Simple SCADA

Documentation

Document Reference:	FHOOE-SS2023-03-0001
Issue:	1
Revision:	0
Date of 1st issue:	11.02.2023
Date of this issue:	02.10.2023

FH OÖ Studienbetriebs GmbH

Document Approval

Title: Simple Scada
Issue: 1
Revision: 0
Date: 02.10.2023
Document reference: FHOOE-SS2023-03-0001

Prepared by	Organisation	Signature	Date
Roman Froschauer	AT		

Reviewed by	Organisation	Signature	Date
Kenan Dautovic	AT		02.10.2023

Approved by	Organisation	Signature	Date
Lauretta Feichtmair	AT		02.10.2023

Released by	Organisation	Signature	Date
Lukas Brajdic	AT		02.10.2023

Content

Document Approval **2**
Content **3**
1 General Information **6**
1.1 General 6
1.2 Applicable documents 6
1.3 Referenced documents 6
2 Requirements Engineering **7**
2.1 Identification of Requirements 7
2.1.1 Type of Requirements 7
2.1.2 Description of Requirement 8
2.1.3 Verification Method & Level 8
2.2 User Stories 9
2.2.1 General Information 9
2.2.2 Story 1: User Types 9
2.2.3 Story 2: Monitor Production 9
2.2.4 Story 3: Start Production Recipe 10
2.2.5 Story 4: View Production History 10
2.2.6 Story 5: Error-Handling 11
2.2.7 Story 6: Recipe Queue 11
2.3 Technical Requirements 12
2.3.1 System Requirements Table 12
2.3.2 Subsystem 1 Controller / Gateway Requirements Table 20
2.3.3 Subsystem 2 Database Requirements Table 21
2.3.4 Subsystem 3 PC App Requirements Table 23
2.3.5 May Subsystem 4 Mobile App Requirements Table 28
3 Detailed Design **31**
3.1 Introduction 31
3.2 SYSTEM ARCHITECTURE 32
3.3 SUBSYSTEM 1 - Controller / Gateway 33
3.3.1 SUBSYSTEM1 ARCHITECTURE 33
3.3.2 SUBSYSTEM1 INTERFACE 33
3.3.3 SUBSYSTEM1 Paths 33
3.3.3.1 General information 33
3.3.3.2 OPCUA Library 35
3.3.3.3 Code implementation 35
3.3.3.4 Problems 36
3.3.4 SUBSYSTEM1 Transitions 36
3.3.5 SUBSYSTEM1 Conveyor belt control 38
3.3.5.1 General information 38
3.3.5.2 Code implementation 39
3.3.6 SUBSYSTEM1 Machine listener 43

3.3.6.1 General information 43
3.3.6.2 Code implementation 44
3.4 SUBSYSTEM 2 – SQL Database 46
3.4.1 SUBSYSTEM2 ARCHITECTURE 46
3.4.1.1 The Database Model Classes and Enums 46
3.4.1.2 The Database 47
3.4.1.3 The Database Communication Managers 48
3.4.2 SUBSYSTEM2 INTERFACES 48
3.4.3 SUBSYSTEM2 MODULE 1: Recipe Manager 49
3.4.3.1 Model Utility 52
3.4.3.2 Model Station 53
3.4.3.3 Model Recipe 54
3.4.3.4 Model RecipeStation 55
3.4.4 SUBSYSTEM2 MODULE 2: ProductionCycle Manager 56
3.4.4.1 Model ProductionCycle 59
3.4.5 SUBSYSTEM2 MODULE 3: Plant Manager 60
3.4.5.1 Model Plant 63
3.4.5.2 Model SensorData 63
3.4.6 SUBSYSTEM2 MODULE 4: User Manager 64
3.4.6.1 Model User 68
3.4.7 SUBSYSTEM2 MODULE 5: Log Manager 69
3.4.7.1 Model Log 72
3.4.8 SUBSYSTEM2 MODULE 6: Error Manager 73
3.4.8.1 Model Error 75
3.4.9 SUBSYSTEM2 MODULE 7: OEE Manager 76
3.4.9.1 Model OEE 76
3.4.10 SUBSYSTEM2 MODULE 8: DbCreationContext 81
3.4.11 SUBSYSTEM2 MODULE 9: Linux Database 83
3.4.12 SUBSYSTEM2 MODULE 10: Azure Database 83
3.5 SUBSYSTEM 3– Desktop Application 85
3.5.1 SUBSYSTEM 3 ARCHITECTURE 85
3.5.2 SUBSYSTEM 3 INTERFACES 86
3.5.3 SUBSYSTEM 3 MODULE 1: Plant Overview 86
3.5.3.1 Plant Overview View 86
3.5.3.2 Plant Overview Presenter 89
3.5.3.3 Plant Overview Model 92
3.5.4 SUBSYSTEM 3 MODULE 2: Error and Log Overview 92
3.5.4.1 Log View and Production-History View 92
3.5.4.2 Error View 93
3.5.5 SUBSYSTE 3 MODULE 3: Production Management 94
3.5.5.1 Recipe Management 94
3.5.5.1.1 Recipe View 94
3.5.5.1.2 Recipe Presenter 97

3.5.5.1.3 Recipe Model 99
3.5.5.2 Production Order Management 99
3.5.5.2.1 Job Processing View 99
3.5.5.2.2 Job Processing Manager Presenter 101
3.5.5.2.3 Job Processing Model 102
3.5.6 SUBSYSTEM 3 MODULE 4: User management 102
3.5.6.1 User Management 102
3.5.6.1.1 User View 102
3.5.6.1.2 User Manager Presenter 104
3.5.6.1.3 User Manager Model 105
3.5.6.2 User Log In 105
3.5.6.2.1 Log in View 106
3.5.6.2.2 Log In Presenter 107
3.5.6.2.3 Log In Model 108
3.5.6.3 Password Management 108
3.5.6.3.1 Password Change View 108
3.5.6.3.2 Password Change Presenter 110
3.5.6.3.3 Password Change Model 111
3.5.7 SUBSYSTEM 3 MODULE 5: Productivity 111
3.5.7.1 OEE View 111
3.5.8 SUBSYSTEM 3 MODULE 6: User Entertainment 111
3.5.8.1 Joke View 111
3.5.8.2 Joke Presenter 113
3.5.8.3 Joke Model 114
3.6 SUBSYSTEM 4 – Android App 115
4 Factory Acceptance Test 116
4.1 General Information 116
4.2 Possible Tests 116
A Appendix: List of abbreviations 118

1 General Information

1.1 General

The aim of this project is to design and program a Supervisory Control and Data Acquisition (SCADA) software for a simulated production line. Therefore, a database, a controller and a user interface should be created.

1.2 Applicable documents

Reference	Title
AD0	System architecture https://www.figma.com/file/rrMxINddZ9KSlrU8H3VICM/Softwarearchitektur?type=whiteboard&node-id=0%3A1&t=OA49rN8BJD3kAUa4-1
AD1	Design desktop app and mobile app https://www.figma.com/file/rrMxINddZ9KSlrU8H3VICM/Softwarearchitektur?type=whiteboard&node-id=0%3A1&t=OA49rN8BJD3kAUa4-1

1.3 Referenced documents

Reference	Title
RD1	http://www.agilemodeling.com/artifacts/userStory.htm
None	None

2 Requirements Engineering

2.1 Identification of Requirements

The requirements are characterized by their type, verification method, verification level, responsible entity, and their current status. These attributes are defined as follows.

2.1.1 Type of Requirements

Type	
Functional	Functional requirements define what the system shall perform in order to conform to the needs of the customer. E.g.: "The system shall store data from connected PLCs."
Non-Functional	Non-Functional requirement define how the system accomplishes the functional requirements in the meaning of Quality of Service. E.g.: "The system shall enable secure data storage according to Standard xy".
Interface	Requirements related to the interconnection or relationship characteristics of items/modules (either hardware or software). This includes different types of interfaces (electrical, physical, software, protocol, etc…). E.g.: "The servo drives are controlled by the real-time protocols CAN and Profibus."
Environmental	Requirements related to the system life-time environment (temperature, humidity, vibration, contamination, etc…). E.g.: "The system shall operate within the temperature range of -10°C to 50°C."
Operational	Requirements related to events or conditions to which the system shall respond. E.g.: "A LED shall indicate whether the system is on or off."
Logistics	Requirements related to the installation, operation, maintenance, transportation and storage of the system. E.g.: "The system shall be designed to be installed at the customer's site within 1 day."
Physical	Requirements related to mechanical and electrical characteristics, the dimensions and the appearance of the system. E.g.: "The main PCB board shall fit into a box with LxBXH: 10x4x4cm"
Design	Requirements related to the imposed design and construction standards such as choice of material, components or methods. E.g.: "The user interface shall be designed according to Microsoft UX Guide"
Verification	Requirements related to the process of verification and the applied standards or methods. E.g.: "The

	measurements shall be conducted applying 128 times averaging of the samples.”
Human Factor	Requirements related to the human-machine interface of the system. E.g.: “It should be able to operate the system by one skilled person.”
Software	Requirements related to software aspects like the definition of input data, the way the code is implemented or the functional objectives of a driver module. E.g.: “The software shall run on Windows 10 operating system.”
Quality & Software	Requirements related to ensure full lifetime operability. This can include the reliability and availability of a component or aspects of maintainability. Also safety issues are addressed. E.g.: “The support of the procured components shall be guaranteed by the manufacturer for at least 10 years.”

2.1.2 Description of Requirement

Technical requirements should be stated in performance or “what-is-necessary terms”, as opposed to telling “how-to”. The description should be expressed in a positive way. Each requirement should be unambiguously described. Each requirement should be unique and not in conflict with other specifications. The requirement should be verifiable by the defined verification method. A range of parameters or variables within which the conformity is accepted is the preferred of description.

The terms “shall”, “should”, “may”, and “can” are used whenever a provision is required, recommended, permitted or indicated as possible, respectively.

2.1.3 Verification Method & Level

The verification is executed by one or more of the following methods:

→ Test (measurement, experiment)
→ Demo (functional demonstration)
→ Analysis (calculation, simulation)
→ Review of Design (verified against drawings, schematics, software code)
→ Inspection (review of another person).

The list shows the order of precedence that provides more confidence in the results.

The verification method is performed at different system decomposition levels. For the accomplished project, these are:

→ System level
→ Sub-system level
→ Equipment level and
→ Component level.

2.2 User Stories

2.2.1 General Information

A user story is an informal, natural language description of one or more features of a software system. User stories are often written from the perspective of an end user or user of a system.

Often the stories are expressed as shown below:

```
As a <role> I can <capability>, so that <receive benefit>
```

A role can be User, Engineer, Customer, etc.

The priority expresses the degree of necessity of the customer to accomplish the user story.

The risk expresses the difficulty of implementing it. A story point typically indicates 2,5hours work load for a team of two developers.

For more information see [RD1].

2.2.2 Story 1: User Types

Priority: HIGH		Story Points Estimate: 2
Description: As a user, I can log in as either a plant spectator or plant operator to have access to either plant monitoring or recipe creation, or I can log in as superuser to create the other user types.		
Risk: MEDIUM		

2.2.3 Story 2: Monitor Production

Priority: HIGH		Story Points Estimate: 9
Description: As both plant spectator and operator, I can see a graphic visualization of the plant, including the status of the sensors and the position of the pallets, in order to be able to monitor the production process.		
Risk: MEDIUM		

2.2.4 Story 3: Start Production Recipe

Priority: HIGH		Story Points Estimate: 2
Description: As a plant operator, I can feed in a recipe (=sequence of the completed stations) to start the production process.		
Risk: LOW		

Priority: HIGH		Story Points Estimate: 2
Description: As a user, I can log in as either a plant spectator or plant operator to have access to either plant monitoring or recipe creation		
Risk: MEDIUM		

2.2.5 Story 4: View Production History

Priority: HIGH		Story Points Estimate: 2
Description: As a plant spectator and plant operator, I can view the histories of various products (arrival and departure times of the product at the respective stations) so that I can analyse past production processes.		
Risk: LOW		

2.2.6 Story 5: Error-Handling

Priority: HIGH		Story Points Estimate: 2
Description: As a plant spectator, I would like the control system to stop automatically in the event of a fault and an error message to appear on the display so that I can react to it.		
Risk: LOW		

2.2.7 Story 6: Recipe Queue

Priority: LOW		Story Points Estimate: 2
Description: As a Creator I want that various recipes are processed at the same time, to maximise the output of the plant. As plant operator, I can put several recipes in a queue, so that the plant can execute one recipe after the other without further need of action.		
Risk: LOW		

2.3 Technical Requirements

2.3.1 System Requirements Table

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Functional	1		The system shall be able to detect connection, internal and critical errors.	Test	System	Controller/ Gateway	KD u. ST	closed
Functional	2		The system shall be able to automatically stop the plant in case of an error that prevents the operation of the plant.	Test	System	Controller/ Gateway	KD u. ST	closed
Functional	3		The system may be able to process various recipes simultaneously.	Test	System	Controller/ Gateway	KD u. ST	open
Operational	4		The Controller shall communicate with the SPS (Simulation) via OPC UA	Test	System	Controller/ Gateway	KD u. ST	closed
Functional	5		The system may be able to process a whole recipe queue (up to 10 recipes)	Test	System	Controller/ Gateway	KD u. ST	closed
Functional	6		A recipe shall be ordered by stations and duration at each station.	Database Design Review	System	Data-base	JM u. BL	closed
Functional	7		The system shall store the production data in a database.	Database Design Review	System	Data-base	JM u. BL	closed

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Software	8		The production data contains at least time stamp of each pallet, station, recipe, executor, identification number of product.	Database Design Review	System	Data-base	JM u. BL	partly
Physical	9		The database and controller may run on a raspberry pi with Linux.	Review of Hardware	Hardware	Data-base	JM u. BL	open
Software	10		The database shall be implemented with EF-Core.	Review of Code	System	Data-base	JM u. BL	open
Quality & Software	11		The production history SHALL NOT PASS over one year	Test	System	Data-base	JM u. BL	open
Non-Functional	12		The database shall store the users (spectator and operator) with a hashed password.	Test	System	Data-base	JM u. BL	closed
Functional	13		The database shall store the login credentials (password as hash) of 1 superuser that can create the other 2 user types (spectator and operator).	Database Design Review	System	Data-base	JM u. BL	closed
Functional	14		The database shall store the information of each station containing it`s different utilities and duration.	Database Design Review	System	Data-base	JM u. BL	closed

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Functional	15		A station shall have different utilities (eg assamble, weld) with different times to choose from when the operator is creating a recipe.	Test	System	PC App	FL	closed
Functional	16		The system shall support at least the following types of users: spectator, operator, admin	Test	System	PC App +Mobile App	FL	closed
Functional	17		The system shall support an authentication on the basis of user name and password	Test	System	PC App +Mobile App	FL	closed
Functional	18		The system shall support complexity rules for passwords.	Test	System	PC App	FL	closed
Functional	19		The spectator role shall support the functions as mentioned in the following requirements: 25,31	Test	Test	PC App +Mobile App	FL, NH and. KS	closed
Functional	20		The operator role shall support the functions as mentioned in the following requirements:25,30,31,42	Test	Test	PC App +Mobile App	FL , NH and KS	closed
Functional	21		The system shall be able to priorities errors.	Test	System	PC App +Mobile App	KS	closed
Functional	22		The system may indicate the Overall Efficiency of the Machine.	Test	Test	PC App +Mobile App	NH	closed

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Human Factor	27		The system shall graphically indicate the status of the sensors via green or grey lights.	Review of Design		PC App	FL and KS	closed
Human Factor	28		The system shall indicate the following states, running, pausing, maintenance, disturbance.	Review of Design	System	PC App +Mobile App	KS	closed
Functional	29		The system shall be able that the operator can create, edit, delete recipes.	Review of Design	System	PC App	FL	closed
Operational	30		Only the operator shall be able to start and a production cycle.	Test		PC App	FL	closed
Operational	31		The system shall be able to be stopped by a spectator or operator.	Test		PC App +Mobile App	FL	closed
Human Factor	32		The system shall show by default the UI where only the log in function is accessible.	Test		PC App	FL	closed
Functional	33		The system shall adapt the UI depending on the logged-on user.	Review of Design		PC App	FL	closed
Human Factor	34		The system shall support an indicator that displays a warning symbol.	Review of Design		PC App	KS and NH	Open

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Human Factor	35		The system shall support a window that displays the currently occurring error(s).	Review of Design		PC App	KS	closed
Operational	36		The error window shall support an acknowledge button that deletes the error from the window.	Test		PC App	KS	open
Functional	37		The error window supports a priority ranking of occurring errors.	Test		PC App +Mobile App	KS	Open
Functional	38		The system shall open a pop-up window in case of a high priority error.	Test		PC App +Mobile App	KS	Open
Human Factor	39		The system shall show the progress of the current recipe.	Review of Design		PC App	FL	closed
Human Factor	40		The production data history shall be shown in a separate UI in a separate window (view).	Review of Design		PC App	NH	Open
Environ-mental	41		The UI and its elements are not scalable.	Review of Design		PC App +Mobile	FL, NH and KS	Open

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Functional	42		The operator shall have his own view (window) when creating new recipes	Review of Design		PC App	FL	closed
Functional	43		The system shall have a graphical user interface enabling the modification of the users password.	Test		PC App	FL	closed
Human Factor	44		The system shall have a graphical user interface presenting a schematic overview of the plant	Review of Design		PC App	FL and KS	closed
Human Factor	45		The schematic overview shall show a 2D bird-view perspective of the plant layout	Review of Design		PC App	KS	closed
Human Factor	46		The schematic overview shall indicate the current position of material carriers using colored squares.	Review of Design		PC App	KS	closed
Human Factor	47		The user interface shall show the current active production order	Review of Design		PC App +Mobile	FL	closed
Human Factor	48		The user interface shall show the current user	Review of Design		PC App +Mobile	FL	closed
Environ-mental	49		The UI software shall be implemented via Windows Forms in C#	Review of Design		PC App	FL u. NH u. KS	closed

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Environ-mental	50		The software shall be executable on PCs running Windows 10 or higher	Review of Design		PC App	FL u. NH u. KS	closed
Functional	51		The interface shall be able to show the errors of a time period.	Test		PC App	NH	open
Operational	24		A recipe shall be ordered by stations the palette runs through and duration at each station.	Test	System	PC App	FL	open
Operational	25		The operator and spectator shall be able to continue a production cycle.	Test	System	PC App	FL	open
Functional	26		The admin role shall only support the administration of users and their user rights.	Test	System	PC App	FL	closed
Enviromental	52		The system may have an App.			Mobile App	Project Group	open
Functional	53		The app may have an error overview of the current detectable errors.			Mobile App	Project Group	open
Functional	54		The app may display the current recipe and the queue of recipes.			Mobile App	Project Group	open
Environmental	55		The app may run on Android devices.			Mobile App	Project Group	open
Functional	56		The app may support the operator and spectator user.			Mobile App	Project Group	open

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Functional	57		The app may show the current user.			Mobile App	Project Group	open
Functional	58		The app may have an authentication on the basis of user name and password			Mobile App	Project Group	open
Operational	59		The app may support an acknowledge button that deletes the error from the window.			Mobile App	Project Group	open
Functional	60		The app may show the error history for a time period.			Mobile App	Project Group	open
Functional	61		The app may show the production history for a time period.			Mobile App	Project Group	open
Functional	62		The app may show the current OEE.(Overall Equipment Efficiency)			Mobile App	Project Group	open

2.3.2 Subsystem 1 Controller / Gateway Requirements Table

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Functional	1		The system shall be able to detect connection, internal and critical errors.	Test	System	Controller/ Gateway	KD u. ST	Closed
Functional	2		The system shall be able to automatically stop the plant in case of an error that prevents the operation of the plant.	Test	System	Controller/ Gateway	KD u. ST	Closed
Functional	3		The system may be able to process various recipes simultaneously.	Test	System	Controller/ Gateway	KD u. ST	Open
Operational	4		The Controller shall communicate with the SPS (Simulation) via OPC UA	Test	System	Controller/ Gateway	KD u. ST	Closed
Functional	5		The system may be able to process a whole recipe queue (up to 10 recipes)	Test	System	Controller/ Gateway	KD u. ST	Closed

2.3.3 Subsystem 2 Database Requirements Table

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Functional	6		A recipe shall be ordered by stations and duration at each station.	Database Design Review	System	Data-base	JM u. BL	closed
Functional	7		The system shall store the production data in a database.	Database Design Review	System	Data-base	JM u. BL	closed
Software	8		The production data contains at least time stamp of each pallet, station, recipe, executor, identification number of product.	Database Design Review	System	Data-base	JM u. BL	partly
Physical	9		The database and controller may run on a raspberry pi with Linux.	Review of Hardware	Hardware	Data-base	JM u. BL	open
Software	10		The database shall be implemented with EF-Core.	Review of Code	System	Data-base	JM u. BL	open
Quality & Software	11		The production history SHALL NOT PASS over one year	Test	System	Data-base	JM u. BL	open
Non-Functional	12		The database shall store the users (spectator and operator) with a hashed password.	Test	System	Data-base	JM u. BL	closed

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Functional	13		The database shall store the login credentials (password as hash) of 1 superuser that can create the other 2 user types (spectator and operator).	Database Design Review	System	Data-base	JM u. BL	closed
Functional	14		The database shall store the information of each station containing it`s different utilities and duration.	Database Design Review	System	Data-base	JM u. BL	closed
Interface	23		The system may have an API (Application Interface) that can be used for the APP and the desktop app.	Review of Code	System	Data-base	JM u. BL	open

2.3.4 Subsystem 3 PC App Requirements Table

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Functional	15		A station shall have different utilities (eg assamble, weld) with different times to choose from when the operator is creating a recipe.	Test	System	PC App	FL	closed
Functional	16		The system shall support at least the following types of users: spectator, operator, admin	Test	System	PC App	FL	closed
Functional	17		The system shall support an authentication on the basis of user name and password	Test	System	PC App	FL	closed
Functional	18		The system shall support complexity rules for passwords.	Test	System	PC App	FL	closed
Functional	19		The spectator role shall support the functions as mentioned in the following requirements: 25,31	Test	Test	PC App	FL, NH and. KS	closed
Functional	20		The operator role shall support the functions as mentioned in the following requirements:25,30,31,42	Test	Test	PC App	FL , NH and KS	closed
Functional	21		The system shall be able to priorities errors.	Test	System	PC App	KS	closed
Functional	22		The system may indicate the Overall Efficiency of the Machine.	Test	Test	PC App	NH	closed

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Human Factor	27		The system shall graphically indicate the status of the sensors via green or grey lights.	Review of Design		PC App	FL and KS	closed
Human Factor	28		The system shall indicate the following states, running, pausing, maintenance, disturbance.	Review of Design	System	PC App	KS	closed
Functional	29		The system shall be able that the operator can create, edit, delete recipes.	Review of Design	System	PC App	FL	closed
Operational	30		Only the operator shall be able to start and a production cycle.	Test		PC App	FL	closed
Operational	31		The system shall be able to be stopped by a spectator or operator.	Test		PC App	FL	closed
Human Factor	32		The system shall show by default the UI where only the log in function is accessible.	Test		PC App	FL	closed
Functional	33		The system shall adapt the UI depending on the logged-on user.	Review of Design		PC App	FL	closed
Human Factor	34		The system shall support an indicator that displays a warning symbol.	Review of Design		PC App	KS and NH	Open

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Human Factor	35		The system shall support a window that displays the currently occurring error(s).	Review of Design		PC App	KS	closed
Operational	36		The error window shall support an acknowledge button that deletes the error from the window.	Test		PC App	KS	open
Functional	37		The error window supports a priority ranking of occurring errors.	Test		PC App	KS	Open
Functional	38		The system shall open a pop-up window in case of a high priority error.	Test		PC App	KS	Open
Human Factor	39		The system shall show the progress of the current recipe.	Review of Design		PC App	FL	closed
Human Factor	40		The production data history shall be shown in a separate UI in a separate window (view).	Review of Design		PC App	NH	Open
Environ-mental	41		The UI and its elements are not scalable.	Review of Design		PC App	FL, NH and KS	Open
Functional	42		The operator shall have his own view (window) when creating new recipes	Review of Design		PC App	FL	closed
Functional	43		The system shall have a graphical user interface enabling the modification of the users password.	Test		PC App	FL	closed

