Master in Software Engineering for Professionals
Month Year

Integration of Service-Oriented Embedded Systems with External Systems in Software Product Lines

A Design Science Research Study

Nils Johansson

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master in Software Engineering for Professionals. The thesis is equivalent to 12 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Nils Johansson
E-mail: nijb16@student.bth.se

E-mail: nils.johansson.nj@gmail.com

University advisor:
Dr. Michael Unterkalmsteiner
Department of Software Engineering

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. Developing software for complicated systems is often done by collaboration and consists of deliverables by a multitude of organisations. The deliverables can range from smaller devices and commercial-off-the-shelf software components, to larger systems. This is the situation during the development of the embedded system for large vehicles or machines. Many companies within the embedded industry are transitioning to using Service-Orientation to develop high-quality software and reduce costs. However, when integrating different external systems with an internal, service-oriented system there may arise difficulties since the communication patterns, i.e. interface, cannot be changed to fit the internal system.
Objectives. This study aims to develop a design solution that can be used to integrate different external systems with an internally developed service-oriented system in an entire software product line, including the handling of variability by parametrization. The solution is evaluated by software developers at a company in such a situation.
Methods. To develop the design solution *design science* methodology is applied, which is an iterative process that continuously improves the candidate solution until satisfactory according to various stakeholders.
Results. The resultant design solution includes the use of wrappers-based interaction between systems, where so-called *adapters* are used when the internal system acts as a client to an external system, and using *gateways* for when the internal systems acts as a server to an external system. We also observe the need for a system integration view to describe the relations and available communication mechanisms between systems, i.e. the gateways and adapters.
Conclusions. We conclude that to integrate a service-oriented software system with non-service-oriented systems, there can be benefits to using an abstraction layer between systems to protect the internally developed software architecture from being affected by the nature of the external system. Attempting to integrate external systems with an internal system as if also developed internally may become troublesome in terms of defining and upholding an appropriate service-oriented architecture. This is especially important when considering variability of the complete system, where different external systems are used or replaced in specific variants.

Keywords: Service-oriented, integration, abstraction, software architecture

Acknowledgments

Here you can add your acknowledgements.

Contents

Abstract i

Acknowledgments iii

1 Introduction 1
- 1.1 Context and Problem Statement . . . 1
- 1.2 Aim and Objectives . . . 3
- 1.3 Scope and Limitations . . . 3

2 Background 5
- 2.1 SPLE and Variability . . . 5
- 2.2 Service-Oriented Architectures . . . 6
- 2.3 Service Identification . . . 6
- 2.4 Relations Between Services in Embedded Systems . . . 7
- 2.5 Service Identification to Modernize Legacy Systems . . . 7
- 2.6 Configuration of Services . . . 9

3 Related work 11
- 3.1 Research Gap . . . 13

4 Research Methodology 15
- 4.1 Research Questions . . . 15
- 4.2 Research Design . . . 16
- 4.3 Engineering Cycle Framework . . . 17
  - 4.3.1 Problem Investigation . . . 18
  - 4.3.2 Solution Design . . . 18
  - 4.3.3 Design Validation . . . 18
  - 4.3.4 Solution Validation . . . 19
  - 4.3.5 Implementation Evaluation . . . 19
- 4.4 Data Collection . . . 19
  - 4.4.1 Focus Groups . . . 19
  - 4.4.2 Interviews . . . 20
  - 4.4.3 Literature Review . . . 21
  - 4.4.4 Data Analysis . . . 21

5 Results 23
- 5.1 Cycle I . . . 23
  - 5.1.1 Problem Investigation . . . 23

5.1.2 Solution Design 29
5.1.3 Solution Validation 31
5.1.4 Solution Implementation 33
5.1.5 Implementation Evaluation 35
5.2 Cycle 2 37
5.2.1 Solution Design 38
5.2.2 Solution Validation 41
5.2.3 Solution Implementation 42
5.2.4 Implementation Evaluation 45

6 Discussion 47
6.1 Research Questions 47
6.2 Validity Threats 49
6.2.1 Construct Validity 49
6.2.2 Internal Validity 49
6.2.3 External Validity 50
6.2.4 Reliability 50
6.3 Conclusions and Recommendations 51

References 53

A 2nd Cycle Implementation Evaluation Questionnaire 57

Chapter 1

Introduction

1.1 Context and Problem Statement

Developing a large embedded system, such as an automotive system, is and has been a task requiring collaboration between many suppliers of hardware and software components for many years. Without dividing responsibilities between numerous organisations, the scope would become unmanageable and likely result in poor quality. At a high level, the division of responsibilities between departments of an automotive manufacturer are often divided based on functional domains [18, 33]. Typically these have been the power train-, chassis-, body-, HMI- and telematics domains. Each functions as a sub-system with its own API used by other subsystems when needed. It is also common that a sub-system is reused over multiple products, meaning that the same interface is also reused. Structuring the division of responsibilities in this manner, separating the concerns by functionality domain, enables departments to focus on one particular subset of full system functionality. In addition, manufacturers of original equipment utilize many third-party solutions in their products. These may range from simple sensors and actuators, to complete highly intelligent systems. Third-party solutions are often called commercial-off-the-shelf (COTS) components. There is also a possibility that during the development of new software, old legacy systems may be reused and incorporated into the new product to decrease time-to-market and development costs.

While working with software development of any organisation involved in the development of such a large system, one can determine one important common feature of these systems. The interfaces and how the systems communicate to perform some functionality, usually cannot be changed. Suppliers of software of functional domains may reuse the software for multiple products and/or have finished the development of the software's responsibilities. The interface to a COTS component is not changeable, as it comes with the purchase of said component. Legacy systems are already defined in their entirety. However, a development organisation with responsibilities to develop a part of the complete embedded system must have a strategy to integrate its system with these third-party systems. Hereafter, these will be called *external systems*, as described in Figure 1.1.

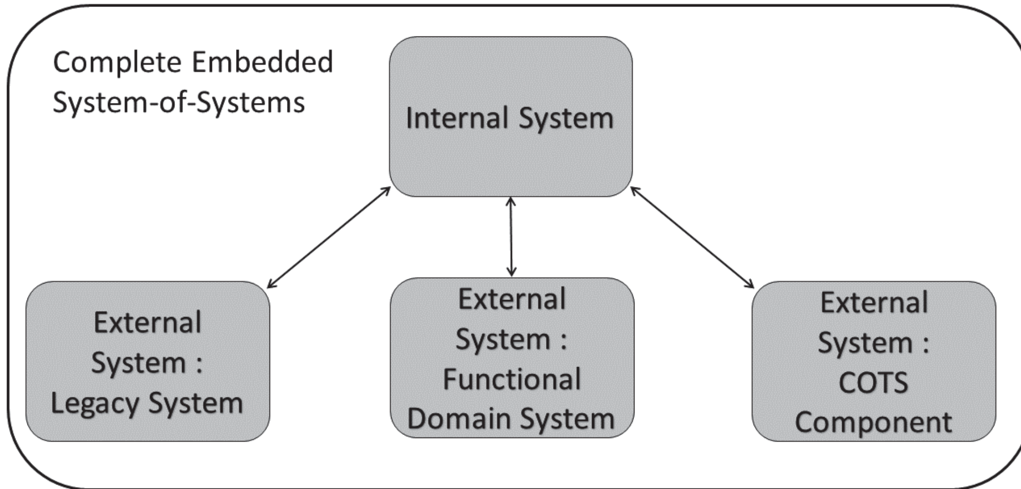

Figure 1.1: Embedded System-of-Systems

It is common that these systems communicate on a Controller Area Network (CAN). In addition to reuse, software systems may be subject to handle variability to varying degrees [4]. These variations can range from changes that appear small, such as a button being installed or not, to larger changes, such as having a combustion or electrical engine installed. Many other forms of variability exist, for example how the software shall control the behaviour of the system to achieve a specific goal. This variability must be handled suitably by some strategy, to avoid unnecessary work during software development, and to avoid improper software being deployed to a customer. Developers of products with variability often apply Software Product Line Engineering (SPLE) in order to manage variability. SPLE may be used both during requirements definition and software development in order to facilitate e.g., reuse and scalability of requirements and software.

Recently, Service-Oriented Architectures (SOA) have become increasingly popular within the software domain. It has been shown to be utilized successfully to develop embedded systems, including automotive systems [16,33]. One can obtain high quality, such as flexibility, interoperability, scalability, reuasability while also reducing complexity [22]. Selecting a particular architecture style may however limit the interfacing capabilities between systems. This can prove difficult when integrating different external systems, such as a reused legacy system or a COTS component. The interface to such systems or components is already defined, and usually cannot be altered to fit the principles of e.g. SOA.

In an entire product line, there can be different legacy systems or COTS that are utilized to complete different products in the line. Some purchased components may be included in one product but not the next, where perhaps a different component, from

a different supplier is used instead, with a completely new interface. This adds a new dimension to variability that also must be managed, since using different components may also mean that the allocated responsibilities between systems will change. This leads to that the structure and relations of the software elements can be affected, i.e. that the software architecture differs between products in a product line.

1.2 Aim and Objectives

Embedded software systems must be able to integrate different external systems into a distributed system, or system-of-systems. How this can be achieved over an entire product line is not sufficiently established, as is described in Section 3. The aim of this research is to investigate how variable third-party software components can be integrated into a software product line of embedded systems, using Service-Oriented Architectures.

The research objectives of this study are:

1. Develop a general design solution that can be used to integrate any third-party software component in a service-oriented embedded system that uses primarily CAN-based communications technology.

2. Determine how different third-party software components can be integrated into a service-oriented software product line.

3. Evaluate the solution by gathering empirical data from software engineers and other stakeholders that are likely to use and implement it.

1.3 Scope and Limitations

The scope of this study is limited to research on embedded systems. The primary communication between systems allocated to Electronic Control Units is done using CAN bus technology. The software variability due to differently installed hardware is realized by configuring the system using parametrization at compile-time. The selected architecture style is Service-Orientation, or specifically Service-Object-Oriented Architectures.

Chapter 2

Background

This section will briefly cover topics that are necessary to understand how the objectives and aims may be reached. These include *Software Product Line Engineering*, *Service-Oriented Architectures*, *Service Identification*, how relations between services are often defined in embedded systems, *Service Identification* is used to modernize legacy systems and how to configure services.

2.1 SPLE and Variability

Software Product Lines Engineering has been and are extensively explored within academia and industry [4]. The reason to adopt Software Product Lines (SPL) in an organisation is often to support the reuse of software over a long time-span [20]. By treating the product offerings as a line (or family) instead of multiple single products, long-term reductions of development costs and time to market may be obtained. One can decompose the features of a product line into three categories: first, all features which are common for all products, i.e., the *commonalities*; second, one can find certain features that should not be equal in all products, i.e. the *variabilities*; third, some features may be only found in only one product. These *product-specific* features can be requested by individual customers and not necessarily offered to the general market, but must still be supported, meaning that these are a form of variable features. Describing how a product will include or not include some features may be done using *variation points*, where one point determines which alternatives or options can be present in the product. After setting the variation points, one individual product *variant* is described.

The variability of a software system determines its ability to adapt to changing circumstances. A software product line may include different types of variability. These may be decomposed into three different types: (1) variability as a result of varying operating environment, (2) differently installed hardware or (3) varying mission of the product [12]. To achieve the variation of the software, i.e., to make software adapt to some new circumstances, configuration mechanisms must be used [4]. Many implementation techniques can be used to realize a variation point specified in the product line requirements, which enables a *configurable platform*. One of the methods used to configure the platform to a specific product is parametrization. The software system can use the parameters to adapt at design-, compile- or runtime. The choice of the binding times should be made depending on the situation. For instance, configuring a system following hardware installation should preferably be done early, as the

hardware generally does not change during runtime. Other forms of adaptations, such as calibration, often must be able to occur during runtime.

2.2 Service-Oriented Architectures

As a result of rapidly changing environments, requirements, and generally greater expectations of software, Service-Orientation has become increasingly popular among many different lines of business. Service-Oriented Architectures are constructed by services that encapsulate certain functionality and inter-operate to fulfill some business goal or task. Services are typically autonomous and platform-independent that may be described, published, discovered and potentially dynamically assembled to realize some end-to-end service [28]. Services are by definition loosely coupled, enabling their reuse for different business goals, as well as software flexibility and resilience to technology shifts. A service is described by a *service contract*, showing how the service is used [36]. In the process of combining services, there are a number of different roles. A service that requires another service and makes some request is called the service *consumer*. The service that makes the called service available is denoted as the service *provider*, in this particular exchange. Services can be both consumers and providers in different situations. A service that wishes to provide services to others may do so by registering itself and its available capabilities at a service *orchestrator*. The orchestrator, also called the broker, can then orchestrate the relations between service consumers and providers dynamically during run-time. What has been described here is also shown in Figure 2.1. This aspect of SOA can be realized on any network, although it is viewed as specifically applicable for the internet, but also where different network technologies are used [20]. However, the more dynamic the system is in terms of the emergence of new service consumers and providers, the more one can gain by applying this mechanism, realizing the self-adaptation of a distributed system. This particular aspect of SOA makes it suitable for implementing the adaptive software necessary in dynamic software product lines, as it facilitates scalability, reusability, flexibility, and interoperability [22].

2.3 Service Identification

To build a system that conforms to the principles of a Service-Oriented Architecture, an important procedure to understand is "service identification". It involves how different functionality shall be grouped in order to establish the different services that are e.g., reusable, flexible, and autonomous. This is particularly important during the early stages of the life-cycle of a service-oriented system. An important aspect of service identification is to use a suitable level of abstraction with the right granularity [22].

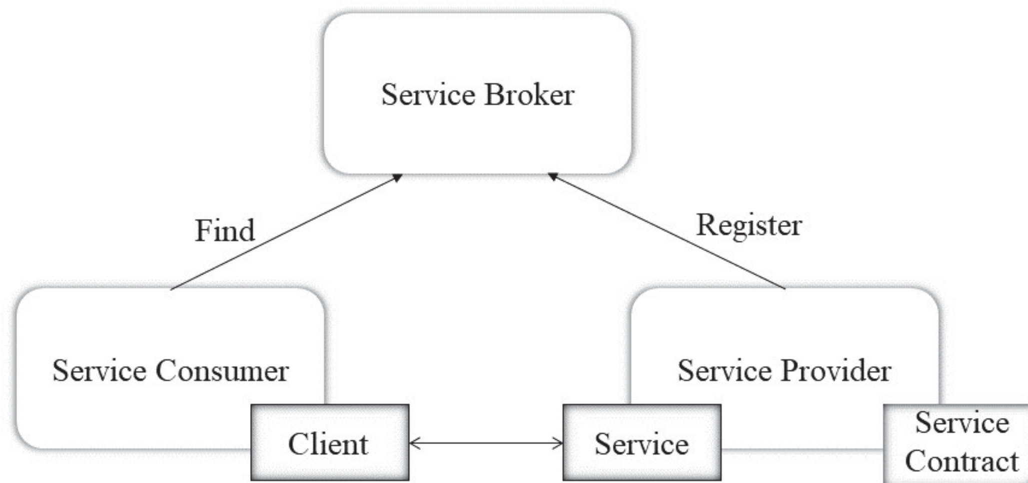

Figure 2.1: Service broker design pattern.

