A XML-based diagnostic tool using the product line approach

Dejan Subotic
ABSTRACT

This is a Master Thesis at the Computer Science Program at Växjö University. It has been made at BSR in Växjö where the development of a diagnostic tool has been in the running since the beginning of 2008. Previously there was a base developed within the company which I was to use for developing the two layers that I was responsible for – ProtocolLayer and PresentationLayer. In the end it should lead to a XML-based diagnostic tool. The technologies used have been .NET with its language C# and XML. The future purpose for this tool is for it to be used within the company to receive important information about the cars when developing other BSR products.

This thesis has focused on developing the tool to be working with the car manufacturer VOLVO and its protocols. The idea is that the diagnostic tool in the future could be extended to be working with all possible car manufacturers’ protocols and to enlighten the extensibility the project has been done using the product-line approach.

Keywords: CAN, K-line, car diagnostics, Product-line, XML, VOLVO

SAMMANFATTNING


I första hand har det här verktyget gjorts för bilmärket VOLVO och dess protokoll men i framtiden är tanken att det ska kunna utökas och kunna användas till alla möjliga bilmärken och protokoll. För att detta ska underlättas har projektet gjorts enligt product-line-tekniken.

Nyckelord: CAN, K-line, fordonsdiagnostik, Product-line, XML, VOLVO
Table of Content

ABSTRACT ......................................................................................................................... ii
SAMMANFATTNING .......................................................................................................... ii

1. INTRODUCTION ........................................................................................................... 1
  1.1 Task description ........................................................................................................ 1
  1.2 Project description .................................................................................................... 2
    1.2.1 Goals .................................................................................................................. 2
    1.2.2 Project Model ...................................................................................................... 2
  1.3 Report description ..................................................................................................... 3

2. SUPPORTING TECHNOLOGIES .................................................................................. 4
  2.1 CAN – Controller Area Network ............................................................................. 4
    2.1.1 Introduction ........................................................................................................ 4
    2.1.2 What is CAN? ..................................................................................................... 4
    2.1.3 Interesting CAN properties for the thesis .......................................................... 5
    2.1.4 Frames ............................................................................................................... 5
    2.1.5 An example with CAN-messages ...................................................................... 5
  2.2. K-LINE ................................................................................................................... 6
    2.2.1 Introduction ........................................................................................................ 6
    2.2.2 K-line initiation .................................................................................................. 7
    2.2.3 Example on K-line communication ................................................................... 7
  2.3. XML ....................................................................................................................... 8
    2.3.1 Introduction ........................................................................................................ 8
    2.3.2 Important XML properties ................................................................................. 8

3. PRODUCT-LINE .......................................................................................................... 10
  3.1 Introduction ............................................................................................................... 10
  3.2 Decomposing software product lines ....................................................................... 10
    3.2.1 First dimension .................................................................................................. 10
    3.2.2 Second dimension ............................................................................................. 11
    3.2.3 Third dimension ................................................................................................ 11
  3.3 Initiating a product-line ............................................................................................ 11
    3.3.1 Evolving an existing set of products into a product-line .................................... 12
    3.3.2 Replace an existing set of products with a software product-line ...................... 12
    3.3.3 Evolving a new software product-line ................................................................ 12
    3.3.4 Develop a new software product-line ................................................................ 13
  3.4 Designing a product-line architecture .................................................................... 13
    3.4.1 Business case analysis ..................................................................................... 13
    3.4.2 Scoping .............................................................................................................. 14
    3.4.3 Product-line architectural design ...................................................................... 15

4. PROCESSING .............................................................................................................. 17
  4.1 Preparation ............................................................................................................... 17
  4.2 XML construction .................................................................................................... 17
  4.3 Developing the ProtocolLayer ................................................................................ 17
    4.3.1 Creating a product-line architecture ................................................................. 18
    4.3.2 Classes of the ProtocolLayer ............................................................................ 18
  4.4 Developing the PresentationLayer ............................................................................ 21

5. RESULTS ..................................................................................................................... 22
  5.1 XML ......................................................................................................................... 22
    5.1.1 Header ............................................................................................................... 22
    5.1.2 Message Syntax ................................................................................................. 22
    5.1.3 Ack Message ...................................................................................................... 23
1. INTRODUCTION

This work is a Master Thesis at the Computer Science Program at Växjö University. The work is done by me, Dejan Subotic, at the company BSR Svenska AB in Växjö. BSR – BilSpecialisterna Racing - is an internationally known tuning company which is based in Växjö but also has its offices in Stockholm and Malmö. BSR was named Super Company of the year in both 2006 and 2007. The main task of BSR is tuning cars engines but also ethanol converting cars is a part of the work. BSR are very environment-conscious and have been leading in some of the amendments made by the Swedish government in the last years.

BSR’s work is dependent on the possibility to communicate with cars and they are therefore in a great need of diagnostic tools. A diagnostic tools purpose is to communicate to the cars different modules, i.e. computers with all the information about the car. The user of a diagnostic tool is able to ask the modules all different kind of questions. In the process of optimizing a car’s engine a technician is in need of analyzing the data stored in the car’s different modules to evaluate which impact the changed data has made on the car’s performance. Today BSR use some of the diagnostic tools which are available on the market. These diagnostic tools are often bound to one manufacturer and are extremely expensive. This combination makes it very expensive for BSR to be in possession of all the different diagnostic tools. Therefore they are in a great need of a diagnostic tool of their own. To be able to develop a diagnostic tool you first have to analyze the protocols used by the car manufacturers. The communication with the car is mostly over the CAN-bus but also K-line is in some cases used. K-line is a serial bus which resembles to the RS232 protocol with the difference that it runs on other voltage levels.

1.1 Task description

BSR in Växjö decided to start a new project in the spring of 2008. It is named BSR Diagnostic Tool. This diagnostic tool’s purpose is to replace the need of manufacturer specific diagnostic tools and decrease the expenses of the company. The tool is planned to be developed using the .NET environment and the programming language C#.

My task is to develop a base to a diagnostic tool which should be used within the company. This diagnostic tool should be easy to extend – which will be shown in be presenting which steps have to be performed to achieve this - and later to be working with several different manufacturers’ cars. The idea with this diagnostic tool is that it should consist of three different layers, hardware layer, protocol layer and presentation layer. There is already an existing hardware layer within the company. The hardware layer handles the communication to the car. The purpose of this work will therefore be to work with the protocol and presentation layers. The protocol layer’s job will be to communicate with the hardware layer and later deliver some data to the presentation layer. The diagnostic tool should be XML-based which means that the protocol layer gets its data from a XML (eXtensible Markup Language) document. Each protocol, there can be several from one manufacturer, should have its own XML document. The protocol layer will assemble a message, extracted from the XML document, and send it to the hardware layer. When the hardware layer gets an answer it will notify the protocol layer which then tries to verify the message according to the format set by the XML document. Finally the protocol layer will return some data, depending on the verification, to the presentation layer. The presentation layer will basically be a GUI (Graphical User Interface) where the user will be able to control the tool and make some
settings adjustments. The different parts of the diagnostic tool will also be more clearly described in section 1.2.2.

I will work with one manufacturer, Volvo, but the goal is to make the tool easy to extend. The work of extending the tool to other manufacturers should, except for some minor trivial changes in the tool, basically consist of implementing a new protocol file and connecting it to the application. To achieve extensibility the diagnostic tool is implemented with the product-line approach.

1.2 Project description
The following part will give you an overview of the project as a whole, including my parts in it. The description has been provided to me by my tutors, Philip Albertson and Patrik Eklund.

1.2.1 Goals
Most of the existing diagnostic tools are bound to one manufacturer and are very expensive. BSR is therefore in a need of a diagnostic tool which is works with more than one manufacturer. They want me to develop such a diagnostic tool and it should be easy to extend. In the future they want to develop a hardware by them self to be independent of the Kvaser Tool which shall be used in this project. Mostly this tool will be used within the company but if it turns out to be a success it may also be sold to the resellers.

1.2.2 Project Model
Here the structure of the project is shown and the different parts are presented and described. Some of them shall be developed and of them already exist.