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Human Factor	44		The system shall have a graphical user interface presenting a schematic overview of the plant	Review of Design		PC App	FL and KS	closed
Human Factor	45		The schematic overview shall show a 2D bird-view perspective of the plant layout	Review of Design		PC App	KS	closed
Human Factor	46		The schematic overview shall indicate the current position of material carriers using colored squares.	Review of Design		PC App	KS	closed
Human Factor	47		The user interface shall show the current active production order	Review of Design		PC App	FL	closed
Human Factor	48		The user interface shall show the current user	Review of Design		PC App	FL	closed
Environ-mental	49		The UI software shall be implemented via Windows Forms in C#	Review of Design		PC App	FL u. NH u. KS	closed
Environ-mental	50		The software shall be executable on PCs running Windows 10 or higher	Review of Design		PC App	FL u. NH u. KS	closed
Functional	51		The interface shall be able to show the errors of a time period.	Test		PC App	NH	open

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Operational	24		A recipe shall be ordered by stations the palette runs through and duration at each station.	Test	System	PC App	FL	open
Operational	25		The operator and spectator shall be able to continue a production cycle.	Test	System	PC App	FL	open
Functional	26		The admin role shall only support the administration of users and their user rights.	Test	System	PC App	FL	closed

2.3.5 May Subsystem 4 Mobile App Requirements Table

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Human Factor	28		The system shall indicate the following states, running, pausing, maintenance, disturbance.			Mobile App	Project Group	open
Operational	31		The system shall be able to be stopped by the operator and spectator.			Mobile App	Project Group	open
Functional	34		The system shall support an indicator that displays a warning symbol.	Test		Mobile App	Project Group	open
Operational	37		The error window supports a priority ranking of occurring errors.	Test		Mobile App	Project Group	open
Functional	38		The system shall open a pop-up window in case of a high priority error.	Test		Mobile App	Project Group	open
Environ-mental	41		The UI and its elements are not scalable.			Mobile App	Project Group	open
Human Factor	47		The user interface shall show the current active production order			Mobile App	Project Group	open
Human Factor	48		The user interface shall show the current user			Mobile App	Project Group	open

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
	52		The system may have an App.			Mobile App	Project Group	open
Functional	53		The app may have an error overview of the current detectable errors.			Mobile App	Project Group	open
Functional	54		The app may display the current recipe and the queue of recipes.			Mobile App	Project Group	open
Environmental	55		The app may run on Android devices.			Mobile App	Project Group	open
Functional	56		The app may support the operator and spectator user.			Mobile App	Project Group	open
Functional	57		The app may show the current user.			Mobile App	Project Group	open
Functional	58		The app may have an authentication on the basis of user name and password			Mobile App	Project Group	open
Operational	59		The app may support an acknowledge button that deletes the error from the window.			Mobile App	Project Group	open

Type	No.	Requirement	Description	Verific. Method	Verific. Level	Sub-System	Responsibility	Status
Functional	60		The app may show the error history for a time period.			Mobile App	Project Group	open
Functional	61		The app may show the production history for a time period.			Mobile App	Project Group	open
Functional	62		The app may show the current OEE.(Overall Equipment Efficiency)			Mobile App	Project Group	open
Functional	16		The system shall support at least the following types of users: spectator, operator, admin	Test	System	Mobile App	Project Group	open
Functional	17		The system shall support an authentication on the basis of user name and password	Test	System	Mobile App	Project Group	open
Functional	19		The spectator role shall support the functions as mentioned in the following requirements: 6,45,55			Mobile App	Project Group	open
Functional	20		The operator role shall support the functions as mentioned in the following requirements:3,5,6,55			Mobile App	Project Group	open
Functional	21		The system shall be able to priorities errors.	Test	System	Mobile App	Project Group	open
Functional	22		The system may indicate the Overall Efficiency of the Machine.			Mobile App	Project Group	open

3 Detailed Design

3.1 Introduction

The relevant top level requirements are considered as design goals and are summarized in the following tables.

Functional Requirements	
7	The system shall store the production data in a database.
2	The system shall be able to automatically stop the plant in case of an error that prevents the operation of the plant.
16	The system shall support at least the following types of users: spectator, operator, admin
33	The system shall adapt the UI depending on the logged-on user.

Operational Requirements	
4	The Controller shall communicate with the SPS (Simulation) via OPC UA
10	The database shall be implemented with EF-Core.
31	The system shall be able to be stopped by the operator and spectator.

3.2 SYSTEM ARCHITECTURE

The Simple Scada software consist of 3 software sub-systems. The project is divided into the subsystems:

- **Controller /Gateway:** The main responsibility of the Controller is to control the virtual production site so that the manufacturing process is according to the users wishes.
- **Database:** The main purpose of the Database is to store the information which is used by the submodules of the project. Furthermore, the database is used as a communication interface between the subsystems.
- **Desktop UI**: The UI is used to make the information which is generated by the program accessible for the human operators. In this submodule the user is also able to enter information into the program which is used to generate virtual products according to the users requirements.

In the figure xx an overview of the general system architecture is shown.

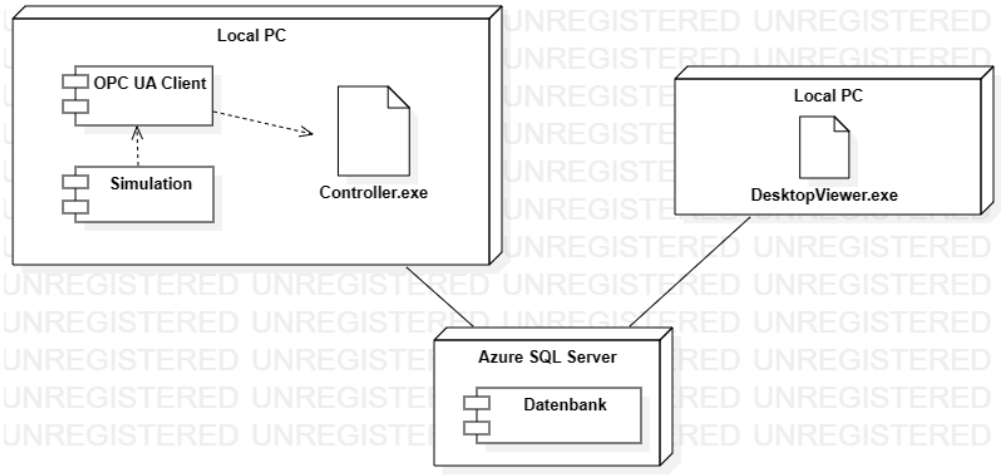


Figure 1 Architecture Simple Scada project

The detailed design of each sub-system is described in the following chapters.

3.3 SUBSYSTEM 1 - Controller / Gateway

3.3.1 SUBSYSTEM1 ARCHITECTURE

The Subsystem Controller/Gateway contains 2 main models:

- Conveyor belt control module
- Machine listener module

In the following chapters all modules will be explained in detail. The main purpose of this subsystem is to enable a way of executing instructions and inform about the state of the real plant / simulation to the graphical user interface. The interface that provides the communication channel between the interface and the controller subsystem is the database. For the control part its all a matter of requested state changes. The figure below gives and idea of how the modules relate together.

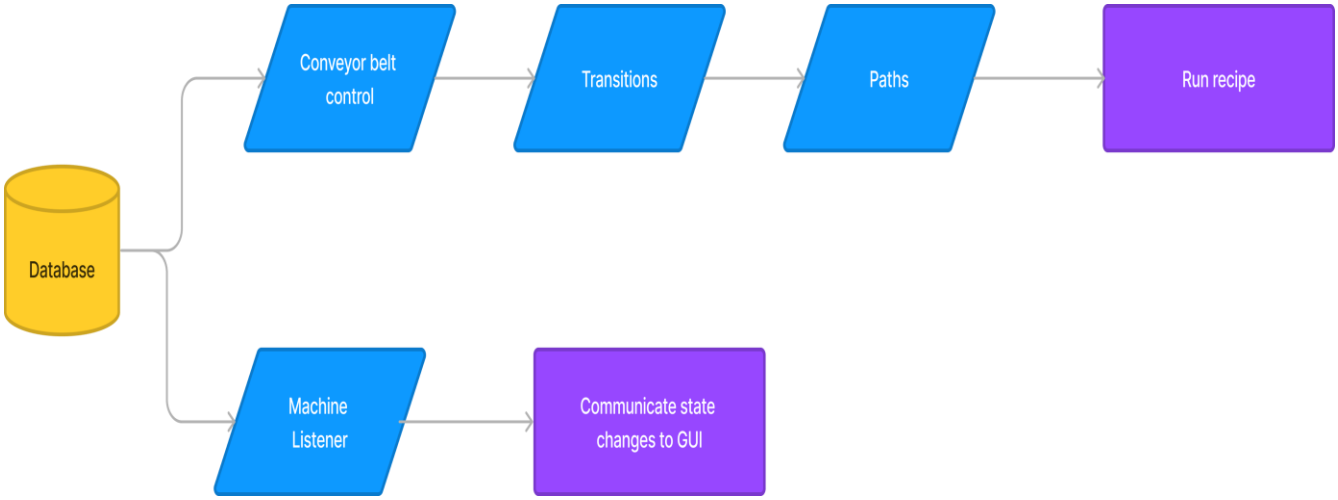


Figure 2: Main modules and relation

Using the database as the communication platform for the whole application provides and easy and reliable option to communicate between subsystems without having the need of another external communication way.

3.3.2 SUBSYSTEM1 INTERFACE

- OPC UA SDK for. NET Client/Server

3.3.3 SUBSYSTEM1 Paths

3.3.3.1 General information

To make the code reusable and modular the whole conveyor belt complex is split into smaller portions. This approach has plenty of advantages in terms of testability, changeability, and integration. However, each path needs specific commands for the simulated PLC, that means that each movement from one to another transfer-station had to be tested and programmed. In the following illustration the splitting of the paths is shown:

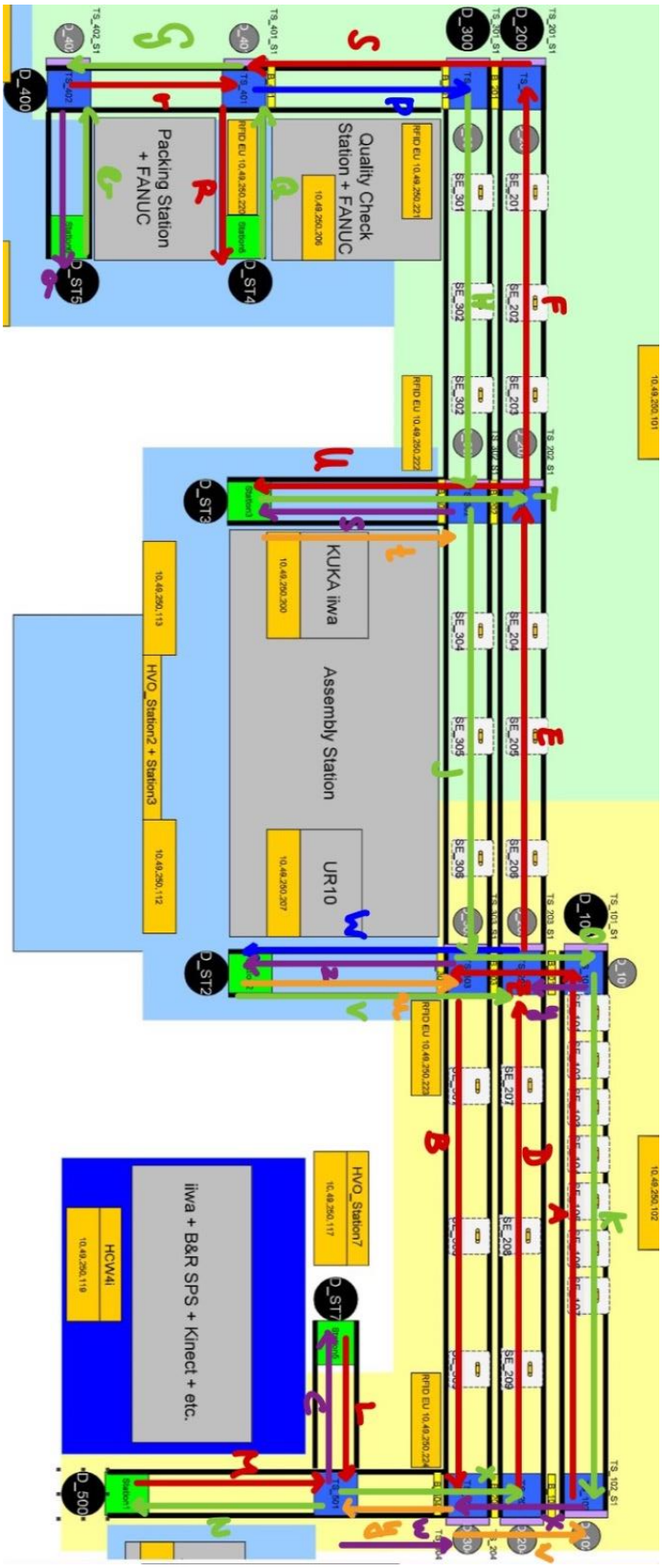


Figure 3: Split up – path map

In the figure it is visible how all the possible paths are named with a specific letter from the alphabet, the further on usage of these paths is explained later in the chapter Transitions.

3.3.3.2 OPCUA Library

Before continuing with the implementation of the code, first a general explanation of how and what was used with the library.

The simulated PLC acts as a server, so the subsystem is the client. The main features that have been used are first, the writing and second the reading of nodes. After the library is included the connection to the server needs to be established:

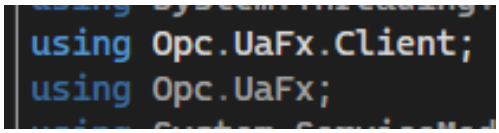


Figure 4: Including the OPCUA Client Library

```
_Client = new OpcClient("opc.tcp://127.0.0.1:48102");
_Client.Connect();
```

Figure 5: Establishing a connection to the server.

The only thing that needs to be done is putting the right address for the constructor of the Opclient-class and connecting it with the "Connect"-Method. To execute write and read instructions the following lines of code were used:

```
_Client.WriteNode("ns=6;s=::AsGlobalPV:D_100_F", false);
_Client.ReadNode("ns=6;s=::AsGlobalPV:D_100_F");
```

Figure 6: Reading and writing nodes.

The string contains node-address information and for writing nodes a Boolean need to be sent too to declare what state should be performed. (Dependent on what datatype your node is that datatype should be sent)

3.3.3.3 Code implementation

When the OPC UA- connection is established, for each letter in the path map a command-chain for the PLC needs to be coded. All specific lines of code used are already explained in the previous topics. For example, a route from Transferstation 302 to Station 3 realized in code looks like this (Path S):

```
10 references
public void runPath_S()
{
    _Client.WriteNode("ns=6;s=::AsGlobalPV:TS_201_U", true);
    _Client.WriteNode("ns=6;s=::AsGlobalPV:TS_301_U", true);

    _Client.WriteNode("ns=6;s=::AsGlobalPV:TS_201_F", true);
    _Client.WriteNode("ns=6;s=::AsGlobalPV:TS_301_F", true);

    _Client.WriteNode("ns=6;s=::AsGlobalPV:D_400_F", true);


    while (_Client.ReadNode("ns=6;s=::AsGlobalPV:TS_401_S1").ToString() == "False")
    {
        //Wait until sensor true
    }

    _Client.WriteNode("ns=6;s=::AsGlobalPV:TS_201_U", false);
    _Client.WriteNode("ns=6;s=::AsGlobalPV:TS_301_U", false);

    _Client.WriteNode("ns=6;s=::AsGlobalPV:TS_201_F", false);
    _Client.WriteNode("ns=6;s=::AsGlobalPV:TS_301_F", false);

    _Client.WriteNode("ns=6;s=::AsGlobalPV:D_400_F", false);

}
```

Figure 7: Path S (TS 304 to ST3)

3.3.3.4 Problems

Due to the deterministically buggy plant simulation, sometimes special handling of paths had to be done. At some certain spots intentionally driving forwards and backwards a few times was needed to get out a stuck position. That explains various coding approaches for very similar routes.

3.3.4 SUBSYSTEM1 Transitions

This object contains all the possible transitions from one station to another one. Each transition consists of multiple paths that get executed in a specific order to get the carrier from one to another place. The approach chosen to realize a solution to this is by a big switch-case that triggers a controlled execution chain of paths. It provides only one method for public uses that takes parameters to declare from where to where the carrier should drive and a Cancelation token to realize asynchronous tasks for the other module. The implementation of this function looks as shown:

```
public void RouteAsync(int[] sequence, CancellationToken ct)
{

    if (ct.IsCancellationRequested)
    {
        ct.ThrowIfCancellationRequested();
    }

    //erste Zahl=Von  zweite Zahl=Bis

    string sequenceString = sequence[0].ToString() + sequence[1].ToString();

    switch (sequenceString)
    {

        FromStart


        #region From1

        case "10":
            sequence_1to0();
            break;

        case "12":
            sequence_1to2();
            break;

        case "13":
            sequence_1to3();
            break;

        case "14":
            sequence_1to4();
            break;
```

Figure 8: Snippet of the routing algorithm

The string input displays a sequence of two numbers, that means from where to where the carried should go.

The series of commands in the beginning just asks if a cancellation Request was being executed before.

The other 30 methods classify and execute the needs sequence to transition the carrier to its wished destination, containing all of the alphabetical- functions from the last chapter.

```
2 references
private void sequence_1to0()
{
    _Paths.runPath_M();
    _Paths.runPath_W();
    _Paths.runPath_V();

}

2 references
private void sequence_1to2()
{
    _Paths.runPath_M();
    _Paths.runPath_X();
    _Paths.runPath_D();
    _Paths.runPath_W();
}
```

Figure 9: Snippet of the sequence-methods

3.3.5 SUBSYSTEM1 Conveyor belt control

3.3.5.1 General information

Another core-module of this subsystem is the controlling one. It is able to check requested states cyclically and at the same time to execute the movement of the carrier. The figure below shows what the two main processes are:

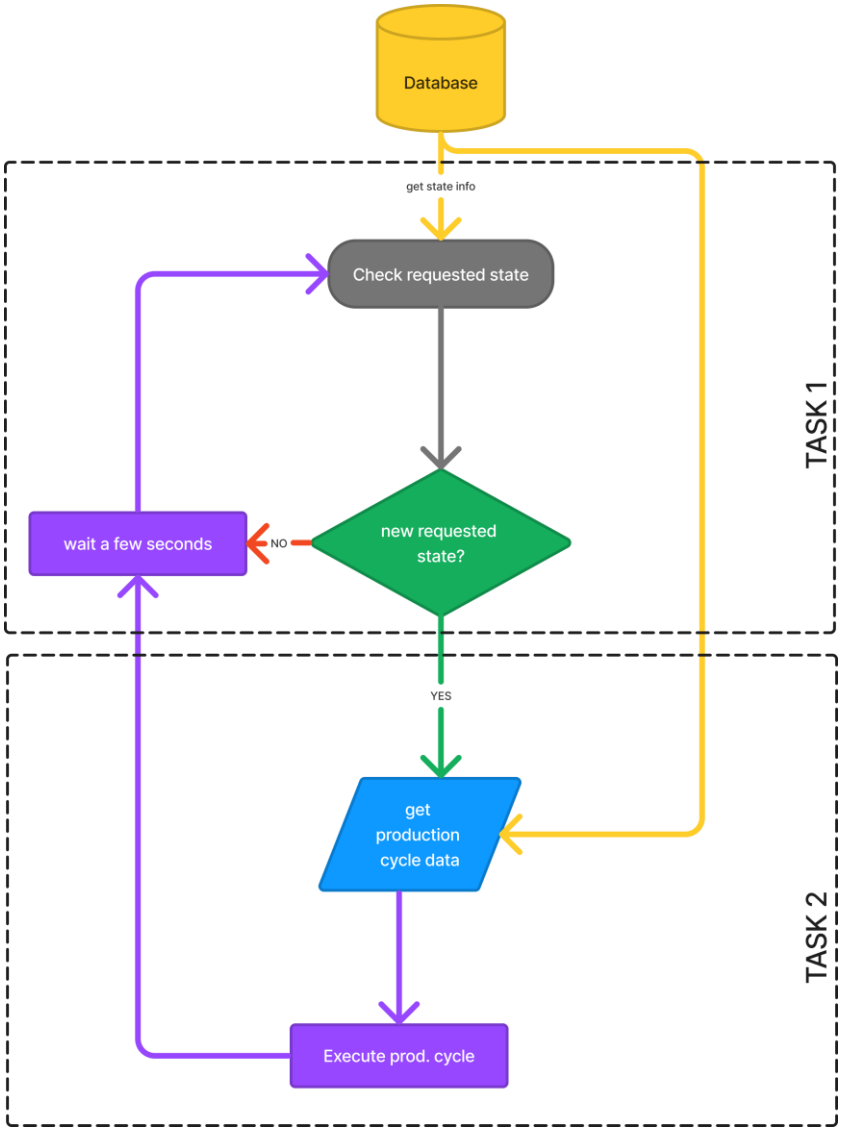


Figure 10: Flow diagram of control conveyor belt class

The rough idea is to split the two mentioned jobs in two tasks as you can see. This approach requires precise control over the code because aborting, starting, or pausing the plant shouldn't bring any errors in any possible combination.

3.3.5.2 Code implementation

First of all, a bunch of shared resource objects as variables had to be included to be able to use the database information. Also instances for the asynchronous programming had to be created. The following figure shows the included libraries and the created instances or variables for this module. Their specific implementation and usage are explained later.

```
using Microsoft.IdentityModel.Tokens;
using Opc.Ua;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
using SimpleSCADA_SharedResources;
using static System.Collections.Specialized.BitVector32;
using Syncfusion.Windows.Forms.Chart;

namespace SimpleSCADA_DesktopViewer.Control_f
{
    3 references
    public class ConveyorbeltControl
    {
        int _ControlVar;
        Transitions _Transitions;
        SimpleSCADA_SharedResources.Recipe _CurrentRecipe;
        SimpleSCADA_SharedResources.Plant _Plant;
        SimpleSCADA_SharedResources.Plant _NewPlant;
        SimpleSCADA_SharedResources.PlantManager _PlantManager;
        SimpleSCADA_SharedResources.ProductionCycle _CurrentProductionCycle;
        SimpleSCADA_SharedResources.Error _Error;
        SimpleSCADA_SharedResources.Log _Log;
        Task _Task;
        CancellationTokenSource _CancellationTokenSource;
        CancellationToken _CancellationToken;
        static ManualResetEvent _PauseEvent = new ManualResetEvent(true);
```

Figure 11: Included libraries and variables.

In the constructor some needed objects got instantiated.

```
public ConveyorbeltControl()
{
    this._Transitions = new Transitions();
    this._CurrentRecipe = new SimpleSCADA_SharedResources.Recipe();
    this._NewPlant = new SimpleSCADA_SharedResources.Plant();
    this._PlantManager = new SimpleSCADA_SharedResources.PlantManager();
    this._ControlVar = 0;
}
```

Figure 12: Constructor

The ExecutePlant function is one of the main tasks this class contains, it makes sure to check the requested state every couple seconds. Besides that it starts, stops or pauses the plant whenever the user wishes for it. But lets start from the beginning, first of all the most current Plant objects gets extracted from the database and sets up a few cancellation token objects for aborting a running task:

```
_Plant = new Plant();
_Plant = _PlantManager.GetPlant();
_CancellationTokenSource = new CancellationTokenSource();
_CancellationToken = _CancellationTokenSource.Token;
```

Figure 13: Set up part.

When starting the endless while-loop old plant state and new plant state gets compared to abort loop execution if needed:

```
_NewPlant = new Plant();
_NewPlant = _PlantManager.GetPlant();

if (_NewPlant.requestedPlantState == _Plant.requestedPlantState)
{
    Thread.Sleep(1000);
    continue;
}

_Plant = new Plant();
_Plant = _NewPlant;

bool abortCondition = false;
```

Figure 14: Comparing mechanism.

