Teemu Vähä-Impola

Architectural Improvement of Display Viewer 5 Software

Master's Thesis in Information Technology

November 6, 2020

University of Jyväskylä

Faculty of Information Technology

Author: Teemu Vähä-Impola

Contact information: teemu.vaha-impola@neste.com

Supervisors: PhD Raino Mäkinen, MSc Tommy Rikberg, and MSc Lauri Saurus

Title: Architectural Improvement of Display Viewer 5 Software

Työn nimi: Display Viewer 5 -ohjelmiston arkkitehtuurin parantaminen

Project: Master's Thesis

Study line: Software Technology

Page count: 76+1

Abstract: In this thesis, an improved architecture for Display Viewer 5 (DV5) software was studied. The new architecture would enforce MVVM architecture more strongly, make clearer divisions of the software's parts and enhance maintainability and reusability of the software, thus making the software more customizable for new projects and suitable for the customers' needs. As a result, the existing MVVM architecture was strengthened by enforcing division into models, views and viewmodels. In addition, redundant duplications were removed and certain code was divided into their own separate entities.

Keywords: architecture, software engineering

Suomenkielinen tiivistelmä: Tässä tutkielmassa Display Viewer 5 (DV5) -ohjelmistolle pyrittiin löytämään parempi arkkitehtuuri, jonka seurauksena huollettavuus ja uudelleenkäytettävyys kasvavat ja ohjelmiston kustomointi uusille asiakkaille helpottuu. Tuloksena päädyttiin vahvistamaan jo nykyistä MVVM-arkkitehtuuria tekemällä jokaiselle luokalle tarvittavan arkkitehtuurin vaatiman jaon, poistamalla turhia duplikaatteja koodissa ja jakamalla itsenäiset kokonaisuudet omiin luokkiinsa.

Avainsanat: arkkitehtuuri, ohjelmistotekniikka

Preface

"Toinen gradu on helpompi" -Tuntematon tutkija ca. 2017.

This master's thesis was done for Neste Engineering Solutions and NAPCON organization during the year of 2020. The thesis was mainly written in Porvoo.

I would like to thank Neste Engineering Solutions and NAPCON for giving me such an opportunity to work on an interesting master's thesis topic that has practical value to the company. I would also like to thank my supervisors Tommy Rikberg and Lauri Saurus for their valuable knowledge and guidance to DV5 software.

Special thanks to Heidi and Bilberry for all the support and motivation during the year.

Jyväskylä, November 6, 2020

Teemu Vähä-Impola

Glossary

ADD	Attribute-Driven Design
ASC	Architectural Separation of Concerns
API	Application programming interface
BAPO	Business Architecture Process and Organization
DCAR	Decision-Centric Architecture Reviews
DCS	Distributed control system
DV5	Display Viewer 5 software
Flavor	Customized and specialized DV5 version
HTML	Hypertext Markup Language
IS	Information system
OTS	Operator training simulator
PASA	Performance Assessment of Software Architectures
RUP 4+1	Rational Unified Process 4+1 views
S4V	Siemens' 4 Views
SPE	Software Performance Engineering
SQL	Structured Query Language
UI	User interface
UML	Unified Modeling Language
XML	Extensible Markup Language
XAML	Extensible Application Markup Language

List of Figures

Figure 1. Causes of losses in the industry. (Kallakuri et al. 2018) 6
Figure 2. DV5 main window. 9
Figure 3. DV5 hierarchy window. 10
Figure 4. DV5 picture window. 10
Figure 5. DV5 faceplate. 11
Figure 6. DV5 alarm window. 12
Figure 7. DV5 trend window. 13
Figure 8. NAPCON Simulator Trainer Dashboard. 14
Figure 9. Graphical presentations of layered architectures. (Koskimies and Mikkonen 2005) 23
Figure 10. Pipes-and-filters architecture. (Koskimies and Mikkonen 2005) 24
Figure 11. Client-server architecture. (Koskimies and Mikkonen 2005) 25
Figure 12. Message dispatcher architecture. (Koskimies and Mikkonen 2005) 27
Figure 13. The overview of model-view-controller architecture. (Potel 1996) 28
Figure 14. The operation of the model-view-controller architecture. (Koskimies and Mikkonen 2005) 29
Figure 15. The overview of the model-view-presenter architecture. (Potel 1996) 30
Figure 16. The overview of the Model-View-ViewModel architecture. ("The MVVM Pattern" 2012) 31
Figure 17. Interpreter architecture. (Koskimies and Mikkonen 2005) 33
Figure 18. Product platform and a product. (Koskimies and Mikkonen 2005) 34
Figure 19. Framework with specialization interface. (Koskimies and Mikkonen 2005) 36
Figure 20. Architecture based software development process. (Koskimies and Mikkonen 2005) 37
Figure 21. Architectural design activities. (Koskimies and Mikkonen 2005) 38
Figure 22. Architecture of NAPCON Simulator. 44
Figure 23. The current architecture of DV5. 46
Figure 24. The proposed new architecture of DV5 50

List of Tables

Table 1. Number of new files by type that are created by separating entities. 53
Table 2. Threshold values for cyclomatic complexity. (Bray et al. 1997) 56
Table 3. Measurement results of Sonarqube static code analysis 57
Table 4. Additional measurement results of Sonarqube static code analysis 58
Table 5. The number of methods within each cyclomatic complexity threshold category. 59
Table 6. The number of methods within each cognitive complexity threshold category. 59
Table 7. The amount of duplications between different flavors. 60

Contents

1 INTRODUCTION 1
1.1 Background 1
1.2 Objective 2
1.3 Scope 2
1.4 Research methodology 2
1.5 Research questions 3

2 OPERATOR TRAINING SIMULATORS 5
2.1 Demand for operator training simulators 5
2.2 Advantages of using OTS 6
2.3 NAPCON Simulator 7
2.3.1 DV5 8
2.4 Training setup 13

3 SOFTWARE ARCHITECTURE 15
3.1 Definition 16
3.2 Architecture's objective 18
3.3 Software architecture description 19
3.4 Design patterns 20
3.5 Architectural styles 21
3.5.1 Layered architecture 21
3.5.2 Pipes-and-filters architecture 23
3.5.3 Client-server architecture 24
3.5.4 Message dispatcher architecture 26
3.5.5 Model-View-Controller (MVC) architecture 27
3.5.6 Model-View-Presenter (MVP) architecture 29
3.5.7 Model-View-ViewModel (MVVM) architecture 30
3.5.8 Repository architecture 32
3.5.9 Interpreter architecture 32
3.6 Product-line architecture 33
3.7 Object-oriented framework 35
3.8 Architectural design 36
3.9 Architectural evaluation 38

4 CURRENT ARCHITECTURE 43
4.1 NAPCON Simulator architecture 43
4.2 DV5 architecture 45
4.3 Other architectural styles expressed in DV5 48

5 PROPOSED IMPROVEMENTS TO ARCHITECTURE 50
5.1 MVVM pattern and code quality factors 52

6 RESULTS 55

6.1 Metrics used in the evaluation 55
6.2 Results of the analyses 57
6.3 Additional results 58

7 DISCUSSION 61
7.1 Current architecture 61
7.2 New architecture 62
7.3 Utilization of the results 63

8 CONCLUSION 64

BIBLIOGRAPHY 66

APPENDICES 70

1 Introduction

The purpose of this master's thesis is to review and evaluate the existing architecture of Display Viewer 5 (DV5) software, to study alternate architectures and to create and design an improved and more modular architecture to be used in future projects. The existing architecture has many parts that need to be modified to meet the requirements of different customers. Thus, the aim is to create a modular structure that allows the modules to be reused in specifically customized versions of DV5.

1.1 Background

NAPCON DV5 is a software that is used concurrently with NAPCON ProsDS and NAPCON Informer to run NAPCON Operator Training Simulator (OTS) also known as NAPCON Simulator. NAPCON Simulator is an essential part of the NAPCON Train product family, which offers operator training services for customers. NAPCON Simulator is used by Neste in Finland and by other companies internationally.

ProsDS is a dynamic process simulator that is developed by Neste Engineering Solutions and uses ANSI Common Lisp as its programming language. NAPCON Informer is both a real-time and history database that collects and stores real-time process data during simulations. It is programmed with Microsoft's .NET technology.

DV5 is used to show displays of simulation that imitates the processes of the customer's plant. In other words, DV5 is an emulation of a plant's distributed control system (DCS) user interface (UI). It has been developed using C# programming language and Windows Presentation Foundation (WPF) technology. DV5 is used in simulation to view and modify equipment conditions and process values and to control the process by the operator. Simulation allows the operator to train how operating the plant would be in real world and to face simulated emergencies that could cause massive losses to the company that faces them in reality.

1.2 Objective

The objective of this thesis is to create an improved and more maintainable architecture that allows the same modules to be reused in multiple different DV5 software. A DV5 flavor is a customized and specialized version of DV5 and all different flavors of DV5 are uniquely modified to satisfy a certain customer's processes' needs. A certain DV5 flavor also has a unique style and visual elements that imitate the actual UI of the desired DCS. Thus, there should be a base version of DV5 that requires minimal programming and moderate customization in order to be reused. Improved architecture and maintainability would enhance project throughput and lessen possible errors that would otherwise delay the delivery process.

1.3 Scope

Scope of this thesis is the full architecture of one single DV5 flavor. The changes of architecture do not extend to other flavors. Other related software that are used to run the simulator will not be included in the scope. This is done in order to limit the workload that would otherwise be too high.

1.4 Research methodology

The first research method used in this thesis is literature review. Relevant literature will be reviewed and studied to gain valuable knowledge in software architecture and the information systems (IS) field as a whole. The second research methodology used is design-science research that aims to create something new or improved (an artefact) to solve a problem.

Hevner et al. (2004, p.82–83) identify design science to be a problem solving process that aims to gain knowledge, understanding and the solution for a design problem through building and applying an artifact. The authors propose seven guidelines that make design-science research meaningful. First guideline is that "design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation." Second guideline is that "the objective of design-science research is to develop technology-based solutions to important and relevant business problems". Third guideline is that "the utility,

quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods. The other four design-science research guidelines proposed by Hevner et al. (2004) concern research contributions, research rigor, design as a search process and communication of research.

Hevner et al. (2004, p.86) divide design evaluation methods into five categories: Observational, analytical, experimental, testing and descriptive methods. Observational methods include case study and field study. Analytical methods include static analysis, architecture analysis, optimization and dynamic analysis. Experimental methods are controlled experiments in controlled environments and simulations that execute the artifact with artificial data. Testing consists of functional testing and structural testing that aim to find defects and test all execution paths of the artifact. Lastly, descriptive methods include informed arguments and scenarios that support and demonstrate the artifact's utility.

(Hevner et al. 2004, pp. 80-81) state that research methodologies provide guidelines that can be used in the evaluation phase of information systems research. The authors elaborate that in behavioural science, methodologies are usually connected to data collection and empirical analysis techniques, whereas in design science, mathematical and computational methods are used more often to evaluate the resulting quality and effectiveness of the created artifacts. They also note that some empirical techniques can be used in design science as well.

Analytical evaluation methods are used in this thesis to evaluate the current architectural design and the proposed improvements. The evaluation will be made by using static code analysis methods and by comparing the new and the old architectures. The comparison will be made by collecting certain metrics given by the static code analysis and by checking which aspects of the software have improved.

1.5 Research questions

This thesis will aim to answer the following two research question:

1. What is the current architecture for DV5?
2. How can the current architecture be improved?

The first step will be to examine and map the current architecture of DV5. This examination itself will bring valuable knowledge for DV5 developers to further understand the structure of the software. It will also be helpful for other stakeholders to understand how certain parts of the software might affect their work.

The second step will be to examine how to improve the current architecture of DV5. This examination may reveal possibilities for improvements that might enhance modularity, performance, understandability of the code and the ability to be more easily customizable. To verify how the improvements affect the software, the current and the new improved architecture will be analyzed with static code analysis methods and the two architecture will be compared side-by-side.

2 Operator training simulators

In this chapter, a background for operator training simulators is built from literature written in the field. Firstly, the chapter addresses why there is a need for operator training simulators. Secondly, the advantages of using OTS will be discussed and thirdly, NAPCON Simulator will be reviewed.

2.1 Demand for operator training simulators

Operators have a plurality of different tasks they need to perform in order to manage a process plant. These tasks include controlling, monitoring, operating and maintaining the plant and its equipment during production as well as start-up, turn-around, shut-down and other various unit operations. They have to be able to navigate through all the distributed control system's (DCS) views, monitor all the various process parameters, take correct actions and handle abnormal process conditions if the need arises. (Kallakuri et al. 2018, p. 79) This is why there is a need for training simulators that imitate and represent the real-world plant operations and conditions but in a virtual world where the actual plant's operations are not disturbed.

According to Cameron, Clausen, and Morton (2002, p. 393) over past decades dynamic simulations have matured and have been used for training operators in capital-intensive and safety-critical areas, such as oil platforms, power stations and nuclear reactors. Advances in computer technology, such as computer speed, programming methods and user interface (UI) facilities have elevated dynamic simulation to be a robust, effective and inexpensive way of training operators. The author also mentions that due to improved computer speeds, dynamic simulators can also be used to aid engineering design and operations support.

According to the Center for Chemical Process Safety of American Institute of Chemical Engineers, the biggest cause of losses in the process industry is human error. This is due to the fact that many employees are not fully prepared for emergency situtions because they have not received any training on the plant control systems. Figure 1 shows how losses are divided in hydrocarbon sector and how they are caused by different factors. It is shown that the biggest losses are caused by human error. (Kallakuri et al. 2018, pp. 79-80)

Figure 1. Causes of losses in the industry. (Kallakuri et al. 2018, p.80)

2.2 Advantages of using OTS

Operator training simulators have been widely adopted in oil and gas industries. There are multiple advantages to training operators on a simulator rather than with conventional methods. The conventional methods do not train for the seldom-occuring emergency situations where correct choices by the operator are most important. Thus, operator training simulators are a good alternative in training operators without any actual danger in the plant site. (Patle, Ahmad, and Rangaiah 2014)

Operator training simulators are used in other areas and sectors for training purposes as well. These areas include aviation, chemical industries and building constructions. Oil and gas industry uses simulators to train operators for start-up, shut-down, normal and abnormal process situations. After initial training the response skills of an operator drop significantly unless proper training is given through simulated scenarios and repeated uses of the training simulator. (Kallakuri et al. 2018, p. 80)

With operator training simulators becoming more widely used, they can be utilized to analyze operations and their safety issues, as well as to further train operating staff to handle plant failures. To achieve these, it is important that the simulated process and its variables are

reasonably accurate when compared to their real-world counterparts and the process model. (Balaton, Nagy, and Szeifert 2013, pp. 335-336)

Kallakuri et al. (2018, p. 86) found in their study that simulators can be used to optimize procedures, enhance operator confidence in executing tasks, improve the operators' skills and thus improve product quality and reduce the amount of human errors due to higher competency.

Even though operator training simulators clearly yield an advantage to companies, there is only little research done on the area. Kallakuri et al. (2018, p. 80) cite research by Salas (2006) that discusses how confidence in simulators and simulator training might be due to the fact that the research on effectiveness of simulators is mostly based on subjective evaluations of trainees and not based on objective performance data. The authors also state that the literature in simulators lacks significant research on how different elements in simulator training can enhance active learning and engagement.

2.3 NAPCON Simulator

NAPCON Simulator is used as a desktop application. This means that DV5 software is run locally with both NAPCON Informer and ProsDS. NAPCON Simulator has many advantages and offers the following features for its users ("NAPCON Simulator" 2020):

- Comprehensive productions states, including start-up, shut-down and special breakdown situations,
- easy-to-use graphical interface that allows teachers to control the process and create faults at will,
- independence from a plant's automation system and DCS manufacturer,
- high fidelity process models and
- data visualization for the simulated process.

DV5 is most thoroughly studied in this thesis. Thus, other parts of NAPCON Simulator will not be examined as deeply as DV5. There will be brief explanations of different parts that help to understand the architecture presented in chapter 4.

2.3.1 DV5

DV5 emulates the DCS of a plant. The most important parts of the DV5 user interface are:

- Main window, which helps users get started and to navigate the simulator environment,
- hierarchy window, which helps navigating the process area through hierarchical structure of process displays,
- picture window that shows the displays of process areas,
- faceplates or loop windows that are opened by clicking on any equipment and through which the equipment's operation can be managed,
- alarm window that keeps track of ongoing alarms that might need attention, and
- trend window, which presents continuously tracked process values on a graph.