2.4 Relations Between Services in Embedded Systems

Usually, when discussing Service-Oriented Architecture, one often assumes that dynamic binding between services is possible. This is often the case when working with web services, where it is possible to perform service discovery over the internet to find usable service capabilities. It is common to use a *Service bus* that is used by a broker to connect service providers and consumers, as shown in Figure 2.2. However, for closed systems like an embedded system, this is often not achievable. A closed system with a specific hardware configuration that does not change during run-time has a limited need for dynamic service discovery. Relations between services of such systems can be configured at design time and be static from that point on. Service Object-Oriented Architectures (SOOA) or Service-Component Architectures (SCA), can be used for that purpose, where services are represented in the application by different objects that have relations to and from other objects which are used to request *service operations*. In other words, the service orchestration is not performed by the application at run-time, but by the engineer at design-time [34].

2.5 Service Identification to Modernize Legacy Systems

Recently, service identification has been used as a method to re-engineer and reuse legacy systems. A legacy system can be integrated into or used by a new system that is built using SOA. Many approaches to this modernization exist [1]. These methods may include using functional breakdown [22] of the legacy system as well as analyzing the interface [8]. The range of different methods to modernize, or re-engineer legacy systems that exist is a result of that legacy systems are naturally different,

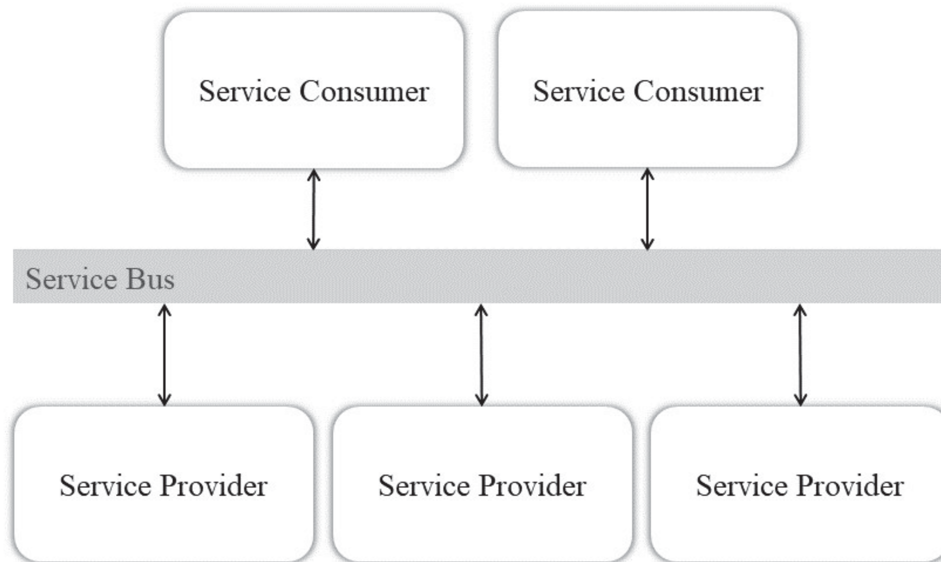

Figure 2.2: Service broker pattern realized using a service bus.

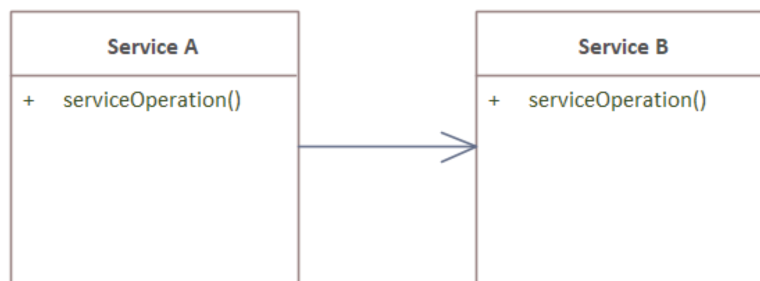

Figure 2.3: Static relations between services.

and different methods for different situations have been required. Generally, service identification and implementation can often be done by software developers without any formal process to follow.

An identified service should represent a capability to perform some task, with a suitable level of abstraction and can have different degrees of granularity. It can represent a complete service in the service-oriented system, or be a smaller component of a service, more resembling a micro-service. To re-engineer a legacy system, one may use different approaches. Employing a "top-down" approach means first performing a high-level decomposition of domain artifacts, modeling and implementing the needed services, and lastly implementing an orchestration of the services. A "bottom-up" approach may start by identifying dependencies to the legacy system, then finding reusable functionality by analyzing the applications that can be regarded as services. Services are then formed by removing the dependency on the legacy infrastructure [1]. Lastly, applications are adapted to use the newfound services. If a bottom-up or top-down approach is not suitable, a third approach to re-engineering by using a "hybrid" strategy [1]. A commonality of most strategies to modernize

legacy systems using service identification is that some wrapper will likely have to be used. The wrapper may for instance convert the interface of a legacy system to an appropriate abstraction of a service or service operation [6].

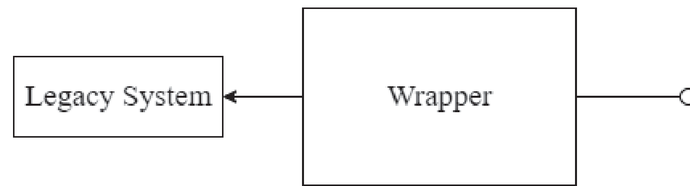

Figure 2.4: Legacy system wrapper.

2.6 Configuration of Services

When configuring a service-oriented system, several methods can be employed. Service Prioritization and Selection are central terms when describing how the system shall adapt to changing circumstances [23]. Service selection regards which service should be invoked at some time based on some conditions, which leads to the configured software system. Configuration is necessary for both static and dynamic services, where static services shall not change during run-time [19]. Different configuration techniques that use service selection can be divided into "Predefined", "Constrained" and "Unconstrained" selection, depending on the level of flexibility [2].

Chapter 3

Related work

We conducted a literature study to find related work of integrating third-party components in an embedded or automotive, Service-Oriented Architecture product line.

To find suitable literature, the following search parameters were used, with different permutations:

- (service-orient*)
- (config* OR variab*)
- (embedded OR automotive)
- (predefined OR "design-time" OR "compile-time")
- ("commercial-off-the-shelf" OR "cots" OR legacy OR re-engineer OR moderniz*)

The goal was to find articles that regard service-oriented software systems within embedded or automotive industries. As well as including descriptions of configuration mechanisms or software variability handling, that primarily is done during design- or compile-time. Some changes to the search string were also made to find additional articles, such as removing keys. Both forward and backward articles snowballing was also used. Table 3.1 summarizes notable related articles reviewed.

Table 3.1: Summary of related work following literature review.

Article	Summary
Model-Based Design of Service-Oriented Architectures for Reliable Dynamic Reconfiguration [26].	Describes a way to model (re)configuration of services in an automotive system during operation (run-time), including different components used during reconfiguration. The method is evaluated by applying it to an example of configuration of brakes. Does not describe how to integrate third-party components.

Model-based resource analysis and synthesis of service-oriented automotive software architectures [17].	Discusses that certain functionality is static from design-time, while others may be configured during run-time. Seeks to develop a method to analyse the installed hardware resources and from that form a service-oriented software system bound at a late time, such that a software component does not need to be tightly coupled to a specific location in the network/distributed system. With late binding, an optimized system and software architecture can be defined. I.e., this study aims to introduce to automotive systems the dynamic service discovery and orchestration commonly found in e.g. SOA's of the web, but not in embedded systems.
Open and Collaborative Micro Services in Digital Transformation [27].	Evolves legacy systems to be integrated into a SOA, based on the "Informatics System of Systems" concept. This is similar to the system (of systems) that constitutes an automotive software system.
Interoperability for Industrial Cyber-Physical Systems: An Approach for Legacy Systems [13].	Describes how to use an interoperability layer to make legacy devices function in a configurable SOA. How to configure the system is not described, e.g. if to configure during design, compile, or run-time.
Integration of a Legacy Automation System into a SOA for Devices [11].	Describes how to use an abstraction(/integration) layer to provide services that can be reused in a SOA. Discusses how not all production systems are "SOA-ready" and therefore there is a need for integration techniques. A service is defined by its interface – "design by contract". Focuses only on devices, i.e. hardware abstractions, and not general systems(domains) or services(provider/consumer). Describes different components of a device: service, interface, and implementation. Uses service discovery to register services, i.e. not at design/compile-time.
Service-Oriented Control Architecture for Reconfigurable Production Systems [21].	Reconfiguration of production systems, using dynamic service discovery.
Migration of industrial process control systems to service-oriented architectures [7].	Reconfiguration of production systems, using dynamic service discovery.
Migration of a Legacy Plant Lubrication System to SOA [24].	Similarly using a broker and a service bus to orchestrate composite services. Requires certain network capabilities often found in web computing.
Towards runtime adaptation in AUTOSAR: adding Service-orientation to automotive software architecture [37].	Released as the software architecture standard AUTOSAR was being fitted with the potential use of Service-orientation. Describes that AUTOSAR, today AUTOSAR Classic, would have to be changed to be compliant with runtime adaption. AUTOSAR Classic is built upon using a federated architecture, whereas dynamic runtime adaption usually requires an integrated architecture.
Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools [9].	Describes challenges when a distributed system shall be implemented by an integrated instead of a federated architecture.

3.1 Research Gap

From reviewing the mentioned articles, many interesting topics can be noted. For more than 10 years, there has been a large effort made into researching Service-Oriented Architectures within the embedded and automotive industries. In [11], a method to reuse legacy software systems in a dynamically adaptive system was described in detail, where an abstraction layer is used to convert the communication interface of industrial devices to fit into a SOA. How to use service-oriented architectures to build a runtime adapting system that may be self-configuring has been deeply covered [17, 26] within the automotive and embedded domains. One area where this behaviour is highly beneficial is the industry of production systems, where a system is composed of a large variety of different devices, such as robots, belts etc. However, of all literature found, none describes how to reuse devices or legacy components in a configurable system without applying modern Service-Oriented mechanisms such as a service broker and a service bus. Having such technologies is not the reality for many within the embedded or automotive industries. For example, it requires the use of ethernet communication, whereas CAN-bus technology is still prevalent in many of the products developed today. Still, there are many potential benefits of using a service-oriented architecture style, e.g., attained by implementing autonomous software components which may increase flexibility, scalability, and reuse. To enable software development of such systems, there is a need for further research on how to get the most out of service orientation when not having all the prerequisites. Such is the case of the problem described earlier, where dynamic adaption is not suitable as the variations are limited and known as they are coupled with an external software component, e.g., a legacy system, device, or domain. High-performance communication technology such as ethernet is not used. This means that the dynamic adaption possible with runtime service discovery and orchestration is not necessary, nor is it available.

Chapter 4

Research Methodology

4.1 Research Questions

Based on the currently available literature and the scope of this study, the following main research question is established:

"What is an effective way to integrate external systems in a Service-Oriented embedded system when high-performance (ethernet) communication is unavailable?"

This can be decomposed into two sub-questions, as follows.

Research Question 1:

What is an effective way to make a Service-Oriented embedded system adapt at compile-time using parametrization to a specific configuration of external systems?

Motivation: A system that is one part of a distributed embedded system, composed of several installed sub-systems, needs to be configured to inter-operate with said installed external systems. Based on the current research, this is primarily done in a SOA by a service broker at run-time using a service bus and ethernet communication. However, how this adaption can be done without such a mechanism is not clearly defined in the available literature. In this study, a delimitation is to use parametrization that configures a system at compile time.

Research Question 2:

What is an effective software design that can be used to integrate identified services of external systems into a Service-Oriented embedded system?

Motivation: Based on available research, external systems are commonly integrated into a Service-Oriented system by employing the Service Consumer and Provider design pattern. Service provider interacts with a service broker which can make them available for service consumers. Without the possibility of implementing this design, services must be integrated in another way. To answer this question, a software design that can be used to perform the corresponding functionality of the service consumer/provider pattern without a service broker and a service bus is required.

4.2 Research Design

First and foremost, the term "effective" from the research questions needs to be defined. In this setting, a solution is deemed effective if it successfully satisfies different stakeholder needs and goals. Stakeholders may range from developers to end-users of a certain product. Thereby, a stakeholder analysis should be included at some point during the research execution to be able to motivate if a software design or a different solution is effective or not.

The research method chosen to develop a possible design is *Design Science* (DS), as primarily established for Software Engineering by Wieringa [39]. To answer the research questions, a practical example of a company that develops software will be used. This problem is to be solved and thereby finding information that may answer the questions. Design Science has been found to be an appropriate method to do this while also discovering knowledge to answer research questions [10]. Alternatives to DS could be e.g. conducting an intervention (experiment)- or feasibility study. To perform any of these methods one must have a possible solution ready to apply and then observe the effects, which may be used to evaluate if the solution is effective. However, a solution is not prepared at this time. This applies to a simulation study as well, where a simulation model is needed. The focus of this study needs to be put on the development and probable effects of a solution itself, not only how it functions when deployed.

In Design Science research, an *artifact* (solution) is investigated and improved in a specific context. This can be implemented in programming code, a structural detail, an electronic system, or some other designed object. In this study, an artifact that is a software design solution is investigated. The artifact can be implemented in the Unified Modeling Language (UML). DS also recognizes that the social context, consisting of a number of different stakeholders, should influence the design solution. At the core of DS research, there are *problem conceptualization*, *solution/artifact design*, and *validation*. Problem conceptualization regards how to understand the problem that needs to be solved by a design. A solution design candidate can then be created that addresses the problem previously established. The candidate is then empirically validated to check if it can be applied to the context and solves the problem. If the candidate is deemed to not be satisfactory during the evaluation, the process is repeated and a new cycle of the process starts. During these cycles, new knowledge is systematically found and used to build the complete artifact. This methodology is described in Figure 4.1 [31]. Central to DS is also the differentiation between *design problems* and *knowledge questions*. Design problems require a change in the world in order to be answered, which is done by meeting the needs of real stakeholders. A knowledge question can be answered only by retaining new information. It is common that a problem is at first formulated as a knowledge question, and then a design problem is solved by a design solution. Depending on the problem type, different methods can be applied as specified by Wieringa [39].

In order to be able to use design science research to answer the research questions, the problem is formulated into a certain context. In this case, a problem is conceptualized

at a software development department at a construction equipment manufacturer, based upon a current practical problem. The engineering cycle framework is initiated by investigating the problem in its context. As the cycle is underway, a number of *knowledge creating activities* are performed to move the process between domains, e.g. from a *problem construct (formulation)* to a design candidate [31]. These activities can be both context/practice-dependent or theoretical. At the final step of a cycle, an empirical validation attempt is done to determine if the solution does solve the conceptualized problem, originating from the research questions.

Problem domain
Solution domain
Theory
Solution Design
- Create design pattern in UML
Problem conceptualization
- Research questions
Problem construct
Design construct
Solution Validation
Implementation evaluation / Problem investigation
- Stakeholder analysis
- Problem analysis
- Focus group method
- Empirical data analysis
Problem instance
Solution instance
- Motivate design choices
- Sensitivity analysis
- Trade-off analysis
Solution Implementation
- Implement in context using UML
Practice

Figure 4.1: Research design.

4.3 Engineering Cycle Framework

In practise, Design Science research is conducted using five phases in a cyclic manner. This cycle is called *The Engineering Cycle*. These phases are presented as follows [31, 38]:

1. **Problem Investigation**: During the first phase of a cycle, the focus is to understand the problem at hand and to describe the problematic aspects.

2. **Solution Design**: A design candidate is created by using the knowledge from the Problem Investigation phase.