In the following switch case statement depending on what plant state should be different actions take place. Looking at the RUNNING-state a asynchronous task starts to make the plant running while still checking on the requested state. There is a cancellation token sent with it to be able to cancel the task and some error handling with the try-catch method.

```
switch (_Plant.requestedPlantState)
{

    case PlantState.RUNNING:
        update_DB_tblPlant_state(PlantState.RUNNING);
        if (_ControlVar > 0) { _PauseEvent.Set(); break; }
        _Task = Task.Run(() =>
        {
            try
            {
                // Your function code here
                _ControlVar++;
                runPlant();
            }
            catch (Exception ex)
            {
                if (ex.ToString().Substring(0, 3) == "Opc")
                {
                    _Error = new Error("No connection to OPCUA-Server",ErrorType.OPC_UA,ErrorSeverity.SHITTY,DateTime.Now);
                    ErrorManager.AddError(_Error);
                }
            }
        }, _CancellationToken);

        break;
```

Figure 15: Run state.

The other states basically all do the same things, notable is the execution of the Cancel- function when stopping the plant and the Pause-Reset-event to pause the plant.

```
case PlantState.STOP:
    update_DB_tblPlant_state(PlantState.STOP);
    _CancellationTokenSource.Cancel();
    _ControlVar = 0;
    break;

case PlantState.PAUSED:
    update_DB_tblPlant_state(PlantState.PAUSED);
    _PauseEvent.Reset();
    break;

case PlantState.IDLE:
    update_DB_tblPlant_state(PlantState.IDLE);
    //chillex
    break;

case PlantState.FINISHED:
    update_DB_tblPlant_state(PlantState.FINISHED);
    _ControlVar = 0;
    break;

case PlantState.MAINTENANCE:
    update_DB_tblPlant_state(PlantState.MAINTENANCE);
    _PauseEvent.Reset();
    break;

default:
    abortCondition = true;
    break;
```

Figure 16: Other cases.

Another method that was used a few times is to update the plant state. It just updates the plant state to the database.

```
6 references
private void update_DB_tblPlant_state(PlantState plantState)
{
    _Plant.ActualPlantState = plantState;

    _PlantManager.EditPlant(_Plant);

}
```

Figure 17: Update plant state

The other core method that was implemented is the runPlant-method. It holds all necessary logic to perform transitions from one to another station executing all the mentioned Methods from above. In the beginning of the function mandatory initializations are performed:

```
1 reference
private void runPlant()
{
    //Initialize vars

    _CurrentProductionCycle = _Plant.ActiveProductionCycle;
    _CurrentRecipe = _CurrentProductionCycle.Recipe;

    List<StationWithChosenUtility> neededStationsWithUtility = _PlantManager.GetOrderedStationsToDrive(_Plant);

    //get parameter for process

    int cycles = _CurrentProductionCycle.NumberOfProducts;

    List<int> stationOrder = new List<int>();

    foreach (StationWithChosenUtility curStatition in neededStationsWithUtility)
    {
        stationOrder.Add(int.Parse(curStatition.Station.Name.Remove(0, 8)));
    }
```

Figure 18: Preparing for conveyor belt execution.

Its followed by an for-loop with a specific number of iterations fitting to the recipe to be executed. This loop also presents the main loop where all core operations happen. It mainly performs the transition and methods for carrier movement. It also takes care of the right communication to the database to always be up to date for the interface.

```
foreach (int Goal in stationOrder)
{

    if (i == stationOrder.Count - 1) //ending logik
    {
        Console.WriteLine("Going from " + previousGoal.ToString() +" to "+ Goal.ToString());
        _Transitions.Route(new int[] { previousGoal, Goal });
        Thread.Sleep(neededStationsWithUtility[i].ChosenUtility.SecondsToWait * 1000);
        Console.WriteLine("Going from " + Goal.ToString() + " to Startpoint") ;
        _Transitions.Route(new int[] { Goal, 0 });
        continue;

    }

    if (i == 0) //starting logik
    {
        Console.WriteLine("Going from " +  "Startpoint to " + Goal.ToString());
        _Transitions.Route(new int[] { 0, Goal });
        Thread.Sleep(neededStationsWithUtility[i].ChosenUtility.SecondsToWait * 1000);
        previousGoal = Goal;
        i++;
        continue;
    }

    Console.WriteLine("Going from " + previousGoal.ToString() + " to " + Goal.ToString());
    _Transitions.Route(new int[] { previousGoal, Goal });

    Thread.Sleep(neededStationsWithUtility[i].ChosenUtility.SecondsToWait * 1000);

    previousGoal = Goal;
    i++;

    _NewPlant = new Plant();
    _NewPlant = _PlantManager.GetPlant();

    if (_NewPlant.requestedPlantState == PlantState.PAUSED || _NewPlant.requestedPlantState == PlantState.MAINTENANCE)
    {
        _PauseEvent.WaitOne();
    }

    if (_NewPlant.requestedPlantState == PlantState.STOP)
    {
        break;
    }

}
```

Figure 19: Core loop for plant execution.

As the ending part some ending logic was implemented to get the plant and the user interface to a defined state. State handling and throwing errors if they exist is the focus here.

```
    if (_NewPlant.requestedPlantState == PlantState.STOP)
    {
        if (_CancellationToken.IsCancellationRequested)
        {
            _Error = new Error("Plant is stopped means uncertain plant situation", ErrorType.OPC_UA, ErrorSeverity.WARNING, DateTime.Now);
            ErrorManager.AddError(_Error);
            break;
        }
    }
    _Plant.ActiveProductionCycle.FinishedProducts++;
    Console.WriteLine("Finished products in active cycle: "+_Plant.ActiveProductionCycle.FinishedProducts.ToString());
    _PlantManager.EditPlant(_Plant);
}

if (_Plant.ActiveProductionCycle.FinishedProducts == _Plant.ActiveProductionCycle.NumberOfProducts)
{
    _Plant.ActualPlantState = PlantState.FINISHED;
}
```

Figure 20:Handlling of the ending procedure of a Production cycle.

3.3.6 SUBSYSTEM1 Machine listener

3.3.6.1 General information

To supervise the plant this subsystem needs to acquire, filter, and finally send relevant data to the database and therefore the interface. To do that OPCUA Subscriptions were used. These Subscriptions observe data nodes and execute a task if the observed node value changes.

3.3.6.2 Code implementation

To use the OPCUA methods, they have to be included first. This module also needs to run asynchronous, so the required Library was also used. To access the methods in this class the default constructor was used.

```
using Opc.UaFx.Client;
using Opc.UaFx;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SimpleSCADA_SharedResources
{
    3 Verweise
    public class MachineListener
    {
        #region Constructor
        1 Verweis
        public MachineListener()
        {

        }
        #endregion
```

Figure 21: included libraries and constructor

One of the two important methods in the MachineListener class is the SubscribeNodes method. It contains the addresses of all relevant sensor nodes and subscribes them via the OPCUA method. If the value of one of these nodes changes, the "HandleDataChanged" method is invoked.

```
OpcSubscribeDataChange []
commands = new OpcSubscribeDataChange[] {

Sensors

Transferstations

#region Stations
new OpcSubscribeDataChange("ns=6;s=::AsGlobalPV:Station_1_SO", HandleDataChanged),
new OpcSubscribeDataChange("ns=6;s=::AsGlobalPV:Station_2_SO", HandleDataChanged),
new OpcSubscribeDataChange("ns=6;s=::AsGlobalPV:Station_3_SO", HandleDataChanged),
new OpcSubscribeDataChange("ns=6;s=::AsGlobalPV:Station_4_SO", HandleDataChanged),
new OpcSubscribeDataChange("ns=6;s=::AsGlobalPV:Station_5_SO", HandleDataChanged),
new OpcSubscribeDataChange("ns=6;s=::AsGlobalPV:Station_6_SO", HandleDataChanged),
#endregion

};

OpcSubscription subscription = client.SubscribeNodes(commands);
```

Figure 22: SubscribeNodes method

When invoked, the acquired data from the node consists of two variables: “value” and “NodeId”. “NodeId” contains the node address mentioned previously and “value” either a “False” or “True”. Both variables need to be parsed into the correct data types. The address-string from “NodeId” is also filtered. With the converted variables a sensorData object is created. If the “value” variable was “True”, meaning the sensor registered an object, the sensorData object is sent to the database via the EditSensorData method from the plantManager class.

```
PlantManager plantManager = new PlantManager();

44 Verweise
private void HandleDataChanged(object sender, OpcDataChangeReceivedEventArgs e)
{

    OpcMonitoredItem item = (OpcMonitoredItem)sender;

    bool value = bool.Parse(e.Item.Value.ToString());
    string NodeId = item.NodeId.ToString();
    NodeId = NodeId.Substring(20, NodeId.Length-23);

    SensorData sensorData = new SensorData(NodeId, value);

    if (value)
    {
        plantManager.EditSensorData(sensorData);
    }
}
```

Figure 23: HandleDataChanged method

3.4 SUBSYSTEM 2 – SQL Database

3.4.1 SUBSYSTEM2 ARCHITECTURE

The architecture of the Subsystem SQL Database, also known as SharedResources in the Project Repository, consists of 3 basic pillars:

- The Database Model Classes and Enums
- The Database itself
- The Database Communication Managers

3.4.1.1 The Database Model Classes and Enums

The Architecture of the Models as basis for the Tables of the Database looks as follows:

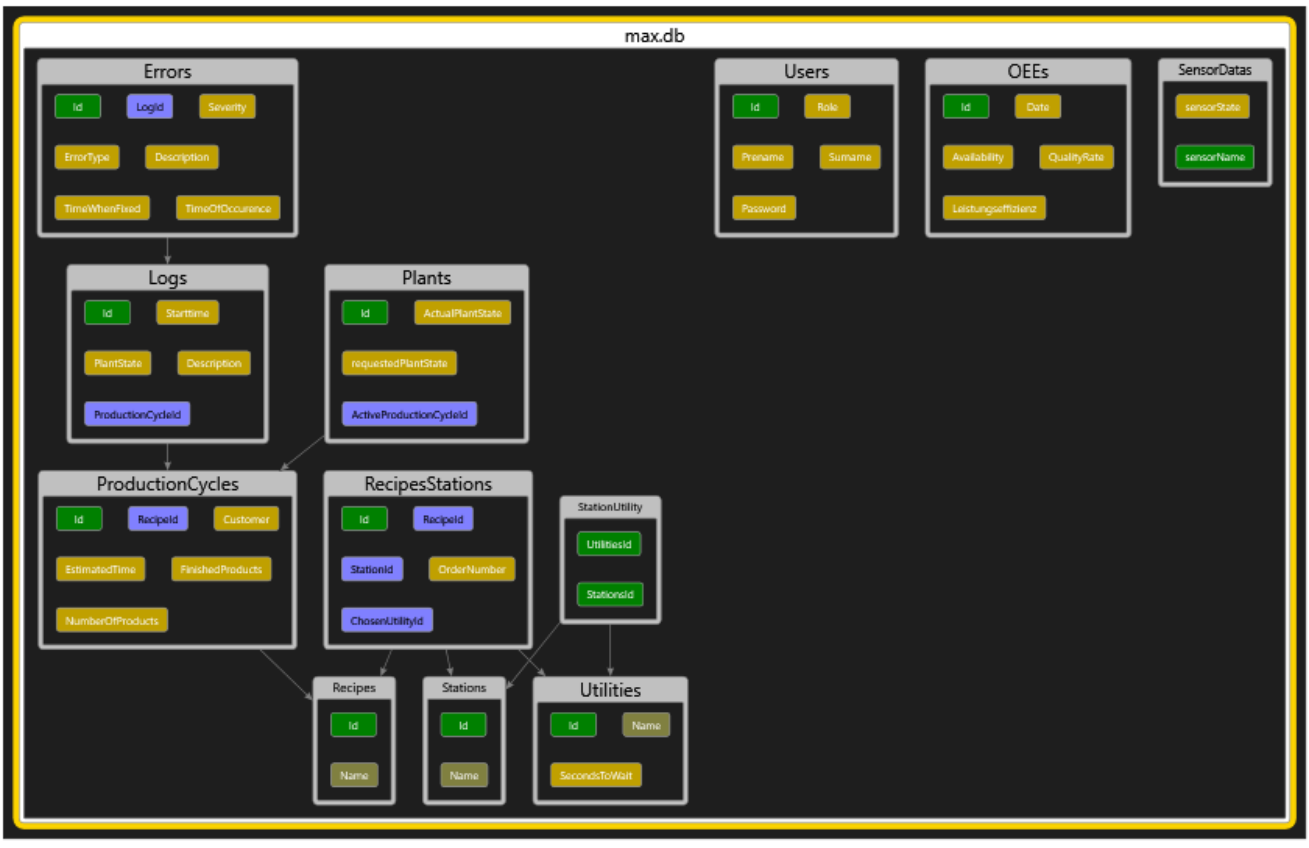


Figure 24 Data base architecture

The following Models were used:

- Model Utility
- Model Station
- Model Recipe
- Model StationWithChosenUtility (also known as the JunctionTable "RecipesStations" in the Database, realizes the m:n relationship between Recipe and Station

- Model ProductionCycle
- Model Plant
- Model SensorData
- Model User
- Model Log
- Model Error
- Model OEE

The properties, methods and use of each Model class are being described further detailed in the according Managers below from chapter 3.4.3 to 3.4.9.

To achieve a categorization of certain properties such as "UserRole" in the Model User, the following Enums were used to organize such Properties:

- UserRole: With the states SUPERUSER, OPERATOR and SPECATOR
- PasswordLevel: with the states VeryLow, Low, Medium, Strong, VeryStrong
- PlantState: with the states RUNNING, DISTURBANCE, MAINTAINENCE, PAUSED, IDLE, STOP and FINISHED
- ErrorType: with the states DATABASE, OPC_UA, DESKTOP_VIEW, THIS_IS_AN_ERROR
- ErrorSeverity: with the states CRITICAL, SHITTY and WARNING

3.4.1.2 The Database

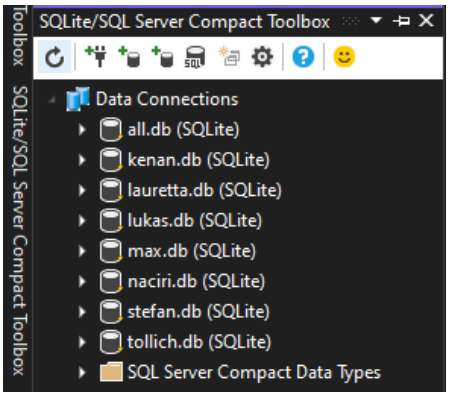
1. In Development-Phase a local Sqlite Database for each Developer was used
2. In Production-Phase a central Azure Database was used.

Development-Phase:

For the Development-Phase local Sqlite Databases, one for each developer, had to be used. With only one database in the Github repo at this stage of the project, parallel development wouldn't be possible, because each change of the database from two different developers would throw a merge at the git push.

On the other hand, simply using one Sqlite Database per developer and gitignore all .db files would also complicate things, because updates of the databases with Migrations wouldn't be possible anymore because they are not visible in the Github repo.

So a different idea was born: A local Sqlite database per developer visible in the repo, and only gitignore one .txt file in the /SharedResources/Databases folder. In that .txt file lies the path to each developers database, it can be changed by each developer without uploading to the repo. Also, because all databases are visible in the repo now, Migrations for updating the databases could be applied. Problem solved.



Another advantage with local databases per developer is that at each change of the database from a developer, the change can be looked up quickly without having to exit Visual Studio, with a Visual Studio Extension called "SQL Compact Toolbox". That makes it also easy for non-database developers to quickly look up if their methods really caused changes the database.

One might ask: Why not use a central online database, for example from the Azure Cloud, also in the Development-Phase?

The problem in our case with such a solution was, that in Development-Phase the Program often was executed by two or more developers at the same time while developing, and with the start of the program also the (for each developer local) Simulation of the Plant in Unity started. Now the OPC_UA part of the two developers would constantly overwrite each other in the Database Table "SensorDatas", and thus the database would return garbage positions of the palette for both developers.

Production-Phase:

In the Production-Phase, a central online database with its according database server in the Azure Cloud was hosted.

Now, in the Production-Phase with basically only one physical Plant located in the building of the FH Wels, the program only has to be executed one time locally and connected with that Plant to upload the positions of the palettes to the online database.

Now, a user can create, start and stop recipes and interact with the real physical Plant from anywhere in the world.

The Azure-SQL-Database and its Database Server are described in further detail in chapter 3.4.12.

3.4.1.3 The Database Communication Managers

All Manager-classes used for sending and receiving data from the Database.

3.4.2 SUBSYSTEM2 INTERFACES

- EntityFrameworkCore 6.0.0

- EntityFrameworkCore.Sqlite 6.0.0
- EntityFrameworkCore.SqlServer 6.0.0

3.4.3 SUBSYSTEM2 MODULE 1: Recipe Manager

The Recipe Manager is responsible for creating, editing, and deleting recipes. Therefore, a table of the model Recipe, Station Utility and StationWithChosenUtility is in the Database with DbSet< >.

The Database settings are set in the region "Setting DbContext", see Figure 25.

```
10 Verweise
public class RecipeManager : DbContext
{
    #region Members
    9 Verweise
    public DbSet<Recipe>? Recipes { get; set; }
    6 Verweise
    public DbSet<Station>? Stations { get; set; }
    1 Verweis
    public DbSet<Utility>? Utilities { get; set; }
    0 Verweise
    public DbSet <StationWithChosenUtility>? RecipesStations { get; set; }
    #endregion

    #region Constructor
    2 Verweise
    public RecipeManager()
    {

    }
    #endregion

    #region Settings DbContext
    0 Verweise
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        var conn :string = $"Data Source=" + AbsoluteDbPath.PathGeneration();
        optionsBuilder.UseSqlite(conn);

        base.OnConfiguring(optionsBuilder);
    }

    0 Verweise
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        DbCreationContext.setAllModelConnections(modelBuilder);

        base.OnModelCreating(modelBuilder);
    }
    #endregion
```

Figure 25 Source-Code Recipe Manager except Methods

All Methods are implemented with a simple errorhandling, this means try-catch and in the case of an error, the error gets logged.

The Method GetRecipe, see Figure 26, returns the requested Recipe per id with the standard method SingleOrDefault() of DbSet< >.

```
#region Methods Recipes
2 Verweise
public Recipe GetRecipe(int id)
{
    try
    {
        return Recipes.SingleOrDefault<Recipe>(x :Recipe => x.Id == id);
    }
    catch (Exception e)
    {
        Error error = new Error( description: "Unable to fetch Recipe from Database!",
            ErrorType.DATABASE,ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
        return null;
    }
}
```

Figure 26 Source-Code Recipe Manager Method GetRecipe

The Method GetRecipes, see Figure 27, returns all recipes from the database with the attached properties, with the standard methods of DbSet< >.

```
12 Verweise
public List<Recipe> GetRecipes()
{
    try
    {
        return Recipes // DbSet<Recipe>?
            .Include( navigationPropertyPath: e :Recipe => e.Stations) // IIncludableQueryable<Recipe,ICollection<…>>
                .ThenInclude(i :StationWithChosenUtility => i.ChosenUtility) // IIncludableQueryable<Recipe,Utility>
                .ThenInclude(i :Utility => i.Stations) // IIncludableQueryable<Recipe,List<…>>
            .ToList(); // List<Recipe>

    }
    catch (Exception e)
    {
        //Console.WriteLine(e.ToString());
        Error error = new Error( description: "Unable to fetch Recipes from Database!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
        return null;
    }
}
```

Figure 27 Source-Code Recipe Manager Method GetRecipes

The Method AddRecipe, see Figure 28, adds an non existing Recipe to the Database, for this a foreach and if-elseif combination is implemented. The Recipe gets added to the DbSet<Recipe> Recipes with the standard method Add from DbSet< > and with the command SaveChanges() the added recipe gets saved to the database. This method can log two different errors.

```
1 Verweis
public void AddRecipe(Recipe recipe)
{
    try
    {
        bool isNameEqual = false;

        foreach (var checkRecipe in Recipes)
        {
            if (checkRecipe.Name == recipe.Name) isNameEqual = true; break;
        }

        if (Recipes.Contains(recipe) == false && isNameEqual == false)
        {
            Recipes.Add(recipe);
            SaveChanges();
        }
        else
        {
            Error error = new Error(description: "Recipe already exists!",
                ErrorType.DATABASE, ErrorSeverity.CRITICAL,
                DateTime.Now);
            ErrorManager.AddError(error);
        }
    }
    catch (Exception e)
    {
        Error error = new Error(description: "Unable to save Recipe to Database!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
    }
}
```

Figure 28 Source-Code Recipe Manager Method AddRecipe

The Method RemoveRecipe, see Figure 29, removes an already existing recipe from the database. Therefore the recipe must get fetched from the database with the standard method SingleOrDefault() from DbSet< > and gets then removed from the DbSet< > with the standard method Remove(). The change to database gets then saved with the command SaveChanges().

```
1 Verweis
public void RemoveRecipe(int recipeId)
{
    try
    {
        Recipe recipe = Recipes.SingleOrDefault(x :Recipe => x.Id == recipeId);
        Recipes.Remove(recipe);
        SaveChanges();
    }
    catch (Exception e)
    {
        Error error = new Error( description: "Unable to remove Recipe from Database!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
    }
}
```

Figure 29 Source-Code Recipe Manager Method RemoveRecipe

The Method EditRecipe, see Figure 30, edits an already existing recipe in the database, therefore the properties of the existing recipe must be assigned new values.

```
1 Verweis
public void EditRecipe(Recipe editedRecipe)
{
    try
    {
        Recipe recipeToEdit = GetRecipe(editedRecipe.Id);
        if (recipeToEdit != null)
        {
            recipeToEdit.Name=editedRecipe.Name;
            recipeToEdit.Stations=editedRecipe.Stations;
        }

        SaveChanges();
    }
    catch (Exception e)
    {
        Error error = new Error( description: "Unable to edit Recipe from Database!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
    }
}
```

Figure 30 Source-Code Recipe Manager Method EditRecipe

3.4.3.1 Model Utility

The Model Utility describes the purpose of a Station with the properties Id, Name, SecondsToWait , List of Station and ICollection of StationWithChosenUtility.

Therefore, the model Utility has a many to many relationship with the model Station, which gets modelled with the class StationWithChosenUtility, see 3.4.3.4. To model this many to many relationship, on the one hand is needed a List of Station and on the other hand a "joining" class StationWithChosenUtility. For further details look up the Microsoft EF-Core documentation.

Figure 31 is a screenshot of the source-code from the model Utility.

```
28 Verweise
public class Utility
{
    #region Properties
    3 Verweise
    public int Id { get; private set; }
    24 Verweise
    public string? Name { get; private set; }
    3 Verweise
    public List<Station> Stations { get; set; } = new();
    2 Verweise
    public ICollection<StationWithChosenUtility> ChosenUtilites { get; } = new List<StationWithChosenUtility>();
    35 Verweise
    public int SecondsToWait { get; set; }
    #endregion

    #region Constructor

    0 Verweise
    public Utility() {
    }
    5 Verweise
    public Utility(string name, int secondsToWait)
    {
        this.SecondsToWait = secondsToWait;
        this.Name = name;
    }
    #endregion

    #region Methods

    #endregion

}
```

Figure 31 Source-Code Utility

3.4.3.2 Model Station

The Model Station describes the place where a material carrier can stop and the material can get processed with the chosen utility, with the properties Id, Name, List of Utility and ICollection of StationWithChosenUtility.

Therefore, the model Station has a many to many relationship with the model Utility, which gets modelled with the class StationWithChosenUtility, see 3.4.3.4. To model this many to many relationship, on the one hand is needed a List of Utility and on the other hand a "joining" class StationWithChosenUtility. For further details look up the Microsoft EF-Core documentation.

The proper connection for this relationship gets set in the class DbCreationContext, see 3.4.10.

Figure 32 is a screenshot of the source-code from the model Station.

```
37 Verweise
public class Station
{
    #region Properties
    4 Verweise
    public int Id { get; set; }
    7 Verweise
    public string? Name { get; set; }

    21 Verweise
    public List<Utility> Utilities { get; set; } = new();

    4 Verweise
    public ICollection<StationWithChosenUtility> ChosenUtilities { get; } = new List<StationWithChosenUtility>();

    #endregion

    #region Constructor
    // Default Constructor for EF-Core
    0 Verweise
    public Station()
    {
        //Recipes = new List<Recipe>();
    }
    1 Verweis
    public Station(string name, List<Utility> utilities)
    {
        this.Name = name;
        this.Utilities = utilities;
    }
    #endregion

    #region Methods

    #endregion

}
```

Figure 32 Source-Code Station

3.4.3.3 Model Recipe

The Model Recipe describes the sequence of stations which the material carrier has to follow and which Production-Cycle has this Recipe. For this the following properties are needed Id, Name ICollection of StationWithChosenUtility and ICollection Productioncycle.