- **Presentation Layer**
  - Gives the user the opportunity to choose XML-file
  - Sends a command to the Protocol Layer telling what function should be run
  - Presentation of the answer messages
  - Gives the user the opportunity to make some settings

- **Protocol Layer**
  - Parses the chosen XML-file
  - Assembles the correct message to be sent
  - Verifies the answer and returns it to the User Interface

- **XML file**
  - Each protocol has its own XML-file which contains of the correct messages to be sent
  - Contains a format for the interpretation of the answer messages

- **Hardware Layer (already exists)**
  - Creates the communication to the car via the Kvaser interface
  - Sends the message it has received from the protocol layer
  - Receives the answer in hexadecimal format and translates it into a message which is returned to the protocol layer
- **Kvaser Memorator Professional II (already exists)**
  - Handles the communication with the CAN-bus
  - Contains a API which helps with initiation, sending and receiving of message over CAN-bus

- **K-line tester (already exists)**
  - A tool developed by BSR which is a modified PPC (Portable Program Carrier). It transforms voltage levels that differ between K-line and RS232

### 1.3 Report description

In chapters two (2) and three (3) I will describe the different technologies that have been used to develop this tool. I will only mention the parts of the technologies that have been relevant to make this tool and leave out the rest. The chapters following that will describe the process of development, show the result of the development and discuss the result.
2. SUPPORTING TECHNOLOGIES

In this chapter the supporting technologies used to develop this tool will be briefly presented. That doesn’t include the product-line, which is fundamental for this thesis.

2.1 CAN – Controller Area Network

2.1.1 Introduction

In 1983 the German company Robert Bosch GmbH – better known as just Bosch - started developing the CAN-bus. By 1986 the development was done and CAN was introduced to the market. The main reason for the development was the absence of communication between the control units in Mercedes Benz vehicles. Bosch wanted a reliable and fast communication protocol and none of the existing fitted the profile. With some help and involvement from the car manufacturer Mercedes and semiconductor manufacturer Intel they developed CAN and made it into a success story. Besides from speed and reliability CAN have the advantage of its use of serial communication which leads to reduced wiring. In areas with a lot of electromagnetic noise CAN has also proved to withstand and still deliver in a fast and reliable way. (Voss 2005: 8-14).

CAN became quickly very popular and today many large micro-processor manufacturers produce CAN-chips. This has led to lower prices. CAN’s properties have meant that several other products, besides cars, are using the CAN-bus. Some of them are listed in the table below. (Voss 2005: 15).

- Trucks, Busses and Off-Road Vehicles
- Passenger and Cargo Trains
- Aircraft and Aerospace Electronics
- Refrigerators
- Industrial Machine Control
- Building Automation
- Lifts and Escalators
- Medical Equipment and Devices

2.1.2 What is CAN?

CAN is a serial network technology which uses two wires, CAN-high and CAN-low. Between these two wires a voltage level difference is measured and is presented as a recessive level, one on 0V, and a dominant level, zero on 5V. (Voss 2005: 35). The maximum baud rate on the CAN-bus is 1Mbit/s and the traffic runs only in one direction at a time. 1Mbit/s may not sound much but the combination of very short messages, effective collision avoidance, error detection and fault confinement makes CAN much more suitable for real-time applications than for instance the TCP/IP protocol. TCP/IP can transmit in up to 100Mbit/s but it was designed for data transfers where timing isn’t mandatory. Other main properties of CAN are listed in the table below. (Voss 2005: 20).

- Bus access is based on Multi-Master priority
- Non-destructive contention-based arbitration
- Automatic retransmission of messages that lost arbitration
- Automatic retransmission of messages that were destroyed by errors
- Configuration flexibility
2.1.3 Interesting CAN properties for the thesis

CAN – as all other communication protocols - consists of a lot of data and properties which has to be there to uphold the communication. A lot of this data is not interesting for this thesis. The parts that were interesting are listed below and will be described a little deeper. These properties were used in the design of a class Message will be discussed further down in this thesis. All of these properties are part of the term frame.

- CAN-id’s
- DLC
- Data

2.1.4 Frames

In a CAN network all messages are referred to as frames. The information sent on the CAN-bus has to have the same structure as the predefined frames. There are four different frame types: data frame, remote frame, error frame and overload frame. The data frame is the only frame for actual data transmission. The remote frame looks almost the same as the data frame with the exception that it doesn’t transmit any data. Instead it requests a transmission of a message from another node. The error frame exists to present that an error has occurred on the network. The overload frame is almost the same as the error frame with the difference that the overload frame resends the last sent message. An overload frame is sent to indicate that a node is overburdened and can’t participate in the CAN-communication. (Voss 2005: 20, 38, 40, 57, 66-67)

![Figure: 2.1.4.1: CAN Data Frame Architecture](image)

A CAN network can be configured to be working with two different formats: the standard frame format and the extended frame format. The only difference between the two formats is that the standard format supports a length of 11 bits for the identifier, and the extended frame supports a length of 29 bits for the identifier, made up of the 11-bit identifier (standard identifier) and an 18-bit extension. They are no conflicts on the network if both of the formats are used in the same session. The identifier is stored in the control field of the frame. The control field also contains a DLC (Data Length Code) which is 1 byte. It has the value of 0 to 8 and indicates how much data is transmitted in the message. The data is zero to eight bytes in hexadecimal format. (Voss 2005: 47, 53)

2.1.5 An example with CAN-messages

In table 2.1.5.1 a communication session on the CAN-bus is presented and described in detail. The question asked to the car is for it to provide its VIN (Vehicle Identification Number). When plugging in to the cars OBD-II (On Board Diagnostics) outlet a lot of data is transmitted from the car. Most of it is the communication between the different modules and we are not interested in it. Therefore I have filtered out this data.
<table>
<thead>
<tr>
<th>ID</th>
<th>DLC</th>
<th>Data</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>726</td>
<td>8</td>
<td>03 22 f1 90 00 00 00 00</td>
<td>I am asking the module with the id 726 (CEM) for the VIN of the car. 03 is a byte counter, number of data bytes I am sending. 22 f1 represents that you want to readDataByIdentifier and 90 is the parameter for read VIN.</td>
</tr>
<tr>
<td>72e</td>
<td>8</td>
<td>10 1b 62 f1 90 59 56 31</td>
<td>The car answers on the id 72e. 10 mean that it’s an answer with more than 8 bytes i.e. more messages to come. 1b is a data counter; we are to receive 27 bytes of data. 62 means that it’s a positive answer (22+40 is positive, 7f would be negative). 90 again represent the read VIN parameter. At 59 the VIN data begins.</td>
</tr>
<tr>
<td>726</td>
<td>8</td>
<td>30 00 00 00 00 00 00 00</td>
<td>After received a message with the first byte 10 we send an ACK. An ACK means that we have received the first message and want the rest.</td>
</tr>
<tr>
<td>72e</td>
<td>8</td>
<td>21 42 57 35 36 35 31 38</td>
<td>The rest of the data. 21 is a counter and is not a part of the expected 27 bytes of data. Because it’s a multiple message answer it begins at 21 and runs. 42 and forward is VIN data.</td>
</tr>
<tr>
<td>72e</td>
<td>8</td>
<td>22 31 30 30 36 34 31 33</td>
<td>22 is the counter and the rest is VIN data.</td>
</tr>
<tr>
<td>72e</td>
<td>8</td>
<td>23 00 00 00 00 00 00 00</td>
<td>23 is the counter and the zeros fill out the rest so that we receive 27 bytes in total. The zeros are not part of the VIN.</td>
</tr>
</tbody>
</table>

Table: 2.1.5.1: An example of CAN communication

Often some kind of interpretation of the received data has to be done to extract the correct information. In this example the VIN is expected to be 17 bytes long and to be interpreted with the help of the ASCII-table (American Standard Code for Information Interchange). 0x59 becomes the letter Y and so on until you get YV1BW565181006413.

2.2. K-LINE

2.2.1 Introduction

K-line is a serial bus which much looks like the RS232 protocol. The difference is that K-line uses other voltage levels. Because of the similarities a short introduction to the RS232 protocol will be given.

The RS232 standard is one of the oldest communication standards in the computer world. It is an asynchronous serial communication method. The word serial tells us that the information one bit at a time. Asynchronous means that the information is not sent in predefined time slots. The data transfer can start at any given time and the receiver has the task to detect when the message starts and ends. The data bits are sent at a predefined baud rate. Both the transmitter and the receiver have to be program to use the same baud rate. In the beginning 20kbps was set as the maximum baud. Today with
the current devices a maximum speed of 1.5 Mbps can be achieved. (Lammert Bies: 2008)

2.2.2 K-line initiation

In this thesis K-line is used to initialize communication with the D2 protocol from Volvo. To initialize over K-line a baud rate of 5bps is needed. If the baud rate is correct the control module (in the car) answers with a synchronization bit and two keywords. The second keyword is inverted and sent back. If the initialization is successful the control module returns the initialization address in inverted format. To continue the communication on K-line the baud rate is then changed to 10400bps. (ISO 9141-2). These steps are all handled in the already existing hardware protocol.

2.2.3 Example on K-line communication

The big difference between CAN and K-line is that in CAN the communication is organized and you can only send or receive a maximum of eight data bytes at the time. In K-line the size of the messages varies a lot and even though there is a structure, it is much harder to get an overview of the communication. In table 2.2.3.1 I will present some K-line communication and explain the different steps in table 2.2.3.2.