The main window usually has multiple sections in it. Depending on the flavor, the main window might contain different sections and button layouts due to the original DCS button layout. In addition, the size and orientation of the main window varies depending on the flavor. Some flavors do not have a main window, but rather open the picture window with a default starting display on startup. These flavors have navigation and other functionalities built into the menu bar.

In NAPCON Generic simulator that is used as an example in this thesis, the most important navigational buttons are on the top part of the vertical main window. These buttons are used to open new windows, hierarchy window, favourites window, the user login window and operating guides. Below the buttons are active alarms that have different colours depending on the alarms level of criticality. Below the alarms are buttons that are used to either open the alarm window or mute all alarms. At the bottom are general information of the current simulation that is run. The main window of NAPCON Generic simulator is shown in figure 2.

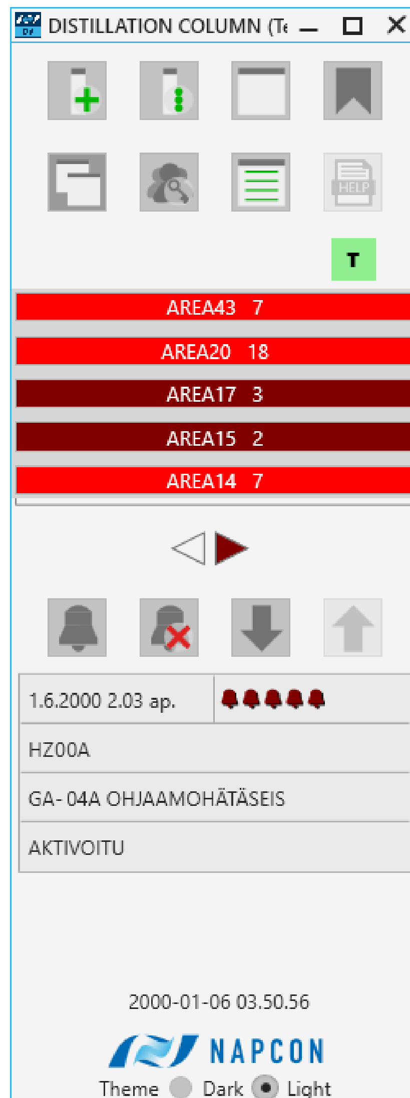

Figure 2. DV5 main window.

The hierarchy window is used to navigate the tree structure of the simulator process. The displays that are higher in the tree show more of the process area than the displays that are lower in the hierarchy. The lower the hierarchy goes, the more detailed the displays become. For example, third level of a display might show the process of only one equipment and only little of the surrounding process. The hierarchy window is shown in 3.

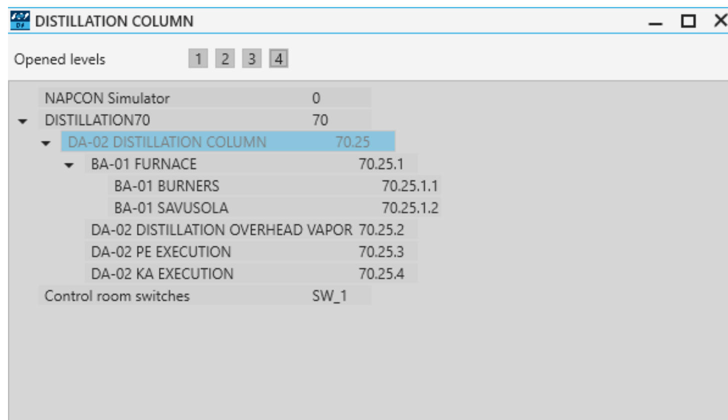

Figure 3. DV5 hierarchy window.

The picture window is used to show the various process areas that hold the most important process information. Depending on the hierarchy level, a number of interconnected process equipment and their current values are shown. The trainee can access these equipment via the picture window and manage their state. The picture window is shown in figure 4.

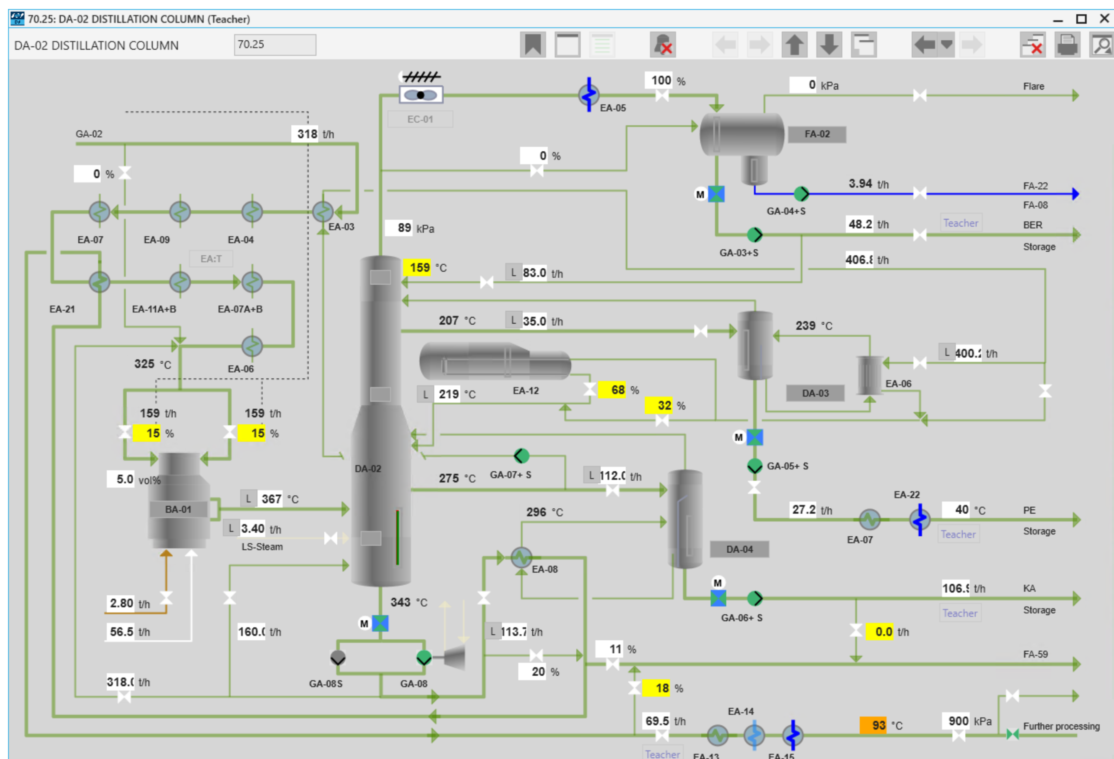

Figure 4. DV5 picture window.

Faceplates or loop windows are smaller windows that are opened by clicking an equipment value or icon and through faceplates the user can gain more detailed information about the state of the equipment. The information that is shown in faceplates includes value range for the equipment, current value, setpoint, output range, output, control mode and vertical bar visualising the value, setpoint and output. Modifying the equipment's state and setpoint is made possible through faceplates.In addition, faceplates can include a teacher expansion that allows further controlling and faulting of the equipment for training purposes. The expansion is not visible or controllable for a trainee. An example faceplate is shown in figure 5.

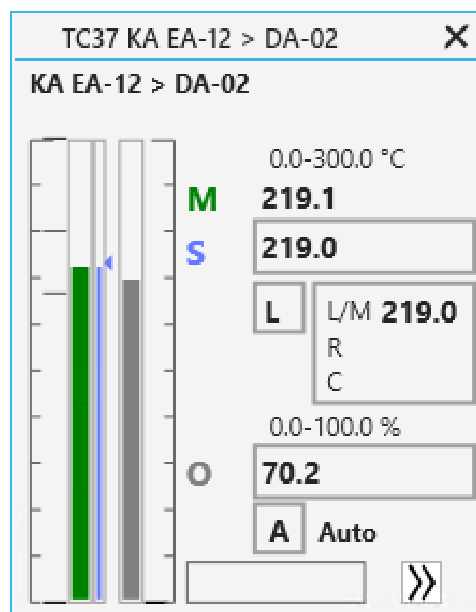

Figure 5. DV5 faceplate.

Alarm window is used to show current active alarms in the process. Alarms are triggered when a certain equipment value exceeds defined limits or the equipment's fault state is set to faulted in database. Alarms have a priority value that marks its severity or criticality. The number and color of bell symbols are the severity indicators. The alarms can be acknowledged to mute the alarm sound and then the cause of the alarm can be tracked and fixed. A teacher can fault any equipment by hand in a training session to create different training scenarios for the trainee. The alarm window is shown in figure 6.

60.11: Alarms Display (Teacher)

Alarms Display 60.11

UTC   Automatic scrolling

Time	Priority		Tag	Tag description	Event	nbr.	Area	Acknowledge	Removed
02:00:00:000		P	ESD02_M1	BA-01 PILOTK FLUSH		700	M1	02:02:10:000	
02:00:00:000		P	ESD03_M3	MAIN BURNER PROTECTION		700	M3	02:02:10:000	
02:00:00:000		P	ESD03_M4	MAIN BURNER PROTECTION		700	M4	02:02:10:000	
02:00:00:000		P	ESD05_M2	GA-71005 + EOV71105		700	M2	02:02:10:000	
02:00:00:000		P	FZ42_LL1	BA-01 COMBUSTION AIR		700	LL1	02:02:10:000	
02:00:00:000		P	HC15_M1	FLUSH OF PILOT LINE		700	M1	02:02:10:000	
02:00:00:000		P	HS05A	BA-01 PILOT NAT.GAS/FUEL GAS		700	HS05A	02:02:10:000	
02:00:00:000		P	HS05A_M1	BA-01 PILOT NATURAL GAS/FUEL GAS		700	M1	02:02:10:000	
02:00:00:000		P	LCA18	EA-12 STEAM GENER.		540	LCA18		05:21:22:120
02:00:00:000		P	LDA11	MEASUREMENT DIFFERENCE LI11A		700	LDA11		06:05:04:120
02:00:00:000		P	LDA30	MEASUREMENT DIFFERENCE LI10A		700	LDA30		
02:00:00:000		P	PZ33_HH2	BA-01 COMBUSTION AIR		700	HH2	02:03:24:000	
02:00:00:000		P	TDA04	MEASUREMENT DIFFERENCE TIA04A, B, C		700	TDA04	02:03:24:000	
02:00:00:000		P	TDA103	MEASUREMENT DIFFERENCE TIA103A, B, C		700	TDA103		
02:00:00:000		P	TIZ02B	BA-01 TUBE 1 SKIN	MITT. > YLÄRAJA	90	TIZ02B		
02:00:00:000		P	ZZ01	FUEL GAS > PILOT HAND VALVE		700	ZZ01		
02:29:38:080		P	LDA15	MEASUREMENT DIFFERENCE LI15A		700	LDA15		04:21:18:080
03:02:20:080		P	LDA20	MEASUREMENT DIFFERENCE LI20A		700	LDA20		04:21:08:080
04:29:08:120		P	TCA68	COMBUSTION AIR >GB-01		540	TCA68		04:31:22:120
14:32:12:330		P	TIA11	BA-01 TUBE 1 SKIN	MITT. > YLÄRAJA	90	TIA11		14:32:56:330
15:34:42:340		P	PC35	ÖP > WASTE	MITT. > YLÄRAJA	90	PC35		15:43:34:340
15:34:44:340		P	FC28	ÖP EA-21 > EA-13	MITT. < ALARAJA	90	FC28		15:43:22:340
15:34:54:340		P	TCA60	ÖP EA-15 > WASTE	MITT. < ALARAJA	90	TCA60		15:45:24:340
15:43:48:340		P	PC35	ÖP > WASTE	MITT. < ALARAJA	90	PC35		15:45:12:340
15:47:08:340		P	TCA60	ÖP EA-15 > WASTE	MITT. > YLÄRAJA	90	TCA60		16:47:50:340
16:11:22:340		P	FMS11	BA-01 FEED TUBE 1		700	FMS11		16:53:44:340

Figure 6. DV5 alarm window.

Trend window is used to present process data of the selected equipment. Certain measurements of multiple equipment can be shown and compared in the same graph. In addition, multiple measurements of a single equipment can also be drawn in the same graph. The trend window and the chosen measurements are continuously updated if a simulation is running in ProsDS. Thus, the trend lines are being drawn in real-time, which makes it easier for operators or trainees to track how the equipment is working. The trend window can also display history data for the equipment and their measurements even though ProsDS simulation is not running. Through trend window, the trainee can add or remove measurements from the graph and change linestyles and other visual elements of the graph. The trend window is shown in figure 7.

Figure 7. DV5 trend window.

All of the windows can be opened concurrently and used across multiple computer monitors. Commonly an operator has up to six monitors to be used for different windows. Faceplates have no limit as to how many can be open at the same time. This way all the necessary information is available quickly for the operator.

2.4 Training setup

A training session can be arranged with one teacher handling the trainings of multiple trainees. Every trainee has their own role in the training simulation and that role mimics the role that they would have in an actual work environment. In reality, each operator is responsible for certain processing unit within the plant's process. During training sessions, the teacher imposes faults in to the process or equipment malfunctions to trigger an emergency situation. Every trainee will then work together to solve that situation and handle the emergency in best possible manner.

The teacher can launch a training session by using NAPCON Simulator Trainer Dashboard. Before the session begins, the teacher must configure certain settings. Lesson subject indicates what type of emergency or other situation will be practised during the training simulation. The teacher will add the participating trainees to the list and manage the settings for them. The objective that is added for each trainee tracks and saves the progress of their training program. Different training programs exist for operators of different skill level and work history. In addition, for each trainee, a workstation and processing unit is assigned. The overview of NAPCON Simulator Trainer Dashboard is shown in figure 8.

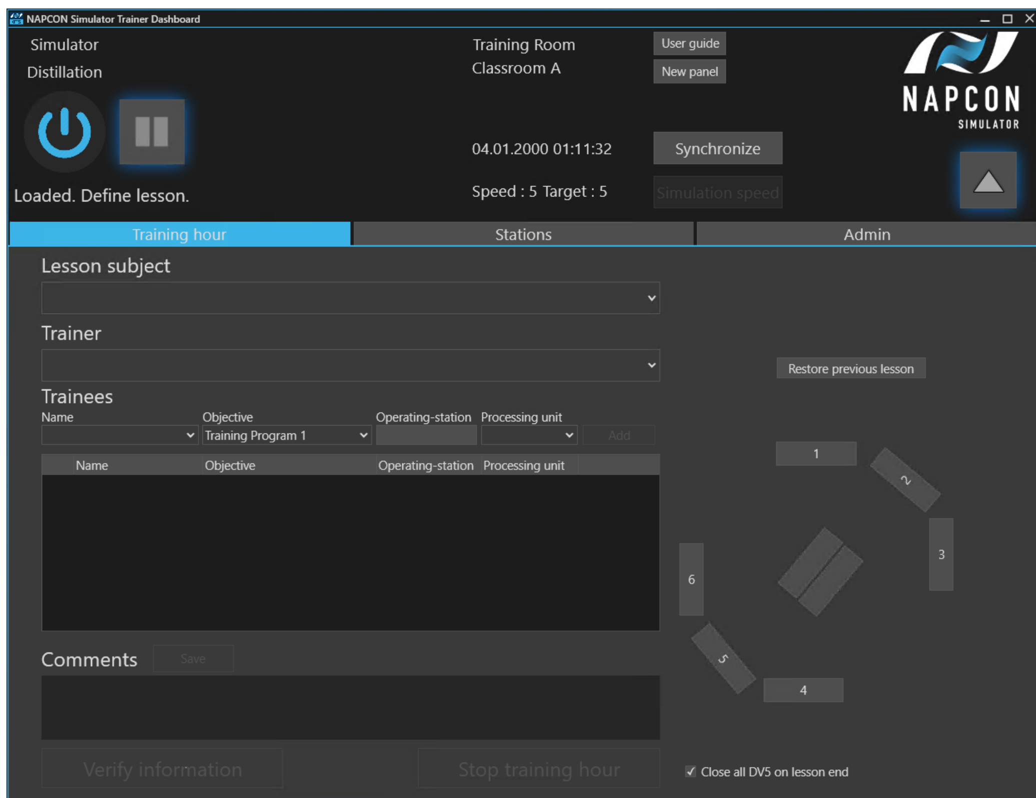

Figure 8. NAPCON Simulator Trainer Dashboard.

After the settings have been configured, the teacher can launch the simulation. This will cause DV5 to be opened in each workstation with a common simulation running for all of the stations. The training data will be recorded and can be evaluated after the training has finished. The teacher can also give comments for each trainee on how they managed the training. The connection between the Trainer Dashboard and other parts of the simulator are shown more closely in chapter 4.