Therefore, the model Recipe has a one to many relationship with ProductionCycle, which means a Recipe can have many ProductionCycles but a ProductionCycle can have only one Recipe.

The proper connection for this relationship gets set in the class DbCreationContext, see 3.4.10, for further details look up the Microsoft EF-Core documentation.

The method `AddStation(StationWithChosenUtility station)` is for adding a Station to the Recipe and `RemoveStation(StationWithChosenUtility station)` is for removing a Station from the Recipe.

Figure 33 and Figure 34 are screenshots of the source-code from the model Recipe.

```
51 Verweise
public class Recipe
{
    #region Properties
    11 Verweise
    public int Id { get; private set; }
    20 Verweise
    public string? Name { get; set; }
    16 Verweise
    public ICollection<StationWithChosenUtility> Stations { get; set; } = new List<StationWithChosenUtility>();
    2 Verweise
    public virtual ICollection<ProductionCycle> ProductionCycles { get; } = new List<ProductionCycle>(); // 1 to many with ProductionCycle

    #endregion

    #region Constructor
    // Default Constructor for EF-Core
    2 Verweise
    public Recipe()
    {

    }

    3 Verweise
    public Recipe(string name)
    {
        Name = name;
    }

    // Constructor only for Use in EditRecipe()
    1 Verweis
    public Recipe (int existingId, string name)
    {
        Id = existingId;
        Name = name;
    }
    #endregion
```

Figure 33 Source-Code Recipe Part 1/2

```
    #region Methods
    1 Verweis
    public void AddStation(StationWithChosenUtility station)
    {
        try
        {
            Stations.Add(station);
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
            throw;
        }
    }
    0 Verweise
    public void RemoveStation(StationWithChosenUtility station)
    {
        Stations.Remove(station);
    }
    #endregion

}
```

Figure 34 Source-Code Recipe Part 2/2

3.4.3.4 Model RecipeStation

The model RecipeStation is the class StationWithChosenUtility and has the purpose of creating the many to many relationship between Station and Utility. Also, this model is needed to get the proper relationship between Recipe and Station. For this the properties Id, Ordernumber, StationId, Station, RecipeId, Recipe ChosenUtiltyId and ChosenUtility are needed.

Figure 35 is a screenshot of the source-code from the model RecipeStation.

```
39 Verweise
public class StationWithChosenUtility
{
    #region Properties
    2 Verweise
    public int Id { get; private set; }
    8 Verweise
    public int OrderNumber { get;set; }
    10 Verweise
    public int StationId { get; set; }
    13 Verweise
    public Station Station { get; set; } = null!;

    2 Verweise
    public int RecipeId { get; set; }
    4 Verweise
    public Recipe Recipe { get; set; } = null!;

    2 Verweise
    public int ChosenUtilityId{ get; set; }
    38 Verweise
    public Utility ChosenUtility { get; set; } = null!;


    #endregion

    #region Constructor
    0 Verweise
    public StationWithChosenUtility()
    {

    }

    0 Verweise
    public StationWithChosenUtility(Recipe recipe,Station station, Utility chosenUtility)
    {
        Recipe = recipe;
        Station = station;
        ChosenUtility = chosenUtility;
    }
    1 Verweis
    public StationWithChosenUtility(Recipe recipe, Station station, Utility chosenUtility, int orderNumber)
    {
        Recipe = recipe;
        Station = station;
        ChosenUtility = chosenUtility;
        OrderNumber = orderNumber;
    }
    #endregion

    #region Methods

    #endregion
}
```

Figure 35 Source-Code StationWithChosenUtility

3.4.4 SUBSYSTEM2 MODULE 2: ProductionCycle Manager

The ProductionCycle Manager only holds a DbContext of ProductionCycles.

```
11 Verweise
public class ProductionCycleManager : DbContext
{
    #region Properties
    9 Verweise
    private DbSet<ProductionCycle>? ProductionCycles { get; set; }

    #endregion

    #region Constructor

    4 Verweise
    public ProductionCycleManager()
    {

    }
```

The Method GetProductionQueue figures out the not-finished ProductionCylces and puts them in a Queue order. The Method GetProductionQueueAsync does the same job only as an async Task for the GUI Team.

```
public  List<ProductionCycle> GetProductionQueue()
{
   return ProductionCycles
          .Include(e => e.Recipe)
            .ThenInclude(e => e.Stations)
               .ThenInclude(i=>i.ChosenUtility)

           .Include(e => e.Recipe)
            .ThenInclude(e => e.Stations)
               .ThenInclude(e => e.Station)

           .Include(e=>e.Plant)

          .ToList()
          .FindAll(productioncycle => productioncycle.FinishedProducts == 0 && productioncycle.Plant == null);
}
```

The Method AddProductionCycle adds a new ProductionCycle to the existing ones.

```
1 Verweis
public void AddProductionCycle(ProductionCycle productionCycle)
    {
        try
        {
            bool isOrderNumberEqual = false;
            foreach (var checkProductionCycle in ProductionCycles)
            {
                if (checkProductionCycle.Id == productionCycle.Id) isOrderNumberEqual = true;
            }

            if (ProductionCycles.Contains(productionCycle) == false && isOrderNumberEqual == false)
            {
                ProductionCycles.Add(productionCycle);
                 SaveChanges();
            }
            else
            {
                Error error = new Error("Recipe already exists!",
                    ErrorType.DATABASE, ErrorSeverity.CRITICAL,
                    DateTime.Now);
                ErrorManager.AddError(error);
            }
        }

        catch (Exception e)
        {
            Error error = new Error("Unable to save ProductionCycle to Database!",
                ErrorType.DATABASE, ErrorSeverity.CRITICAL,
                DateTime.Now);
            ErrorManager.AddError(error);
        }
    }
```

The Method EditProductionCycle edits a chosen ProductionCycle from the existing ones.

```
1 Verweis
public void EditProductionCycle(ProductionCycle editedProductionCycle)
{
    try
    {
        ProductionCycle productionCycleToEdit = GetProductionCycle(editedProductionCycle.Id);

        if (productionCycleToEdit != null)
        {
            productionCycleToEdit.Customer = editedProductionCycle.Customer;
            productionCycleToEdit.NumberOfProducts = editedProductionCycle.NumberOfProducts;
            productionCycleToEdit.RecipeId = editedProductionCycle.RecipeId;
        }

        SaveChanges();
    }
    catch (Exception e)
    {
         Error error = new Error("Unable to edit ProductionCycle from Database!",
             ErrorType.DATABASE, ErrorSeverity.CRITICAL,
             DateTime.Now);
         ErrorManager.AddError(error);
    }
```

The Method DeleteProductionCycle deletes a chosen ProductionCycle from the existing ones.

```
1 Verweis
public void DeleteProductionCycle(ProductionCycle productionCycle)
{

    try
    {
        ProductionCycle productionCycleToDelete = ProductionCycles.SingleOrDefault(x => x.Id == productionCycle.Id);
        ProductionCycles.Remove(productionCycleToDelete);
        SaveChanges();
    }
    catch (Exception e)
    {
        Error error = new Error("Unable to remove ProductionCycle from Database!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
    }
```

The Method GetProductionCycle gets a single chosen ProductionCycle out of the exisiting ones. That was especially useful and often used by the GUI Team.

```
4 Verweise
public ProductionCycle GetProductionCycle(int id)
{
    try
    {
        return ProductionCycles.SingleOrDefault<ProductionCycle>(x => x.Id == id);
    }
    catch (Exception e)
    {
        Error error = new Error("Unable to fetch ProductionCycle from Database!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
        return null;
    }
}
```

For ProductionHistoryView we needed to show all already finished ProductionCycles. GetFinishedProductionCycles does the job.

```
1 Verweis
public List<ProductionCycle> GetFinishedProductionCycles() // only for ProductionHistoryView
{
    return ProductionCycles
        .Where(x => x.NumberOfProducts == x.FinishedProducts)
        .Include(e => e.Recipe)
        .ToList();
}
```

3.4.4.1 Model ProductionCycle

The properties of the model class ProductionCycle are as follows:

```
45 Verweise
public class ProductionCycle
{
    #region Properties
    25 Verweise
    public int Id { get; private set; }
    8 Verweise
    public int RecipeId { get; set; }// 1 to many with recipe
    24 Verweise
    public Recipe Recipe { get; set; } = null!; // 1 to many with recipe
    2 Verweise
    public virtual ICollection<Log> Logs { get; set; } = new List<Log>(); // 1 to many with Logs
    6 Verweise
    public Plant? Plant { get; set; } // 1 to 1 with Plant

    16 Verweise
    public string Customer { get;  set; }
    23 Verweise
    public int NumberOfProducts { get;  set; }
    23 Verweise
    public int FinishedProducts { get; set; }
    1 Verweis
    public double EstimatedTime { get;  set; }
    #endregion
```

Multiple Constructors for various usecases had to be developed, both for the database team and for the GUI Team in some cases. One Constructor was developed for the OPC-UA Team, but was later than not used for a better solution was found.

```
#region Constructor
0 Verweise
public ProductionCycle (string customer, int numberOfProducts) // constructor for kenan
{
    Customer = customer;
    NumberOfProducts = numberOfProducts;
}
// Für Datenbank
1 Verweis
public ProductionCycle(string customer, int recipeId, int numberOfProducts)
{
    Customer = customer;
    //Recipe = recipe;
    NumberOfProducts = numberOfProducts;
    FinishedProducts = 0;
    RecipeId = recipeId;
    //EstimatedTime = calculated from recipe and numberOfProducts, right?
}
// Nur für die View
1 Verweis
public ProductionCycle(string customer, Recipe recipe, int numberOfProducts)
{
    Customer = customer;
    Recipe = recipe;
    NumberOfProducts = numberOfProducts;
    FinishedProducts = 0;
    //EstimatedTime = calculated from recipe and numberOfProducts, right?
}
```

```
// Nur für die View
1 Verweis
public ProductionCycle(int existingId, string customer, int recipeId, int numberOfProducts)
{
    Id = existingId;
    Customer = customer;
    //Recipe = recipe;
    NumberOfProducts = numberOfProducts;
    FinishedProducts = 0;
    RecipeId = recipeId;
    //EstimatedTime = calculated from recipe and numberOfProducts, right?
}
// Nur für die View
1 Verweis
public ProductionCycle(int existingId, string customer, Recipe recipe, int numberOfProducts)
{
    Id = existingId;
    Customer = customer;
    Recipe = recipe;
    NumberOfProducts = numberOfProducts;
    FinishedProducts = 0;
    //EstimatedTime = calculated from recipe and numberOfProducts, right?
}

#endregion
```

3.4.5 SUBSYSTEM2 MODULE 3: Plant Manager

The PlantManager was probably the most quickly developed and the last thing we tweaked before Presentation to make the program run smoothly with the probability of a critical error as low as possible. The PlantManager holds only a DbSet of Plants and a DbSet of SensorDatas for figuring out the position of the Palette on the Plant.

```
#region Properties
2 Verweise
private DbSet<Plant> Plants { get; set; }
5 Verweise
private DbSet<SensorData>? SensorDatas { get; set; }

#endregion

#region Constructor

8 Verweise
public PlantManager()
{

}
```

The Method GetPlant returns the Plant With all the Properties it holds, such as the Active ProductionCycle with its Recipe and Stations and their Chosen Utility. The Method GetPlantAsync does the same job for the GUI Team only as an async Task.

```
1 Verweis
private Plant GetPlantPrivate()
{
    return Plants

            .Include(e => e.ActiveProductionCycle)
                .ThenInclude(e => e.Recipe)
                    .ThenInclude(e => e.Stations)
                        .ThenInclude(e => e.ChosenUtility)

            .Include(e => e.ActiveProductionCycle)
                .ThenInclude(e => e.Recipe)
                    .ThenInclude(e => e.Stations)
                        .ThenInclude(e => e.Station)
            .ToList()[0];
}
```

The Method GetOrderedStationsToDrive orders the Stations according to the settings the user chose when creating the ProductionCycle. This is needed for the OPC-UA Team to drive to the right stations inside the Recipe in the right order.

```
4 Verweise
public List<StationWithChosenUtility> GetOrderedStationsToDrive(Plant plant)
{
    var active = GetPlant().ActiveProductionCycle;

    if (active != null)
    {
        return PlantManager.SortStations(active.Recipe.Stations.ToList());

    }
    else return new List<StationWithChosenUtility>();
}
```

The Method GetOrderedStationsToDrive uses a method SortStations to sort the Stations according to their order number. At first it was thought that method was longer so it was

getting it own method declation, but it turned out it was a oneline. That was not changed later because of time reasons.

```
1 Verweis
public static List<StationWithChosenUtility> SortStations(List<StationWithChosenUtility> stationsToOrder)
{
    return stationsToOrder.OrderBy(station => station.OrderNumber).ToList();
}
```

The Method EditPlant is the most sophisticated Method of the PlantManager, because it has to figure out when to stick to the Current Acitve ProductionCylce, but if all productions are finished, it must determine a new ActiveProductionCycle out of the ProductionQueue and store it in the Database. Error Handling had to wait because the core functionality was not given until the very end before the Presentation, and sadly it wasn't added afterwards.

```
9 Verweise
public void EditPlant(Plant editedPlant)
{

    Plant plant = Plants.Find(editedPlant.Id);
    ProductionCycleManager productioncyclemanager = new ProductionCycleManager();
    if (plant.ActiveProductionCycle is null)
    {
        plant.ActiveProductionCycle = productioncyclemanager.GetProductionCycle((int)plant.ActiveProductionCycleId); //achtung, kann null sein
    }

    if (plant == null)
    {
        // ERROR HANDLING HERE
    }
    else
    {
                    // if all Products are Finished, change ActiveProductionCycle to oldest ProductionCycle from ProductionQueue:
        if (plant.ActiveProductionCycle.NumberOfProducts == editedPlant.ActiveProductionCycle.FinishedProducts)
        {
            plant.ActiveProductionCycle.FinishedProducts = editedPlant.ActiveProductionCycle.FinishedProducts;
            SaveChanges();
            ProductionCycleManager context = new ProductionCycleManager();
            context.SaveChanges();

            plant.ActiveProductionCycleId = context.GetProductionQueue()[0].Id;
        } // if not all Products are Finished, only change FinishedProducts of ActiveProductionCycle
        else if (plant.ActiveProductionCycle.NumberOfProducts > editedPlant.ActiveProductionCycle.FinishedProducts)
        {
                        plant.ActiveProductionCycle.FinishedProducts = editedPlant.ActiveProductionCycle.FinishedProducts;
        }
        plant.ActualPlantState = editedPlant.ActualPlantState;
        plant.requestedPlantState = editedPlant.requestedPlantState;

        SaveChanges();
    }
}
```

Now come the Methods of the SensorData Class.

The Method EditSensorData changed the state of a single chosen SensorData in the Database.

```
#region Methods for SensorData
2 Verweise
public void EditSensorData(SensorData changedSensorData)
{
    foreach (SensorData sensordata in SensorDatas.ToList())
    {
        SensorDatas.Find(sensordata.sensorName).sensorState = false;
        SaveChanges();
    }
    SensorDatas.Find(changedSensorData.sensorName).sensorState = true;
    SaveChanges();

}
```

The Method GetSensorData gets only the one SensorData that is currently switched on with a "1". This method was chosen in consulation of the OPC-UA Team instead of returning all SensorDatas of the Database. The Method GetSensorDataAsync does the same job only as an async Task.

```
// GetSensorData() returns only one SensorData-Object bc only one is set to "true" every time
0 Verweise
public SensorData GetSensorData()
{
    List<SensorData> trueSensorDatas = SensorDatas.Where(sd => sd.sensorState == true).ToList();
    if (trueSensorDatas.Count == 0) return null;
    else return trueSensorDatas[0];

}
```

3.4.5.1 Model Plant

The Model Class "Plant" holds the following Properties:

```
#region Properties
3 Verweise
public int Id { get; private set; }
4 Verweise
public int? ActiveProductionCycleId { get; set; }
39 Verweise
public ProductionCycle? ActiveProductionCycle { get; set; } = null!;
11 Verweise
public PlantState ActualPlantState { get; set; }
18 Verweise
public PlantState requestedPlantState { get; set; }
```

The Plant Class only has a simple Constructor without any arguments and no Methods for it to keep things simple and all Managing-Functionality inside the PlantManager-Class.

3.4.5.2 Model SensorData

The Model Class SensorData holds only sensorName and sensorState as Properties, with sensorName defined as its key value. For the Constructor a key value for sensorName and a predefined sensorState is needed, but the Constructor itself was only needed once for creating all the SensorData-Database-Entries in the beginning.

```
18 Verweise
public class SensorData
{
    #region Properties
    6 Verweise
    public string sensorName { get;  private set; }
    5 Verweise
    public bool sensorState { get;  set; }
    #endregion

    #region Constructor
    2 Verweise
    public SensorData(string sensorName, bool sensorState)
    {
        this.sensorName = sensorName;
        this.sensorState = sensorState;
    }
    #endregion
}
```

3.4.6 SUBSYSTEM2 MODULE 4: User Manager

The User Manager is responsible for the usermanagement. Therefore a table of the model User is in the Database, with the code `public DbSet<User>? Users { get; set; }`. Also a it is important that the User Manager knows the active User therefore `public User activeUser { get; private set; }`.

The Database settings are set in the region “Setting DbContext”, see Figure 36.

```
9 Verweise
public class UserManager : DbContext
{
    #region Properties
    7 Verweise
    public DbSet<User>? Users { get; set; }
    5 Verweise
    public User activeUser { get; private set; }
    #endregion

    #region Constructor
    1 Verweis
    public UserManager()
    {

    }
    #endregion

    #region Settings DbContext
    0 Verweise
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        var conn :string = $"Data Source=" + AbsoluteDbPath.PathGeneration();
        optionsBuilder.UseSqlite(conn);

        base.OnConfiguring(optionsBuilder);
    }

    0 Verweise
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        DbCreationContext.setAllModelConnections(modelBuilder);


        base.OnModelCreating(modelBuilder);
    }
    #endregion
```

Figure 36 Source-Code User Manager except Methods

All Methods are implemented with a simple errorhandling, this means try-catch and in the case of an error, the error gets logged.

The Method GetUser returns the Requested User with the standard method SingleOrDefault(), see Figure 37.

```
1 Verweis
public User GetUser(int id)
{
    try
    {
        return Users.SingleOrDefault(x :User => x.Id == id);
    }
    catch (Exception e)
    {
        Error error = new Error( description: "Unable to fetch User from Database!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
        return null;
    }
}
```

Figure 37 Source-Code User Manager Method GetUser

The Method GetUsers returns all Users from Database with the command ToList( ), see Figure 38.

```
3 Verweise
public List<User> GetUsers()
{
    try
    {
        return Users.ToList<User>();
    }
    catch (Exception e)
    {
        Error error = new Error(description: "Unable to fetch Users from Database!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
        return null;
    }
}
```

Figure 38 Source-Code User Manager Method GetUsers

The Method AddUser, see Figure 39, adds an non existing User to the Database, for this a foreach and if-elseif combination is implemented. The User gets added to the DbSet<User> Users with the standard method Add from DbSet< > and with the command SaveChanges() the added user gets saved to the database. This method can log two different errors.

```
1 Verweis
public void AddUser(User user)
{
    try
    {
        bool isNameEqual = false;

        foreach (var checkUser in Users)
        {
            if (checkUser.Prename == user.Prename &&
                checkUser.Surname == user.Surname) isNameEqual = true; break;
        }

        if (Users.Contains(user) == false && isNameEqual == false)
        {
            Users.Add(user);
            SaveChanges();
        }
        else
        {
            Error error = new Error(description: "User already exists!",
                ErrorType.DATABASE, ErrorSeverity.CRITICAL,
                DateTime.Now);
            ErrorManager.AddError(error);
        }
    }
    catch (Exception e)
    {
        Error error = new Error(description: "Unable to save User to Database!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
    }
}
```

Figure 39 Source-Code User Manager Method AddUser

The Method EditUser, see Figure 40, edits an already existing user in the database, therefore the properties of the existing user must be assigned new values.

```
3 Verweise
public void EditUser(User oldUser, User newUser)
{
    try
    {
        User userToEdit = GetUser(oldUser.Id);
        userToEdit.Prename = newUser.Prename;
        userToEdit.Surname = newUser.Surname;
        userToEdit.ChangePassword(oldUser.Password, newUser.Password);
        userToEdit.Role = newUser.Role;
        SaveChanges();
    }
    catch (Exception e)
    {
        Error error = new Error(description: "Unable to edit User!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
    }
}
```

Figure 40 Source-Code User Manager Method EditUser

The Method DeleteUser, see Figure 41, removes an already existing user from the database. Therefore the user must get fetched from the database with the standard method SingleOrDefault() from DbSet< > and gets then removed from the DbSet< > with the standard method Remove(). The change to database gets then saved with the command SaveChanges().

```
1 Verweis
public void DeleteUser(int id)
{
    try
    {
        User user = Users.SingleOrDefault(x :User => x.Id == id);
        Users.Remove(user);
        SaveChanges();
    }
    catch (Exception e)
    {
        Error error = new Error(description: "Unable to remove User!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
    }
}
```

Figure 41 Source-Code User Manager Method DeleteUser

The Method SetActiveUser, see Figure 42, sets the property activeUser.

```
2 Verweise
public void SetActiveUser(User user)
{
    activeUser=user;
}
#endregion
```

Figure 42 Source-Code User Manager Method SetActiveUser

3.4.6.1 Model User

The model User describes a user with the following properties Id, Prename, Surname, Password and Role.

The Method ChangePassword( ) changes the password and the Method ToString() returns the username as a string.

```
40 Verweise
public class User
{
    #region Properties
    6 Verweise
    public int Id { get; private set; }
    10 Verweise
    public string Prename { get; set; }
    10 Verweise
    public string Surname { get; set; }


    9 Verweise
    public string Password { get; private set; }
    5 Verweise
    public UserRole Role { get; set; }

    #endregion

    #region Constructor
    2 Verweise
    public User(string prename, string surname, string password, UserRole role)
    {
        this.Prename = prename;
        this.Surname = surname;
        this.Password = password;
        this.Role = role;
    }
    #endregion

    #region Methods
   3 Verweise
   public void ChangePassword(string oldPassword,string newPassword)
    {
        if (Password == oldPassword)
        {
            Password=newPassword;
        }
        else
        {
            throw new AccessViolationException(message: "Ihr altes Passwort ist falsch eingeben worden.");
        }

    }

    1 Verweis
    public override string ToString()
    {
        return $"{Prename}.{Surname}";
    }

    #endregion
}
```

Figure 43 Source-Code Model User

3.4.7 SUBSYSTEM2 MODULE 5: Log Manager

The Log Manager is responsible for the logmanagement. Therefor, a table of the model Log is in the Database, with the code `private DbSet<Log>? Logs { get; set; }`.

