Linnéuniversitetet
Kalmar Växjö

Thesis work

API development to improve integration and resource efficiency.

Decreasing user interactions and automating verification processes of information between applications.

Author: Christoffer Roth
Supervisor: Jesper Andersson
Examinator: Jesper Andersson
Supervisor, company: Magdalena Hallbrandt, Volvo CE
Datum: 2018-05-26
Course code: 2DT00E, 15 ECTS
Subject: Computer Engineering
Level: Bachelor

Institution for Computer Science

Abstract

Companies today often have a variety of applications used in the daily work. The problem that companies face with these applications is that they often are brought in to deal with a specific task, and they are often brought in at different times by different third-party developers. This results in the applications being independent units and integrates poor with each other, making work and maintenance with the applications inefficient. To improve efficiency the applications need better integration with each other. Better integration can be achieved by either replacing the current applications with a new software or develop a software that helps the applications communicate.

This project covers the development of the later, an API to improve the efficiency at Volvo Construction Equipment in Braa s. The API is developed with the Enterprise Service Bus (ESB) as inspiration. The purpose of the ESB is to act as a middleware for the applications. Due to time limitations for the project integration between the applications wasn't achieved. Instead, the focus was set on improving one of the moments in the work process at Volvo, that is verifying information between applications. The verification is today done manually which makes it time-consuming and this is the API set out to deal with. The API results in a reduction and improvement regarding the verification. The API still needs a manual input of data from the applications, but the API has automated the verification of the information between the applications resulting in hours of reduced work for the staff at Volvo.

Keywords: API, Enterprise Service Bus, ESB, Enterprise Application Integration, EAI, Legacy Systems, LS, Middleware.

Preface

To complete the assignment the company, Volvo Construction Equipment (CE) in Braås, enabled the possibility to do the project at their site. This possibility made the development of the API more convenient, and support from the supervisors at Volvo CE was always close at hand. I would like to thank Volvo CE for the assignment enabling the making of this thesis work. Also, a big thanks to the supervisors Magdalena Hallbrandt and Sebastian Estberg for taking the time and the support they have given during the 10 week period. A thank also goes to all the staff that I've met during at Volvo CE for making the work very enjoyable.

Contents

1 Introduction

This report will cover the development of a software API that will act as a prototype to show one way on improving integration, increase productivity and decrease sources for potential errors. The project has been conducted at Volvo Construction Equipment (CE), located in Braås and is part of the thesis work, 15 ECTS, that's required to complete a *Bachelor Degree in Computer Science* at Linnaeus University, Växjö.

First, a background will be given to the current state of businesses today when it comes to older software and why these types of software are often very resource demanding. The process of designing the new software to improve integration, make it as scale-able and reusable will be presented.

1.1 Background

Today continues improvements and advances in software development proceeds each year and software just a few years old is today be outdated. These outdated software goes under the collective term Legacy Systems (LS) and businesses today tends to have LS integrated into the daily workflow. As the LS serves a function, maybe a critical, they are not easily replaced by new software. Also, a lot of resources have been allocated to these LS over the years and are therefore heavy investments for businesses. Resulting in companies holding on to LS instead of upgrading to new software. One of the main reasons to why this could result in a decrease in productivity to the companies is that LS integrate poorly with each other, in most cases not at all. Leading to companies having systems that don't communicate with each other, require separate updates and maintenance, they also need that employees know all the LS or at least some of them.

Integration challenges can also occur when implementing new software in businesses. Even if the new software is developed with the latest knowledge and experience in software development, they don't take into account the communication with the existing LS. Since LS is independent software in the companies new software would have to implement interfaces for each LS to enable point-2-point communication. The same implies for the existing LS to allow communication between each other. This for obvious reasons, scales very poorly.

The cost of replacing existing LS for businesses could not only be expensive but could also result in a resource demanding businesses process overhaul. Point-2-point integration between LS is not only a poorly scalable solution,

but also resource demanding when regarding maintenance and updates. An update on one LS could require updates on all LS integrated with it.

One term that is becoming more common today is Enterprise Application Integration (EAI). EAI is a general term for implementation patterns that deal with integration problems between Legacy Systems. EAI patterns can work as middleware for the different LS, dealing with the communication and data formatting so that the different LS can communicate. The EAI implementation removes coupling between LS resulting in reduced development cost, the new system is added to the middleware instead of adding point-2-point connections to every LS and improves scalability.

There is not one EAI-pattern that solves all integration problems for all businesses. Legacy Systems have a tendency to be developed for a specific purpose at companies, so they have often been developed by different third-party developers, in different programming languages and so on. The Enterprise Service Bus (ESB) is one EAI-pattern that have presented good results with the integration of both LS and the addition of new software. The ESB deals with scalability and coupling between LS effectively. The development can, however, be complicated since the ESB often needs to be tailored to each business unique situation.

1.2 Problem formulation

The products developed and built at Volvo CE site in Braås are articulated haulers. There are an array of different models that not only differ in size and loading capacity but also in model configurations. The engines on the machines have Engine Control Units (ECUs) and a variety of different software are installed on the ECUs.

To keep track of software implemented, or under development, for the different ECUs, various applications is used. These applications are independent of each other, and this can be related to the scenario with Legacy Systems, which need updates with the correct software implemented on the ECUs. The applications are named SE-tool (requirement tool), Roadmap (excel-sheet), Configurator (ESW-tool) and Kola (PDM system). Each of the applications holds information on the implemented software on the ECUs.

The process today to checking that the applications hold the correct software for the ECUs is manually made. What software that should be implemented on the ECUs is listed in the application SE-tool, therefore, it acts a hindsight when verifying that the other applications information.

Since there are an array of machine-models and one machine has a variety of ECUs the process to verify the information between the different applications results in a time-consuming process. Not only is it slow but there is a lot of part-numbers, the software on the ECUs are translated into part-numbers so when software is updated it generate a new part-number for that software, which makes it a potential source for errors.

1.3 Project goal and Motivation

To reduce not only the time to make the verification but also decreasing potential sources for errors between the different applications an API to run the verification is to be developed. The primary goal for the API is to reduce the work and time effort for the verification considerably, but also to reduce the sources for errors. The APIs will make the current manual verification process automated, reducing the impact of the human factor for verification process.

The API will be developed with the Enterprise Service Bus (ESB) as inspiration, meaning that the API needs to a have good scalability but also low coupling between the components of the API. This to make the API components more reusable and act as a prototype or inspiration for future development.

With the API a lot of time for the staff can be reduced so the focus can be on the development on the machines instead. In turn making better and more competitive products on the market. Also with potential sources for error reduced a scenario, e.g. where the ECUs not running the latest or updated software is lowered. Wrong software implemented on the ECUs could result in poor performance for the engines, or worse the engines are not functional, resulting in production loses for the customer.

1.4 Scope

Due to time restriction, the API will not be a fully integrated solution between the applications. Instead, the focus will be on automating the verification process to remove the manual check conducted today. The API will notify the user when an error occurs, but the possibility to change errors via the API will not be implemented. Also, the API will be excluded functionality for adding new information to the different applications.

1.5 Outline

The outline of the report is as follows. Section two details the theory of concepts and terms relevant to the report, terms as *Enterprise Service Bus (ESB)* and *Model-View-Controller (MVC)*. For section three the method for extracting the data from the existing applications will be presented. Each application will be given a small description and then how to obtain data from them. Section four describes the development process. Here each domain in the MVC will be detailed with its components and functionalists. Section five will present an evaluation of the API function and if it is an improvement to the current situation at Volvo CE. Section six gives the conclusions about the development of the API and also provides a thought on future development.

2 Theory

In this section, a brief description is given on terms used in studies covering solutions for better integration at businesses.

