList = cutArea;
32                innerList = newBorderAdder(innerList, checkList, current.outsideIndex, current.outsideNextCutIndex, currentList);
33                newBorder = false;
34                current = cutData[current.cutListOutsideNextCutIndex];
35            }
36            else
37            {
38                checkList = cutArea;
39                innerList = newBorderAdder(innerList, checkList, current.insideIndex, current.insideNextCutIndex, currentList);
40                newBorder = true;
41                current = cutData[current.cutListInsideNextCutIndex];
42            }

```

Figure 36: cutArea code snippet 1.

The method goes from cut point to cut point until it finds the starting cut point again. It is therefore necessary to do one run before entering the loop, so that the current cut point in use is not the starting cut point. Since it is the first cut point, it is a given that it does not appear in a hole, it can thereby be assumed, that by shifting to the drawn polygon list and moving to the next cut point, it will either be a hole, inside the current polygon or a cut on the same polygon as the current cut, or a cut on a different polygon. If it is a different polygon, the algorithm should not move towards it, but instead not shift polygon lists and move towards the next cut point that exists within it. If the next cut is not on another polygon, it is safe to

switch lists and move towards the next cut.

After the first run this process is repeated, each time shifting between the original polygon list, and the drawn polygon list.

```
1  while (current.cutPoint != firstCut)
2  {
3
4      if (newBorder)
5      {
6          currentList = cutArea;
7          cutData[current.cutListIndex].useInside();
8          checkList = outlineList;
9          if (current.isHole)
10         {
11             innerList = holeBorderAdder(innerList, current.insideList, current.outsideIndex, current.outsideNextCutIndex, currentList);
12         }
13         else
14         {
15             innerList = newBorderAdder(innerList, checkList, current.outsideIndex, current.outsideNextCutIndex, currentList);
16         }
17
18         current = cutData[current.cutListOutsideNextCutIndex];
19         newBorder = false;
20     }
21     else
22     {
23         currentList = current.insideList;
24         cutData[current.cutListIndex].useInside();
25         checkList = cutArea;
26
27         innerList = newBorderAdder(innerList, checkList, current.insideIndex, current.insideNextCutIndex, currentList);
28
29         current = cutData[current.cutListInsideNextCutIndex];
30         newBorder = true;
31     }
32 }
```

Figure 37: cutArea code snippet 2.

It is no longer a given that the current cut is a hole or not, but the newBoarder variable has been set which shows if the list should be the original or the drawn polygon. When the newBoarder is true, points from the drawn polygon have to be added, and when it is false, points from the original polygon list are being added.

After the process finds the first cut point, the process repeats one more time. The first run finds all the points that appear inside the drawn polygon, and the second run finds all the points that appear outside the drawn polygon.

```

1  ////////////////////////////////////////////////// add outer ///////////////////////////////////
2  if (!cutData[i].outsideUsed)
3  {
4      // first run 1 time so current isn't first cut anymore
5      current = cutData[i];
6      currentList = current.insideList;
7      cutData[i].useOutside();
8      checkList = outlineList;
9
10
11
12     if (cutData[current.cutListOutsideNextCutIndex].isHole || currentList.Contains(cutArea[current.outsideNextCutIndex]))
13     {
14         currentList = cutArea;
15         outerList = newBorderAdder(outerList, checkList, current.outsideIndex, current.outsideNextCutIndex, currentList);
16         newBorder = false;
17         current = cutData[current.cutListOutsideNextCutIndex];
18     }
19     else
20     {
21         currentList = current.insideList;
22         checkList = cutArea;
23         outerList = oldBorderAdder(outerList, checkList, current.insideIndex, current.insideNextCutIndex, currentList);
24         newBorder = true;
25         current = cutData[current.cutListInsideNextCutIndex];
26     }
27
28     while (current.cutPoint != firstCut)

```

Figure 38: cutArea code snippet 3.

The run again starts one run outside the loop before entering the while loop. The main difference between the two runs is the use of collision check with the drawn area, and that the found points are added to different lists.

Once all cut points are used, the method returns the two new lists.

#### 5.2.1.10 borderAdders

There are two different border adder methods. New border and old border. The two methods only have one difference, one checks if the point in shape returns true, and one checks if it returns false.

```

1  private List<List<float>> newBorderAdder(List<List<float>> workList, List<List
2  {
3      if (pointInShape(currentList[index1 + 1], checkList))
4      {
5          workList = loopForwardAdder(workList, currentList, index1, index2);
6      }
7      else if (pointInShape(currentList[index1 - 1], checkList))
8      {
9          workList = loopBackwardAdder(workList, currentList, index1, index2);
10     }
11     return workList;
12 }

```

Figure 39: newBorderAdder.

The cutArea method knows what list it should add from, but it does not know what direction

it should move in to add the correct points. The method works by checking if the next point in the lists or if the previous point in the list collides with the checklist. It will then move in that direction, until it reaches the given endpoint called index2. The old border adder works exactly the same, but it moves in the direction that does not collide with the given checklist.

#### 5.2.1.11 loopAdders

There are two different loop adders, forward and backwards. The two methods are very similar, but one loops by adding to variable j and the other subtracts.

```
1 private List<List<float>> loopBackwardAdder(List<List<float>> t
2 {
3     if (index1 > index2)
4     {
5         for (int j = index1; j >= index2; j--)
6         {
7             workList.Add(currentList[j]);
8         }
9     }
10    else
11    {
12        for (int j = index1; j >= 0; j--)
13        {
14            workList.Add(currentList[j]);
15        }
16        for (int j = currentList.Count - 1; j >= index2; j--)
17        {
18            workList.Add(currentList[j]);
19        }
20    }
21    return workList;
22 }
```

Figure 40: loopBackwardAdder.

The border adder methods know what direction to move, but they do not know where the list starts and ends. It might be that the end index comes before the starting index. That means that the loop has to reach the end of the list and then move to the beginning of the list, in order to find all the points between the starting and end index.

#### 5.2.1.12 unchangedAdder

The cut area method only looks at cut points, this means that any area that does not have a cut point will be ignored.

```
1 public List<List<List<List<List<float>>>>>> unchangedAdder(List<List<List<
2 {
3     for (int i = 0; i < mainArea.Count; i++)
4     {
5         bool fullList = true;
6         for (int j = 0; j < mainArea[i].Count; j++)
7         {
8             if(!mainArea[i][j].SequenceEqual(mainWithCuts[i][j]))
9             {
10                 fullList = false;
11             }
12         }
13         if (fullList)
14         {
15             if (pointInShape(mainArea[i][0][0], newArea))
16             {
17                 dividedArea[0].Add(mainArea[i]);
18             }
19             else
20             {
21                 dividedArea[1].Add(mainArea[i]);
22             }
23         }
24     }
25 }
```

Figure 41: unchangedAdder code.

The unchanged adder functions by comparing the original MultiPolygon with the MultiPolygon that has been modified to include the found cut points. If the two lists are the same that means that the cut area method must have ignored the list. It is then again checked if the found list collides with the drawn polygon or not to determine which division list it should be added to.

### 5.2.2 Subdivided areas should have new labels

The analysis and design reflected that the naming would happen in the backend. However, it was easier to handle JSON data using JavaScript in the frontend and use the in-built functions of leaflet.

```
function handleAreaNamingAndLabels() {
    for (let i = 0; i < subdividedAreasObj.features.length; i++) {
        var geojsonLayer = L.geoJSON(subdividedAreasObj.features[i]);
        var bounds = geojsonLayer.getBounds().getCenter();
        var i2 = i + 1;
        subdividedAreasObj.features[i].properties = {
            "area": `${selectedPriceAreas[0]}.${i2}`,
            "priceAreaLabelCoordinates": [bounds.lng, bounds.lat]
        }
    }
}
```

Figure 42: handleAreaNamingAndLabels code.

The Figure 42 shows the code snippet for setting the labels. The method is called after sub-

dividing the areas. The method loops through each of the subdivided areas and parses the subdivided area to GeoJSON format using leaflets in-built functions. This is done because it enables the use of other leaflet methods such as `getBounds().getCenter()` which basically gets the center of the subdivided area. This is necessary because referring to Figure 56 the `refreshGenericLabels()` function sets the labels automatically based on the properties of the areas where the name of the area is needed and the `priceAreaLabelCoordinates` and the label of each area should be centered.

### 5.2.3 Power plants must be distributed to subdivided areas

When the subdivide button is pressed, the areas are subdivided and the power plants for the subdivided areas are mapped to the correct area.

```
var features = JSON.stringify(subdividedFeaturesArray);  
var featureCollection = `{"type": "FeatureCollection", "features": ${features}}`;   
distributePowerPlants(featureCollection);
```

Figure 43: Subdivide button onclick event.

Figure 43 is a code snippet of what happens at the end of the function with the subdivide button onclick event. The subdivided areas are parsed as a string which is passed into a specific `featureCollection` format. The method `distributePowerPlants()` is called. `distributePowerPlants()` takes a parameter which is the subdivided areas where the power plants are needed to be found.



```
function distributePowerPlants(subdividedPriceAreas) {  
  
    var operatingYear = JSON.parse(sessionStorage.getItem("operatingYear"));  
    var selectedFeaturesObjects = JSON.parse(selectedFeatures);  
  
    //Clear selectedPriceAreas  
    selectedPriceAreas.length = 0;  
  
    for (let i = 0; i < selectedFeaturesObjects.length; i++) {  
        selectedPriceAreas.push(selectedFeaturesObjects[i].properties.area);  
    }  
  
    var dividedPriceAreas = parseGeoJSONtoDict(subdividedPriceAreas);  
  
    request = {  
        operatingYear: operatingYear,  
        areaNames: selectedPriceAreas,  
        dividedPriceAreas: dividedPriceAreas,  
    }  
  
    GetDataAsync('/api/subdivision/DistributePowerPlants', 'POST', request, function (data) {  
        exportPowerPlantString = JSON.stringify(data);  
    }, null);  
}
```

Figure 44: distributePowerPlants code.

Figure 44 shows a code snippet of the `distributePowerPlants()` method. The operating year and the selected area are needed to get the plants from the database which we will come to. First, we clear the `selectedPriceAreas` list which is preserved from previous iterations. From the `selectedFeaturesObjects`, we get the area name from the properties of the object which is added to the `selectedPriceAreas` list. The area name is not the name of the subdivided areas, but is the name of the selected countries before subdividing.

A list of areas is made to make it possible to get power plants for multiple selected countries. The `subdividedPriceAreas` string is passed to the `parseGeoJSONtoDict()` function.

```
function parseGeoJSONtoDict(subdividedPriceAreas) {  
  
    var dict = [];  
    var geoJSON = JSON.parse(subdividedPriceAreas);  
  
    for (let i = 0; i < geoJSON.features.length; i++) {  
        var feature = geoJSON.features[i];  
  
        var type = feature.geometry.type;  
  
        var coordinates = feature.geometry.coordinates;  
        var coordinatesToString = JSON.stringify(coordinates);  
  
        var area = feature.properties.area;  
  
        //This is the format GeoLibrary is able to parse  
        var geoJSONString = `{"type":"${type}","coordinates":${coordinatesToString}}`;  
  
        dict.push({  
            key: area,  
            value: geoJSONString  
        })  
    }  
  
    return dict;  
}
```

Figure 45: parseGeoJSONtoDict code.

The method loops through the subdivided areas. The type of each feature and the coordinates and the area name are added to a dictionary with the area name as key. The dictionary containing the subdivided coordinates in the correct string is returned. Referring to Figure 45 the function `GetAreaAsync()` is called which makes a HTTP request using AJAX. The parameters passed are the API route prefix, the HTTP method, the message body data and a callback and error callback.

The data passed to the web API controller is defined in the request object which consists of the operating year, the area name and the subdivided areas.

The endpoint call is “api/subdivision/DistributePowerPlants” which is defined in the `SubdivisionController`.