**Data stream**

<table>
<thead>
<tr>
<th>Datastream</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 55 EF 8F 70 FE 82 10 F1 10 89 1C 82 F1 10 50 89 5C 82 10 F1 1A 9B 38 B0 F1 10 5A 9B 30 33 47 39 30 36 30 31 36 42 20 20 35 36 34 35 03 00 00 47 00 01 02 03 04 05 52 34 20 31 2C 39 4C 20 45 44 43 20 20 20 20 20 20 20 20 20 C9</td>
<td>Initialization address</td>
</tr>
<tr>
<td>01 55 EF 8F</td>
<td>Synchronization bit, keyword 1, keyword 2 received</td>
</tr>
<tr>
<td>70 FE</td>
<td>Inverted initialization address</td>
</tr>
<tr>
<td>82 10 F1 10 89 1C</td>
<td>82 is the format byte, 10 is target node, F1 is source node, 10 89 is data and 1C is the checksum</td>
</tr>
<tr>
<td>82 F1 10 50 89 38</td>
<td>82 is format byte, F1 is target node, 10 is source node, 50 89 is data and 38 is checksum</td>
</tr>
<tr>
<td>B0 F1 10 5A 9B 30 33 47 39 30 36 30 31 36 42 20 20 35 36 34 35 03 00 00 47 00 01 02 03 04 05 52 34 20 31 2C 39 4C 20 45 44 43 20 20 20 20 20 20 20 20 20 C9</td>
<td>B0 is format byte, F1 is target node, 10 is source node, 5A to C9 is data and C9 is the checksum</td>
</tr>
</tbody>
</table>

**Table 2.2.3.1:** An example of K-line data

<table>
<thead>
<tr>
<th>Data</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Initialization address</td>
</tr>
<tr>
<td>55 EF 8F</td>
<td>Synchronization bit, keyword 1, keyword 2 received</td>
</tr>
<tr>
<td>70</td>
<td>Inverted keyword 2</td>
</tr>
<tr>
<td>FE</td>
<td>Inverted initialization address</td>
</tr>
<tr>
<td>82 10 F1 10 89 1C</td>
<td>82 is the format byte, 10 is target node, F1 is source node, 10 89 is data and 1C is the checksum</td>
</tr>
<tr>
<td>82 F1 10 50 89 38</td>
<td>82 is format byte, F1 is target node, 10 is source node, 50 89 is data and 38 is checksum</td>
</tr>
</tbody>
</table>

**Table 2.2.3.2:** An explained K-line example
2.3. XML

2.3.1 Introduction
XML – eXtensible Markup Language – is a creation of the W3C (World Wide Web Consortium) which are the same group who are responsible for HTML (HyperText Markup Language) and other such specifications. XML was derivated from another markup language SGM (Standard Generalized Markup Language) in 1998 and has been around since. The first XML recommendation from W3C was XML 1.0, which was set in October 2000. The latest XML recommendation is XML 1.1. (Holzner: 2003)

XML is all about creating your own markup. Technically it’s a meta-language, i.e. it’s a language that lets you create your own language. Unlike HTML, XML is developed for storing data, not displaying it. It provides you with a way of structuring your data and lets you decide on exactly how you want to store it. XML is often used for Internet because XML documents are text and you can send them using the existing Internet technology that was developed for HTML. You can save any sort of information in a XML document and if you do not want to save a lot of information, or something important, it’s a perfect substitute for a database. (Holzner: 2003)

2.3.2 Important XML properties
There are a few XML properties which need to be known for creating and understanding a XML document. They will be explained below. (Holzner: 2003)

Declaration. First of all in a XML document you have to make a declaration. In the declaration you have to set the XML version and the encoding used in the document. These properties are set using attributes, see below. A declaration might look like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
```

Here we see that the XML 1.0 specification is used and the encoding UTF-8 (Unicode Transformation Format) is used for the text. UTF-8 contains the characters A-Z and a-z, the numbers 0-9 and some normal special characters. The Swedish characters are not represented and Swedish text wouldn’t be interpreted in the correct way. (Holzner: 2003)

Elements. The elements are the most important part of a XML document. All data in a XML document has to be inside elements. Elements start with an opening tag and the element can be named whatever the creator wants to. Each element has to be closed with an ending tag. An element can contain other elements, just text data or no data at all i.e. empty elements. Empty elements may have to normal format with and opening tag and an ending tag, or as in the example below. Elements can also have attributes within it.

```xml
<document>
  <heading>Hello From XML</heading>
  <message>This is an XML document!</message>