3. **Design Validation**: At this phase, the created design is evaluated by investigating if the design would help the stakeholders accomplish their goals.

4. **Solution Validation**: The design is implemented in the context.

5. **Implementation Evaluation**: During this last step, the artifact/design as well as the process is evaluated to increase the knowledge and prepare for a new iteration.

During the development of a particular artifact, one may choose to use only the first three phases of the engineering cycle. This is called the *design cycle*. If it is deemed that the design candidate validated at that point is not satisfactory, it can be appropriate to not proceed to solution validation but start a new cycle and continue to investigate the problem. The following sections, 4.3.1 to 4.3.5, summarizes the stages in the development cycle as found in [31] and chapter 3 of [38].

4.3.1 Problem Investigation

To solve the problem in its context, it is necessary to gain sufficient knowledge. During problem investigation, one seeks this necessary information but does not attempt to solve the problem at hand; the goal is only to describe the problem and its context. Important questions to answer are: what phenomena exist today, how and why do they happen and how does the artifact relate to this? At this phase, an analysis of stakeholders and their needs in relation to the current problem will be performed. This information can then be used to identify the reasons that make the current state ineffective and thereafter also how one should focus on the improvement of the artifact. The ramifications of not solving the problem and leaving it untouched will be investigated. Four different reasons to investigate a problem can be distinguished, where certain aspects are put more into focus than other aspects. One can use a problem-driven investigation approach, a goal-driven, a solution-driven, or an impact-driven investigation. These are not exclusive, as for instance, one could choose to apply a problem-driven investigation and include some aspects of a solution-driven investigation. The selection will depend on the nature of the problem.

4.3.2 Solution Design

During the solution design phase, the artifact is created or improved in an attempt to produce a design solution. This should be possible due to the information gained when investigating the problem. The goals and needs of the stakeholders to the problem should after this step be met, and the description of how to meet them is the proposed solution. This includes specifying functional and non-functional requirements, and how these relate to the needs of the stakeholders. For the requirements, it should also be specified how they may be evaluated by "indicators", which are some measurable properties or other attributes that can be estimated. Design choices can be made by researching available designs, for instance from the reviewed literature. Changes may also be derived from collected data of the focus group or interviews.

4.3.3 Design Validation

The design is here evaluated by examining if the proposed design solution would aid the stakeholders to get closer and meet their goals. During this phase, one should investigate three knowledge questions. The first is the question of having internal validity, i.e. that the implemented design would satisfy the problem specified during problem investigations. This may be decomposed into a causal question, that the solution implemented in the intended context would have sought effects. The question of attained value is also asked, i.e. that the effects attained will satisfy

all stakeholder goals. After internal validity is examined, trade-offs between different variations of design can be explored, and how the variations could satisfy the stakeholders. External validity is lastly examined. Here, the ability of the design implemented in a different context to solve the problem at hand is investigated. In other words, how sensitive is the design to the current context, and can it function as a general solution?

4.3.4 Solution Validation

The design is then implemented. This can be performed in many ways, where for instance a software problem can be implemented in actual code in the intended context, or in a prototype. Another alternative is to express the design in some software model. In this study, design candidates are implemented in UML.

4.3.5 Implementation Evaluation

This step is sometimes regarded as a part of problem investigation, but the purpose to include it as its unique step is to have a checkpoint where it is decided if a new iteration of the cycle should be started. Here, the implemented solution is evaluated in terms of how well it solves the problem and satisfies the stakeholders' needs. If the stakeholders are satisfied and the problem is deemed solved, the research questions shall also be answered. This includes generalizing the solution and building a scientific theory from the contextualized problem and its solution. If the solution is determined to not sufficiently solve the problem, a new iteration of the cycle is started.

4.4 Data Collection

This section describes the methods used to gather the required data for the engineering cycle.

4.4.1 Focus Groups

The *focus group* method is a general, cost-effective data collection tool [15]. It can provide quick qualitative information and be used in several phases of a study and in different research methodologies [15]. One possible flaw of the method is that the attendees of a focus group are required to be knowledgeable of the subject studied. Using focus groups is however particularly powerful when it comes to generating feedback on a topic, or new ideas as well as finding potential problems. In this study, focus groups can be formed by assembling a number of engineers or other practitioners at the company.

To apply the focus group method, one may use the following four steps [15].

1. **Research planning**: Select which topic and questions to study. One should define the scope of the topic and what not to delve into, as well as define a

potential hypothesis. The selected topic should be easy enough to understand and interact with during a short session.

2. **Focus group design**: Decide on how many sessions to perform, how many people in each session to involve, and who to invite. Typically, one session can include four to eight people. Next, the questionnaire to use in the session should be defined. It should also be decided on who to act as a moderator. Group dynamics plays an important role when designing the groups, as for instance age of attendees or knowledge of the topic should be considered. If it is deemed unsuitable to mix people with varying knowledge in the same group, *segmentation* may be used. If the knowledge of attendees will likely affect the outcome of the focus group session, or if time is limited, one may consider using pre-session questionnaires to asses how familiar the people are with the topic.

3. **Conduct focus group session(s)**: The focus group session(s) are hosted. The session should be structured and use a pre-defined schedule. There is an option to ask additional questions during the session, e.g. to divert from the prepared questions and explore new questions. The moderator should help the group by opening deeper discussions in certain areas without affecting the opinions of the group. However, the moderator will also be responsible for keeping with the time of the schedule, ensuring that the allocated time will suffice to go through all questions. It is also required to have a clear strategy of *data capturing*. This may be e.g. recording audio, video, or taking notes. Ideally, data capturing should not be done by the moderator if it requires actively working with it during the session. One could also consider having a post-session meeting, where interpretations of the sessions are clarified with the group. During the sessions, a discussion is instigated by the moderator focused at the design artifact(s) to find potential threats and points of improvements. Data is collected during this discussion which are later used to proceed in the design science cycle.

4. **Data analysis and result reporting**: Lastly, the data from the session is analyzed. The method used for analysis is described in Section 4.4.4

Focus groups are used as a data collection method to gather empirical data. This will be required during *problem investigation*, *design validation* and *implementation validation*. To collect data from the focus group session, someone will be responsible for taking notes. A follow-up email with open- or closed-ended questions are included as support for the attendants to answer. Individuals are also asked to explore further and explain certain remarks made during the session. The questions are formed based on what stage at the engineering cycle is in focus during that session. For instance, to gather data at the problem investigation, questions that regard involved stakeholders and their needs are asked. The participants of focus groups are working with software development, with different roles, at the company where the study is performed.

4.4.2 Interviews

Interviews may be used to collect qualitative data, where series of unstructured, semi-structured or fully structures questions are asked to an interviewee. In this

study, semi-structured interviews will primarily be used. The purpose of using semi-structured interviews is to allow for spontaneous exploration, e.g. of the problem description or design space. Similarly to the data collected from focus groups, the empirical data collected by interviews are primarily required for *problem investigation*, *design validation* and *implementation evaluation*. Informal Conversational Interview can be used throughout the execution of the design cycle. However since it does not follow a pre-defined structure the answers cannot be compared and are therefore unreliable. Informal interviews will be avoided to some extent but are likely to occur during the execution [32]. The use of interviews will be prioritized over focus group sessions depending on the availability of attendees, or when a lengthy conversation might be appropriate. In such cases one may prefer a smaller group to carry the discussion.

4.4.3 Literature Review

Reviewing literature can be used during the solution design phase of the engineering cycle. This can include finding potential design candidates, e.g. design patterns. Especially the literature reviewed in related work are of interest, but also the work cited in the background. It is less appropriate to use literature for data collection during other phases, e.g. problem investigation, design validation and implementation evaluation as these often require empirical data from the specific context. The applicability of data found through reviewing literature is dependent on how general a research question, or problem, is. If the practical context dependent variables can be excluded from a problem, then other sources of data not gathered in the same context may be used. Runeson et al. describe this in [31], by describing a separation of a problem- and solution domain, and a separation of theory and practice.

4.4.4 Data Analysis

The method to be used to analyze data will depend on the nature of the gathered data, which in turn depends on which phase of the design cycle is currently ongoing. Early in the process, primarily exploratory data will be gathered to seek new knowledge about the problem and its context. This means that open-ended questions are used in favor over closed questions. The collected data will naturally be unstructured and in that case, thematic analysis [5] will be used to find patterns and themes which can be used to progress in the engineering cycle. As the exploratory data is obtained and a solution begins to take form, the collection method used is increasingly more close-ended questions that can be used to validate a candidate design or implementation. The data should be less open for interpretation so that the validation is reliable.

Chapter 5

Results

5.1 Cycle I

To gather data for the first cycle, interviews were used during *problem investigation* and *solution validation*. A focus group session was held to collect data at the *implementation evaluation*. It was prioritized to select people with a background working with software architecture to some degree since understanding software architecture is a big part of the problem statement. It should also be noted that everyone included has experience in working with software development in general, such as coding, as well. Software architects and product owners at the company regularly code and review code. During implementation evaluation, a larger group was chosen to be included to gain more viewpoints on the design.

Table 5.1: Data gathering, 1st cycle.

Phase	**Collection method**	**Person**	**Role**
Problem Investigation	Interviews	P1	Software Architect
		P2	Software Architect
		P3	Product Owner
Solution validation	Interviews	P1	Software Architect
		P2	Software Architect
Implementation evaluation	Focus group	P1	Software Architect
		P2	Software Architect
		P3	Product Owner
		P4	Software Developer
		P5	Software Developer
		P6	Software Developer

5.1.1 Problem Investigation

Firstly, we establish a baseline that describes the current state of software development at the company of Volvo Construction Equipment in Braås, Sweden. Here, articulated haulers are developed and manufactured that are to be used primarily for construction and mining operations. In this study, we focus on the interaction between the software system developed at the Braås site, and software systems that it collaborates with to compose the full product functionality. The organisation in Braås carries full machine responsibility, meaning that it is responsible for asserting

that all systems have been integrated properly. The software system developed here is thereby commonly an "integrator" in the machine system-of-systems. This *internal* system interacts with many other systems, but those systems rarely interact with each other, instead they signal the internal system that relays data to other external systems. The internal system of the Braås department interacts with the following external systems:

1. A complete drive line system.
2. An inertia monitoring unit (IMU).
3. An engine system.
4. An electro-mobility system
5. A gear lever device.
6. An autonomous system that can control the machine and replaces a human operator.

These external systems are very different, as some are strongly coupled with a hardware device e.g. the gear lever and IMU, whereas some are complex systems. The internal system can be a client (requesting data) or a server (providing data) in different situations,

Software architects at the department have decided to employ Service-Object-Oriented Architectures to design the system. The architecture is composed of different *Service Objects* that are reusable, autonomous objects that together realize end-to-end services. A reference architecture for large vehicles, developed internally within Volvo AB resulting from an object-oriented design analysis of a large vehicle, is used. From this reference, service objects that have certain responsibilities are selected for use in the developed system, i.e. architects migrate objects that hold some suitable responsibilities from the reference to the actual software architecture. Along with responsibilities, the relations between them are also carried over.

To integrate the external systems into the software architecture of the internally developed system, software architects have decided to denote some service objects as "external" service objects, and thereby incorporate the interactions between systems. External service objects can be both acting as a client or server in relation to internal service objects, as shown in Figure 5.1. This originates from the fact that external systems are required to be able to act as a client and/or server.

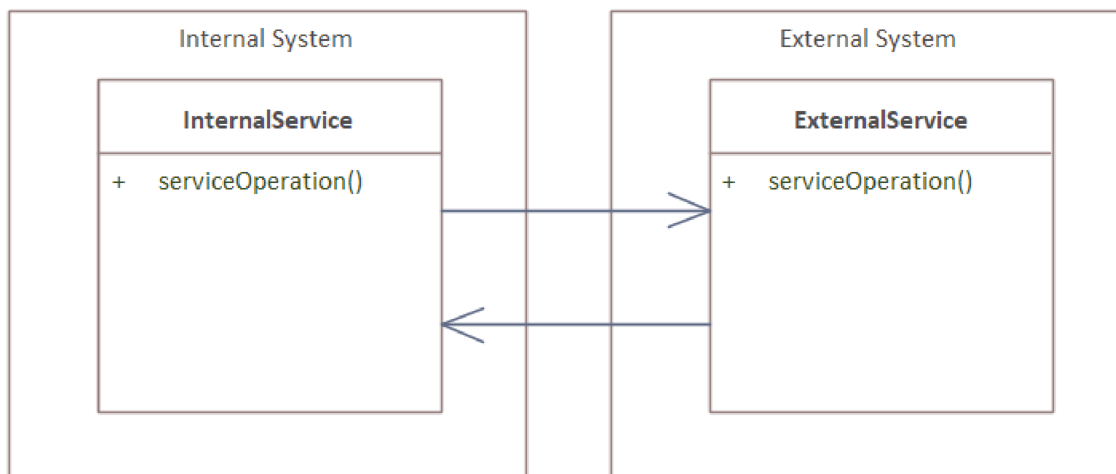

Figure 5.1: Relations between internal and external services.

The architecture, i.e. the responsibilities of- and relations between service objects, that is obtained by using the process of selecting parts from the reference architecture, is not easy to realize and follow in practice. The responsibilities of external systems are pre-defined, since the interfaces to the systems are set before acquisition of the external system, e.g. purchasing a commercial-off-the-shelf component. These interfaces do not necessarily correspond to the division of responsibilities by the service objects of the reference architecture. The responsibilities of one service object from the reference architecture are often *shared* between the internal and an external system. This is a result of the division of responsibilities between service objects in the reference architecture, formed by object-oriented analysis, that was not done with the various systems in mind. This leads to the creation of new service objects, not included in the reference architecture, by splitting the responsibilities of some service object from the reference architecture. While external services are described in the software architecture as similar to internal ones, they differ in implementation. Internal service objects are instantiated classes that contain often complicated logic, functionality representing how some aspect of the product should behave. The external service objects are realized as proxies or stubs, with no logic more than perhaps interpreting CAN-signals. External service objects are a part of the internal system, but only act similarly to a translator between the internal and some external system. Proxies are used for all method calls/service operations where an internal service object acts as the client. Stubs are used when the internal service is the provider/server. This is described in Figures 5.2 and 5.3. One reason to use this design pattern is to make the system agnostic to deployment and communications media, so that the internal service objects do not have dependencies to any specific communication media data. This information is encapsulated in the proxies and stubs.

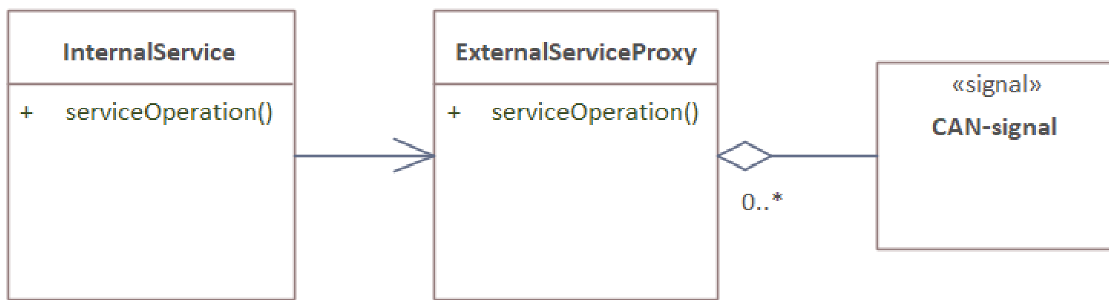

Figure 5.2: Implemented design of external service objects when providing services.