3 Software architecture

This chapter discusses the basics of software architecture as well as the various architectural styles, design methods and evaluation methods. A literature review was conducted for both operator training simulators and software architecture. More literature was found in software architecture field that is reviewed in this chapter.

Software architecture has become its own area of study relatively late, only at the end of 20th century. Before that, software architecture was considered only high level design. Now software architecture's significance as part of software engineering is constantly growing with new architecture related books being published. (Koskimies and Mikkonen 2005)

One of the first ground breaking architectural studies was carried out by Dijkstra (1968) in his research paper named "The structure of the 'THE'-multiprogramming system". He wanted to divide the computer's inner architecture into layers that had independent areas of responsibility and functions. The interoperability of these layers decided on how user commands were executed. His layered architecture also enabled multitasking, meaning that multiple processes were executed simultaneously within the computer. It was not the first computer to enable multitasking, but it was the first one to do so with layered architecture.

In Dijkstra's layered structure, the higher layers are only dependent on the lower layers. The layers were developed in ascending order with layer 0 being the first one. This allowed incremental development and testing of the system. The layers were as follows (Dijkstra 1968):

- Layer 0, which was responsible for enabling multiprogramming and allocating processes to processors.
- Layer 1, which was responsible for allocating memory for processes.
- Layer 2, which was responsible for the communication between the operating system and console.
- Layer 3, which managed the I/O of the attached devices.
- Layer 4, which consisted of users' programs.
- Layer 5, which was the user itself.

Another revered study in software architecture was written by Parnas (1972), who published his article "On the criteria to be used in decomposing systems into modules". Parnas believed that modularization as a mechanism improves flexibility and understandability in systems, and simultaneously shortening its development time. The author also notes that the effectiveness of modularization is dependent on the criteria that are used to divide the system into modules.

According to Parnas (1972) a module is a responsibility assignment rather than a subprogram. This paragidm has guided the advancement of software architecture, because a module can be built on multiple parts that work on the same assignment and thus are under the same module. Parnas also mentions that the division into modules can be done in many ways, but the most effective way is to make modules independent and hide the implementation from other modules. This would mean that changes in one module would not affect the functionality of other modules. These types of independent modules are very normal today in most of software.

3.1 Definition

According to IEEE's standard, software architecture is defined as "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution." (”IEEE Recommended Practice for Architectural Description for Software-Intensive Systems” 2000)

Koskimies and Mikkonen (2005) note that the definition does not merely apply to the system's division into separate components, but also the connections between the components and how the components should evolve. The authors add that architecture concerns both structure and behaviour, because most connections between components have behaviour that manifests during runtime of the software. Architecture does not apply to only software's static code structure but also software's runtime structure, for instance dynamic object structures.

Architecture of a software is often viewed from a certain point-of-view, for example from file structure, logical structure or process structure perspectives. Architecture is also considered

to hold reasoning of certain solutions and not only description. In addition, architecture often concerns a set of rules and principles that regulate how systems are developed to fit the said architecture. (Koskimies and Mikkonen 2005)

While constructing a system according to certain architecture, there are rules that guide which technologies can be used, which algorithms are chosen, what data structures should be utilized and which design and implementation models should be used. Therefore, software architecture can be considered to be a system's fundamental law. (Koskimies and Mikkonen 2005)

Bass, Clements, and Kazman (2003) state that "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." This definition is similar to IEEE's definition, but does not consider the design aspect of software architecture.

According to Clements et al. (2010) "The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both." This implies similarly to Bass, Clements, and Kazman (2003) that every software has structures that are chosen and designed by a software architect and are used to achieve the system's design goals. The structures are also used to understand the underlying architecture. The structures consist of software elements and their relations as well as their properties, which implies that they should be documented to describe a software architecture.

Bosch (2004) concludes in his study that "... software architecture is, fundamentally, a composition of architectural design decisions. These design decisions should be represented as first-class entities in the software architecture and it should, at least before system deployment, be possible to add, remove and change architectural design decisions against limited effort." This concurs with IEEE's definition of software architecture that includes the design aspect in their definition, but focuses more on the design decisions as the main component. This means that design decisions determine the shape and functionality of the components that form a software's architecture.

Architecture defines the system's core, which stays unchanged during development and maintenance. Components that are not part of the core can be freely modified to their appropriate form. This mindset applies well to product-line architectures. The modifiable parts are application specific and can be customized as needed. This definition is common, because it extends to architectures where the software description is not the most important part but may describe a procedure which dictates how a software is to be modified. For example, some operating system may demand that all resources are visible to application programs through servers that manage them. These types of principles are called architectural philosophies. (Koskimies and Mikkonen 2005)

3.2 Architecture's objective

Every software has an architecture. Even if no one designed or documented it, it has been created during software development. Koskimies and Mikkonen (2005) cite Bass et al. (2003) who state that there is no absolute good or bad software architecture, but more or less suitable architecture for its designed use. For example, short piloting system does not require scalable and easily maintained laborious architecture.

Koskimies and Mikkonen (2005) cite Clements et al. (2002) who establish that a system's implementation and operation can be divided into parts with help of software architecture. This division enables the system's different parts to be developed simultaneously. Therefore software developers from different organizations and from different countries can simultaneously work towards a common goal without being dependent on others' work. Interface design that is done with help of architecture is crucial in order to specify how data is transmitted between the system's or application's different parts.

Koskimies and Mikkonen (2005) cite Bosch (2000) who proposes that good software architecture design can reduce the costs that arise from software development, enhance the software's continual development and maintainability, and reduce the time spent on developing the software until it is released. For example, architecturally badly defined interface can cause delays to development work in a situation where the interface's developer and user implement the functionality simultaneously. Bosch also states that only after the 1990's

has there been an understanding of how software architecture affects software quality and software development times while software sizes are constantly increasing and systems are becoming more complicated.

3.3 Software architecture description

Software architecture is the most important information that characterizes software. For this reason, communication between different stakeholders in software often involves architectural issues. In order for everyone to have the same understanding of the software architecture, the architecture should have the most comprehensive and unambiguous description possible. Only on the basis of such a description is it possible to express precise facts about the architecture and the system as a whole. Architecture thus has an important role as a software artifact enabling intelligent communication, providing key software solutions and a concept and vocabulary based on which the system can be spoken of. The importance of architectural descriptions is also understood in the industry and has become a key document in software development processes. However, the description of software architectures is still a relatively new and evolving area of software engineering. (Koskimies and Mikkonen 2005)

Software architecture should specify what kind of components there are in the system and what kind of relationships they have with each other. However, due to multiple relationships the components have, it is difficult to know from such a general definition what things should be included in the description of the architecture and what structure the description should have. For this purpose, the concept of viewpoint is introduced: the architectural description consists of views of the system according to certain viewpoints. (Koskimies and Mikkonen 2005)

Viewpoint refers to a general, system-independent way of describing a particular architecturally relevant feature of software. A view is an actual system-dependent description that follows some aspect. The relationship between viewpoint and view is similar to, for example, a class and an object. A view is an instance of viewpoint in a particular system. The viewpoint is orthogonal to the structuring of the system, meaning that any part of the system

can in principle be viewed from any viewpoint, although some perspectives may in practice be more meaningful in certain use. (Koskimies and Mikkonen 2005)

The 4 + 1 model by Kruchten (1995) presents five views for describing software architectures. These views are logical view, process view, development view, physical view and scenarios or use cases. Different stakeholders might view the architecture from a different viewpoint to gain important information of it. The logical view concerns the functionality of the system and how end-users might use those functionalities. Process view deals with dynamic aspects of the system, e.g. how processes communicate with each other. Development view concerns the software management and thus is the most important viewpoint for software developers. Physical view concerns the physical aspect of the system and is most useful for system engineers. Lastly, scenarios and use cases are the fifth view that are used to validate the architecture through exploring interactions between components and processes.

3.4 Design patterns

Design patterns are descriptions of known and proven solutions to common software design problems in specific situations. Thus, the design model has three essential parts: the problem, the context, and the solution. (Koskimies and Mikkonen 2005, p. 102)

The key parts of the design pattern are (Koskimies and Mikkonen 2005, p.105):

- Problem. The problem solved by the design pattern must be a general design problem that does not require, for example, a specific programming language. The problem occurs repeatedly in a wide variety of systems. Problems other than architecture related or detailed design are excluded.
- Context. The problem manifests itself in the broader context defined by the design pattern. The connection tells you in what situations the design pattern is applicable. The context also determines the requirements for the solution. The requirements are usually related to quality feature that the solution aims to improve.
- Solution. The solution must also be general and can be described by well-known formalisms (e.g. UML). The solution meets the requirements, but may lead to a deterioration of some other quality features. For example, the application of some design

patterns often degrades performance slightly, but not to the extent that it is relevant.

A design pattern always applies to several program units (components, interfaces, classes, methods) that are arranged in a certain way in the solution. The design pattern thus defines the relationships that the components associated with the solution have with respect to each other for this design model. The same unit may be associated with several versions of a different design pattern. A design pattern can be considered as an aspect-like, complementary, cross-sectional structure of a system component that is essential for understanding certain solutions in a system. (Koskimies and Mikkonen 2005, p. 105)

3.5 Architectural styles

Architectural styles are basically a generalization of the idea of design patterns as the underlying principle of system-wide architecture (Shaw and Garlan 1996). In fact, it is not always quite clear when a particular solution principle is a design pattern and when an architectural style, especially when a design pattern - such as the Observer Model - is generalized as the basis of architecture. Although this usually does not matter very much, the fundamental difference can be considered that the design pattern often appears in the system in many instances, solving different local design problems in a uniform way, while the architectural style determines the overall structure of the system. (Koskimies and Mikkonen 2005, p. 125)

Often, the architectural style is used to help perceive the actual system. In this case, in a system according to the architectural style, a number of components play the same role in terms of style. Therefore, these architectural styles can be understood as an explanation of the structure of the actual implementation but also as a grouping or perception technique. The main architectural styles used for grouping are layered architectures and pipes-and-filters architectures. From these two, layered architecture in particular can be used to describe almost any system. (Koskimies and Mikkonen 2005, p. 125)

3.5.1 Layered architecture

The layered architecture consists of layers arranged in ascending order according to some abstraction principle. Thus, the basic idea of layered architecture is that a component or

individual service at a certain level is implemented using components or services provided by a lower layer. For various reasons, this basic idea usually has to be deviated from, and pure layered architecture is quite rare. There are two types of deviations: a service call can pass from a lower layer to an upper one (hierarchy breach), or a service call can bypass layers as it travels from top to bottom (bridging). Bypassing layers is often necessary for efficiency reasons, as the service is often found to be more efficient on the lower layers. Bypass may also be necessary because the service in question is not directly available immediately from the lower level. Bypassing layers is not generally considered a serious deviation from layered architecture, although when used extensively, it can lead to the weakening of the idea of layered architecture. (Koskimies and Mikkonen 2005, p. 126)

Invoking from the lower to the upper layer is a serious problem with the layered architecture if it causes the lower layer to become dependent on the upper layer. In some cases it is necessary for the lower layer to invoke the upper. This happens in situations where the lower layer has to adapt its own service to the upper layer, and therefore, during its own service, invoke the code contained in the upper layer. This is a typical callback situation, and such call must be made according to callback principle so that the lower layer does not become dependent on the upper one. The lower layer provides some registration operation, which is then used by the upper layer to register its code to be used by the lower layer. (Koskimies and Mikkonen 2005, p. 127)

Layered architecture is a common model that can be applied to almost any system on a smaller or larger scale. At a high level, it divides the system into parts which make the system easier to understand and each part can also be understood as its own whole. The layered architecture is intuitive enough and easy to understand that it can be used to support communication when discussing a system with different stakeholders. Layered architecture guides software to minimize design dependencies, as layers ideally depend only on lower layers. This makes it easier to change and maintain the system. Each layer implements its own level of abstraction on which certain implementers, testers, and administrators can specialize. Examples of layered architectures are shown in figure 9. (Koskimies and Mikkonen 2005, pp. 130-131)

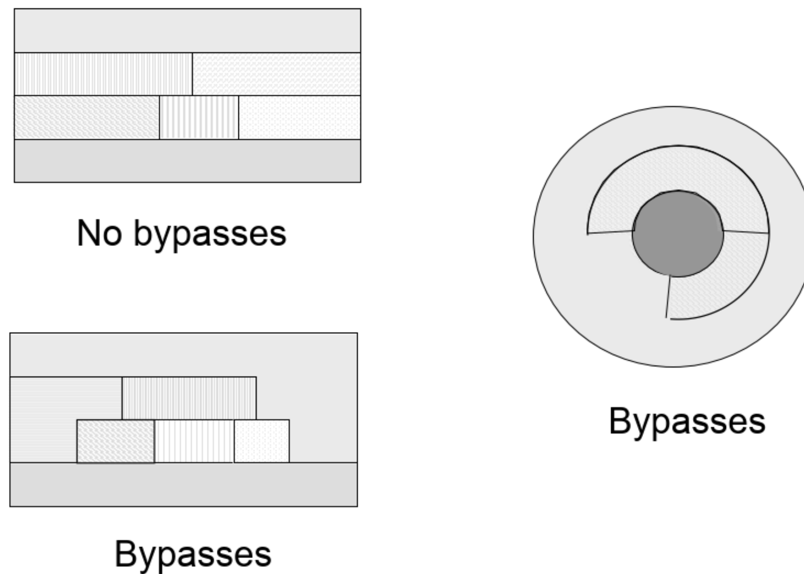

Figure 9. Graphical presentations of layered architectures. (Adapted from Koskimies and Mikkonen 2005, p.127)

3.5.2 Pipes-and-filters architecture

The pipes-and-filters architecture consists of processing units (filters) and connecting data-carrying buses (pipes), the roles of which are active data processing and passive data transport. Each processing unit operates independently by reading its own input stream and producing its own output stream. Combining the output stream of one unit with the next input stream provides aggregated data stream processing. The most obvious example of applying this style is Unix-enabled transfer of process results as input to the next process using pipes, but the style also has other applications (Koskimies and Mikkonen 2005, p. 132)

The application of a pipes-and-filters architecture requires that each processing unit can be implemented as an independent unit that reads its own input and produces its own output, independent of other units. These units should not have shared status information and should not have knowledge of the other units, therefore they only depend on the format of their own input. In addition, the processing should take place in one step so that the processing of a particular data item should not depend on the processing of any future data item. If it depends, the matter can be handled by creating an internal buffer in the unit, in which the data stream is read until the expected data element is obtained. After the wanted data element

is found, the data stream is read from the buffer. Such an arrangement violates the basic idea of the architecture, making it more difficult to sensibly parallelize processing units. The architecture is illustrated in Figure 10. (Koskimies and Mikkonen 2005, p. 132-133)

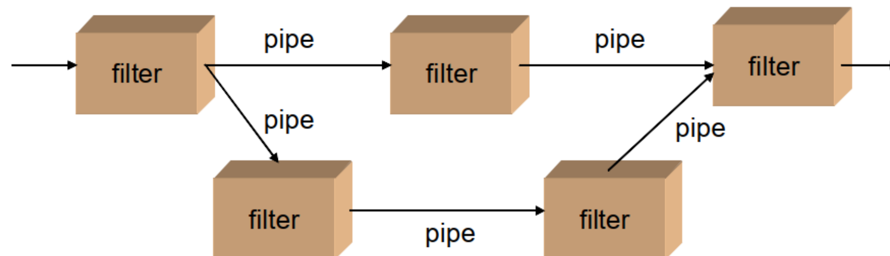

Figure 10. Pipes-and-filters architecture. (Koskimies and Mikkonen 2005, p.132)

The simplest and most common form of pipes-and-filters architecture is pipeline architecture, in which the data stream travels without branching along a single chain of processing units. In this case, the communication can be implemented synchronously in principle in two ways, either by pushing or pulling the information. In the push option, the original producer of the data first calls the first processing unit by entering the first data element as parameter. Based on this, the unit generates its own output element and passes it as a parameter to the next unit. Finally, the last unit calls the end user of the data stream as a parameter of the data item produced by the last unit. This is repeated until the entire data stream has been processed. (Koskimies and Mikkonen 2005, p. 133)

In the pull option, the end user of the data stream first requests a result element from the last processing unit. This in turn requests the element from the previous unit until the first unit requests the first element from the data source. The first unit receives it and generates its own output based on it, returning it to the previous unit that requested it, until finally the last unit can return its own output to the data stream user. This is repeated until the entire data stream has been processed. (Koskimies and Mikkonen 2005, p. 133)