2.1 Engine Control Unit (ECU)

Engine Control Unit (ECU) are systems used in engines to monitor and control, for example, fuel injection, engine temperature or throttle position. The ECU is a vital component in engines today, and with the data from sensors, it's possible to optimise the engine performance depending on internal and external environments. For improving fuel efficiency, the optimal mixture of air and fuel intake is monitored and configured by an ECU. The ECU can then produce better ignitions in a combustion engine that in return reduce emissions, ware on the pistons and so on [8]. In vehicles today multiple ECUs are integrated and used to monitor different parts of the engine to improve the overall performance.

2.2 Enterprise Application Integration (EAI)

Enterprise Application Integration (EAI) is the name of the collection of patterns that present solutions on integration between applications. One definition is given as:

Enterprise Application Integration (EAI) provides methodologies and tools to design and implement EAI solutions. The goal of an EAI solution is to keep a number of applications data in synchronous or to develop new functionality in top of them, so that applications do not have to be changed and are not disturbed by the EAI solution [6].

The EAI solution should not change the current applications that exist in the companies. Maintaining and chaining old applications is resource demanding, it also makes the scalability poor. With EAI we can eliminate the scenario where every application needs a point-2-point connection to every other application in the company to achieve better integration and communication. Instead, EAI solutions can produce a situation where applications only need one connection, a connection to a middleware. One implementation of this is referred as Enterprise Service Bus.

2.3 Enterprise Service Bus (ESB)

The Enterprise Service Bus (ESB) is one implementation of Enterprise Application Integration (EAI). The ESB acts as a middleware that applications connect to instead of having point-2-point connections. Reducing the scalability problem but also provides a low coupling solution.

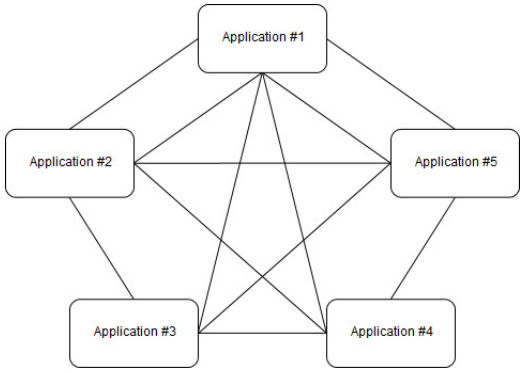

Figure 1: A point-2-point solution. Connections **N(N-1)/2**

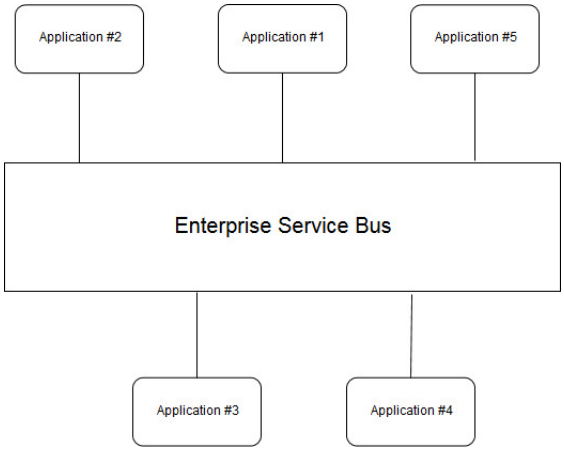

Figure 2: A ESB solution. Connections **N**

Figure 1 displays the integration solution for five applications that uses point-2-point connections. It's easy to see with only five applications the scheme start to become complicated. Adding new applications would result in a lot of work not only by integrating them but also to modify the existing applications to enable integration with the new. In figure 2 an ESB have been used, removing the coupling between the applications. Also, adding new applications doesn't affect the existing ones.

The ESB makes this possible by having some core functions [9]. The ESB has to be able to integrate between applications that use a variety of protocols. It also has to handle messages by transforming them between formats, route messages between applications and also enhance messages that miss information. The layer or API between the Legacy Systems and the *ESB* can result to be complicated to develop because of this. Security, encryption, monitoring and management are also core abilities that the ESB implements.

2.4 Model-View-Controller (MVC)

A *Model-View-Controller* [1] pattern is to separate different components or domains of the program into a modelling domain, presentation domain and controller domain [1].

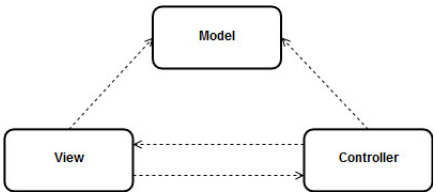

Figure 3: Model-View-Controller pattern.

Figure 3 shows the pattern. The view and controller has a dependency on the model, but the model has no dependencies back. With no dependencies form the model to the view and the controller, they can be changed or updated without affecting the model. Also, implementing multiple views can be done to the ”same” model. Resulting in that various views can be applied to user interactions, e.g. a GUI and webpage.

2.4.1 Model

The Model domain conducts operations on the data and performs all the logic in the MVC-pattern. The Model responds to request from the View or Controller.

2.4.2 View

The View is the domain that manages the presentation to the user. A display can vary in a lot of ways e.g. we can have a view that presents data in a different language, one that shows data on a webpage or present via a GUI window.

2.4.3 Controller

The Controller is the domain that manages the user inputs. When the user interacts with e.g. the GUI window and make some inputs, the Controller takes the input, and conduct request to the Model that perform the operations.

2.5 Java and JavaFX

Java is a high-level programming language developed at Sun Microsystems. A development group, led by James Gosling and Patrick Naughton in 1995 introduced Java to the world at the SunWorld exhibition. Java quickly grown in popularity and programmers embraced since it easier use compared to C++ but also because of the rich Java library. Java can also run the same program on multiple operating systems (OS) without any changes needed to the program. It's made possible with the Java Virtual Machine (JVM) that simulates a CPU, so a compiled Java program isn't translated into CPU instructions directly [7]. JavaFX is a part of the Java library that enables the programmer to create applications, such as a GUI, across multiple platforms [2].

2.6 Apache POI

The Apache POI is an open-source API developed to do work with the Microsoft Office 365[3] files in Java. The Apache Software Foundation [4] developed the API. It's a community that supports and develop non-profitable software open for the public. There are two applications of Apache POI, the HSSF (Horrible Spreadsheet Format) and the XSSF (XML Spreadsheet Format)[5]. The difference comes in the implementations years for the Excel-file formats, HSSF library is used in formats before 2007, the XSSF library is used in formats (.xlsx) from 2007-forward.

3 Method

To run the verification, the information existing in the applications has to be extracted. Because of the time limitation for the project, the scope section 1.4, it's not possible to produce a complete integration solution. Meaning, there isn't time to look at the source code or how to directly work with the file systems for the applications.

The decision has fallen on extracting the information into excel-documents. The reasons for this, first the Roadmap application is an excel-document so an open-source software, Apache POI [5], will be used to extract the data from the excel-file. The Apache POI has excellent support, and it's Java library enables work with excel-files in an efficient way, especially since the API only reads the content in the files.

3.1 SE-tool (requirement tool)

The information gathered from the SE-tool application is essential, because it acts as the hindsight for the verification for the information in the other applications. To make the confirmation today the SE-tool information is copy/pasted from the application into an e-mail, and then sent to the person conducting the verification. With this approach, it would require a lot of work for the API to fetch the information e-mail. The solution became instead to create an Excel-template for the SE-tool file.

The Excel-document is attached to the mail and then downloaded to the local computer. The user can then fetch the file using a GUI-window, section 4.2 presents the GUI-window, it passes the file into the Model domain of the API, and the SE-tool class extract the information. Figure 4 shows the table for *ECU 1*, the names and number are changed due to confidentiality. The live version follows the same structure as this example.