The Database settings are set in the region “Setting DbContext”, see Figure 44.

```
7 Verweise
public class LogManager : DbContext
{
    #region Properties
    3 Verweise
    private DbSet<Log>? Logs { get; set; }
    #endregion

    #region Constructor
    2 Verweise
    public LogManager()
    {

    }
    #endregion

    #region Settings DbContext
    0 Verweise
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        var conn :string = $"Data Source=" + AbsoluteDbPath.PathGeneration();
        optionsBuilder.UseSqlite(conn);

        base.OnConfiguring(optionsBuilder);
    }

    0 Verweise
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        DbCreationContext.setAllModelConnections(modelBuilder);


        base.OnModelCreating(modelBuilder);
    }
    #endregion
```

Figure 44 Source-Code Log Manager except Methods

The Method FilterLog, see Figure 45, returns Logs filtered in a specified timeframe in a descending order, for this standard methods from DbSet< > where used with lambda expressions.

```
#region Methods
0 Verweise
public List<Log> FilterLog(DateTime fromDateTime, DateTime tilDateTime)
{
    List<Log> logs = Logs.OrderByDescending(x :Log => x.Starttime).ToList();
    List<Log> filteredLogs = logs.FindAll(i :Log => i.Starttime >= fromDateTime
                                                 && i.Starttime <= tilDateTime);
    return filteredLogs.OrderByDescending(x :Log => x.Starttime).ToList();
}
```

Figure 45 Source-Code Log Manager Method FilterLog

The Method AddLog, see Figure 46, adds a Log to Logs with the standard method Add() from DbSet < > and the command SaveChanges() saves the changes in the database.

```
0 Verweise
public void AddLog(Log log)
{
    Logs.Add(log);
    SaveChanges();
}
```

Figure 46 Source-Code Log Manager Method AddLog

The Method GetAllLogs, see Figure 47, returns all logs with the standard methods of DbSet< >.

```
1 Verweis
public List<Log> GetAllLogs()
{
    return Logs // DbSet<Log>?
        .Include( navigationPropertyPath: e :Log => e.ProductionCycle) // IIncludableQueryable<Log,ProductionCycle?>
        .Include( navigationPropertyPath: e :Log => e.Error) // IIncludableQueryable<Log,Error?>
        .ToList(); // List<Log>

}
```

Figure 47 Source-Code Log Manager Method GetAllLogs

The static Method CreateLog, see Figure 48, creates a Log. To get the activeproductioncycle a plantmanger is needed. The log gets created depending on if there is an active productioncycle.

```
1 Verweis
public static Log CreateLog(string description)
{
    PlantManager plantmanager = new PlantManager();

    Plant plant = plantmanager.GetPlant();
    ProductionCycle activeProductionCycle = plant.ActiveProductionCycle;


    if (activeProductionCycle != null)
    {
        return new Log(activeProductionCycle.Id,description, DateTime.Now, plant.ActualPlantState);
    } else
    {
        return new Log(description, DateTime.Now, plant.ActualPlantState);
    }

}
#endregion
```

Figure 48 Source-Code Log Manager Method CreateLog

3.4.7.1 Model Log

The model Log describes a log with the properties Id, ProductionCycleId, ProductionCycle, Error, Description, Starttime and Plantstate.

The model Log has a one to many relationship with ProductionCycle therefore the Properties ProductionCycleId and ProductionCycle is needed and a one to one relationship to Error.

The proper connection for these relationships gets set in the class DbCreationContext, see 3.4.10, for further details look up the Microsoft EF-Core documentation.

Figure 49 is a screenshot of the source-code from the model Log.

```
25 Verweise
public class Log
{
    #region Properties
    3 Verweise
    public int Id { get;private set; }
    3 Verweise
    public int? ProductionCycleId { get; set; } // 1 to many with ProductionCycle
    5 Verweise
    public ProductionCycle? ProductionCycle { get; set; } = null!;  // 1 to many with ProductionCycle
    3 Verweise
    public Error? Error { get; set; } // 1 to 1 with Error
    4 Verweise
    public string Description { get; private set; }
    7 Verweise
    public DateTime Starttime { get; private set; }
    4 Verweise
    public PlantState PlantState { get; private set; }
    #endregion

    #region Constructor
    1 Verweis
    public Log( string description, DateTime starttime, PlantState plantState) // Constructor for ErrorManager
    {


        Description = description;
        Starttime = starttime;
        PlantState = plantState;
    }

    1 Verweis
    public Log(int productionCycleId, string description, DateTime starttime, PlantState plantState) // Constructor for PlantManager
    {

        ProductionCycleId = productionCycleId;
        Description = description;
        Starttime = starttime;
        PlantState = plantState;
    }

    #endregion

    #region Methods

    #endregion

}
```

Figure 49 Source-Code Model Log

3.4.8 SUBSYSTEM2 MODULE 6: Error Manager

The Error Manager is responsible for the errormanagement. Therefore, a table of the model Error is in the Database, with the code `private DbSet<Error>? Errors { get; set; }`.

The Database settings are set in the region “Settings DbContext”, see Figure 50.

```
25 Verweise
public class ErrorManager : DbContext
{
    #region Properties
    3 Verweise
    private DbSet<Error> Errors { get; set; }
    #endregion

    #region Constructor
    2 Verweise
    public ErrorManager()
    {

    }
    #endregion

    #region Settings DbContext
    0 Verweise
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        var conn :string = $"Data Source=" + AbsoluteDbPath.PathGeneration();
        optionsBuilder.UseSqlite(conn);

        base.OnConfiguring(optionsBuilder);
    }

    0 Verweise
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        DbCreationContext.setAllModelConnections(modelBuilder);



        base.OnModelCreating(modelBuilder);
    }
    #endregion
```

Figure 50 Source-Code Error Manager except Methods

The Method FilterErrors, see Figure 51, is not implemented yet, it just returns a list of Errors without the attached Log.

```
#region Methods
0 Verweise
public List<Error> FilterErrors()
{
    return Errors.ToList<Error>(); //dummy
}
```

Figure 51 Source-Code Error Manager Method FilterErrors

The static Method AddError, see Figure 52, adds an Error to the database, therefore the same DbContext must be initialised and with SaveChanges() the error gets saved into the database.

```
20 Verweise
public static void AddError(Error error)
{
    ErrorManager errormanager = new ErrorManager();

    error.Log = LogManager.CreateLog(description: "Error in " + error.ErrorType.ToString());

    errormanager.Errors.Add(error);
    errormanager.SaveChanges();
}
```

Figure 52 Source-Code Error Manager Method AddError

The Method GetAllErrors(), see Figure 53, returns all Errors from the database with the attached Logs.

```
1 Verweis
public List<Error> GetAllErrors()
{
    return Errors // DbSet<Error>
        .Include(navigationPropertyPath: e :Error => e.Log) // IIncludableQueryable<Error,Log?>
        .ToList(); // List<Error>
}
#endregion
```

Figure 53 Source-Code Error Manager Method GetAllErrors

3.4.8.1 Model Error

The model Error describes an error with the properties Id, LogId, Log, Description, ErrorType, Severity, Time OfOccurence and TimeWhenFixed.

The model Error has a one to one relationship with Log therefore the Properties LogId and Log is needed.

The proper connection for this relationship gets set in the class DbCreationContext, see 3.4.10, for further details look up the Microsoft EF-Core documentation.

Figure 54 is a screenshot of the source-code from the model Error.

```
53 Verweise
public class Error
{
    #region Properties
    3 Verweise
    public int Id { get; private set; } //hier ID statt ErrorID
    2 Verweise
    public int? LogId { get; set; } // 1 to 1 with Log
    7 Verweise
    public Log? Log { get; set; } = null!; // 1 to 1 with Log
    2 Verweise
    public string Description { get; private set; }
    3 Verweise
    public ErrorType ErrorType { get; private set; }
    2 Verweise
    public ErrorSeverity Severity { get; private set; }
    2 Verweise
    public DateTime TimeOfOccurence { get; private set; }
    1 Verweis
    public DateTime TimeWhenFixed { get; private set; }
    #endregion

    #region Constructor
    20 Verweise
    public Error(string description, ErrorType errorType,
        ErrorSeverity severity, DateTime timeOfOccurence)
    {
        Description = description;
        ErrorType = errorType;
        Severity = severity;
        TimeOfOccurence = timeOfOccurence;
    }
    #endregion

    #region Methods

    #endregion
}
```

Figure 54 Source-Code Model Error

3.4.9 SUBSYSTEM2 MODULE 7: OEE Manager

3.4.9.1 Model OEE

The OEE Dashboard constitutes a comprehensive interface providing users with a clear view of our plant's production process performance and efficiency. Aligned with our adoption of the Model-View-Presenter pattern, this documentation is divided into distinct sections dedicated to the Model, View, and Presenter components, offering comprehensive insights into each facet of the dashboard's architecture.

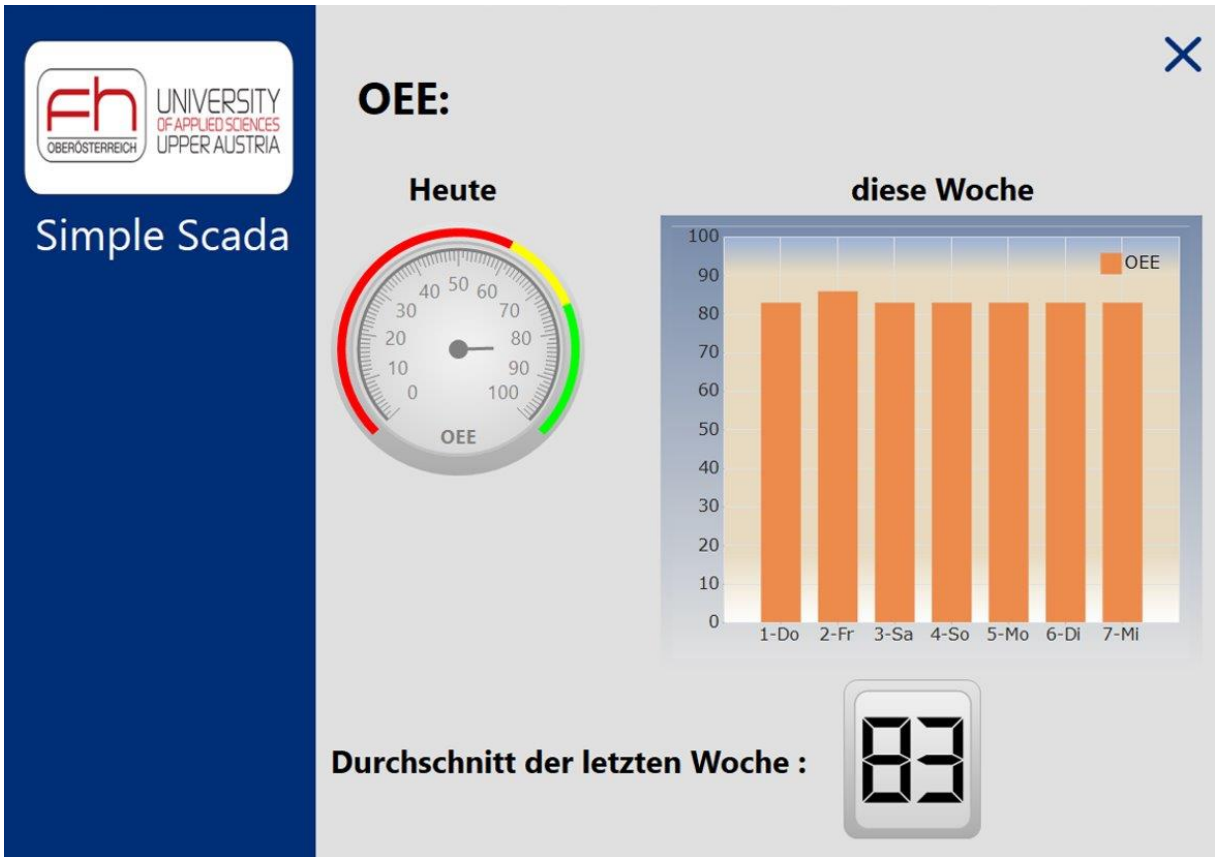


Figure 55: OEEView

OEE window consists of three important elements:

- Radial gauge (today): shows today's OEE on a scale from 0 to 100
- Chart(this week): Indicates OEE from the last seven days on a scale of 0 to 100
- Digital gauge (Average of last week): is displayed the weekly OEE-Average

```
private void CalaculateOEEbyDate(IGrouping<DateOnly, Log> group)
{
    var availability = 0d;
    var performance = 0d;
    var ttNumberProducts = 0;
    var ttFinishedProducts = 0;
    var ttMinutesSum = 0.0d;
    // estimated time
    var estimatedTimeSum = 0.0d;
    IEnumerable<IGrouping<ProductionCycle?, Log>> LogsByProductionCycle = group.GroupBy(e => e.ProductionCycle);
    // for each production caycle get estimated time
    foreach (IGrouping<ProductionCycle?, Log> prodCycleGroup in LogsByProductionCycle)...

    // calculate time spend per plant production
    DateTime startTime = DateTime.MinValue;
    DateTime endTime = DateTime.MinValue;

    foreach (IGrouping<ProductionCycle?, Log> prodCycleGroup in LogsByProductionCycle)
    {
        foreach (var log in prodCycleGroup)
        {
            if(PlantState.RUNNING == log.PlantState)
            {
                startTime = log.Starttime;
            }
            else if (PlantState.FINISHED == log.PlantState)
            {
                endTime = log.Starttime;
                ttNumberProducts += log.ProductionCycle.NumberOfProducts;
                ttFinishedProducts += log.ProductionCycle.FinishedProducts;
            }
        }
        // calculate diff time
        if (startTime.CompareTo(DateTime.MinValue) > 0 && endTime.CompareTo(DateTime.MinValue) > 0)
        {
            ttMinutesSum += endTime.Subtract(startTime).TotalMinutes;
        }
    }
    if (ttNumberProducts == 0 || ttMinutesSum == 0)...
    availability = (estimatedTimeSum / ttMinutesSum);
    performance = (ttFinishedProducts / ttNumberProducts);
    OEE oee = new OEE();
    oee.Availability = availability;
    oee.Leistungseffizienz = performance;
    oee.QualityRate = 1;
    oee.Date = endTime;
    _OEEManager.Add(oee);
    _OEEManager.SaveChanges();
```

Figure 56: Source-Code Model OEE

This method, see Figure 48, calculates Overall Equipment Effectiveness (OEE) metrics for a given date based on a collection of logs grouped by that date. It involves two main steps:

- It calculates the estimated time for each production cycle associated with the logs for the given date.
- It iterates through the logs, tracking the start and end times of plant operation cycles, and calculates the total time spent, total number of products, and total finished products.
- It then computes the availability and performance metrics based on the calculated data and creates an OEE object. This object is saved to Database using the OEEManager.

```
// clean oee table to store last 7 days
_OEEManager.Clean();
// calculate last 7 days OEEs
// logs of last 7 days
IEnumerable<IGrouping<DateOnly, Log>> LogsByDate = _LogManager.GetLogsfromWeek().GroupBy(e => DateO

//go over each day
foreach (IGrouping<DateOnly, Log> group in LogsByDate)...

// latest quality rate gauge
OEE oeefromDB = _OEEManager.GetLatestOEE();
if (oeefromDB == null)
{
    return;
}
var oeeValue = _OEEManager.CalculateOEE();
_OEEView.radialGauge1.Value = oeeValue;

// last week quality rate bar chart
var oeeSum = 0;
List<OEE> oeeList = _OEEManager.GetOEEfromWeek();
ChartSeries series1 = new ChartSeries();
series1.Name = "OEE";
series1.Type = ChartSeriesType.Column;
series1.Text = series1.Name;
int i = 7;
foreach (OEE item in oeeList)
{
    var calculatedOEE = _OEEManager.CalculateOEE(item);
    series1.Points.Add(i + "-" + item.Date.ToString("ddd"), calculatedOEE);
    oeeSum += calculatedOEE;
    i--;
}
_OEEView.chartControl1.Series.Clear();
_OEEView.chartControl1.Series.Add(series1);

// last week avg digital gauge
_OEEView.digitalGauge1.Value = String.Format("{0}", oeeSum / 7);
```

Figure 57: Source-Code Model OEE

This method, see Figure 49, is responsible for updating the user interface with OEE-related data and visuals. It performs the following tasks:

- Cleans the OEE table to make space for new data.
- Retrieves logs from the past week and groups them by date.
- Calculates OEE for each day using the CalculateOEEbyDate method.
- Retrieves the latest OEE value and updates a radial gauge with it.
- Prepares and updates a bar chart with daily OEE values from the past week.
- Calculates and updates a digital gauge with the average OEE value for the past week.
- Implements error handling to catch and handle any exceptions that might occur during the process.

```
2 Verweise
public int CalculateOEE(OEE oee)
{
    return Convert.ToInt32(oee.QualityRate * oee.Availability * oee.Leistungseffizienz * 100);
}

3 Verweise
public OEE GetLatestOEE()
{
    OEE lastestOee = OEEs.OrderByDescending(o => o.Date).FirstOrDefault();


    return lastestOee;
}

2 Verweise
public List<OEE> GetOEEfromWeek()
{
    try
    {
        return OEEs.OrderByDescending(o => o.Date).Take<OEE>(7).ToList<OEE>();
    }
    catch (Exception e)
    {
        Error error = new Error("Unable to fetch OEE from Database!",
            ErrorType.DATABASE, ErrorSeverity.CRITICAL,
            DateTime.Now);
        ErrorManager.AddError(error);
        return null;
    }


}

1 Verweis
public void Clean()
{
    OEEs.RemoveRange(OEEs.ToArray<OEE>());
    this.SaveChanges();
}
```

Figure 58: Source-Code Model OEE

OEE Manager is a class, see Figure 50, that manages operations related to Overall Equipment Effectiveness (OEE) calculations and data storage to database. it has the following methods:

- CalculateOEE(OEE oee): This method calculates the OEE value based on the provided OEE object. It multiplies the Quality Rate, Availability, and Performance efficiency metrics, converts the result to an integer, and returns the calculated OEE as a percentage.
- GetLatestOEE(): This method retrieves the latest OEE data from a collection of OEEs stored in database. It orders the collection by date in descending order and retrieves the first (latest) entry. It returns an OEE object containing the latest OEE data.
- GetOEEfromWeek(): This method attempts to retrieve a list of OEE data from the past week. It orders the collection of OEEs by date in descending order and takes the first 7 entries (representing the past 7 days). If successful, it returns a list of OEE objects from the past week; otherwise, it handles the exception, logs an error, and returns null.

- Clean(): This method is responsible for cleaning the stored OEE data. It removes all existing OEE objects from the collection and then calls SaveChanges() to persist the changes in database.

3.4.10 SUBSYSTEM2 MODULE 8: DbCreationContext

In order to apply migrations to the database (migrations are changes in the structure of the tables of the database, for example adding a possible "column" in a table, which means adding a property to a DbContext), all DbContext had to be combined in a single file DbCreationContext, which holds all DbSets with all its KeyValues and all relationships between them defined. Because with all this in one file, it is easy to apply migrations with only one line of code calling only one source-code-file in the Developer PowerShell.

All used DbSets in the DbCreationContext are here as follows:

```
14 Verweise
public class DbCreationContext : DbContext
{
    #region Members
    0 Verweise
    private DbSet<Recipe>? Recipes { get; set; }
    0 Verweise
    private DbSet<Station>? Stations { get; set; }
    0 Verweise
    private DbSet<Utility>? Utilities { get; set; }
    0 Verweise
    private DbSet<Log>? Logs { get; set; }
    0 Verweise
    private DbSet<ProductionCycle>? ProductionCycles { get; set; }
    0 Verweise
    private DbSet<Error>? Errors { get; set; }
    0 Verweise
    private DbSet<OEE>? OEEs { get; set; }
    0 Verweise
    private DbSet<User>? Users { get; set; }
    0 Verweise
    private DbSet<SensorData>? SensorDatas { get; set; }
    0 Verweise
    private DbSet<StationWithChosenUtility> RecipesStations { get; set; }
    0 Verweise
    private DbSet<Plant>? Plants { get; set; }
```

All KeyValues for all DbSets had to be defined also. With ToTable(), you can also set a custom Table-name and overwrite the naming-structure that EntityFramework uses.

```
7 Verweise
public static void setAllModelConnections(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Recipe>().HasKey(b => b.Id);
    modelBuilder.Entity<Station>().HasKey(b => b.Id);
    modelBuilder.Entity<Utility>().HasKey(b => b.Id);
    modelBuilder.Entity<Log>().HasKey(b => b.Id);
    modelBuilder.Entity<ProductionCycle>().HasKey(b => b.Id);
    modelBuilder.Entity<Error>().HasKey(b => b.Id);
    modelBuilder.Entity<User>().HasKey(b => b.Id);
    modelBuilder.Entity<SensorData>().HasKey(b => b.sensorName);
    modelBuilder.Entity<StationWithChosenUtility>().HasKey(b => b.Id);
    modelBuilder.Entity<Plant>().HasKey(b => b.Id);


    modelBuilder.Entity<Recipe>().ToTable("Recipes");
    modelBuilder.Entity<Station>().ToTable("Stations");
    modelBuilder.Entity<Utility>().ToTable("Utilities");
    modelBuilder.Entity<Log>().ToTable("Logs");
    modelBuilder.Entity<ProductionCycle>().ToTable("ProductionCycles");
    modelBuilder.Entity<Error>().ToTable("Errors");
    modelBuilder.Entity<OEE>().ToTable("OEEs");
    modelBuilder.Entity<User>().ToTable("Users");
    modelBuilder.Entity<SensorData>().ToTable("SensorDatas");
    modelBuilder.Entity<StationWithChosenUtility>().ToTable("RecipesStations");
    modelBuilder.Entity<Plant>().ToTable("Plants");
```

Also all Relationships between all DbSets had to be configured and could be changed in only that single file.

```
modelBuilder.Entity<Station>()
    .HasMany(e => e.Utilities)
    .WithMany(e => e.Stations);

modelBuilder.Entity<ProductionCycle>()
    .HasOne(e => e.Recipe)
    .WithMany(e => e.ProductionCycles)
    .HasForeignKey(e => e.RecipeId);

modelBuilder.Entity<Log>()
    .HasOne(e => e.ProductionCycle)
    .WithMany(e => e.Logs)
    .HasForeignKey(e => e.ProductionCycleId)
    .IsRequired(false);

modelBuilder.Entity<Log>()
    .HasOne(e => e.Error)
    .WithOne(e => e.Log)
    .HasForeignKey<Error>(e => e.LogId)
    .IsRequired(false);

modelBuilder.Entity<StationWithChosenUtility>()
    .HasOne(e => e.Station)
    .WithMany(e => e.ChosenUtilities)
    .HasForeignKey(e => e.StationId);

modelBuilder.Entity<StationWithChosenUtility>()
    .HasOne(e => e.ChosenUtility)
    .WithMany(e => e.ChosenUtilites)
    .HasForeignKey(e => e.ChosenUtilityId);

modelBuilder.Entity<StationWithChosenUtility>()
.HasOne(e => e.Recipe)
.WithMany(e => e.Stations)
.HasForeignKey(rs => rs.RecipeId);

modelBuilder.Entity<StationWithChosenUtility>()
HasOne(e => e.Station)
WithMany(e => e.ChosenUtilities)
HasForeignKey(e => e.StationId);

modelBuilder.Entity<Plant>()
HasOne(e => e.ActiveProductionCycle)
WithOne(e => e.Plant)
HasForeignKey<Plant>(e => e.ActiveProductionCycleId);
```

3.4.11 SUBSYSTEM2 MODULE 9: Linux Database

For the production-phase there is an azure-sql-edge server available on a raspberry pi. For using this database the IP-Address in the software has to be changed the rest is already implemented, see Figure 59 and Figure 60

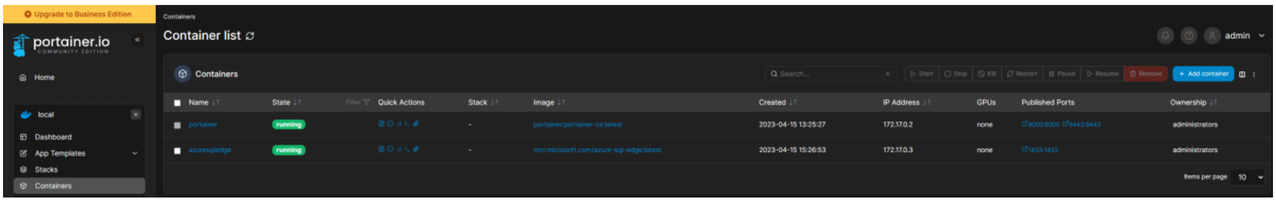


Figure 59 Screenshot portainer.io

```
root@pi64lite:~# docker ps
CONTAINER ID   IMAGE                                  COMMAND                  CREATED       STATUS       PORTS                                                                              NAMES
d6bed67ce35b   mcr.microsoft.com/azure-sql-edge:latest "/opt/mssql/bin/perm…"   6 weeks ago   Up 3 weeks   1401/tcp, 0.0.0.0:1433->1433/tcp, :::1433->1433/tcp                                azuresqledge
f44f46dd0248   portainer/portainer-ce:latest           "/portainer"             6 weeks ago   Up 3 weeks   0.0.0.0:8000->8000/tcp, :::8000->8000/tcp, 0.0.0.0:9443->9443/tcp, :::9443->9443/tcp, 9000/tcp   portainer
```

Figure 60 Screenshot RaspberryPi

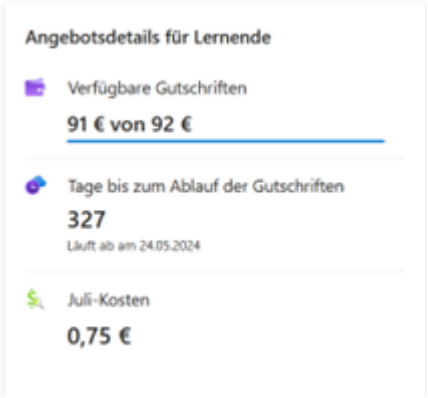
3.4.12 SUBSYSTEM2 MODULE 10: Azure Database

In the ProductionPhase, a cloud-based Azure Database was finally used to show one central Plant and how it behaves and that one Plant could be accessed and controlled, but also only viewed via as many end devices as whished.