```
[EnableCors("*", "*", "*")]
[Route("DistributePowerPlants")]
[HttpPost]
0 references
public Dictionary<string, List<SystemPowerPlant>> DistributePowerPlants(DistributePowerPlantsToAreasRequest request)
{
    Dictionary<string, int> priceAreas = new Dictionary<string, int>();

    foreach (var priceArea in request.areaNames)
    {
        priceAreas.Add(priceArea, priceAreasDict[priceArea]);
    }

    var allPowerPlants = _repositoryContext.MarketSimRepository.ListOperatingPowerPlantsByPriceArea(priceAreas, request.operatingYear);
    var subdividedAreasWithPowerPlants = _subdivide.MapPowerPlantToCorrectCountry(request.dividedPriceAreas, allPowerPlants);
    return subdividedAreasWithPowerPlants;
}
```

Figure 46: DistributePowerPlants code 2.

The figure 46 shows the code snippet for controller method DistributePowerPlants() in the SubdivisionController.

The DTO used is DistributePowerPlantsToAreasRequest. It can be seen on figure 47

```
1 reference
public class DistributePowerPlantsToAreasRequest
{
    1 reference
    public int operatingYear { get; set; }
    1 reference
    public List<string> areaNames { get; set; }
    1 reference
    public Dictionary<string, string> dividedPriceAreas { get; set; }
}
```

Figure 47: DistributePowerPlantsToAreasRequest code.

It contains the operating year, the area names and the subdivided areas. The format of the DTO matches the format of the request data sent from the frontend. This allows ASP.NET to deserialize the JSON data and map it to the corresponding DTO.

The DistributePowerPlants() method instantiates a dictionary of area names where the price areas are added. Then all the power plants are fetched from the database using the MarketSimRepository class where existing methods for fetching data from the database are. The power plants data and the subdivided areas are passed with a call to the MapPowerPlantToCorrectCountry() method.

```

1 reference
public Dictionary<string, List<SystemPowerPlant>> MapPowerPlantToCorrectCountry(Dictionary<string, string> dividedAreas, List<SystemPowerPlant>
{
    var subdividedAreaWithPowerPlants = new Dictionary<string, List<SystemPowerPlant>>();
    var currentPowerPlants = new List<SystemPowerPlant>();

    foreach(var dividedFeature in dividedAreas)
    {
        //Parse to geojson
        var polygon = Geometry.FromGeoJson(dividedFeature.Value);
        //Convert WKT to WKT geometry
        var polygonWKT = Geometry.FromWkt(polygon.ToWkt());

        foreach (var powerPlant in systemPowerPlants.ToList())
        {
            string cityLat = powerPlant.CityLatitude.ToString().Replace(",", ".");
            string cityLong = powerPlant.CityLongitude.ToString().Replace(",", ".");

            string checkedPoint = $"POINT ({cityLong} {cityLat})";

            if (polygonWKT.IsIntersects(Geometry.FromWkt(checkedPoint)))
            {
                currentPowerPlants.Add(powerPlant);
                systemPowerPlants.Remove(powerPlant);
            }
        }

        subdividedAreaWithPowerPlants.Add(dividedFeature.Key, new List<SystemPowerPlant>(currentPowerPlants));
        currentPowerPlants.Clear();
    }

    return subdividedAreaWithPowerPlants;
}

```

Figure 48: MapPowerPlantToCorrectCountry code.

Figure 48 shows the method for distributing the power plants across the subdivided countries. In the start of the method, there is instantiated a dictionary and a list. The currentPowerPlants list holds the current power plants that are found to intersect with the current subdivided area in consideration. The subdividedAreaWithPowerPlants is a dictionary containing all the subdivided areas as key with the list of power plants as value.

The library GeoLibrary version 1.2.1 was used to parse the subdivided areas to the correct format and contained useful methods such as checking for intersection between two geometry figures.

The method loops through each divided area and deserializes the string as a GeoJSON object. Then the GeoJSON object is parsed to WKT format which is easier to work with when doing interpolation of values in a string. All the power plants are looped through and checked for whether they intersect with the subdivided geometry figure and are added to the currentPowerPlants list if they do.

At the end the list and the subdivided area name is added to the “subdividedAreaWithPowerPlants” dictionary and the list of “currentPowerPlants” is cleared to prepare for the next distribution of power plants for the next subdivided area.

When the method is done, the Dictionary is returned to the E2G.WebAPI which returns the Dictionary to the HTTP call in the frontend where the Dictionary is parsed to a string and

saved to a variable ready to be exported.

### 5.3 Export generated subdivision data

This requirement is simple to fulfill. There is inbuilt JavaScript functionality for downloading files. The only important thing to understand is what exactly needed to be exported, which has been discussed in analysis section 3.3.3

```
238 //Utility function to download files
239 function downloadGeoJSON(content, fileName, contentType) {
240     var a = document.createElement("a");
241     var file = new Blob([content], { type: contentType });
242     a.href = URL.createObjectURL(file);
243     a.download = fileName;
244     a.click();
245 }
```

Figure 49: MarketSimSubdivideController.js download utility function.

Figure 49 shows an utility method that is gonna be used when pressing the export button. JavaScript has a Blob data structure that represents binary or text data. Using Blobs as input, we create an object url that's gonna be downloaded.

```
222 exportConfigBtn.onclick = function () {
223     var areas = subdividedFeaturesArray;
224     if (arrayNonDividedFeatures.length != 0) {
225         areas = areas.concat(arrayNonDividedFeatures);
226     }
227
228     //Correct the format needed for subdivision window
229     var featureCollection = {};
230     featureCollection.features = areas;
231     featureCollection.type = "FeatureCollection";
232
233     var configFile = "[" + JSON.stringify(featureCollection) + "," + exportPowerPlantString + "]"
234
235     downloadGeoJSON(configFile, 'exportconfig.json', 'text/plain');
236 }
```

Figure 50: MarketSimSubdivideController.js export button onclick function.

The first part of the config file is the featureCollection, which represents all the areas that should be exported. These areas are both the newly subdivided areas but can also be non subdivided areas that the user has selected from the PMS site. The second part is all the power plants that correspond to each area.

## 5.4 Import generated subdivision data

### 5.4.1 Button to import data

A button is created Inside the `_SimulationMenu.cshtml` which contains other features such as the existing export and import settings. The button to import subdivision data is created using the same `.cshtml` structure as the other buttons to keep the consistency. A code snippet can be seen on figure 51.

```
@{
    if (isSubdivisionOfArea == "True")
    {
        <div id="import-map-menu-option" class="general-settings-menu-entry">
            Importer subdivision data</label>
        </div><input type="file" id="Import_settings_Map_Selector" name="files[]" />
    }
}
```

Figure 51: `SimulationMenu.cshtml` code.

However, it makes sense that the button is only available when accessing the new subdivision version.

Another variable is introduced called `isSubdivisonOfArea` which contains a boolean of whether the user is accessing the new subdivision version which can be seen on figure 52. More information about the `isSubdivisonOfArea` can be seen in section 5.5.2.

```
@{
    var isDemo = Model.Content.GetVortoValue("IsDemoMode").ToString();
    var isSubdivisionOfArea = Model.Content.GetVortoValue("isSubdivisionOfArea").ToString();
}
```

Figure 52: `isSubdivisonOfArea` code.

This makes it possible to hide the button by checking if the version is the subdivision version. The figure 51, shows that the new add-on of the button consists of different tags which are `<div>` and `<input>` tags. The `div` is the element which the user is able to see in the UI. The `<input>` is hidden and is of the type “file”. Clicking on the input element, will bring up a file dialog. The input button is actually hidden using JavaScript in the frontend which can be seen on figure 53.

### 5.4.2 Importing data using button

Firstly, the button should have the same functionality as the other buttons on the menu.

```
if (isSubdivisionOfArea == "True") {  
  //Import subdivision option  
  var tmp = document.getElementById("import-map-menu-option");  
  $("#Import_settings_Map_Selector").hide();  
  
  tmp.onclick = function () {  
    $("#Import_settings_Map_Selector").trigger('click');  
  }  
  tmp.onmouseenter = function () {  
    document.getElementById("import-map-menu-icon-white").style.display = "none";  
    document.getElementById("import-map-menu-icon-yellow").style.display = "inline-block";  
  }  
  tmp.onmouseleave = function () {  
    document.getElementById("import-map-menu-icon-white").style.display = "inline-block";  
    document.getElementById("import-map-menu-icon-yellow").style.display = "none";  
  }  
}
```

Figure 53: MarketSimController.js code.

Figure 53 shows a code snippet from MarketSimController.js. The code snippet shows the use of the isSubdivisionOfArea variable and the initial functionality for the button. The isSubdivisionOfArea variable is important as it is used to check whether the user is accessing the subdivision version as it will give problems if the user accesses another version where the button is hidden as it will not exist.

The ID of the input button, that can be seen in figure 51, is used to hide the button using the JQuery "hide" function.

The div created in figure 51, uses the "onclick" event to check whether the div is being clicked which will trigger the input button and will bring up the file dialog.

Figure 54 shows an event added to the input tag. An if statement surrounds the code lines to check that the version accessed is the subdivision version.

```
if (isSubdivisionOfArea == "True") {  
  document.getElementById('Import_settings_Map_Selector').addEventListener('change', handleFileMapSelect, false);  
}
```

Figure 54: handleFileMapSelect code.

When a file is selected, the function "handleFileMapSelect" is called which will wait for the



data to be available and then parse the data when ready and then call a function to update the subdivided power plants and function to load the new subdivision map. This can be seen on figure 55.

```
function handleFileMapSelect(evt) {  
  var files = evt.target.files; // FileList object  
  var reader = new FileReader();  
  reader.onloadend = function (evt) {  
    if (evt.target.readyState == FileReader.DONE) {  
      var rawImport = evt.target.result;  
      var importData = JSON.parse(rawImport);  
      var mapData = importData[0];  
      var powerPlantData = importData[1];  
      map.updateSubdividedPlants(JSON.stringify(powerPlantData));  
      map.loadSubdivisionGeoJsonMap(JSON.stringify(mapData));  
    }  
  }  
}
```

Figure 55: handleFileMapSelect code 2.

The function `updateSubdividedPlants()` saves the power plants from the import file in an instantiated dictionary with the key being the name of the subdivided area and the value being a list of the corresponding power plants.

The other function “loadSubdivisionGeoJsonMap” can be seen on figure 56. The function uses Leaflet in-built functions to remove the current map and add each of the areas from the imported map. At the end, the “refreshGenericLabels” function is called which is an existing function that loads the area names of the subdivided areas.

```
self.loadSubdivisionGeoJsonMap = function (geojson) {  
  
  _map.removeLayer(_priceAreasLayer);  
  _geoJsonObject = JSON.parse(geojson);  
  _areas = JSON.parse(geojson);  
  
  _priceAreasLayer = L.geoJson(_geoJsonObject, {  
    style: styleGenerator,  
    onEachFeature: onEachFeature  
  }).addTo(_map);  
  
  self.refreshGenericLabels("-");  
}
```

Figure 56: updateSubdividedPlants code.



## 5.5 A separate version of the PMS

To fulfill requirement R5.1, a subdivide page was created using one of the existing customized document types of the Umbraco system which was MarketSimulatorPage.

The document type can be thought of as a class. The existing version of the PMS, consists of the full version and the demo version which are both an instance of the document type MarketSimulatorPage. A third version called SubdivisionOfArea was created using the very same template.

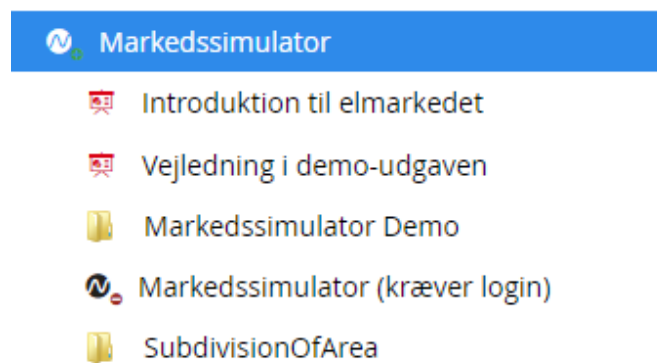


Figure 57: From Umbraco.

After creating the new instance of MarketSimulatorPage, there were a lot of properties that had to be filled as it was used and needed by the page. To determine which version of the PMS that was accessed another variable was added.

isSubdivisionOfArea ☒

Figure 58: Boolean value only for SubdivisionOfArea.

The variable is called “IsSubdivisionMode”. It is a boolean variable that can be used to check which version of the PMS that is making calls.

By creating the new page this way, we would still be able to access the implementations of the existing program and add further implementations specific to the current version accessed by checking whether the user is using the full version, the demo version or new PMS subdivision version.

### 5.5.1 Integrate with GetAreaPrice

This section fulfills requirement 5.2.2 to make the new PMS subdivision version be able to calculate the area prices using the imported subdivision data. This section will only go through the important parts of integrating the power plants data to be able to calculate the area prices of the subdivided areas.

To integrate with the current `GetAreaPrice()` function, the imported data needs to replace or work together with the current data of power plants as described in section 4.2.3.

In section 5.5, it was described how a variable for the PMS subdivision version was created. This variable is important as it is used to differentiate between the different versions making calls to the backend.

The `IsSubdivisionMode` value is queried from Umbraco which is then passed to the `MainController` of the PMS when it is instantiated.

The list of subdivided power plants described in 5.4.2 and the `isSubdivisionOfArea` is passed into the message body of the `GetAreaPrice` http request.