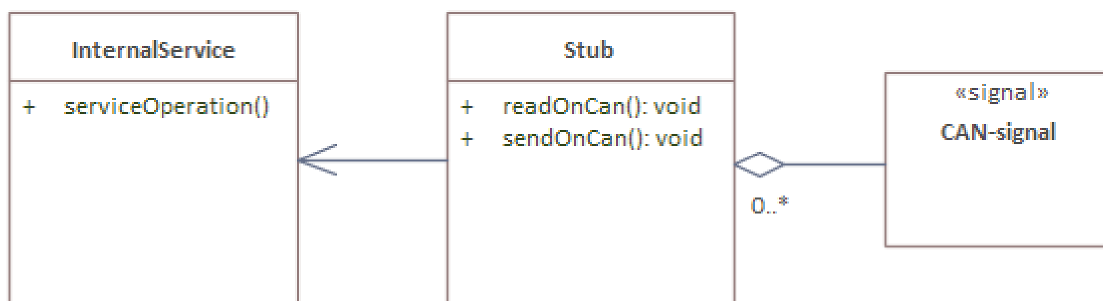

Figure 5.3: Implemented design of external service objects when consuming services.

Next, we explore an example of having an installed combustion or electric engine. This is one example of variants that the system shall manage, as shown in Figure 5.4. It describes how two internal service object classes, *CombustionEngine* and *ElectricEngine* are used to interact with one specific external system each, represented by one proxy and stub class per internal system class. These proxies and stubs hold a subset of responsibilities of the service object established from the reference architecture. The internal service object classes hold the remaining functional responsibilities and a parameter that is used to configure the system at compile-time. Classes are separated by interfaces. The proxies and stubs are communicating via CAN through send and receive signals.

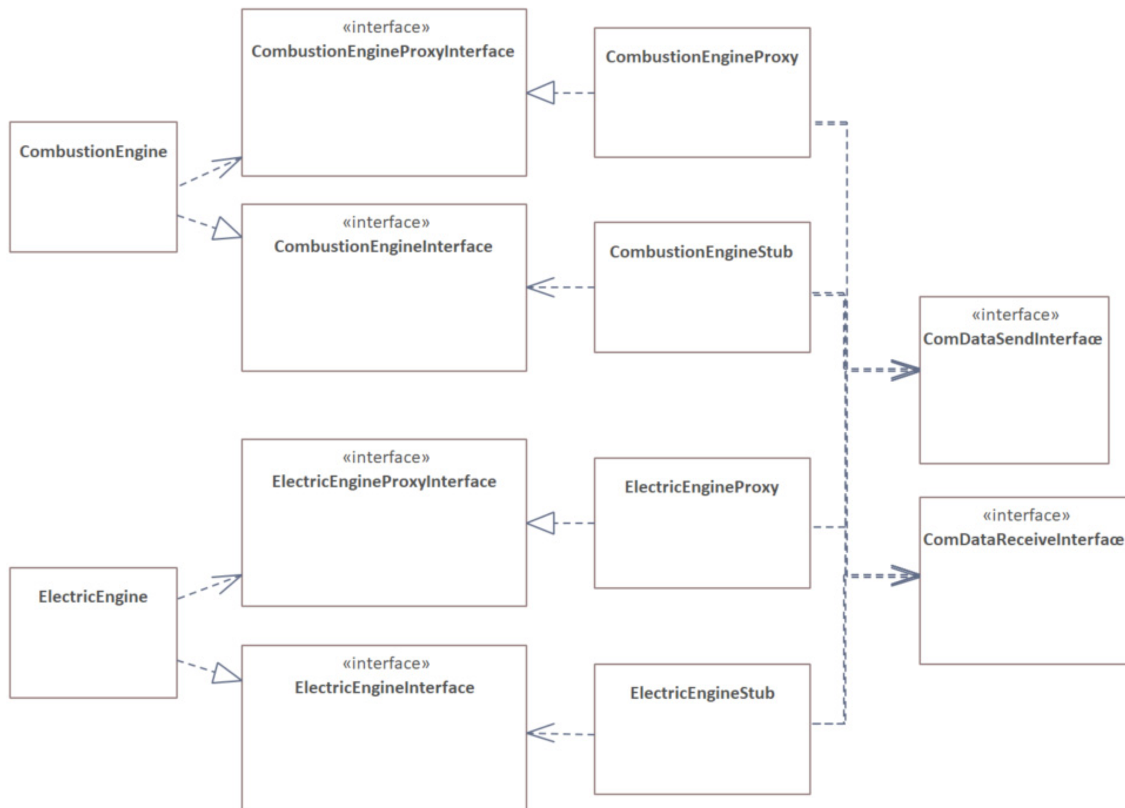

Figure 5.4: Baseline implementation of installed combustion or electric engine.

This phenomenon may be generalized. A service object is either defined based on some reference architecture or by an analysis of the complete product that results in an object-oriented design. The results of either method are an object-oriented design constituted by a number of service objects having certain responsibilities and relation between these objects. In Figure 5.5, the responsibilities are denoted as responsibilities X, Y and Z. The total responsibilities(and functionality) of a whole product are often divided between several systems, e.g. when the development effort is a collaboration between organisations. As a result of responsibilities of some external system being pre-defined before development of the internal system, it entails that the interface to the external system is already set. In SO terms, the *interface* corresponds to *available services* provided by some system. For either of the variants, it may require to both request and serve data between systems, leading to that a proxy and stub will be implemented. The systems are denoted as the internal system and external systems A and B.

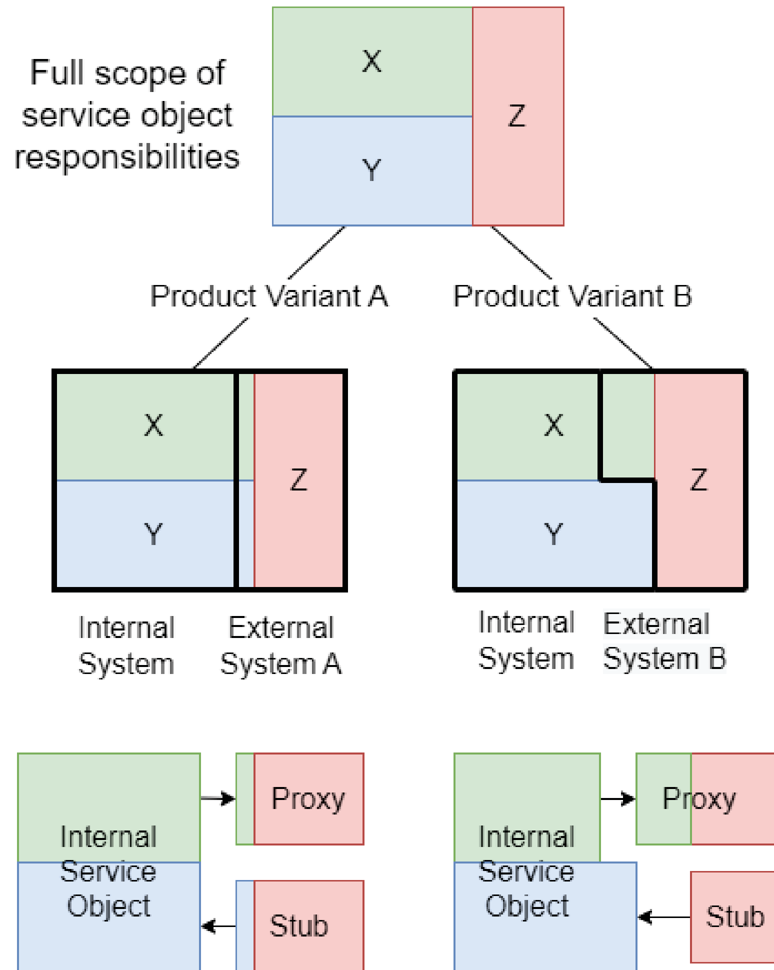

Figure 5.5: Architectural differences between products in a line.

Since the implementation of the internal service object and proxies and stubs are dissimilar between variants, it has the effect that the structure of service objects will be different between variants, i.e. the relations between classes. As explained, it originates from the fact that the "split" of service object responsibilities between systems is different depending on product variant. With the current way of working, this effect will persist and a common structure reused between variants cannot be defined by placing external systems in "external service objects". This also causes the software to be less understandable for instance when debugging. Analyzing a piece of functionality may be allocated to a certain service object in variant A, but divided in one internal service object and one external service object in variant B. We can also see that the internal implementation is heavily reliant on the CAN-interface obtained from external systems.

The system is a federated distributed embedded real-time system, where each system is deployed on a dedicated ECU that is connected to a CAN. The software binary deployed is one monolithic component per ECU. Only static memory allocation is used, meaning that all objects are instantiated regardless of whether they are necessary during run-time or not. Testing is done on several levels, including unit-testing

(L0), simulated binary testing (L2), Hardware In the Loop-tests (L4) and complete machine testing (L5).

The problem described in Section 5.5, originates from attempting to follow a reference architecture, or a certain object-oriented design, when developing a software system-of-systems that is composed of systems from different organisations. In short, the reference architecture cannot be realized in the embedded object-oriented system since responsibilities of single objects found in the reference are shared between multiple organisations. We now identify three stakeholders that are affected by this problematic phenomenon. First, the software architects and developers that are end users of the design solution and who implement the code. A fitting design would for instance decrease work load as reuse of software components is enabled. The engineering project client is another important stakeholder. He/she has a vested interest in that the development time and cost is reduced as much as possible. The results of a stakeholder analysis is described in Table 5.2.

Table 5.2: Stakeholder needs analysis.

Stakeholder	Need
Software architect	Reduce work of architecture definition
Software developer	Reduce work of code implementation Understand system Test system Integrate different systems Use system in several products
Software project client	Reduce cost Reduce time-to-market Sufficient time behaviour Sufficient functional safety

5.1.2 Solution Design

We formulate requirements based on the problem investigation. Table 5.3 and 5.4 describe functional and non-functional (i.e. quality attributes) requirements, as well as a motivation for their inclusion and which design decision that follows as a result.

Table 5.3: Functional requirements specification.

Functional requirement	Motivation	Design Decision
The design shall include a software element that can adapt at compile time to different configurations using a parameter value.	Configuration of the system by parametrization is a constraint from the current way of working at the company. This must still be possible.	Keep parameters in Service Objects used to configure system.

The design shall include a software element that can convert CAN-signals to identified externally provided services.	Software developers have described that the current integration of external system is very coupled with the used interfaces to external systems. It limits and constrains the development and implementation of the internal system. This appears to be due to use of the proxy-stub design pattern used to communicate with "external service objects".	Use a wrapper-based design to define identified services of external systems
The design shall include a mechanism for the internal system to request external services.	The internal system is required to be able to act as a *client*, and request data or actions of external systems.	The service objects of the internal system can request services found in the wrapper(s)
The design shall include a software element that make internal services available for external systems on the CAN-bus.	The internal system is required to be able to act as a *server*, and acting upon requests of external systems.	A wrapper based solution is used for this case as well. Here, a component that makes requests of the internal system based on signals on the CAN is created.

Next, the stakeholder needs are formulated as non-functional requirements, accompanied by indicators.

Table 5.4: Non-functional requirements specification.

Quality Attribute	**Stakeholder need/Motivation**	**Indicator(s)**	**Design Decision**
Reusability of architecture, scalability	Reduce work of architecture definition, reduce cost, reduce time-to-market	Reused software structure elements (commonalities), effort required to modify solution	Adhered to by using wrappers to define and use identified services of the internal and external systems.
Reusability of code, scalability	Reduce work of code implementation	Reused lines of code, effort required to modify solution	Adhered to by using wrappers to define and use identified services of the internal and external systems.
Complexity	Understand system	Number of branches, descriptive documentation	Adhered to by using wrappers to define and use identified services of the internal and external systems.
Testability, complexity	Unit-test system	Number of branches	No action, deemed not worse than current approach.
Interoperability	Integrate different systems	Solution being compatible with different systems/Effort required to add new external system	Adhered to by using wrappers to define and use identified services of the internal and external systems.
Performance	Sufficient time behaviour	Execution time to responses	No action, deemed not worse than current approach.

Reliability	Sufficient functional safety	Mean time between failures	No action, deemed not worse than current approach.

The proposed software design is illustrated as UML class diagrams in Figures 5.6 and 5.7.

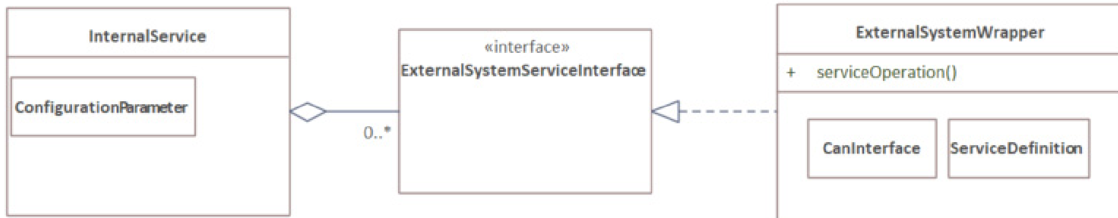

Figure 5.6: Services identified of external systems.

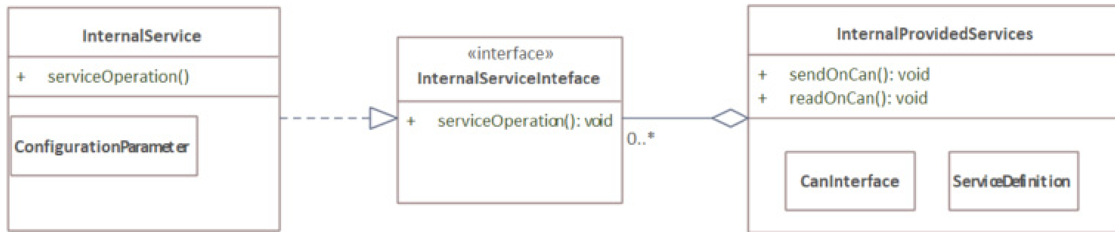

Figure 5.7: Services identified of the internal system.

5.1.3 Solution Validation

To convert the CAN signals to services can be done in several ways, but one very common way is to use a wrapper [1,6]. This wrapper owns both the interface to the available CAN signals as well as how to use the signals to create usable services. We decided to use one wrapper per external system. Another alternative is to use a complete abstraction layer as done in [11] that converts the API of all external systems to identified services. The rationale behind our decision of using one wrapper per external system is to reduce the ripple effects that may occur if one supplier changes the API of its delivered system. The identified provided services from an external system are available to internal services by an interface, as we deem it important to adhere to the interface segregation principle, one of the SOLID [25] principles. These external services are necessary for the internal service object to use in order to fulfill its responsibilities, which is the rationale behind the *dependency* relationship between the `InternalService` and the `InternalServiceInterface`. One can also argue that the *single responsibility* principle is upheld, which should make the classes testable.

An important property to note is that using this design would mean that the *ownership of data* is moved from the so-called *external services* into the internal domain, in comparison to the current state described in Section 5.1.1 and Figure 5.5. Presently,

the ownership of data is allocated to the external systems, when the external system supplies the service/functionality over CAN, through a proxy or stub object. However by using the proposed solution, as an example, if an internal service fetches vehicle speed using externally provided services, vehicle speed data ownership is now included in the responsibilities of the internal service. In other words, this design makes the internal system incorporate also external system functionality. This occurs since some responsibilities that were previously allocated to some proxy or stub will be allocated to an internal service (object), for instance the responsibility of knowing the vehicle speed. However, that is only the case for functionality or data that the internal system requires to fulfill its total responsibilities. There can still be many more responsibilities of external systems that are hidden from the viewpoint of the internal system. Meaning that there is functionality in external systems not needed to be known in the internal system. This should have the effect that the architectural structure of the software can be reused over an entire product line, even when replacing the usage of different external systems between variants.