	A	B	C	D	E	F	G
1	**ECU 1**	**A25**	**A30**	**A35**	**A40**	**A45**	**A60**
2	NT	7584					
3	PartNo 1	7586	7586	7586	7586	7586	7586
4							
5	PartNo 2	7585	7585	7585	7585	7585	7585
6	PartNo 3	3480	3480	3480	3480	3480	3480
7							
8	PartNo 4	1787	1787	1787	1787	1787	1787
9	PartNo 5	7588	7588	7588	7588	7588	7588
10							
11	PartNo 6	7587	7587	7587	7587	7587	7587
12	PartNo 7	7590	7591	7592	7593	7594	7398
13							
14	PartNo 8	7589	7589	7589	7589	7589	7589
15	PartNo 9	1788	1788	1788	1788	1788	1788
16							
17	PartNo 10	1789	1789	1789	1789	1789	1789

Figure 4: The template for ECU 1 in the SE-tool excel-document

3.2 Roadmap (excel-sheet)

The Roadmap is a large Excel-document used to map existing software implemented on the ECUs installed on the engines. It has more or less the same process to begin its operations as for the SE-tool. The file gets passed to class as an object, and its converted into a *XSSFWorkbook*-object. The difference with the Roadmap-file is that it consists of multiple sheets, were each sheet represent a machine configuration. The structure of the Excel-document showing *ECU 1* for the machine A45G and A45Gfs can be seen in the figure 5

	A	B	C	D	E	F	G	H
1	**Roadmap ART ESW A45G + A45G_FS**						ART X	
2	Yellow = New Part No.					*Release:	X	
3							X	
4	Status info in columns: DRAFT = Not released yet					Bucket week	12345	
5						KOLA :	X	
6						PROST :	X	
7		Field	Parameter			SW-objekt, ESW:	12345	
8						PLA, SE-Tool	ART 12345	
9						External ECU reference	A45	
10						Comment:		
11	ECU	No	Code	Value		Caption		
12	ECU 1						12345	
13								
14					NT		7531	
15								
16		X			PartNo 1		1430	P01
17					PartNo 2		7778	01
18								
19		X			PartNo 3		7779	P01
20					PartNo 4		7780	01
21								
22		X			PartNo 5		7532	P01
23					PartNo 6		7457	01
24								
25		X			PartNo 7		7460	P01
26					PartNo 8		7459	01
27								
28		X			PartNo 9		7465	P01
29					PartNo 10		5442	01
30								

Figure 5: The template for ECU 1 in the Roadmap excel-document

Note that the information displayed in the document is marked up to only present one ECU. The live version contains more ECUs. However, the structure remains the same, and the information provided in the figure is enough to explain the verification process that is conducted by the API. Using the Apache POI Java library the information in the Excel-table can be extracted, making it possible to run the verification.

3.3 Configurator (ESW-tool)

The Configurator is an ”in-house” application communication using the web, Volvo CE describes as:

SW Configurator is an application which allows PD (Product Development) to document the electronic architecture of a truck and release this information to Manufacturing and Aftermarket.

The Configurator holds information about the software running on the ECUs. The functionality to extract the currently installed software, or the previous,

to an Excel-file is already implemented to the Configurator. The downside is for each ECU one Excel-file gets generated. For this project that implies three different Excel-files is used to verify the three ECUs. But, they have the same structure, shown in figure 6, so all the files uses the same class in the API to run the verification.

	A	B	C	D
1	Version Name			12345
2	NTP Doc. Number			7531
3	Template Status			X
4	Issue Status			X
5	Main Object			
6	Transition			X
7	Release			X
8	Allow On Aftermarket			X
9	AM Introduction Date			X
10	AM Withdrawal Date			
11	DCN			X X
12	DCN Release Week			X
13	DCN Introduction Week			
14	Assembly Parts			
15	Compatible HW			X
16	PartNo 1 - PartNo 2		MOD-30/MOD-35/MOD-25/MOD-40/MOD-45	7779.7780.01
17	PartNo 3 - PartNo 4		MOD-30/MOD-35/MOD-25/MOD-40/MOD-45	7532.7457.01
18	PartNo 5 - PartNo 6		MOD-30/MOD-35/MOD-25/MOD-40/MOD-45	7460.7459.01
19	PartNo 7 - PartNo 8		MOD-30	7462.5442.01
20			MOD-35	7463.5442.01
21			MOD-25	7461.5442.01
22			MOD-40	7464.5442.01
23			MOD-45	7465.5442.01

Figure 6: The template for ECU 1 in the Configurator excel-document

3.4 Kola (PDM-tool)

Kola is a Product Data Management system. It doesn't only contain information on software implemented on the ECU. It has information about the complete machine, i.e. all the components that build the machine. So this application is used by employees throughout the entire company, e.g. engineers, designers and so on.

The components to be verified is the same as the other applications, software implemented on the ECUs. The application has a "Table of Content". It displays information such as software implementations on ECUs, the *PartNo* and the part-numbers. This table gets extracted to an Excel-file, the file is referenced to the API for the verification. As with the Configurator, each ECU has separate table resulting in three separate Excel-files for the three *ECU*s. The structure, shown in figure 7, is the same for all the Excel-files, so the API only has one class to extract and verify the different ECUs from Kola.

	A	B
1	Number	Name
2	7532	SOFTWARE, PartNo 1,ECU1,A25-45G
3	7534	SOFTWARE, PartNo 1 ECU2,A25-45G
4	7536	SOFTWARE, PartNo 1 ECU3,A25-45G
5	7537	DATASET, PartNo 2, ECU3, A35G
6	7538	DATASET, PartNo 2, ECU3, A40G
7	7539	DATASET, PartNo 2,ECU3, A45G
8	7531	NODE TEMPLATE, NT, ECU1,A25G-A45G
9	7533	NODE TEMPLATE, NT ECU2 A25-A45G
10	7535	NODE TEMPLATE, NT, ECU3,A25-45G

Figure 7: The template for ECU 1 in the Kola excel-document

4 API Development

This section gives presentations on the development and implementations for the API. It also provides a presentation regarding the setup but also the structure of the API to meet the primary object, developing the API to be better scalable and have reusable components resulting low coupling between the API components. To realise the goals of scalability and low coupling the Model-View-Controller (MVC) pattern is used to structure the API.

4.1 The Model Domain

The Model components are essential for the API. It's in the Model domain the extraction of data from the Excel-documents is made to run the verification process. Figure 8 displays the Model domains classes and packages.

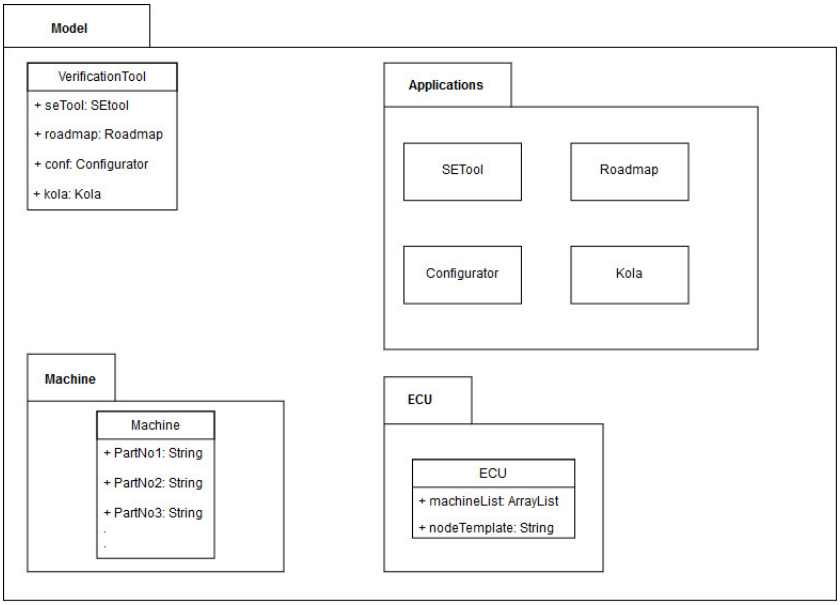

Figure 8: Packages and classes in the Model domain

4.1.1 ECU.class and Machine.class

To verify that each machine has the correct part-numbers throughout the applications a class is created to represent the machine. The machine class is

created in a separate package. The machine class have fields for the model-name and the part-numbers. Each of software is translated to a part-number in the applications at Volvo. ECUs are also created and then added to a separate package. An interface, *ECUInterface*, is created and implemented by the existing ECUs but also future ECUs. An ECU-class contains of a list, *ArrayList<Machine>*, stores machine-objects, but also a template-number. The first step when initialising the Model components is to create the different ECU-objects. Figure 9 bellow illustrates the associations between the machine-class and the ECU-classes/interface. From the figure we can see the ECUs are independent components, so adding new ECUs would not require any changes to the existing ECUs or the machine. Same is true if reversed, changes to the machine-class don't affect the ECU-classes.