For that, a Student Account with 100$ free Account volume for a Student of the FH Wels was created, and a Database Server with a Database hosted on it was rented.

Name	Typ
simplescada	SQL-Datenbank
simplescadaserver	Computer mit SQL Server
simplescada	Ressourcengruppe

The costs are about 4$ a month, so if we leave it running, our programm could theoratically run with the cloud database for long over a year for following developer-teams to inspect. But our 100$-Credit from our FH-Account is only valid for 1 year.



The simple and convenient switch from a local Database for each developer to the cloud-based solution was amazing, it was basically a one-liner.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    var conn = $"Data Source=" + AbsoluteDbPath.PathGeneration();
    optionsBuilder.UseSqlite(conn);

    base.OnConfiguring(optionsBuilder);
}
```

The only thing that had to be changed in the "OnCofiguring" Method was to switch from "UseSqlite" for the local databases to "UseSqlServer" with it's according connection-string for the cloud-based Azure-Database on the SqlServer also hosted on Azure.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    var conn = $"Server=tcp:simplescadaserver.database.windows.net,1433;Initial Catalog=simplescada;" +
        $"Persist Security Info=False;User ID=simplescada;Password=[redacted];" +
        $"MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;";
    optionsBuilder.UseSqlServer(conn);

    base.OnConfiguring(optionsBuilder);
}
```

3.5 SUBSYSTEM 3– Desktop Application

3.5.1 SUBSYSTEM 3 ARCHITECTURE

Generally, the Desktop PC App has been implemented according to the MVP Pattern. The abbreviation stands for Model View Presenter. As the Application is a Multi Window Application also multiple Presenters, Views and Models are necessary.

For each window a Presenter as well as View has been implemented. As Models the Management classes as described in the chapters 3.4 SUBSYSTEM 2 – SQL Database have been used.

Due to the multi window approach for the Application the Presenters have been implemented according to a hierarchical structure. The architecture of the Desktop Application can be seen in the Figure 61 below.

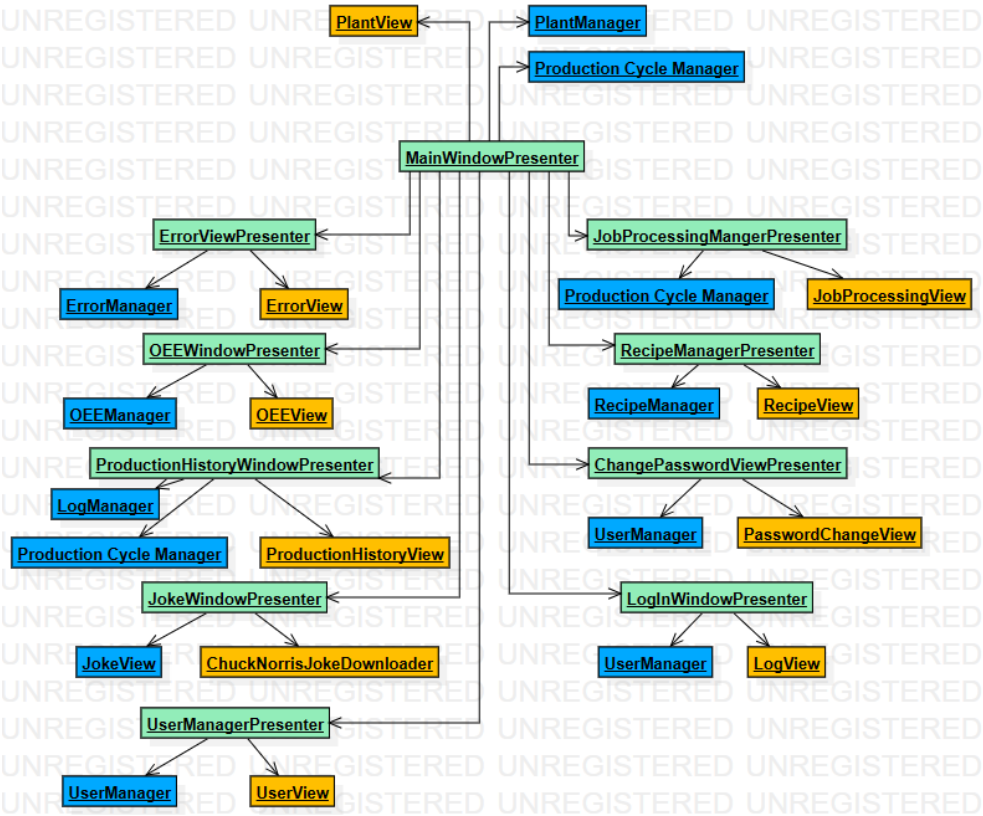


Figure 61 Architecture Overview of the Desktop Application

All Presenters have the colour light green. The Managers have been coloured in the colour light blue and the Views have been coloured in the colour orange.

As in the architecture diagram mentioned there are presenters which refer to the same model. The architecture has been implemented so that all models of the same type refer to the same instance of an object. This is possible because the Models are reference datatypes.

The Desktop Application consist of the following Views and the according Presenters and Models:

- Plant View (Main View)

- Error View
- Job processing View
- Log View
- OEE View
- Recipe View
- User View
- Production history View
- Password Change View
- Joke View

The Views with the according Presenters and the references to the individual Models are described in the Modules below.

3.5.2 SUBSYSTEM 3 INTERFACES

- .NET 6.0
- Operation System: Windows
- Syncfusion Chart Windows 21.2.10
- Syncfusion Gauge Windows 21.2.10
- Simple Scada Shared Resources

3.5.3 SUBSYSTEM 3 MODULE 1: Plant Overview

The Plant Overview is responsible to inform the user about the general state of the machine as wells as to manipulate the state of the plant. In addition, the plant overview informs about the active order which is currently produced and where in the plant the active order is located. Another feature on the main view is the production queue which indicates the next orders that are going to be produced on the plant. The last aspect of the plant overview is the navigation thru the software.

The accessibility of the plant View changes according to the logged-on user and his user permissions.

3.5.3.1 Plant Overview View

The Plant Overview consists of the bird's eye view plant, a control panel to operate the plant, an indicator for the active production cycle and the production queue. Furthermore, the user can navigate thru the application via the navigation bar located at the bottom of the view.

The modules of the plant overview are marked and numbered in the figure 62 below.

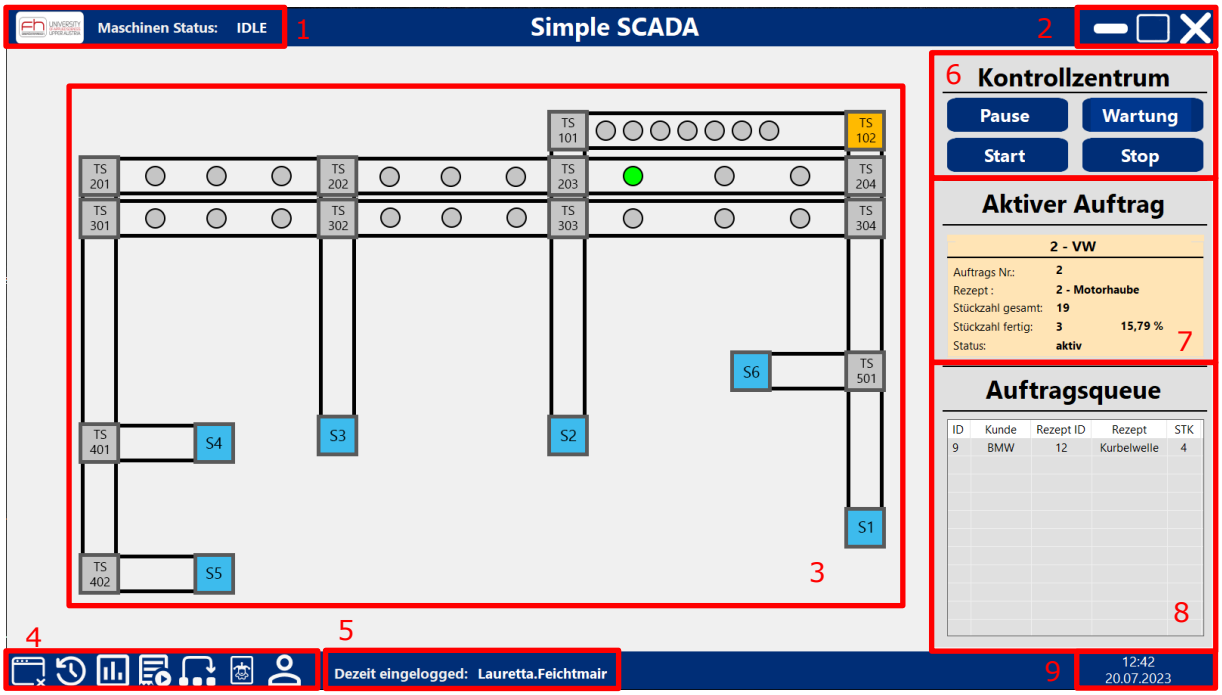


Figure 62 Screenshot of Plant Overview

1. **Plant State:** Show the current State of the plant (possible plant states are RUNNING, DISTURBANCE, MAINTAINENCE, PAUSED, IDLE, STOP and FINISHED)
2. **Resizing Main Window:** main Window can be minimized, maximized and closed with the close button
3. **Bird's Eye View of Plant:** shows the layout of plant as well as the position of the palette on which the current order is produced. The position of the pallet is indicated with green colour on the Convery belts and with yellow at the stations.
4. **Navigation Bar:** With the navigation bar the user can navigate between the windows. The Icons are only clickable if the user has the permission to see and operate the window. The Icons stands for the following windows:
   a.  Opens the Error Overview Window.
   b.  Opens the Production history window.
   c.  Opens the OEE Window.
   d.  Opens the job processing window.
   e.  Opens the Recipe window.
   f.  Opens the joke window.
   g.  Opens a Submenu which is implemented as following: 

i.  Opens the Log In Window.

ii.  Opens the Password Change Window.

iii.  Opens the user editor Window

iv.  Logs out the currently logged in user.

5. **Current User:** Indicator for currently logged on User. If no user is logged in the indicator shows a small line.
6. **Control Panel:** With the control panel the user can manipulate the state of the machine. The buttons are only clickable if a user with the correct permission is logged on. The following buttons/states have been implemented: Start, Stop, Pause and Maintenance.
7. **Active Order:** The indicators show the active order, including the order number, the corresponding recipe, the total amount of pieces, the already produced pieces, the percentage of produced pieces and the status of the order.
8. **Production Queue:** The production order shows the list of the orders which are going to be produced after the active production order has been finished. The production orders in the queue are scheduled in a chronologic order.
9. **Indicator for time and date**

The plant view implements events for each button when clicked. An Overview of the implemented events /EventHandler can be seen in the figure 63.

```
//Events
public event EventHandler RecipeWindowRequired;
public event EventHandler ErrorViewRequired;
public event EventHandler JobProcessingWindowRequired;
public event EventHandler LogInWindowRequired;
public event EventHandler OEEViewRequired;
public event EventHandler PasswordChangeViewRequired;
public event EventHandler PoductionHistoryWindowRequired;
public event EventHandler UserViewRequired;
public event EventHandler JokeViewRequired;
public event EventHandler LogOutUser;
public event EventHandler<PlantView> DateTimeRequested;
public event EventHandler<PlantState> PauseStateRequested;
public event EventHandler<PlantState> StartStateRequested;
public event EventHandler<PlantState> StopStateRequested;
public event EventHandler<PlantState> MaintanceStateRequested;
```

Figure 63 Overview Implemented Events Main Window

Furthermore, the following methods have been implemented in the View:

```
1 reference
public List<Panel> GetPlantControls()...
2 references
private void InitialStateOfView()...
```

Figure 64 Implemented methods Main window

The GetPlantControls method returns the List of Panels which indicate the Sensors, transfer stations and stations. The details can be seen in the source code in the class PlantView.cs

The InitialStateOfView method sets the state of the main view to the initial state which can be seen in the figure 65 below. The buttons which cannot be clicked, if no user is logged on, are disabled in the initial state.

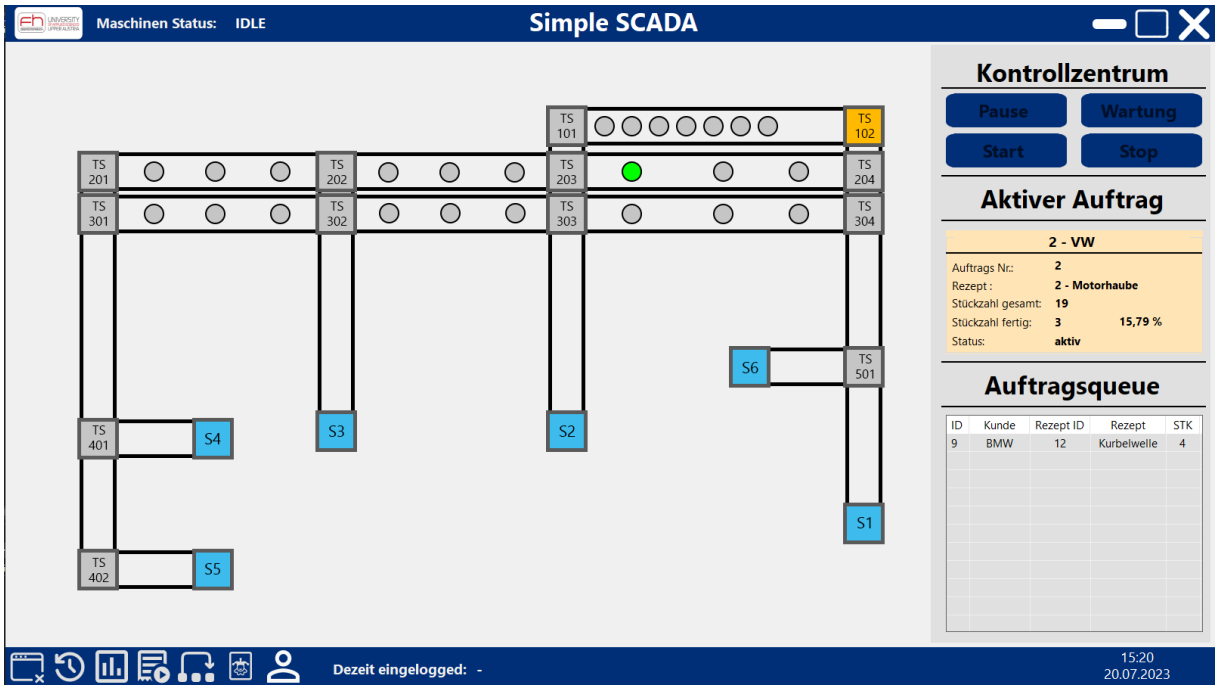


Figure 65 Main Window in the initial state

3.5.3.2 Plant Overview Presenter

The MainWindowPresenter manipulates the data from the models so that the data can be shown correctly on the View.

Therefore, the MainWindowPresenter has the MainView, the models for the MainView and the other Presenters as member variables. The list of all member variables can be seen in the figure 66.

```
#region Membervariables
//Models
private PlantManager _plantManager;
private ProductionCycleManager _productionCycleManager;

//Presenter
private ChangePasswordViewPresenter _changePasswordViewPresenter;
private ProductionHistoryWindowPresenter _historyWindowPresenter;
private ErrorPopUpWindowPresenter _errorPopUpWindowPresenter;
private ErrorViewPresenter _errorViewPresenter;
private LogInWindowPresenter _logInWindowPresenter;
private OEEWindowPresenter _oEEWindowPresenter;
private JobPorcessingManagerPresenter _jobPorcessingManagerPresenter;
private RecipeManagerPresenter _recipeManagerPresenter;
private UserManagerPresenter _userManagerPresenter;
private JokeWindowPresenter _jokeWindowPresenter;

//View ....
private PlantView _plantView;

//Backgroundworker
private BackgroundWorker _backgroundWorkerAktiveProductioncycle;
private BackgroundWorker _backgroundWorkerSensorData;
private BackgroundWorker _backgroundWorkerProductionQueue;

private List<Panel> SensorDataControls;
#endregion
```

Figure 66 Implemented member variables main presenter

The data of the active production cycle, production queue and the sensor data must be updated cyclical. Therefore, the background workers are used. For each cyclic task an individual background worker is used, but the principal described below applies to all three backgroundworkers, so only one is described in detail. The source code to the backgroundworkers can be seen in the class MainWindowPresenter.cs

The process of getting the data in a cyclic time period is described for the production queue.

- Define Backgroundworkers and start the backgroundworkers in the Constructor of the Presenter
- Define DoWork method, which is responsible for the timing.
- Define the ProgressChanged method, which gets the data from the database with an async method and then load the data into the view.

The DoWork method and the ProgressChanged method are documented in the figure 67.

```
private void UpdateProductionQueueList(object sender, DoWorkEventArgs e)
{
    while (true)
    {
        // Do the long-duration work here, and optionally
        // send the update back to the UI thread...
        int p = 0;// set your progress if appropriate
        object param = "something"; // use this to pass any additional parameter back to the UI
        _backgroundWorkerProductionQueue.ReportProgress(p, param);
        Thread.Sleep(millisecondsTimeout: 1000);
    }
}

1 reference
private async void OnProgessChangedProductionQueueList(object sender,ProgressChangedEventArgs e)
{
    List<ProductionCycle> queue = await _productionCycleManager.GetProductionQueueAsync();

    if (queue != null)
    {
        _plantView.lv_Queue.Items.Clear();
        _plantView.lv_Queue.AutoSize = true;

        int i = 0;
        foreach (ProductionCycle cycle in queue)
        {
            ListViewItem context = new ListViewItem(text: cycle.Id.ToString(),i);
            context.SubItems.Add(cycle.Customer);
            context.SubItems.Add(text: cycle.RecipeId.ToString());
            context.SubItems.Add(cycle.Recipe.Name);
            context.SubItems.Add(text: cycle.NumberOfProducts.ToString());

            _plantView.lv_Queue.Items.Add(context);
            i++;
        }
    }

}
```

Figure 67 Implementation of the timing for time based updating

The following methods are implemented for the cyclic updating of the data:

```
1 reference
235 private void UpdateProductionQueueList(object sender, DoWorkEventArgs e)...
247
1 reference
248 private async void OnProgessChangedProductionQueueList(object sender,ProgressChangedEventArgs e)...
272
1 reference
273 private void BackgroundWorker_RunWorkerCompletedProductionQueueList(object sender, RunWorkerCompletedEventArgs e)...
277
1 reference
278 private void UpdateSensorData(object sender, DoWorkEventArgs e)...
290
1 reference
291 private async void OnProgressChangedSensorData(object sender, ProgressChangedEventArgs e)...
340
1 reference
341 private void BackgroundWorker_RunWorkerCompletedSensorData(object sender, RunWorkerCompletedEventArgs e)...
345
1 reference
346 private void BackgroundWorker_RunWorkCompledededQueue(object? sender, RunWorkerCompletedEventArgs e)...
350
1 reference
351 private async void OnProgressActiveProductionCycle(object? sender, ProgressChangedEventArgs e)...
373
1 reference
374 private void UpdateActiveProductionCycle(object? sender, DoWorkEventArgs e)...
```

Figure 68 Overview of methods for time based updating

The other implemented methods of the MainView Presenter are:

- private void DateTime(object sender, PlantView plantView): Sets the current date and time on the instance of the plantView.
- private void ViewSettingsOnUserPermission(object sender,User UserToLogIn): Enables and disables the buttons on the Main View according to the permissions of the user which has logged on the application. The possible permissions are SUPERUSER, OPERATOR and SPECTATOR. The individual permissions for the different user roles can be seen in the sorce code.

- private void RequestPlantState(object sender, PlantState state): sets the requested plant sate in the data base so that the controller changes the status of the plant if possible according to the input of the user in the control panel.

3.5.3.3 Plant Overview Model

For the plant Overview the following models have been used:

- PlantManager: The plant manager is described in 3.4.5 SUBSYSTEM2 MODULE 3: Plant Manager.
- ProductionCycleManager: The production cycle manager is described in 3.4.4 SUBSYSTEM2 MODULE 2: ProductionCycle Manager.

3.5.4 SUBSYSTEM 3 MODULE 2: Error and Log Overview

3.5.4.1 Log View and Production-History View

The LogView was combined with the Production-History View because there was place for both scrollable tables in one window.

At the end it was implemented with a "Refresh" Button because of time reasons and because we didn't want as much background-workers running at the same time to focus the periodic communication with the database for the position of the palettes to save interaction time with the database.

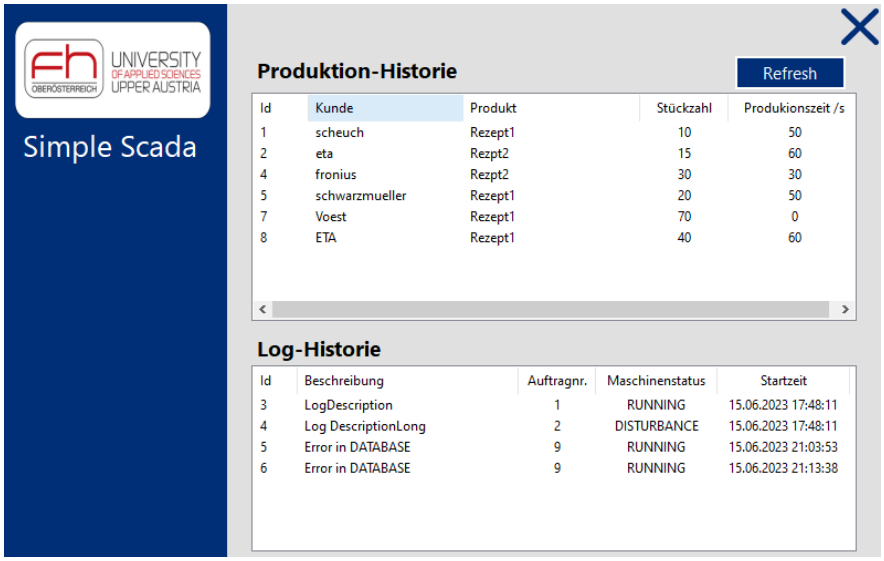


Figure 69 log and production history view

The only EventManager that had to be implemented besides opening and closing the window, was an Event Listener for the Refresh Button. It simply fetches the List of ProductionCycles and Logs from the affected Managers from the Database-Team and gives it back to the View.

```
1 Verweis
private void OnLoadLogsAndFinishedProductsRequested(object? sender, EventArgs? e)
{
    List<ProductionCycle> finishedProductionCycles = _productionCycleManager.GetFinishedProductionCycles();
    List<Log> allLogs = _LogManager.GetAllLogs();

    _productionHistroyView.LoadDataInList(finishedProductionCycles, allLogs);
}
```

Figure 70 Event Listener Refresh Button

Each Object of the two lists gets turned into a "Line" of the Table in the View according to fixed Windows-Forms Coding Descriptions.

```
1 Verweis
public void LoadDataInList(List<ProductionCycle> finishedProductionCycles, List<Log> logs)
{
    // fill ListView of ProductionCycles
    listView_ProductionCycle.Items.Clear();

    int i = 0;
    foreach (var pc in finishedProductionCycles)
    {
        ListViewItem item = new ListViewItem(pc.Id.ToString(), i);
        item.SubItems.Add(pc.Customer);
        item.SubItems.Add(pc.Recipe.Name);
        item.SubItems.Add(pc.FinishedProducts.ToString());
        item.SubItems.Add(pc.EstimatedTime.ToString());

        i++;


        listView_ProductionCycle.Items.Add(item);

    }
    // fill ListView of Logs
    listView_Logs.Items.Clear();
    i = 0;
    foreach (var log in logs)
    {
        ListViewItem item2 = new ListViewItem(log.Id.ToString(), i);
        item2.SubItems.Add(log.Description);
        if (log.ProductionCycle.Id != null)
        {
            item2.SubItems.Add(log.ProductionCycle.Id.ToString());
        }else
        {
            item2.SubItems.Add("-");
        }
        item2.SubItems.Add(log.PlantState.ToString());
        item2.SubItems.Add(log.Starttime.ToString());
        i++;
        listView_Logs.Items.Add(item2);
```

Figure 71 Methods to visualize data

3.5.4.2 Error View

The ErrorView got its own Window because the Error Messages could be read easier. Of course, in the end product no errors appeared at all so no Error Messages are displayed in the ErrorView.