Furthermore, another design is needed for the requirement of making internal services available for the external system. The design selected for this is similar to a gateway, and its responsibilities is to make services available for consumers on the CAN. This can be both to publish data (for getter service operations) and act on requests from an external system (i.e. setter service operations). To perform this, it is required to use a number of relations to interfaces of internal services, and cannot perform its assigned responsibilities without using these interfaces.

Both design patterns, described and shown in Section 5.1.2 are general and we deem them insensitive for different contexts, while also being able to scale by adding or extending wrappers. The candidate design is not dependent on CAN being used as the communication method between systems, but in this draft the relation to CAN-signals is specified as a property of the wrappers. To be able to properly evaluate the design and if it fulfills all stakeholder needs, the design is instantiated in a sample context which in turn can also be used for evaluation.

Table 5.5 summarizes how and whether or not a requirement previously elicited has been fulfilled by the candidate design.

Table 5.5: Requirements fulfillment motivation.

Requirement	**Fulfilled by design decision**	**Motivation**
The design shall include a software element that can adapt at compile time to different configurations using a parameter value.	Keep parameters in Service Objects used to configure system	Fulfilled as it is the current way of adapting system.

The design shall include a software element that can convert CAN-signals to identified externally provided services.	Use a wrapper-based design to define identified services of external systems	The wrappers provide a software element where one can translate and process the CAN-signals to appropriate methods that can be accessed by the internal software system.
The design shall include a mechanism for the internal system to request external services.	The service objects of the internal system can request services found in the wrapper(s)	Service Objects of the internal system may have dependencies/relations to the wrappers containing identified services, in order to fulfill its responsibilities.
The design shall include a software element that make internal services available for external systems on the CAN-bus.	A wrapper based solution is used for this case as well. Here, a component that makes requests of the internal system based on signals on the CAN is created	A wrapper that has the responsibility of providing services by reading and writing data to the CAN-bus will fulfill this. When an external system requests an action of the internal system, this software element will call certain methods of service objects based upon some value in one or more CAN-signals. When an external system requests data of the internal system, it will fetch data from service objects and set an appropriate value to one or more CAN-signals.
Reusability of architecture, reusability of code, scalability, complexity, Interoperability	Use a wrapper-based design to define identified services of the internal and external systems	Removing external service objects shall provide less classes which decreases code base. Service objects found in the internal system will incorporate all responsibilities related to the entire internal system by using the wrappers. This should increase the ability to reuse the software structure between variants.
Interoperability	Use wrappers to provide a layer of software between systems.	The use of wrappers between the internal and external systems de-couples the systems from each other, making them less dependent on a specific implementation of other system.
Complexity	Use wrappers to provide functionality to the internal system.	The removal of external service objects and instead letting the internal systems incorporate that responsibilities should increase commonalities between variants. This problem is described in 5.1.1. The more common software structure should make it easier to understand the system. The wrappers defined shall provide understanding of interfaces between systems

5.1.4 Solution Implementation

We instantiate now the design solution, illustrated in Figure 5.6 and Figure 5.7, in an exemplary implementation. In Figure 5.8, the previously used example of having a combustion or electric engine installed is used. There are two classes representing the service objects holding a parameter that can determine which class is to be used for the current configuration. These two service objects each depend on one wrapper, `EngineSystemServices` and `ElectroMobilitySystemServices`, to request actions and data as a client and thereby fulfilling their responsibilities. There is one wrapper per external system, encapsulating its services.

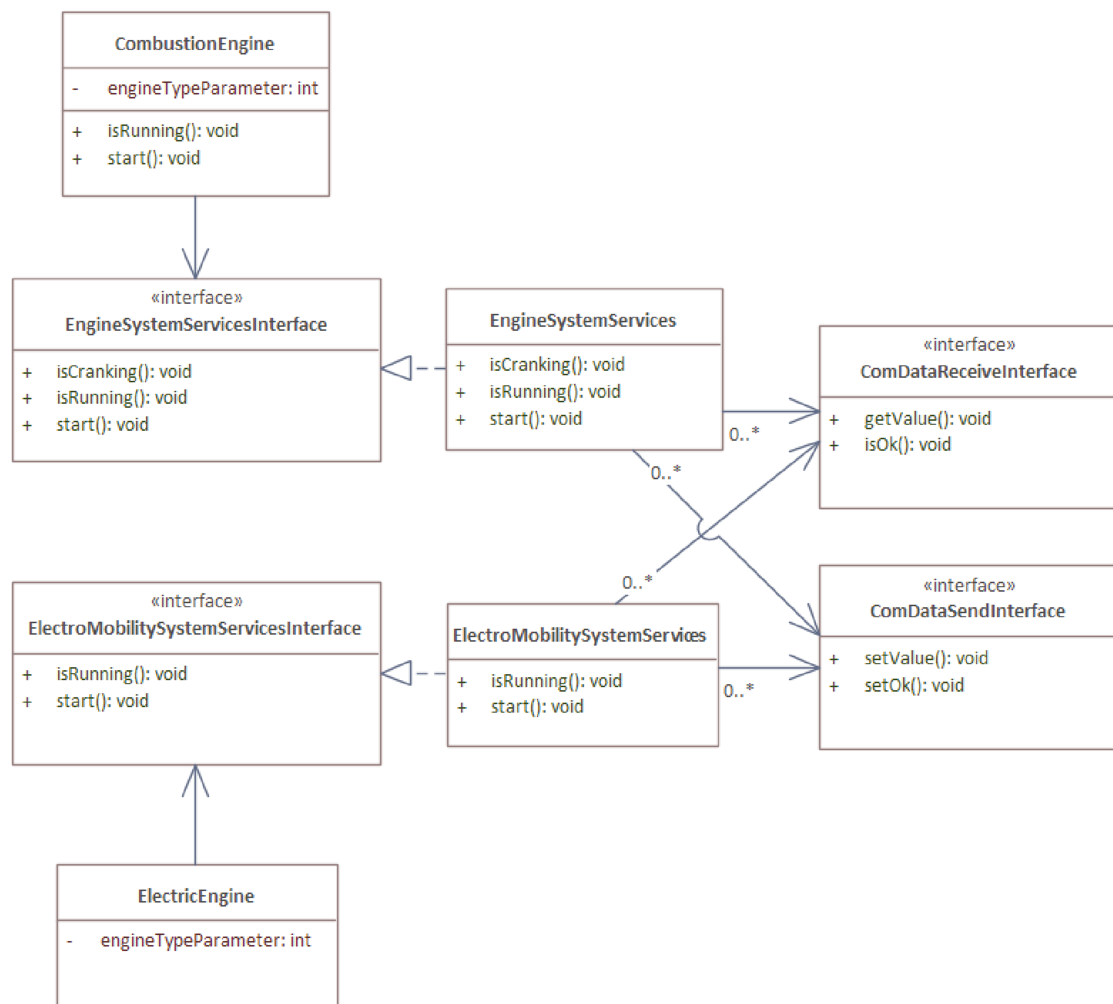

Figure 5.8: Instance of external provided services.

Figure 5.9 illustrates how the wrapper for when the internal system is providing services, i.e. acting as the server, is instantiated. A wrapper class, *ProvidedServices*, fetches data from some service objects, *ConvetionalTorqueSupplyInterface*, *CombustionEngineInterface* and *MotionInterface*, to output on CAN or makes requests to service objects based on the values of CAN-signals.

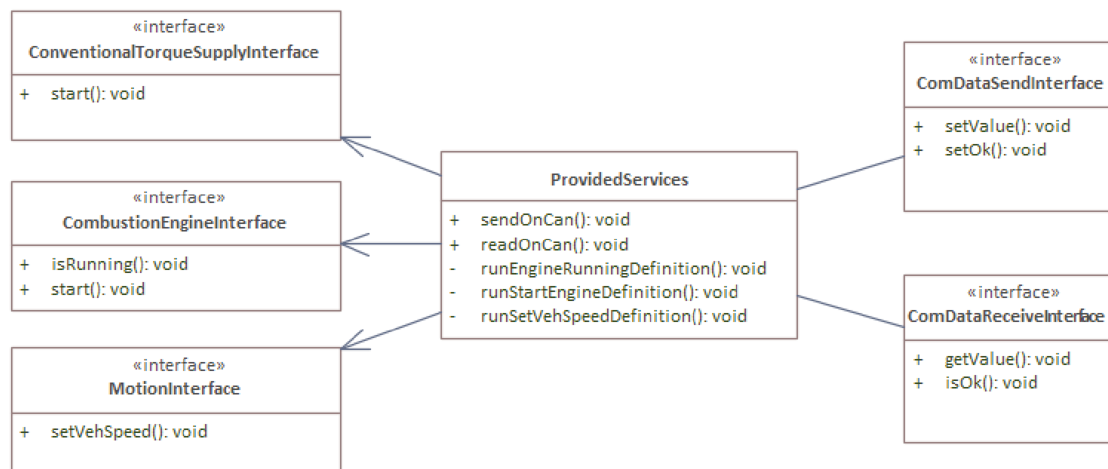

Figure 5.9: Instance of internal provided services.

5.1.5 Implementation Evaluation

During a focus group session, we evaluated the candidate design and its implementation in relation to the stakeholder needs that have been established. These results are explained in Table 5.2.

Table 5.6: Stakeholder needs satisfaction.

Stakeholder need	**Satisfied?**	**Motivation**
Reduce work of architecture definition	Yes	The candidate design would lead to more ability to keep the same software structure in the product line. Service objects and relations between them is more likely to be common in the product line. This should decrease to work needed to create new suitable software structure(s) in different variants.
Reduce work of code implementation	No clear result	The proposition will lead to both code reduction and increase. Some of what is currently external service objects will need to be refactored to internal/regular service objects. The wrappers will also need to be created and maintained. However, some of what is currently external service objects will be possible to remove.
Understand system	Partially	The new way of handling/integrating external systems will require some effort to understand, including how the wrappers are to be used. However the internal system will become easier to understand.
Test system	No change	The attendants can not clearly argue that the proposal is better or worse than the current way.
Integrate different systems	Yes	The respondents think that the design proposal will give better ability integrate the internal system with external systems, from having a wrapper/layer between them which decouples the systems.

Use system in several products (lines)	Yes	Just as the integration is made easier, the developers asked express that it will also be easier to use the software systems in several product line. However, that does not say that it will be possible and appropriate to do, just that it is likely easier.
Reduce cost	Yes	The increase of commonalities of the internal system for different variants should make it easier to set the architecture for them. Currently, a lot of work is needed to find suitable architecture, i.e. define internal and external service objects and relations between them, but this is expressed to be easier using the proposal.
Reduce time-to-market	Yes	According to the software developers asked, a decreased time-to-market can be obtained for the same reason as cost is reduced.
Sufficient time behaviour	No change	We assess that the performance is not affected negatively.
Sufficient functional safety	No change	The software engineers at the session do not think that functional safety of the complete product will be affected positively or negatively.

During the implementation evaluation, two additional stakeholders are identified. Software system testers verify the software using the CAN-bus signals to verify that ECU's perform as intended on a Hardware-In-the-Loop rig. Furthermore, developers of external systems have a need to be able to see how the internal system can be used by their system. Both stakeholders need to be able to view the interface between systems, which is in this case the CAN-signals. Additionally, it should be possible to define different interfaces to specific external systems when the internal system provides some services. That is, not all provided services of the internal system should be available to all other systems. These new requirements are collected in a new stakeholder needs analysis in Table 5.7.

Table 5.7: Added stakeholder needs analysis.

Stakeholder	**Need**
Software system tester	Understand system interfaces
System developer	Decrease bus load
External system developer	Understand system interfaces
System developer	Use solution for communication between systems internally of a node

We also learned that the solution should not be specific to systems that always have dedicated ECU's (nodes). For similar distributed systems that employ a federated hardware architecture, the different systems will likely be put on individual nodes and communicate using some bus technology. However, it is also possible that two separate systems are placed on the same node and do not need e.g. CAN-signals to communicate, but can do so by internal software signals.

Some signals or services will be *safety classed*, meaning that they are involved in the control of a certain safety classed function. Braking and steering are typical examples of this. The safety classes signals and services should ideally stand out in some way to separate them from non-safety classed services. Signals that are coupled with safety functions may have stronger requirements applied, such as redundancy.

The shift of data ownership from an external system to the internal may entail that the bus-load increases. This can happen since for instance an internal service that needs to know an attribute that is obtained from a provided service of an external system, should now get the data from an internal service. If these two internal services are allocated to different nodes, a CAN-signal is needed to move the data between nodes. However, there must already be a signal carrying the data from an external system node to the internal service that now owns the data. There will then be two CAN-signals carrying the same value, one is sent from an external node and one from an internal.

The purpose of the wrapper can be further specified by calling it an *adapter*, which is a common design pattern. An *adapter* is commonly used to enable objects with incompatible interfaces to communicate, but also a specific form of a *wrapper* [3]. Even though adapters are a form of wrappers, using the adapter notation provides some insight into how it works and what its purpose is. Since not all stakeholder needs are met (see Table 5.6, we decided to start a new design iteration with this new information.

5.2 Cycle 2

To gather data for the second cycle, interviews were also used during *solution design*, and *solution validation*.

Table 5.8: Data gathering, 2nd Cycle.

Phase	**Collection method**	**Person**	**Role**
Solution Design	Interviews	P1 P2 P3	Software Architect Software Architect Product Owner
Solution Validation	Interviews	P1 P2 P3 P5	Software Architect Software Architect Product Owner Software Developer
Implementation evaluation	Focus group with post-session questionnaire	P1 P2 P3 P5	Software Architect Software Architect Product Owner Software Developer

5.2.1 Solution Design

From the implementation evaluation in Cycle 1, we found some new functional and non-functional requirements as described in Tables 5.9 and 5.10. Additionally, remarks were provided concerning safety classed CAN-signals. We deem to not take these into account further, as the proposal should not affect it negatively, and some delimitations of the scope are necessary to uphold to keep the scope reasonably small.

Table 5.9: Added functional requirements specification, 2nd cycle.

Functional requirement	Motivation	Design Decision
The solution shall include a mechanism to dedicate internally provided services to specific external systems.	Dedicating a specific set of services to a certain external system is needed to control the services provided to some external system. Otherwise, changes of a service that some external system requires will be forced onto others as well, which might not be possible.	Define one wrapper per external system for providing services of the internal system
The solution shall be non-sensitive to method of communication between systems.	A case already exists where more than one system (from more than one organisation) are deployed to one node, and therefore the CAN-interface does not exist.	Replace the references to CAN with a more general communication media.

Table 5.10: Added non-functional requirements specification, 2nd cycle.

Quality Attribute	Stakeholder Need/Motivation	Indicator(s)	Design Decision
Understandability	Understand interfaces between systems	Effort required to read and view system interfaces	Add a new architecture view type to the solution. Component diagrams will describe the dependencies between entire systems.
Performance	Decrease bus load	Bus traffic required	Introduce a way-of-working to reduce required CAN-traffic of the internal system.