<empty />
</document>
```

In the example above we see that an element document is created. It contains three other elements. The first two of these elements start with an opening tag, contain text data and end with and ending tag. The third is an empty element. (Holzner: 2003)
Attributes. Attributes belong to elements in XML. An attribute can be used as a substitute to the normal element format, or an addition to it. Instead of having text data within an element you can have an empty element with attributes, as shown in the example below. An element may contain more than one attribute. (Holzner: 2003)

<info author="John Smith" genre="science-fiction" />

Comments. Comments are a normal feature in markup or programming languages. Comments help to make the code more understandable. They may be inputted before or after elements. (Holzner: 2003)

<!-- This is a comment -->
3. PRODUCT-LINE

This chapter describes the, which is fundamental for this thesis, product-line technology. It presents the different steps that are needed to be performed when developing a product-line architecture.

3.1 Introduction

Reuse of software has for a long time been an aim for the software engineers. It started in the 1960s, during the software crisis, when the notion of construction systems by composing components was discussed. During the following decades different techniques were proposed but these only addressed reuse at the level of individual components that could be used as the building blocks of new applications. This led to the development of object-oriented frameworks.

In time researchers have learnt two important lessons. The first is that a successful reuse program must be planned and proactive. It is not effective to practice opportunistic reuse. The second is that bottom-up reuse, i.e. composing some random components to construct systems, doesn’t work in practice. For a reuse programs to be successful it needs to use a top-down approach. This means that components are developed that fit into the higher-level structure. However, if there are existing components that need to be incorporated into the product line, you may in a few cases use the bottom-up approach. It must be remembered that when choosing to use already existing components it will have effects on the design of the product-line architecture. (Bosch 2000: 161)

3.2 Decomposing software product lines

The concepts of the product line can be split into three different dimensions. The three dimensions are: (1) architecture, component and system; (2) business, organization, process and technology; and (3) development, deployment and evolution. Below I will shortly describe these three dimensions. Some of the concepts in the dimensions are more vital than others for this project. Those who are not that vital will be mentioned and some of them will be explained very shortly. (Bosch, 2000: 163-166)

3.2.1 First dimension

The first dimension divides the product-line into assets that are part of the reuse-based development.

- **Architecture.** The main purpose is to design an architecture for the product-line that covers all the products in the product line. It should include the features that are shared between the different products.
- **Component.** The product-line architecture has to identify the components and their changeability. The components are developed first after the functionality and changeability has been specified. In the end the components tend to be quite large parts with up to 100 KLOC and are more similar to object-oriented frameworks than classes in an object-oriented system.
- **System.** The product-line architecture has to be adjusted to fit the system architecture. That may involve adding or removing of components/classes from/to the architecture.
3.2.2 Second dimension
The second dimension is describing the different views on the organization.

- **Business.** All activities performed should provide benefits to the business.
- **Organization.** The most common approach is to have a domain engineering unit which is responsible for the development and evolution of the architecture and components of the system.
- **Process.**
- **Technology.**

3.2.3 Third dimension
The third dimension mentions the lifecycle of each of the assets in the organization.

- **Development.** The product-line architecture, the components and the set of products are developed at some point in time. Some are developed from scratch and other from other already available code which is reused.
- **Deployment.** Both the product-line architecture and the components created are deployed during the product development. The product-line architecture has to be modified to represent useful product architecture. Also the components need to be altered to serve their task in the product.
- **Evolution.** The three main assets evolve frequently because the requirements for the products changes. These changing requirements have an effect on the product-line requirements so they too evolve and so on.

3.3 Initiating a product-line
To create software product-lines have to perform careful planning. They require a conscious and clear effort from the organization involved. There are basically two relevant dimensions needed to taken into consideration. First, there is the choice between the evolutionary and the revolutionary approach. The revolutionary approach is more risky but in return if successful gives more back. Secondly, you can either choose to apply the product-line to existing products or to a totally new system/product family which the will help the organization to expand its market. (Bosch 2000: 166). All the necessary actions are presented in table 3.3.1.

<table>
<thead>
<tr>
<th>Existing product-line</th>
<th>Evolutionary</th>
<th>Revolutionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>With the help from existing family members, develop idea for product-line architecture. Develop, by evolving the existing components, the different components for the architecture one at a time.</td>
<td>Product-line architecture and the components are developed based on a super-set of product-line members’ requirements and expected future requirements.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New product-line</th>
<th>Evolutionary</th>
<th>Revolutionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>When the requirements increase, with adding new products, the product-line architecture and the components grow.</td>
<td>Product-line architecture and components are developed to go with requirements of predicted product-line entities.</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.3.1:** Two dimensions of product-line initiation
3.3.1 Evolving an existing set of products into a product-line

For this approach the developing organization keeps a set of existing software products, but without exploiting the similarities they have. Once an analysis of the products is done and the advantages of a product-line approach have been identified, the work continues by creating a product-line, instead of having a set of products, in an evolutionary way. First it requires the development of a product-line architecture which is based on the existing products’ architectures. Then, during the evolution of the products that will become a part of the product-line, components that will implement requirements of more several components are identified. First the most important component, i.e. the one handling the most requirements, is created. As the project continues more and more components are converted to be part of the product-line. These product-line components cover in the end the requirements of more than one product.

A great benefit of the evolutionary approach is that the products in the family continue to evolve, although at a lower pace, whereas the revolutionary approach basically stops production and runs first when the new product-line architecture is planned. The drawback is that is more costly to get a new architecture and new components than with the revolutionary approach. This has to do with the work on the components which is only done to support the current requirements. (Bosch 2000: 166-167)

3.3.2 Replace an existing set of products with a software product-line

If recognizing that the revolutionary approach, for some reasons, is too time-consuming, the optimal approach is to replace the existing set of products with product-line architecture and the required components. When choosing this approach the development on the existing products is basically stopped and all the focus is on designing the architecture and developing the components for the product-line. Previously used legacy software may be made to fit into the architecture and included in the components. To the contrary of the evolutionary approach, the objective is to create a new platform that does not have the problems to products themselves had.

The main advantage with this approach is the generally smaller total investment required to develop the architecture and components for the family. One of the greatest disadvantages is the increased risk level. A big investment has been made and it may prove to be useless if the important requirements change. The second is the delayed time-to-market of the first products developed.

Another important factor in the decision to either evolve or replace the products is the availability of mechanical and hardware parts. If there are great differences between the hardware and mechanical parts it is often hard to take on the evolutionary approach. Then you have to use the revolutionary approach. (Bosch 2000: 167-168)

3.3.3 Evolving a new software product-line

Here the evolutionary approach fits the requirements of the new product into the product-line architecture and the component set. What is done here is focusing on keeping all the products functionality in the product-line and having as little product-specific code as possible. (Bosch 2000: 168)

The advantage is quite similar to evolving an existing set of products: the initial investment is smaller and the time-to-market of the first products of the new domain is quicker. The disadvantage is that new products may have requirements that impact the architecture. To fit these requirements might me hard and costly.
3.3.4 Develop a new software product-line
The last technique mentioned in the matrix is the development of a complete product-line architecture and set of components. First when that is done the developing of the products for the new domain is performed. This technique requires a lot from the software architects and engineers. They need to identify all requirements the upcoming products in the family may demand.

The advantages are that once the product-line architecture and the components are in place, new products can very quickly be developed and the total investment will be small in relation to the evolutionary approach. The disadvantage has more or less to do with risk: the organization is entering a new domain and therefore the domain requirements may be hard to recognize. The architecture and components may therefore not support the system as planned. (Bosch 2000: 168-169)

3.4 Designing a product-line architecture
There are a couple of steps need to be done when designing a product-line architecture. As previously mentioned there is a hope that this will lead to decreased development and maintenance cost and time-to-market and increased software quality. (Bosch 2000: 189-191)

3.4.1 Business case analysis
Before deciding to change courses and use the product-line approach the organization has to somehow verify that the new approach will be successful. There are three primary reasons for adopting the product-line approach. (Bosch 2000: 191-192)

Cost. The cost existent to develop and evolve a is high, therefore there is less instant profitability. An attempt to solve this might involve identifying common parts of the products and capture them in shared assets. This means that these parts will only be developed once and the quality of them usually becomes higher which in the end means that there is less about them needed to be changed in the long run.

Time-to-market. For a lot of organizations developing software products, the cost that comes with the development, does not really have a big impact. What they want to know is the time-to-market, meaning the amount of time needed to convert an idea into a reliable and sellable product. Products which are developed within an organization often are related and the product-line approach is perfect for the developers since it is based on the idea of sharing features. They will usually only need to spend time on unique features. This is a great factor in reducing the time-to-market.

Staff. Today it may be a problem is to hire more personnel, despite having funds and projects available. If having competent personnel, the product-line approach offers great possibilities by training them to learn how to use the product-line approach.

Other reasons. There are also other reasons to be taken into consideration when deciding to create a product-line architecture. First it is important to analyze the current situation and finding the strengths and weaknesses with this. Second, predict and try to calculate future costs and benefits assuming that the current approach is being used. Third, try to calculate what the cost will be trying to converting to the product-line approach. Fourth and last, analyze and predict what the costs and benefits with the new approach. This made to see that it is profitable to convert to a product-line approach. (Bosch 2000: 192-193)
3.4.2 Scoping

The activity of scoping has to do with the selection of product features that should be a part of the product-line architecture. It is an important activity because the chosen are inexpensive to develop since they are included in several products and therefore only developed once. There is minus to it. The features that are part of the product-line architecture may be unfavorable for products not including them. It may be difficult to implement the feature as optional. Even if it is succeeded it may be still leave its traces in the remaining parts of the product and may affect the quality of the architecture.

There are a couple of things in the scoping activity. It can be decomposed into steps as candidate product selection, candidate feature selection, product-line scooping and the specification of product-specific requirements. (Bosch 2000: 194-200)

Defining a feature. A feature can be defined as a logical unit of behavior. It is specified by functional and quality requirements. It basically describes a set of requirements and is used to group requirements. As said a feature also represents a logical unit of behavior, which is from the perspective of one or several stakeholders.

Candidate product selection. First in scoping you have to find out what products may be included in the product line. How to perform this step depends on whether the product line is created for an existing set of products or for a new set. If there is an existing set of products the selecting of the candidates is easy. In the beginning we include all products as candidates.

If using a product line approach in a new domain, selecting candidate products is naturally more difficult since the organization has no real-world knowledge about how to perform this for the specific domain. Here an idea is for the organization to create a strategic plan which uses existing products developed by other companies. What is important is to aim for the next generation of products rather than defining the candidates on the current versions of the products.

Candidate feature selection. To select features you first have to identify them and define them. That is unless the candidate products, mention above, doesn’t have features connected to them.

It is hard to say how granular a feature should be but it should at least capture a relatively big set of requirements. Features in bigger products contain more functionality than in smaller products.

Once there has been an identification of the feature set for each candidate product is a possibility to create a matrix, as seen in figure 3.4.2.1. Here the products and the features make the two dimensions. Then a location in the matrix is marked if the feature is provided by the product. In the end we end up with a matrix where you can easily see which features are more used than others. If there is a feature only used by one product then that feature can be implemented in the product-specific features.

<table>
<thead>
<tr>
<th>Product/feature</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>P4</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.4.2.1: Example of a product and feature matrix