3.5.3 Client-server architecture

Service-based architectural styles are constructed in a way that considered having two types of roles: service providers and their users. However, the roles are usually not strict, and a

service provider may act as a user of another service. The idea of a service is based on some resource that has a software component built around it and can provide the service to its environment. In practice, such a component can also monitor the use of a resource, which often serves as a selection criterion for this approach. Similarly, other resource-affecting functions, such as backup, are often easier to implement centrally. Most common service-based architectures are client-server architecture and message dispatcher architecture. (Koskimies and Mikkonen 2005, p. 136)

Client-server architecture is perhaps the most commonly used architecture solution at the moment. The basic idea is to encapsulate resource management (server) at a certain architectural level so that resource users (clients) do not have to deal with technical aspects related to resource use such as exclusion, but can request a particular resource-related service from the server independently of other clients. Therefore, the client-server architecture can be thought of as an architecture-level solution corresponding to the object paradigm. An example of client-server architecture is presented in figure 11. (Koskimies and Mikkonen 2005, p. 136)

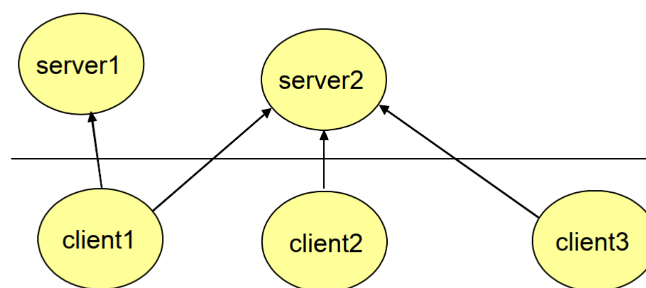

Figure 11. Client-server architecture. (Adapted from Koskimies and Mikkonen 2005, p.137)

Typically the interaction between the client and the server takes place within a session. During the session, a service entity that is meaningful to the client is performed. Typically, servers wait inactive until a client contacts them. Once the client has completed their task, it ends the session. The session can include transactions, the integrity and cancellation of which the server takes care of as needed. Communication between the client and the server is often more precisely regulated than between individual objects. In addition, the server always runs in its own thread or process, which keeps its implementation separate from the clients. If the server load greatly increases and slows down the operation of applications, the

internal implementation of the server can be changed to multi-threaded or multi-processing and thus increase capacity. (Koskimies and Mikkonen 2005, pp. 136-137)

3.5.4 Message dispatcher architecture

Message dispatcher architecture (also known as implicit invocation architecture or message bus architecture) refers to an architecture in which a number of components communicate with each other through a centralized messenger or bus. The key difference to a client-server architecture is that roles are not attached to the messaging architecture. (Koskimies and Mikkonen 2005, pp. 139)

In message dispatcher architecture, the components have a common interface that includes the necessary operations to receive messages. The message contains information that tells the component what to do. This can be considered a dynamic interface where a message of a new type does not change the static structure of the system, but a new component can handle it and bring substantially new functionality to the system. (Koskimies and Mikkonen 2005, p. 139)

The implementation of message dispatcher architecture can be constructed so that the components register with the broker, telling them they want to receive certain types of messages, and the broker in turn delivers the messages to the components as they are sent, or using configuration files that allow messages to be routed. Sometimes different queues or mailboxes are also used. The operation is illustrated in figure 12. (Koskimies and Mikkonen 2005, p. 139)

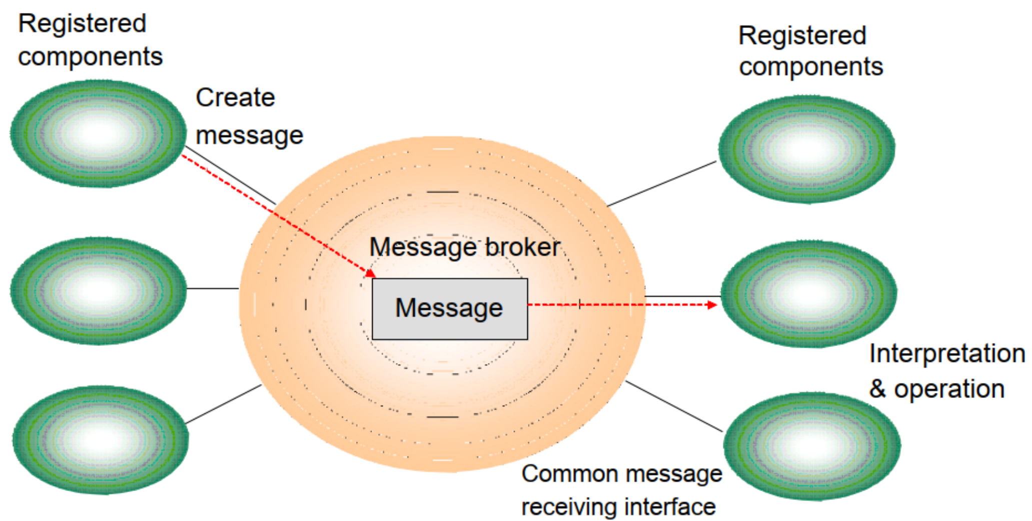

Figure 12. Message dispatcher architecture. (Adapted from Koskimies and Mikkonen 2005, p.140)

The message dispatcher architecture is defined by the following features (Koskimies and Mikkonen 2005, pp. 139-140):

- A set of components that communicate with each other,
- messages that allow components to communicate without the sender of the message knowing where the message should be delivered or the recipient from whom the message originated,
- operations by which components respond to messages,
- rules for registering components and messages the system,
- rules that allow the broker to know which component the message must be sent and
- parallelism model: the extent of parallel operations of the components and the broker.

3.5.5 Model-View-Controller (MVC) architecture

According to Burbeck (1992) the model-view-controller paradigm was quite novel when Smalltalk-80 programming language was invented in the 1970s by Xerox PARC researchers. It was considered elegant and simple, but it was different from other paradigms that were used at that time. To use the MVC architecture efficiently, one must understand the division

of work between the three main parts and how they communicate between each other. The view manages the graphical output that is displayed to user, the controller interprets inputs given by the user and commands the view and model to change accordingly. Lastly, the model manages data and behaviour of the application domain, informs dependent parts of its state and changes its state when commanded to. The overview of MVC architecture is show in figure 13.

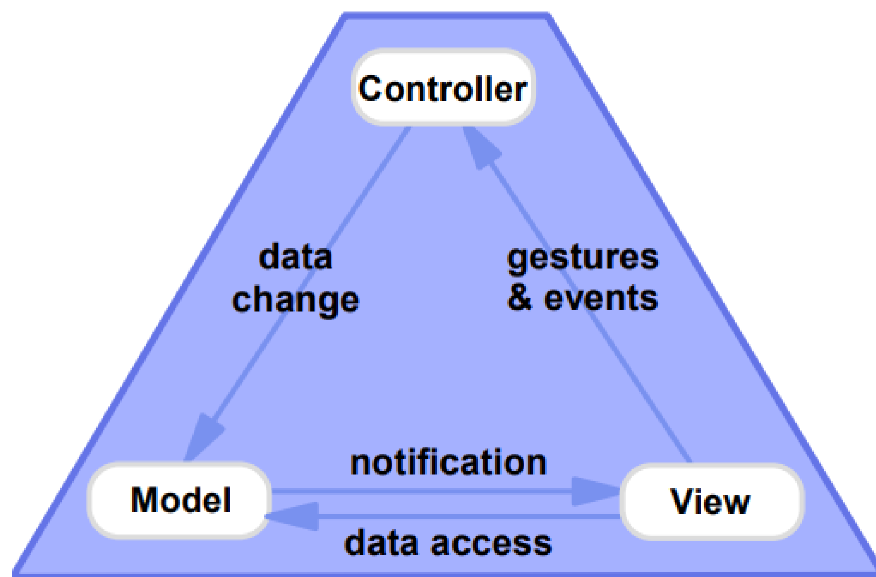

Figure 13. The overview of the model-view-controller architecture. (Potel 1996, p.1)

The basic idea of the model-view-controller (MVC) architecture is to separate the user interface from the actual application logic and data. The aim is to make the user interface modifiable and to make the system to be easily transferred to another graphical platform. In addition, it is desired that the user interface always reflects the state of the application data and displays it in the correct format in all various views. (Koskimies and Mikkonen 2005, p. 142)

According to Buschmann et al. (1996) it is beneficial to separate the model from the view and controller, because it allows multiple views to be created from the same model. A change in one view will also be updated to all other dependent views. In this case, the model notifies all dependent views if its data has been altered and all connected views receive the new data and update their information accordingly.

The system is divided into three types of parts: models that represent some part of the application data or the logical state of the application, views that represent some part of the visible interface, and controllers that act as adapters for the models and views, making sure they match. The model is responsible for managing the application data and to providing logical application operations that change this data. The model provides operations to observe its changes, and it notifies them when changes have occurred. The view ensures that the screen updates to match the status of the model. The same template can have several different views depending on the application. The controller receives user commands and converts them into logical application functions. The operation of the model-view-controller architecture is shown in figure 14. (Koskimies and Mikkonen 2005, pp. 142-143)

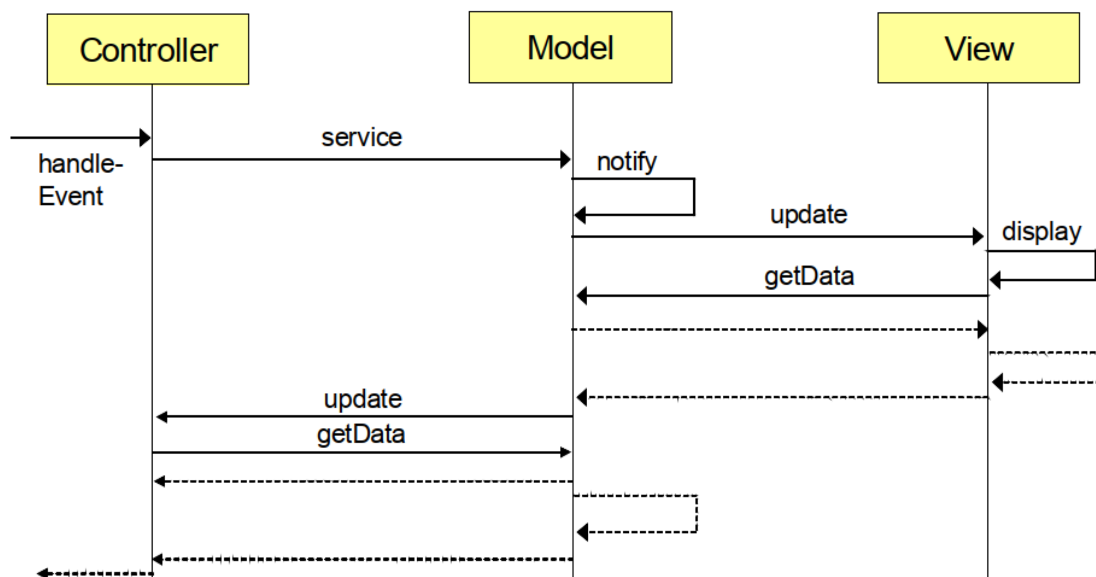

Figure 14. The operation of the model-view-controller architecture. (Koskimies and Mikkonen 2005, p.143)

3.5.6 Model-View-Presenter (MVP) architecture

MPV is an architecture refined by IBM's subsidiary Talingent. According to Potel (1996) the model-view-presenter architecture is a generalized form of MVC architecture. The model still deals with data management and the view deals with user interface. The view hands off events to the presenter that observes the model and allows the view to update itself. The

presenter in MVP architecture interprets the events and gestures that are initiated by the user, and provides business logic that maps the given input onto the right commands that change the model accordingly. When compared with MVC's controller, the presenter is elevated to application level and takes into account the interactor concept, commands and selections. An overview of MVP architecture is shown in figure 15.

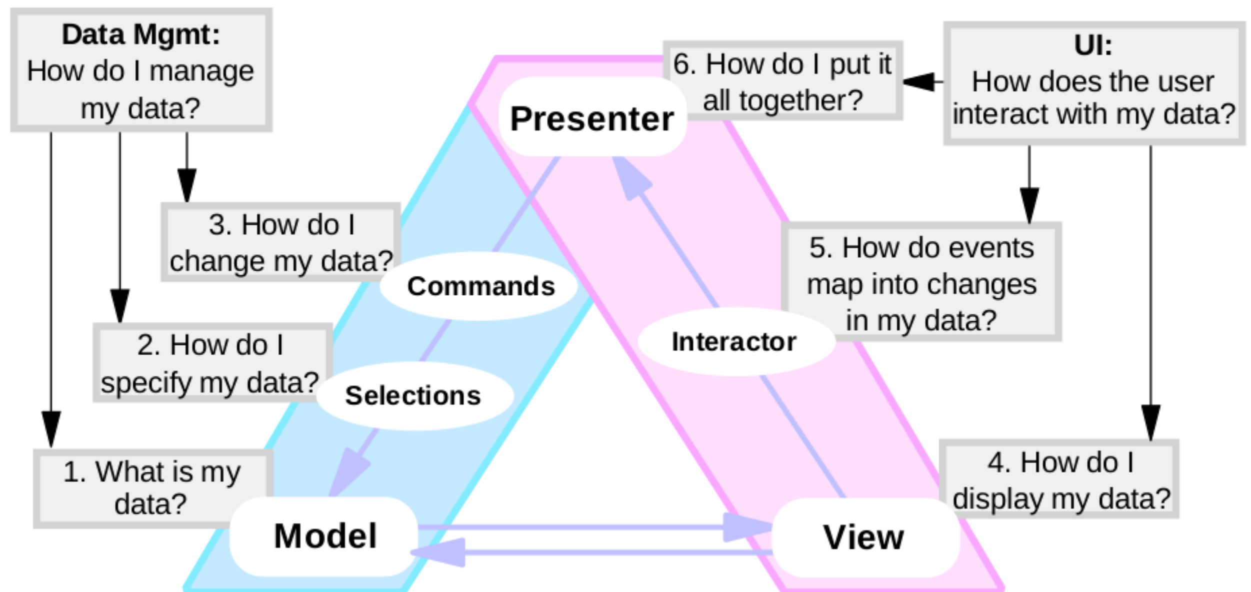

Figure 15. The overview of the model-view-presenter architecture. (Potel 1996, p.6)

3.5.7 Model-View-ViewModel (MVVM) architecture

Model-View-ViewModel architecture is a pattern first introduced by John Gossman (2005) in his Microsoft blog. Gossman states that MVVM is a variation of MVC that is suitable for modern UI development platforms where the view is often handled by a designer rather than a developer. Design is often done in either Hypertext Markup Language (HTML) or Extensible Markup Language (XAML), which are usually different languages than the ones that are used to handle business logic and backend data. In addition, different parts of the application are often developed by different people. Gosmann has since written more on MVVM architecture and gives an accurate description of it over multiple blog posts on Microsoft site.

MVVM has been compared to Presentation Model architecture in the past, as mentioned by

Smith (2009). Presentation Model architecture was introduced by Martin Fowler (2004) and it clearly separated a view from its state and behaviour similarly to MVP architecture. Thus, MVVM can be considered as a specialization of the Presentation Model pattern.

Unlike the Presenter in MVP architecture, viewmodel does not need a reference to the view. In MVVM, the view binds to the properties of viewmodel and the viewmodel exposes data that is contained in the model that affects the view. The binding of view and viewmodel is constructed by setting the viewmodel as DataContext of the view. The viewmodel is in charge of modifying the model data, whereas the view never does that. An overview of MVVM architecture is shown in figure 16. (Smith 2009)

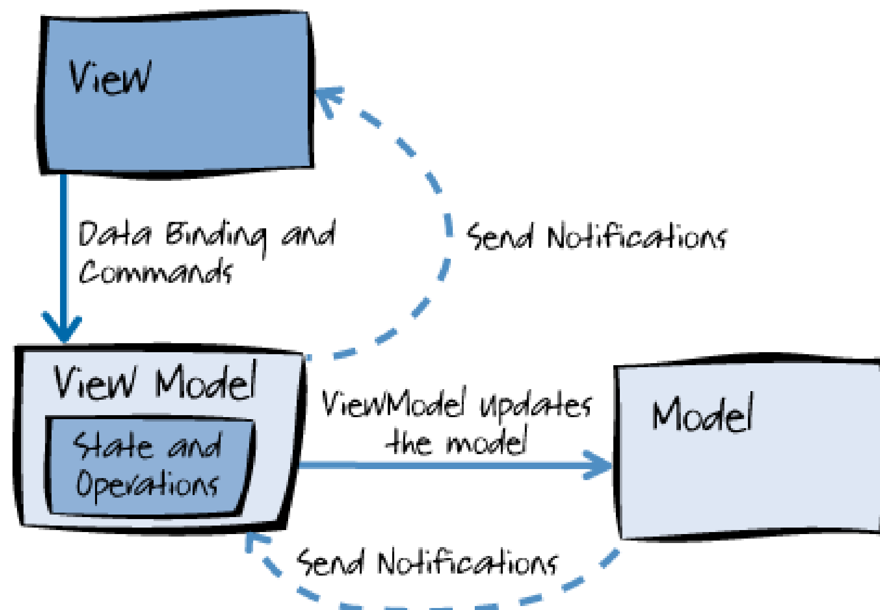

Figure 16. The overview of the Model-View-ViewModel architecture. ("The MVVM Pattern" 2012)

More closely, the view handles defining the layout and appearance of what is shown on the screen. Usually the view is coded in XAML and only little code-behind that does not contain business logic. The model implements the the application's domain model that contains a data model, business logic and validation logic. The viewmodel acts between the view and the model and handles view logic. ("The MVVM Pattern" 2012)

The main benefits of implementing MVVM architecture are the following ("The MVVM Pattern" 2012):

- Developers can focus on viewmodel and model components and designers can work independently on the view components without interfering each others' work,
- developers can create unit tests for the model and viewmodel components without affecting the view,
- user interface can be redesigned at any point without touching the code,
- existing implementations of model classes can be left unchanged to ensure correct operations when viewmodel acts as an adapter and can be modified more easily to satisfy the altered needs.