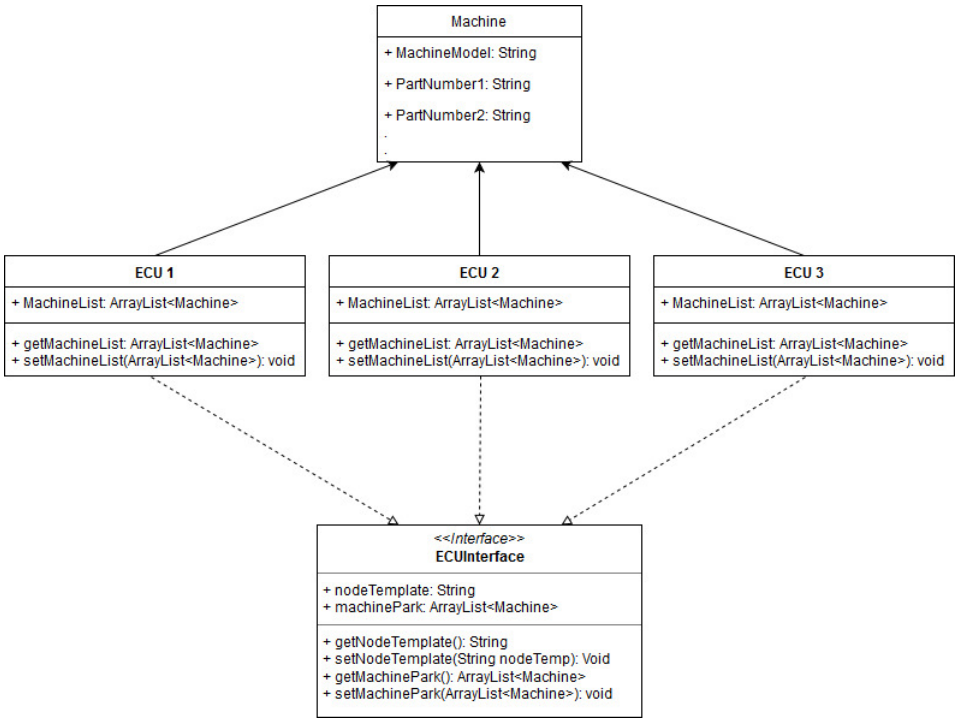

Figure 9: Machine-class, three ECU-classes and the ECU-interface

4.1.2 Verification-Tool.class

The *VerificationTool* class can be regarded as the "main" class for the Model. It's the class in the Model domain that communicates with the Controller domain. At initialisation, its constructor creates the ECU-objects and the machine-objects. The machines are added to the ECUs *ArrayList<Machine>* so that they can be passed on to the application classes. The application classes

are initiated when they are to begin their execution. So for every verification process creates a new application object whereas the ECUs and machines remain the same.

The Controller receives the user input and Excel-documents, to start a new verification process the function *public void runVerificationTool(File SEToolFile, File roadmapFile, File configuratorFile, File KolaFile)* in the *VerificationTool* is called from the Controller. From the arguments in the function, there is a referenced file-object for all the application. The *runVerificationTool(...)* will always start the verification process for all the application classes. However, if no, e.g. *configurationFile* has been referenced by the user the Controller handles it, first by making a file-object that has its file-path set to an empty string. Then the individual application classes check the file-path whether it's empty or not. If the file-path is empty, the application class won't start the verification process.

The *VerificationTool* class gathers the result from the different applications classes and build it into a complete verification result, same goes for the failure counter. When an application-class, e.g. the SE-tool class, is done with its operations the result is fetched and appended into a string in the *VerificationTool* class. With their respective operations completed for the application classes, so the Controller and View classes can fetch it and display to the user. Failure counter follows the same procedure, its gathered into a single counter variable and receivable to the other two domains.

4.1.3 Application classes and interface

In the package *Application*, classes are implemented to represent the applications at Volvo and to deal with the data from the respective application. Each application has a separate class since the structure of the generated Excel-documents differs in the placement of the information and size. One interface has to be implemented by the *Application* classes, *VerificationMessageInterface*. It deals with how the applications present the results of the verification. The classes implemented are named the same as their respective application.

For the user to get the result of the verification process a standard was needed to store the results, and then for the view and controller to access the information to present it. The interface has two functions:

```
public interface VerificationMessageInterface {

    public String getVerificationResult();

    public int getVerificationFailureResult();

}
```

The application classes implement the functions, so the result of the verification can be fetched and displayed, same for the number of errors that occurred. How to gather and build the report for the result and failure count are not included in the interface. The reason is to keep the results for each application-class private and unaffected by external factors.

4.1.4 SE-Tool.class

The *VerificationTool* class runs the function *readSEToolEcxelFile(File seToolFile, ECU1 ecu1, ECU2 ecu2, ECU3 ecu3)* to start the SE-tool class operations. With the Excel-file referenced as a parameter, it's possible to create new *XSSFWorkbook*-object. The *XSSFWorkbook*-object enables the API to iterate and fetch the content in the Excel-file. The first iterator iterates over the sheets in the Excel-document. The sheets are retrievable with a function *getSheetAt()*. The wanted sheet is fetched, and a new iterator is then used to iterate through the rows of the sheet. The iterator extracts the rows and convert them to objects, and with a third iterator extracting cells from the row-objects. The cells are fetched and converted to objects, to enable reading of the cell content. The content in the cells is extracted and converted into strings, using the function *cell.getStringCellValue()*. In figure 4, section 3.1, the API continue its iteration to find the cell with the string *ECU 1*. If found, the API knows that a table for *ECU 1* follows on the following rows. It also knows what ECU-object to add the content to, in this case, *ECU 1*. The SE-tool class differs from the other application classes, its content acts as the hindsight, so the content is only extracted and added to the machines in the ECUs *ArrayList<Machine>*. It makes no verification on the content.

After the ECU string have been found the correct ECU *ArrayList<Machine>* can be fetched. The API continues iteration of the rows in the table. It's now searching for the *PartNo* cells. If-statements matches the *PartNo*, and adds the content in the upcoming cells to the correct fields in the machine. The algorithm bellow displays the process of finding the correct *PartNo* and extracting value from the cell and add it to the machine *PartNo* filed. The process is the same for all the *PartNo*.

```
Function 'setPartNrToMachine' param Cell 'cell',
                                ArrayList<Machine> 'machineList',
                                String 'ecu'

    IF 'counter' is not 0

        IF 'ecu' is "PartNo 1"
            Call 'setPartNo1' method with args 'cell', 'machineList'

        ELSE IF 'ecu' is "PartNo 2"
            Call 'setPartNo2' method with args 'cell', 'machineList'
        .
        .
        .

    ELSE
        increment 'counter' by 1;

END 'setPartNrToMachine'


Function 'setPartNo1' param Cell 'cell',
                        ArrayList<Machine> 'machineList'

    Initialize index is set to 'counter - 1'

    IF cellValue is ""
        get machine at 'index' in 'machineList'
        Call 'machine.setPartNo1' with arg ""

    ELSE IF 'counter' less then 7
        get machine at 'index' in 'machineList'
        Call 'machine.setPartNo1' method with arg 'cellValue'

    ELSE IF 'counter' larger then or equals 7
        get machine at 'index-6' in 'machineList'
         Call 'machine.setPartNo1' method with arg 'cellValue'

    ELSE
        'counter' is set to 0

END 'setPartNrToMachine'
```

When the SE-tool class is done adding all part-numbers to the machines the *VerificationTool* fetches the ECU-objects with *get* functions in the SE-tool class. The ECU-objects is then passed on to the other applications so they can access the various ECUs *ArrayList<Machine>* to run their respective part-

number verification.