```
var listOfSubdividedPlants = new List<SystemPowerPlant>();
if (request.IsSubdivisionMode)
{
    foreach (var areaName in request.AreaNames)
    {
        if (priceAreasDict.ContainsKey(areaName) && !request.SubdividedPowerPlants.ContainsKey(areaName))
        {
            var powerPlantsForNonDividedAreas = _repositoryContext.MarketSimRepository.ListOperatingPowerPlantsByPriceArea(areaName, (int)request.SliderValues["Year"]);
            listOfSubdividedPlants.AddRange(powerPlantsForNonDividedAreas);
        }
        else
        {
            foreach (var powerplant in request.SubdividedPowerPlants[areaName])
            {
                powerplant.PriceArea = priceAreasDict[areaName];
            }
            listOfSubdividedPlants.AddRange(request.SubdividedPowerPlants[areaName]);
        }
    }
}
```

Figure 59: `isSubdivisionOfArea` code.

Figure 59 shows the controller method and endpoint which is requested by the HTTP request in the frontend. The `PriceAreaRequest` model is expanded to handle the boolean `IsSubdivisionMode` and the dictionary of subdivided power plants which can be seen on figure 60.

```
1 reference
public bool IsDemoMode { get; set; }
2 references
public Dictionary<string, List<SystemPowerPlant>> SubdividedPowerPlants { get; set; }
5 references
public bool IsSubdivisionMode { get; set; }
```

Figure 60: `IsSubdivisionMode` code.

Referring to Figure 59, it is checked whether the PMS subdivision version. If it is the PMS subdivision version making requests, then we will loop through each of the area names of which that are selected to perform the calculation on. Then we check whether the dictionary of all the known price areas of the PMS contains the selected area and that the dictionary of subdivided areas should not contain the selected area, because if the area name exists in the dictionary of the subdivided areas, then the area is a new subdivided area. This check is important because if the area is inside the price area dictionary and the area is not a key in the subdivided area dictionary, then we fetch all the power plants for the area which is placed into the a new list of power plants with the name “listOfSubdividedPlants”. Otherwise we use the power plants in the dictionary based on the area name which is added to the “listOfSubdividedPlants”. All the power plants in the list get their price area ID changed to a new one corresponding to the new subdivided area which is added to the area dictionary containing all the known areas. The list of the subdivided power plants are passed in the parameter of the next set of methods together with the boolean “IsSubdivisionMode”

```
if (isSubdivisionMode)
{
    plantsInScope = subdividedPowerPlants;
}
else
{
    plantsInScope = _repositoryContext.MarketSimRepository.ListOperatingPowerPlantsByPriceArea(priceAreaId, (int)userSelectedYear);
}
```

Figure 61: IsSubdivisionMode code 2.

Figure 61 shows a check using isSubdivisionMode. The variable “plantsInScope” is the variable which contains all the power plants used for the calculation of the area. This is where the subdivided power plants replace the power plants directly from the database.

```
foreach (var plant in plantsInScope)
{
    if (IsPlantIncludedInCalculation(plant, sliderValues, (int)sliderValues["Year"]))
    {
        if (plant.PlantName == "Randers (Randersvaerket)-new")
        {
            var tmppp = plant;
        }

        plant.MarginalCost = GetMarginalCost(plant, sliderValues, simpleCalculationConstants,
            complexCalculationConstants, fuelCalculationConstants);
        if (isSubdivisionMode)
        {
            plant.Production = GetPlantActualSizeSubdivision(plant, sliderValues);
            plant.NetCapacity = GetPlantActualSizeSubdivision(plant, sliderValues);
        }
        else
        {
            plant.Production = GetPlantActualSize(plant, sliderValues);
            //update plant size with actual size
            plant.NetCapacity = GetPlantActualSize(plant, sliderValues);
        }

        solverInputPowerPlants.Add(plant);
    }
}
```

Figure 62: GetPlantActualSize code.

In Figure 62 The method `GetPlantActualSize` calculates the “NetCapacity” and the “Production” based on a dictionary with some slider values for availability values such as solar, wind, water and power plant. The subdivided area did not exist in the dictionary so a new method, “`GetPlantActualSizeSubdivision()`” was created based on the code from the original method and modified to not use the values from the dictionary.

```
// Calculate max capacity from provided bubbles
if (solverInputPowerPlants.Any())
{
    double capacityForChosenArea = powerplantsforarea.Sum(item => item.Production);
    if (isSubdivisionMode)
    {
        priceArea.MaxConsumption = priceArea.Consumption = capacityForChosenArea *
                                                                    areaDefaultProductionProfile.Profile *
                                                                    0.8 * GetGrowthFactor(0.01,
                                                                    (int)sliderValues[YEAR_SLIDER_NAME]);
    }
    else
    {
        priceArea.MaxConsumption = priceArea.Consumption = capacityForChosenArea *
                                                                    areaDefaultProductionProfile.Profile *
                                                                    0.8 * GetGrowthFactor(
                                                                    areaConsumptionGrowthCoefficients[
                                                                    priceArea.AreaName],
                                                                    (int)sliderValues[YEAR_SLIDER_NAME]);
    }
}
}
```

Figure 63: areaConsumptionGrowthCoefficients code.

Figure 63 shows a code snippet where the code had to be extended with another check of `isSubdivisionMode`. The code snippet shows that a dictionary is used to get the “areaConsumptionGrowthCoefficients” for the area. This needed to be replaced with a hardcoded value of “0.01” which was decided together with the product owner. All the other places which needed a specific value from a dictionary and did not exist were replaced with a hardcoded value in decision with the product owner, so that it still gave a proper end result for the calculation of the area price.

The end result is returned to the frontend using as a `MarketSimulatorSolverResult` DTO which is automatically translated to a JSON object.

### 5.5.2 Integrate with GetAreaChartResult

This section deals with requirements 5.2.3 which defines the PMS subdivision version being able to perform the existing feature `GetAreaChartResult` using the existing features which is also described in section 3.3.5.2.

The endpoint being called from the frontend is “api/marketSim/GetAreaResult”. To be able to differentiate between the subdivision version and the others the variable “`isSubdivisionMode`” is passed together with the dictionary of power plants in the message body of the HTTP request to the `E2G.WebAPI`. This allows us to use the variable for multiple checks.

To enable ASP.NET to deserialize the message body, the `PowerPlantsBubblesRequest` model is expanded to handle the dictionary of power plants and the boolean “`IsSubdivisionMode`”

which can be seen on figure 64 below.

```
1 reference
public class PowerPlantBubblesRequest
{
    3 references
    public string PriceAreaName { get; set; }
    2 references
    public Dictionary<string, double> SliderValues { get; set; }
    1 reference
    public Dictionary<string, double> AreaConsumptionGrowthCoefficients { get; set; }
    1 reference
    public IEnumerable<UserPowerPlant> UserPowerPlants { get; set; }
    1 reference
    public IEnumerable<ConsumptionOverwriteItem> ConsumptionOverwriteItems { get; set; }

    1 reference
    public bool IsDemoMode { get; set; }

    1 reference
    public Dictionary<string, List<SystemPowerPlant>> SubdividedPowerPlants { get; set; }
    1 reference
    public bool IsSubdivisionMode { get; set; }
}
```

Figure 64: PowerPlantsBubblesRequest code.

The SubdividedPowerPlants dictionary and the IsSubdivisionMode variables are sent in the parameter of the method GetAllPowerPlantsBubble() that calculates the power plant bubbles on figure 65.

```
result.PowerPlantBubbles = _manager.GetAllPowerPlantsBubbles(
    request.PriceAreaName,
    request.SliderValues,
    request.UserPowerPlants,
    request.SubdividedPowerPlants,
    request.IsSubdivisionMode);
```

Figure 65: GetAllPowerPlantsBubble code.

The figure 66 below shows a code snippet where the “isSubdivisionMode” variable is being used for a check. We check that a slider value for the operating year set in the frontend is existing and also that the dictionary of subdivided plants contains the name of the area that is in question. The other change is implementing the method GetPlantActualSizeSubdivision() which is a method that has been copied from the original method GetPlantActualSize(). This is briefly explained in below Figure 62.

The method will calculate the different properties of the power plant and add the end result in

a list of finished power plant bubbles.

```

else if(isSubdivisionMode)
{
    includedpriceareas.Add(priceArea, priceAreasDict[priceArea]);

    var sliderValuesContainsOperatingYear = sliderValues.ContainsKey(YEAR_SLIDER_NAME);
    //Area bubbles for subdivided areas
    if (sliderValuesContainsOperatingYear && SubdividedPowerPlants.ContainsKey(priceArea))
    {
        foreach (var plant in SubdividedPowerPlants[priceArea])
        {
            if (IsPlantIncludedInCalculation(plant, sliderValues, (int)sliderValues[YEAR_SLIDER_NAME]))
            {
                MarketSimulatorPowerPlantBubble plantBubble = new MarketSimulatorPowerPlantBubble
                {
                    PlantName = plant.PlantName,
                    MarginalCost = this.GetMarginalCost(plant, sliderValues, simpleCalculationConstants,
                        complexCalculationConstants, fuelCalculationConstants),
                    Capacity = this.GetPlantActualSizeSubdivision(plant, sliderValues),
                    FuelType = plant.Type,
                    Year = plant.CommissionYear,
                    TechnologyType = (EnergyTechnology)plant.Technology
                };
                list.Add(plantBubble);
            }
        }
    }
}

```

Figure 66: GetPlantActualSizeSubdivision code.

Another modification that had to be made was checking whether the area in question has any power plants in the database which can be seen on figure 67.

```

//Get plants in case of non subdivided areas
var areaHasPlants = _repositoryContext.MarketSimRepository.ListOperatingPowerPlantsByPriceArea(
    includedpriceareas, (int)sliderValues[YEAR_SLIDER_NAME]).Any();

//Perform area bubbles for non divided areas
if (areaHasPlants)
{
    foreach (var plant in _repositoryContext.MarketSimRepository.ListOperatingPowerPlantsByPriceArea(
        includedpriceareas, (int)sliderValues[YEAR_SLIDER_NAME]))
    {
        if (IsPlantIncludedInCalculation(plant, sliderValues, (int)sliderValues[YEAR_SLIDER_NAME]))
        {
            MarketSimulatorPowerPlantBubble plantBubble = new MarketSimulatorPowerPlantBubble
            {
                PlantName = plant.PlantName,
                MarginalCost = this.GetMarginalCost(plant, sliderValues, simpleCalculationConstants,
                    complexCalculationConstants, fuelCalculationConstants),
                Capacity = this.GetPlantActualSizeSubdivision(plant, sliderValues),
                FuelType = plant.Type,
                Year = plant.CommissionYear,
                TechnologyType = (EnergyTechnology)plant.Technology
            };
            list.Add(plantBubble);
        }
    }
}

```

Figure 67: areaHasPlants code.

This modification was important because if the user uploads a config file which contains the

subdivided areas and a non-subdivided area and the user wanted to see the power plant bubbles for non-subdivided area in the subdivision version, then it would be possible because the power plants would be retrieved from the database.

After calculating the required properties for each power plant, they are added to a list of the which is returned to the E2G.WebAPI and sent back to the frontend as a MarketSimulator-BubbleChartResult DTO. The DTO can be seen on the figure 68.