3.5.8 Repository architecture

In a repository architecture, a set of systems or components maintain a common state in a repository. There are different variations of the architecture depending on how active the data repository is. In the extreme case, the architecture can be thought of as resembling a client-server architecture in which a data repository management server provides data storage services to its clients. (Koskimies and Mikkonen 2005, p. 145)

The core of data repository architecture is a shared data repository that each component of the system can access and modify. The components do not communicate directly but through the data repository. The components attached to the data repository can operate in the same process or in different processes in parallel. Maintaining data consistency requires data repository support for the concept of transaction: a change event involving multiple data items is always performed completely or not at all. The system can be distributed or it can run on one machine. Typical examples of data repository architectures are integrated development environments where a common data repository consists of an internal representation of a program that is handled by various tools. (Koskimies and Mikkonen 2005, p. 145)

3.5.9 Interpreter architecture

In an interpreter architecture, an interpreter reads and executes a functional description according to a certain known format by utilizing the services of an execution platform. The latter can be a support software made for an application area that has an application program-

ming interface (API) that the interpreter relies on. An example of an interpreter architecture is a database management system that supports Structured Query Language (SQL). It can be used to make applications that are easily transferable from one database system to another. (Koskimies and Mikkonen 2005, p. 146)

Another typical example of an interpreter architecture is a scripting-enabled system in which a system user can write their own functional descriptions and execute them immediately within the system. Often Extensible Markup Language (XML) is used to present functional descriptions. Virtual machine-based programming systems (e.g., Java) can also be understood as applications of an interpreter architecture. Interpreter architecture is illustrated in figure 17. (Koskimies and Mikkonen 2005, p. 147)

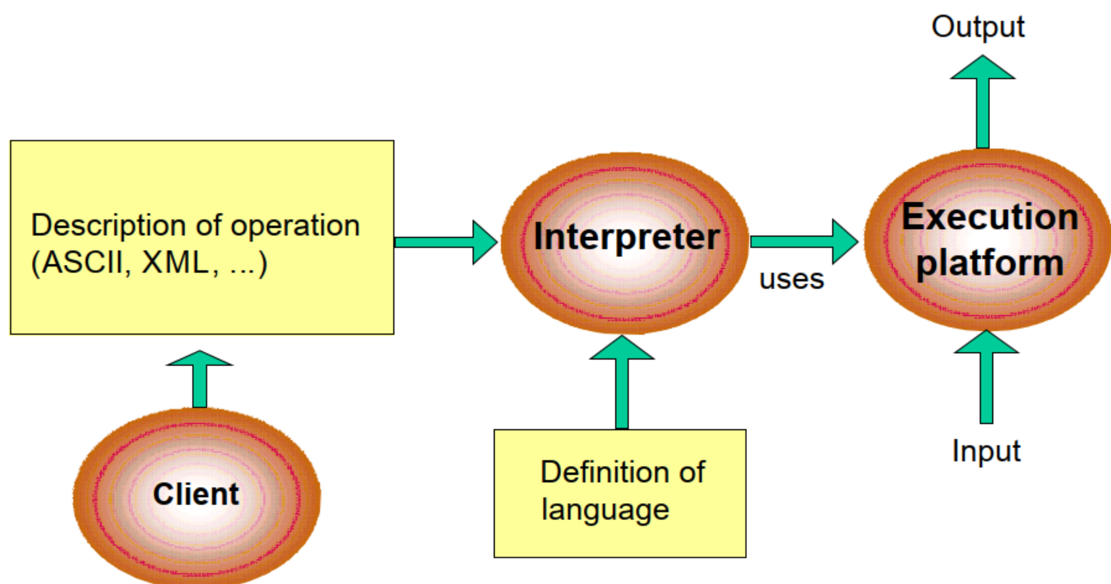

Figure 17. Interpreter architecture. (Adapted from Koskimies and Mikkonen 2005, p.147)

3.6 Product-line architecture

A key challenge in software development is software reuse: how to utilize the same components in multiple different software products. Software reuse has become a necessity in many areas, enabling the rapid development of new software products that meet high quality standards. (Koskimies and Mikkonen 2005, p. 157)

Product family is a set of software products with similar functions and structures implemented in a specific application area. Typically, a product family covers software products made by a particular company for its own area of operation. A product platform is software that implements the common structure and functionality of a product family. The architecture of a product platform (and its product family) is called product-line architecture (PLA). The product-line architecture is thus an enabling architecture for software products of a product family. (Koskimies and Mikkonen 2005, p. 160)

Modifiability and variance control are key issues in product platform development. Product platform usually has a software platform that is used in all products, as well as a number of components that are used in the product if needed. These components can be optional and do not need to be included in the product, but some components can be alternative, meaning that one of a certain set must be included in the product. Products often include unique product-specific components that are not used anywhere else. This procedure is shown in figure 18.

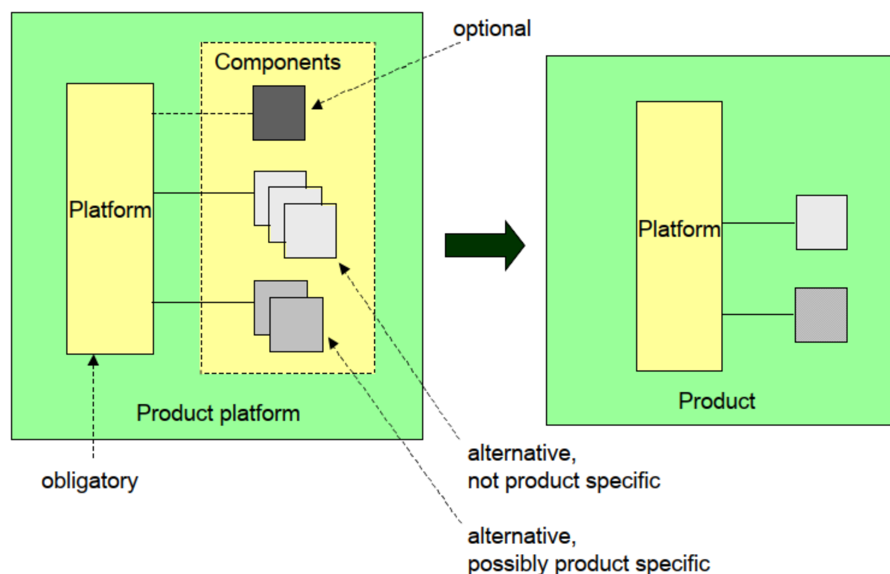

Figure 18. Product platform and a product. (Adapted from Koskimies and Mikkonen 2005, p.169)

Product platform enables reusing the basic structures of implementations, and yields the following benefits (Koskimies and Mikkonen 2005, pp. 160-161):

- Improves software quality, because the code has been tested repeatedly in various configurations,
- speeds up software development, because most of the code already exists,
- eases project management, because similar process models can be reused,
- eases moving from one project to another, because environment and tools are already familiar,
- standardizes products, because common things work in a same manner in different products, and
- enchances operations, because time consuming architectural design has already been completed.

3.7 Object-oriented framework

Software frameworks are an object-oriented way of implementing a product platform. Thus, frameworks have the same basic goal as product platforms, large scale and systematic reuse of software. Framework technology reuses not only a set of components, but also the architecture and basic functionality of software. This allows you to quickly produce quality software products. (Koskimies and Mikkonen 2005, p. 187)

An object-oriented framework is a collection of classes, components, and / or interfaces that implement the common architecture and basic functionality of a set of software. Such a set of software may be a set of applications of the same type for a particular application area (e.g., simulation applications), a set of applications with a particular infrastructure (e.g., distributed business applications), a set of applications with a particular type of graphical user interface, or a variety of component variations. (Koskimies and Mikkonen 2005, pp. 187-188)

The frame can be understood as a software frame that contains gaps known as hot spots. The framework itself is not an executable software, but the desired software is obtained by means of a software framework by filling in the gaps with new code that implements the specific functionality required by that software without changing the architecture. This is called specialization. If the specialization results in an independent application, the frame is

called an application framework. If specialization results in a component, the frame is called a framelet. A framework with specialization interface is shown in figure 19. (Koskimies and Mikkonen 2005, pp. 188-190) (Koskimies and Mikkonen 2005, p. 168)

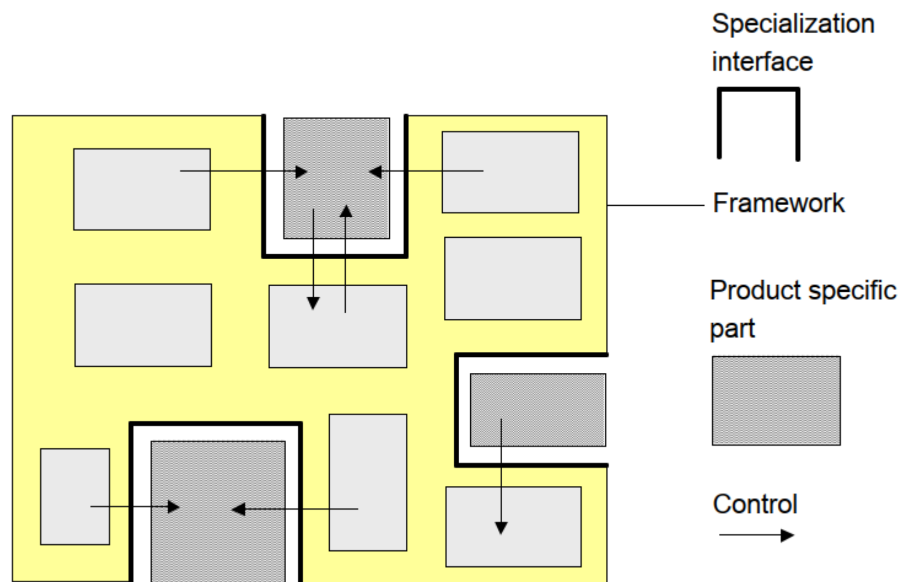

Figure 19. Framework with specialization interface. (Adapted from Koskimies and Mikkonen 2005, p.190)

3.8 Architectural design

Architectural design is commonly done before moving into detailed design and implementation of software systems. Architecture is based on functional and non-functional requirements imposed on the system. According to Koskimies and Mikkonen (2005) first version of architecture is usually based on the functional requirements and it is evaluated taking into account the non-functional requirements. Afterwards, the architecture can be modified to meet the non-functional requirements. After the non-functional requirements are met, the basic architecture is completed. After that, secondary functional requirements can be assessed and the architecture modified to accommodate those as well. If architecture design is done utilizing use cases, primary and secondary functional requirements can be acquired by dividing use cases into architecturally critical and less critical requirements. An architecture based software development process is presented in figure 20.

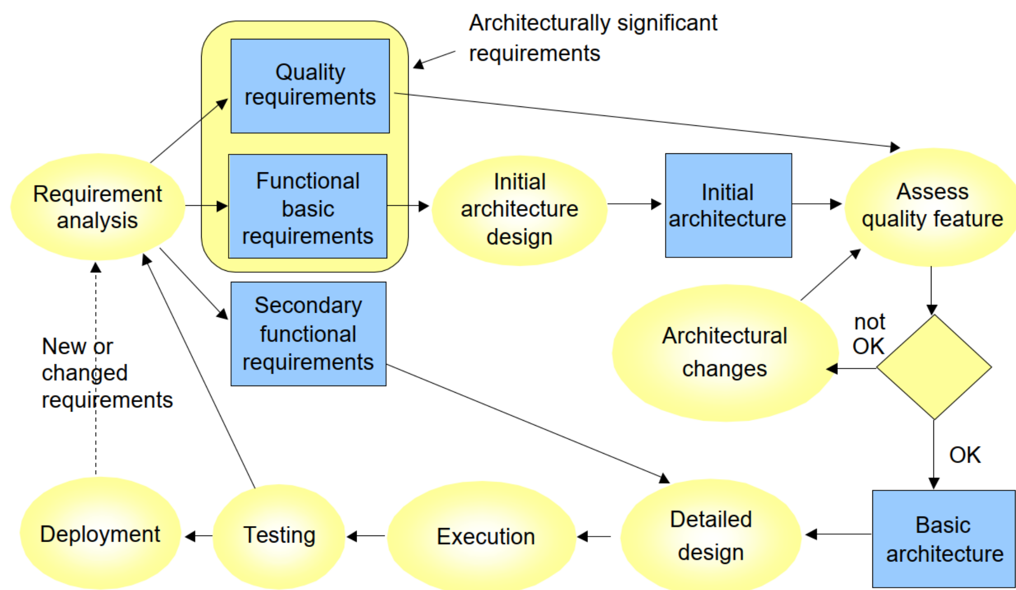

Figure 20. Architecture based software development process. (Adapted from Koskimies and Mikkonen 2005, p.22)

Hofmeister et al. (2007) studied five industrial software architecture design methods and compared their similarities. By identifying these similarities they propose a general software architecture design approach. The five selected design methods were:

- Attribute-Driven Design (ADD) method (Bass, Clements, and Kazman 2003),
- Siemens' 4 Views (S4V) method (Hofmeister, Nord, and Soni 1999),
- the Rational Unified Process 4+1 views (RUP 4+1) (Kruchten 1995, 2003),
- Business Architecture Process and Organization (BAPO) (America, Rommes, and Obbink 2003), and
- Architectural Separation of Concerns (ASC) (Ran 2000).

Some more detailed analysis of the different methods here

Hofmeister et al. (2007) note that the aforementioned methods were developed independently, therefore they use different vocabulary and seem quite different from each other. Some of the differences are essential since they are aimed to be used in different domains. However, all of the design methods also share a lot of common properties that can be identified and collected into one generalized model of architecture design method. Similarities

were analysed by comparing the artifacts and activities used in each method. The finding was that the five approaches have much in common and can be generalized into an ideal pattern. This pattern is shown in figure 21.

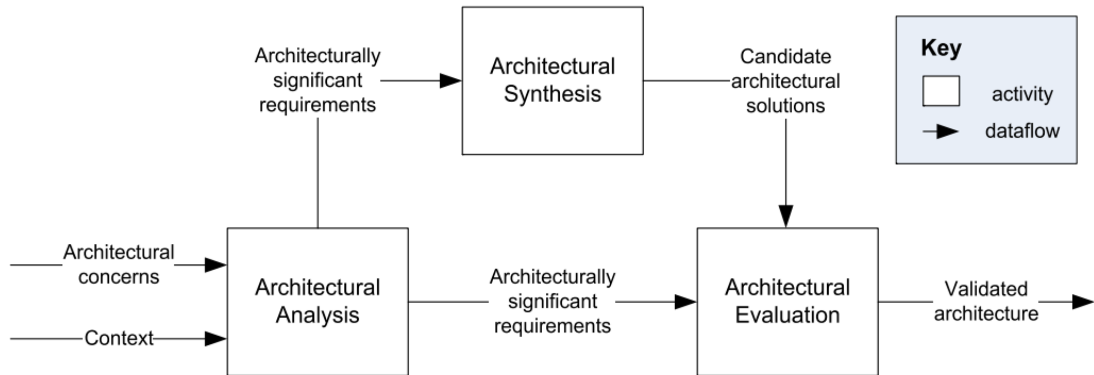

Figure 21. Architectural design activities. (Hofmeister et al. 2007, p.113)

Hofmeister et al. (2007) also found differences between the five methods. These differences include how certain method is intended to be used, what processes and perspectives are emphasized in the design, what are the driving forces of each method (quality attributes vs. functional requirements), architectural scope and process scope. This means that even though the architectural design methods can be generalized into one ideal method, the individual methods are designed for certain applications and thus are more suited to be used in certain situations.