4.1.5 Roadmap.class

The operations in the Roadmap class begins with the *VerificationTool* initiate the function *public boolean readRoadmapExcelFile(File roadmapFile, ECU1 ecu1, ECU2 ecu2, ECU3 ecu3, String ESWDelivery)*. An additional parameter exists for this function, *String ESWDelivery*. The parameter is a string used to find the correct column in the referenced Roadmap Excel-document.

First, the process is more or less the same as for the SE-tool class. A *XSSFWorkbook*-object is created from the *roadmapFile*, the only difference is that the API makes a check on the file-object. If the file-object has an empty string as file-path, the API regards the file-object not having any reference to a Excel-document. After making this check, and a file is referenced then the sheet iterator starts and iterate over the sheets. As described in the method section (3.2) each sheet is a representation of the machine-model, resulting in that the API needs to iterate over all the sheets. Using a for-loop its possible to pick out the sheets one by one in order to then iterate over the rows and cells. The algorithm displays the iterator for fetching the sheets.

```
Function 'listSheets' with param XSSFWorkbook 'roadmapWb'

    BEGIN LOOP

        Iterate over the sheets in 'roadmapWb'

        Fetch next 'Sheet' from the iterator and store
        in 'sheet'

        Call 'setReadExcelCol' method with arg ""
        Call 'setESWArtValid' method with arg 'true'

        IF sheetName is "SEMS"
            END LOOP

        else:

            Call 'setSheetName' method with arg 'sheetName'
            Call locateESWDevArtNr method with arg 'sheet'

END 'listSheets'
```

The extracted sheet is passed to the next function in operation, *private void*

locateESWDevArtNr(Sheet sheet). The *ESWDelivery* is searched after by extracting the sheet content using iterators for the rows and cells. If the *ESWDelivery* is found, the same column stores the information to verify. Therefore, the cell-address is fetched, using the function *cell.getAddress().toString()*, later trimmed so the letters of the column can be extracted and stored in a global variable.

Locating the string values for the ECU is the next step after storing letters for the column. The process is the same as before, using row and cell iterators the content of the cells are extracted and matched with if-statements. When locating the string for an ECU, the API fetches the that ECUs *ArrayList<Machine>*. The iterations continue until the cell content equals any of the *PartNo*. With the *PartNo* and using the column address form the global variable its possible to locate the part-number. By reading the current sheet-name the API knows which machine-model to fetch from the ECU *ArrayList<Machine>*. The part-number stored in the field for the machine-object is compared to the content of the cell. If they are equal, the Roadmap-document have the correct information. Otherwise, the information is wrong. Either way, the result gets added to the verification report.

4.1.6 Configurator.class

The operations in the Configurator class is started with the function *public boolean readConfiguratorExcelFile(File configuratorFile, ECU1 ecu1, ECU2 ecu2, ECU3 ecu3)* from the *VerificationTool*. A check is made on the *configuratorFile* to check if the file-path is a empty string, if so the API makes no further operations.

The referenced file gets converted into a *XSSFWorkbook*-object and iterators are used to iterate over the content. The Excel-document only consists one sheet. Therefore the sheet iterator isn't used in this class. Instead, with the function *getSheetAt(0)* the API extracts relevant sheet.

The first task for the Configurator-class is locating what ECU to verify. In the Excel-document there is no reference to the ECUs except for a nodetemplate-number, with this number, saved as a variable for the ECUs, a match is possible. The nodetemplate-number is located in the Excel-document and compared to the stored value in ECUs. If equal, the API fetches the wanted ECU *ArrayList<Machine>*, and the verification of the part-numbers can start.

As with the Roadmap-class, the API iterates over the cells to first locate the different *PartNo*. After founding a *PartNo*, the next step is locating which machine to have its part-numbers verified. As seen from the figure 6, section

3.3, truncation of machine names into the same cell is made if multiple machines have the same part-number. If the part-numbers differs, the machine names are spread out on individual rows. For the API to know the number of truncated machines that have the same part-number(s), the string is trimmed. By using the string function *machineStr.split("/")*, finding a backslash splits the string, the trimmed parts are stored in an array. The size of the array gives how many machine-models have the same part-number.

```
Function 'splitMachineModels' with param String 'machineStr'

    New String array 'machinSplit' with a split at
    char "/" of 'machineStr'

    New String array 'temp' with size set to
    'machineSplit' length

    FOR LOOP from 0 to 'machineSplit' length with i

        New String 'str' to 'machineSplit' at position i
        New String 'tempStr' to empty string

        FOR LOOP from 0 to 'str' length with j

            IF character in 'str' at position j is digit

                add character at j position 'str' to 'tempStr'

            END IF

            add 'tempStr' to 'temp' at position i

        END LOOP
    END LOOP

return  'temp'

END 'splitMachineModels'
```

After extracting the machines-models and the number of machines having the same part-number its possible to run the verification. The Excel-document always has the same structure, so after the cells containing the machine models, the cells with the part-number follows. First, by extracting the part-number from their cells and then trim the cell content, this is because they are two part-number merged into one string. It's a similar process as the previous. The difference is that instead of using the split-function the chars are extracted one by one using a while-loop, counter and *charAt(counter)* to build the new strings. As long as the loop don't fetches a dot the chars are retrieved. When a

dot is found the previous fetched chars builds the first part-number. To obtain the second part-number a second function continues the counter and extracts the chars until the next a dot appears. Finding the second dot results in the second part-number is also obtained.

With the *PartNo*, machine-model(s) and part-numbers the verification can be made. First, by retrieving the corresponding machine from the wanted ECU *ArrayList<Machine>*, and then make the comparisons. The verification stores the result of the comparison, and the *VerificationTool* fetch the result completing the operations for the Configurator-class.

4.1.7 Kola.class

The *VerificationTool* starts the operations for the Kola-class with the function *public boolean readKolaExcelFile(File kolaFile, ECU1 ecu1, ECU2 ecu2, ECU3 ecu3)*. First, the API checks the file-path for the *kolaFile* to see if the user have referenced a file. If the file-path is not equal to an empty string the Kola-class operations starts.

A new *XSSFWorkbook*-object is created from the referenced file and the first sheet is fetched using *kolaWb.getSheetAt(0)*. The Excel-document for the Kola-class only has one sheet. In the method section, section 3.4, the figure 7 shows the structure for the Excel-document referenced. There are only two columns used. Column ”A” contains the part-numbers and nodetemplate-number. Column ”B” contains a string where the *PartNo*, ECU and machine-model(s) information are all merged to one. Also, because of the structure for the Excel-document, the sheet needs to be iterated over two times. The first iteration extracts the referenced ECUs in the document. When the ECUs are extracted the second iterator is used to extract the *PartNo* and to run the verification process.

The first iteration is to locate the referenced ECUs in the document. The cell content form the cells in the column ”B” is extracted and with if-statements the program checks if the string contains a ECU, e.g. *ECU1*. If the program finds a cell that contains, e.g. *ECU1* the content of column ”A”, on the same row, is the nodetemplate-number for *ECU1*. The nodetemplate-number stored in a ECU-object is accessed and compared to the extracted string of the cells. If the contents are equal a boolean value is set to *true* which includes the ECU in the verification process and its *ArrayList<Machine>* is used for the second iteration. When the ECUs referenced in the document is located and verified from the first iteration, the second starts.