```
3 references
public class MarketSimulatorBubbleChartResult
{
    2 references
    public IEnumerable<MarketSimulatorPowerPlantBubble> PowerPlantBubbles { get; set; }
    1 reference
    public double? BaseAreaConsumption { get; set; }
    1 reference
    public int AreaPowerPlantCount { get; set; }
}
```

Figure 68: MarketSimulatorBubbleChartResult code.



## 6 Verification

This section is about verifying if the requirements that we have set during the project have been fulfilled. Different kinds of tests have been set up such as manual acceptance testing, JMeter performance testing and unit testing to verify the requirements.

### 6.1 Validation and review

By using Scrum, we have worked iteratively and thus been able to iteratively develop our product and frequently get feedback from the product owner to clear up any misunderstandings and verify the implementations of the requirements listed in section 2.2 by seeking input and validation from the product owner. In the last sprint of Scrum, we held a meeting with the Product Owner to verify that the product held up with his expectations and discussed the product scope and future implementations which is explained in section 7.1.

### 6.2 Manual acceptance testing

During the development of the requirements, we performed manual acceptance testing based on acceptance test scenarios. These scenarios were created on the basis of the detailed use cases created in the analysis section.

Before merging the feature branches into the develop branch, we performed a series of test scenarios to verify that the existing features had not been broken by the new implementations. If the test cases did not pass, we made sure to find the problem and fix the problem before merging the feature branch into develop and retested the scenarios.

After the last sprint of development, before merging into the master branch, we performed the test case scenarios again to ensure that the scenarios passed. Figure 69 shows the scenarios based on use cases, with the condition, test case, expected results and the final results.

| Test ID | Use case   | Precondition  | Test case   | Expected results  | Test results |
|---------|--|---|---|---|--------------|
| 1       | The user must be able to draw polygons on a map                            | 1.The user must be on the subdivision editing view  | The user clicks on the polygon figure and draws a polygon   | A polygon should be drawn on the map                                  | Passed       |
| 2       | The user must be able to divide a country into one or more subareas.       | 1. The user is on the subdivision editing view  | The user selects the area to subdivide and draws a polygon on the area having 2 intersection points and clicks the subdivide button | The area should have been split into 2 areas with different names     | Passed       |
| 3       | The user must be able to export data                                       | 1. The user is on the subdivision editing view<br>2. The user has subdivided an area  | The user clicks on the export button  | A config file should be downloaded onto the user's machine            | Passed       |
| 4       | The user must be able to import the exported subdivision data              | 1. The user is on the subdivision version of PMS<br>2. The user must have the correct config file                                       | The user clicks import map button and selects the correct config file   | The map should change and data of the power plants should be uploaded | Passed       |
| 5       | The user can calculate the area prices of the areas on the map             | 1. The user is on the subdivision version of PMS<br>2. The user must have uploaded the config file<br>3. The user must select a country | The user clicks on Fast Calculation button  | The user should see the area price for the selected area              | Passed       |
| 6       | A chart can be shown that represents the area prices of the selected areas | 1. The user is on the subdivision version of PMS<br>2. The user must have uploaded the config file                                      | The user clicks on right-clicks an area with the mouse  | The user should see all the power plants for the selected area        | Passed       |

Figure 69: Scenario test case table.

### 6.3 Validation of Performance and reliability

JMeter is used to perform load testing and measure the performance and the reliability of the backend of the system and more specifically on the newly created subdivision component.

Since we were working on an existing project, we did not have much influence on the non-functional requirements of the system. The project's requirements were therefore implemented while keeping the architectural structure of the system.

Therefore, the best way to verify the performance of the system is to establish a baseline measuring the performance of the existing features and compare them with the implementation of the new feature.

The endpoints that were tested for this comparison were the new endpoint "api/subdivision/-Subdivide" and the existing "api/marketSim/GetAreaPrice".

### 6.3.1 Performance

Performance is about time and the ability of the system to meet the timing requirements. Every system has a performance requirement and since the performance requirements has not been explicitly stated by the stakeholder, we defined in NF2, that the implementation of the new subdivision feature should run as fast as the other features of the system with a small deviation allowed.

Therefore a performance baseline is established by load testing the get area price method to provide a reference point for comparison when evaluating the new feature.

The first test scenario will be based on 1000 users each making a request to the backend with a ramp-up period of 1 second. This means the 1000 requests will be distributed over 1 second which is very demanding.

Table 10: 1000 requests and ramp up time of 1 second.

| Features     | Get Area Price (baseline) | Subdivide area |
|--------------|---------------------------|----------------|
| Samples      | 1000                      | 1000           |
| Average (ms) | 3238                      | 5452           |
| Median (ms)  | 3302                      | 5395           |
| 90%          | 5554                      | 9543           |
| 95%          | 5808                      | 10032          |
| 99%          | 5938                      | 10297          |
| Max (ms)     | 82                        | 77             |
| Min. (ms)    | 5973                      | 10327          |
| Error %      | 0,00                      | 0,00           |
| Throughput   | 143,9/sec                 | 88,4/sec       |

Analyzing the table 10, the existing feature has a lower latency than the newly implemented features. The important numbers are 99% percentile. The existing feature `GetAreaPrice()` has a 5938 ms latency in the 99% percentile which means 99% of users can expect their requests to be completed within 5938 ms which is not bad considering how many requests the system was receiving in such a short amount of time.

Looking at the 99% for the subdivide area function, it guarantees that 99% of users will have their request fulfilled within 10327 ms which is a lot slower than `GetAreaPrice`. However, looking at the minimum latency for the subdivide area, it is actually faster than `GetAreaPrice` by a small margin.

The next test scenario will make 1000 requests but with a ramp-up time of 60 seconds.

Table 11: 1000 requests with a ramp-up time of 60 seconds.

| Features     | Get Area Price | Subdivide area |
|--------------|----------------|----------------|
| Samples      | 1000           | 1000           |
| Average (ms) | 33             | 22             |
| Median (ms)  | 29             | 21             |
| 90%          | 38             | 25             |
| 95%          | 63             | 29             |
| 99%          | 113            | 59             |
| Max (ms)     | 203            | 98             |
| Min. (ms)    | 25             | 14             |
| Error %      | 0,00           | 0,00           |
| Throughput   | 16,7/sec       | 16,7/sec       |

The table shows that the subdivide area function is actually faster than calculating the area price. For the get area price, the 99% percentiles says that 99% of users can expect to have their requests fulfilled within 113 ms which is very fast compared to the other test scenario. The subdivide area function is faster than the Get Area Price with a 99% of 59 ms.

For both table 1 and table 2, the focus is on the average latency for the requests as it is the variable being assessed and compared when evaluating whether performance of the subdivision features pass the test scenarios.

The table 12 below, concludes the results for the test scenarios based on comparing the average latency for a request with an allowed deviation of 5%.

Table 12: Average latency for a request.

| Test case                               | Verification |
|---|--------------|
| Test case 1 (1000 requests, 1 second)   | Failed       |
| Test case 2 (1000 requests, 60 seconds) | Passed       |

The performance non-functional requirement passed the first test case and passed the second test case.

Diagrams and graphs for the test scenarios can be seen in the appendix 10.2.2.

### 6.3.2 Reliability

Reliability is about the ability of the system to carry out its functions consistently and without failures. The reliability of the system has not been explicitly stated by the stakeholder, but as with performance every system will have reliability requirements. We decided the reliability of the system should be based on the subdivision feature being able to carry out 99,5% of its requests out of 10000 requests.

A test scenario is set up in JMeter to perform 10000 requests with a ramp up time of 300 seconds.

Table 13: For reliability.

| Features     | Get Area Price | Subdivide area |
|--------------|----------------|----------------|
| Samples      | 10000          | 10000          |
| Average (ms) | 53             | 27             |
| Median (ms)  | 46             | 24             |
| 99%          | 165            | 67             |
| Max (ms)     | 580            | 196            |
| Min. (ms)    | 8              | 14             |
| Error %      | 0,08           | 0,02           |

The table above sums up the results. The existing feature GetAreaPrice has an error percentage of 0,08% and the Subdivide Area has an error percentage of 0,02%. As stated in the NF1 requirement the system had to be able to carry out 99,5% of its requests which means that the subdivide area passes the test scenario and the reliability requirement is verified based on this specific test scenario.

A more detailed diagram of the results can be seen in appendix 10.2.3.

## 6.4 Validation of Modifiability and Maintainability

Validating for modifiability and maintainability means analyzing the cyclomatic complexity, depth of inheritance and class coupling of the feature.

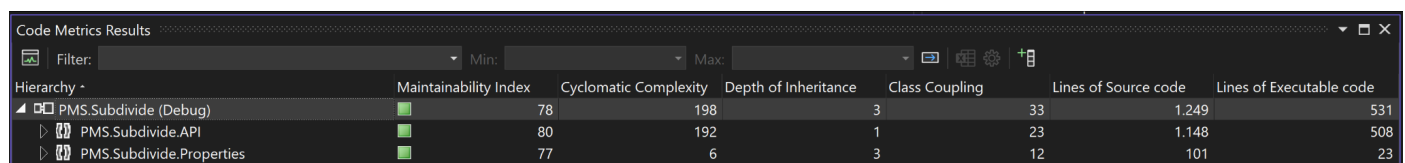
Cyclomatic complexity is the amount of different execution paths and is used to measure the maintainability of a system. A decision point, such as control flow such as a loop or a if-else statement, adds to the complexity of the program. Higher complexity leads to a higher probability of errors occurring, which makes the code harder to maintain.

The depth of inheritance measures the level of parent-child relationships of the feature. A deep inheritance makes the code harder to understand, which makes it less modifiable.

Class coupling measures how dependent one class is to another. Tight coupling means there is a strong interdependent relationship between classes, where one change will heavily affect the other. More preferred is loose coupling, which makes the program more modifiable, flexible and reusable, since they are independent of other classes.

The design tried to achieve modifiability and maintainability by encapsulating all the business logic of the subdivision feature inside its own module. Furthermore, as seen in the diagram architecture layer in section 3.1.1 the architecture is preserved, where the module is only communicating with the layers below it, which lessens the coupling.

Visual Studio has inbuilt code analysis features that calculates metrics such as maintainability index, cyclomatic complexity, depth of inheritance and class coupling.



| Hierarchy                  | Maintainability Index | Cyclomatic Complexity | Depth of Inheritance | Class Coupling | Lines of Source code | Lines of Executable code |
|----------------------------|-----------------------|-----------------------|----------------------|----------------|----------------------|--------------------------|
| ▲ PMS.Subdivide (Debug)    | 78                    | 198                   | 3                    | 33             | 1,249                | 531                      |
| ▶ PMS.Subdivide.API        | 80                    | 192                   | 1                    | 23             | 1,148                | 508                      |
| ▶ PMS.Subdivide.Properties | 77                    | 6                     | 3                    | 12             | 101                  | 23                       |

Figure 70: Visual Studio Code Analysis.

Figure 70 shows the result of the Visual Studio code analysis. There are no fixed values for the metrics to determine whether or not the feature is modifiable. It is a case-on-case basis depending on the feature and the organization's standards. These are the advised values from National Institute of Standards [5] [14] of a complexity limit of 10, Chidamber in "A metrics suite for object oriented design" [3] advises class coupling of most 9 and Shatnawi in "A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems" [12] advises a depth of inheritance of most 6.

The depth of inheritance is acceptable but the other metrics are below standard. The rea-

son the class coupling is quite high is because of the use of libraries such as GeoLibrary and GeoJSON. However these are required to work with the interactive map and geometric objects. Therefore the class coupling can be viewed as acceptable. However, the cyclomatic complexity is way above the advised standard. The code contains mathematical formulas such as linear intersection, and using the GeoJSON data as points in the formula creates many decision points. The code will therefore naturally be very complex.

The metrics show that the feature is hard to maintain because of the code complexity, but the acceptable depth of inheritance and coupling makes the feature modifiable. The complexity does however also make it harder to modify since the code is harder to understand.

## 6.5 Validation of Functional requirements

### 6.5.1 Unit testing

Most of the logic concerning the subdivision calculation has been tested via unit testing. Given that the division results are fairly visual, a simple test example has been created where the presumed results can be visually seen.

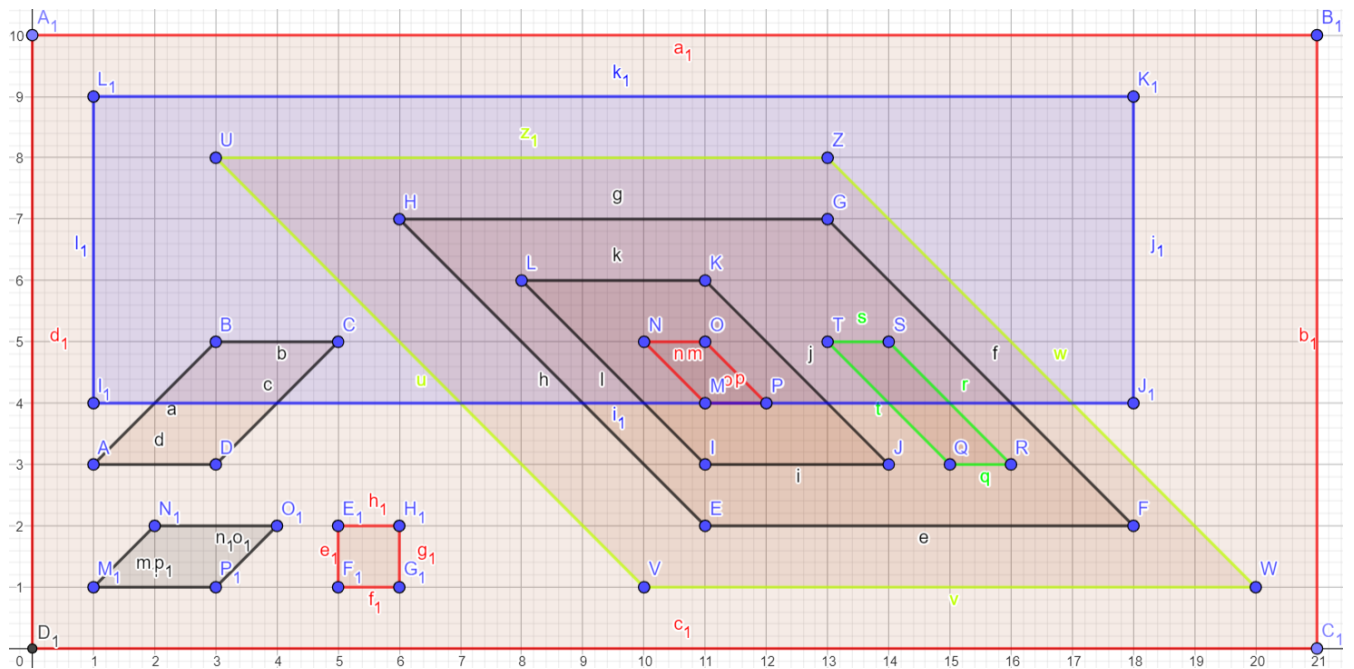


Figure 71: Visualization of the unit test example.

The example consists of multiple test examples. The polygons with black lines in figure 71 represent a Multipolygon that has to be divided. The red polygons represent a drawn cut that will not result in a division, green will drill a new hole into the MultiPolygon and yellow will

split the MultiPolygon into two. Finally the blue polygon will cut through the MultiPolygon to create two MultiPolygons. It is possible to see how many cuts there are and what their coordinates are.

#### 6.5.1.1 lineIntersectionTest and distOf2PointsTest

The mathematical methods have been tested using decimal numbers and calculated manually to compare the results.

Assert.AreEqual failed. Expected:<3,69000005722046>. Actual:<3,69>.

Figure 72: calculating decimals.

Both the distOf2Points and the lineIntersection methods show the computational problem with calculating decimals.