3.9 Architectural evaluation

Architectural evaluation is an important part of the design process to gain valuable knowledge how well the designed architecture serves the software's needs. According to Koskimies and Mikkonen (2005) the evaluation of architecture differs from other technical reviews in that its quality is not based purely on technical quality during design, but also on its ability to meet the long-term objectives set for it. These objectives include expandability, modifiability, and scalability without sacrificing performance or memory consumption.

The architecture usually decides how well the software meets the quality requirements, and the architecture is often designed specifically in terms of the quality requirements. Thus,

the main focus in evaluating architectures is on quality attributes and non-functional requirements rather than functional requirements. (Koskimies and Mikkonen 2005, p. 222)

In order to comprehensively assess the qualitative characteristics of software, it is important that the architecture includes all or most solutions that affect the qualitative characteristics. This can be considered as a criterion of architectural perfection: if some qualitative property cannot be assessed on the basis of architecture, the architecture is deficient. However, this is not always the case, because the details of the user interface often have a significant effect on the usability of the system, which is a qualitative feature but cannot be seen at the architectural level. Similarly, the way the architecture is implemented can have a big impact on the efficiency of the system. It is essential, however, that the architecture allows the quality requirements to be met within the frames of known implementations. (Koskimies and Mikkonen 2005, pp. 222-223)

A system can be assessed by many quality requirements. Some general quality requirements include (Koskimies and Mikkonen 2005, p. 223):

- Performance: the resources consumed by the system to process a specific amount of data, transactions, or users,
- reliability: the ability of the system to remain operational,
- availability: relative portion of system uptime,
- security: the ability of the system to block unauthorized users without causing harm to legitimate users,
- modifiability: the ease of making changes,
- portability: how well the system supports its migration to different resource environments, and
- variability: how well the system has taken into account the variation of certain requirements.

In addition, certain other quality requirements might be considered for assessment. These requirements are:

- Usability: the ease of using the system as user,
- testability: how efficiently the system can be tested,

- safety: can the system be used without imposing risk of injury on the user,
- maintainability: ease of maintaining the system and keeping it operational,
- reusability: how well the software can be reused in other projects, and
- scalability: how easily the performance of the system can be increased without disrupting the system's operations, for example to serve more users concurrently.

Different architectural evaluation methods have been proposed to assess the state of architecture in software. These methods usually propose a process by following steps to complete an evaluation. According to Koskimies and Mikkonen (2005) these evaluation methods are aimed to give answers to following questions:

- Does the designed architecture suit the system?
- Which alternative architecture best suits the system and why?
- How good will a certain quality attribute be, assuming the system is implemented reasonably?

Most well known architecture evaluation methods are review methods that brings different stakeholders together to review the architecture and to refine it. These methods include:

- SAAM (Scenario based Architecture Analysis Method) (Kazman et al. 1994),
- ATAM (Architecture Tradeoff Analysis Method) (Kazman et al. 1998), and
- DCAR (Decision-Centric Architecture Reviews) (Heesch et al. 2014).

SAAM and ATAM methods, along with other scenario-based review methods have been studied by Babar, Zhu, and Jeffery (2004) and through gained knowledge they built a frame for selecting a suitable evaluation method. The authors focused on analysing the evaluation methods' processes, steps, objectives, the use of quality attributes, execution period, focus of evaluation, participation of stakeholders, tools support and resource requirements. ATAM was found to be of good maturity and offers detailed guidance for each step in the evaluation process.

Babar, Brown, and Mistrik (2013) state that scenario-based evaluation methods are more suitable for development-time quality attributes, such as maintainability and usability, whereas run-time quality attributes, such as performance and scalability, can be better assessed by us-

ing quantitative methods such as simulation or mathematical models.

Williams and Smith (2002) propose an architectural approach to fixing software performance problems. They describe a method called Performance Assessment of Software Architectures (PASA). PASA harnesses the principles and techniques of Software Performance Engineering (SPE) to assess if an architecture is able to support its performance objectives. The authors propose several techniques for analyzing the performance of a software's architecture (Williams and Smith 2002, p. 5):

- Identifying the underlying architectural styles,
- identifying the performance antipatterns, and
- performance modeling and analysis.

By identifying the underlying architectural styles or patterns, one can choose general performance characteristics of that style to test its performance. For layered architecture, high throughput situations might prove difficult since there is a lot of overhead as requests are passed between layers. If deviations from the architectural archetype are found, these deviations can be examined to determine if they have a negative impact on the software's performance. (Williams and Smith 2002, p. 5)

Antipatterns are similar to patterns, but their use knowingly produce negative consequences. Antipatterns help document common mistakes during software development process, which means that antipatterns help to avoid and fix problems when you find them. Similarly, performance antipatterns are used to document common performance problems and how they can be fixed. Antipatterns will be refactored to preserve correctness of the software, while transforming it into an improved version. (Williams and Smith 2002, pp. 5-6)

Performance modeling and analysis aims to quantitatively assess and evaluate the software or parts of it. A simple analysis of a software might be sufficient to identify problem areas. If the performance does not meet the requirements, a more detailed modeling and analysis can be done. These models allow architects to easily explore architectural options to overcome the problems. A software execution model and a system execution model can provide information for architecture assessment. Software execution model usually sufficiently identifies performance problems that are caused by poor architecture. The system execution

model is a dynamic model that takes into account different factors, such as multiple users, which can cause contention for resources. Solving the system execution model provides more precise metrics to be used in evaluation, an identification of bottleneck resources and comparative data for different options to improve performance by workload changes and software changes. (Williams and Smith 2002, p. 6)

In this thesis the architectural improvement is measured by running static code analysis by using Sonarqube. The focus of these architectural improvements is making the software more reusable and maintainable in structure, but the changes are also expected to improve performance. A clearer structure to the architecture will clarify which modules are to be reused in all existing and future DV5 flavors. The new structure is aimed to improve maintainability as well by lowering complexity and by dividing the code into independent components that can be tested and developed in isolation.

4 Current architecture

One research question of this thesis was to examine what is the current architecture of DV5. This examination will reveal important information and clarify the existing situation of the software. To fully understand DV5 as a software, it is beneficial to first understand the architecture of full NAPCON Simulator. After clarifying the architecture of the simulator, DV5 is taken under investigation. Scope of this thesis is to improve DV5, but the full simulator architecture can be explained. Improvements will not affect NAPCON Simulator as a whole, but some improvement between the different parts may be achieved.

As mentioned in chapter 3, software architectures can be described from multiple viewpoints. These viewpoints serve different stakeholders and aim to give valuable and usable information of the architecture. In this thesis, NAPCON Simulator architecture is presented from process viewpoint, which aims to describes how the simulator's different parts communicate with each other and how DV5 is connected to the whole simulator. On the other hand, DV5 software architecture description will be made from development viewpoint due to the fact that the architectural improvements aim to increase understandability and maintainability of the software, thus helping the development work and developers the most.

4.1 NAPCON Simulator architecture

NAPCON Simulator works by combining several software that run simultaneously on a local workstation. The necessary software are DV5, ProsDS and NAPCON Informer. These parts are needed to simulate an existing real world plant and its processes. These parts can be complemented by installing DV5 Teacher extension, Trainer Dashboard or Information Manager that are aimed to improve the the simulation handling and overall training experience. Overall architecture of NAPCON Simulator is presented in figure 22.

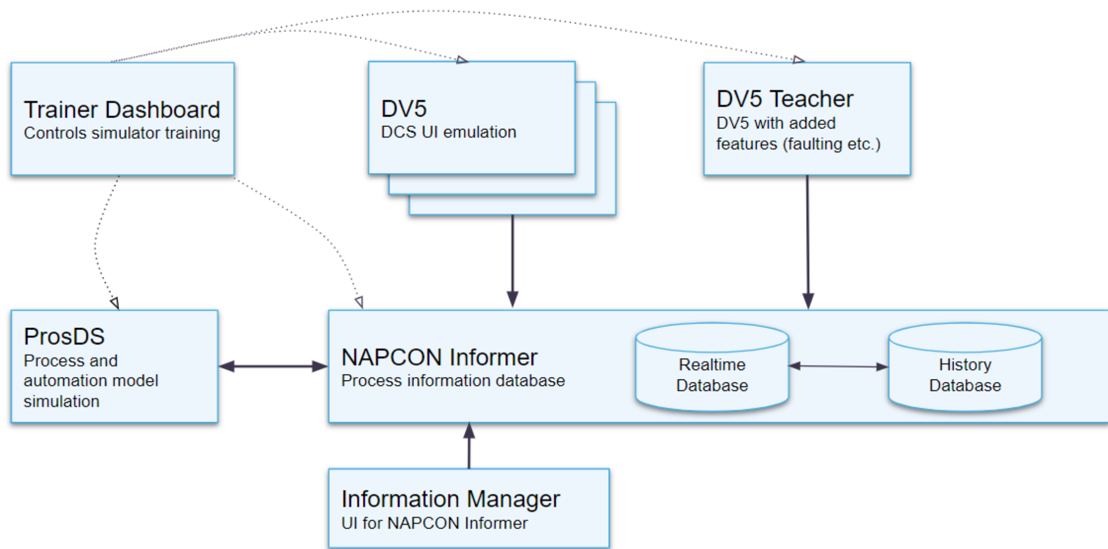

Figure 22. Architecture of NAPCON Simulator.

Trainer Dashboard is used to control the simulator training in every aspect. It has the ability to access and control DV5, as well as DV5 Teacher extension, ProsDS process simulation and NAPCON Informer database. This is a requirement in order to fully control simulator training events.

ProsDS process and automation model simulation has a two-way connection with NAPCON Informer process database. ProsDS has the ability to control values in the database due to the fact that any given process has fluctuation and changes during its uptime and ProsDS is simulating those changes as accurately as possible. ProsDS can change all the values for each of the equipment in the database. On the other hand, if values are changed in the database, then Informer will change ProsDS process calculations parameters. If ProsDS affects the values in the database, the changes are relayed forward to DV5 to be shown to the operator trainee.

DV5 user can see the state of the equipment and process in real time. Whenever a value is changed in the database, it can be viewed in DV5 displays. On the other hand, the user can modify the equipment and their values within certain limits. This means that DV5 has a one-way connection to the database. The database values are only shown on DV5 displays, but it cannot change anything within DV5. DV5 Teacher extension has similar capabilities

to DV5, but additionally teachers can change the database value more freely than the basic user can. The teacher can for example intentionally fault a valve in order to create an alarm and changes within the process.

Information Manager is the user interface that is used to control the Informer database. All data in the database can be viewed and controlled with Information Manager. This can be done in order to repair or develop the training simulator, or to be used in development work of DV5 in testing capacity. Information Manager is not a necessary part of the simulator, but is more focused on helping with the development work, database validation and verification.

4.2 DV5 architecture

The current architecture of DV5 is Model-View-ViewModel (MVVM) pattern. This is due to the technologies that are used to create it. As mentioned in section 3.5.7, the MVVM architecture is suitable graphical user interface software that is programmed by using C# and .NET technologies. In the case of DV5, the architecture has never been designed in detail, but has emerged during its multiple years of development. Additionally, no documentation of the architecture or its design processes can be found. This means that DV5 has been born naturally during its development. This is why mapping the current architecture is very valuable. The current architecture of DV5 is presented in figure 23.

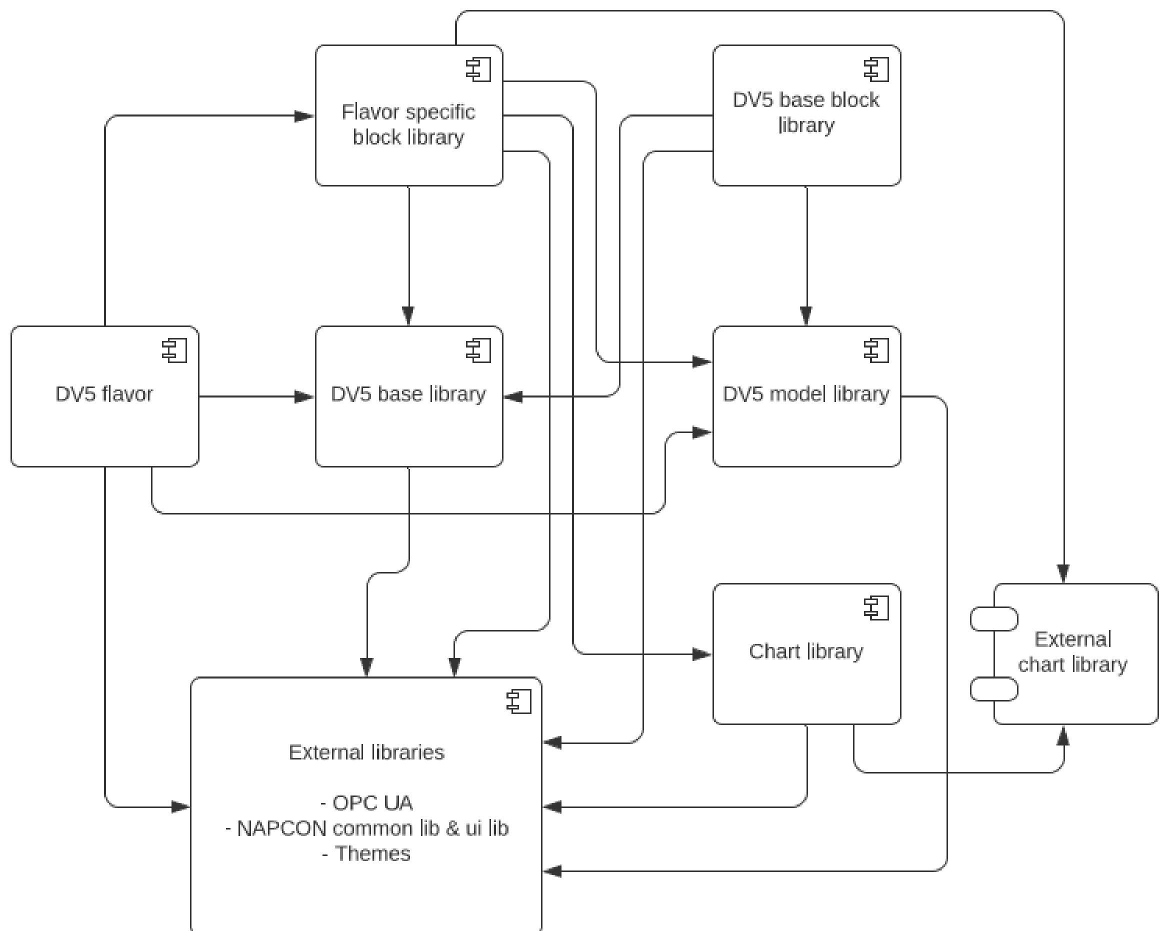

Figure 23. The current architecture of DV5.

The DV5 flavor component consists of parts that are only used with one flavor of DV5, meaning that these files and classes will not be used in other flavors of DV5. The flavor specific files include configuration files, converters, alarm list handlers, icons, theme specific files, viewmodels and views, such as windows or graphical elements that are defined in XAML format.

The DV5 base library is usable with every flavor of DV5. It includes configuration files, command handlers, converters, factories, models, viewmodels, views, teacher extension views and various classes that have no clear place in the current architecture. Models for DV5 are only stored in this library, and other flavor specific libraries merely extend the models or use them as they are.

The flavor specific block library consists of mostly of different views and themes for the flavor in question. The block library means that equipment (or blocks) that are displayed on DV5 picture window are stored here with all the necessary graphical elements. These graphical elements specify what valves, pumps, motors and tanks look like. They also handle the graphics for various shapes and different elements that hold and visualize data, such as dynamic text boxes that are clickable texts that are used to open faceplates. In addition, certain windows have their unique graphical elements and those files are accessed in the flavor specific block library.

DV5 base block library has merely converters and general views that can be used in any flavor of DV5. These views include blocks, shapes and a window. The blocks are dynamic text boxes and different indicators that have visualization built in. Shapes include graphics for generic bars, valve symbols and switch shapes. There is only one window included in this library's views. Base block library contains general elements that are shown in the process displays, but general blocks are not very useful, since all flavors are custom made and the graphical elements look different from each other. This means that the reusability of these parts is low.