**Product-line scooping.** At this point the architect has all the information needed to select the products and features for the product line. This step has to do with selecting these products and features. They who support the overall goals in the business case best.

There are two extremes in selecting the features for the product line. One is the minimalist approach and the other is the maximalist approach. The minimalist approach means that you only pick the features included in all the products. All the others features are handled by the products them self. The disadvantage with this is that the product line benefits are not used to the fully due to the conservative approach. The maximalist approach obviously means selecting all the features. As often the best choice is somewhere in between. Features used by most products are included and those only used by a few are excluded and handled differentially.

**Specification of product-specific requirements.** When you have defined the scope of the product-line it is possible to define which functionality is to be product-specific.

### 3.4.3 Product-line architectural design

There are a number of different approaches to take when using a product-line architecture. If we look at the extremes, there is one where the software architecture defined for the product line can be used ‘as-is’ for the products in the family. Then the difference between the products in the family is presented in the variability of the components. For the other extreme, the product-line architecture is only used as a core of the product architecture. The architecture of the product may extend or change some parts of the product-line architecture. There are a lot of possibilities in between these two. One is where the product-line architecture is used for all the products, but the products then may have their own implementation anyway. In general, it has been recognized that the product architectures almost always use the same archetypes and make small extensions to the product-line architecture.

What also needs to be taken into consideration is that product-specific features need to be well thought-out when designing the product-line architecture. The product-line architecture will affect both the design and implementation of product-specific features. They have to be considered, otherwise there may be consequences with quality attributes, and design decisions made for the product-line architecture will make it hard for products to implement their own features. (Bosch 2000: 201-202)

**Functionality-based architectural design.** The first step is to create a requirement specification for the product-line architecture. It is a set of functional requirements, containing requirements from all the features. Then the same thing is done for the quality requirements. Finally a reorganization of features into a set of functional and a set of quality requirements is performed for each one of the products. It includes requirements of potential future features.

There is also a need to define in which contexts the products are members. They may not be specified for the product line as a whole. Some context aspects have to be covered in the product-line architecture and others can be handled individually by each product since they have minimal effect on the architecture. Mostly several products context aspects are not included in all products, but rather in a subset of them. Handling each aspect for each product independently is not ideal due to the redundancy problem and it would decrease reuse between the products which is what is wanted to be achieved. Adding it to the product-line architecture may though cause overheads for the products not using this aspect of the context. What is wrong and what is right differs
from case to case, but generally on how hard or easy it is to exclude the aspect and what overheads it causes.

Another step needed to be performed is identifying archetypes. Archetypes describe the main ideas used for modeling the software architecture and for describing how to initiate products. It is not easy to identify archetypes optimal for the product-line since they should not only be based on the product-line requirements but also on the primary product-specific requirements. The product-specific archetypes should extend the product-line archetypes. Overlap between these two should be removed where possible.

Finally you have to describe product instantiations. What needs to be done is to verify that the chosen archetypes are suitable and that the current architecture represents all variations of the product well. (Bosch 2000: 202-205)
4. PROCESSING

In this chapter the whole development process is presented and explained. Here the technologies presented in the previous chapters have been applied to create the end product.

4.1 Preparation

In the first meeting with my two supervisors, Philip Albertson and Patrik Eklund, I was given some information on what was wanted. It was basically just that a diagnostic tool was needed, and I was given an explanation on exactly what that meant. First though they suggested for me to make some research about the technologies to be used. Then we would to be having a follow-up meeting where we would discuss the project in details.

A fundamental technology to understand for this project was the CAN protocol. Before starting on this thesis I had never seen or even heard of CAN communication. Therefore the first step read up on it. Then I continued by looking at some CAN log files existing in the company and tried to get an understanding of the structure of the messages. For better understanding I plugged in to a VOLVO car, using an already existing diagnostic tool, asked the car (control modules) some questions and logged the answers. The logging was done through the tool Kvaser Navigator which is a User Interface provided by the same company who’s made the Kvaser Tool. Kvaser Navigator also has an inbound script which is quite similar to C and it allowed me to write some scripts to extract the data. I was instructed to invoke the different kind of messages that are present in the CAN protocol and to try to understand the structure of them.

4.2 XML construction

After learned to known the CAN protocol I sat down with my instructors and they provided me with a XML template. It was an incomplete template which they used to loosely explain what application structure they wanted and which we used for discussion and further specification of the project. In the beginning the XML document consisted of six main elements: header, messageSyntax, ack, ECUs, SIDs and manufacturerSIDs. After the discussion the data of the elements was changed and also a seventh element was added, parameters. My job was to extend the document to a certain point where it later could be parsed and tested by the diagnostic tool. The extension was made thanks to the previous logging of the CAN communication.

The choice of using XML files for storing the important data is its simplicity. If the BSR Diagnostic Tool is to be sold to resellers, the XML documents could easily be sold containing the functions the reseller wants. If the reseller would want other functions the XML document could be extended. If, for example, the data was stored in a database each computer containing the BSR Diagnostic Tool would need to have a connection to the database. The XML documents decreases the requirements for the resellers. The worry that every reseller could easily access the data in the XML documents could easily be solved by an encryption on the data.

4.3 Developing the ProtocolLayer

I started of by taking notes on how I wanted the system I was to build. Flexibility was a keyword because there is a huge chance that the tool will be extended into working with a lot of protocol. This was to be achieved with the help from the product-line approach. Those ideas worked as reference point through out the making of the architecture.
Obviously the revolutionary approach was used because there were no existing products before the development started.

4.3.1 Creating a product-line architecture
Looking at the procedure the first step is to do the activity of scoping. That means to first pick out candidate for products. In this thesis I have chosen the protocol as candidates for products. The idea is simple. Each protocol has a unique way of communicating to the car. The messages sent and received have a different structure depending on the protocol chosen. Therefore it is clear that the products will be the protocols.

The next step in the line is choosing candidates for features. Because the idea of the diagnostic tool is for me to use the already existing HardwareLayer, which handles direct communication to the car, it’s obvious that there also has to be a communication from my layer, the ProtocolLayer, to the HardwareLayer. I decided to put this on as a feature because all the protocols need to be using this communication. Instead of each protocol having its own communication, which means that redundancy occurs, it is better to use this as a feature. Other than that I also noticed during the beginning of the development that some settings were needed to be stored when communicating to the HardwareLayer. This did not fit in to the feature handling the communication so this I decided to put on as a feature of its own. These settings are set by the products themselves. The relation between the two features is that the communication feature needs the settings feature to keep track of some properties, as which baud rate it should send at, if it should resend messages and so on. The relation is only one-sided.

Besides from this there is a controlling class in the architecture which makes some initiating work and then delegates the rest of the work to the correct parts of the system. This makes the system flexible and easy to extend because the idea is that there should be just minor changes in this class when extending the tool. Other than that the extension is simple because the when adding new products what needs to be developed is just the new product and some minor changes to the existing architecture. Possibly if there is a new way of communication, the communication feature may need extension, but the knowledge available today show that the only techniques used to communicate are already supported by this feature.

4.3.2 Classes of the ProtocolLayer
I used the UML diagrams to draw up the potential system and the different dependencies of the components and classes. This was done to get an overview of the whole system and to have a plan and architecture before starting to implement. This also meant trying to identify how to integrate the product-specific classes to the scope. First of all a component diagram was created to clarify the different components and their connections.

![Component diagram of the system](image)

**Figure 4.3.2.1:** Component diagram of the system

My plan was to make a main class for ProtocolLayer which I named to ProtocolHandler. The idea was to have a class which handles only some of the steps that always have to be performed and delegates the work to the correct classes depending on the user’s choice. Some of the other work would be done by the protocol
specific classes and in that way the application will be easy to extend. Finally there also had to be some way of communicating to the HardwareLayer and this was given a class of its own.

Because the foundation of the diagnostic tool is the XML document – all the instructions needed to have a successful communication is in there - there was a need to save the information from the document in some way. Therefore I planned and built the necessary classes to save the information. The XML document became an excellent foundation for a lot of the classes. It gave a clear structure of how the different parts are connected and therefore an easy mapping from the main elements to classes could be made. Some sub elements in the XML document were also used to enlighten the saving of the data.

A very important task of the ProtocolHandler class is to parse the chosen XML document. By parsing the XML document it becomes clear which functions the application supports. This part was what I first implemented and tried to get functioning. I saved the information in the classes mentioned above and to try it out I made my project into a ConsoleApplication. It gave me the possibility to write out the different XML elements in the console and to see that the relations between the classes were correct and therefore the parsing as well.

Besides from parsing XML documents and delegating work the ProtocolHandler class has the task of notifying the upper – presentation – layer when it should extract data from a buffer and present it to the user. All though in the beginning the presentation layer was not developed I used the same idea used in by the HardwareLayer. That is to raise an event. Therefore the PresentationLayer has to have a listener which waits for an event to be raised in the ProtocolHandler class. To try this out – because the ProtocolLayer was developed before the PresentationLayer - I made a test class which was instead was listening on the ProtocolHandler class and presenting the answers. This approach was used throughout the development of the ProtocolLayer to separate the different components and see that they are standalone components.

For the designing of the product-specific - protocol - classes I decided to name them after the protocol they were to represent. In them I put just methods which were only unique for each protocol. That mainly means methods to assemble and verify messages, which differ from protocol to protocol. Besides from that they were designed to contain a tester present message and any other possible unique message which did not look like the standard messages. Other than that its function is to raise an event when verified and extracted the data out of a message. The ProtocolHandler is then to recognize that an event has been raised.

Another class that was designed is the Settings class, which is also one of the features in the product-line architecture. I noticed early that there were a lot of properties that were needed in all protocol classes and that would later be used by a communication class. Instead of having a lot of classes containing the same properties I decided to keep the protocol classes small and to create a new class containing all the possible settings there were to be made. So when extending the diagnostic tool with a new protocol most certainly the different settings that should be made is present in the Settings class. If you extend the tool this class – feature - may be extended as well.

Finally in the ProtocolLayer there had to be a class which handles the communication to the hardware layer. Again, a possibility – as mentioned above - could be that every protocol could handle its own communication but then there would be redundancy. So I designed the class SenderAndReceiver which gets a message to send from a protocol, sends it to the hardware layer, receives the answer, and raises an event for the active protocol to recognize. The class can also resend if necessary and also handle tester present messages. It is also connected to the Settings class which has been
set by the active protocol. It checks the Settings class to know how handle the communication. This is the second feature in the product-line architecture.

To test the work throughout the development of the ProtocolLayer a test program was created which presented a menu with all the implemented protocols, ECUs, SIDs and manuSIDs. This was made to be able to test this layer before starting to work on the PresentationLayer. The format used for the presentation was the same as found in a logging application used by myself, Kvaser Navigator.

![Figure 4.3.2.2: A picture showing the temporary test program](image-url)
4.4 Developing the PresentationLayer

The PresentationLayer I was told by my supervisors should be simple. It should be and is only a user interface. For the user it had to be clear what to do. It should basically focus on presenting the answer it has received from the ProtocolLayer. The functional requirement that I was given was that there had to be a possibility to present its answers in real time. That meant that I would have to have some kind of functionality which kept a function running for an undefined number of times and showing the change of the result of the chosen function.

Before starting on developing the PresentationLayer I sat down and drew some prototypes of the user interface and checked with my supervisors what they thought and which things should be where and so on. In the end I had something which could be used.
5. RESULTS

This chapter will be used to present how the issues in the previous chapter were solved in terms of how it ended up being.

5.1 XML

5.1.1 Header

The header of the document contains information about the protocol which the XML file specifies. It consists of basic information about the protocol, and some are vital to uphold a communication between the diagnostic tool and the control modules. For example the canInit parameter describes what baud rate has to be set in the initialization phase.