```

1 public void lineIntersectionTest()
2 {
3     List<List<float>> l1 = new List<List<float>> {
4         new List<float> { 7.37f, 15.06f },
5         new List<float> { -1.45f, 9.36f }
6     };
7     List<List<float>> l2 = new List<List<float>> {
8         new List<float> { -2.34f, 15.13f },
9         new List<float> { 8.82f, 10.6f }
10    };
11
12
13    List<float> result = subdiv.lineIntersection(l1, l2);
14
15
16    List<float> expected = new List<float> { 3.69f, 12.68f };
17
18
19    Assert.AreEqual(expected[0], (float)(result[0] - (result[0] % 0.01)), "wrong " + (result[0] - (result[0] % 0.01)) + " is not " + expected[0]);
20    Assert.AreEqual(expected[1], (float)(result[1] - (result[1] % 0.01)), "wrong " + (result[1] - (result[1] % 0.01)) + " is not " + expected[1]);
21 }

```

Figure 73: lineIntersection unit test.

The results are therefore truncated to adjust for the margin of error, in a similar way that the system does it.

#### 6.5.1.2 noIntersectionSolverTest

The 3 cases where no intersections accrue are tested using the example in figure 5.



```

1 public void noIntersectionSolverTest()
2 {
3     List<List<List<List<List<float>>>>>> split = subdiv.noIntersectionSolver(testMultiPolygon, testSplitPolygon);
4
5
6     Assert.IsTrue(split[0][0][0][0].SequenceEqual(testMultiPolygon[1][0][0]));
7     Assert.IsTrue(split[0][0][1][0].SequenceEqual(testMultiPolygon[1][1][0]));
8     Assert.IsTrue(split[1][0][0][0].SequenceEqual(testMultiPolygon[0][0][0]));
9     Assert.IsTrue(split[1][1][0][0].SequenceEqual(testMultiPolygon[2][0][0]));
10 }

```

Figure 74: noIntersectionSolver unit test.

The test works by checking if the given polygon lists have been distributed correctly into the resulting list. The test in figure 74 showcases the yellow cutting polygon from figure 71, but the green and red have also been tested the same way.

### 6.5.1.3 allCutDataFinderTest

The ability to find cut points have been tested by writing up the cut points between the blue and black polygons from figure 71.

```

1 public void allCutDataFinderTest()
2 {
3     List<CutDataClass> cutData = subdiv.getCutData(testMultiPolygon, testCutPolygon);
4
5     Assert.AreEqual(6, cutData.Count, "wrong");
6
7
8     Assert.IsTrue(cutData[0].cutPoint.SequenceEqual(expectedCut1) || cutData[1].cutPoint.SequenceEqual(expectedCut1) ||
9     | cutData[2].cutPoint.SequenceEqual(expectedCut1) || cutData[3].cutPoint.SequenceEqual(expectedCut1) ||
10    | cutData[4].cutPoint.SequenceEqual(expectedCut1) || cutData[5].cutPoint.SequenceEqual(expectedCut1)
11    );
12 }

```

Figure 75: allCutDataFinderTest unit test.

It is asserted that the correct amount of cuts have been found and that they all have the expected coordinates.

### 6.5.1.4 allIntersectionAdderMainTest and allIntersectionAdderNewTest

It is tested if the found cut points are added correctly to the polygon list. Since it is known in what sequence the polygons in figure 71 have been drawn in, first A then B and so on, it will be possible to predict their indexes in the list.

```
1 public void allIntersectionAdderMainTest()
2 {
3     List<CutDataClass> cutData = subdiv.getCutData(testMultiPolygon, testCutPolygon);
4     List<List<List<List<float>>>> result = subdiv.allIntersectionAdderMain(testMultiPolygon, cutData);
5
6
7     Assert.IsTrue(result[0][0][1].SequenceEqual(expectedCut1));
8     Assert.IsTrue(result[0][0][4].SequenceEqual(expectedCut2));
9
10
11     Assert.IsTrue(result[1][0][2].SequenceEqual(expectedCut6));
12     Assert.IsTrue(result[1][0][5].SequenceEqual(expectedCut3));
13
14
15     Assert.IsTrue(result[1][1][2].SequenceEqual(expectedCut5));
16     Assert.IsTrue(result[1][1][5].SequenceEqual(expectedCut4));
17 }
```

Figure 76: allIntersectionAdderMainTest unit test.

Something notable here is that expectedCut1-6 are formatted so that 1 is the point on the far left and 6 on the far right. In the list the point 6 comes before 3, because the list starts in E and moves to F, G and H and therefore the cut 6 is discovered first. The allIntersectionAdderNew method is tested the same way, however here the expectedCut1-6 comes in order.

#### 6.5.1.5 fixCutDataTest

It is tested if all the cut points receive all the correct data. Every point contains a lot of data, again given that the sequence of drawing is known, it can be predicted what the data should be.

```

1  public void fixCutDataTest()
2  {
3      List<CutDataClass> cutData = subdiv.getCutData(testMultiPolygon, testCutPolygon);
4      List<List<List<List<float>>>> mainWithCuts = subdiv.allIntersectionAdderMain(testMultiPolygon, cutData);
5      List<List<float>> newWithCuts = subdiv.allIntersectionAdderNew(testCutPolygon, cutData);
6      List<CutDataClass> fixedCutData = subdiv.fixCutData(cutData, newWithCuts);
7
8
9      Assert.IsTrue(fixedCutData[5].cutPoint.SequenceEqual(expectedCut4));
10     Assert.IsTrue(fixedCutData[5].isHole);
11     Assert.IsTrue(fixedCutData[5].cutListIndex == 5);
12     Assert.IsTrue(fixedCutData[5].insideList == mainWithCuts[1][1]);
13     Assert.IsTrue(fixedCutData[5].insidePreCoordinate.SequenceEqual(testMultiPolygon[1][1][3]));
14     Assert.IsTrue(fixedCutData[5].insideIndex == 5);
15     Assert.IsTrue(fixedCutData[5].insideNextCutIndex == 2);
16     Assert.IsTrue(fixedCutData[5].cutListInsideNextCutIndex == 4);
17     Assert.IsFalse(fixedCutData[5].insideUsed);
18     Assert.IsTrue(fixedCutData[5].outsidePreCoordinate.SequenceEqual(testCutPolygon[0]));
19     Assert.IsTrue(fixedCutData[5].outsideIndex == 4);
20     Assert.IsTrue(fixedCutData[5].outsideNextCutIndex == 5);
21     Assert.IsTrue(fixedCutData[5].cutListOutsideNextCutIndex == 4);
22     Assert.IsFalse(fixedCutData[5].outsideUsed);
23 }

```

Figure 77: fixCutData unit test.

All cut points are tested, but the chosen point in figure 71 is the point with coordinate [10,4] in figure 71. It is first asserted that the point has the correct coordinates, the correct index in the correct list and that it is a hole. The point needs to direct to the next cut in the polygon list, which should be in index 2, as explained earlier. This point should also appear in the list of cuts under index 4. It also directs to the next point in the drawn polygon, which should be in index 5, and in the list of cuts it should also be in index 4.

#### 6.5.1.6 cutAreaTest and unchangedAdderTest

To test if the division with cuts works, an expected outcome of the division from figure 71 is set up.