The DV5 model library consists of models that handle the business data and back-end of DV5. They are at the core of DV5 and are used in every flavor of DV5. The models handle how pictures or displays are view in the picture window during simulation, how favourites window operates and how users are managed. The base classes likes process.cs, area.cs and user.cs are included in this library.

The chart library consists all the files that are used with creating a graph of desired process values. These graphs that are also called trends and they visualize how process data changes along time. The trend window view is contained in flavor specific block libraries, but the actual graph creation and how different parts works and look are handled in this library. The chart library has all the views, viewmodels and models that are used to create graphs. Out of all the libraries this is the one that most strictly follows the MVVM patterns and is most modular in a sense that it can be reused in any flavor of the DV5 as it is. The chart library uses an external chart library, and it is meant to be the only library that connects to that external library. This is not the case right now, and the situation will be studied in order to

unveil redundant dependencies.

Other external libraries that are used in DV5 include the OPC Unified Architecture (OPC UA) library that handles machine to machine communication in industrial automation. The OPC UA protocol is developed by OPC Foundation. It is based on client-server communication and focuses on communication between industrial equipment and data collection and control. Thus, this protocol needs to be implemented in order for DV5 to communicate with Informer. NAPCON common library and UI library consist of common code that is used for .NET development and common code that is related to graphical user interface development. Themes library is self evident and contains code for all the various themes used in DV5.

The various libraries and their contents will be studied in order to clarify the division of code within DV5. Some libraries have redundant files and files that should not exist within that library. This means that restructuring of DV5 libraries can be done. In addition, the relations between views, viewmodels and models will be closely examined to see how strictly MVVM pattern is followed.

4.3 Other architectural styles expressed in DV5

DV5 architecture has elements of other architectural styles as well. Firstly, it utilizes layered architecture to some extent in how communication is done within the software. As mentioned in section 3.5.1, higher layers of the software are only dependent on the lower layers and not vice versa. This can be seen in the DV5 architecture and how the dependencies between various components are built. On the lowest level are the view logic and business logic of the software. These are utilized by the second layer, which contains the graphical blocks and elements. The third layer has specific functionalities, such as trends OPC UA communication, that utilizes components from the first and second layers. The layers are not currently very strict and there are bypasses from the highest level to the lowest level, but the direction of dependencies stays true to the layered architectural style.

Another architectural style that is utilized in DV5 is the interpreter architecture. DV5 aims to visualize the process displays from the DCS it emulates. The process displays are written and handled in XAML format by simulation engineers. Within DV5 operation, these displays

are sent to the interpreter that understands the language and translates the description of operation given in the XAML file to produce the visual elements shown on the displays during a simulation. The simulation user the gives input to DV5, such as changing the state or value of a process equipment and that change is then handled by DV5 execution platform itself. This architectural style is required to have a working DCS emulator and cannot be removed from DV5 at this point. In this thesis, the interpreter architecture or its improvement will not be studied further due to the scope of the study.

A third architectural style that is present in DV5 is the product-line architecture. There are multiple components that are reused in every flavor of DV5. The flavors can be considered a product family due to the fact that they have very similar functions and structures, but different visual elements and some specialized functions. This study will essentially improve maintainability and understandability, but also emphasize reusing of the components. The more reused parts there are in DV5, the easier it will be to create new flavors.

Product-line architecture is considered in this thesis in order to improve reuse of components, although it will not extend to class-level refactoring. This means that current classes will not be refactored to change inheritance patterns and the design principles will not be changed to favour the use of interfaces over inheritance of classes, as suggested by Gamma et al. (1994). Although such programming principles yield many advantages to development, the scope of the thesis cannot be extended to such detail. It was noted during studying the architecture that DV5 utilized a mixture of design patterns, such as inheritance from parent classes, interface-based compositions and factory methods. More detailed study of existing code and design decisions on coding patterns will be suggested as possible future work.

5 Proposed improvements to architecture

After examining the existing architecture, a new and improved architecture was studied. The new architecture aims to streamline and simplify division of contents within the project and make the reusable modules as understandable as possible. Proposed new architecture is shown in figure 24.

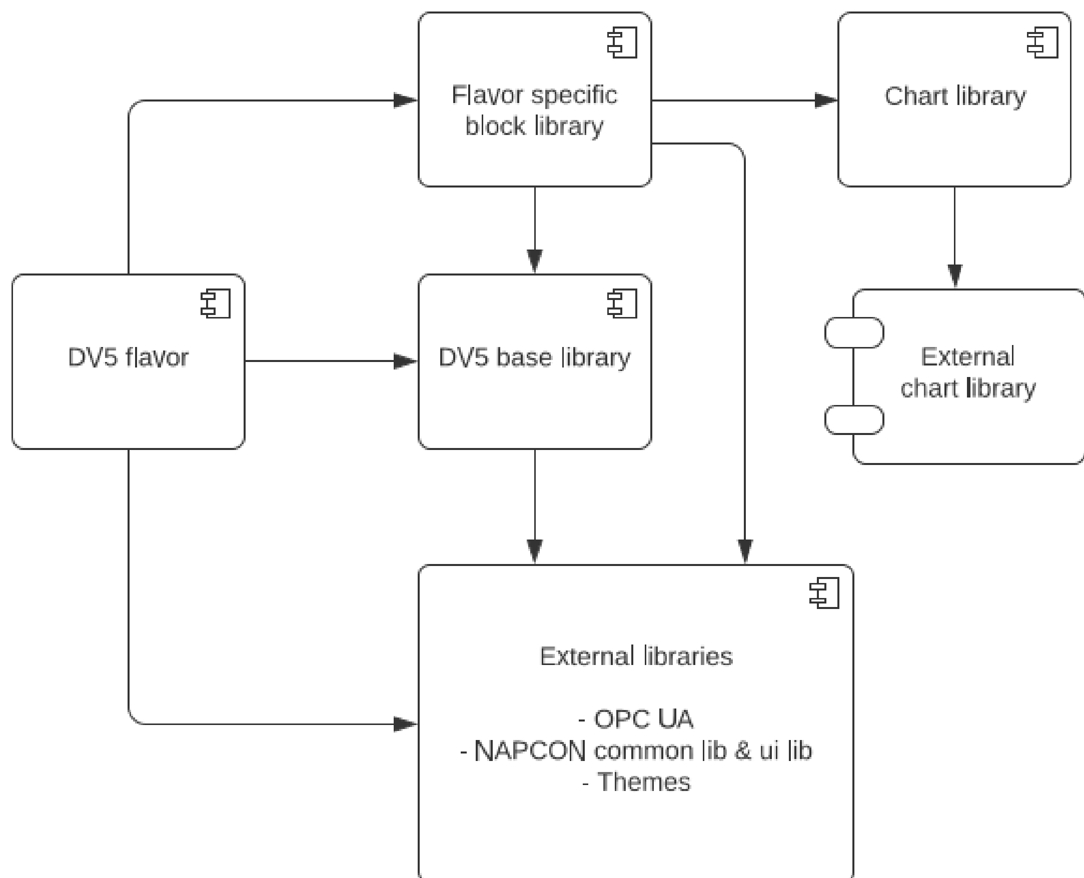

Figure 24. The proposed new architecture of DV5.

First major modification to the structure is that the contents of two different libraries are embedded into other existing libraries. This unifies the contents to follow a more clear structure and makes it easier to find files that are connected operationally. The contents of DV5 model library was embedded into DV5 base library, that already had a variety of model

files in it. By moving all models to base library, DV5 model library became obsolete and could be deleted. Base library, chart library, external chart library and other external libraries are the components that can be transferred and reused in new projects in full. On the contrary, DV5 flavor and flavor specific block library are always project dependent and will contain specialized code to facilitate the outlooks and functions of the certain flavor.

It was observed that the DV5 base block library contained only few reusable components within, and rest of the components were flavor specific parts that were unused by the project that is studied in this thesis. The reusable parts that were embedded into flavor specific block library included a variety of converters, blocks and symbols. The rest of the code in base block library was removed from the project.

By removing two separate libraries and embedding their reusable contents within other libraries streamlines the dependency pattern and makes it more understandable. The topmost level of dependency consists of the external libraries. This is due to DV5 being dependent on ProsDS and Informer. These external libraries contain code that enables communication between these software and by having them as external libraries, the developers of different software cannot damage the components that work as interfaces between the software.

Second level of dependency consists of the chart library, flavor specific block library and DV5 base library. These libraries inherit classes and methods from the external libraries and extend them to be used in a more specialized manner. DV5 base library has all the base classes, methods, handlers, models, viewmodels and views that are needed to run an unspecialized simulator. Base library will also contain configuration files, factories, interfaces, teacher extensions and different theme files. Chart library is the basis for creating charts that map trends on customizable time scales. Both base library and chart library are will be used in new projects to facilitate correct functions for new training simulators.

On the second level of dependency, the first specialized library will be the flavor specific block library. It depends on external libraries, the chart library and the base library. The flavor specific block library should consist of the following items:

- Specialized converters,
- specialized themes files,

- customized and brand new views, such as blocks, shapes and windows, and
- customized files to facilitate charts and trends for certain flavor.

Different flavors of DV5 have different visual elements, specialized windows, customized blocks that will be created in flavor specific block library. These parts are needed to make the OTS look visually like the DCS system it emulates, but the functionality of the software is in base library and DV5 flavor library.

On the bottom level of dependency is the DV5 flavor component or library that is the base for running the software and compiling is handled from this point onwards. DV5 flavor uses contents from flavor specific block library, DV5 base library and external libraries. Configurations and assembly of DV5 is handled from this library. Even though the block library has all the required theme related resources, the theme configuration is handled in this component. Specialized functionality of certain DV5 flavor is contained within flavor library. This means models, viewmodels, views and some converters for specialized functionality is contained within.

For the flavor that is studied in this thesis, specialized functionality that is contained within is the alarm list and its dependent files. Alarm list works closely with ProsDS that actually simulates the process statuses and invokes the alarms when they are encountered. The alarm list is a core functionality of the OTS and thus classes and methods from other libraries cannot access any part of it. This is done in order to ensure correct and limit access to certain ProsDS interfaces. Other functionalities are the main window functions, the ability for the operator to save and load favourite windows within the simulation and an event list.

5.1 MVVM pattern and code quality factors

In addition to the new architecture proposed in this thesis, the MVVM architectural pattern was further studied. This includes analyzing how different parts of the software are divided between views, viewmodels and models. Also code quality was examined to see if there are redundant imports, wrong access modifiers that might risk the functionality of the software and wrongly chained calling of methods that result in multiple branches of possible calls and makes the software more complex. In addition, the dependency pattern was studied to

improve functionality, maintainability and security.

It was found that some file contents should be divided into their individual files, in stead of having a vast number of lines and many separate entities and structures within one file. Following a clear pattern that separates different entities to their own files improves understandability and maintainability. Types that were separated and moved into their own files are interfaces, enumerables, exceptions and classes that act as models in the MVVM pattern. Table 1 shows the number of new entities that were divided from existing files to separate files.

Type	Number of new files
Interface	10
Enumerable	25
Exception	1
Model	22

Table 1. Number of new files by type that are created by separating entities.

Out of the 22 new model classes found in the existing files, 2 were found within viewmodel files and others were found within model files. The separation of model and viewmodel contents in MVVM pattern are not always clear, but as mentioned in section 3.5.7, model cannot know about the view, and viewmodel acts as connector between the view and the model.

Code quality was studied in order to increase understandability and maintainability of the software. The following steps were taken in order to improve the code quality:

- Removed redundant code,
- changed access modifiers,
- optimized the use of built-in types,
- optimized 'using' directives, and
- shortened qualified references.

These changes also improve the performance of the software in terms of memory usage.

They also streamline the use of good coding conventions that aim to improve readability and maintainability. Other code quality changes can also be made to improve readability, for example changing file layout to follow certain template and to unify syntax between files.

The new proposed architecture yields many advantages. These advantages are more closely evaluated in the next chapter.

6 Results

The results of the Sonarqube static code analysis are presented in this chapter. Evaluation of the new proposed architecture is done by using measurements gained from the static code analysis. Firstly, the old architecture was analyzed with Sonarqube to achieve a baseline of various metrics. Secondly, the new architecture was analyzed to gain new measures that are then compared with the old ones. This way, the the architectures can be numerically compared and the changes in the metrics can be accurate evaluated.

6.1 Metrics used in the evaluation

Certain metrics were selected to be used in the evaluation phase of the thesis after static code analysis was decided as the research method. These metrics are relevant to how the software quality can be assessed. The measures selected to be used in the evaluations are the following:

- Lines of code,
- cyclomatic complexity, and
- cognitive complexity.

Lines of code is the most basic metric that a software can have. It is used to measure the size of a software by counting the number of lines in the program's source code. The bigger the software, the more lines of code it has. If the same program can be created with fewer lines of code, it will be better in many aspects.

Cyclomatic complexity is a software metric that measures how complex a program is. More accurately it measures the number of linearly independent paths through a software's source code. It was developed by McCabe (1976). Basically, the complexity of a software increases with the number of conditionals and decisions points that divide a single path to multiple paths through the code. Cyclomatic complexity can thus be seen as a measure of how many test cases are needed to gain complete branch coverage. In this case as well, the smaller the complexity, the better.

According to Bray et al. (1997) cyclomatic complexity is commonly used by comparing it to a set of threshold values. These threshold values of complexity have been derived from measuring a large number of programs and the ranges of complexity are aimed to help software engineers determine a program's inherent risk and stability. The authors state that studies show a correlation between cyclomatic complexity and error frequency, which leads them to conclude that a low cyclomatic complexity helps making the program more understandable and more modifiable. A method's cyclomatic complexity also indicates its testability. The thresholds suggested by the authors are presented in table 2.

Cyclomatic Complexity	Risk Evaluation
1-10	a simple method, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk method
greater than 50	untestable method (very high risk)

Table 2. Threshold values for cyclomatic complexity. (Bray et al. 1997, p.147)

While cyclomatic complexity does give an indication of how complex a system is, cognitive complexity measures how difficult the application is to understand. Campbell (2018) states that cyclomatic complexity was created to measure testability and maintainability, but only excels at measuring testability. That is why the author developed a better metric to actually measure maintainability. The improvements that cognitive complexity introduces to cyclomatic complexity are the following (Campbell 2018, p.4):

- Ignores structures that are shortened to one line but contain multiple statements,
- increments the measurement when there is a break in the linear flow of the code (loop structures, conditionals and recursion cycles), and
- increments the measurement when flow-breaking structures are nested within each other (loops within loops).

According to Campbell (2018) cognitive complexity's primary goal is to calculate a score that accurately reflects the relative understandability of the software. It is more focused on methods level understandability than full software architecture understandability, so it does

not fully measure how restructuring the architecture might improve maintainability.

A threshold value for a good cognitive complexity measure is 15 at a method level, as mentioned by Cambell (2017). Campbell explains that for a package with only simple classes, a cognitive complexity of 5 to 10 might be too high, but for packages that hold business logic classes, a cognitive complexity of 150 and above indicate that the classes should be split. Thus, package or project level cognitive complexity thresholds are hard to estimate and should not be used. Only the threshold for method level is recommended.

6.2 Results of the analyses

Table 3 presents the measurement results that were achieved by analyzing the old and the new architecture with Sonarqube static code analysis.

	Lines of Code	Cyclomatic C.	Cognitive C.
Old architecture	48410	9907	6304
New architecture	45682	9552	6069
Change	-5,6%	-3,6%	-3,7%

Table 3. Measurement results of Sonarqube static code analysis

Static code analysis shows that by changing the architecture of DV5 the number of lines of code decreased by 5,6 percent, the cyclomatic complexity decreased by 3,6 percent and the cognitive complexity decreased by 3,7 percent. All of the results are in line with the original hypothesis of the study. By modifying the architecture, a reduction in the size of the software and complexity can be achieved without losing any functionality.

The reduction in lines of code means that there is smaller amount of code to maintain and the software requires less hard drive space. DV5 has greatly expanded through the years and by improving the architecture with the help of this thesis, some redundant code and components were removed and DV5 was restructured in a more maintainable manner.

Both cyclomatic and cognitive complexity decreased in the software due to architectural changes. Both complexity metrics changed by nearly same amount, meaning that the original

code contained only little code that had low understandability. If the code that was removed due to architectural changes were overly complex, the difference between the change of the two different complexity metrics would be larger. The change in complexities is measured relatively when comparing the old and the new architecture. A method level complexity evaluation is made in section 6.3.

6.3 Additional results

By further analyzing how the architectural changes might have affected the software, two additional metrics are gathered from the static code analysis. These metrics are the number of classes and number of files. These metrics are interesting when analyzing how the architecture has changed. The results of these metrics are shown in table 4.