It was also implemented only with an "Refresh"-Button because of time reasons and to save periodic interaction time with the database.

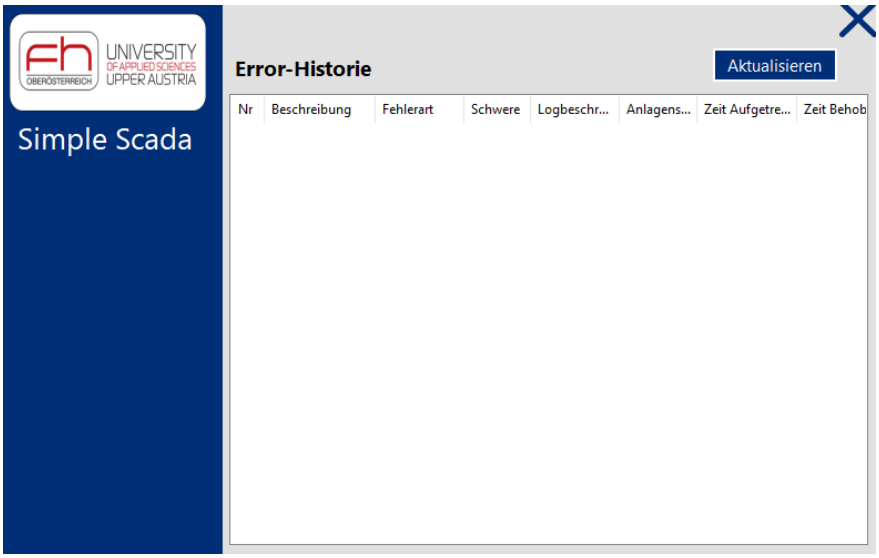


Figure 72 Screenshot error view

The only Method to be added to a EventListener is "OnLoadErrors" that again fetches the List of Errors from the ErrorManager and gives it to the ErrorView, where the Method "LoadDataInList" works very similarly to the Method in the ProductionHistoryView.

```
1 Verweis
private void OnLoadErrors(object? sender, EventArgs? e)
{
    _errorView.LoadDataInList(_errorManager.GetAllErrors());
}
```

Figure 73 Method to load errors

3.5.5 SUBSYSTE 3 MODULE 3: Production Management

The Subsystem Production Management is responsible for the management of the production. The Submodules further divides in the management of Recipes and the management of production orders.

3.5.5.1 Recipe Management

The recipe management allows the user to manipulate the recipe according to his wishes. The user is able to create new recipes, change already created recipes and delete recipes.

3.5.5.1.1 Recipe View

The Recipe View consists of a loading section where already created recipes can be loaded, a setting section where the recipe can be design and a button strip. The modules of the recipe view are marked and numbered in the figure 74 below.

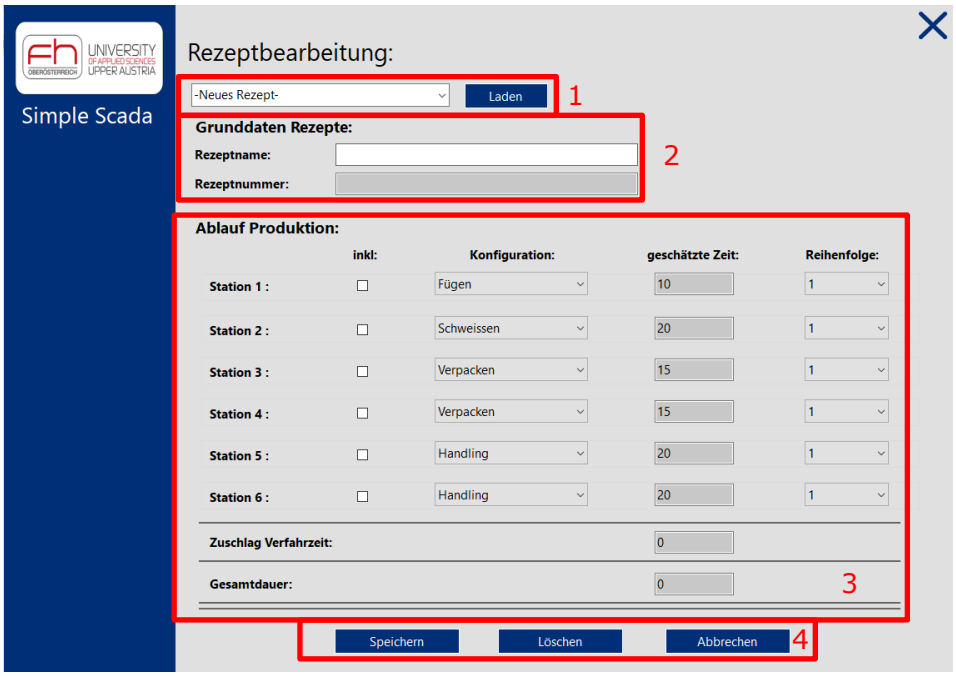


Figure 74 Screenshot of recipe manager

1. **Load Recipe:** The dropdown box implements a list of all already existing recipes. If a recipe from the list is selected and the Button "Laden" (Load) is clicked the already existing recipe is loaded into the recipe manager and can be viewed or manipulated.
   If a new recipe should be saved the dropdown box must be left on the element "-Neues Rezept- "
2. **Recipe Name:** If a new Recipe should be saved then the user has to insert the name of the recipe into the textbox with the caption "Rezeptname" (recipe name). When an already existing recipe is manipulated the current recipe name is loaded, but can be manipulated after loading.
   The textbox with the caption "Rezeptnummer" (recipe number) is filled with the recipe Id when a recipe is loaded from the database. Because the recipe Id is generated by the system automatically. When a new recipe is entered into the system no "Rezeptnummer" (recipe number) is necessary.
3. **Design of Recipe:** The recipe design section consists of several elements. For each station of the plant there are the following elements:
   a. "Inkl." (Include) Checkbox: Determines if the station is included in the recipe or not. If checkbox is checked than the station is included.
   b. "Konfiguration" (Configuration) Dropdown Menu: Sets the configuration of the station in the recipe. The configuration can be selected from the dropdown menu, but no new configurations can be specified.
   c. "geschätzte Zeit" (Estimated Time): Indicates the estimated time according to the configuration from the station.
   d. "Reihenfolge" (Order number): Determines the position of the station in the recipe. A number can only selected once and the numbers must be in order. No numbers are allowed to be missing.

e. "Zuschlag Verfahrzeit" additional time for driving: Calculates the additional time for driving based on the selected stations.
f. "Gesamtdauer" Total time: Indicates the estimated total time for the recipe.

4. **Button Strip:** The button strip shows three buttons with different actions:
   a. Button "Speichern" (save): When the button is clicked either a new recipe is saved or the changes of a recipe are saved.
   b. Button "Löschen" (delete): When the button is clicked the recipe, which is currently loaded will be deleted.
   c. Button "Abbrechen" (abort): When the button is clicked the mask is cleared and closed.

The recipe view implements events for each button when clicked (except the abort button). Furthermore, events for getting the estimated time to each station which the selected configuration are implemented. The naming of these events always start with GetEstimatedTimeOnIndexChanged and end with the number of the station. The last implemented event is the ClaculateTime event which is used to calculated the manufacturing time of the recipe. An Overview of the events can be seen in the figure 75.

```
#region Eventhandler

public event EventHandler<string> GetRecipe;
public event EventHandler<Recipe> SaveRecipe;
public event EventHandler<string> DeleteRecipe;
public event EventHandler<int> EditRecipe;
public event EventHandler UpdateView;
public event EventHandler<string> GetEstimatedTimeOnIndexChanged1;
public event EventHandler<string> GetEstimatedTimeOnIndexChanged2;
public event EventHandler<string> GetEstimatedTimeOnIndexChanged3;
public event EventHandler<string> GetEstimatedTimeOnIndexChanged4;
public event EventHandler<string> GetEstimatedTimeOnIndexChanged5;
public event EventHandler<string> GetEstimatedTimeOnIndexChanged6;
public event EventHandler CalculateTime;
```

Figure 75 Implemented Events at the recipe view

The corresponding methods which fire the events can be seen in the source code in the class RecipeView.cs.

Furthermore, the following methods have been implemented in the View:

```
1 reference
private void DataBindingOrderComboBox()...

4 references
private void ResetViewToDefault()...

1 reference
public void LoadRecipeFromDB(object sender, Recipe recipe)...
```

Figure 76 Methods Recipe Manager

The DataBindingOrderComboBox binds a list of Integer to the order Comboboxes so that the user can select the position of the station in the recipe.

The ReserViewToDefault method sets the state of the main view to the initial state which can be seen in the figure 77 below.

The LoadRecipeFromDB method is called when a recipe should be loaded into the view from the database. The recipe which is transferred via EventArgs is splitted into the stations and each included station is loaded into the view.

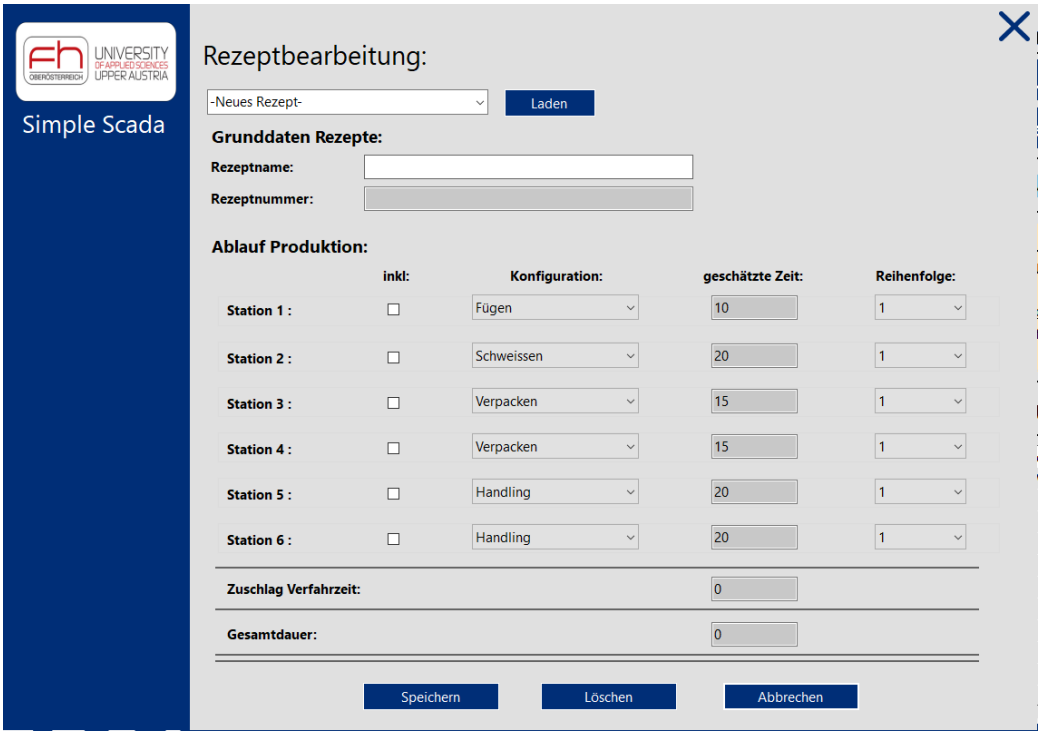


Figure 77 Default state of the recipe manager view

3.5.5.1.2 Recipe Presenter

The RecipeManagerPresenter manipulates the data from the RecipeManager, which is the model, so that the data can be shown correctly on the RecipeView.

Therefore, the Recipemanager and the RecipeView are implemented as member variables. Additionally, an Event Handler to Load a Recipe from the Database is implemented. The member variables and Event Handler can be seen in figure 78.

```
#region Membervariables
//Models
private RecipeManager _recipeManager;
//View
private RecipeView _recipeView;
//Events
public event EventHandler<Recipe> LoadRecipe;
#endregion
```

Figure 78 Implemented member variables recipe manager presenter

An overview of the implemented methods of the RecipeManagerPresenter can be seen in figure 79 and are described below.

```
private void SetUpLinks()...
1 reference
private void SaveRecipe(object? sender, SimpleSCADA_SharedResources.Recipe recipeToAdd)...
1 reference
public void OpenWindow(object sender, EventArgs e)...
1 reference
private void GetRecipe(object sender, string e)...
1 reference
private void UpdateRecipe(object sender, int recipeToEdit)...
3 references
private void UpdateView(object sender, EventArgs e)...
1 reference
private void DeleteRecipe(object sender, string e)...
1 reference
private void ShowEstimatedTimeToUtility1(object sender, string utilityName)...
1 reference
private void ShowEstimatedTimeToUtility2(object sender, string utilityName)...
1 reference
private void ShowEstimatedTimeToUtility3(object sender, string utilityName)...
1 reference
private void ShowEstimatedTimeToUtility4(object sender, string utilityName)...
1 reference
private void ShowEstimatedTimeToUtility5(object sender, string utilityName)...
1 reference
private void ShowEstimatedTimeToUtility6(object sender, string utilityName)...
2 references
private Recipe CheckIfRecipeExistces(string recipename)...
6 references
private void AddStationwithChosenUtitiltyToRecipe(string needeedUtility, int stationID, int order, Recipe recipeToAdd)...
2 references
private void AddStationsToRecipe(Recipe recipe)...
#endregion
7 references
private void CalculateTime(object sender, EventArgs e)...
```

Figure 79 Implemented methods recipe manager presenter

The methods of the RecipeManager Presenter implement the following functionality:

- private void SetUpLinks (): The method is used to connect event source and event sink between the Recipe View and the RecipeManagerPresenter.
- private void SaveRecipe (object? sender, SimpleSCADA_SharedResources.Recipe recipeToAdd): Method is used to save a Recipe.
- public void OpenWindow (object sender, EventArgs e): is used to open the recipe window so that the user is able to interact with the system.
- private void GetRecipe (object sender, string e): Loads a recipe from the Model (Database) based on the name of the recipe.
- private void UpdateRecipe (object sender, int recipeToEdit): Saves the changed recipe to the model (database).
- private void UpdateView (object sender, EventArgs e): Updates the view so that all already existing recipes are shown in the dropdown menu and all utilities of all stations is up to date.
- private void DeleteRecipe (object sender, string e): Deletes a recipe from the model based on the name of the recipe.
- private void ShowEstimatedTimeToUtility1(object sender, string utilityName): Represents the method for all stations. The only difference is the last number which represents the station. The method gets the wright estimated time for the chosen utility (chosen in the view). Furthermore, the methods calls the method CalculateTime.
- private Recipe CheckIfRecipeExistces (string recipename): is used so to check if a recipe if the inserted recipe name exists. The method is called in the methods GetRecipe and DeleteRecipe method.
- private void AddStationwithChosenUtitiltyToRecipe (string needeedUtility, int stationID, int order, Recipe recipeToAdd): Methods adds a station to a recipe with the chosen utility.

- private void AddStationsToRecipe (Recipe recipe): Adds the stations which should be included in the recipe to the recipe.
- private void CalculateTime (object sender, EventArgs e): Calculates the drive time and the estimated time for the recipe production cycle.

3.5.5.1.3 Recipe Model

For the Recipe Management the following model has been used:

- RecipeManager: The recipe manager is described in 3.4.3 SUBSYSTEM2 MODULE 1: Recipe Manager

3.5.5.2 Production Order Management

The Production Order Management allows the user to manipulate the production cycles according to his wishes. A production cycle basically is a recipe which is repeated a certain amount of times. The user is able to create new production cycles, delete production cycles and change already created production cycles. Important to mention is that all production cycles can be altered except the active production cycle.

3.5.5.2.1 Job Processing View

The Job Processing (Production Cycle) View consists of a loading section where already created production cycles can be loaded, a setting section where the production cycle can be design and a button strip. The modules of the recipe view are marked and numbered in the figure 80 below.

Figure 80 Screenshot production order management

1. **Load Production Cycle:** The dropdown box implements a list of all already existing production cycles. If a production cycle from the list is selected and the Button "Laden" (Load) is clicked the already existing production cycle is loaded into the job processing manager and can be viewed or manipulated. The active production cycle can not be

viewed or manipulated so the production cycle is missing in the dropdown menu on purpose.
If a new recipe should be saved the dropdown box must be left empty.

2. **Design of Production Cycle:** The production cycle design section consists of several elements.
    a. "Auftragsname" (job name) Textbox: is filled with the name of the Production cycle when a production cycle is loaded. The production cycle name is generated from the customer name so no input is necessary when a new production cycle is entered.
    b. "Auftragsnummer" (order number) Textbox: is filled with the production cycle Id when a production cycle is loaded into the mask. Because the production cycle Id is generated by the system automatically, when a new production cycle is entered no "Auftragsnummer" (order number) is necessary.
    c. "Kunde" (customer) Textbox: Sets the customer for the production cycle.
    d. "Anzahl" (amount) Textbox: Determines the amount of the repetitions of a recipe.
    e. "Fertigungsrezept" (recipe) Dropdown menu: Sets the recipe which is used for the production cycle.
3. **Button Strip:** The button strip shows three buttons with different actions:
    a. Button "Speichern" (save): When the button is clicked either a new production cycle is saved or the changes of a production cycle are saved.
    b. Button "Stornieren" (cancel): When the button is clicked the production cycle, which is currently loaded will be deleted.
    c. Button "Abbrechen" (abort): When the button is clicked the mask is cleared and closed.

The job processing view implements events for each button when clicked (except the abort button). An Overview of the Events can be seen in the figure 81.

```
#region Events
public event EventHandler<ProductionCycle> NewProductionCycle;
public event EventHandler<ProductionCycle> EditProductionCycle;
public event EventHandler<int> RequestProductionCycle;
public event EventHandler<int> DeleteProductionCycle;
#endregion
```

Figure 81 Overview Implemented Events job processing view

The corresponding method which fire the events can be seen in the source code in the class jobProcessingView.cs.

Furthermore, the following methods have been implemented in the View:

```
7 references
public void UpdateView(List<ProductionCycle> productionCycles, List<Recipe> recipes, ProductionCycle selectedProductionCycle)...

3 references
private int GetProductionCycleIdFromComboBox()...
```

Figure 82 Implemented methods job processing view

The method UpdateView updates the job processing view after any action on the view. In the method the textboxes are either cleared when no job should be loaded or are filled with the information from the selected job from dropdown menu. Additionally, the dropdown menu with the already existing production cycles is updated.

The GetProductionCycleIdFromComboBox method is used for getting the Id of the production cycle.

3.5.5.2.2 Job Processing Manager Presenter

The JobProcessingManagerPresenter manipulates the data from the RecipeManager and the ProductionCycleManager, which are the models, so that the data can be shown correctly on the JobProcessingView. Therefore, the models and the view are implemented as member variables. The implemented member variables can be seen in the figure 83.

```
//Model
private ProductionCycleManager _productionQueue;

private RecipeManager _recipeManager;

//View
private JobProcessingView _jobProcessingView;
```

Figure 83 Implemented member variables job processing manager presenter

An overview of the implemented methods of the JobProcessingManagerPresenter can be seen in figure 84 and are described below.

```
private void SetUpLinks()...

1 reference
public void OpenWindow(object? sender, EventArgs e)...

1 reference
private void GetOrder(object? sender, int id)...

1 reference
private void SaveOrder(object? sender, ProductionCycle newProductionCycle)...

1 reference
private void EditOrder(object? sender, ProductionCycle editProductionCycle)...

1 reference
private void DeleteOrder(object? sender, int id)...
```

Figure 84 Implemented member variables job processing manager presenter

The methods of the Job Processing Manager Presenter implements the following functionality:

- private void SetUpLinks (): The method is used to connect event source and event sink between the Production Cycle View and the JobProcessingManagerPresenter.
- public void OpenWindow (object? sender, EventArgs e): is used to open the production cycle window so that the user is able to interact with the system.
- private void GetOrder (object? sender, int id): loads the production cycle from the database to the view based on the production cycle Id.
- private void SaveOrder (object? sender, ProductionCycle newProductionCycle): Saves the production cycle to the model (database).
- private void EditOrder (object? sender, ProductionCycle editProductionCycle): Saves the changes of a production cycle to the model.

- private void DeleteOrder (object? sender, int id): deletes a production cycle from the database based on the production cycle Id.

More detailed information can be seen in the source code in the class JobProcessingManagerPresenter.cs.

3.5.5.2.3 Job Processing Model

For the production order management the following models have been used:

- RecipeManager: The recipe manager is described in 3.4.3 SUBSYSTEM2 MODULE 1: Recipe Manager
- ProductionCycleManager: The production cycle manager is described in 3.4.4 SUBSYSTEM2 MODULE 2: ProductionCycle Manager.

3.5.6 SUBSYSTEM 3 MODULE 4: User management

The Subsystem User Management is responsible for the management of the users. The Submodule further divides into the management of the users, a log in function and a password change function which can be used by each user.

3.5.6.1 User Management

The user management allows the Superuser to manipulate the users according to his wishes. The superuser is able to create new users, change users and delete users. The user management is only accessible for users with the user permission of a superuser.

3.5.6.1.1 User View

The User Management View consist of a loading section where already created users can be loaded, a section where the user's information can be inserted a and a button strip. The modules of the user management view are marked and numbered in the figure 85 below.

Figure 85 Screenshot User Manager

1. **Load User:** The dropdown box implements a list of all already existing users. If a user from the list is selected and the Button "Laden" (Load) is clicked the already existing user is loaded into the user manager and can be viewed or manipulated. If a new user

should be saved the dropdown menu item must be left on the element "-Neuer Benutzer-"(new user).

2. **Design of Users:** The user design section consists of several elements.
   a. "Vorname" (first name) Textbox: here the first name of the user should be entered when a new user is created or the first name of the user is here displayed if a user is loaded into the mask.
   b. "Nachname" (last name) Textbox: here the last name of the user should be entered when a new user is created or the last name of the user is here displayed if a user is loaded into the mask.
   c. "Initales Passwort" (initial password) Textbox: Here the initial password of the user should be entered when a new user is created. If the user forgets his password the superuser can set a new initial password when the user is loaded and a new password is inserted. If no new password is necessary the textbox should be left empty.
   d. "Rechte" (user permission) Dropdown menu: Sets the permission of the user. The permissions spectator, operator and superuser are possible.
3. **Button Strip:** The button strip shows three buttons with different actions:
   a. Button "Speichern" (save): When the button is clicked either a user is saved or the changes of a user are saved.
   b. Button "Löschen" (delete): When the button is clicked the user, which is currently loaded will be deleted.
   c. Button "Abbrechen" (abort): When the button is clicked the mask is cleared and closed.

The user management view implements events for each button when clicked (except the abort button). Furthermore a event to update the view is implemented. An Overview of the Events can be seen in the figure 86.

```
#region Events
public event EventHandler<string> GetUser;
public event EventHandler<User> SaveUser;
public event EventHandler<EditUserEventArgs> UpdateUser;
public event EventHandler<string> DeleteUser;
public event EventHandler UpdateView;
#endregion
```

Figure 86 Implemented events User management view

The corresponding method which fire the events can be seen in the source code in the class UserView.cs.

Furthermore, the following methods have been implemented in the View:

```
4 references
private void ResetViewToDefault()...

1 reference
public void LoadUserFromDB(object sender, User user)...
```

Figure 87 Implemented methods user management view

The methods implement the following functionality:

- private void ResetViewToDefault(): The methods sets the user manager view to its default state, which means all textboxes are emptied, the list of already existing users is refreshed and the dropdown menu for the user permissions is also reset to its default state.
- public void LoadUserFromDB(object sender, User user): The method is used to load the user from the source into the view based on the user which is handed over by an event.