```

1  public void cutAreaTest()
2  {
3      List<CutDataClass> cutData = subdiv.getCutData(testMultiPolygon, testCutPolygon);
4      List<List<List<List<float>>>> mainWithCuts = subdiv.allIntersectionAdderMain(testMultiPolygon, cutData);
5      List<List<float>> newWithCuts = subdiv.allIntersectionAdderNew(testCutPolygon, cutData);
6      List<CutDataClass> fixedCutData = subdiv.fixCutData(cutData, newWithCuts);
7
8      List<List<List<List<List<float>>>>> dividedArea = subdiv.cutArea(mainWithCuts, newWithCuts, fixedCutData);
9
10     for (int i = 0; i < dividedArea[0].Count; i++)
11     {
12         for (int j = 0; j < dividedArea[0][i].Count; j++)
13         {
14             for (int k = 0; k < dividedArea[0][i][j].Count; k++)
15             {
16                 Assert.IsTrue(dividedArea[0][i][j][k].SequenceEqual(expectedDivision1[i][j][k]));
17             }
18         }
19     }
20     for (int i = 0; i < dividedArea[1].Count; i++)
21     {
22         for (int j = 0; j < dividedArea[1][i].Count; j++)
23         {
24             for (int k = 0; k < dividedArea[1][i][j].Count; k++)
25             {
26                 Assert.IsTrue(dividedArea[1][i][j][k].SequenceEqual(expectedDivision2[i][j][k]));
27             }
28         }
29     }
30
31     Assert.AreEqual(2, dividedArea[1].Count);
32 }

```

Figure 78: cutArea unit test.

Every single point in the result is compared to the expected result. The cutArea() method ignores everything that does not have a cut point, so it is asserted that the list only contains 2 polygons. The unchangedAdder() method is then also tested to see that the final result contains all 3 polygons.

### 6.5.1.7 mapPowerPlantsToCorrectCountryTest()

```
[TestMethod()]
0 references
public void mapPowerPlantsToCorrectCountryTest()
{
    string DK1 = "{\"type\":\"MultiPolygon\",\"coordinates\":[[[[10.87276,56.30318],[8.131143,56.28619],[8.131143,56.28619],[8.122404,56.28619],[8.122404,56.30318],[10.87276,56.30318]]]]}";
    string DK2 = "{\"type\":\"MultiPolygon\",\"coordinates\":[[[[10.87276,56.30318],[8.131143,56.28619],[8.131143,56.28619],[8.1338,56.28619],[8.1338,56.30318],[10.87276,56.30318]]]]}";
    string DE1 = "{\"type\":\"MultiPolygon\",\"coordinates\":[[[[14.96835,51.1085],[6.198476,51.45246],[6.198476,51.45246],[6.193262,51.45246],[6.193262,51.1085],[14.96835,51.1085]]]]}";

    var dictionaryWithAreas = new Dictionary<string, string> {
        { "DK1", DK1 },
        { "DK2", DK2 },
        { "DE1", DE1 }
    };

    var powerPlants = new List<SystemPowerPlant>
    {
        {new SystemPowerPlant { PlantName = "DE1", CityLatitude = (float)52.5551567, CityLongitude = (float)13.4000453}},
        {new SystemPowerPlant { PlantName = "DE1", CityLatitude = (float)53.57663439, CityLongitude = (float)12.09602037}},
        {new SystemPowerPlant { PlantName = "DE1", CityLatitude = (float)52.909599516, CityLongitude = (float)11.90959951}},

        {new SystemPowerPlant { PlantName = "DK1", CityLatitude = (float)56.4902977, CityLongitude = (float)8.209207}},
        {new SystemPowerPlant { PlantName = "DK2", CityLatitude = (float)56.49192, CityLongitude = (float)10.01999}},
        {new SystemPowerPlant { PlantName = "DK1", CityLatitude = (float)56.7040138, CityLongitude = (float)8.984131}},

        {new SystemPowerPlant { PlantName = "DK2", CityLatitude = (float)55.768055, CityLongitude = (float)9.931248}},
        {new SystemPowerPlant { PlantName = "DK2", CityLatitude = (float)56.1482162, CityLongitude = (float)10.2002916}},
        {new SystemPowerPlant { PlantName = "DK2", CityLatitude = (float)55.57478, CityLongitude = (float)8.591777}},
    };

    var result = subdiv.MapPowerPlantToCorrectCountry(dictionaryWithAreas, powerPlants);

    //All three areas should contain 3 power plants each
    Assert.AreEqual(result["DK1"].Count(), 3);
    Assert.AreEqual(result["DK2"].Count(), 3);
    Assert.AreEqual(result["DE1"].Count(), 3);