The process for the second iterator is more or less the same as the first. The difference is that the second searches for the different *PartNo* using a func-

tion that makes string comparisons. Meaning that the program searches for the *PartNo*-names in the cell content. When finding a *PartNo* the function returns a new string that represents the located *PartNo*. First, the program finds the concerned machine-model(s) before the verification process, also achieved with a string comparison. A function takes the cell content that is extracted from column ”B” and uses if-statements to check for the machines names. Finding the string *A25G-A45G* means that the part-number extracted from the column ”A”, on the same row, is equal for all the machine models. The function then returns the integer ”6”, to enable access to the machines in the ECU *ArrayList<Machine>* using a for-loop. If finding a machine name, e.g. *A40G*, the extracted part-numbers is only used by that specific model. The function then returns the integer ”3” for the *A40G*. Since the integer is less the six, extraction of one specific machine-model in the *ArrayList<Machine>* is made. The returned integer then represent the index for the machine in the list.

```
Function 'locateMachineToBeVerified' args cell

    IF 'cellVaule' contains "A25-A45G" or "A25G-A45G"
     return 6;

    ELSE IF 'cellVaule' contains "A25G"
     return 0;

    ELSE IF 'cellVaule' contains "A30G"
     return 1;

    ELSE IF 'cellVaule' contains "A35G"
     return 2;

    ELSE IF 'cellVaule' contains "A40G"
     return 3;

    ELSE IF 'cellVaule' contains "A45G"
     return 4;

    ELSE IF 'cellVaule' contains "A60G"
     return 5;

    ELSE
     return 6;

END 'locateMachineToBeVerified'
```

With the machine index returned, the concerned machine can be fetched from the *ArrayList<Machine>* so its the part-number fields can be accessed. The API then makes the comparison of the machine fields and the extracted value

from the Excel-document. The result of the comparison is added to the verification report for the class, later fetched by the *VerificationTool*.

4.2 The View Domain

The view domain is the component implementing the user interface, for this API a GUI-window. By using the JavaFX library, part of the Java API, its convenient to develop a GUI-window with Java code.

Buttons, in the GUI (Graphical User Interface), enable the user to chose files that is to be referenced and be a part of the verification process. There are text-fields connected to each button displaying the file-paths for the selected files. When the user has selected the files that are to be included in the verification the ”Run verification”-button send the files to the controller, starting the verification process. To display the results from the verification the results gathered by the *VerificationController* is fetched and presented in a text-area in the GUI-window. With a text-label above the text-area, displaying the failure results from the verification process.

The GUI-window is implemented and started in the *VerificationGUIView*-class, the class contains the Java **main** function. The main function creates the *VerificationController*-object and the *VerificationTool*-object used to fetch the user inputs and perform the verification. The main function also launches the JavaFX application that starts the GUI by calling the function:

```
public void start(Stage primaryStage) throws Exception {...}
```

In the *start*-function the layout and all the components such as buttons and text-fields are created and added to the GUI. Figure 10 shows the layout of the GUI-window.

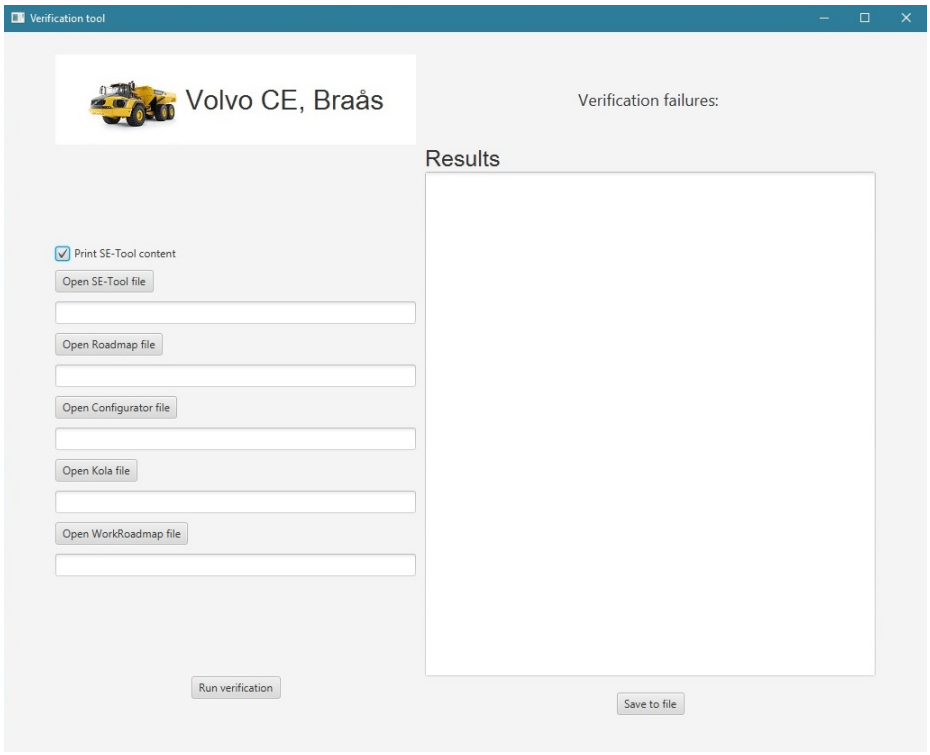

Figure 10: GUI layout

Creating the GUI layout is achieved by stacking different layers on top of each other. The base layer for the GUI creates a grid, using the *GridPane* form the JavaFX library, where its possible to add new layers or components into individual cells that builds a grid. In this case, different private-functions implemented in the *VerificationGUIView* returns new layers that have the button and fields added to them. The list below shows how to add new layers to specific positions in the grid layout.

```
gridPane.add(this.addTitleField(), 0, 0);

gridPane.add(this.addVboxFileChooser(), 0, 1);

gridPane.add(this.addFailureLabel(), 1, 0);

gridPane.add(this.addVboxConsole(), 1, 1);}

gridPane.add(this.addRunButtonLayout(), 0 , 2);

gridPane.add(this.addSaveButtons(primaryStage), 1, 2);
```

4.2.1 addTitleFeild()

The title is designed with a image and label. The two components are then added to a *GridPane*, the pane is returned and added to the ”main”-grid of the GUI layer. The label is a *Label*-object and with the function *label.setText(”Title”);* the label display the text string. The font is changed for the label with the function *label.setFont(new Font(”Arial”, 30));*.

The image is a ”.jpg” and in order to display it a *FileInputStream* is used to first read it. A image-object is created using the input-stream variable as argument, *Image image = new Image(imageStream);*. The new image is then added to a *ImageView*-object, the *ImageView* is the object that is added to the grid, *pane.add(new ImageView(image), 0, 0);*.

4.2.2 addVboxFileChooser()

This function builds the buttons and the text-fields that the user uses when selecting Excel-documents to be included in the verification process. A *VBox*-object is used to construct the layout, it takes the components and adds them to a downward growing list. Four buttons are created named after the different applications. Under each button, there is a text field that displays the file-path.

Using an *EventHandler* for each button it’s possible to register user interactions. Setting up the *EventHandler* is the same process for the all buttons, therefore, this example with the *SE-tool*-button is representative for the rest. The events are set up by first passing the button and text-field into a new function, *seToolBtnEvent(seToolBtn, seToolTextField)*:

```
private void seToolBtnEvent(Button btn, TextField textField) {

    btn.setOnAction(new EventHandler<ActionEvent>() {

        @Override
        public void handle(ActionEvent event) {
            textField.setText(selectFile(stage, "SE-TOOL"));
        }
    });
}
```

The user presses a button to select a file and the program then sets the file-path to the text-field. The *selectFile(...)*-function uses a *FileChooser*-object to open a dialog window for the user, to enable selection of an Excel-document. When a document is selected a *File*-object is created, and the file-path is returned to

the text-field. The File-object is passed into the controller, via the function *sendFileToController(selectedFile, ApplicationName)*, along with the application name passed on from the *selectFile(...)*-function.