	Files	Classes	Functions
Old architecture	583	654	5352
New architecture	615	602	5075
Change	5,5%	-7,9%	-5,2%

Table 4. Additional measurement results of Sonarqube static code analysis

In table 4 we can clearly see that the number of classes decreases due to the new architecture, whereas the number of files increases. This is due to the fact that interfaces, enumerables, exception classes and model classes were separated from existing files and moved into their own file. This separation enforces the MVVM pattern and makes the code more maintainable and less prone to bugs and vulnerabilities.

The number of new files would be even bigger, but due to removal of duplicate code, some files were completely removed from the project. The amount of classes decreasing means that the software was made simpler, thus more understandable and less complex. The number of classes could also be decreased by the new architecture through changes in code and dependencies of the components.

The cyclomatic complexities of methods were measured and categorized by using the thresholds given in table 2. The number of methods per each cyclomatic complexity category are

shown in table 5.

Cyclomatic Complexity	1-10	11-20	21-50	50+
Old architecture	5296	42	12	1
New architecture	5020	41	12	1
Change	-276	-1	0	0

Table 5. The number of methods within each cyclomatic complexity threshold category.

Even though the cyclomatic complexity of DV5 decreased relatively by 3,6 percent, the methods with a cyclomatic complexity of 21 or more did not decrease. This is due to the fact that the architectural improvements were aimed at higher level architecture and did not take into account the refactoring of individual methods within DV5. In the lowest risk category of cyclomatic complexity, the number of methods decreased by 276 and in the category of 11-20, the number of methods decreased by 1. This change happened due to the removal of duplicate code within DV5.

The cognitive complexities were also measured on method level. For cognitive complexity, there is only one threshold value that divides the methods. The number of methods in each cognitive complexity category are shown in table 6.

Cognitive Complexity	15 or smaller	15+
Old architecture	5297	55
New architecture	5020	55
Change	-277	0

Table 6. The number of methods within each cognitive complexity threshold category.

It is seen that the number of methods with good cognitive complexity value has decreased by 277, whereas the methods with higher cognitive complexity still remain due to same reasons that methods with higher cyclomatic complexity remained: The architectural improvements did not aim to refactor methods. It can also be noted that the change in low cyclomatic complexity methods and low cognitive complexity methods is correlating, meaning that the methods in DV5 and their complexity values are correlating.

Duplication between different flavors was also measured to get an undestanding of how components are reused. The measurements are achieved by comparing the new architecture of Flavor 1 to old architectures of two other flavors (Flavor 2 and Flavor 3). The scope of this thesis did not include changing the architecture of the other flavors, meaning that the change in the amount of duplicated components between the old and new architectures is not measured.

The measure of duplication is used to gain an understanding of the current situation, and that understanding can be used in the future to further increase reusability of the components. In DV5 development, Flavor 1 is the oldest flavor used and is often used as the basis of other flavors. This means that flavor 1 code is reused and refactored to create a new flavor. The percentage of duplication is an indication of how much duplicate code a flavor has when compared to other flavors. The amount of duplications are shown in table 7.

Flavor #	Flavor 1	Flavor 2	Flavor 3
Duplications	65,5%	58,9%	54,0%

Table 7. The amount of duplications between different flavors.

Flavor 1 is the basis for other flavors. This means that Flavor 1 will always have more duplicate code than other flavors. This is due to the fact that some flavors may be able to reuse some parts of Flavor 1 that are not reusable for other flavors. That leads to Flavor 1 having a higher duplications number than other flavors.

7 Discussion

The purpose of this thesis was to study the current architecture of DV5 and find ways to improve it to gain better maintainability, understandability and reusability. The two research questions set for this study were "What is the current architecture for DV5?" and "How can the current architecture be improved?". In this chapter, the results of the thesis are analyzed and discussed and conclusions will be drawn based upon those.

7.1 Current architecture

In the beginning of the study, there were no pre-existing documentation of the architecture, and none of the original developers of DV5 worked at the company any longer. This meant that methods to unveil the current architecture were pratical and required very exploration of current code base. By exploring the classes and dependencies of the components, an initial understanding of the software was gained. It was considered important to map the architecture in order for current developers to gain an understanding of it. The mapping was done with Lucidchart and with the help of Code Map extension to Visual Studio.

By exploring the code, it was understood that the architecture is MVVM architecture. This is due to the technologies used in DV5 as well as the fact that it is very suitable for graphical user interfaces. The division between models, views and viewmodels makes development and testing easier due to the division of code between the different types. On top of MVVM architecture, some other architectural styles are also expressed in DV5. These styles are layered architecture, interpreter architecture and product-line architecture. As the improvement work of the architecture progressed, it was decided that MVVM architecture is most suitable for this type of software, and that the layers are sufficiently built to keep certain parts of the software inaccessible to inhibit undesired interactions within. The product-line architecture was also considered to be suitable, because DV5 is meant to be customized for multiple customers and by reusing components, a lot of development work is already done and tested in use. The interpreter architecture was not studied further in the scope of this thesis.

7.2 New architecture

The improvements to the architecture were aimed to increase understandability and maintainability, as well as making components and their contents reusable for future development of new flavors. This meant that the MVVM architecture would be enforced and the components and libraries would have to be restructured. It was noted that the there were a lot of duplicate code within the flavor studied and thus certain classes could be deleted. It was also noticed that the division of classes and files demanded by MVVM architecture was not fully complied with in the original architecture and was repaired in the improved architecture. In addition, several files contained code from separate classes, and in the new architecture they were divided into their own classes. This was aimed to improve structure and understandability. Lastly, all redundant code was erased to decrease the amount of code within the code base.

As shown in chapter 6, the new architecture decreased the the amount of lines of code, cyclomatic complexity and cognitive complexity of the software by a fair amount. In addition, number of classes was reduced while creating new files in order to enforce MVVM pattern and good code quality. These are the metrics that could be gathered by static code analysis, and the results clearly show improvement in structure, understandability and maintainability. On top of that, qualitative improvements to the architecture that cannot be measured quantitatively are the improved structuring of the various libraries and their contents within DV5. By streamlining dependency between the different components, further development of DV5 becomes easier when developers understand how the software is built and how to implement new functionalities without breaking the program.

Method level analysis of both cyclomatic and cognitive complexities shows that mainly DV5 contains methods with low complexity, meaning that the methods in DV5 are already quite understandable and maintainable. The new architecture removes a number of low complexity methods by removing duplicate code within a DV5 flavor, but does not tackle the few high complexity methods. The high complexity methods are found within classes that hold the base of business logic within DV5 and thus cannot easily be refactored.

The new structure of components are clearly aimed to improve reusability of existing code.

By having two libraries that contain the core code of DV5 and that can be used in any new DV5 project makes further development easier. It also means that DV5 is becoming more maintainable by decreasing the complexity of the software. Measurements show that between different DV5 flavors over half of the code is exactly the same, meaning that over half of DV5 is fully reusable. The amount of reused code depends on the flavor, because some flavors have more customized code than others. The percentage of reused code may increase due to the architectural improvements, but proving that would require more studying how the modules are used in new DV5 projects.

7.3 Utilization of the results

The results of the study are meant to be implemented in the future by restructuring all the flavors to accommodate the new component division and library structure. By having a documented architecture diagram of the current and the new architectures, much needed information is gained on the software's current state and a suggestion to how DV5 should be changed in the future. The suggested improvements are viable and possible to implement in a sensible time frame. In addition, the benefits of the changes in architecture would have an impact in the future development work in terms of how easily the software can be understood and maintained.

If the changes in architecture cannot be done in the near future, the mapping of the current architecture should be shared with the organization and especially with the developers working on it. By increasing the developers' understanding of the current state, a mutual understanding of development patterns and styles can be gained. This would unify how the people working on DV5 conceive the structure and would likely yield advantages in how all the developers approach tasks in terms of finding alternative solutions to them. The current architecture diagram will also help new recruits to get acquainted with the software.

8 Conclusion

This thesis aimed to investigate the current architecture of DV5 and propose an improved architecture to make it more understandable and more maintainable for current and new employees, and to the components reusable for new DV5 flavors.

Operator training simulators were studied in order to gain an understanding of why they exist and what are their advantages. In addition, NAPCON Simulator and DV5 were introduced superficially. After researching the operator training simulators, a literature review was done to gain knowledge of software architectures, architectural design and architectural evaluation. An understanding was built in order to answer the first research question "What is the current architecture for DV5?". The current architecture was extensively studied and as a result, it was mapped to a diagram. The current architecture diagram can be used by new and current developers to further increase their understanding of the current state of DV5 and to improve their development methods to accommodate the software's needs.

After studying the current architecture, the second research question was focused on. The research question was "How can the current architecture be improved?". After understanding the current architecture, certain improvements could be made to it. These improvements were aimed to enforce the MVVM architecture, to decrease complexity, improve understandability and maintainability, to simplify structure and to have reusable components for future DV5 flavors. After making the improvements, the old and the new architectures were analyzed with Sonarqube static code analysis. The results of the analyses were compared to gain numerical data on the improvements. It was observed that the lines of code and complexity of the software decreased due to the architectural changes. Simultaneously the number of files increased due to the enforcing of class division and moving code entities into their own files.

This thesis left certain things out of its scope due to the limited time that could be allocated for the study. Some future work regarding further development of DV5 may concern coding conventions, such as if views should be mainly programmed in the XAML file and if the amount of code in the code-behind should be minimized. Another possible future work

case might consider optimizing performance of WPF data bindings within DV5. This type of task goes deeper into the programming language and its technology. Third future work topic would be to further study if DV5 should be built using composition of interfaces over inheritance of parent classes. These topics would more effectively improve the performance of the software and would be valuable future research to continuously develop DV5.

Bibliography

America, Pierre, Eelco Rommes, and Henk Obbink. 2003. "Multi-view Variation Modeling for Scenario Analysis". In *PFE,* 44–65. https://doi.org/10.1007/978-3-540-24667-1_5.

Babar, Muhammad Ali, Alan W. Brown, and Ivan Mistrik. 2013. *Agile Software Architecture: Aligning Agile Processes and Software Architectures.* 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Babar, Muhammad Ali, Liming Zhu, and Ross Jeffery. 2004. "A framework for classifying and comparing software architecture evaluation methods". In *2004 Australian Software Engineering Conference. Proceedings.* 309–318.

Balaton, M.G., L. Nagy, and F. Szeifert. 2013. "Operator training simulator process model implementation of a batch processing unit in a packaged simulation software". *Computers & Chemical Engineering* 48:335–344. https://doi.org/https://doi.org/10.1016/j.compchemeng.2012.09.005.

Bass, Len, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice.* 2nd edition. Boston: Addison–Wesley.

Bosch, Jan. 2004. "Software Architecture: The Next Step". In *Software Architecture,* 194–199. Springer Berlin Heidelberg.

Bray, M., K. Brune, D. Fisher, J. Foreman, and M. Gerken. 1997. "C4 Software Technology Reference Guide: A Prototype". Software Engineering Institute.

Burbeck, Steve. 1992. "Applications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC)", http://www.dgp.toronto.edu/~dwigdor/teaching/csc2524/2012_F/papers/mvc.pdf.

Buschmann, Frank, D.C. Schmidt, R. Meunier, H. Rohnert, K. Henney, M. Kircher, M. Stal, P. Sommerlad, and P. Jain. 1996. *Pattern-Oriented Software Architecture, A System of Patterns.* Wiley. ISBN: 9780471958697.

Cambell, G. Ann. 2017. ”SonarQube: Qualify Cognitive Complexity”. Visited on October 25, 2020. https://stackoverflow.com/questions/45083653/sonarqube-qualify-%09%09cognitive-complexity.

Cameron, D., C. Clausen, and W. Morton. 2002. ”Chapter 5.3 - Dynamic Simulators for Operator Training”. In *Software Architectures and Tools for Computer Aided Process Engineering,* 11:393–431. Computer Aided Chemical Engineering. Elsevier. https://doi.org/https://doi.org/10.1016/S1570-7946(02)80019-0.

Campbell, G. Ann. 2018. *Cognitive Complexity - A New Way of Measuring Understandability.* SonarSource S.A.

Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, M. Little, Paulo Merson, Robert Nord, and Judith Stafford. 2010. *Documenting Software Architectures: Views and Beyond.* 2nd edition. Addison-Wesley Professional.

Dijkstra, Edsger W. 1968. ”The structure of the ’THE’-multiprogramming system”. *Communications of the ACM* 11 (5): 341–346.

Fowler, Martin. 2004. ”Presentation Model”. Visited on August 23, 2020. https://martinfowler.com/eaaDev/PresentationModel.html.

Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* 1st edition. Addison-Wesley Professional.

Gossman, John. 2005. ”Introduction to Model/View/ViewModel pattern for building WPF apps”. Visited on August 16, 2020. https://docs.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps.

Heesch, U. van, V. Eloranta, P. Avgeriou, K. Koskimies, and N. Harrison. 2014. ”Decision-Centric Architecture Reviews”. *IEEE Software* 31 (01): 69–76. ISSN: 1937-4194.

Hevner, Alan R., Salvatore T. March, Jinsoo Park, and Sudha Ram. 2004. ”Design Science in Information Systems Research”. *MIS Q.* (USA) 28 (1): 75–105. ISSN: 0276-7783.

Hofmeister, Christine, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, and Pierre America. 2007. ”A General Model of Software Architecture Design Derived from Five Industrial Approaches”. *Journal of Systems and Software* 80 (1): 106–126. ISSN: 0164-1212. https://doi.org/https://doi.org/10.1016/j.jss.2006.05.024.

Hofmeister, Christine, Robert Nord, and Dilip Soni. 1999. *Applied Software Architecture.* USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201325713.

”IEEE Recommended Practice for Architectural Description for Software-Intensive Systems”. 2000. *IEEE Standard 1471-2000,* 1–30.

Kallakuri, Ravikanth, Bahuguna Prakash, Donald Glaser, and Sanjay Shivalkar. 2018. ”Study of Effectiveness of Operator Training Simulators in the Oil and Gas Industry”. In *Proceedings of The 59th Conference on Simulation and Modelling (SIMS 59), 26-28 September 2018, Oslo Metropolitan University, Norway,* 79–86. https://doi.org/10.3384/ecp1815379.

Kazman, Rick, L. Bass, Gregory Abowd, and M. Webb. 1994. ”SAAM: A Method for Analyzing the Properties of Software Architectures”. In *Proceedings of the 16th International Conference on Software Engineering,* 81–90. IEEE Computer Society.

Kazman, Rick, Mark Klein, Mario Barbacci, Thomas Longstaff, Howard Lipson, and S. Carrière. 1998. ”The Architecture Tradeoff Analysis Method.” In *Proceedings on Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'98),* 68–78. IEEE.

Koskimies, Kai, and Tommi Mikkonen. 2005. *Ohjelmistoarkkitehtuurit.* Talentum.

Kruchten, Philippe. 1995. ”The 4+1 View Model of Architecture”. *IEEE Softw.* (Washington, DC, USA) 12 (6): 42–50.

———. 2003. *The Rational Unified Process: An Introduction.* 3rd edition. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321197704.

McCabe, Thomas J. 1976. ”A Complexity Measure”. *IEEE Trans. Softw. Eng.* 2 (4): 308–320. https://doi.org/10.1109/TSE.1976.233837.

”NAPCON Simulator”. 2020. Visited on April 25, 2020. https://www.napconsuite.com/product/napcon-simulator/.

Parnas, D. L. 1972. ”On the Criteria to Be Used in Decomposing Systems into Modules”. *Commun. ACM* (New York, NY, USA) 15 (12): 1053–1058.

Patle, Dipesh S., Zainal Ahmad, and Gade P. Rangaiah. 2014. ”Operator training simulators in the chemical industry: Review, issues, and future directions”. *Reviews in Chemical Engineering* 30 (2): 199–216. https://doi.org/10.1515/revce-2013-0027.

Potel, Mike. 1996. ”MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java", Taligent Inc”, http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

Ran, Alexander. 2000. ”ARES Conceptual Framework for Software Architecture”. In *Software Architecture for Product Families Principles and Practice,* edited by Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. Addison–Wesley.

Smith, Josh. 2009. ”Patterns - WPF Apps With The Model-View-ViewModel Design Pattern”. Visited on August 16, 2020. https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern.

”The MVVM Pattern”. 2012. Visited on August 16, 2020. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10).

Williams, Lloyd, and Connie Smith. 2002. ”PASA(SM): An Architectural Approach to Fixing Software Performance Problems.” In *28th International Computer Measurement Group Conference, December 8-13, 2002, Reno, Nevada, USA, Proceedings,* 307–320. Computer Measurement Group.

Appendices