We decided to include a new diagram in the solution to adhere to the requirements of describing how interfaces are dedicated to specific external systems, and fulfill both new non-functional requirements. To this end, we added a component diagram type, where a component represents a specific system, either the internal or an external system. Such an example view is shown in Figure 5.10, where each system has some important attributes. A system can expose a *providing* interface, represented by ball notation, or a *required* interface, represented by a socket. A system may expose an arbitrary number of providing and required interfaces. Each ball/lollipop can be traced to an interface that is used by an adapter(/wrapper) so that there is a traced relation from a providing interface to an adapter that defines usable services. These

are included in the class diagrams of Figure 5.11 and Figure 5.12, represented by `ExternalSystemComInterface` and `ProvidedServicesComInterface`. The interfaces are an extension of a specific *port*, which can be either one or more CAN-busses that the system has access to, or an node internal API. This is the case when several systems share an ECU.

One addition to the class diagrams describing how services are defined from an external or the internal system was a configuration parameter to the adapters. This parameter can be used in the situation that two or more external systems are used for the same functional purpose. For instance, one may employ two different Inertia Monitor Units from perhaps different subcontractors, that fulfill the same functionality but have a different CAN-signal interface. In this case, one can use a parametrization of the adapter that is aware of configuration and thus contain the ripple effects to this level.

We found that it is highly important to be able to reduce CAN-load as a consequence of changing data ownership by some mechanism or strategy. This is described as follows, and shown in Figure 5.13:

- Each adapter (`ExternalSystemServicesAdapter` in Figure 5.13) of external services shall be able to be instantiated using a subset of the interface required to define all identified services. This gives the ability to deploy the adapter on all nodes, and route e.g. a few (CAN-)signals to a node that requires some service of the adapter. This can for instance be realized using an *builder* design pattern [3].

- The internal service, at node X (`InternalServiceA`), that some other service, at node Y (`InternalServiceB`), wishes to invoke shall be deployed as a proxy on node X (`InternalServiceAProxy`). This proxy has the knowledge that certain methods of the internal service interface can be obtained directly from the adapter. This means that one relies on the competence of the software developer to deduce when one can use the adapter interface instead of an internal service interface. In other words, a developer is required to know or be able to find out whether to call an adapter or the service object on a different node.

- The proxy then calls the adapter on the same node to retrieve the necessary information.

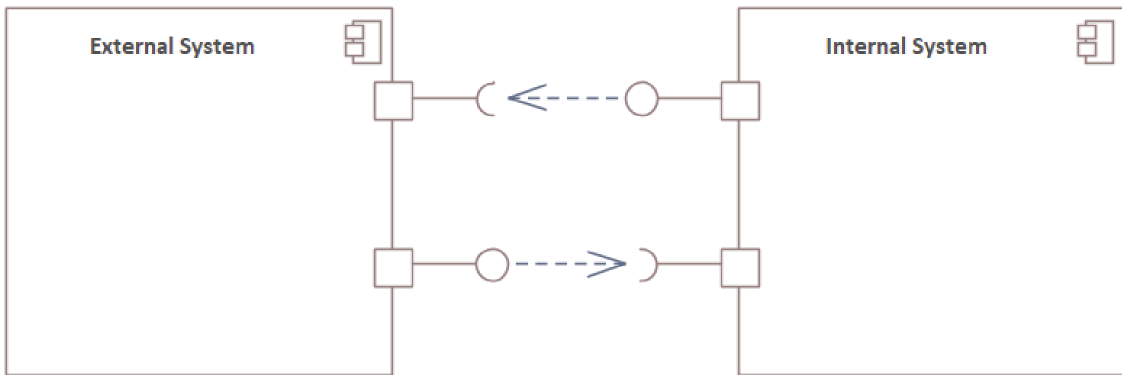

Figure 5.10: System view using component diagram.

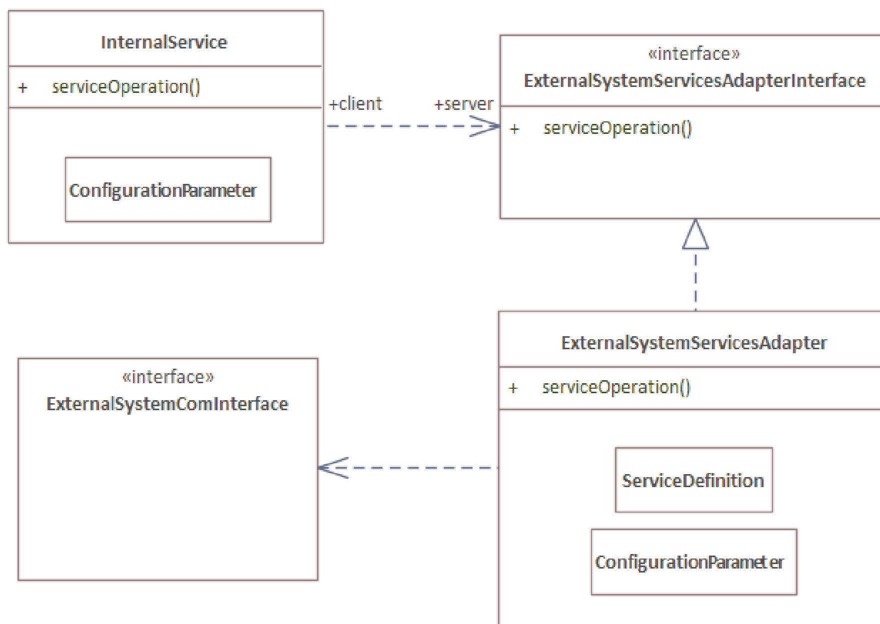

Figure 5.11: External services adapter pattern.

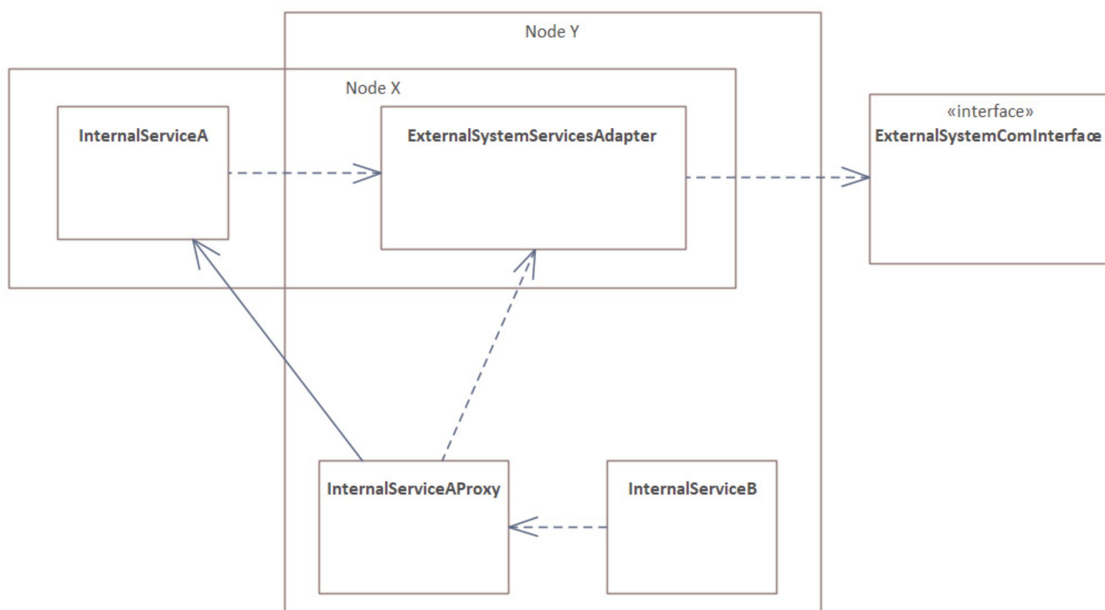

Figure 5.13: CAN-load reduction strategy.

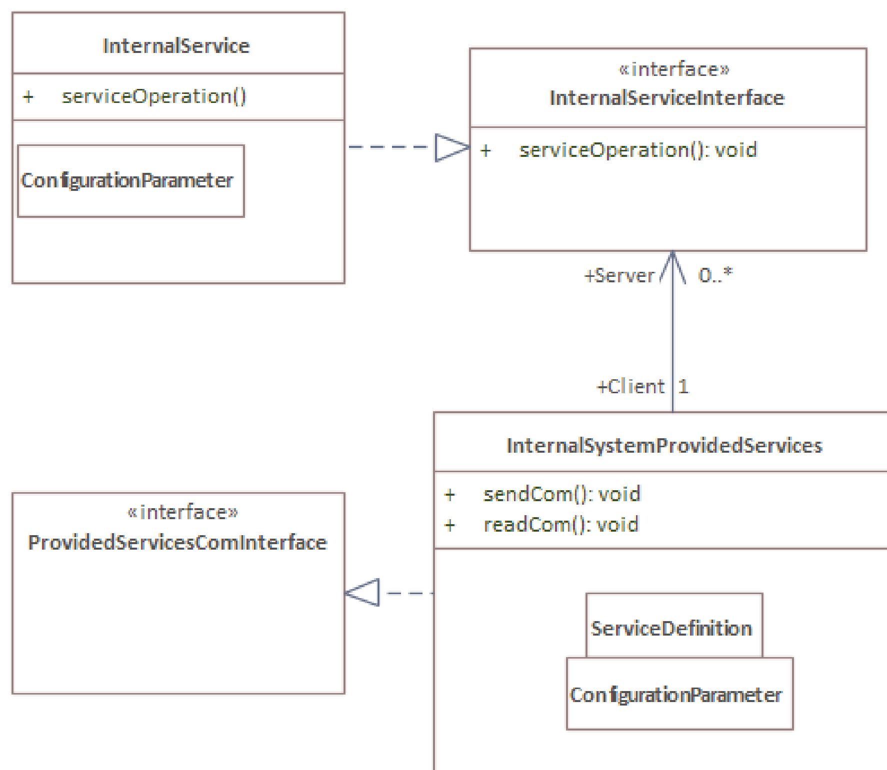

Figure 5.12: Internal services adapter pattern.

5.2.2 Solution Validation

A trade-off situation appears when considering deployment of the adapters to all ECU's. This will in some cases mean some execution of software does not add further value. This occurs when an identified service is exactly equal to what is sent from the external system, e.g. over CAN. A software element will then take the value of the signal, not modify the value, and define a callable service operation without making any modification. One could in that case simply use the value directly from the CAN-signal, without any involvement from the adapter. However, by always requiring to go to the adapter it decreases some complexity of the solution since any software engineer during development does not need to understand whether the value returned from a service operation of the adapter is equal to the value of a certain CAN-signal. When calling a service operation deployed on a different node, as shown in Figure 5.13, the engineer still needs to deduce if the proxy can use the adapter to get the correct return value. The implementation or documentation available for the service should be sufficient to determine this by the engineer. To summarize this example, if the reduction of the CAN-bus load and decreased complexity is higher prioritized than a slightly higher execution time, then the described strategy of deploying all adapters regardless if the adapter has any logic in its service definition, is more suitable.

By removing all mentions of CAN in the design, the solution is invariant to communication technology used. By this point, the functional and non-functional re-

quirements are deemed to be fulfilled in relation to the baseline design, described in Section 5.1.1. The motivation of fulfillment is described in Table 5.11.

Table 5.11: Requirements fulfillment motivation, 2nd cycle.

Requirement	**Fulfilled by design decision**	**Motivation**
The solution shall include a mechanism to dedicate internally provided services to specific external systems.	Define one wrapper per external system for provided services.	The design solution is enabled to dedicate one unique wrapper per external system which separates how services are provided to different external systems.
The solution shall be non-sensitive to method of communication between systems.	Remove references of CAN.	The design is now invariant of communication media.
Understandability of interface between systems	Add component diagram describing relations between systems.	The new, top view of systems show how the systems can invoke methods of the different adapters/wrappers which provide a way to understand the interfaces between systems.
Performance of CAN-bus	Add a described process to use proxies that can fetch data from existing CAN signals and not require new signals.	It is not strictly required to add new signals if the data exist on a CAN-bus already, which decreased the number of required signals.

Next, we will implement the candidate design solution and use it for further evaluation, focusing on how it may fulfill the elicited requirements.

5.2.3 Solution Implementation

We implemented the designs described in Section 5.2.1 in an intended context. First, an instance of externally provided services is shown in Figure 5.14, where a minor adaption has been made from iteration 1. Namely that the individual signals have been replaced with an interface, in order to be able to trace that interface to a provided interface in a component diagram. Similarly, a corresponding change has been done to an instance where the internal system provides services, shown in Figure 5.15. In this diagram, it can also be seen that the wrapper is meant to be dedicated to one specific external system. The services defined therein are then also available for that system, in this case a *ClimateAppSystem*.

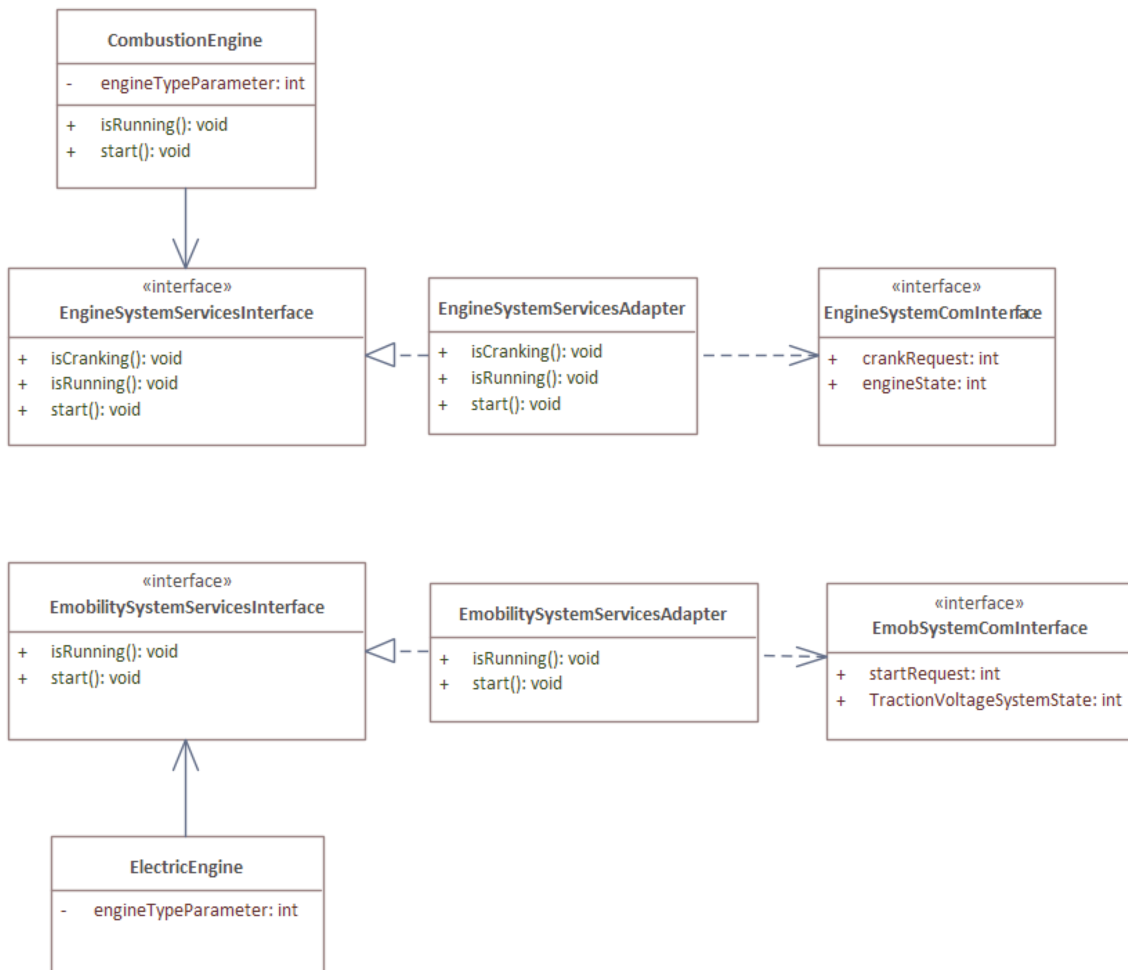

Figure 5.14: Implementation of externally provided services, 2nd cycle.