```
<header>
  <protocolName>D2</protocolName>
  <manufacturerName>VOLVO</manufacturerName>
  <flag>Extended</flag>
  <canInit>500000</canInit>
  <kline>false</kline>
</header>
```

**Figure 5.1.1.1:** An example of a header

5.1.2 Message Syntax

The Message Syntax contains the different message types which can be sent or received through the CAN protocol. There are four or five, depending on the protocol, message types and each message type has a unique structure. They mainly differ in the first two bytes.

The first message type is the broadcast type which is used for broadcast messages. All broadcast messages look the same which means that the first byte is always 0xfe and the second byte is a counter 0x01-0x06. Then there is a maximum amount of six bytes of data. In any protocol a broadcast message is always sent on a predefined id.

The second message type is the single message type which is used for messages up to eight bytes big. Single messages are the most common messages sent on a CAN communication. A single message only has one pre-specified byte, the first one, and it differs within a range. In what range depends on how it is specified by the protocol.

The third message type is the multipleFirst message type. In CAN, as previously described, a message can hold a maximum amount of eight bytes. When wanting to transmit more than eight bytes of data a sequence of messages is transmitted. The first message in this sequence is named in this project a multipleFirst message type. Depending on the protocol either the first or the first two bytes are set within a range. In one VOLVO protocol a multiple first message is represented with a 0x10 byte at the first position.

The fourth and message type is the multipleStream message type. It is a consequence when transmitting more data than can be fitted in one message. The multipleStream differs in the first byte. The first byte is also used as a counter on how many multipleStream messages have been transmitted. The rest of the bytes in a multipleStream message are data. While the three previously mentioned message types can only be sent once in a query/answer, there can be several multipleStream messages depending on the amount of data.

The four mentioned message types are all a part of all protocols. But there is a fifth, the multipleEnd message type, which is only a part of some of the protocols. The
multipleEnd represents the end of a sequence of messages. The first byte in a multipleEnd message is also a counter on how many multiple messages have been transmitted, but it is in a different range than the multipleStream messages.