Creating a *CheckBox*-object enables the user to toggle whether to print the result message for the *SE-tool*-file or not. The new components get added to the layer, then returning the *VBox*-layer to the main grid.

4.2.3 addFailureLabel()

This function creates a label and adds it to a *Vbox*-object. Displaying the failures, gathered from the verification process, is the purpose of this label. The label shows a counter that has recorded failures during the process. In the function, the label is set up and then set to not visible until the failures should be displayed. After adding the label to the *VBox* it's returned to the main grid.

4.2.4 addVboxConsole()

The returned *VBox*-object from this function contains a label and a textarea. The label is set to a fixed string, *label.setText("Result")*. The textarea presents the verification result in the GUI, and its created from a *TextArea*-object. A fixed size for the textarea is set, and the functionality for editing text in the textarea is disabled, using the function *consoleText.setEditable(false)*.

4.2.5 addRunButtonLayout()

This function creates two buttons, one to run the verification, a second to clear all text-fields and the text-area. The two buttons are added to a *VBox*-object and returned to the main grid. The two buttons don't differ from the application-buttons regarding the setup. The difference is the set up for the *EventHandler*. The *EventHandler* for the button that starts the verification is setup via the function *runBtnEvent(runBtn, clearBtn)*, displayed below.

```
Button "Run verification" Call 'runBtnEvent'
                                    with args 'runBtn', 'clrBtn'

Function 'runBtnEvent'

    IF 'SEtool_textField' is ""
        print to 'textArea' "No SEtool file referenced"

    ELSE

        IF checkBox is 'false'
            set 'printSeToolResult' to 'false'

        END IF

        Call 'runVerifcationBtnPressed'
                                    in 'verificationController'

        set 'textArea' with Call 'getVerificationResult'
                                    in 'verificationController'

        set 'failureLabel' with Call 'getVerificationFailure'
                                    in 'verificationController'

        set 'failureLabel' visible 'true'

        set 'clrtBtn' visible 'true'

END 'runBtnEvent'
```

The first operation after pressing the "Run" button is checking for a referenced *SE-tool*-class file. The *SE-tool*-content is the hindsight for the verification of the application-classes, a referenced file is required. If there is no reference to a *SE-tool*-file, the API won't start the verification process.

If the user has referenced a file the first operation is to check the status of the check-box. The status tells the *verificationController* whether to fetch and print the content from the *SE-tool*-file. The Controller starts the verification with *verificationController.runVerificationButtonPressed()*, and this starts the operations in the *VerificationTool*. When the verification has been completed the results are fetched from the Controller and displayed in the textarea using the function: *consoleText.setText(verificationController.getVerifiResult())*. The Controller also fetches the failure counter using: *failLabel.setText("Num of verifi failures: "+verificationController.getVerifiFail())*. If the user wish to clear all the content, textarea and textfields, the "clear" button is enabled.

Using the ”clear” button clears all the printed content in the fields if needed. The button sets the application textfields back to empty strings and the same for the textarea. The button is also set to not visible again. The button also tells the Controller to reset all the file-objects, using the function *verificationController.resetVerification()*, results and failure counter that it has fetched.

4.2.6 addSaveButtons(primaryStage)

If the user wishes to save the results from the report a button is available for that after completion of the verification. The button is created and added to an *HBox* that in turn is added to the main grid. The button is set up similar to the application buttons. The *EventHandler* opens the *FileChooser* so the user can select the location to store the report-file. Pressing the button first runs a check whether the textarea is empty, if empty a report-file can't be created. If the textarea has content a *FileChooser*-object is created. It opens the dialogue window using the function *fileChooser.showSaveDialog(primaryStage)*. A file-object is created when the user adds a new file in the dialogue window. With a *FileOutputStream* the content in the textarea is written, *fileOutput.write(consoleText. getText().getBytes())*, to that file-object.

4.3 The Controller Domain

The Controller is the unit that takes the requests from the user and starts the operations in the Model domain. The Controller developed for this API, *VerificationController*, gets the referenced files. When pressing the ”Run verification” button the API checks the textfields for selected files. The *VerificationController* have four file variables that are initiated to have empty strings as file-paths in the *VerificationController*-constructor.

```
private File seToolFile;

private File roadmapFile;

private File configuratorFile;

private File kolaFile;
```

When the *VerificationController* gets the referenced file-objects it checks the file-paths. If the file-path equals an empty string, its assumed the user hasn't

selected a file. The file-path(s) then remains as empty strings when referencing the file-object into the *VerificationTool*. If the selected file has a file-path the corresponding file variables in the *VerificationController* is set to the selected files. With the file-objects checked they are passed to the *VerificationTool* using the function:

```
Function 'runVerificationButtonPressed'

    IF 'isPrintSEtoolResult' is 'false'
        Call 'verificationTool.setPrintSeToolResult' args 'false'

    END IF

    Call 'runVerificationTool' args 'SeToolFile',
                                    'RoadmapFile',
                                    'ConfiguratorFile',
                                    'KolaFile'

    set 'verificationResult' with Call 'verificationTool
                                        .getVerificationResult'

    set 'verificationFailure' with Call 'verificationTool
                                        .getVerificationFailure'

END 'runVerificationButtonPressed'
```

When called the operations starts in the model domain. The function first checks the state of the SE-tool "print-button". The user can choose to print, or not print, the content of the SE-tool file. The *VerificationTool* is started with the file-objects as arguments. When the function *runVerificationTool(...)* is done with its operations the results and failures are fetched from the *VerificationTool* and stored in the *VerificationController*, this enables the view domain classes to get the results and display them.

5 API Evaluation

In this section, I will evaluate the development and the result of the API. I divide the evaluation into three parts where I discuss the API in three stages: design, development and end product.

5.1 API Design Evaluation

The incentive for developing this API was to deal with the time-consuming process of doing comparisons of part-number between applications at Volvo. The inspiration for the design of the API stems from research made regarding *Enterprise Application Integration (EAI)* and the *Enterprise Service Bus (ESB)*. The research on these two fields has shown good results regarding improvement of integration between applications and systems. But also decrease resource allocation for maintenance, development and work hours. However, the scope of this thesis, section 1.5, was set due to the time limitation, that the API would not be developed to be a full integration solution between the applications.

The focus for the API was instead on removing the manual comparing of part-numbers when making the verification between information in the different applications. The API was designed with the *ESB* in mind, meaning that the API should be able to scale well with the addition of more applications and the coupling between the components building the API should be low to make the API components reusable. The *Model-View-Controller (MVC)* pattern was used to accomplish low coupling between the components.

The use of the MVC pattern divides the API into three domains. The Model domain, section 4.1, is the component that resembles the *ESB*. It builds a layer for each application that enables the extraction of information to be correctly made from the user. Regarding the data from the applications referenced as Excel-documents to the API. The problem with first extracting information into Excel-documents and then via the GUI passing them into the API with the file dialogue window is that the human factor is still present in the verification process. The API components are built to read the content of the excel-files and will not make further checks if the referenced files are correct. It's still up to the user, resulting in a source of error still exists in the early part of the verification process. For the API to delete the manual component completely, it needs access to the applications information or file systems automatically, enable it to extract the data. However, the obtained information in the excel-files still made it possible to make the verification for the respective applications using the open-source software Apache POI [5], that proved to be very useful.