3.5.6.1.2 User Manager Presenter

The User Manager Presenter manipulates the data from the UserManager, which is the model, so that the data can be shown correctly on the User Manager view. Therefore, the model and the view are implemented as member variables. To get the data from a user to the view via the presenter an event is implemented.

The implemented member variables and events can be seen in the figure 88.

```
//Model
private UserManager _userManager;

//View
private UserView _userView;

//Events
public event EventHandler<User> LoadUser;
```

Figure 88 Member variables User manager presenter

An overview of the implemented methods of the UserManagerPresenter can be seen in figure 89 and are described below.

```
private void SetUpLinks()...

1 reference
private void EditUser(object? sender, EditUserEventArgs newUser)...

1 reference
public void OpenWindow(object sender, EventArgs e)...

3 references
private void UpdateView(object sender, EventArgs e)...

1 reference
private void GetUser(object sender, string username)...

1 reference
private void SaveUser(object sender, User user)...

1 reference
private void DeleteUser(object sender, string username)...

3 references
private User CheckIfUserExistces(string username)...
```

Figure 89 Implemented methods user manager presenter

The methods implement the following functionality:

- private void SetUpLinks():The method is used to connect event source and event sink between the UserView and the UserManagerPresenter.
- private void EditUser(object? sender, EditUserEventArgs newUser): The method is used to implement the function of editing the user. Because the user name is designed as firstname.lastname and both can be edited the old username and the edited user are send with custom EventArgs (called EditUserEventArgs). The Eventargs can be seen in the figure 90 below.

```
public class EditUserEventArgs:EventArgs
{
    public string UserName;
    public User EdittedUser;

    1 reference
    public EditUserEventArgs(string UserName, User EdittedUser)
    {
        this.UserName = UserName;
        this.EdittedUser = EdittedUser;
    }
}
```

Figure 90 Source code custom EventArgs editting User

- public void OpenWindow(object sender, EventArgs e): is used to open the user management window so that the user is able to interact with the system.
- private void UpdateView(object sender, EventArgs e): Updates the view so that all already existing users are shown in the dropdown menu.
- private void GetUser(object sender, string username): is used to get a user from the model(database). The username consists of the firstname.lastname.
- private void SaveUser(object sender, User user): is used to save a user from the view to the model (database) based on the input of the user.
- private void DeleteUser(object sender, string username): deletes a user from the model based on the user name of the user. The username is a constellation such as firstname.lastname.
- private User CheckIfUserExistces(string username): the method is used to check if a user with a username in the form of firstname.lastname already exists on the database.

More detailed information can be seen in the source code in the class UserManagerPresenter.cs.

3.5.6.1.3 User Manager Model

For User Management the following model has been used:

- UserManager: The user manager is described in 3.4.6 SUBSYSTEM2 MODULE 4: User Manager

3.5.6.2 User Log In

The User Log In allows the User to log onto the system and manipulate the different views according to his permissions. To make any changes on the system the user has to be logged in.

3.5.6.2.1 Log in View

The Log in View consists of an input section and a button to sign in the user. The modules of the log in window are marked and numbered in the figure 91 below.

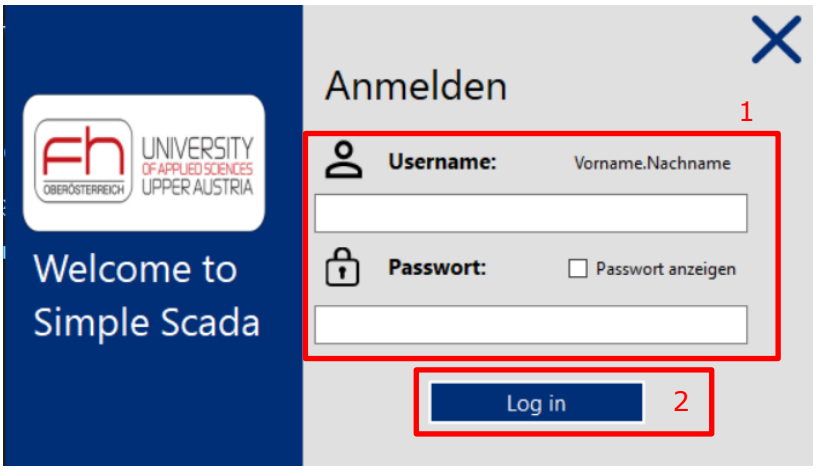


Figure 91 Screenshot Log in Window

1. **Input Area User Log in:** the input section consists of several elements.
   a. "Username" (username) Textbox: here the user should input his username in the form of Firstname.Lastname. The upper- and lower-case writing is important. The first letters of each part of the username should be written with a upper case. The first name and last name should be separated with a dot.
   b. "Passwort" (password) Textbox: here the password can be entered by the user.
   c. "Passwort anzeigen" (show password) Checkbox: if the checkbox is checked then the password is visible else the letters are symbolized with a dot.
2. **Log In Button:** When the button is clicked the username and password are checked and if both are correct and are existing the user is logged into the system else an error message in the form of an pop up window is shown.
   The user is also able to log in with the enter key.

The log in view implements events for each button when clicked. An Overview of the Events can be seen in the figure 92.

```
public event EventHandler<string> CheckLogInDataFromUser;
```

Figure 92 Implemented events log in view

The corresponding method which fire the events can be seen in the source code in the class LogView.cs.

Furthermore, the following methods have been implemented in the View:

```
1 reference
private void cbPasswordView_CheckedChanged(object sender, EventArgs e)...

1 reference
public void ResetView(object sender, EventArgs e)...

1 reference
private void txtbPasswordKeyDown(object sender, KeyEventArgs e)...
```

Figure 93 implemented methods log in view

- private void cbPasswordView_CheckedChanged(object sender, EventArgs e): With this method the password textbox changes between a visible password or the password is shown with placeholders (dots). The method is called when the chechbox check is changed.
- public void ResetView(object sender, EventArgs e): This method resets the log in view to its original state which is shown in the figure 91.
- private void txtbPasswordKeyDown(object sender, KeyEventArgs e): with this method its possible that the user is logged in with the enter key. (The user is only logged on when the username and password are entered correct.)

3.5.6.2.2 Log In Presenter

The Log In Presenter manipulates the data from the UserManager, which is the model, so that the data can be shown correctly on the Log In view. Furthermore, events for data handling have been implemented. Therefore, the model and the view are implemented as member variables.

The implemented member variables and events can be seen in the figure 94 and 95.

```
//View
private LogView _logView;

//Modell
private UserManager _userManager;
```

Figure 94 Implemented member variables log in presenter

```
public event EventHandler<User> UserToLogIn;
public event EventHandler ResetView;
```

Figure 95 Implemented events log in presenter

Figure 96 Implemented member variables log in presenter

The UserToLogIn event is fired when the log in data are valid and the user should be logged on the system. The ResetView event is fired when the vlog in view should be put into its initial state.

An overview of the implemented methods of the LogInWindowPresenter can be seen in figure 97 and are described below.

```
1 reference
public void OpenWindow(object? sender, EventArgs e)...

1 reference
public void LogInSucessful(object? sender, string userToLogIn)...

1 reference
private void SetUpLinks()...

1 reference
public void LogOutUser(object sender, EventArgs e)...
```

Figure 97 Implemented methods log in presenter

- public void OpenWindow(object? sender, EventArgs e): is used to open the log in window so that the user is able to log onto the system.
- public void LogInSucessful(object? sender, string userToLogIn): the method checks if the user input is valid for any user saved on the database. If the user is valid then the active user is set to the user which tries to log on and an event is fired which sets the plant view (main view) according to the permissions of the user.
- private void SetUpLinks():The method is used to connect event source and event sink between the LogInView and the LogInWindowPresenter.
- public void LogOutUser(object sender, EventArgs e): the method logs out the currently logged in user. When no user is logged in then the active user is null.

3.5.6.2.3 Log In Model

For Log In Functionality the following model has been used:

- UserManager: The user manager is described in 3.4.6 SUBSYSTEM2 MODULE 4: User Manager

3.5.6.3 Password Management

The Password Management allows the User to change his personal password, when the existing password is known. The password change view also shows the user the password strength to support the user in his choice of password.

3.5.6.3.1 Password Change View

The Password Change View consists of an input section and a button to acknowledge the password change. The modules of the password change window are marked and numbered in the figure 98 below.

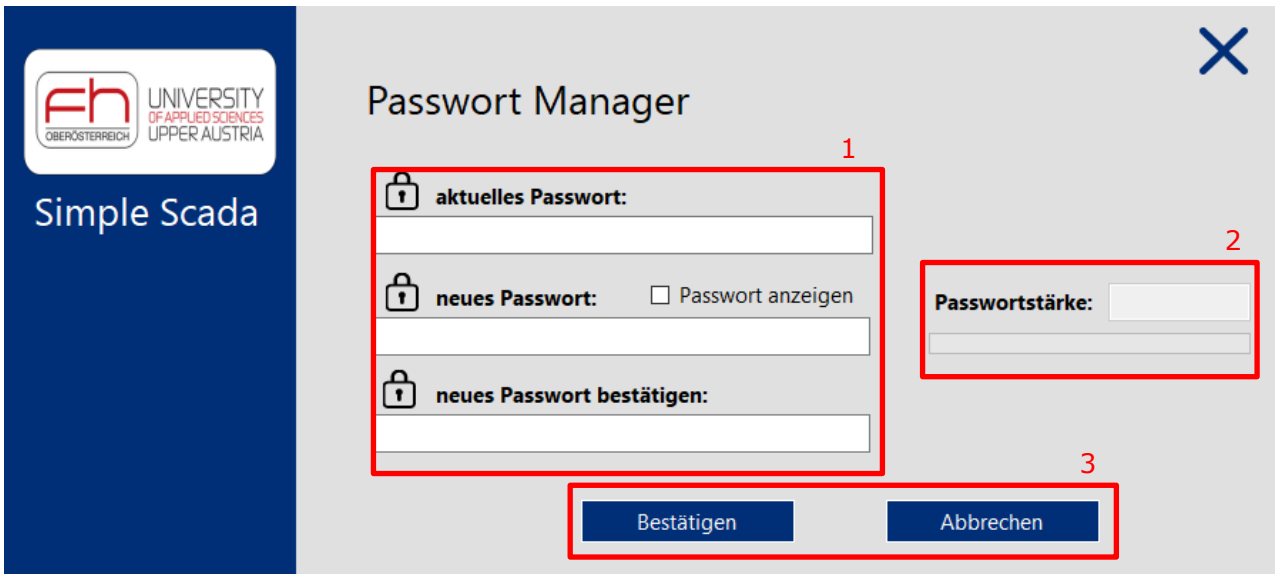


Figure 98 Screenshot password change window

1. **Input Area:** The input area at the password change view consists of more than one element.

a. "aktuelles Passwort" (current password) Textbox: here the user can enter his old password. The input is necessary if the user wants to change his password.
b. "neues Passwort" (new password) Textbox: in this textbox the user can enter his new password. During the entering of the password the password strength is checked and shown.
c. "neues Passwort bestätigen" (confirmation new password) Textbox: In order to make sure the password was entered correctly the new password has to be entered a second time.
d. "Passwort anzeigen" (show password) Checkbox: if the checkbox is checked then the new password and the confirmation of the new password are visible else the letters are symbolized with a dot.

2. **Password Strength Checker:** In this area the password strength of the newly entered password is shown. The complexity rules are as following:
   a. Very Low: length shorter than 8 and no lower case letter
   b. Low: at least one lower case letter and the length is at least 8 characters
   c. Medium: at least one lower case letter, one upper case letter and a length of 8 characters
   d. Strong: at least one lower case letter, one upper case letter, one number and a length of 8 characters
   e. Very Strong: at least one lower case letter, one upper case letter, one number, one special character of the following list "#?!@$%^&*-+" and a length of minimum 8 characters
3. **Button Strip:** The button strip shows two buttons with different actions:
   a. Button "Speichern" (save): When the button is clicked the new password is saved.
   b. Button "Abbrechen" (abort): When the button is clicked the mask is cleared and closed.

The password change view implements events for each button when clicked (except the abort button).Furthermore a event is implemented when the input of the new password is changed. This event is used to check the strength of the password. An Overview of the Events can be seen in the figure 99.

```
public event EventHandler<string> PasswordChange;
public event EventHandler<string> InputNewPasswordChanged;
```

Figure 99 Implemented events password change window

The corresponding method which fire the events can be seen in the source code in the class PasswordChangeView.cs.

Furthermore, the following methods have been implemented in the View:

```
3 references
private void ClearView()...
1 reference
private void cbPasswordView_CheckedChanged(object sender, EventArgs e)...
1 reference
public void ShowMessageBoxPasswordChanged(object sender, EventArgs e)...
1 reference
private void txtbNewPassword_TextChanged(object sender, EventArgs e)...
```

Figure 100 Implemented methods password change window

- private void ClearView(): This method empties all textboxes. The Password Change view always clears after closing for password safety.
- private void cbPasswordView_CheckedChanged(object sender, EventArgs e): With this method the new password textbox and the confirmation new password textbox changes between a visible password or the password is shown with placeholders (dots). The method is called when the checkbox check is changed.
- public void ShowMessageBoxPasswordChanged(object sender, EventArgs e): This method shows a pop up window where the user is informed that the password has been changed after clicking the save button.
- private void txtbNewPassword_TextChanged(object sender, EventArgs e): The method is used to fire a event to the presenter where the password strength is checked.

3.5.6.3.2 Password Change Presenter

The Change Password Presenter manipulates the data from the UserManager, which is the model, so that the data can be shown correctly on the Password change view. Furthermore, events for data handling have been implemented. Therefore, the model and the view are implemented as member variables.

The implemented member variables and events can be seen in the figure 101.

```
#region Variables
private PasswordChangeView _passwordChangeView;
private UserManager _userManager;

#endregion

#region Events
public event EventHandler PasswordChangedInDB;
#endregion
```

Figure 101 Implemented member variables and events password change presenter

The PasswordChangedInDB event is fired when the change of password has been done in the model(database).

An overview of the implemented methods of the ChangePasswordViewPresenter can be seen in figure 102 and are described below.

```
1 reference
public void OpenWindow(object? sender, EventArgs e)...

1 reference
private void SetUpLinks()...

1 reference
private void CheckProtectionLevelOfPassword(object? sender, string e)...

1 reference
private void ChangePasswordInDB(object? sender, string passwords)...

1 reference
private PasswordLevel CheckStrength(string password)...
#endregion
```

Figure 102 Implemented methods password change presenter

- public void OpenWindow(object? sender, EventArgs e): is used to open the password change window so that the user is able to change his own password.
- private void SetUpLinks():The method is used to connect event source and event sink between the PasswordChangeView and the ChangePasswordViewPresenter.
- private void CheckProtectionLevelOfPassword(object? sender, string e): is used to check the protection level of the new password. The method calls the method CheckStrength(string password).
- private void ChangePasswordInDB(object? sender, string passwords):Changes the password in the database after the save button is clicked if possible, else a error message is shown.
- private PasswordLevel CheckStrength(string password):Checks the user input for the new password. The password strength is checked according to the complexity rules described above. The complexity rules are implemented with Regex expressions.

3.5.6.3.3 Password Change Model

For Password Change Functionality the following model has been used:

- UserManager: The user manager is described in 3.4.6 SUBSYSTEM2 MODULE 4: User Manager

3.5.7 SUBSYSTEM 3 MODULE 5: Productivity

3.5.7.1 OEE View

The OEE view was described in the chapter 3.4.9 SUBSYSTEM2 MODULE 7: OEE Manager.

3.5.8 SUBSYSTEM 3 MODULE 6: User Entertainment

The Subsystem User Entertainment is responsible for the entertainment of the user. The subsystem tells you Chuck Norris jokes when asked.

3.5.8.1 Joke View

The Joke View consists of an output section and a button. The modules of the password change window are marked and numbered in the figure 103 below.

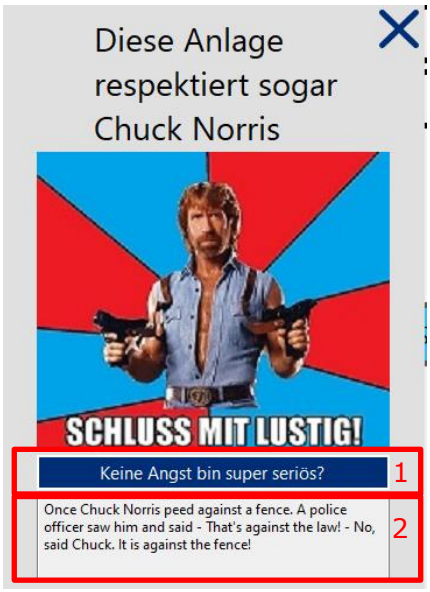


Figure 103 Screenshot Joke view

1. **Button "Keine Angst bin super seriös":** (no worries I am super serious) The button is used to call new Chuck Norris jokes to the output area.
2. **Output Area:** At the output area the jokes are displayed.

The joke view implements events for each button when clicked. An overview of the implemented events is shown in the figure 104.

```
public event EventHandler<EventArgs> SeriousButton;
```

Figure 104 Implemented events joke view

The corresponding method which fire the events can be seen in the source code in the class JokeView.cs.

In order to make the window moveable across the screen the following variables have been implemented.

```
//Variables
bool isMouseDown;
Point lastLocation;
```

Figure 105 Implemented member variables joke view

Furthermore, the following methods have been implemented in the View:

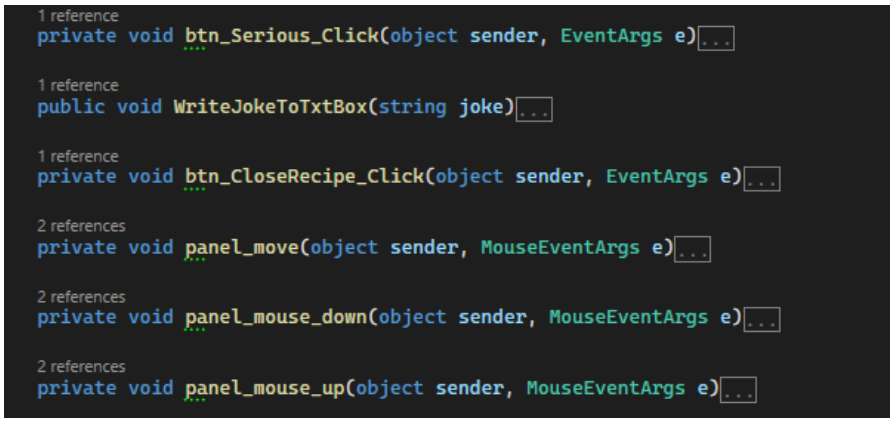


Figure 106 Implemented methods joke view

The method shown in figure 106 implement the following functionality:

- public void WriteJokeToTxtBox(string joke): The method is responsible to write the joke, which is an parameter of the method, into the textbox on the view to make the joke visible for the user.
- private void btn_CloseRecipe_Click(object sender, EventArgs e): The method closes and clears the view.
- private void panel_move(object sender, MouseEventArgs e): The method is used for the movement of the view. Therefore, the relative movement from the last Position to the new position is calculated and the location is saved into the Location variable.
- private void panel_mouse_down(object sender, MouseEventArgs e): The method is responsible to start the movement of the view when a mouse button is clicked. The view should be clicked in the top section of the window.
- private void panel_mouse_up(object sender, MouseEventArgs e): The method is responsible to release the window when the movement of the window is finished an the mouse button is released.

3.5.8.2 Joke Presenter

The Joke Window Presenter manipulates the data from the ChuckNorrisJokeDownloader, which is the model, so that the data can be shown correctly on the Joke view.

The implemented member variables can be seen in the figure 107.

Figure 107 Member variables joke presenter

An overview of the implemented methods of the JokeWindowPresenter can be seen in figure 108 and are described below.

```
1 reference
private void SetUpLinks()...

1 reference
public void OpenWindow(object sender, EventArgs e)...

1 reference
private async void GetJoke(object sender, EventArgs e)...
```

Figure 108 Implemented methods joke presenter

- public void OpenWindow(object? sender, EventArgs e): is used to open the Joke window.
- private void SetUpLinks():The method is used to connect event source and event sink between the JokeView and the JokeWindowPresenter.
- private async void GetJoke(object sender, EventArgs e): the methods calls an async method from the ChuckNorrisJokeDownloader, which is a static Task, to get a Chuck Norris Joke from the API.

3.5.8.3 Joke Model

The joke model has not been implemented on the database instead the method connects to an API and Downloads a Joke as a Json object which is then converted to a string. The method works as an async Task. The ChuckNorrisJokeDownloader represents the Joke Model. The Joke model as it is implemented is shown in figure 109.

```
public class ChuckNorrisJokeDownloader
{
    1 reference
    public static async Task<string> GetJsonString()
    {
        // Request API
        HttpClient client = new HttpClient();
        HttpRequestMessage request = new HttpRequestMessage
        {
            Method = HttpMethod.Get,
            RequestUri = new Uri("https://matchilling-chuck-norris-jokes-v1.p.rapidapi.com/jokes/random"),
            Headers =
            {
                { "accept", "application/json" },
                { "X-RapidAPI-Key", "0deff8ac7dmsh4e5698cdf37b023p144eb4jsn10ab8c50e715" },
                { "X-RapidAPI-Host", "matchilling-chuck-norris-jokes-v1.p.rapidapi.com" },
            },
        };

        Joke joke = new Joke();

        // Download Json
        using (var response = await client.SendAsync(request))
        {
            response.EnsureSuccessStatusCode();
            var body :string = await response.Content.ReadAsStringAsync();

            // Convert Json to Joke
            joke = JsonConvert.DeserializeObject<Joke>(body);
        }

        return joke.value;
    }
}
```

Figure 109 Source code joke model

3.6 SUBSYSTEM 4 – Android App

The Subsystem has been planned and designed, but due to the lack of time the Android App has not been implemented.

4 Factory Acceptance Test

4.1 General Information

This test procedure allows conformity check of the software programme under test with the specifications.

Due to the lack of time there hasn´t been an implementation of the different types of testing. Nevertheless, an overview of possible testing methods has been made as well as an brief overview of the benefits of testing the methods based on the experience which was generated in the project.

4.2 Possible Tests

A part goal of developing this application was also to use and program automated Unit-Tests. Due to organizational issues a realization of these wasn't possible, other modules where always more prioritized. Nevertheless, here are some good reasons and ideas how an implementation of Unit-Test could have helped us in developing the application:

Early Bug Detection: Automated unit tests enable the early identification and rectification of defects within the codebase, minimizing the time and effort expended on debugging in later stages of development.

1. Regression Testing: Over the course of application evolution, modifications to one part of the code may inadvertently disrupt functionality in other areas. Unit tests guarantee that existing features continue to operate as anticipated when new code is introduced or altered.

2. Documentation: Unit tests function as a form of documentation for the code, offering explicit examples of the anticipated behaviour of individual components or modules. This aids other developers, including future team members, in comprehending and collaborating on the codebase.

3. Code Quality: The practice of writing unit tests promotes sound coding practices, including modular and testable code design, adherence to SOLID principles, and minimization of code coupling. These practices lead to a more maintainable and well-structured codebase.

4. Refactoring Safety: Unit tests provide a safety net when refactoring code. When enhancements are made to enhance code quality or performance, running unit tests ensures that the existing functionality remains intact.

5. Continuous Integration and Continuous Delivery (CI/CD): Automated unit tests are an integral component of a CI/CD pipeline. They can be seamlessly integrated into the build and deployment process to verify that code alterations do not compromise existing functionality before being deployed to the production environment.

6. Facilitated Collaboration: Unit tests simplify collaboration among multiple developers on a project. When one developer implements changes, they can execute unit tests to ensure that their modifications do not introduce regressions affecting the work of other team members.

7. Reduced Manual Testing: While unit tests do not supplant all forms of testing (e.g., integration or UI testing), they significantly diminish the necessity for manual testing, thereby conserving time and resources.

A Appendix: List of abbreviations

Abbreviation	Definition
AD	Applicable Document
RD	Referenced Document
MVP	Model View Presenter
JM	Jäger Maximilian
BL	Brajdic Lukas
FL	Feichtmair Lauretta
NH	Naciri Hamza
KS	Kerschbaummayr Stefan
TS	Tollich Sebastian
DK	Dautovic Kenan