```
<messageType>
  <name>single</name>
  <isBroadcast>false</isBroadcast>
  <ID>-1</ID>
  <dlc>8</dlc>
  <staticData>
    <byte>
      <position>0</position>
      <value>
        <dataLength>
          <range>
            <min>0xCD</min>
            <max>0xCF</max>
          </range>
        </dataLength>
      </value>
    </byte>
  </staticData>
  <data>
    <startPosition>1</startPosition>
    <length>7</length>
  </data>
</messageType>
```

**Figure 5.1.2.1:** An example of a single message type in the protocol D2

5.1.3 Ack Message

The ack message is a message which has to be sent in a transmission which consists of multiple messages. When receiving a multipleFirst message the receiving part of the communication has to send an acknowledgement to notify the message has been received and the rest can follow. The ack message is only to be sent after a multipleFirst message, and is not needed between multipleStream messages.

```
<ack>
  <message>
    <name>ack</name>
    <dlc>8</dlc>
    <data>
      <quad>
        <position>0</position>
        <value>0x30</value>
      </quad>
    </data>
  </message>
</ack>
```

**Figure 5.1.3.1:** The standard ack message

5.1.4 ECUs

The ECUs (Electronic Control Units) contains all different ECUs within a car. An ECU represents a control module in a car. A certain ECU has unique properties which differ from the other ECUs. All messages sent to a specific control module are sent with a predefined id at a predefined baud rate. There is also a list of id’s which the control module within the car uses to send answer messages with. Besides from that an ECU has a list of SIDs which contains other information of how to assemble or verify a messages which should be sent or received through the CAN protocol.
Figure 5.1.4.1: An ECU in the protocol D2

5.1.5 SIDs

Each ECU has a lot of SIDs (Service Identifiers) which are representing the services of an ECU which can provide different information about the car. These SIDs are a part of every protocol. There are protocol bound SIDs which will be explained later.

When sending a message each SID is reached with a specific value on the first byte. When a positive answer is received from the same SID the first byte is increased with 0x40. In the same way, when a control module gives a negative response either the first or the second byte is 0x7f. Besides from this each SID has some parameters bound to it, which basically represent the different information which can be reached on this SID.
5.1.6 ManuSIDs

Just like each protocol contains different SIDs it also has some ManuSIDs (Manufacturer Specific Service Identifier). These are bound the protocol and are not present in any other protocol. There is a difference in the structure of ManuSIDs to the structure in SIDs. In ManuSIDs there is still requestCode and negResponseCode but there is no posResponseCode. Instead it's replaced with blockData and posEndCode which has to be present when receiving a positive answer from the control unit. There are also no parameters bound to each ManuSID, instead all the data required to send a message to the control module is in the requestCode.

Figure 5.1.5.1: An example of a SID in the ISO_15765 protocol
5.2.7 Parameters

Just like mentioned above each SID has a reference to some parameters which are used to build the remaining part of a message to be sent or to verify and translate – not part of this thesis – a received message. A parameter refers to the actual questions you can ask the car module. It also has some information of how long the receive message should be, at what position the data start, if the data is unsigned or signed and how to translate the data from bytes to something presentable for the user.

Figure 5.1.6.1: An example of a manuSID in the GM_LAN protocol

<manufacturerSpecificSID>
  <name>GM_LAN_readDTCs</name>
  <mnemonic>GM</mnemonic>
  <timeout>1000</timeout>
  <type>blockTransfer</type>
  <requestCode>
    <type>send</type>
    <data>
      <byte>0xa9</byte>
      <byte>0x81</byte>
      <byte>0x12</byte>
    </data>
  </requestCode>
  <blockData>
    <type>receive</type>
    <messageTypeOverride>True</messageTypeOverride>
    <dlc>8</dlc>
    <data>
      <byte>0x81</byte>
    </data>
  </blockData>
  <posEndCode>
    <type>receive</type>
    <messageTypeOverride>True</messageTypeOverride>
    <dlc>8</dlc>
    <data>
      <byte>0x81</byte>
      <byte>0x00</byte>
      <byte>0x00</byte>
      <byte>0x00</byte>
    </data>
  </posEndCode>
  <negResponseCode>
    <type>receive</type>
    <data>
      <byte>0x7f</byte>
      <byte>0xa9</byte>
    </data>
  </negResponseCode>
</manufacturerSpecificSID>

Figure 5.1.7.1: An example of a parameter in the ISO_15765 protocol

<parameter>
  <name>Vehicle Identification Number</name>
  <data>
    <byte>0xf1</byte>
    <byte>0x90</byte>
  </data>
  <rx-1>24</rx-1>
  <length>14</length>
  <pos>0</pos>
  <phys>X1</phys>
  <sign>unsigned</sign>
</parameter>
5.2 ProtocolLayer

The main class for the ProtocolLayer is the ProtocolHandler. It became the tie that binds it all together. One of the functionalities of this class is to set up and tear down the communication to the HardwareLayer. The decision to put these functionalities there was made because other classes need to get objects of the HardwareLayer delivered to their constructor, and ProtocolHandler is the main class which instantiates them. Other than that the most important method is the AssembleMessageHandler method which gets some parameters and depending on them makes the correct settings in the Settings and SenderAndReceiver classes. Then it delegates the work to the correct protocol which performs the next steps; assembling and verifying messages.

The most complex classes are the protocols and SenderAndReceiver. There a lot of the coding is done and basically all the functionality is stored in these classes. For the SenderAndReceiver class I used two constructors, one for communication with k-line and one without. The only real difference between the constructors is that when a protocol is running on k-line a k-line object is connected to the HardwareLayer and is used for sending and receiving some special messages.

5.2.1 Example on product instantiations

In this thesis I have created and successfully connected two products to the architecture. There was a difference between these products because one product – protocol - needed a K-line initiation and the other did not. Both could smoothly be incorporated to the architecture.

First the one without the K-line was incorporated and what was needed to be done was to create a reference to it in the ProtocolHandler class and a listener on it. Other than that a feature, Settings, needed some extending. That meant adding one attribute in it where it could be seen if a specific function, readDTC, had been run on the protocol. That because there are special answers needed to be interpreted by the protocols in a special way.

The other product, using the K-line, was instantiated in the same fashion. There were no problems with the communication because K-line communication had been already handled. Again the same steps in the ProtocolHandler and the feature Settings were needed.

5.3 PresentationLayer

Due to the requirement of simplicity the PresentationLayer ended up only having two windows, a main window and a properties window for setting the used COM ports of the K-line tester when connecting to a protocol using k-line.

![Figure 5.3.1: Window for setting the COM ports](image)

For the design of the main window a simple tree view was used to present the different functions able to run on a specific protocol. The tree view is received from the structure
built by the XML parsing made in the ProtocolHandler. To use the diagnostic tool against a car the user has to press the button Connect. For the connecting to work all the cables has to be inserted into both the car and computer. A user can then easily scroll through the tree view and find the different ECU’s, SID’s and manuSID’s and pick a function to run. The user can either run the function immediately or add it into a sequence list and when desired run all the functions stored in there. There is also a possibility, just as requested, to run function an undefined number of times. In the case of the sequence list consisting of more than one element, each function will run once before allowing the next one in the queue to run. The answer – received from the ProtocolLayer – is presented in two different boxes. One of the boxes is a history list box and saves all the answers received during a session. The other one is a list box which updates, i.e. if a function has been run before the previous entity in the list is updated.

![Figure 5.3.2: The final interface of the BSR Diagnostic Tool](image)

Figure 5.3.2: The final interface of the BSR Diagnostic Tool
6. FINAL DISCUSSION

6.1 Discussion on result
The two components – ProtocolLayer and PresentationLayer - that were provided during this Master Thesis have successfully been connected to the previous existing BSR component. The already existing HardwareLayer was a good base, by simplifying the car communication, but could not be used in a constructive way by BSR. Now with the two components developed, BSR can in a structured and user friendly way communicate with the cars and extract information about the cars. This without having to spend a lot of money to purchase manufacturer bound diagnostic tools.

The diagnostic tool is reliant on the information stored in the XML documents. The more the XML document has been extended, the more functions the diagnostic tool can support. Because BSR has a long history of working with cars and other diagnostic tools, they have a lot of data which can be transformed into a XML document. To simplify this I have developed a XML editor which can read or create a XML file and expand it. In the future the idea is to extend this editor for it to be able to fill the XML documents automatically, i.e. get a reference to some data and transform it into the right format.

6.2 Future possible extensions
Of course there are possibilities to perform extensions on the diagnostic tool. That was one of the purposes with the product-line architecture which was also mention in chapter 1.1. The extensions might involve new features, modified architecture, new products etc. Here I will look at some possible extensions that might be made on the tool and how.

6.2.1 Features
To add new a feature the architect first need to verify that there is a need for this feature, that the functionality isn’t already handled, that the feature will be used by the products and that the adding of the feature doesn’t change the architecture in the sense that the minimalist or maximalist, or what approach has been chosen, will be changed. When all these tasks are performed the feature can be developed and integrated to the system.

For the diagnostic tool there might be a need to add feature for both the ProtocolLayer and PresentationLayer. BSR has through the years been using other tools to log communication to the cars. Pure data, in hexadecimal format, is in some cases more interesting than the answer itself, especially when things do not go as planned, i.e. when the cars responds in a negative way. So the feature would involve logging and be connected to the SenderAndReceiver feature. All sent and received messages would be sent to this feature in their message format where id, counters and data is present, and then saved into some kind of text format.

![Figure 6.2.1.1: Feature tree after adding new logging feature](image_url)
In the same way BSR is in some cases interested in analyzing change on the data. Because this tool can be used in as a real-time application you can send the same question over and over again and then be interested in the change over a period of time. This would then be a part of the PresentationLayer and would also be the only feature in that layer. Preferably this feature could handle and save the answers in such a way that the timestamp and answer is saved into an Excel document and then easily can be presented with diagrams.

6.2.2 Products
When adding a totally new product to the architecture you first have to verify that the features support the functionality needed by the new product. Then you have to integrate the new product with the existing architecture and components. There might be a need to share some of the functionality of the product to the architecture and features if there is a possibility for it to be an asset for the architecture.

![Architecture Diagram]

**Figure 6.2.2.1:** Adding a new product to the architecture

A realization of product extension means that a new XML document has to be created. Then a new protocol class is needed to be developed, with some protocol specific attributes set, methods to assemble and verify messages and functions to notify the ProtocolHandler that something is ready. Besides from that in most cases only a reference to the new protocol is needed in the ProtocolHandler, some settings to be altered and a listener on the new protocol. That makes the diagnostic tool easy to be extended and maintained. This was also tried out by me when making a new protocol for the car manufacturer Saab. BSR has a great knowledge on Saab so after understanding the structure a new XML document was made. Then just as described only small changes where needed and the new protocol worked perfectly with the rest of the classes.

6.2 Future work
What is now left for BSR to develop is the interpretation of the verified data. This was as mentioned not a part of this Master Thesis, due to its complexity and also due to it was not defined within the company how this was to be solved. In the beginning the idea was to add this to the PresentationLayer by in some way checking the run function to a file consisting of interpretation codes. This was later abandoned and in the end of the thesis the interpretation became a part of the ProtocolLayer. In the XML document an element phys was added, containing information on how to interpret the data. Then the protocol classes will be needed to be extended to contain methods solving the interpretation. When this is solved the diagnostic tool will be ready for full usage within
the company – it partially is already because BSR personal in some cases are interested more in hexadecimal data than concrete answers.

The future of the diagnostic tool is in the hands of BSR. It will most certainly be used by the employees. If it will be sold to the resellers is still unclear. A lot of the resellers are in a need of tools to diagnose the cars of their customers. They are basically in the same situation as BSR; they have some manufacturer bound tools. So there is most certainly a market. If decided that the tool is to be sold the issues of selling and encrypting the XML documents will be important. Also there will also be a need of further development of the tool to be extended it to be working with other manufacturers and protocols. For now there is a great knowledge about VOLVO, Saab and VAG within BSR. To develop the tool to be working with other manufacturers and protocols, a lot of research first has to be done. I believe if BSR use this base and continue to develop it; it will do a lot of good for the company.
7. REFERENCES


ISO 9141 Road Vehicles – Diagnostic Systems
ISO 9141-2 Part 2 – K-line

CAN Standardization, 2008-04-20
http://www.can-cia.org/can/standardization/index.html

Lammert Bies, April 2008


8. APPENDICES

8.1 A XML document
Here I will present a whole XML document. It is for VOLVO and the protocol called ISO_15765.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<protocol>
  <header> <!-- Specifying protocol information -->
    <protocolName>ISO_15765</protocolName>
    <manufacturerName>VOLVO</manufacturerName>
    <flag>Standard</flag>
    <canInit>500000</canInit>
    <kline>false</kline>
  </header>
  <messageSyntax>
    <messageType>
      <name>broadcast</name>
      <isBroadcast>true</isBroadcast>
      <ID>0x101</ID>
      <dlc>8</dlc>
      <staticData>
        <byte>
          <position>0</position>
          <value>
            <dataLength>
              <range>
                <min>0xfe</min>
                <max>0xfe</max>
              </range>
            </dataLength>
          </value>
        </byte>
        <byte>
          <position>1</position>
          <value>
            <dataLength>
              <range>
                <min>0x01</min>
                <max>0x06</max>
              </range>
            </dataLength>
          </value>
        </byte>
      </staticData>
    </messageType>
  </messageSyntax>
</protocol>
```
<messageType>
  <name>single</name>
  <isBroadcast>false</isBroadcast>
  <ID>-1</ID>
  <dlc>8</dlc>
  <staticData>
    <byte>
      <position>0</position>
      <value>
        <dataLength>
          <range>
            <min>0x01</min>
            <max>0x07</max>
          </range>
        </dataLength>
      </value>
    </byte>
  </staticData>
  <data>
    <startPosition>2</startPosition>
    <length>6</length>
  </data>
  <generateAck>false</generateAck>
</messageType>