    //All plantname for the power plants in the area should be the same as the area name
    Assert.IsTrue(result["DK1"].Where(x => x.PlantName == "DK1").Count() == 3);
    Assert.IsTrue(result["DK2"].Where(x => x.PlantName == "DK2").Count() == 3);
    Assert.IsTrue(result["DE1"].Where(x => x.PlantName == "DE1").Count() == 3);
}
```

Figure 79: mapPowerPlantsToCorrectCountryTest().

The figure shows the unit test for testing the requirement R2.4. DK1, DK2 and DE1 are three different strings representing different areas on a map which are added to a dictionary.

powerPlants instantiates a list of 9 power plants which each contain a different set of GIS coordinates. These power plants have been tested and checked that they intersect with the areas in the dictionary so each area should have three power plants each.

The first three assertions, checks that the result, from the method MapPowerPlantToCorrectCountry, each area has three power plants. This is to check that the power plants are mapped correctly with the areas and that the intersection check works.

The next three assertions, check that all the power plants in the corresponding lists for the areas actually are the ones that should be associated with the specific area based on the name of the power plant.

## 7 Discussion

This chapter includes discussions and reflections on the project and how well the final product reflects the product objectives.

### 7.1 The Product Backlog

It is not uncommon for some items in the product backlog not to be implemented when using Scrum. One of items that had to be deprioritized is transmission capacity R5.2.1, which had to be cut because of time constraints. Setting transmission capacity for the subdivided areas is an important feature, when it comes to calculating the price area of multiple countries, as it restricts how much electricity is allowed to pass from one area of the subdivision to another area, which affects the calculation of the prices for the subdivided areas.

Another feature that has been affected by the time constraint is calculating the area prices of the subdivided price areas R5.2.2. The feature has been simplified in terms of only being able to calculate one subdivided area at a time. Setting transmission capacity for the subdivided areas and being able to calculate the area price for multiple areas at the same time, are closely related, because to make transmission capacity useful, the user needs to be able to calculate multiple areas at a time, to consider the transmission capacity between the subdivided areas in the calculation. This means they both have to be implemented at the same time, which did not fit into the time schedule of our project plan.

These decisions were made in consultation with the product owner while considering the best interest of the project's objectives. However, they will be a part of the future project scope.

### 7.2 UX and usability

This project did not focus on the UX aspect of the feature. This project is more meant for a proof of concept, to show the feature working with the existing system. Therefore, the user experience was not taken into consideration. However, in the next iterations this should be put higher on the priority list. These are some of the tools and techniques that could be used to improve the UX. Personas, that describe potential users of the system, to get a better understanding of the potential end user of the system. Scenarios, which build upon the personas, to put the product in a realistic scenario and make the product more user focused. User testing could be done measuring performance metrics such as task success, error effectiveness, time on

task, efficiency and learnability. A and B testing could be added to compare different versions of the site. The user testing should be done with at least 10 people, and these people should correspond to the personas defined earlier. These tools and techniques might be too much for one single feature inclusion. However, the system as a whole has not undergone any UX and usability testing before. Therefore the aforementioned points would help the system as a whole.

## 7.3 Manual testing

Manual testing was done for the functionality that represented the detailed use cases. Manual testing is not always ideal, since they are human error prone. However, for our functionality that includes visual confirmation that the detailed use case works as intended, for example “The user must be able to draw polygons on the map”, a manual test works fine. The current system does not have any automated tests, which meant there was no precedent to work on. Automated tests are good since, you will be able to run a script that checks if everything works as intended. These scripts can be run anytime commits or merge requests happen, to make sure that the changes did not cause any errors. However with the time constraint a pipeline for automatic testing was not prioritized, it was decided that manual testing would suffice for the tests listed in section 6.2.

## 7.4 Evaluation of Requirements

### 7.4.1 Functional Requirements

The subdivision feature mostly works but it still has some issues.

#### 7.4.1.1 More than 2 cuts in one polygon

As the analysis 3.3.2.4 describes, the feature should be able to do multiple cuts in 1 polygon at once. This has not been implemented. As stated the divider is able to tell if a cut point has been used once or twice. The problem however is that this isn’t enough information to solve this issue. The example in figure 18 would be solvable given that 3 directs towards 4, 4 directs towards 1 but has been used twice, then directs at 2 that also has been used twice, and 2 directs at 3. Thereby 3 and 4 can direct to each other given that they both have been used once.

The issue arises when there are even more cuts. In the example in figure 80, the algorithm would use cut 1 and 2 twice, the same way as in analysis x, but this time cut 3, 4, 5 and 6 will be used once. This means that cut 4 will direct to cut 5 and see it as the next step, when it

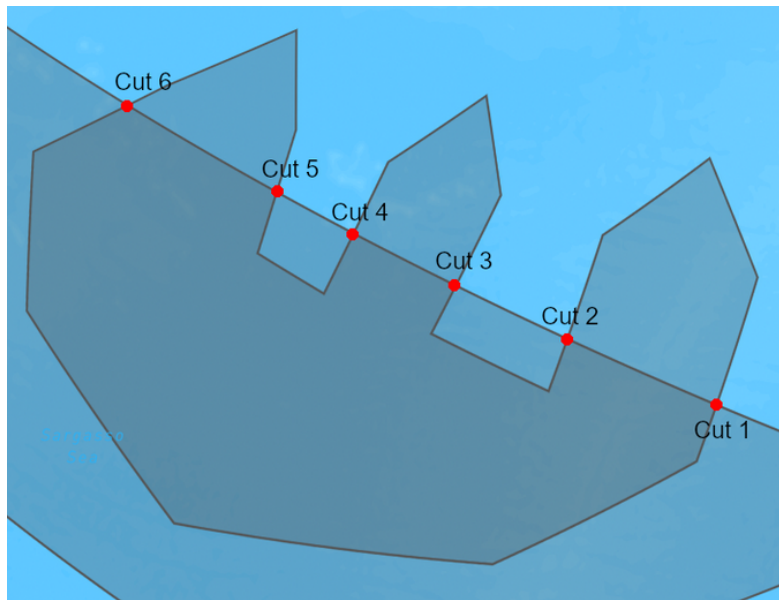


Figure 80: Problematic division.

should see 3 as the next step. A more elegant way of solving this problem was never found. One idea could be to do a collision check on the center between 4 and 5. This would indicate that the algorithm should not consider 5 as the next step, but it is still unclear how to reach 3 in a secure way from this point. One way might be to have all points point both forwards and backwards.

Given more time this issue would be the main focus of bug fixing.

#### 7.4.1.2 Edge case of drilling

As analysis 3.3.2.4 describes, the divider should be able to drill in a way that would combine 2 holes into 1. This case has not been solved, mostly due to time constraints. Out of all the cases this is the most niche case and has therefore been seen as the least important. Functionally it should not be too difficult to implement given more time.

#### 7.4.1.3 Additional edge case

During development a new case was discovered. If the user draws a drill case that encases an existing hole, the hole should be removed from the polygon and added to the drawn polygon. This should be simple to implement, however the problem was discovered very late and has therefore not been implemented.

#### 7.4.1.4 GIS coordinates of power plants

During one of the manual acceptance testing routine section 6.3 a problem was found. While performing the test scenarios in 6.3, more specifically the scenario with test id 6, we found out



that we could not find the charts containing the power plants for one of halves of Germany when subdividing the area in half.

All the power plant units contain a set of GIS coordinates and we found out that all the power plants units in Germany had the same set of coordinates resulting in all the power plants being in one spot of Germany. This meant if we divided the country in half, one half of the area would contain all the power plants and the other one would not have any at all. This was not only the case for Germany, it was also a problem in other countries such as the Czech Republic. This is a problem which can be fixed by giving all the power plants the correct coordinates. Another solution would be by sorting all the power plants in Germany depending on their capacity and mapping the portion of power plants to the other half of Germany, which do not have any power plants, corresponding to the area size of the subdivision. This would give imprecise results, when calculating the area prices, but the user would still be able to perform the existing functionality and experiment further.

However, the requirement R2.4 works properly, if the power plants in the area in question have the correct GIS coordinates which is also verified with the unit test in section 6.5.1.

## 7.4.2 Non-Functional Requirements

Performance and reliability were both non-functional requirements that were verified through test scenarios and using JMeter to perform the test scenarios. The test scenarios gave insight into how well the non-functional requirements were fulfilled but only based on the created scenarios. The validation of the performance and reliability non-functional requirements was more of a proof of concept.

To perform a more meaningful test, there would need to be a lot more test scenarios to cover more areas of the performance. Moreover, the non-functional requirement should have been explicitly stated by the product owner, if this was important for the system, however we figured that these non-functional requirements are always a part of a software system.

The testing with JMeter was done while running the server on the same machine. Since JMeter and the server was running locally, the requests stayed within the same machine, which meant that the requests did not have to travel across a network which masks network latency. It also meant that JMeter and the server could possibly be bottlenecked by the hardware of the computer. This is a problem since it will not bypass actual network infrastructure and can result in a deviation in the actual latency compared to a real world scenario and result in wrong results. A better approach would mean setting up a testing environment with dedicated hardware or

using virtual machines to run the server, which will allow simulating real network conditions and this way it would also be possible to experiment with the allocation of resources to the server, and figure out how much resources is needed to handle a specific amount of load.

One of our non-functional requirements was modifiability. The current system does not dynamically bind modules, which means there is still going to exist build time coupling, since we have to instantiate our interfaces directly in the code.

For example, looking at our design class diagram 8, when our `SubdivideController.cs` uses the `IMarketSimRepistoryContext` interface, it still has to instantiate it with the direct implementation of `MarketSimRepistoryContext`.

The build time coupling increases the dependency depth, which decreases the modifiability of the system. We want to make our modules as independent as possible. This could be fixed using a component model such as dependency injection or a whiteboard query-based model to instantiate our interfaces during runtime. This means our modules would no longer know of each other's direct implementations. This overhaul of implementing a framework, that uses one of these models, is however outside the scope of this project and would require refactoring of the entire system.

## 7.5 Process evaluation

The main development part of the project consisted of six sprints with a duration of two weeks each. We did not implement all of the features in our scope during this time.

The two first sprints of the project, we spent the time setting up the environment for the PMS and gathering the requirements based on the objectives. We also analyzed the existing system and set up a whole new project, as our goal was to create a new product. However, we quickly realized that it was easier to implement the features directly in the existing PMS. We had some discussions about whether the owner of the PMS wanted us to create a whole new software system or continue the work on the existing system. At a meeting with the owner, we clarified that it would make more sense to build upon the existing feature which resulted in scrapping the other product, which could have been the reason for the need to deprioritize certain features from our scope.

Scrum is an agile process, which worked well with this project. The requirements were not set in stone during the initial requirements elicitation between the project team and product owner, because there was still some confusion as mentioned. Therefore the iterative nature of

Scrum meant it was easy to dynamically change our requirements as more clarity was gained with meetings between the team and product owner.

The process of the project went overall well, as the Scrum artifacts and meetings gave the team a good overview of the progress of the project, and whether or not things needed to change. The results were satisfactory as we managed to implement the MVP which consisted of the “must have” prioritized requirements.