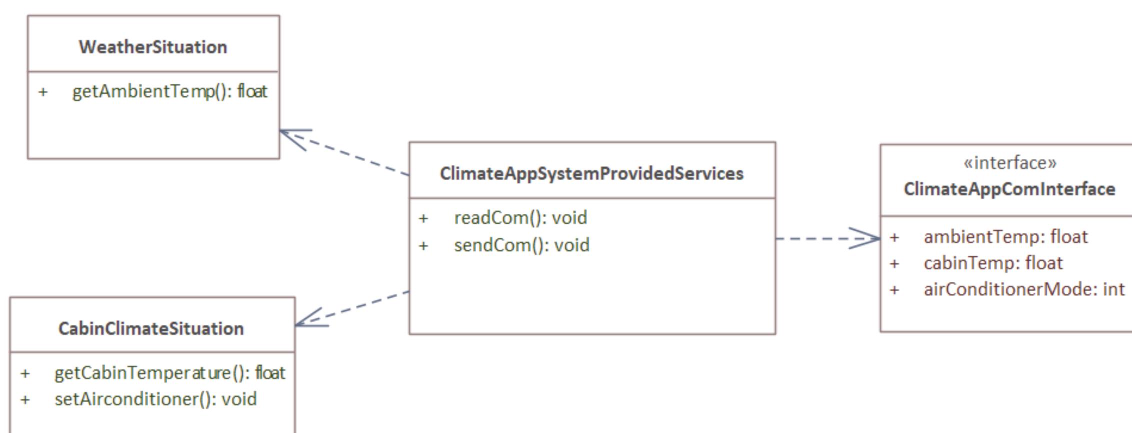

Figure 5.15: Implementation of internally provided services, 2nd cycle.

An implemented *system view* using a component diagram is illustrated in Figure 5.16. Here, several different relations between systems are described. For one case, where the external system is a climate app, the internal system is only acting

as the server to the app, being the client. This corresponds to that the internal system only provides an interface and is not using a provided interface of the external system. Here, the external system is deployed on a node that the internal system also is deployed on, meaning that the port does not correspond to a CAN-bus but some API. There is also a case of the reversed situation, where the internal system is only a client to an external system, being an Inertia Monitoring Unit. Lastly, a drive line system is both a client and server to the internal system. These systems then both provide and require an interface to one another. The communication is in this case done on several CAN-busses.

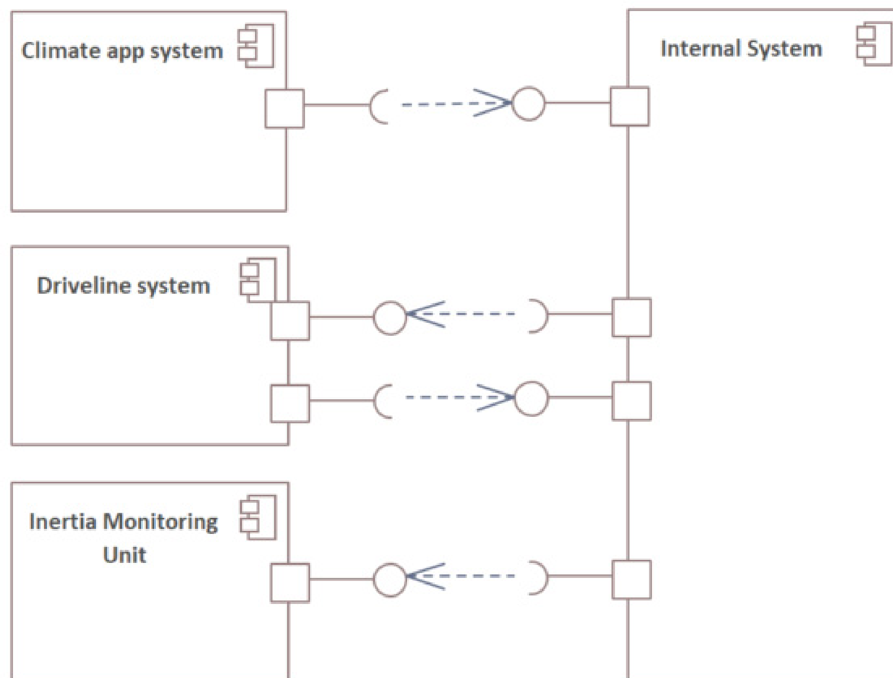

Figure 5.16: System view in component diagram.

Finally, an implemented situation of the strategy defined to reduce CAN-load is shown in Figure 5.17. For simplification, the separation by interfaces have been removed. It can be seen that in this case, when the service *Suspension* invokes the methods *getVehicleSpeed()* or *isBraking()*, the correct return value can be acquired by calling the adapter from the proxy of the service *Motion*, which is deployed on a different node, here *C1ECU*. To be able to deploy the adapter *DrivelineSystem-ServicesAdapter*, one must also be able to read the signals, here *vehicleSpeed* and *brakeState* on that node. This can be done e.g. by reading signals directly if the node is connected to the same bus, or routing the specific signals. However, the method *motionState()* is not found in the adapter so for that case it is necessary to call the service and not the adapter. Then, a new signal connecting the proxy to *Motion* would be required.

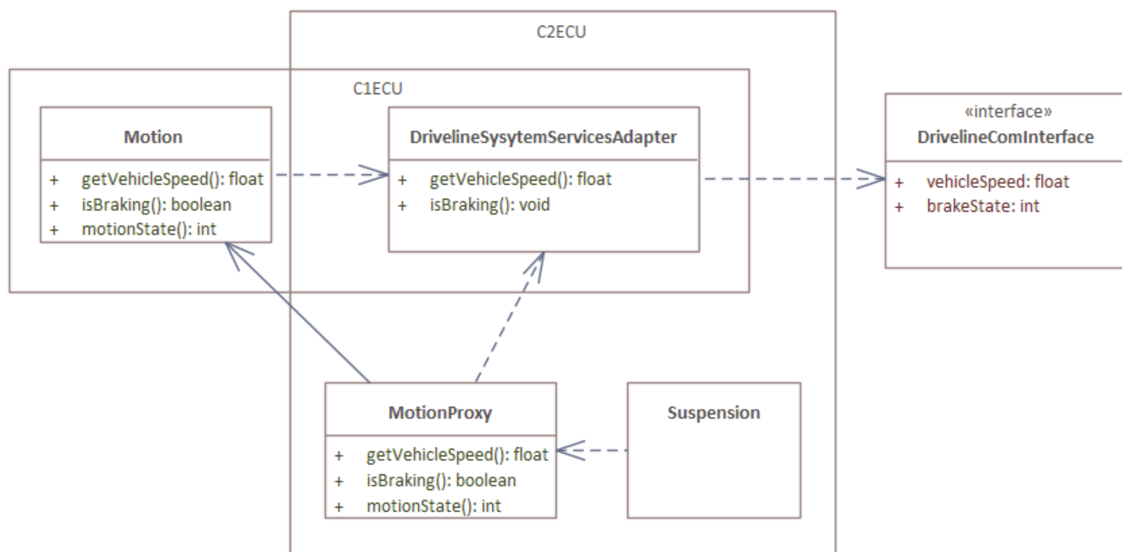

Figure 5.17: Implemented strategy of CAN-load reduction.

5.2.4 Implementation Evaluation

We conducted a focus group session to evaluate the implemented candidate design solution as well as sending out a post-session email with an attached questionnaire. As the design has changed from the first iteration, all stakeholder needs are used to evaluate the new candidate since some stakeholder needs that were deemed as fulfilled may have become less so by the new design. The questions focused on gathering data to answer if the design fulfills the stakeholder needs. To gather more data from the attendants, a post-session questionnaire was sent out which the attendants answered. The questions used can be read in Appendix A. Table 5.12 describes the satisfaction of the stakeholder needs with motivations.

Table 5.12: Stakeholder needs satisfaction, 2nd Cycle.

Stakeholder need	**Satisfied?**	**Motivation**
Reduce work of architecture definition	Yes	The solution should be able to give the architecture more robustness and protect it from the implementation of external systems.
Reduce work of code implementation	Yes	This result is an effect of the architecture being protected. The wrappers/adapters should also be easy to reuse.
Understand system	Yes/Probably	The interfacing between systems should be more clear.
Test system	No change w.r.t current solution	There is not any definite reason to say that the candidate would be more or less suitable to unit test.
Integrate different systems	Yes	The design provides complete strategy to integrate software systems which would make it easier to do so.

Use system in several products (lines)	Yes	The subjects assessment is that because of the increased reuse of architecture should also entail that it will be easier to use the software structure between products and product lines.
Reduce cost	Yes	Cost may be reduced when the architecture of the internal system is protected, and the established reference architecture can be used to a larger extent without being affected by how some external system is structured.
Reduce time-to-market	Yes	Same motivation as "Reduce cost".
Sufficient time behaviour	No change w.r.t current solution	The candidate would mean that more classes/elements are going to be executed in real-time, but that is something that would decrease the response time of the system.
Understandability of systems interfaces	Yes	The interfacing between systems should be more clear when specified in adapters or gateways.
Decrease bus load	No change w.r.t current solution	Even though the ownership of data moves from an external system to the internal, it should not mean that new CAN-signals are required.
Use solution using any communication media	Yes	The design is general and it should be possible to use for representing many forms of communication media, including internal node signals or CAN-signals.

We decide not to start a new iteration since all stakeholder needs are fulfilled or regarded as equally satisfactory as the current way of working. It would of course be possible to attempt finding additional stakeholder needs not discovered or improving the satisfaction of the ones found. However, the work must be closed sometime and at this time there does not appear to be any strong motivation to proceed with a new iteration.

Chapter 6

Discussion

6.1 Research Questions

RQ1: What is an effective way to make a Service-Oriented embedded system adapt at compile-time using parametrization to a specific configuration of external systems?

The qualitative data gathered indicates that the effects of variability within a software product line concerning how different external systems are employed can be managed using adapters or an adaptive layer. This mechanism restrains e.g., ripple effects do not affect the internal system in a negative manner, such as decreasing commonalities of the implementation of a service object. One can apply parametrization both in the adaptive classes (in an object-oriented programming style) and the internal system to manage different forms of variability. Parameter configuration of the internal service objects should be used only when the behaviour of the internal system should vary in the product line. How the configuration using parameters allocated to either the internal systems service objects or the adapters should be performed may become difficult to firmly establish. In an object-oriented setting, the variability should be handled in the object that encapsulates the entity that varies.

RQ2: What is an effective software design that can be used to integrate identified services of external systems into a Service-Oriented embedded system?

Using adapters for performing client requests of an external system and gateways for the internal system to respond to client requests, as a server, may be used to integrate a system with external systems. It is our opinion, formulated after conducting this study, that this can provide an “abstraction-shield” of the internal system and let that software development organization concern only about its responsibilities, not about the responsibilities or behaviour of an external system. This is currently the case at the company being studied, where the external systems are attempted to be integrated as they were a part of the internal organization. The finding of this study argues that the current approach is less than ideal, as it can lead to problems when developing the internal system. It is possible that it is the programmer’s mindset that all systems of some product should function as tightly connected as if they would have been developed by one single organisation, which has led to using the current architecture. The introduction of a “system-view” by a component diagram establishes a different way of considering the system-of-systems, where the system

developed by the internal organisation is merely one part of a larger system. One possible drawback that this can entail may be that any organisation is reluctant to assume complete product responsibility, as it would become more clearly composed by many different systems. On the other hand, the "system view" can also provide clearer insights into the division of responsibilities between systems and/or organisations. However, the positive effects that have been described are also relying on the patterns being implemented correctly. It is possible that developers implement more functionality than intended in these adaptive classes or layers which can defeat their purpose. If the designs are used, it should be made clear to everyone how they should be applied, and especially how not to use them. To this end, it should be sufficiently communicated to developers using the design how to apply it and what the purpose of each element is. During the design science cycle, the meaning of the so-called "service definition" of an adapter or gateway was often discussed. What should constitute a service of an external system and what functionality may one allocate to the service definition? This can be up to the developers using the designs to decide, e.g. if to place only translations of signals to remove the direct dependency of communication media or more comprehensive logic.

To define the main insight gathered from this study, the two research questions should be considered together. We have shown that it is possible to facilitate the integration of external systems in a software product line, where the use and/or installation of external systems varies throughout the line, by using a wrapping technique to provide a suitable abstraction of defined external services. Instead of integrating the architecture of different subsystems, which may entail modeling the external systems as a part of the internal, it can be beneficial to only consider the internal system as one individual entity. This system then collaborates by using and/or being used by external systems to fulfill the complete product that is regarded as a system-of-systems. This viewpoint of regarding the complete product as a set of (sub) systems can be compared with the methodology of *component based software engineering* [35], which focuses on how to integrate different separate components to obtain one working product. However, the Service-Orientation aspect has not been that closely described in the literature, especially in combination with variability [35]. Parizi [29] describes how different developers are moving away from CBSE in favour of SOA to facilitate interoperability between systems. However, in this study, we found how difficult that transition may be to do when some of the systems do not apply the principles of SOA. In [14], a so-called *integration architecture* is described how different components (or systems) are integrated to fulfill the full requirements of a software product. This can be regarded as similar as the "systems view" defined in this study, e.g. in Figure 5.10.

The software architects at the company expressed the difficulties that arose as a consequence of not being able to control the interface or architecture of various external systems. This is also described in [30], which explains that there may be problems with integrating COTS when one wishes to apply certain architectural patterns. This should be able to be somewhat remedied by the design shown in this study, i.e. using wrappers to handle the format of the interface, e.g. abstraction, and a system view to integrate different systems (or components).

6.2 Validity Threats

6.2.1 Construct Validity

To assess whether the design solution was deemed "effective", focus groups were held that discussed the candidate design and its implementation as well as a follow up questionnaire was provided where the attendants responded to questions regarding satisfaction of stakeholder needs. This can be read in Appendix A. The questions were formulated to not be leading so that the responses would not be guided by what the respondents imagined what the author wished. However, since the questions were also formulated so that it could be confirmed that the design would be better than the baseline design in a very straightforward, close-ended way, the risk still exists. The respondents also are affected by a load in their daily work that may lead to the responses would be given in order to avoid starting a new iteration, thereby leading to new focus group sessions or interviews. Since the scope of this study only includes the creation of a design candidate, the long term effects of applying the design is not completely known. Ideally, the effects after implementing the proposal should be studied.