<messageType>
  <name>multipleFirst</name>
  <isBroadcast>false</isBroadcast>
  <ID>-1</ID>
  <dlc>8</dlc>
  <staticData>
    <byte>
      <position>0</position>
      <value>
        <dataLength>
          <range>
            <min>0x10</min>
            <max>0x18</max>
          </range>
        </dataLength>
      </value>
    </byte>
  </staticData>
  <data>
    <startPosition>1</startPosition>
    <length>7</length>
  </data>
  <generateAck>false</generateAck>
</messageType>
<max>0x10</max>
</range>
</dataLength>

<byte>
<position>1</position>
</byte>
</staticData>
<data>
<StartPosition>2</StartPosition>
<length>6</length>
</data>
<generateAck>true</generateAck>
</messageType>
<messageType>
<name>multipleStream</name>
<isBroadcast>false</isBroadcast>
<ID>-1</ID>
<dlc>8</dlc>
<staticData>
<byte>
<position>0</position>
</byte>
</staticData>
<dataLength>
<range>
<min>0x20</min>
<max>0x2f</max>
</range>
</dataLength>
</byte>
</staticData>
<data>
<startPosition>1</startPosition> <length>7</length>
</data>
<generateAck>false</generateAck>
</messageType>
</messageSyntax>
<ack>
<message>
<name>ack</name>
<dlc>8</dlc>
<data>
<quad>
<position>0</position>
<value>0x30</value>
</quad>
</data>
</message>
</ack>

<ECUs>
<ECU>
<name>CEM</name>
<sendID>0x726</sendID>
<baudrate>500000</baudrate>
<receiveID>
<id>0x72e</id>
</receiveID>
<allowedSIDs>
<sid>CEM_REI</sid>
</allowedSIDs>
<allowedManuFacturerSpecificSIDs>
<sid>GRD</sid>
</allowedManuFacturerSpecificSIDs>
</ECU>
<ECU>
<name>DIM</name>
<sendID>0x720</sendID>
<baudrate>125000</baudrate>
<receiveID>
<id>0x728</id>
</receiveID>
<allowedSIDs>
<sid>DIM_REI</sid>
</allowedSIDs>
</ECU>
<ECU>
<name>TEST DTC</name>
<sendID>0x7df</sendID>
<baudrate>500000</baudrate>
<receiveID>
<id>0x111</id>
</receiveID>

<allowedSIDs>
  <sid>DTC</sid>
  <sid>CLD</sid>
</allowedSIDs>
</ECU>

</ECUs>

<SIDs>
  <SID>
    <name>readECUIdentity</name>
    <mnemonic>CEM_REI</mnemonic>
    <timeout>1000</timeout>
    <requestCode>
      <type>send</type>
      <data>
        <byte>0x22</byte>
        <allowedParameters>
          <parameter>Vehicle Identification Number</parameter>
          <parameter>Mileage</parameter>
        </allowedParameters>
      </data>
    </requestCode>
    <posResponseCode>
      <type>receive</type>
      <data>
        <byte>0x62</byte>
        <allowedParameters>
          <parameter>Vehicle Identification Number</parameter>
          <parameter>Mileage</parameter>
        </allowedParameters>
      </data>
    </posResponseCode>
    <negResponseCode>
      <type>receive</type>
      <data>
        <byte>0x7f</byte>
        <byte>0x1a</byte>
      </data>
    </negResponseCode>
  </SID>
  <SID>
    <name> DIM_REI</name>
    <mnemonic>DIM_REI</mnemonic>
    <timeout>1000</timeout>
  </SID>
</SIDs>
<requestCode>
  <type>send</type>
  <data>
    <byte>0x22</byte>
  </data>
</requestCode>

<allowedParameters>
  <parameter>Fuel Level</parameter>
</allowedParameters>

<posResponseCode>
  <type>receive</type>
  <data>
    <byte>0x62</byte>
  </data>
</posResponseCode>

<negResponseCode>
  <type>receive</type>
  <data>
    <byte>0x7f</byte>
    <byte>0x1a</byte>
  </data>
</negResponseCode>

<SID>
  <name>readDTC</name>
  <mnemonic>DTC</mnemonic>
  <timeout>1000</timeout>
  <requestCode>
    <type>send</type>
    <data>
      <byte>0x19</byte>
    </data>
  </requestCode>
  <allowedParameters>
    <parameter>Confirmed DTC</parameter>
  </allowedParameters>
  <posResponseCode>
    <type>receive</type>
    <data>
    </data>
  </posResponseCode>
</SID>
<byte>0x59</byte>
<allowedParameters>
<parameter>Confirmed DTC</parameter>
</allowedParameters>
</data>
</posResponseCode>
<negResponseCode>
<type>receive</type>
<data>
<byte>0x7f</byte>
<byte>0x1a</byte>
</data>
</negResponseCode>
</SID>
<SID>
<name>clearDTC</name>
<mnemonic>CLD</mnemonic>
<timeout>1000</timeout>
<requestCode>
<type>send</type>
<data>
<byte>0x14</byte>
<allowedParameters>
<parameter>All groups</parameter>
</allowedParameters>
</data>
</requestCode>
</SID>
</SID>
</negResponseCode>
</byte>0x54</byte>
<allowedParameters>
<parameter>All groups</parameter>
</allowedParameters>
</data>
</posResponseCode>
<negResponseCode>
<type>receive</type>
<data>
<byte>0x7f</byte>
<byte>0x1a</byte>
<manuufacturerSpecificSIDs>
  <manufacturerSpecificSID>
    <name>GM_LAN_readDTCs</name>
    <mnemonic>GRD</mnemonic>
    <timeout>1000</timeout>
    <type>blockTransfer</type>

    <requestCode>
      <type>send</type>
      <data>
        <byte>0xa9</byte>
        <byte>0x81</byte>
        <byte>0x12</byte>
      </data>
    </requestCode>

    <blockData>
      <type>receive</type>
      <messageTypeOverride>True</messageTypeOverride>
      <dlc>8</dlc>
      <data>
        <byte>0x81</byte>
      </data>
    </blockData>

    <posEndCode>
      <type>receive</type>
      <messageTypeOverride>True</messageTypeOverride>
      <dlc>8</dlc>
      <data>
        <byte>0x81</byte>
        <byte>0x00</byte>
        <byte>0x00</byte>
        <byte>0x00</byte>
      </data>
    </posEndCode>

    <negResponseCode>
      <type>receive</type>
      <data>
        <byte>0x7f</byte>
      </data>
    </negResponseCode>
  </manufacturerSpecificSID>
</manufacturerSpecificSIDs>
<byte>0xa9</byte>
   </data>
</negResponseCode>
</manufacturerSpecificSID>
</manufacturerSpecificSIDs>

<parameters>
   <parameter>
      <name>Vehicle Identification Number</name>
      <data>
         <byte>0xf1</byte>
         <byte>0x90</byte>
      </data>
      <rx-l>24</rx-l>
      <length>14</length>
      <pos>0</pos>
      <phys>x*1</phys>
      <sign>unsigned</sign>
   </parameter>
   <parameter>
      <name>Fuel Level</name>
      <data>
         <byte>0xee</byte>
         <byte>0x10</byte>
      </data>
      <rx-l>24</rx-l>
      <length>2</length>
      <pos>0</pos>
      <phys>x*1</phys>
      <sign>unsigned</sign>
   </parameter>
   <parameter>
      <name>Mileage</name>
      <data>
         <byte>0xdd</byte>
         <byte>0x01</byte>
      </data>
      <rx-l>24</rx-l>
      <length>2</length>
      <pos>0</pos>
      <phys>x*1</phys>
      <sign>unsigned</sign>
   </parameter>
   <parameter>
      <name>Confirmed DTC</name>
      <data>
         <byte>0x02</byte>
         <byte>0x08</byte>
      </data>
      <rx-l>24</rx-l>
      <length>3</length>
      <pos>0</pos>
      <phys>x*1</phys>
      <sign>unsigned</sign>
   </parameter>
</parameters>
<parameter>
  <name>All groups</name>
  <data>
    <byte>0xff</byte>
    <byte>0xff</byte>
    <byte>0xff</byte>
  </data>
  <rx-1>0</rx-1>
  <length>0</length>
  <pos>0</pos>
  <phys>-1</phys>
  <sign>unsigned</sign>
</parameter>
</parameters>
</protocol>
Matematiska och systemtekniska institutionen
SE-351 95 Växjö

Tel. +46 (0)470 70 80 00, fax +46 (0)470 840 04
http://www.vxu.se/msi/