5.2 API Development Evaluation

The large focus for the API is in the model domain since it contains the application classes and it runs the verification. As said above the design for the model was inspired by the *ESB* so the class *VerificationTool* is implemented to acts as the middleware for the *Applications*. Meaning that referenced excel-files is directed to the *Application*-classes by the *VerificationTool*, it also extract the result from the *Applications*.

Dividing the Model into three packages structures the program in a more readable way, with the *VerificationTool*-class as the middle point connecting the packages. The three packages are, the *ECU* packages that contain classes for the different ECUs, the *Machine* package that includes the machine class and the *Application* package that includes the application classes. Resulting in the *VerificationTool*-class having high coupling to the classes in the three packages. The trade-off was to try for a lower coupling between the packages instead. But a high coupling between the ECU classes and the application classes could not be prevented. The reason for this is all the application classes have the ECUs as parameters. To make the verification the application classes needs the ECUs *ArrayList<Machine>*. One adverse effect is the when adding more ECU classes to the existing applications. They would require updating if they are to verify the new ECU class. However, the application classes are unaffected of each other. Changes is possible in the, e.g. *SE-tool*-class without any consequence or modification to the other existing application classes.

To satisfy the goal of good scalability the separation of the *Application* classes is a needed feature for the API. This ties to the *ESB* presented in section 2.3. Adding a new *Application*-class can be done without affecting any of the existing *Applications*. The update needed is a function in the *VerificationTool* that starts the operations for the new *Applications*. The controller and the view need modification, so referencing files into the class from the application on Volvo is possible. To make the process of adding new *Application*-classes less complicated by implementing an interface. Excluding the internal development of the new *Application* class only a couple lines of code across the three domain components is required. In my mind satisfies the goal of scalability regarding the addition of *Application* classes. The scalability for the *ECU*s becomes more complicated since the existing *Applications* needs modification to implement it.

The development for the remaining two domain classes, the *VerificationController* and the *VerificationGUIView* didn't introduce any significant complexity. Since the MVC-pattern is set to keep the logical and operation parts in the Model domain, the remaining two domains more or less passes and fetches data from in this APIs the *VerificationTool* class in the Model domain. The purpose of the Controller is to take the files that the user references. When the

user wishes to run the verification, the Controller instructs the Model, using the *VerificationTool*, to begin execution. The Controller for this API checks the referenced files before the *VerificationTool* gets them. There is no particular reason to do the checks in the Controller more than testing the files before passing them to the *VerificationTool*. The checks could be made there as well.

To make the API more user-friendly, I decided to create a GUI. It makes it easier for a user to open the file-dialogue window reference the excel-files. The reason for creating a GUI is that the tool will be used locally at Volvo and the primary goal of this project was focused on the verification process, not on a user interface. The main propose of the GUI is to enable the user to select the files with a file dialogue window instead of typing the file-paths to a console window. The request from Volvo, regarding the GUI, was to present the result of the verification and the counter for the number of failures that occurred during the verification process.

5.3 API Evaluation

The goals of the API was to remove the human factor from the verification process (VP) to make the process automated. The incentive is to reduce the time effort required for the verification process but also reduce sources for potential errors. With this in mind, I would regard the API to reach the goals that was set up. I believe the API to be on the right course since the verification between the application-information is automated given that the referenced information is correct to the API. However, it's still the user that makes the extraction of data from the application into Excel-files. Resulting in a source of error from the start that is a crucial part of the process. To completely remove the human factor the API needs to be able to extract the data from the application automatically. But even if a completely automated process was out of the scope for the project, I believed more reduction of the human factor could be achieved. I would have liked, for example, to add some pattern recognition in the application classes enabling them to "know" the structure of the excel-file it expects. Then making checks to the referenced files before starting the VP. If the Excel-file isn't following the structure for that application class it could notify that the referenced Excel-file is wrong to the user, and the result of the VP can, therefore, be skewed. But, due to the time limitation of the project this implementation was left out from the application classes. As said with the *Enterprise Service Bus*, section 2.3, the layer, in this case, the application classes, the development can be complicated, resulting in a time-consuming development process.

I also set up some goals for the API that doesn't cover the problem presented by Volvo but was more focused on the design and structure of the API. I wanted to make the API as scalable and reusable as possible. The API uses

the *Model-View-Controller*-pattern for this reason. With the *MVC*-pattern I believe the API has good scalability. It is possible to add more application classes to the API without any significant complications, an interface for the applications helps with this. But it should be said that adding new ECUs is a more complex implementation. Since the current application classes need to be modified to handle the new ECUs. In the current version of the API, there are no interfaces for the applications to deal with the ECUs, each application class has its own implementation for this. In retrospect, interfaces for working with the ECUs could help for better scalability. However, it's hard to develop a software or API that has low complexity or coupling everywhere in the code. And the implementation of interfaces for adding new ECUs is also out of the scope for the project that Volvo presented that this thesis covers. The problem presented was to do verification with three ECUs, this resulted in that the addition of more ECUs wasn't a requirement for the API design.

6 Conclusion

This report covers the development of an API that has the goal of making a manual verification process between applications automated, this to reduce the time effort for the verification but also to reduce sources for errors. To succeed with the goal the API needs to improve integration between the applications to enable passing of information between the applications. The challenge this poses is the development of the applications have happened at different times, and by different developers. Volvo CE is not unique to this problem, businesses, in general, have older systems and applications vital to the daily work that have poor integration. One solution that has shown a lot of promise regarding the integration of older application is the *Enterprise Service Bus (ESB)*, the API uses this solution as inspiration.

The development of the API shows that to achieve a functional *ESB* complexity has to be dealt with, resulting in a time-consuming development process. For this reason could the API not be a complete integration solution that can make the whole verification process fully automated. There still exits moments in the verification process that needs user input. These moments are extracting information from the applications and pass the information into the API. However, with the correct data referenced the API retrieves the information from the applications and verify whether the applications displays the accurate information. This part was the most time-consuming moment of the manual process and it's now automated, resulting in a lot more efficient process.

The results of the API shows that its possible to improve the current situations at businesses that today work with older applications and wishes to make the daily work more efficient. With relative quick and straightforward means a more efficient process can be achieved. However, in the hope for a complete integration solution that "hides" the current applications behind an API the development starts to become more complicated. The time limitations forced compromises for this project. For example, developing a basic GUI-window the API, lack of interfaces for efficiently adding new classes and the API have to be feed with Excel-documents containing information from the applications.

References

[1] URL: https://msdn.microsoft.com/en-us/library/ff649643.aspx. (accessed: 02.05.2018).

[2] URL: https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm. (accessed: 17.05.2018).

[3] URL: https://www.office.com/. (accessed: 02.05.2018).

[4] URL: https://www.apache.org/foundation/. (accessed: 02.05.2018).

[5] URL: https://poi.apache.org/. (accessed: 02.05.2018).

[6] Rafael Z. Frantz and Rafael Corchuleo. "A software development kit to implement integration solutions". In: *SAC '12 Proceedings of the 27th Annual ACM Symposium on Applied Computing* (March 2012), pp. 1647–1652. DOI: https://doi.org/10.1145/2245276.2232042.

[7] Cay S. Horstmann. *Big Java: Late Objects*. John Wiley Sons, Inc, 2012. ISBN: 9781118087886.

[8] Lucas S. Mendonça et al. "Development of an Engine Control Unit: Implementation of the Architecture of Tasks". In: *2017 IEEE International Conference on Industrial Technology (ICIT)* (March 2017), pp. 1142–1146. DOI: https://doi-org.proxy.lnu.se/10.1109/ICIT.2017.7915523.

[9] Jieming. Wu and Xiaoli. Tao. "Research of Enterprise Application Integration Based-on ESB". In: *2010 2nd International Conference on Advanced Computer Control* (March 2010), pp. 90–93. DOI: https://doi-org.proxy.lnu.se/10.1109/ICACC.2010.5487292.