## 7.6 Future work

The first improvement to the system would be to implement requirement R5.2.1, to be able to set the transmission capacity and improve the implementation of R5.2.2 to be able to calculate for multiple areas at a time.

The other improvements would be to implement proper validation to the new features. Such as not being able to go directly to subdivision editing version without selecting a country from the PMS first and not being able to press the subdivide button without having drawn a proper polygon to intersect with the selected area.

Then improvements would have to be made to the UI to make it better looking and more intuitive for users.

The found problems and missing parts discussed in section 7.4.1 provide another list of things that can be improved on.

Another thing to consider would be to do some refactoring on the implemented code in order to reduce the amount of cyclomatic complexity, to make the code more readable and easier to maintain.

## 8 Conclusion

The problem that the project wants to solve is the fact that some countries are not electrically connected, but are still treated as such when doing calculations within the PMS. The goal of this project was to further develop the PMS to have the functionality of subdividing areas to solve this problem.

The methodology used during the project lifespan was Scrum. This software process worked for this project, since not all of the requirements were well defined in the beginning of the project. Scrum is agile, which meant it was easy to dynamically change the requirements. The team also had previous experience with Scrum, which meant less time was spent on learning the process.

This project is a brownfield software development project, which meant most of the efforts during the analysis and design were spent on understanding the constraints and limits of the current system. The main findings were that the system is a combination of service-oriented architecture for the high-level architecture while a model-view-controller pattern defined low level components through ORM. Staying within the constraints meant encapsulating all the business logic regarding subdivision within its own module and making sure the module only communicates with the layers below it. The flow of data was maintained such that the layered architecture was still uni-directional.

The requirements were fulfilled besides requirement R5.2.1 and R5.2.2. These requirements were deprioritized because of time constraints and pushed to the next iteration of the project. The fulfilled requirements did have some issues that could be improved in the next iteration, such as edge cases of dividing areas and delegating power plants with more accuracy according to GIS-coordinates.

The requirements fulfillment were determined on the verification, using testing such as manual acceptance testing, metrics analysis and unit testing. The success criteria can be seen as the objectives laid out in the beginning of the rapport. Not all criteria have been met, such as objective 8 and objective 9. The project can therefore be seen as a medium success and provides a good basis for future work.

## 9 Literature list

[1] Apache JMeter. “Apache JMeter”

<https://jmeter.apache.org/index.html> [Last visited 03/05-2023]

[2] Atlassian. Bitbucket “Gitflow Workflow”

<https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow> [Last visited 08/02-2023]

[3] Chidamber, Shyam R., and Chris F. Kemerer. ”A metrics suite for object oriented design.” IEEE Transactions on software engineering 20.6 (1994): 476-493.

[4] Deacon, John. ”Model-view-controller (mvc) architecture.” Online[Citado em: 10 de março de 2006.] <http://www.jdl.co.uk/briefings/MVC.pdf> 28 (2009).

[5] Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996.

[6] EntityFrameworkTutorial. “What is Entity Framework?”

<https://www.entityframeworktutorial.net/what-is-entityframework.aspx> [Last visited 10/02-2023]

[7] Erl, Thomas. Service-oriented architecture. Upper Saddle River: Pearson Education Incorporated, 1990.

[8] JavaTPoint. “Git Tutorial”

<https://www.javatpoint.com/git> [Last visited 10/02-2023]

[9] GeekForGeeks. “Difference .NET and ASP.NET Framework”

<https://www.geeksforgeeks.org/difference-between-net-and-asp-net-framework/> [Last visited 10/02-2023]

[10] GitHub Docs. “About Projects”

<https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects>

[Last visited 10/02-2023]

[11] Schwaber, Ken. ”Scrum development process.” Business Object Design and Implementation: OOPSLA’95 Workshop Proceedings 16 October 1995, Austin, Texas. Springer London, 1997.

[12] Shatnawi, Raed. ”A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems.” IEEE Transactions on software engineering 36.2 (2010): 216-225.

[13] Umbraco. “What is Umbraco?”

<https://umbraco.com/knowledge-base/umbraco/> [Last visited 08/02-2023]

[14] Watson, Arthur Henry, Dolores R. Wallace, and Thomas J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Vol. 500. No. 235. US

# 10 Appendix

## 10.1 Detailed use cases

Table 14: Use Case Save subdivision

| Use case             | Save subdivision   |
|----------------------|--|
| Brief description    | The user must be able to export data   |
| Actors               | User   |
| Secondary actors     |  |
| Preconditions        | The user must either selected an area to subdivide or subdivided an area and been redirected to the next view  |
| Flow of events       | <ol style="list-style-type: none"> <li>1. The use case starts when the user has subdivided an area (reference use case: “Create subdivision”)</li> <li>2. The user presses the export button.</li> <li>3. The information of the subdivided area, and their corresponding power plants are saved in a file.</li> <li>4. The file with the data from step 3 is downloaded on the user's machine.</li> </ol> |
| Postconditions       | A new configuration file is downloaded on the user’s machine.  |
| Alternative scenario | <ol style="list-style-type: none"> <li>1a. The user has not subdivided an area</li> <li>2a. The user presses the export button</li> <li>3a. No information is saved.</li> <li>4a. An empty file is saved on the user's machine.</li> </ol>   |

Table 15: Use Case Load subdivision

| Use case                    | Load subdivision   |
|-----------------------------|--|
| <b>Brief description</b>    | The user must be able to import subdivision data   |
| <b>Actors</b>               | User   |
| <b>Secondary actors</b>     |  |
| <b>Preconditions</b>        | The user has a file with the GeoJSON data containing areas and the corresponding power plants that belong to each area.  |
| <b>Flow of events</b>       | <ol style="list-style-type: none"> <li>1. The use case starts when the user clicks the import button on the “new” PMS.</li> <li>2. A file explorer opens up.</li> <li>3. The user picks the file with the relevant data.</li> <li>4. A new map is loaded with power plant data for each area.</li> </ol> |
| <b>Postconditions</b>       | The PMS loads in the subdivided area and their corresponding power plants. The user can now do area price calculations on each area, set the transmission capacity and see the chart results for each area.  |
| <b>Alternative scenario</b> | <p>3a. The user picks a file that doesn't fit the format</p> <p>4a. An error message pops up that warns the user to pick another format</p>  |



Table 16: Use Case Delete polygon

| Use case                    | Delete polygon  |
|-----------------------------|---|
| <b>Brief description</b>    | When a polygon has been drawn, the user can delete it.  |
| <b>Actors</b>               | User  |
| <b>Secondary actors</b>     |   |
| <b>Preconditions</b>        | The user has clicked the button that leads them to the subdivision site.  |
| <b>Flow of events</b>       | <ol style="list-style-type: none"> <li>1. The use case starts when the user has drawn a polygon (reference use case: “Draw polygon”)</li> <li>2. The user clicks the delete button.</li> <li>3. The user is now in a delete state.</li> <li>4. The user clicks on a polygon.</li> <li>5. The polygon is deleted.</li> <li>6. The user clicks the delete button again.</li> <li>7. The user exits the delete state.</li> </ol> |
| <b>Postconditions</b>       | A polygon has been deleted from the map.  |
| <b>Alternative scenario</b> | 6a. The user clicks on another polygon<br>7a. The polygon is deleted.<br>8a. Loop step 6a and 7a  |

Table 17: Use Case Set transmission capacity

| Use case                    | Set transmission capacity   |
|-----------------------------|---|
| <b>Brief description</b>    | The user can set the transmission capacity between two areas. One or both of the areas can be subdivided areas.   |
| <b>Actors</b>               | User  |
| <b>Secondary actors</b>     |   |
| <b>Preconditions</b>        | Use case: “Load subdivisions” has been fulfilled.   |
| <b>Flow of events</b>       | <ol style="list-style-type: none"> <li>1. The use case starts when the user clicks on the transmission tab.</li> <li>2. The transmission tab opens up.</li> <li>3. The user inserts the transmission between the two areas.</li> <li>4. The user presses a save button to save the changes</li> </ol> |
| <b>Postconditions</b>       | A transmission capacity has been set between two areas. When calculating area price this transmission capacity is now taken into account.   |
| <b>Alternative scenario</b> |   |

Table 18: Use Case Calculate area prices

| Use case                    | Calculate area prices   |
|-----------------------------|---|
| <b>Brief description</b>    | The user can calculate the area prices of the areas on the map. One to multiple areas can be calculated upon.   |
| <b>Actors</b>               | User  |
| <b>Secondary actors</b>     |   |
| <b>Preconditions</b>        | Use case “Load subdivisions” has been fulfilled.  |
| <b>Flow of events</b>       | <ol style="list-style-type: none"> <li>1. The use case starts when the user selects an area (reference use case “select area”)</li> <li>2. The user clicks the calculate area price button</li> <li>3. The area price of the selected areas are calculated</li> <li>4. A number beside the area shows up, which represents the area price.</li> </ol> |
| <b>Postconditions</b>       | The area price of the selected areas has been calculated. The user can now see area charts of these calculated price areas.   |
| <b>Alternative scenario</b> | <ol style="list-style-type: none"> <li>1a. The use case starts without the user selecting an area.</li> <li>2a. The user clicks the calculate area price button <ol style="list-style-type: none"> <li>a. Nothing happens as no area has been selected to be calculated upon.</li> </ol> </li> </ol>  |

Table 19: Use Case Show chart results

| Use case                    | Show chart results   |
|-----------------------------|--|
| <b>Brief description</b>    | A chart can be shown that represents the area prices of selected areas.  |
| <b>Actors</b>               | User   |
| <b>Secondary actors</b>     |  |
| <b>Preconditions</b>        | Use case “Load subdivisions” has been fulfilled  |
| <b>Flow of events</b>       | <ol style="list-style-type: none"> <li>1. The use case starts when area price has been calculated for selected areas (reference use case: “calculate area prices”)</li> <li>2. The user clicks on the show chart button.</li> <li>3. A new window pops up that shows the chart of the calculated price areas.</li> </ol> |
| <b>Postconditions</b>       | A chart of the price areas are shown on the user's screen.   |
| <b>Alternative scenario</b> | <ol style="list-style-type: none"> <li>1a. The use case starts without calculating an area price</li> <li>2a. The user clicks the show chart button</li> <li>3a. No charts are shown since no area prices have been calculated</li> </ol>  |

Table 20: Use Case Select area

| Use case                    | Select area  |
|-----------------------------|--|
| <b>Brief description</b>    | The user can select specific areas on the map  |
| <b>Actors</b>               | User   |
| <b>Secondary actors</b>     |  |
| <b>Preconditions</b>        | A map has to be shown  |
| <b>Flow of events</b>       | <ol style="list-style-type: none"> <li>1. The use case starts when the user clicks on an area</li> <li>2. The area is selected and is now highlighted</li> </ol>   |
| <b>Postconditions</b>       | An area has been selected.   |
| <b>Alternative scenario</b> | <ol style="list-style-type: none"> <li>1a. The use case starts with an area that has already been selected</li> <li>2a. The user clicks on another unselected area.</li> <li>3a. The selected area switches from the first selected area to the second unselected area.</li> </ol> |

## 10.2 Non-functional requirements

### 10.2.1 Graphs for GetAreaPrice and Subdivide area

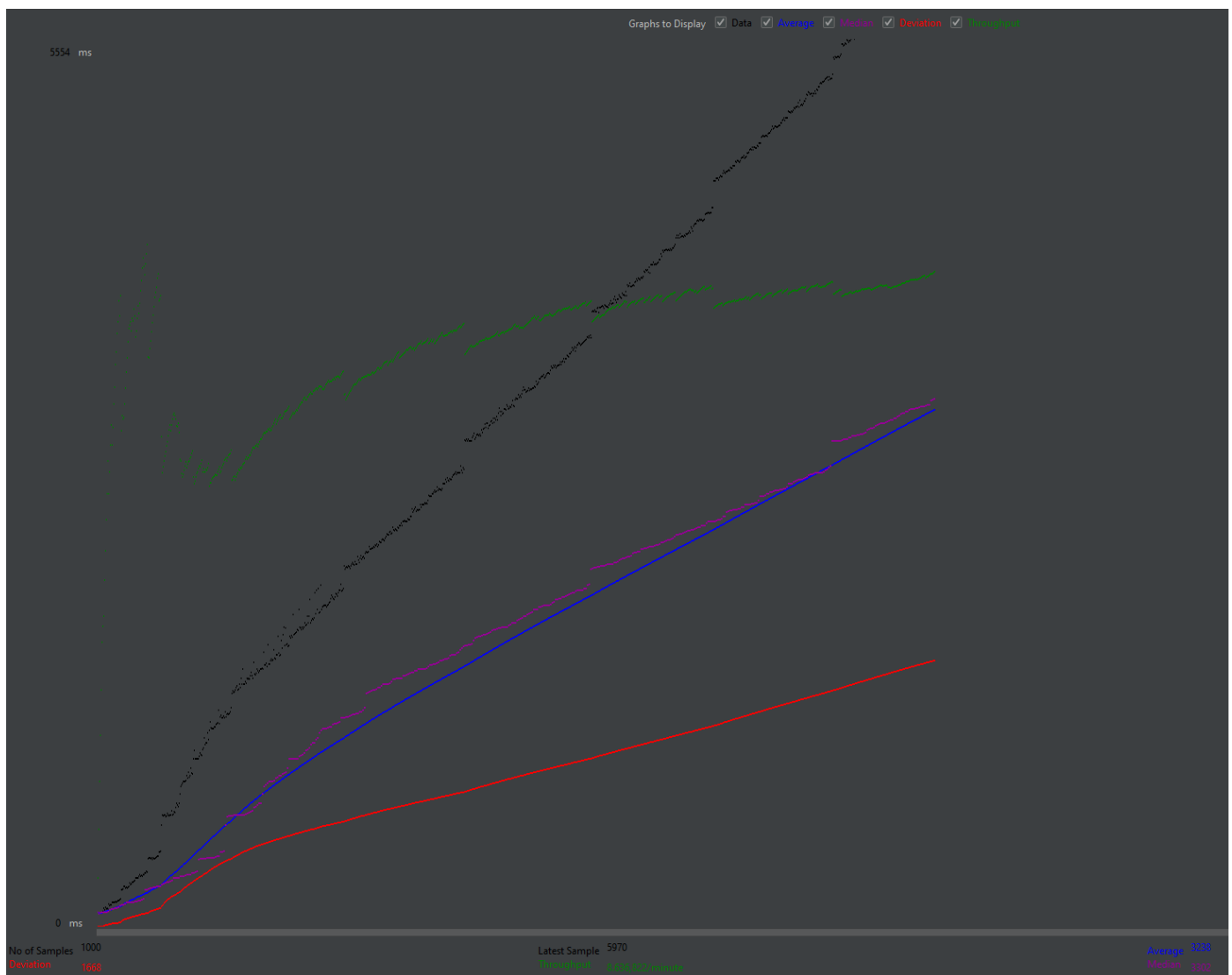


Figure 81: Graph for GetAreaPrice

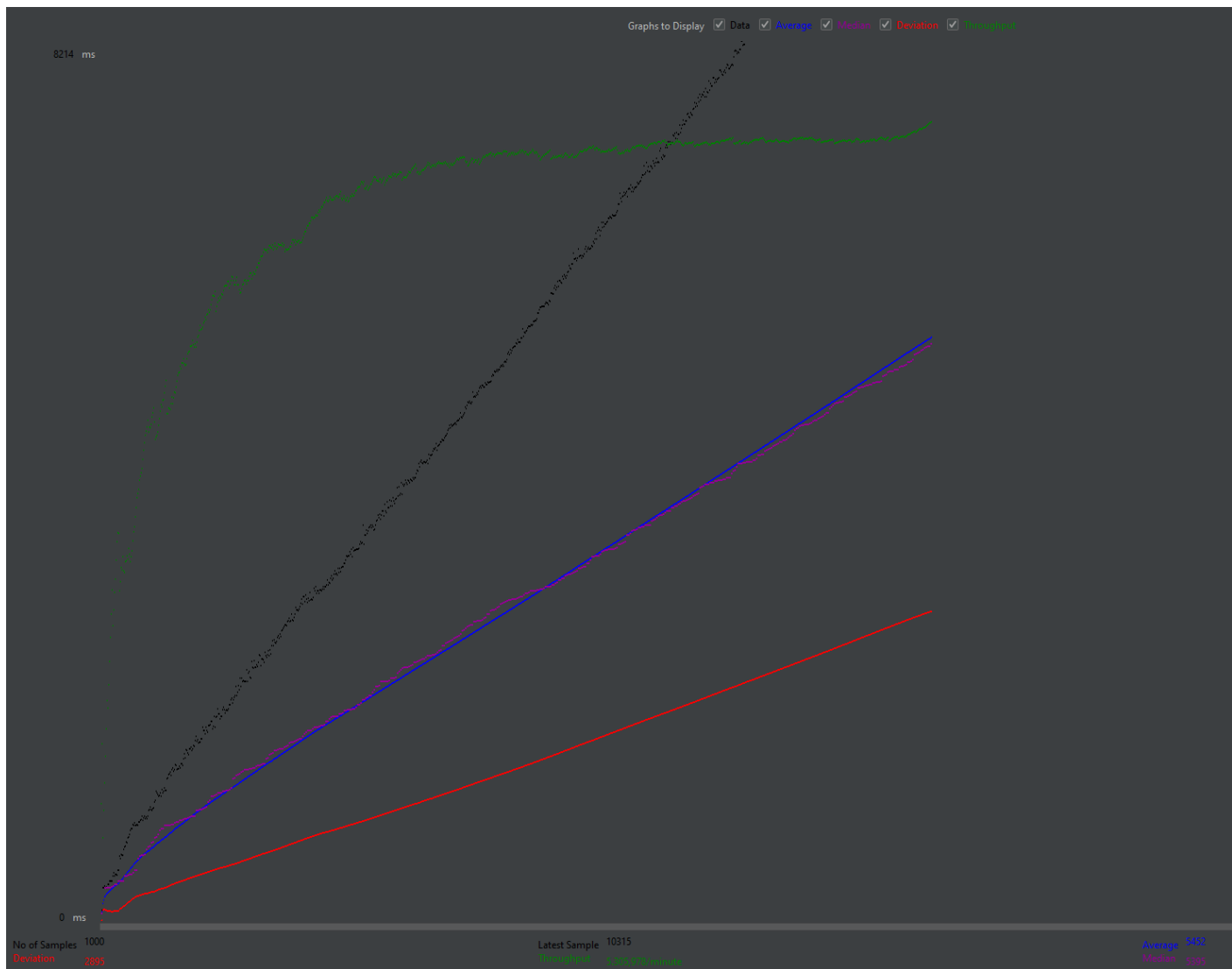


Figure 82: Graph for GetAreaPrice

## 10.2.2 Diagrams for the performance test scenarios

[Link to performance test diagrams](#)

## 10.2.3 Diagrams for reliability test scenario

[Link to reliability test diagrams](#)

## 10.3 Process evaluation figures

### 10.3.1 Product Backlog

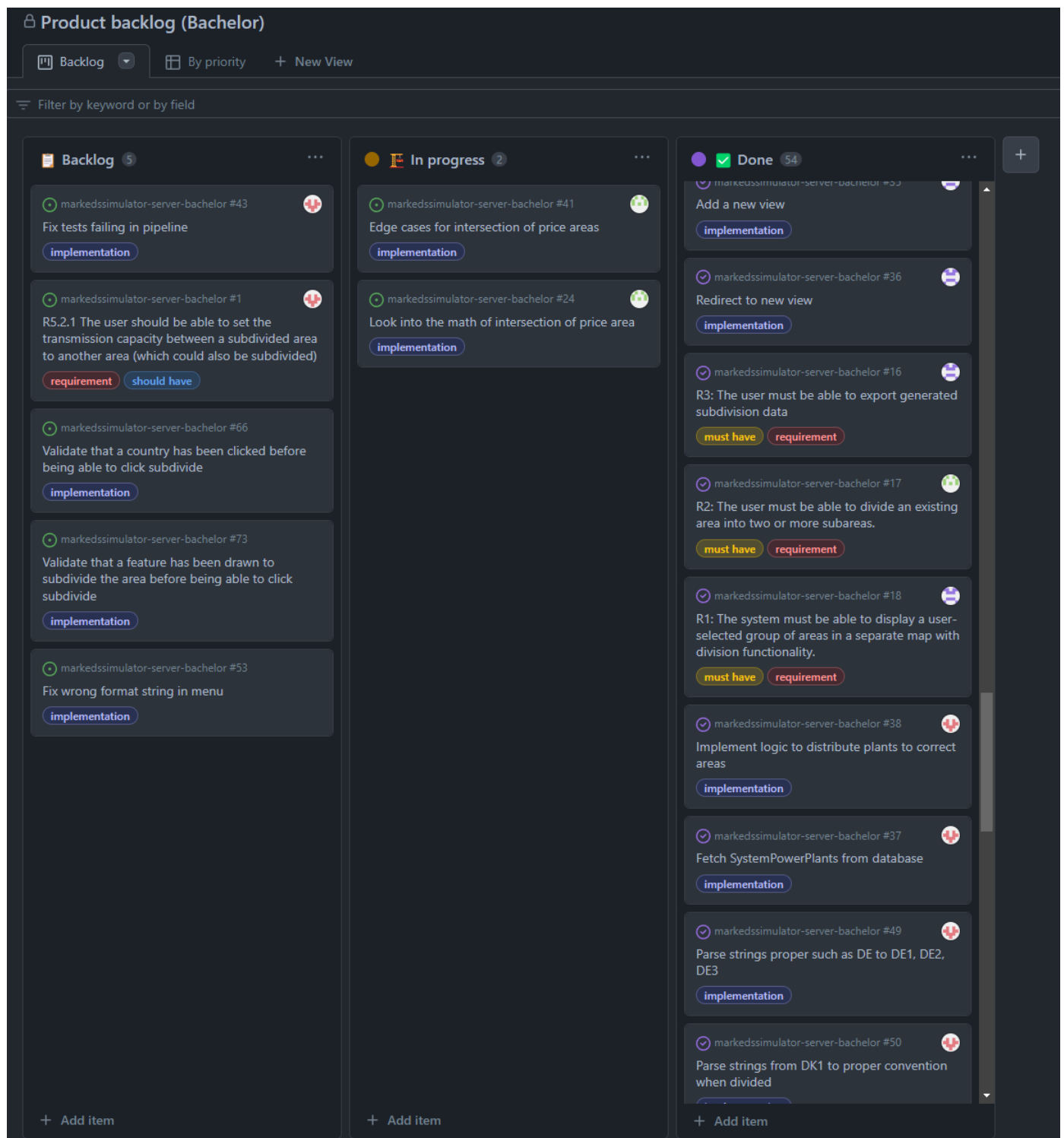


Figure 83: Product Backlog

### **10.3.2 Time Schedule**

[Link to Time Schedule](#)