6.2.2 Internal Validity

The internal validity of the results heavily relies on that the subjects involved in the implementation validation could estimate that the new design would perform better or worse than the baseline design in satisfying the stakeholder needs. For some respondents, it is possible that the designs would be difficult to evaluate with only the UML-diagrams at hand. However, this should have been mitigated by the follow-up conversation that was held, which presented another opportunity for the respondents to ask questions. All attendees of focus groups and interviews are however experienced in desiging software in UML as well as defining the implementation in code. This gives some confidence that the designs can be evaluated without implementation and evaluation thereafter. It should also be noted that I, the author of the study, am an employee at the company used to perform the study. It is difficult to estimate the effect that this had on the result, and on the responses from subjects. One can speculate that answers were given that my colleagues imagined being desired by myself. However, it is also possible that the already established relationship may have provided increasingly valid responses, depending on the person. The problem described had previously been discussed in a more informal manner and some possible solutions had been considered. The starting design from the first iteration was however picked from available literature regarding service identification that integrate different systems, and later improved using the design science methodology. Even though quite an extensive literature review was done prior to formulating the research questions, to find a research gap, it is of course possible that there exists additional non-reviewed literature. The search strings used may have missed important synonyms that would have resulted in finding new data that could have steered the design in a different direction.

6.2.3 External Validity

The external validity of the results is to some degree threatened by that the sample of subjects that contributed to the design science process may not be representative of software developers in general. However, it is likely that it is representative of the vehicle or embedded software industry as the products share important attributes. In addition, the people selected for the focus group sessions and/or interviews were of different backgrounds, including different ages, experience, and specializations of working focused on coding or architecture. For organisations that develop software systems that consist of many sub-systems produced by different organisations, the results should be increasingly valid. Early in the process during the execution of this study, it was delimited to using CAN-based communication media. Later, this dependency was removed in favour of general communication, as one requirement was that the design should hold for internal software signals as well.

6.2.4 Reliability

The study conducted has some reliability concerns. When a methodology as general as design science is applied at a specific company and its employees, the replicability will likely be affected poorly, also affecting the reliability. The responses of the focus group attendees and interviewees are dependent on past experiences, which are specific to this workplace to some degree. The resulting design has been deemed to be “effective” by the subjects. However, the definition of “effective” translates to satisfying the needs of various stakeholders at the company where the study was conducted. Different stakeholders and people would likely entail that the definition of effectiveness also changes. As an example, during the design phase, some trade-offs had to be assessed to choose the most suitable design. However, which choice is most suitable is clearly dependent on the specific situation and past experiences of the respondents of the study. At a different company, it is almost certain that other design decisions would have been made. However, it is still possible that other organisations that develop similar system-of-system software product lines using Service Orientation are experiencing issues of the same nature, and executing the same methodology there would yield a similar result. The different design patterns at each cycle were evaluated and improved using an open process, with low formality and clear guidelines. This consisted of general discussions regarding potential advantages and drawbacks to the baseline, and comparing alternatives. Using a strict architecture design process could also have resulted in a different design proposal.

Using a more standardized methodology to evaluate and compare different design solutions, e.g. using the *Architecture Tradeoff Analysis Method* (ATAM), would perhaps entail that a different result was obtained. However, the less formal methodology applied here and ATAM both use a stakeholder- and quality attribute focus to compare architectures, along with sensitivity and trade-off analysis. It is therefore likely that the potential result of using e.g. ATAM would be similar.

6.3 Conclusions and Recommendations

In this thesis, we have described the problematic questions that can arise when developing a software system that is required to interact and collaborate in different ways with other software systems. Together, this system-of-system composes a full software product. The compatibility may be poor when integrating different systems which do not necessarily follow the same architectural principles or reference architecture, such as a Service-oriented architecture style. These problems are also amplified when considering the development of a software product line, where different systems may be replaced, added, or removed for specific variants. For an organization that develops one of the systems in such an environment, a strategy for handling its so-called "external systems" can be beneficial. We have defined an external system as some system that the "internal" organization cannot fully control. For example, the API to different systems of suppliers is often defined at purchase and cannot be changed upon request. This also entails that the responsibilities between systems are also often pre-defined.

We have shown that the integration of a Service-oriented system that uses CAN-based communication media with different external systems can be handled by taking influence from how the integration of legacy devices and systems has been solved by "service identification" strategies. Such studies have previously been conducted when reusing legacy systems or devices in a Service-Oriented system. Often, the solution is comprised of using a wrapper or abstraction layers to make the device or system compatible. However, these methods relied on the existence of Ethernet-based communication, which is usually much more high-capacity than CAN. A CAN is also configured manually, by specifying the signals between systems at design time. This leads to the self-adapting mechanism often used in SOA, a broker design, is not applicable. However, the compatibility benefits of the solutions described in these studies can still be obtained by following a similar design.

The result of this study shows how a wrapper-based design of providing services of internal as well as invoking services of external systems can be applied. For providing services to other systems, several gateways can be implemented that dedicate specific services of the internal system to specific systems in order to handle the interfacing between these systems in an understandable way. When requesting services from external systems, an adapter-like approach was used. This provides the ability to, for instance, control the abstraction of the external functionality requested as well as remove the dependency on communication media used in that context, making the internal system be less dependent on the environment.

As the software design was not implemented in actual code, only modeled as instances using UML, it is difficult to determine how much the described benefits would be realizable. For example, there is a possibility that developers would reject the ideas since it would lead to more classes, and also possibly more code having to be written, to the benefit of e.g. reuse of code and reused software structure.

The lessons learned from the company and current situation is that it will be difficult

to approach the integration of external systems with the internal system like they are developed closely and as parts of the same system. Instead, one should have a "buffer zone" to protect the architecture of the internal system and not have to make large changes due to an external system. The gateways and adapters constitute this "buffer zone". As the interfaces between systems may change over a product line, and therefore also the responsibilities, when developing the software of the internal system, it may be beneficial to let the internal services incorporate responsibilities of external systems. This can lead to that the software architecture of the internal system is less affected by changes of external systems and less different between specific variants compared to using the current approach.

The basis of the research questions formed was that the CAN-based communication medium is so different from Ethernet that a different integration and adaption strategy is required. However, regarding future work, it may be worth investigating how one can attain a similar self-adaptive, plug-and-play behaviour even when using a CAN of low capacity. Can a service broker be responsible for brokering services on a CAN? This may be possible if, for instance, a more effective signaling strategy is employed. The resultant design of this study is likely largely dependent on the company and software developers working there, which is why one would also be interested in seeing how the result can differ if the same study is performed at a different company, or in a different industry.

References

[1] M. Abdellatif, A. Shatnawi, H. Mili, N. Moha, G. E. Boussaidi, G. Hecht, J. Privat, and Y.-G. Guéhéneuc, “A taxonomy of service identification approaches for legacy software systems modernization,” *The Journal of systems and software*, vol. 173, p. 110868, 2021.

[2] R. Adler, A. Bennaceur, S. Burton, A. Di Salle, N. Nostro, R. L. Olsen, S. Saidi, P. Schleiss, D. Schneider, and H.-P. Schwefel, *Service-Oriented Reconfiguration in Systems of Systems Assured by Dynamic Modular Safety Cases*, ser. Communications in Computer and Information Science. Switzerland: Springer International Publishing AG, 2021, vol. 1462, pp. 12–29.

[3] S. L. Bain, *The Design Patterns Companion.* Chicago: Project Management Institute, 2020.

[4] T. Berger, J.-P. Steghöfer, T. Ziadi, J. Robin, J. Martinez, I. för data-och informationsteknik (GU), IT-fakulteten, G. universitet, G. University, D. of Computer Science, E. (GU), and I. Faculty, “The state of adoption and the challenges of systematic variability management in industry,” *Empirical software engineering : an international journal*, vol. 25, no. 3, pp. 1755–1797, 2020.

[5] V. Braun and V. Clarke, “Using thematic analysis in psychology,” *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[6] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana, “A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures,” *The Journal of systems and software*, vol. 81, no. 4, pp. 463–480, 2008.

[7] O. Carlsson, J. Delsing, F. Arrigucci, A. W. Colombo, T. Bangemann, and P. Nappey, “Migration of industrial process control systems to service-oriented architectures,” *International journal of computer integrated manufacturing*, vol. 31, no. 2, pp. 175–198, 2018.

[8] F. Chen, S. Li, H. Yang, C.-H. Wang, and W. Cheng-Chung Chu, “Feature analysis for service-oriented reengineering,” in *12th Asia-Pacific Software Engineering Conference.* IEEE, 2005, p. 8 pp.

[9] M. Di Natale and A. L. Sangiovanni-Vincentelli, “Moving from federated to integrated architectures in automotive: The role of standards, methods and tools,” *Proceedings of the IEEE*, vol. 98, no. 4, pp. 603–620, 2010.

[10] E. Engström, M.-A. Storey, P. Runeson, M. Höst, and M. T. Baldassarre, “How software engineering research aligns with design science: a review,” *Empirical*

software engineering : an international journal, vol. 25, no. 4, pp. 2630–2660, 2020.

[11] S. Feldhorst, S. Libert, M. ten Hompel, and H. Krumm, "Integration of a legacy automation system into a soa for devices," in *2009 IEEE Conference on Emerging Technologies Factory Automation*. IEEE, 2009, pp. 1–8.

[12] S. Garcia, D. Struber, D. Brugali, A. Di Fava, P. Pelliccione, and T. Berger, "Software variability in service robotics," *Empirical software engineering : an international journal*, vol. 28, no. 2, pp. 24–24, 2023;2022;.

[13] O. Givehchi, K. Landsdorf, P. Simoens, and A. W. Colombo, "Interoperability for industrial cyber-physical systems: An approach for legacy systems," *IEEE transactions on industrial informatics*, vol. 13, no. 6, pp. 3370–3378, 2017.

[14] S. Kiv, Y. Wautelet, and M. Kolp, *Agent-Driven Integration Architecture for Component-Based Software Development*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 121–147. [Online]. Available: https://doi.org/10.1007/978-3-642-34645-3_6

[15] J. Kontio, J. Bragge, and L. Lehtola, *The Focus Group Method as an Empirical Tool in Software Engineering*, ser. Guide to Advanced Empirical Software Engineering. London: Springer London, 2008, pp. 93–116.

[16] S. Kugele, P. Obergfell, M. Broy, O. Creighton, M. Traub, and W. Hopfensitz, "On service-orientation for automotive software," in *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2017, pp. 193–202.

[17] S. Kugele, P. Obergfell, and E. Sax, "Model-based resource analysis and synthesis of service-oriented automotive software architectures," *Software and systems modeling*, vol. 20, no. 6, pp. 1945–1975, 2021.

[18] I. Lee, J. Y. Leung, and S. H. Son, *Handbook of real-time and embedded systems*. Boca Raton: Chapman Hall/CRC, 2008.

[19] J. Lee and G. Kotonya, "Combining service-orientation with product line engineering," *IEEE software*, vol. 27, no. 3, pp. 35–41, 2010.

[20] F. v. d. Linden, K. Schmid, and E. Rommes, *Software product lines in action: the best industrial practice in product line engineering*, 1st ed. New York;Berlin;: Springer, 2007.

[21] J. M. Mendes, P. Leitao, A. W. Colombo, and F. Restivo, "Service-oriented control architecture for reconfigurable production systems," in *2008 6th IEEE International Conference on Industrial Informatics*. IEEE, 2008, pp. 744–749.

[22] B. Mohabbati, M. Asadi, D. Gašević, M. Hatala, and H. A. Müller, "Combining service-orientation and software product line engineering: A systematic mapping study," *Information and software technology*, vol. 55, no. 11, pp. 1845–1859, 2013.

[23] B. Mohabbati, M. Hatala, D. Gašević, M. Asadi, and M. Bošković, "Development and configuration of service-oriented systems families," in *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 2011, pp. 1606–1613.

[24] P. Nappey, C. El Kaed, A. W. Colombo, J. Eliasson, A. Kruglyak, R. Kyusakov, C. Hubner, T. Bangemann, and O. Carlsson, "Migration of a legacy plant lubrication system to soa," in *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2013, pp. 7440–7445.

[25] S. Oloruntoba, "Solid: The first 5 principles of object oriented design," Nov 2021. [Online]. Available: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

[26] F. Oszwald, P. Obergfell, B. Liu, V. Pazmino Betancourt, and J. Becker, "Model-based design of service-oriented architectures for reliable dynamic reconfiguration," *SAE International Journal of Advances and Current Practices in Mobility*, vol. 2, no. 5, pp. 2938–2947, apr 2020. [Online]. Available: https://doi.org/10.4271/2020-01-1364

[27] A. L. Osório, L. M. Camarinha-Matos, T. Dias, C. Gonçalves, and J. Tavares, *Open and Collaborative Micro Services in Digital Transformation*, ser. IFIP Advances in Information and Communication Technology. Cham: Springer International Publishing, 2021, vol. 629, pp. 393–402.

[28] M. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. Krämer, "Service-oriented computing: A research roadmap," *In: Curbera, Francisco (ed.); Krämer, Bernd J. (ed.); Papazoglou, Mike P. (ed.): Service Oriented Computing, pp. 1-29*, vol. 17, 06 2008.

[29] R. M. Parizi, "Microservices as an evolutionary architecture of component-based development: A think-aloud study," *ArXiv*, vol. abs/1805.11757, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:44096875

[30] K. Pohl and A. Reuys, *Considering Variabilities during Component Selection in Product Family Development*, ser. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 22–37.

[31] P. Runeson, E. Engström, and M.-A. Storey, *The Design Science Paradigm as a Frame for Empirical Software Engineering*, ser. Contemporary Empirical Methods in Software Engineering. Springer International Publishing, 08 2020, pp. 127–147.

[32] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering : an international journal*, vol. 14, no. 2, pp. 131–164, 2009.

[33] H. Shokry and M. Ali Babar, "Dynamic software product line architectures using service-based computing for automotive systems." in *Software Product Lines Conference*, 01 2008, pp. 53–58.

[34] N. Taušan, J. Markkula, P. Kuvaja, and M. Oivo, "Choreography in the embedded systems domain: A systematic literature review," *Information and software technology*, vol. 91, pp. 82–101, 2017.

[35] T. Vale, I. Crnkovic, E. S. de Almeida, P. A. d. M. Silveira Neto, Y. C. Cavalcanti, and S. R. d. L. Meira, "Twenty-eight years of component-based software engineering," *The Journal of systems and software*, vol. 111, pp. 128–148, 2016.

[36] M. Wagner, A. Meroth, and D. Zöbel, "Developing self-adaptive automotive systems: On the integration of service-orientation into automotive development processes," *Design automation for embedded systems*, vol. 18, no. 3-4, pp. 199–221, 2014.

[37] M. Wagner, D. Zobel, and A. Meroth, "Towards runtime adaptation in autosar: Adding service-orientation to automotive software architecture," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA).* IEEE, 2014, pp. 1–7.

[38] R. Wieringa, "Design science as nested problem solving," in *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology.* ACM, 2009, pp. 1–12.

[39] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*, 2014th ed. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2014.

Appendix A

2nd Cycle Implementation Evaluation Questionnaire

For all questions, please provide a short answer (yes/no/equal) and a motivation.

Do you think that using the proposed solution, in relation to the current way:

1. Would lead to increased reuse of software structures? (architecture)
2. Would lead to further reuse of code?
3. Would require less effort to understand the complete system(-of-systems) and how it functions?
4. Is more suitable to unit-test?
5. Require less effort(work) to add, replace or remove external systems?
6. Require less can traffic?
7. Require less effort to understand the interface between systems in terms of individual signals?
8. Lead to higher reliability/availability, (e.g. by decreasing down-time)?
9. Do you think the proposal would increase or decrease performance (execution, response time)?

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden