Observing software architectural gaps in industry: A case study

Finding and observing gaps between the implemented and interpreted software architecture of a system used in industry

Master's thesis in Computer science and engineering

David Benjaminsson
Edvin Bengtsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Observing software architectural gaps in industry: A case study

Finding and observing gaps between the implemented and interpreted software architecture of a system used in industry

David Benjaminsson
Edvin Bengtsson

UNIVERSITY OF GOTHENBURG

CHALMERS UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Observing software architectural gaps in industry: A case study
Finding and observing gaps between the implemented and interpreted software architecture of a system used in industry
David Benjaminsson
Edvin Bengtsson

Supervisor: Truong Ho-Quang, Computer Science and Engineering
Advisors: Martin Henriksson, Kvaser AB
& Daniel Berglund, Kvaser AB
Examiner: Robert Feldt, Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Observing software architectural gaps in industry: A case study

Finding and observing gaps between the implemented and interpreted software architecture of a system used in industry

David Benjaminsson & Edvin Bengtsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The software architecture is an important part of any software system, especially as today's software systems become increasingly complex. A key objective of having a software architecture is to encourage a united perception regarding the structure of a system. However, this is no easy task and differences may arise between intention, implementation, and individual workers' interpretation. These different perceptions of a system's architecture may cause a threat to the system's quality. For example, new features and solutions based on a developers perception of the architecture might violate the intended architecture. These violations may in turn cause a threat to system quality attributes such as performance, testability, modularity, etc [1], [2].

This thesis aims to discover and analyze gaps between the implemented architecture and the architecture interpreted by the developers of a software system used in industry. In order to be able to identify such gaps, both the implemented and interpreted architecture first had to be extracted. The implemented architecture was extracted by reverse-engineering, and the interpreted architecture was gathered through interviews with developers working on the system. Several types of gaps were identified while comparing the two architectural views to each other. The second stage of the case study was to gather insights from the developers regarding the causes and effects of the found gaps, establishing how their interpretation and the implementation came to differ and the effects thereof.

The study resulted in the discovery of a selection of gaps, such as missing modules or missing communication between modules. Lack of high-level documentation and lost architectural knowledge were two of the identified causes for gaps to arise, resulting in effects such as increased implementation time and higher coupling.

The system investigated in this study is called Minihydra and is used as firmware in a selection of products produced by the Swedish company Kvaser.

Keywords: software architecture, architecture interpretation, software architectural gaps, architecture erosion, software decay, reverse-engineering

Acknowledgements

Big thanks goes to Truong Ho-Quang, our academic supervisor, as well as Martin Henriksson and Daniel Berglund, our supervisors at Kvaser.

Truong Ho-Quang has given us lots of advice and great feedback throughout the thesis work, from start to finish. Martin Henriksson and Daniel Berglund went above and beyond to ensure that we could collect all the information needed from Kvaser, and were very generous with their time and effort. We would also like to thank everyone at Kvaser who agreed to let us interview them and who shared their interpretation of the system's architecture.

We would also like to thank our friends and families for supporting us throughout our education.

David Benjaminsson & Edvin Bengtsson, Gothenburg, September 2021

Contents

List of Figures

1 Introduction

Software architecture is like a blueprint for a building, specifying what goes where and how different parts of a system interact. Every software system has an architecture, it may not be documented, but there is always an architecture to every software system [5], [6]. If the software architecture is documented, that document is a valuable asset of the software system [5], [7]. Building a software system based on a well-planned architecture will allow the system to adapt more easily to changes in requirements [8], [9].

Developing a new software system would in a perfect world start with gathering requirements to form the basis for the creation of an architecture for the system. Based on the architecture, flawless code would then be written from start to finish. In this ideal situation, the outcome of every step in the process would be perfect, and the software system would fulfill all the requirements and satisfy the customer immediately once completed. Anyone who has ever dipped their toes into software development, however, most likely has a different experience of the development process.

In reality, the path to the complete software system is not as straightforward as previously described. Changing requirements and several other factors can quickly make the architecture of a system outdated, and since working code is often prioritized over comprehensive documentation, it is easy to see how the documentation falls behind and gets outdated over time [10]. The initial software architecture is the mutual idea among the developers of how the software should be constructed based on the initial requirements. If the requirements change but the architectural documentation is not updated, the documented architecture will no longer reflect the architecture implemented during development. In this scenario, the developers will instead follow their interpretation of the intended architecture, their interpreted architecture. This thesis aims to study the differences between the implemented architecture and the interpreted architecture.

Once this scenario has played out, it will be hard to acquire a new documented version of the intended architecture. Asking the developers for their knowledge of the system's architecture can only result in their interpreted architecture, thus likely to be influenced by the current implementation. Likewise, reverse-engineering the source code to extract the architecture will only result in the implemented architecture. The intended architecture is how the system is supposed to be built, implemented is how it is built, and interpreted is most probably a mix between the two, the architecture interpreted by a developer.

The phenomena that lead to differences between implemented and intended architecture is sometimes referred to as *software architecture erosion*. More specifically, software architecture erosion is when the documented architecture is no longer reflecting the system's implementation and/or its requirements [11]. In contrast to software architecture erosion, this thesis has not compared the implemented architecture and the intended architecture, but rather compare the implemented architecture and the interpreted architecture. Any differences between the implemented and interpreted architecture will from this point on be referred to as software architectural gaps or just gaps.

So far, four different versions of software architecture have been mentioned, and to avoid confusion a clarification of wording may be needed. The four versions previously mentioned are:

- **Initial software architecture** - The software architecture agreed upon and aimed for at the moment a software system proceeds into the development phase.
- **Intended software architecture** - The software architecture that is currently aimed for. The intended architecture may change and evolve, for example, in order to meet new requirements.
- **Implemented software architecture** - The software architecture realized by the source code of the system.
- **Interpreted software architecture** - The software architecture according to a developer. The interpreted architecture is most likely a mix of the intended and the implemented architecture, as a developer is exposed to both during development.

Sticking to the analogy of software architecture being what blueprints are for a building, the initial architecture is the architecture that is used when the construction of the building starts. The architect then decides to move the stairs half a meter and gives the builder the new intended blueprints slightly different from the initial. Soon after the windows arrive and turn out to be ordered to the wrong size. Since it would take a too long time to order new windows, the builder decides to install the wrong sized windows, and so the house does no longer matches the intended blueprints. A family moves into the home and decides to take down an interior wall to open up between the kitchen and living room. Afterward, the builder is asked to reconstruct the blueprints for the house. Since he remembers receiving the wrong sized windows but does not know that a wall is missing, he believes that the windows are the only thing differentiating the house from the intended architecture. The initial, intended, implemented and interpreted architecture are now all varying from each other.

In this thesis, the gaps between the implemented architecture and the interpreted architecture have been investigated. The word gap is used to reference a specific inconsistency between two compared architectural descriptions. This means that a gap could, for example, be a relation or a module present only in the implemented architecture but lacking in the interpreted, or that a module has different responsibilities in the two architectures.

In order to analyze gaps, they first have to be identified. Identifying gaps requires a comparison between the implemented and interpreted architecture, both of which first have to be extracted from their respective sources. The implemented architecture was extracted by reverse-engineering the source code of the system, while the interpreted architecture was extracted by conducting interviews with developers at the company. The gaps between the two architectures are then identified via mapping the two architectural descriptions against each other, revealing the gaps as differences between them. The findings from the comparison form the basis for a second round of interviews with the developers in search of insights regarding possible causes and effects of any discovered gaps.

1.1 Objective

This thesis had two primary objectives. The first objective was to discover software architectural gaps in a software system that is currently used in industry. This was done by reverse-engineering the software architecture of a system's implementation and comparing it to the developer interpreted software architecture of the same system. The second objective was to use any discovered gaps as points of discussion with the developers of the system in order to gain possible insights about the gaps' causes and effects.

The software system examined in this study is called Minihydra and is developed and used by the Swedish company Kvaser as firmware in a selection of their products. The products using the Minihydra system are all CAN-interfaces (Controller Area Network) but have differences when it comes to both hardware and software. In order to narrow the scope into something feasible for this thesis, the software of one specific product was chosen to be examined.

1.2 Scientific contribution

While there have been several studies addressing the phenomena of software architecture erosion, papers comparing implemented architecture to interpreted architecture are more scarce. Finding gaps between the implemented and intended architectures merely requires a comparison between pre-existing artifacts. However, this thesis aimed to provide new insights by comparing the developers understanding of the architecture to the implemented architecture. This was done by working in close collaboration with the developers to reproduce their view of the system's architecture and comparing it to the architecture extracted from the source code of the system.

Studies conducted regarding software architectural erosion have primarily been examining either fictive, non-commercial, or open-source software [12], [13], [14]. This study contributed to insights regarding software architectural gaps on a system that is currently being used in industry and investigated their potential causes and effects.

These insights can be used to highlight the potential effects that a software system with software architectural gaps is subjected to, as well as identifying potential causes of the gaps to help repair or prevent them. Furthermore, the outcomes of this study might help to identify future research topics regarding software architecture interpretation and software architectural gaps.

1.3 Structure of the paper

The structure of this paper is as follows. In Chapter 2, background and theory are presented, providing an introduction to the field and the underlying concepts. Chapter 3 is comprised of earlier research related to the study and in Chapter 4 the research questions of the performed study are outlined. In Chapter 5, the study's research methodology is presented followed by Chapter 6 where the results of the study can be found. Chapter 7 is a discussion chapter, touching on subjects such as reverse-engineering and threats to validity. Finally, the conclusion of the study is found in Chapter 8.

2

Background and theory

This chapter presents essential concepts and theory needed in order to fully grasp the scope and context of this thesis. Section 2.1 is an introduction to the concept of software architecture, Section 2.1 presents the definition of a gap used in this thesis, Section 2.3 presents the system analyzed as well as the company behind it, and finally, Section 2.4 introduces the C4-model.

2.1 Software architecture

Software architecture is like a blueprint for a software system, specifying what components are to be used and the relationships between them. The architecture can be seen as a bridge between the business goals for the software system and the system itself, as a middle step where abstract business goals are boiled down into tangible tasks forming a concrete system description [5].

There are several definitions to what software architecture is, and in this thesis, the following definition given by Bass, Clements, and Kazman will be used "The software architecture of a system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both." [5, p. 18].

An important remark is that each and every software system has a software architecture. The level of documentation of said architecture can however vary a lot. Some projects may have hundreds of pages describing the system all the way from system context and design overview down to each individual function, while other architectural descriptions are written down on a single post-it note or can even be completely absent [5], [1].

When developing software using agile methods less software architecture is created upfront compared to when, for example, using the waterfall model. However, creating an architecture before the implementation phase starts is still a key step in the development process. As previously mentioned, all software systems will have an architecture whether it is planned for or not, and making a few wise design decisions early on in the process is often beneficial as big changes can be expensive later on [15].

2.2 Software architectural gaps

Software architectural gaps are in this thesis defined as differences between how a developer interprets the architecture in use, i.e. how they view the system, and the implemented software architecture of the system, i.e. how the system is implemented. A gap is one single mismatch between these two architectures, for example, that two separate tasks pass messages between each other and thus have a relationship between them, without the developer knowing that they are communicating. In this scenario, the two tasks are connected to each other, and running one without the other could lead to undefined behavior, but as far as the developer knows the tasks are independent and believes that running one without the other should be fine.

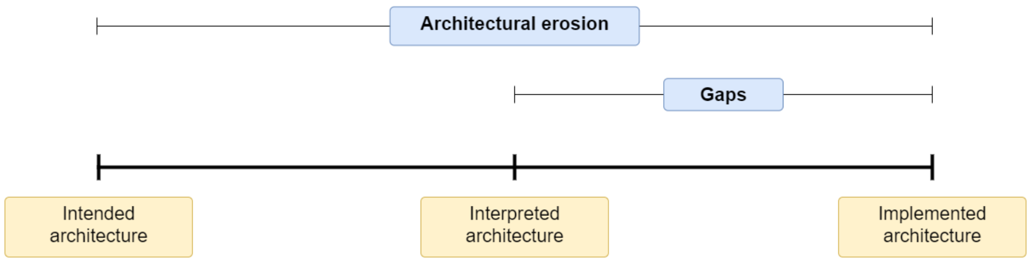

Figure 2.1: A gap can be discovered when comparing an implemented architecture with an interpreted architecture. Software architecture erosion is the overall difference between an implemented and an intended architecture.

To avoid any misunderstandings there may be a need to clarify the difference between a gap, as the word is defined in this thesis, and software architectural erosion [16]. Software architecture erosion is measured between the implemented and the intended architecture, while a gap in this study is defined as a specific difference between the implemented and interpreted architecture. The term software architectural erosion will be described further in Section 3.1. As illustrated in Figure 2.1, the interpreted architecture is likely placed somewhere in between the implemented and intended architecture. The interpreted architecture is the developers mental model of the system's architecture, and can be affected and altered in multiple ways, for example:

- By the intended architecture - Reading about or getting told by the architect about the intended architecture. It does not matter if it is text or speech, the words will be interpreted by the developer, whose interpretation may differ from the architects, and thus from the intended architecture.

- By the implemented architecture - While working on the software system the developer will interpret the architecture through code, affecting his[1] own mental model of the software architecture.

- By discussions between developers - If learning about the architecture from a colleague or comments in the code one should be aware that what is learned

[1]For simplicity, the pronouns he/him/his will be used when referring to a developer or interviewee in this thesis.

is the interpretation of an interpretation, and everyone who has played the Chinese whisper game as a kid knows that the starting sentence is seldom the same as the final one.

- By ignoring - It may be enough for the developer to know some parts of the systems architecture, allowing the developer to ignore other parts of the system's architecture and exclude them in his interpreted architectural model.

Recreating a lost or outdated intended architecture from the interpreted architecture is likely not possible. The interpretation will be affected by both how the system was designed when there was an intended architecture and also by the current implementation and other factors as previously described. The interpretations are likely to result in a less optimized architecture than an intended architecture would have been, and important design decisions taken in order to realize the intended architecture will likely be lost. Changes in requirements can also have affected the intended and implemented architecture in different ways, but if the intended architecture is absent the developer will interpret only the implemented architecture, and hence the differences between the intended and interpreted architecture will grow. Even just after the intended architecture has been interpreted by the developer, one can argue that the intended architecture can not be extracted from the interpreted architecture, as the developer can only share his interpretation of the architecture.

2.3 Case description

This section will offer insight into the Minihydra system and go through what it is and how it is used. This section will also briefly present Kvaser AB, the company at which this study was conducted, and who is also the owner of the Minihydra system.

2.3.1 Kvaser AB

Kvaser AB is a company based in Sweden, focused on supplying CAN solutions and interfaces to its customers. Their products are used by engineers from all over the world during the development, deployment, and maintenance of CAN systems. Kvaser AB has over the last 30 years developed more than 60 CAN-related products, and are experts within the field. Kvaser is using an agile development method (Scrum) and has been doing so for the majority of the development of the Minihydra system.

CAN (Controller Area Network) is a data bus primarily used in the automotive industry. A modern car can contain 70 ECUs (electronic control units), and while some of these are independent, others need to communicate with each other [17]. A CAN bus uses only two wires, but yet allows for fast and secure message transmission between the nodes of the network. CAN also support prioritization between nodes, extensive error detection, and fault tolerance.

2.3.2 The Minihydra system

Minihydra is the name of the system this thesis has come to analyze. The system is used as firmware in a selection of products and has been developed and maintained for more than 10 years. The products using the Minihydra system as firmware are all CAN-interfaces but differ from each other in the area of use, connections, number of CAN channels, etc. The Minihydra system is almost exclusively written in the language C and the entire family of products is made up of roughly 800 000 lines of code.

The name comes from a mythological creature who bears the name Hydra. Hydra was a serpentine creature with multiple heads, and the name was given as a representation of the initial hardware configuration of the system. The main processor represented the body and multiple connected smaller input-output-processors to handle communication each represented one of the heads. Since the product was first launched the hardware configuration has changed, and the system is now using a single processor to run the software, but the name remains.

The Minihydra system is compiled with product-specific variations, due to adaptations for both different hardware and software configurations. A build system is used to speed up the build process, by compiling the software into libraries that are then combined into the firmware versions used by the products. To narrow down the scope into something feasible for a thesis, one of the simpler products was selected to be examined in-depth, hence some modules that were not used by the products firmware could be left out of the analysis. The Minihydra system is supplemented with a small operating system to run on, however, this operating system will also not be examined in detail in this thesis. The operating system is size-wise the biggest part with over 500 000 lines of code, and the Minihydra system itself consists of over 300 000 lines of code. This thesis is scoped to investigate the software for one single product, the *Kvaser USBcan Light 4xHS*. Due to only analyzing one single product, there is no need to include code for variations in hardware or software, further reducing the lines of code to be inspected to just above 150 000.

There is a document describing the architecture of Minihydra, however as this document presents the system at a low abstraction level it is hard to form an initial understanding of the system. The documentation was not used during the case study but is briefly discussed in Section 7.4.

In short, the Minihydra system is comprised of a number of prioritized operating system processes called tasks. All tasks share the same memory space and are run on the same processor. Some of these tasks are of a special type called Hydra Entities (HEs), and have extra features and attributes making for example message passing between them easier. What HEs and tasks a product is running is determined during boot, where the product configures itself based on its EAN (European Article Number). HEs communicate through Hydra Commands, a special message type representing a command or request. Hydra Commands are based on a message type provided by the operating system, but includes some extra features.

2.4 The C4-model

The software architecture of a system can be expressed on different levels of abstraction to show different levels of detail. The C4-model defines four different levels of abstraction to represent the software architecture of a system:

1. Context diagram - Showing the context of the system, where users interact with it, and how it is connected to other systems. An example context diagram can be seen in Figure 2.2.
2. Container diagram - A zoomed-in view of the system, showing all of its major parts and how they are communicating with or related to each other, external systems, and users. An example container diagram can be seen in Figure 2.3.
3. Component diagram - A zoomed-in view on a specific container, showing all components and how they are interacting with each other, other containers, or other systems. An example component diagram can be seen in Figure 2.4.
4. Code diagram - A zoomed-in view of a component, showing its implementation details through for example a UML diagram.

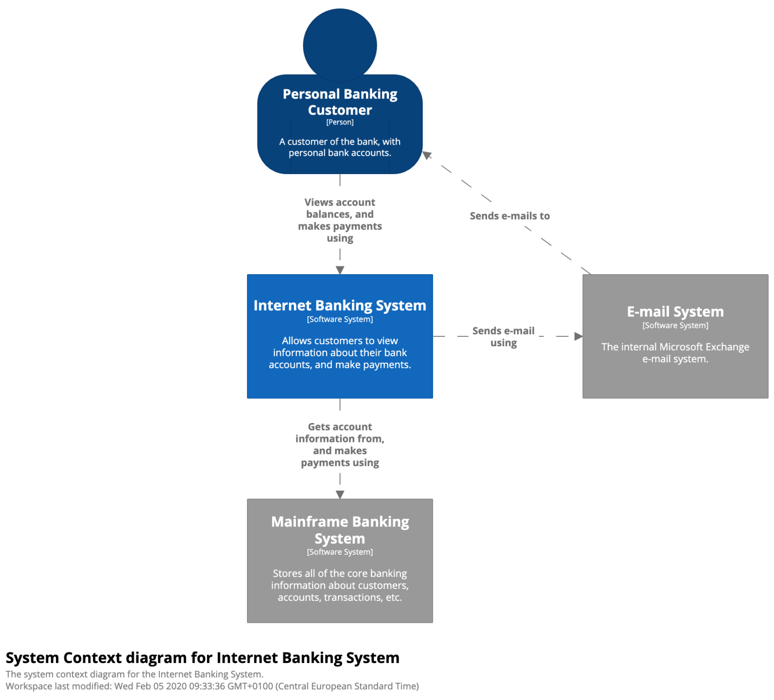
Personal Banking Customer
[Person]
A customer of the bank, with personal bank accounts.
Views account balances, and makes payments using
Sends e-mails to
Internet Banking System
[Software System]
Allows customers to view information about their bank accounts, and make payments.
Sends e-mail using
E-mail System
[Software System]
The internal Microsoft Exchange e-mail system.
Gets account information from, and makes payments using
Mainframe Banking System
[Software System]
Stores all of the core banking information about customers, accounts, transactions, etc.
System Context diagram for Internet Banking System
The system context diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

Figure 2.2: A context diagram shows the system and its context, in this case featuring an internet banking system. The example was borrowed from the C4-model's website [3].

There is no specified notation within the C4-model, but clear descriptions and captions are emphasized, and all acronyms used should be understandable by all audiences. The diagrams should ideally be self-standing and understandable even

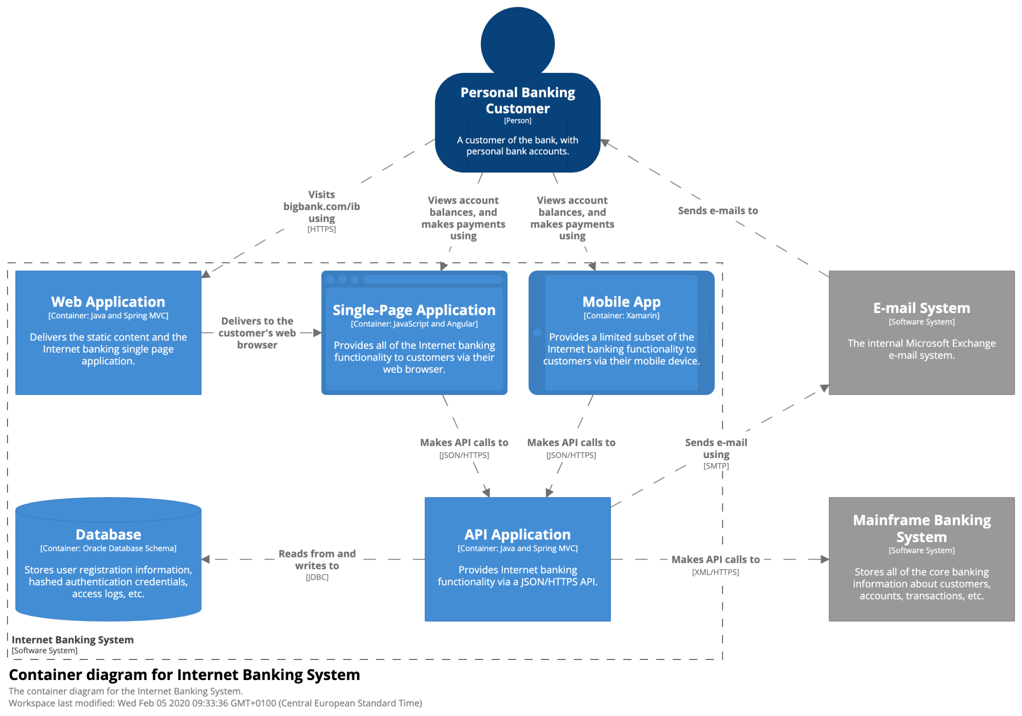
Personal Banking Customer
[Person]
A customer of the bank, with personal bank accounts.
Visits bigbank.com/ib using
[HTTPS]
Views account balances, and makes payments using
Views account balances, and makes payments using
Sends e-mails to
Web Application
[Container: Java and Spring MVC]
Delivers the static content and the Internet banking single page application.
Delivers to the customer's web browser
Single-Page Application
[Container: JavaScript and Angular]
Provides all of the Internet banking functionality to customers via their web browser.
Mobile App
[Container: Xamarin]
Provides a limited subset of the Internet banking functionality to customers via their mobile device.
E-mail System
[Software System]
The internal Microsoft Exchange e-mail system.
Makes API calls to
[JSON/HTTPS]
Makes API calls to
[JSON/HTTPS]
Sends e-mail using
[SMTP]
Database
[Container: Oracle Database Schema]
Stores user registration information, hashed authentication credentials, access logs, etc.
Reads from and writes to
[JDBC]
API Application
[Container: Java and Spring MVC]
Provides Internet banking functionality via a JSON/HTTPS API.
Makes API calls to
[XML/HTTPS]
Mainframe Banking System
[Software System]
Stores all of the core banking information about customers, accounts, transactions, etc.
Internet Banking System
[Software System]
Container diagram for Internet Banking System
The container diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

Figure 2.3: A container diagram shows the high-level modules forming the system, as well as its interactions with users or external systems. The example was borrowed from the C4-model's website [3].

without textual context. The use of shapes and colors is encouraged to enhance readability. An element in a C4-model diagram should have a specified type (e.g. person, container, etc), a short description, and for components and containers also specify the technology used (e.g. C, Java, etc). Lastly, every relation should be represented with a label and preferably be uni-directional [3].

The examples shown in Figure 2.2, 2.3, and 2.4 feature an internet banking system, and were presented to the developers during their interviews to better explain the C4-model. The examples feature the three levels of abstraction used in this case study, as code diagrams are not used in this thesis. The internet banking system presented in the diagrams features an example system from the C4-model's website and is hence not related to the system studied in this thesis [3].

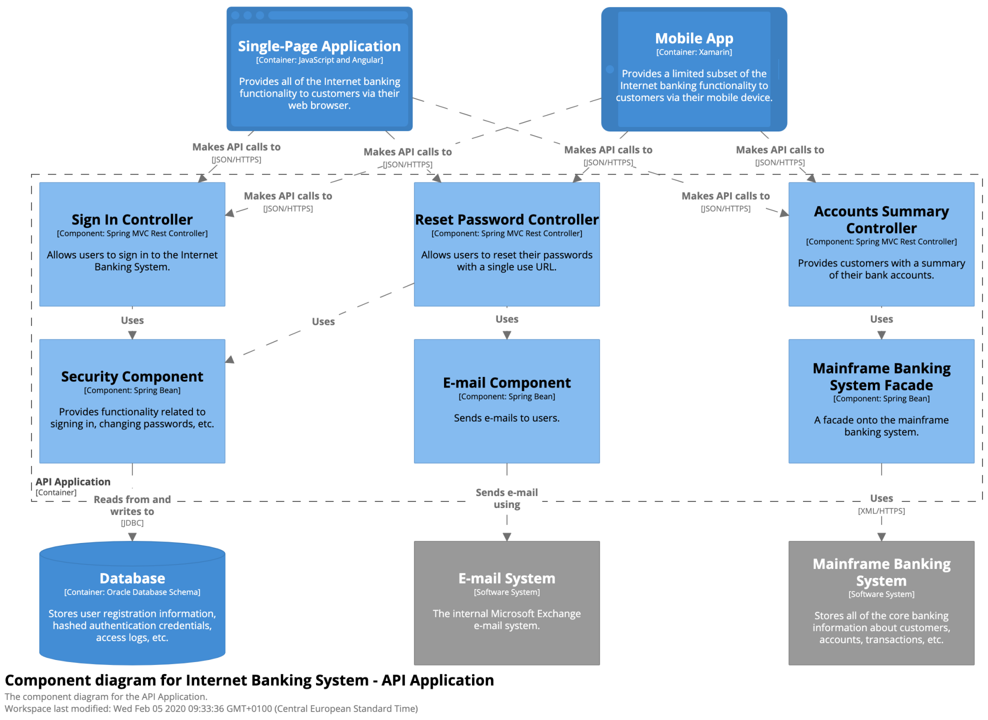
Single-Page Application
[Container: JavaScript and Angular]
Provides all of the Internet banking functionality to customers via their web browser.
Mobile App
[Container: Xamarin]
Provides a limited subset of the Internet banking functionality to customers via their mobile device.
Makes API calls to
[JSON/HTTPS]
Makes API calls to
[JSON/HTTPS]
Makes API calls to
[JSON/HTTPS]
Makes API calls to
[JSON/HTTPS]
Makes API calls to
[JSON/HTTPS]
Makes API calls to
[JSON/HTTPS]
Sign In Controller
[Component: Spring MVC Rest Controller]
Allows users to sign in to the Internet Banking System.
Reset Password Controller
[Component: Spring MVC Rest Controller]
Allows users to reset their passwords with a single use URL.
Accounts Summary Controller
[Component: Spring MVC Rest Controller]
Provides customers with a summary of their bank accounts.
Uses
Uses
Uses
Uses
Security Component
[Component: Spring Bean]
Provides functionality related to signing in, changing passwords, etc.
E-mail Component
[Component: Spring Bean]
Sends e-mails to users.
Mainframe Banking System Facade
[Component: Spring Bean]
A facade onto the mainframe banking system.
API Application
[Container]
Reads from and writes to
[JDBC]
Sends e-mail using
Uses
[XML/HTTPS]
Database
[Container: Oracle Database Schema]
Stores user registration information, hashed authentication credentials, access logs, etc.
E-mail System
[Software System]
The internal Microsoft Exchange e-mail system.
Mainframe Banking System
[Software System]
Stores all of the core banking information about customers, accounts, transactions, etc.
Component diagram for Internet Banking System - API Application
The component diagram for the API Application.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

Figure 2.4: A component diagram shows the modules of a specific container, as well as their interaction with other containers, users, or external systems. The example was borrowed from the C4-model's website [3].

3
Related works

This section will offer insights into related works previously performed in or around subjects relevant to this thesis. The aim of this section is to put this thesis and the methods used in a larger context in regards to previously performed research. There is little research on the topic of software architecture interpretation and software architectural gaps. However, the closely related topic of software architectural erosion has been studied extensively over the last decades.

3.1 Software architecture erosion

The term software architecture erosion is used when describing the differences between the intended architecture and the implemented architecture. The concept of software degrading over time is nothing new and was mentioned already in the late '60s, however, the term *software architecture erosion* was not yet used back then. Instead, it was framed as the *software crisis* [18]. The term software architecture erosion differs somewhat between researchers but Terra et.al described it as "the progressive gap normally observed between the planned and the actual architecture of a software system as implemented by its source code" [16, p. 1]. The overall difference between the two architectures, referenced by Terra et.al as progressive gap, should however not be confused with the definition for gap used in this paper as defined in Section 2.2.

Software architecture erosion can also be described as when a system's implemented and intended architecture drift apart [5], or when the architecture of a system becomes out of sync compared to its requirements and implementation [11]. According to both Parnas as well as Bosch and van Gurp all software systems erode over time, and using good methods during development does not prevent the erosion but can only postpone it [19], [12].

Since this thesis is comparing the implemented architecture and the interpreted architecture, the term software architecture erosion can not be used for the findings of this thesis, but the topic is similar and a lot of research has been done on the subject of software architecture erosion. The interpreted architecture gathered from the developers and used in this thesis is likely to be affected by the implementation, thus the interpretation is likely more similar to the implemented architecture than the intended architecture. This means that comparing the implemented architecture to the interpreted will likely result in fewer and/or smaller gaps than if the comparison would have been between the implemented and intended architecture.

3.1.1 Identification of software architecture erosion

There are several ways of identifying software architecture erosion. Studies have used both manual and automatic conformance checking in order to map the architecture under study to its implemented source code [6], [20], [21], [22]. These strategies are often used for identifying, preventing, and repairing software architecture erosion. An example of such a strategy is the reflexion model. The reflexion model approach is to create a high-level architectural model during development that, together with a source code model and mapping, can highlight architectural misalignment [22]. However, other studies have found that in some cases the reflexion model has concealed inconsistencies and, as such, contradicted its purpose as an architectural evaluation tool [21].

The method of discovering architectural gaps in this thesis differentiates from the process of identifying software architecture erosion by using developer interpreted architectural views elicited from interviews instead of using pre-existing architectural descriptions. Other processes, such as mapping one architectural view to another and manual conformance checking is similar to the methods used to identify software architectural erosion.

3.1.2 Consequences of software architecture erosion

The purpose of an intended software architecture is to ensure that the software system, once completed, achieves its requirements and quality attributes. A software system has many quality attributes, and one of the main challenges for the software architect is to prioritize and balance them according to the needs of the specific system. However, once the architecture gets eroded, several quality attributes such as maintainability, performance, and susceptibility to defects become worsened. This can drastically affect the lifespan of a software system [1], [2], [6].

As the interpreted architecture is a mix between the implemented and the intended software architecture, a comparison between the implemented architecture and the interpreted software architecture can be considered to be a subset of software architectural erosion. Due to this, the above-mentioned consequences are to some extent also true for gaps between the implemented and the interpreted software architecture.

3.1.3 Causes for software architecture erosion

There are several potential causes for software architecture erosion, Bosch and van Gurp were in their study of a fictive software system able to identify four of these [12], they are as follows:

- Lack of traceability of design decisions - The languages used to describe architecture and implementation are very different, and there is often no direct link between the two, making it difficult to track which design decision relates to which implementation.
- Increasing maintenance cost - As the system becomes more complex developers become more prone to take short-cuts, either because they lack the architectural knowledge or that a proper solution would demand too much effort.
- Accumulation of design decisions - Design decisions interlock with each other, and if one of the decisions has to be changed, others may have to be altered as well. These changes may result in a system that is no longer optimal for the given requirements.
- Iterative methods - Agile practices inherently welcome changes to the requirements of the system under development. These changes sometimes require architectural changes, and as the system becomes more complex the optimal solution for these changes in regards to the architecture may become too effort demanding to implement. This leads to an eroded architecture that no longer represents the requirements and implementation [11], [12].

In another paper, Eick et al. listed no less than 8 causes for an eroding architecture [8]:

- Ill-suited architecture - An architecture that does not allow for changes will erode quickly, as changes are almost inevitable in most larger software development projects.
- Violations of the design - One change to the software that is not according to the design can force upcoming changes to breach the design again, forcing the implementation further and further away from the design.
- Nonspecific requirements - Not specifying requirements for the system precisely enough can lead to a higher number of changes needed later on in the development, increasing the risk of eroding the architecture.
- Time pressure - Developers taking shortcuts, not investigating the effects of their changes, not testing enough, or writing quick and dirty solutions.
- Inadequate programming tools - Not getting enough support from development tools can lead to misses both architecturally and in form of bugs.
- Organizational aspects - Staff turnover rate, limited communication within the company, code ownership, and even dipping morale can result in code with lower quality being produced.
- Capabilities of different programmers - There may be complex parts of the code that is hard to understand and work with for less experienced programmers.

- Change implementation - Having good procedures for implementing changes developed in parallel by different developers will reduce the risk of one change corrupting another.

Some of the causes listed in the different papers are similar to each other, but both papers also cover unique aspects. For example, the organizational aspects listed by Eick et al. are very interesting, as behind every team of developers there is a group of individuals [8]. Organizational aspects are not investigated or taken into account in the study performed by Bosch and van Gurp, as they investigated a fictive system, hence no developers were involved, and no organizational aspects could be studied [12].

3.1.4 Techniques to prevent or repair eroded software architecture

Balasubramaniam and de Silva investigated through a survey potential strategies to minimize or prevent software architecture erosion or repair architectures that have already eroded. Several different strategies were examined, for example, process-oriented architecture conformance and architecture recovery. Process-oriented architecture conformance is a strategy to minimize software architecture erosion by ensuring conformance during the development of a software system, and architecture recovery is a way to repair eroded architecture involving reverse-engineering. The study concluded that none of the presented strategies can on their own provide a solution to the problem of software architecture erosion. It was, however, possible to evaluate when a certain strategy might be the most effective. The decision of which strategy was the most effective was based on several factors such as what practices are being used for development, what type of system is being developed, and in what phase during development the system is currently in [2].

Other studies have also examined the effectiveness of different software tools used to either repair already eroded software architecture or to be used as a part of the development process. A typical feature of these tools is to automatically evaluate the source code against the intended architecture [20], [6], [13], this does however require the architecture to be documented in such a way that an automatic comparison is possible. Naturally, creating and maintaining such documentation would require additional effort from the software architects of the system, but it has been shown that the estimated effort of this would be minimal in contrast to the effort typically spent on evolving a software system [13].

3.2 Interpretation of software architecture

The importance of having a well thought through software architecture has already been emphasized, but in order for the intended software architecture to be correctly implemented, it first has to pass through the mind of a developer.

To understand in retrospect why certain design decision were taken is no easy task, and can hence make some decisions seem unnecessary or even redundant. This is due to that software architecture have no way of representing the underlying reasons for the design decisions taken, even when they are evidently followed in the documented and intended architecture. Even when the underlying reasons for the design decisions are documented, these notes are often hard to decipher for someone outside of the initial design loop [10].

There are countless ways of representing software architecture, but many of them include boxes with names and lines between them. Such a representation of software architecture can offer a quick overview of a system, but will likely not cover important details nor reasons for the underlying decisions, and can thus be interpreted in many different ways [23]. Despite the risk of misinterpreting the architecture and the traceability issues with the underlying design decisions, different interpretations of software architecture have not been studied extensively.

4 Research questions

The research questions this thesis aims to answer are presented in this chapter.

4.1 Research question 1 (RQ1)

The first research question is of quantitative nature and aims to find gaps between the implemented architecture of a software system, obtained through reverse-engineering, and the interpreted software architecture, obtained through interviews with developers of said system:

RQ1: Are there gaps between the implemented software architecture and the interpreted software architecture of the system?

4.2 Research question 2 (RQ2)

The second research question is of qualitative nature, and aims to offer further insight into the gaps found in RQ1 by looking into possible causes and effects through interviews with the developers:

RQ2: What are the possible causes and effects of the gaps found in RQ1?

- SQ2.1: What are the possible causes of the gaps?
- SQ2.2: What are the possible effects of the gaps?

5 Methods

In this chapter, the different methods and procedures used in this study are presented and explained. Firstly, the reason to follow the exploratory case study design is addressed. Secondly, the different procedures used for data collection are described, such as reverse-engineering the implemented software architecture and eliciting the interpreted software architecture. Lastly, the data analysis procedures are presented. An overview of all methods used in this thesis, and in what order, can be seen in Figure 5.1.

5.1 Case study design: exploratory

The primary goal of this study was to discover software architectural gaps and to use these as points of discussion with developers of the studied system in order to gain new insights and ideas regarding the gaps. Because of this, the exploratory case study design was used since it focuses on observing a phenomenon in the field in order to generate new ideas and hypotheses. Furthermore, case studies are often the norm when it comes to studies regarding software engineering. This is because software engineering is a multidisciplinary profession which is where case studies are normally conducted. Case studies provide knowledge to people and organizations by exploring certain phenomena, in this case study it is the occurrence of gaps and their causes and effects that have been studied [24].

It is, however, important to note that this study does not result in some statistically significant conclusion since that is not possible while performing a primarily qualitative study of this design [24]. This study instead aimed to present its possible findings by reasoning around the collected data and presenting a chain of events such as to convincingly declare its conclusions.

5.2 Data collection procedures

In order to find gaps between the implemented and interpreted software architecture of the system and to gain new insights about these, three different activities were performed. Firstly, the implemented software architecture was acquired through reverse-engineering of the system's source code. Secondly, the interpreted software

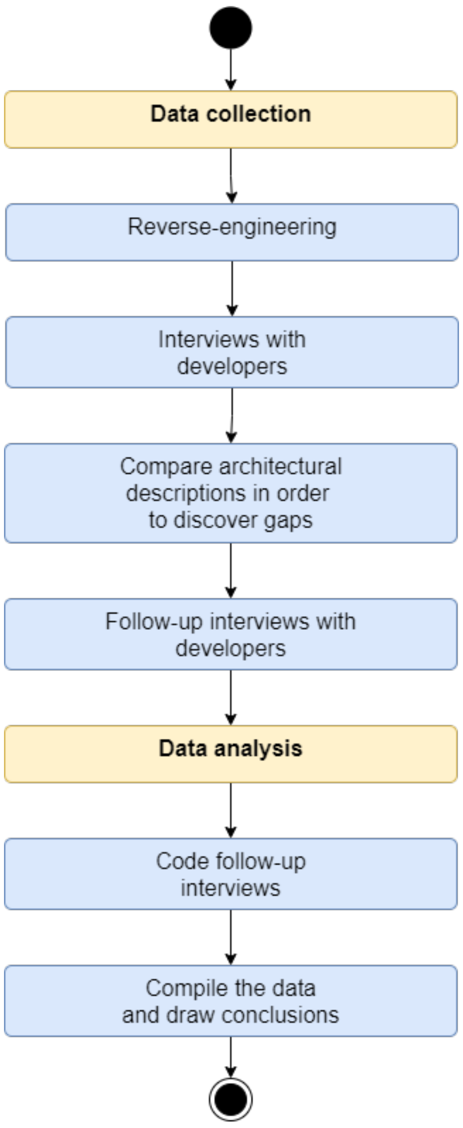
Data collection
Reverse-engineering
Interviews with developers
Compare architectural descriptions in order to discover gaps
Follow-up interviews with developers
Data analysis
Code follow-up interviews
Compile the data and draw conclusions

Figure 5.1: Activity diagram showing the different activities of this study and in what order they were performed.

architecture was elicited by interviewing developers of the system and asking them to describe its architecture. Lastly, any gaps between the architectures were presented to the developers in a follow-up interview in order to gain new insights of their potential causes and effects.

5.2.1 Reverse-engineering the implemented software architecture of the system

To be able to compare the implemented architecture with another architectural view, the implemented architecture must first be extracted from the source code. Extracting the architecture from source code is like extracting the recipe from a cake, and can be quite a tricky process. Source code is the most specific description of the behavior of the software that is still readable to humans. The process of reverse-engineering the complete system or product back to its architectural descriptions is a task of both careful information extraction and a balancing act of abstraction.

Before selecting what method to use to reverse-engineer the software, several methods were tested and evaluated. This section will only present the method that was finally used to produce the reverse-engineered architectural description in this thesis, while the other methods will be discussed in Section 7.1. The method that was settled upon was the one presented by Tilley et al. [25] and Granchelli et al. [4].

The process can be divided into three stages, as depicted in Figure 5.2, namely extraction, abstraction, and presentation [25], [4]. As the name of the first step suggests, the initial step of the method is to gather information about the system from the source code. The second step is to compile the gathered information in order to map out the implemented architecture and create different descriptions on multiple abstraction levels. Finally, in the third step, diagrams and other representations of the software architecture are created to present the knowledge gathered and gained during the process [25], [26].

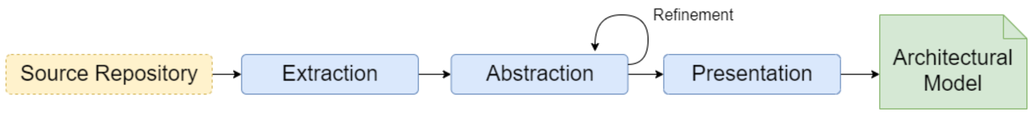

Figure 5.2: The reverse-engineering process consist of three phases, namely extraction, abstraction, and presentation. This figure was inspired by the works of Granchelli [4].

The extraction process started by locating all relevant source files. As the Minihydra system is used for various hardware and software configurations in different products, there are files in the system folder structure that are not used for the product under investigation in this thesis. To find the files belonging to the product under investigation, the system build files were studied.

The build system combines files into libraries, and then combines said libraries into the different firmware versions. Each library can include files, folders, and other libraries, and the content of a library is specified in a build file. The build files are spread across the folder structure of the system's source code. To find the files used by the product under investigation, the build file specifying the included libraries forming the firmware for the product was located. Each library included was then traced to a build file, until the source files of all included libraries had been found.

In order to start the reverse-engineering process, the initialization point of the program had to be located. Only one of the files used in the build process turned out to have a function called *main(void)*, who in turn was never called for by any other function. An important configuration function used by all Minihydra products during start-up was called early on and exclusively in the main function, confirming the finding of the initialization point.

The configuration file is used to configure the Minihydra system for the specific product it is currently running on, as one firmware version can be run on different products using different configurations. The product investigated in this thesis is one of the simplest products in the Minihydra product family, hence many of the available features are disabled and the number of running tasks are fewer, making the program easier to grasp.

The source code was then inspected manually line by line. This method of reverse-engineering may seem like an odd and primitive choice but was selected due to that it would produce an architectural description based on data flow rather than dependencies. The choice of reverse-engineering method is further discussed in Section 7.1.

The program was divided into its different running tasks to be able to study the communication between them. All relations and messages sent between tasks were noted down for the upcoming steps of the reverse-engineering process: abstraction and presentation.

The C4-model includes four different types of diagrams, namely context, container, component, and code diagrams. Creating code diagrams for the whole system would be too time-consuming for the scope of this thesis. Furthermore, it is unlikely that they would have impacted the results of the study as the developers most likely would not have had a detailed perception of the code on that level, thus code level diagrams were never generated.

The process of abstracting and presenting diagrams goes somewhat hand in hand, as abstraction can be a good solution for a cluttered diagram, but the clutter is not unveiled before the diagram is presented. The component diagrams, for example, which are the diagrams with the lowest abstraction level presented in this thesis, did not need much abstraction since they are supposed to be specific and detailed.

The diagrams were visualized using PlantUML, a text-based tool for generating diagrams. The finished diagrams are presented in Section 6 for easier comparison between the implemented and interpreted architectures side by side.

5.2.2 Eliciting the interpreted software architecture of the system

To elicit the interpreted software architecture of the system, individual interviews with several developers of the system were performed. In an attempt to keep all initial interviews comparable, they were all following a common structure. The interviews started with a short introduction about our study and an establishing of definitions, including the phenomena under study, gaps, as well as the objective of the study. The task of creating diagrams was presented, and example diagrams were shown showcasing the three different levels of abstraction used. The example diagrams were unrelated to Minihydra and depicted a banking system, which can be seen in Figure 2.2, 2.3, and 2.4 respectively. The reason for showing the example diagrams was to show the developers how the C4-model could be used to describe software architecture, and thus minimizing the risk of misunderstandings and/or misinterpretations.

After this introduction, the developers were tasked with creating their own diagrams of the system. Firstly, a context diagram showing how the system interacts with other external systems and/or users was created. Secondly, a container diagram showing the major software parts that the system is comprised of including noteworthy dependencies and communications between these. Finally, the developers were asked to create a component diagram for the container that they were the most familiar with. There are primarily two reasons for letting the developers pick what container to focus on and draw a component diagram for. Firstly, creating component diagrams for an entire system would require too much time, and we would rather let the developers use the time to more accurately recreate one or two component diagrams than to hastily speed through all of them. Secondly, in order to describe a container and its components the developers need to have sufficient knowledge about it. By letting the developers choose which container to describe there is a higher chance that they will be able to create diagrams that they are satisfied with.

The diagrams were initially drawn on a whiteboard by the developers but have since been digitalized after the interviews using PlantUML to make comparisons between the implemented diagrams and the interpreted diagrams easier. The digitalized versions of the diagrams are presented in Chapter 6 and Appendix A.

The interviews were structured using the pyramid model as described by Seaman [27]. In short, the pyramid model is to first establish all common definitions (e.g. what is a gap and how to use the C4-model to depict the software architecture) and then gradually open up for improvised questions based on where the interview goes [27]. The goal of following this specific interview method was that the developer gradually could lean more and more into those parts of the architecture that he feels he knows the best and thus creating a better representation of his interpreted architecture.

5.2.3 Obtaining developer insights regarding discovered gaps

While the first round of interviews was all about eliciting the developers' interpreted views of the system, the follow-up interviews were instead focused on the gaps discovered between the implemented architecture and their individual interpreted architecture. The gaps were used as points of discussion in order for the developers to share their insights regarding possible causes and effects. The outcome of the follow-up interviews is presented and discussed in Chapter 6.

Just like the initial interviews, the follow-up interviews followed the pyramid model to allow for important definitions to be established in the beginning yet more open discussion towards the end. It was established that the interviews and the diagrams previously created by the developer were in no way an evaluation of the developer's knowledge, skill level, or understanding. This was an important statement in order to establish trust between the interviewer and the interviewee to let the interviewee provide honest answers without worrying about consequences or judgment [28].

5.3 Data analysis procedures

In this section, the procedures performed in order to analyze the collected data are presented. This includes how the implemented architecture was compared to the interpreted architectures in order to find gaps as well as how the answers given in the follow-up interviews were analyzed.

5.3.1 Finding gaps

Once both the implemented and interpreted architecture had been obtained through reverse-engineering and developer interviews respectively, they were analyzed and compared in order to find potential gaps. Following our definition of software architectural gaps, any architectural difference between the two compared diagrams, for example, a missing component or an incorrect relation, is denoted as a gap.

The diagrams depicting the developers' interpreted architecture were digitalized before the comparison to improve readability and diagram structure and thereby making the comparison easier. In order to ensure that no information held by the hand-drawn diagrams was altered or changed during the process of digitalization, the original diagram and the digital copy were carefully compared by another person than the one responsible for the digitalization process.

All the gaps discovered were noted down, both to answer RQ1, but also to form the basis for the follow-up interviews with the developers, and thereby contributing towards RQ2.

5.3.2 Analysing developer insights from the follow-up interviews

To make sure that no information was lost during the interviews they were all recorded, and in order to utilize the *coding* analysis method, they were also transcribed. *Coding* has been described as the critical link between data collection and their explanation of meaning in qualitative data analysis [29]. In this study coding allowed for analysis of the gathered developer insights in a structured way by examining patterns and relations between the different interviews. This is done by coding different quotes from the developers. For example, if several developers conveyed a similar insight during their interviews, those quotes were grouped together by the same code.

The process of coding made it easier to build stronger and more cohesive arguments for the findings of the study. However, since coding is interpretive, it is unlikely that two different coders will end up with exactly the same codes even when coding the same data. In an attempt to mitigate coding biases three different strategies were used. Firstly, the coding was done in parallel by two different coders, and then cross-checked and negotiated. Secondly, the coding was done as the data was transcribed, and lastly, our interpretations of the transcriptions were shared and checked with some of the interviewed developers themselves [30].

6 Results

This section will present the results of the study and aims to answer the two research questions defined in Chapter 4. Discovery of the software architectural gaps was done by comparing the diagrams created by the developers of the system with the diagrams created through reverse-engineering. As per our definition of software architectural gaps any difference, for example, conflicting descriptions or missing components, between the diagrams were regarded as gaps. Analyzing these results it is possible to answer RQ1. To answer RQ2 the earlier discovered gaps were presented to the developers in a follow-up interview. In this interview the developers were encouraged to reason about the gaps regarding why they exist and what cause and effect they might have, to provide new insights about the software architectural gaps.

6.1 Discovered software architectural gaps

By comparing the diagrams created by the developers of the system with the diagrams obtained through reverse-engineering several gaps could be observed. These comparisons were done for both context and container diagrams, as well as for the component diagrams where interpreted diagrams were available.

To gather information regarding how the developers perceive the system, interviews with four of the developers at the company were conducted. Three of the interviews resulted in diagrams roughly following the C4-model, and could hence be compared to the diagrams produced through the reverse-engineering process. The diagrams following the C4-model are be presented in this chapter, but the diagram not following the C4-model is instead presented in Chapter 7. A short explanation of the produced diagrams is provided for each of the three abstraction levels (context, container, and component), after which the gaps for that particular abstraction level are presented.

6.1.1 Presenting context diagrams

The purpose of a context diagram is to present the system in its context and set it into relation to the user and other systems. The context diagram obtained through reverse-engineering can be seen in Figure 6.1 and one of the diagrams drawn by

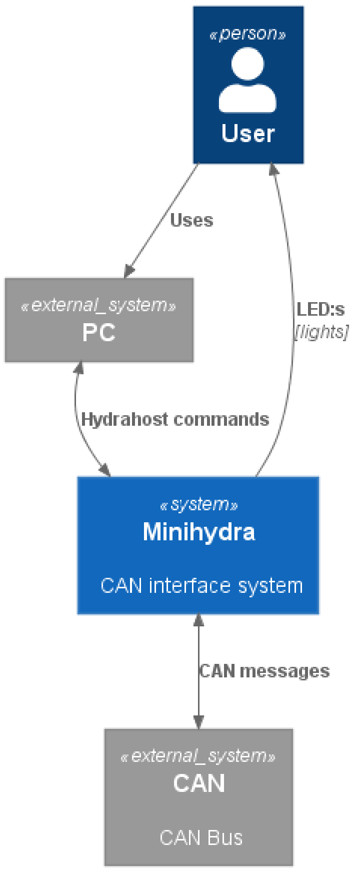

Figure 6.1: Context diagram showing the implemented software architecture of Minihydra.

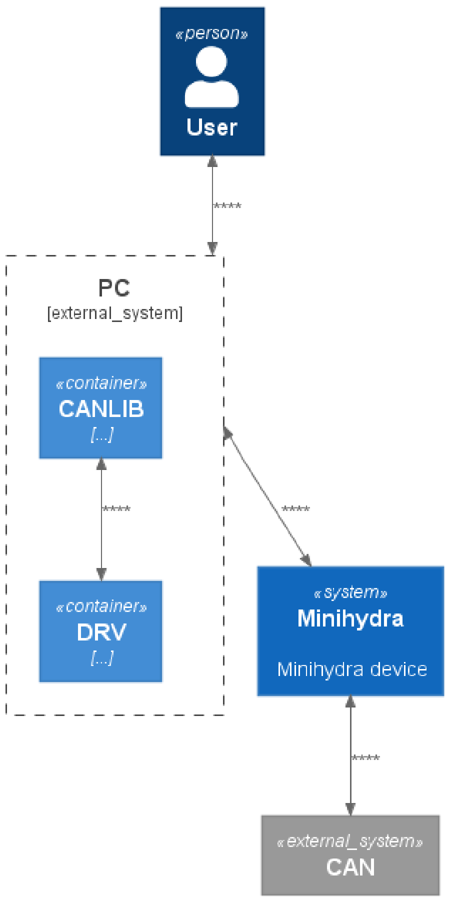

Figure 6.2: Context diagram showing an interpreted software architecture of Minihydra.

one of the developers during their initial interview can be seen in Figure 6.2. The comparable context diagrams from the other interviews with the other developers were very similar to the one shown in Figure 6.2, with only minor changes inside of the external PC system.

The reverse-engineered diagram, seen in Figure 6.1, shows how the Minihydra system interacts with the user through the PC and LEDs, and how the system is also connected to the CAN-bus. To be noted is that this diagram is only true for the particular product investigated in this thesis, and not for all products running the Minihydra system.

Featured inside of the external PC system on the interpreted diagram, seen in Figure 6.2, is a driver, *DRV*, as well as a library used by software running on the PC to interact with the product called *CANLIB*.

6.1.2 Comparing context diagrams

There were not that many gaps observed when comparing the context diagrams. The differences inside of the external PC system will be ignored and are not considered to be gaps, as the software running on the PC is not a part of the Minihydra system, and hence outside of the scope of this study. There are mainly two other noticeable differences between the diagrams, namely that the communication between modules is labeled only on the reverse-engineered diagram, and that the user can get information from the system directly through LED lights on the product, without the use of a PC as an intermediator. Both of these are minor differences but are still regarded as gaps.

Communication gaps

Common for both of the gaps found when comparing the context diagrams, was that they regarded communication. Not to tag a relation as to what information is sent may seem like a small gap, but for the reader of the architectural description it is an important element in understanding how the system interacts with its surrounding.

The products running the Minihydra system generally have two ways of communicating with the user, either through the use of a PC or using its LED lights. The LED lights can indicate sent or received CAN-packets, as well as error frames and other important information. The LED lights are not able to communicate as detailed information as the PC interface, but never the less it is presenting the user with information, and hence should be included in the context diagram of the product.

6.1.3 Presenting container diagrams

The container view is a more detailed view than the context diagram, and the system in focus is described more in-depth. The system is divided up into its abstracted primary modules and just like the context diagram relates the system to external systems and users. In the reverse-engineered diagram, shown in Figure 6.3, each of the containers represents an area of responsibility. The names of the containers are either similar to the underlying component name within the container, or a collective name representing all included components within the container. On each of the containers there is also a description of its responsibility.

Seen in Figure 6.4 is one of the container diagrams drawn by one of the developers, depicting his interpretation of the system's architecture. The interpreted diagram contains six different containers, five of which correlates to the reverse-engineered diagram. The correlating containers are as seen in Table 6.1.

The logging container presented in the interpreted diagram is a part of the Minihydra system, but not a part of the product investigated in this thesis. Thus, it does not have any counterpart in the reverse-engineered diagram and, as such, this difference is not regarded as a gap. This wider view of the Minihydra system (not product

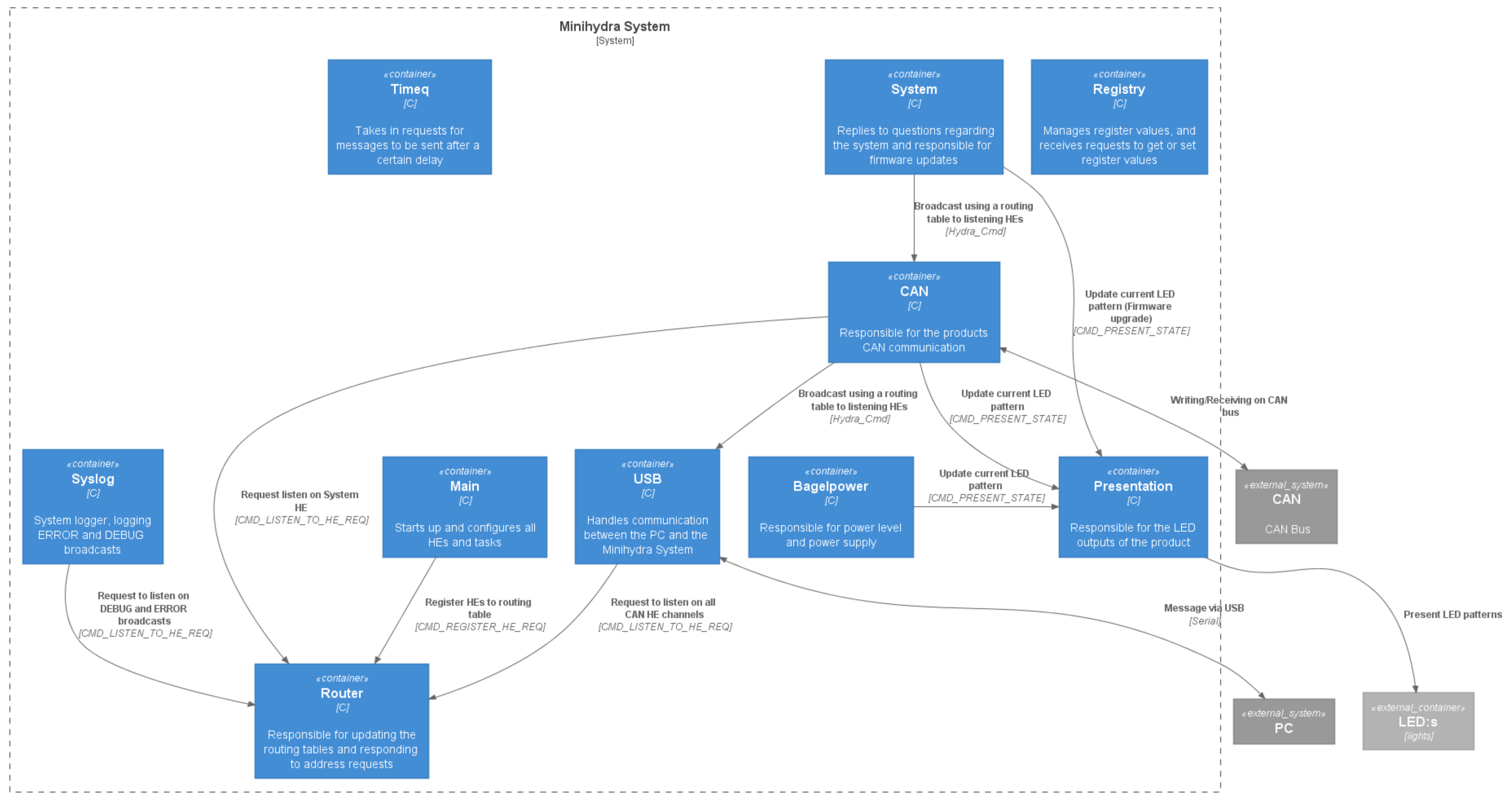
Minihydra System
[System]
«container»
Timeq
[C]
Takes in requests for messages to be sent after a certain delay
«container»
System
[C]
Replies to questions regarding the system and responsible for firmware updates
«container»
Registry
[C]
Manages register values, and receives requests to get or set register values
Broadcast using a routing table to listening HEs
[Hydra_Cmd]
«container»
CAN
[C]
Responsible for the products CAN communication
Update current LED pattern (Firmware upgrade)
[CMD_PRESENT_STATE]
Broadcast using a routing table to listening HEs
[Hydra_Cmd]
Update current LED pattern
[CMD_PRESENT_STATE]
Writing/Receiving on CAN bus
«container»
Syslog
[C]
System logger, logging ERROR and DEBUG broadcasts
Request listen on System HE
[CMD_LISTEN_TO_HE_REQ]
«container»
Main
[C]
Starts up and configures all HEs and tasks
«container»
USB
[C]
Handles communication between the PC and the Minihydra System
«container»
Bagelpower
[C]
Responsible for power level and power supply
Update current LED pattern
[CMD_PRESENT_STATE]
«container»
Presentation
[C]
Responsible for the LED outputs of the product
«external_system»
CAN
CAN Bus
Request to listen on DEBUG and ERROR broadcasts
[CMD_LISTEN_TO_HE_REQ]
Register HEs to routing table
[CMD_REGISTER_HE_REQ]
Request to listen on all CAN HE channels
[CMD_LISTEN_TO_HE_REQ]
Message via USB
[Serial]
Present LED patterns
«container»
Router
[C]
Responsible for updating the routing tables and responding to address requests
«external_system»
PC
«external_container»
LED:s
[lights]

Figure 6.3: Container diagram showing the implemented software architecture of Minihydra.

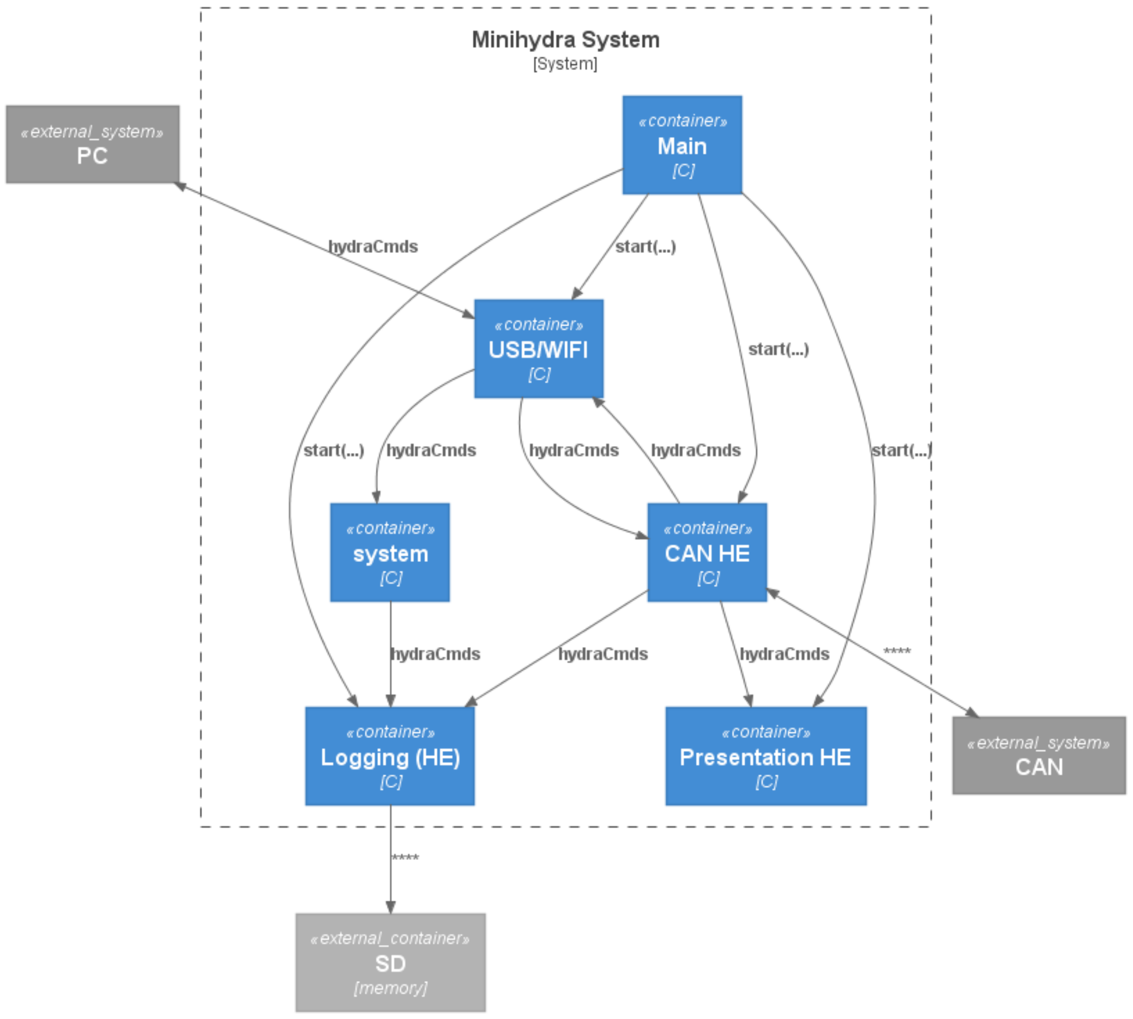

Figure 6.4: Container diagram showing an interpreted software architecture of Minihydra.

Developer interpreted	**Reverse-engineered**
main	Main
USB/WIFI	USB
system	System
CAN HE	CAN
Presentation HE	Presentation

Table 6.1: Naming correlation between a developer interpreted container diagram and the reverse-engineered container diagram.

specific) is also why this developer decided to name the USB container *USB/WIFI*, as other products use WiFi interfaces instead of or in combination with USB.

Another difference between the diagrams that is not considered a gap, since it is only related to abstraction, is that the file *Utility* is included in the interpreted diagram seen in Figure 6.5. Utility is not featured on the reverse-engineered diagram as it is a static file, and the diagrams, interpreted and reverse-engineered, generally feature tasks and information flow and not individual files. All container diagrams can be seen in appendix A.

6.1.4 Comparing container diagrams

Several gaps were identified when comparing the container diagrams produced by the developers and the container diagram that was reverse-engineered from the implementation. Some gaps were appearing in multiple interviews, indicating that those gaps may not be exclusive to the people interviewed. Instead, this shows that there may exist architectural gaps at an organizational level as well. The container diagram produced using reverse-engineering can be seen in figure 6.3, and one of the container diagrams drawn by a developer can be seen in figure 6.4.

Missing containers

Missing containers was a common type of gap when comparing the interpreted container diagrams to the reverse-engineered container diagram. Some freedom was allowed when naming and grouping containers, as the components bunched together into the containers are not strictly bound to each other. The grouping depend on mental models and dependencies between tasks, and can hence produce slight variations from person to person.

A container is seen as missing if it is not present or if its responsibility is not covered by any of the other containers. An example of a missing container in the diagram shown in Figure 6.4 is the lack of the router container. In the reverse-engineered diagram, seen in Figure 6.3, the router is an integral part of the system, being responsible for storing and distributing addresses to the other HEs in the system. However, that container and its responsibility are absent in the interpreted diagram. The missing router container was a common gap, and the only developer that put the router in his container diagram, see Figure 6.5, had misinterpreted parts of its functionality as well as how it was interacting with the other components.

Communication gaps

Gaps regarding communication were discovered in a couple of ways, in some cases, the communication between two containers was missing all together and in other cases, the wrong or no means of communication was specified. Missing communication could further be divided into two categories: one where the two communicating containers were included in the diagram, but communication between them missing, and another where one or both of the communicating containers were missing, and

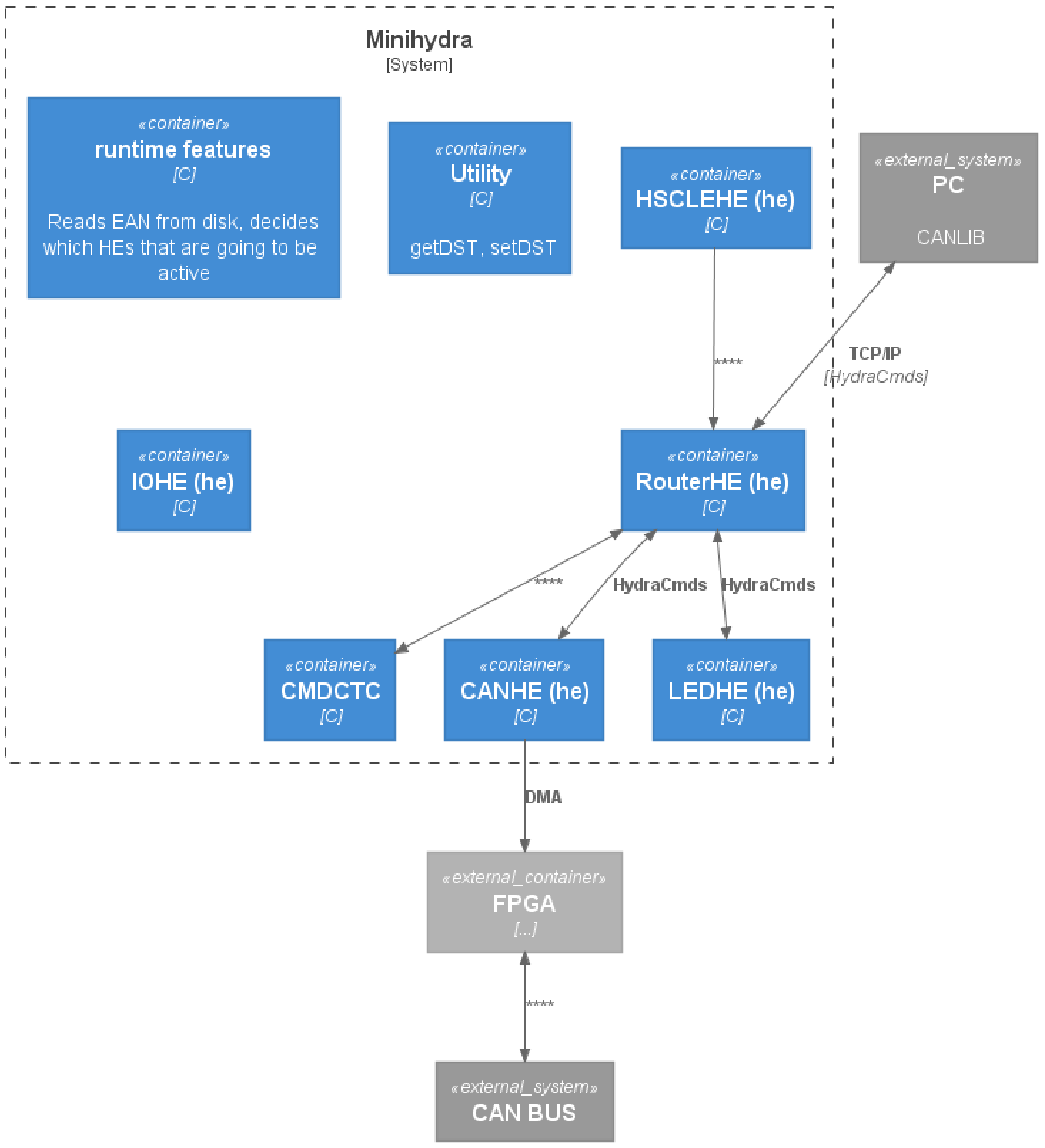

Figure 6.5: Container diagram showing an interpreted software architecture of Minihydra.

hence their communication. This type of gap can be seen when comparing Figure 6.4 and Figure 6.3, where for example the System container sends information to both the CAN and Presentation containers in the reverse-engineered diagram.

Responsibility gaps

In some cases, a gap in responsibility could be found, where the developer thought the responsibility of a container was something else than its actual responsibility. An example of this is the router container, where two developers thought it was responsible for passing all messages between the other HEs, when it actually is keeping track of the addresses of the running HEs, and hand them out on request.

Labeling as Hydra Entity (HE)

Early on in the reverse-engineering process, a key part of the Minihydra systems software architecture was uncovered: the Hydra Entity (HE). In short, a HE is an extension of a task, giving it a few extra features and characteristics. In the reverse-engineered container diagram, seen in Figure 6.3, the label HE is not used for any of the containers, however, the label is used in all of the container diagrams drawn by developers for some or all of the containers (e.g. Figure 6.4).

Whether to label a container HE or not can sometimes be a matter of abstraction. We use the term ourselves but only in the component diagrams. Since some of the containers only contain a single HE, one could in those cases argue that the label could be put on the container as well. However, in cases where a container consists of more than one task or HE the label HE can no longer be used on the container as it would contradict the meaning of HE, a task with extra features and characteristics. Hence, the label can not be used for containers containing more than one HE, as a task can not contain another task. The label Hydra Entity is further discussed in Section 6.2.4.

6.1.5 Presenting component diagrams

In total, five different component diagrams were drawn by the developers. However, two of the component diagrams depicted modules that unfortunately are not used in the firmware for the product investigated in this thesis, leaving three interpreted component diagrams to compare to their reverse-engineered counterparts. One of the component diagrams drawn by one of the developers depicted the CAN-container, responsible for the products CAN-communication, see Figure 6.7, and the reverse-engineered counterpart can be seen in Figure 6.6. The reverse-engineered diagram features four different components: *Can HE*, *Can_tx HE*, *Can_rx HE* and *EventTask*. Each of the components briefly states their responsibility in their description. The only relationship abstracted from the reverse-engineered diagram is that all four components are started by the main container. This relation was not featured as it was common for all components in all containers and did not add much information regarding the architecture once the system was up and running.

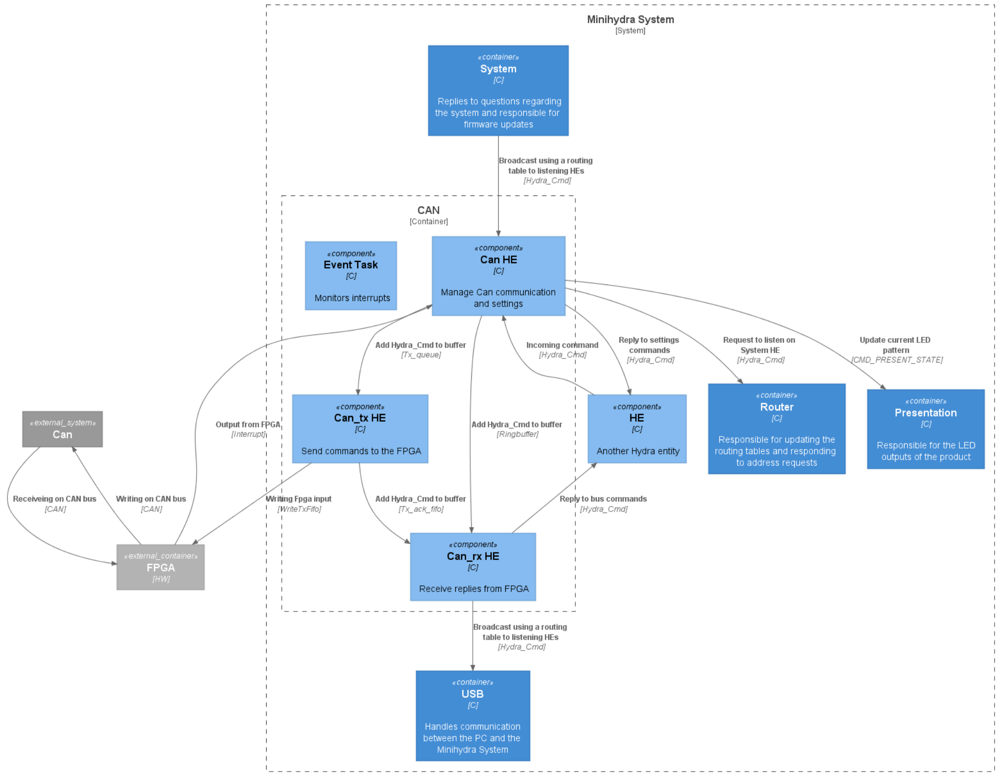

Figure 6.6: Component diagram showing the implemented software architecture of the *CAN* container in Minihydra.

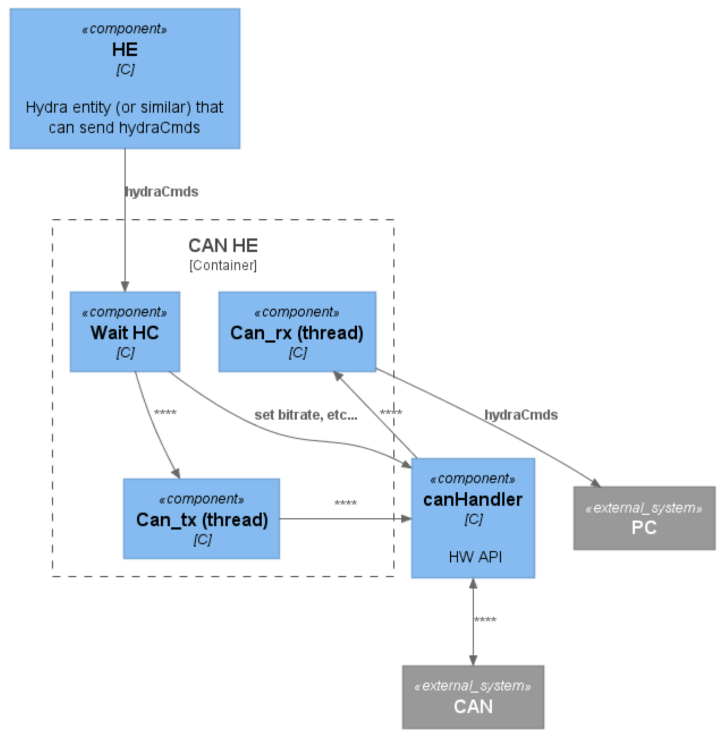

Figure 6.7: Component diagram showing an interpreted software architecture of the *CAN HE* container in Minihydra.

The component *Can HE* is a HE and is responsible for all settings regarding CAN-communication, as well as adding incoming messages to a buffer for *Can_rx HE* to handle. *Can_tx HE* and *Can_rx HE* are both Hydra Entities and are responsible for sending and receiving CAN-messages respectively. Lastly, the *EventTask*, which is the only task in the container that is not a HE, monitors hardware interrupts. *EventTask* not being a HE mainly changes one thing, namely that the task can not send or receive Hydra Commands from the other HEs, but also how the task is created and what information it stores.

The interpreted version of the CAN container can be seen in Figure 6.7, and contains three of the components previously presented, albeit with other names. The names stated in this diagram are the same names the developer used when the diagram was drawn, and even though changing them would make the diagrams easier to compare, it will soon become apparent why the names have remained. The component *Wait HC* is the same component as *Can HE*, and the name comes from that the component is waiting for Hydra Commands, for example, a request to change a CAN setting.

The last component featured in the interpreted CAN component diagram is called *canHandler*, and is a static file. Thus, it has been removed due to abstraction from the implemented diagram. The reason that the developer thought it was important to include this file in the diagram was that there are hardware differences between the products running the Minihydra system, and there are different canHandlers for different hardware configurations. As the scope of this thesis is set to a specific product, only one of the canHandler files is used and hence the component can be disregarded.

6.1.6 Comparing component diagrams

When the developers were asked to draw component diagrams, they were asked to draw one or two components that they felt they knew the best. Allowing them to pick what container to visualize in the component diagram was a decision taken because not all developers had been working in all parts of the code. Furthermore, the Minihydra system is also an older system, and most of the developers are nowadays working primarily on other systems, making the lower abstraction levels harder to reproduce.

There were mainly three different types of gaps found in the comparison between the component diagrams, namely, communication gaps, missing components, and labeling errors. Making a perfect diagram during an interview is hard, and even though the diagram in Figure 6.7 is quite a close representation of the CAN container, it still contains examples of all three types of gaps.

Communication gaps

Gaps regarding communication can when comparing component diagrams be, for example, communication between components that are missing, or if the wrong or no mean of communication was specified. In the interpreted diagram presented in

Figure 6.7, no means of communication is specified between the components *Wait HC* and *Can_tx*. The diagram is also missing communication through a buffer between the components *Can_tx* and *Can_rx*.

Missing components

An example of a missing component gap can be seen when comparing the diagrams in Figure 6.6 and Figure 6.7, namely that the component called *Event Task* is represented only in the reverse-engineered diagram. As the component manages interrupts, it is possible that the component is hardware-specific and thus only used in a selection of products running Minihydra. However, the missing component is still deemed to be a gap according to the definition used in this thesis, since the component itself is not present nor is its responsibilities included in another component.

Labeling errors

Describing the architecture of a system using the C4-model requires abstraction, and with that abstraction comes the possibility that two people describing the same system will not come up with the same representation. The abstraction work is dependant on the perception, interpretation, and imagination of the person creating the diagrams. One small detail that is likely to differ between two people's diagrams is the use of different labels, and to a large extent, that is very acceptable. There are however times where a label is wrong, and that is when it conveys incorrect information.

In the diagram shown in Figure 6.7 the container is labeled *CAN HE*. The HE label indicates that the whole container is one single task, when it is in fact made up of four different HEs, three of which are present in the diagram.

6.2 Developer insights regarding discovered gaps

After mapping and identifying the gaps between our reverse-engineered diagrams and the developers' interpreted diagrams, follow-up interviews were conducted with all the previously interviewed developers. In these interviews, the gaps discovered by comparing the interviewee's diagrams and the reverse-engineered diagram were presented in order to gain possible new insights about their causes and effects.

6.2.1 Communication gaps

There were several communication gaps discovered in the system and many of them were related to the router container. Most developers did not include the router container at all in their diagrams and the one developer who did had misunderstood how it was operating. As previously mentioned in Section 2.3.2, the Minihydra system is comprised of several HEs that are able to send messages to each other. The router plays a core part in this as it is responsible for updating the routing tables used by all other HEs when sending messages. Most developers believed that for a message to go from one HE to another it would first be sent to the router and then passed on to the receiving HE. The router is able to forward messages, however, this is not how the traffic normally is routed. After start-up, the first time a HE wants to send a message to another HE, the transmitting HE asks the router for the index of the receiving HE in the routing table. The transmitting HE will then store the index, and can thus send messages directly to the recipient using the routing table, without involving the router in future transmissions. This method works since all HEs in the Minihydra system share the same memory space and uses a shared routing table controlled by the router.

One developer speculated that their misunderstanding of the router might be because of the router name itself. "I thought that it actually worked like a router. Because what is a router? A router is something that forwards something to another destination. So I thought of it as some kind of crossroads where all messages meet but it is probably just a name with little overlap to the actual definition of the word.". This statement indicates that inaccurate naming of a software module may lead to misinterpretations of its responsibility. It is likely that the router used to route messages, since it still has the ability to do so, but that its responsibility has evolved while its name has remained, causing the confusion.

The optimization, passing all messages directly, is most likely due to performance requirements. If every message had to be passed through the router, each message from one HE to another would require first sending a message to the router, who in turn would send a new message to the receiving HE. Such an implementation would effectively double the amount of traffic in the system, while also making communication slower between HEs. Not knowing how the router is intended to work may thus lead to a suboptimal implementation. Even though the router has the ability to route messages, not sending direct messages would be slower and less efficient, hence jeopardizing the performance of the product.

Some HEs in the Minihydra system use a special form of messaging called broadcast. However, the broadcast message type implemented in the Minihydra system shares more similarities with a publish-subscribe architecture pattern than a broadcast architecture pattern. This is due to HEs interested in receiving broadcasts must first register themselves as listeners to a certain HE's broadcast messages. Most developers believed that broadcast messages were sent by the router and that the HE transmitting the broadcast only sent a message to the router who then would distribute the messages accordingly. In the implementation however a broadcast is quite similar to a normal message. The transmitting HE starts by looking at a routing table maintained by the router. This routing table declares all HEs wishing to receive broadcasts from the transmitting HE and messages are then sent directly from the transmitter to all HEs present in the table.

This gap is very similar to the one previously described and also suffers from matching causes and effects. Since the Minihydra broadcast feature does not behave as a common broadcast architectural pattern, there is a possibility that it is misinterpreted. If changes are made to the broadcast feature or features related to it, there is a risk that it could cause undesired behavior of the system or simply increase the time spent on understanding its implementation.

Another example of a communication gap was the absence of the LED communication to the user on the interpreted context diagrams. Most of the developers admitted that they simply forgot about this aspect of the system, or that they thought of the LEDs as hardware, and thus did not include them nor their communication to the user. However, one developer gave another reason “Now when I look at your diagram it feels obvious. I think the reason that I forgot about it might be that in these times (covid-19 pandemic) I am mostly working from home. I never see the LEDs anymore!”. When working from home the developers connect to their workstation at the office, and any products connected to this PC are out of sight from the developer. The developer’s statement reveals that working with the product out of sight can be a factor impacting a developer’s interpretation of the architecture.

6.2.2 Missing modules

A common type of gap found when comparing implemented and interpreted diagrams were modules missing from the interpreted diagrams. During the follow-up interviews, the developers were asked why the modules were missing, and common reasons were:

- Mixing up or combining responsibilities - Some developers mixed up or combined responsibilities of multiple modules and therefore only depicted one of them.
- Not knowing about the module - In some cases modules were left out of the diagram since the developer did not know about their existence.
- Not knowing it was part of Minihydra - Believing that the responsibility of a module was covered by the platform or OS, and not by a Minihydra module.
- Undervaluing the module’s architectural value - Thinking the module did not add any value to the architectural representation.
- Forgetting the module - Thinking that a module should have been included in their diagram but that they simply had forgotten to do so.

For example, several of the interviewed developers omitted to include a *Main* container and component in their diagrams. The developers that omitted it completely gave the reason to do so because it does not contribute to the system’s architecture much once the system is up and running. The developers that did include it in their diagrams instead argued that it is important since during the start-up phase, *Main* starts and configures all running HEs and tasks.

Two possible effects of a missing module were collectively established by the developers. The first effect is that further development may be more time-consuming, as the developer may have to spend time on investigating a, to him, unknown module. The second effect is that a developer may implement a new and redundant solution, due to not knowing the problem is already solved in another module.

6.2.3 Architectural knowledge gaps

During the interviews, several developers mentioned that there exists parts of the system where architectural knowledge has been lost. One developer expressed “In some places, there are implemented solutions that are very smart and really fast (as in performance) but you should avoid making changes to them because they are really hard to understand”. However, the developers argued that some of these implementations are necessary. Writing simple code should be the go-to method, but sometimes, for example, performance requirements in combination with restricted hardware and complex problems require more complex solutions. These solutions may execute much faster but are much harder to understand through code inspection.

The developers also talked about code that had been written by employees that are no longer working at the company. Over the years, as developers were switched out, the knowledge of exactly how some implementations work was lost, but the perception of what the implementations do remains. Some of the interviewed developers said that these sections of code can be treated as “black boxes”, as they are thoroughly tested and have largely remained unchanged.

Common for the complex solutions and the code by prior employees is the lack of documentation, making the code harder to work with. New implementations in these modules would first require tedious code inspection to recover the knowledge of how the code works, increasing development cost in terms of time spent.

6.2.4 Labeling errors

As previously explained in Section 2.3.2 the Hydra Entity (HE) is an integral part of the Minihydra software system. The source code of Minihydra shows that a HE is a task with extra features and attributes, however, when talking to the developers and looking at the documentation several other definitions arose.

The documentation of the system states that HEs are “... units that have a specified purpose and means to communicate with the outside world”. When the developers were presented with this definition they all either disagreed with it or thought it was too vague to the point of not being helpful. One developer said “That could be anything. I feel that it could describe anything from an LED to a car. It is true, but it is not really helpful.”.

Even though comparing the software architecture against the documentation of the system was not in the scope of this thesis it provided the developers with a good starting point for reflection about the HEs. It was discovered during these reflections that the term HE could have different meanings to the developers based on context. When asked to describe a HE most developers described it as a software task with added functionality, such as being able to transmit messages to other HEs. However, when creating the diagrams the developers tended to use HE as a label for a responsibility, for example, the HE responsible for the CAN communication or the HE responsible for the USB communication. This usage gives HE a new contradicting meaning since both of the mentioned responsibility areas are handled by several different HEs. This description of HEs seems to be a way for the developers to abstract the system to what we in our diagrams call containers. When asked about this the developers agreed that the definition was fuzzy and gave several reasons why.

One possible reason relates back to when the initial system's architecture was created. Back then a plan was made and the HEs were defined based on their inputs, outputs, and responsibility. However, over time, as requirements changed and new challenges arose this idea was adjusted, and the implementation was changed. An example of this could be the container responsible for the CAN communication. From an architectural point of view, it is reasonable to think of the CAN HE as some sort of module with clear inputs, outputs, and responsibility. However, sending and receiving messages are two very different activities. Therefore, it made sense from a performance standpoint to split up the responsibility between two HEs.

Another reason to why the original idea of the HE changed was that if a task was made into a HE, it gained access to features that made it easier to implement. One developer theorized that some components that shouldn't be HEs have been made so solely to be able to use some of the HEs software features, such as debug-printouts. According to the developer, the reason was probably a mix of not being knowledgeable enough about the architecture of the system as well as being lazy in the sense that they were trying to fix a problem or implement a feature as fast and easy as possible.

The final reason given for the confusion regarding the HE was its name. One developer described that since the name states the unit is an entity, you become tempted to think that it is one independent unit when it is actually not. Using the example of the CAN container again, it is comprised of three HEs *Can HE*, *Can_tx HE* and *Can_rx HE*. None of these can function correctly without each other but when the developers describe the CAN area of responsibility for the system they say *the Can HE* when they actually mean all three HEs.

One effect of the conflicting HE definitions is that it gets harder to understand the system, especially if you have no prior experience working with it. Furthermore, the system becomes less modular since some HEs depend on each other and you can no longer swap them in our out as easily. With the system being less modular more extensive testing is also required since when changes are made more parts of the system may be affected due to increased coupling.

In summary, possible causes for the gap regarding the definition of HE were according to the developers:

- The need for better performance which forced some HEs to split into several HEs.
- Lazy solutions, taking shortcuts and producing implementations violating the overall system architecture. An example of this could be to change a task into a HE only to make it easier to debug.
- Misinterpretations regarding the name Hydra Entity.

7 Discussion

This chapter contains discussion about the methods, results, and challenges of this study. Section 7.1 describes the reverse-engineering of the software architecture. Section 7.2 discusses architectural differences that were not considered to be gaps. Section 7.3 presents the challenges of the initial research questions and how they were adjusted. Section 7.4 explains the state of the existing documentation of the Minihydra system. Section 7.5 presents some of the feedback from the participants of the study and Section 7.6 highlights the validity threats of the study as well as procedures to mitigate them.

As previously stated in Section 5.2, this study has due to its nature, being a qualitative case study, not resulted in any statistically significant result. However, the result of the study may still be valuable to both practitioners as well as other researchers. Common for all the effects identified in this thesis is that they are all negative effects, and thereby indicate that minimizing or avoiding gaps is desirable. The method used in this thesis can be applied to other systems to identify gaps, causes, and effects specific to them, and thereby enabling patching of existing gaps as well as preventing new gaps from arising.

Due to only studying one single system at one single company no generalizations can be made for gaps in other systems regarding, for example, their type or their size. However, being aware of the causes identified in this study may mitigate the risk of similar gaps arising in other systems.

7.1 Reverse-engineering thoughts and process

To be able to compare the implemented architecture to the interpreted architecture, the implemented architecture had to be extracted from the source code. There are several automated tools available exactly for this purpose, lessening the manual workload, however, none of the tools we tried seemed to provide usable results in this case. Enterprise Architect [31], srcML [32] (in combination with srcUML [33]), and StarUML [34] were three of the reverse-engineering tools tested, but it became evident that none of these tools would provide a plug-and-play solution for the software system under investigation. One reason to this was that the system under investigation was predominantly written in C, and C was not well supported when it comes to reverse-engineering software tools.

An attempt was also made to develop some kind of reverse-engineering tool ourselves, adapted especially for C. The idea was to start with the initialization file of the system, note down all its file dependencies, and then examine them one by one in the same manner. One problem early on was the duplicate filenames existing within the system, due to differences between the products using Minihydra, a problem normally handled by the build system including the correct files for the specific product. In an attempt to keep the developed reverse-engineering software more general, and not build a reverse-engineering tool only usable on this particular system, using the build files was not a very desirable solution. Instead, files with identical names not included by the build system when building for the product under investigation were excluded prior to running our reverse-engineering tool.

However, in C, a file dependency can be included but does not have to be used, hence staying on file-level would only show included dependencies and not used dependencies. The next step would have been to go down from file level to function level, by tracking each function call and locating the function in the dependency tree, thus being able to produce more accurate dependency diagrams. However, the development was halted as we came to realize that this method would only be able to give us dependency relations, and the interpreted architecture was much more likely to be based on data flow rather than dependencies.

7.2 Other differences between architectural views

The definition for gaps used in this thesis only covers differences between the implemented and the interpreted architectures, but occasionally during the thesis work differences between other architectural views were noted. As these differences do not fit the definition of a gap they are not a part of the result of the study, but can still be considered interesting findings.

7.2.1 Differences between interpreted architectures

The interpreted views from two different developers could sometimes be contradicting or show differences between each other. One example of such a difference is the router container, as one of the developers thought it was crucial for sending messages between the other Hydra Entities (see figure in Appendix A.3), and two of the other developers did not include the router at all (see figure 6.4 and Appendix A.2).

These differences between the different interpretations by the developers are not considered gaps in this thesis but proves that the views between the developers are not unanimous and that each developer has his own conceptualized view of the system.

Furthermore, in this study, software architectural gaps were discovered by mapping each individual developer interpreted diagram to its reverse-engineered counterpart.

Another possible way of finding gaps would be to analyze the company's interpretation of the architecture as a whole against the implemented one. A study like this would first have to collect interpretations of the architecture from developers at the company and then create a combined interpretation that can be compared to the implemented architecture. It is possible that doing it this way would result in fewer, but more prominent, gaps since possible outliers could be ignored and reoccurring gaps could be more accurately specified. This was something that was discussed early in this study but rejected since it would require a larger sample size of developer interpreted diagrams than was possible given the scope of the study.

7.2.2 Differences between intended and interpreted architecture

During some interviews, examples came up where the developers presented how the system was implemented, and then presented how it was meant to be implemented at the time. These meant-to-be solutions are no longer documented, but sometimes seemed to be the more thought through solution.

An instance where this occurred was when the diagram in Figure 6.7 was drawn. The diagram features a component that is not included in its reverse-engineered counterpart (seen in Figure 6.6), namely *canHandler*. The component also has a description of its purpose, namely being a middle layer between the hardware and the other software components. The *canHandler* component was according to the interviewee meant to be the only component that needed to be replaced when the CAN hardware was changed, while all the other CAN-related components was to stay unchanged.

For one reason or another that was not how the implementation ended up. The *canHandler* component is replaced when switching hardware, as well as a few of the other CAN components. Having a high degree of shared code between the different products is preferred as it makes the code easier to maintain, where as now a discovered bug can need fixing multiple files with only minor differences between them.

Beyond being outside of this thesis definition of a gap, this thesis also only covers one product, hence one hardware setup. Due to said reasons, the discovery was not investigated thoroughly by us, but it was confirmed that it was not only the *canHandler* that was changed between hardware alternations. No time was put into investigating underlying technical reasons, and the explanation given by the developer was that he went on parental leave and came back after the implementation was done. This begs the question if this solution came to be due differentiating interpretations of the intended architecture or if there were underlying technical reasons, to which we can only speculate.

7.3 Initial research question

This thesis was initially intended to be a case study of architectural erosion, with the goal of comparing the documented architecture to the implemented architecture. Before the work was started, the existing documentation was presented to us in the form of a 43 pages long PDF document. Much of what was documented was however not applicable to the product under investigation in this thesis, ruling out many of the pages. A common theme throughout the documentation was the low level of abstraction provided, making it hard to form an overview of the system.

It was thus decided that to complement the pre-existing documentation, we were to interview developers for their view of the systems architecture. We soon came to the realisation however that if we were to gather knowledge from the developers themselves, they would be influenced not only by the intended architecture, but also from the implementation. Their view of the architecture would no longer be a representation of the intended architecture, and most likely not a perfect representation of the implemented architecture, but rather something in between, their interpretation.

Comparing the implemented and the interpreted architecture most likely resulted in fewer and/or smaller gaps, as the interpreted architecture is a mix between the intended and the implemented architecture, hence reducing any differences. Switching to a comparison with the interpreted architecture was a interesting angle, and also ruled out the need for a comprehensive documentation of the system.

Asking questions regarding a specific gap's causes and effects was also more meaningful and prone to give better answers as the developers explained and interpreted gaps from their own interpretation and not from what was documented sometime during the last 10 years by someone else. After gaining insight into both the implemented and interpreted architecture, we have found things in the architectural documentation that was no longer representative of the system.

Comparing the documented architecture to the implemented architecture would have come with other challenges. The architecture gathered by reverse-engineering contains a lot more knowledge on higher abstraction levels (for example relationships between Hydra Entities) that are not specified in the documentation. At the same time the reverse-engineered architecture contains a lot less knowledge on lower abstraction levels (for example Hydra Commands bit by bit), and verifying the low abstraction level information would be very time consuming, and not feasibly given the scope of the thesis.

7.4 The existing Minihydra documentation

There is some existing documentation to accompany the Minihydra system regarding its architecture, and in this section, we will briefly discuss it and discussions we have had regarding it with the developers. The existing documentation of Minihydra contains a lot of details regarding the system, but does not offer the reader an overview of the system. This renders the details very hard to understand and contextualize for someone not used to the project setting. When writing this thesis, we have not used or read the documentation in detail, but have found a few places where the documentation does not reflect the implementation, at least not for the product we investigated.

When discussing the documentation one of the interviewees said that having documentation that was up to date, correct, and easy to read would reduce the risk of misunderstandings, but at the same time thought that it would require too much work keeping that documentation up to date. A lot of time was put into producing the documentation that exists today, but it gradually gets more and more outdated as the software changes. The interviewee suggested that it would perhaps be a better idea to only present a highly abstracted overview of the system, and leave all details out of the documentation. By doing so, the documentation would be easier to read and also less likely to become outdated, as it would only describe the system in broad terms. Furthermore, the interviewee also thought that having a detailed but outdated documentation of a system would create a false sense of security and increase the risk of misunderstandings than having no documentation at all. "... because in that case people would refer to it(the documentation) and get ideas instead of looking at the actual code.".

During the first round of interviews, the interviewees were asked if they had read the documentation about the system. One of the developers had not read it, another said that the documentation was written after he had been working on the system for a long time and had thus not read it. One of the developers said that he had tried to read it, but that it was very hard to read as a newcomer.

One developer said that he learned about the architecture by working on the system, and when starting out he was referred towards the correct files by developers with more experience regarding the system. Relying on having a few individuals with key insight into the architecture of the system like this may slightly mitigate the need for a better-documented system overview, but also makes the company very dependant on a few of their employees. The developer that started working in this way said that it resulted in the formation of a *local view*, a subset of architectural knowledge about the system, that then grew over time as more information was gathered through working on the code. This particular developer's architectural knowledge of the system is now quite sufficient, and the context, container, and component diagrams drawn by this developer were all used to locate gaps.

The interpreted views of the Minihydra system from three of the developers were previously used to find gaps in Section 6, but the diagram from the last interviewed developer can be seen in Figure 7.1. The diagrams previously presented all roughly followed the C4-model, and were thus comparable, however, the diagram drawn by the last developer does not. The last diagram instead supports the theory of forming local views of the system while working on it and shows that the developer's architectural knowledge of the system was limited to the parts he had been working

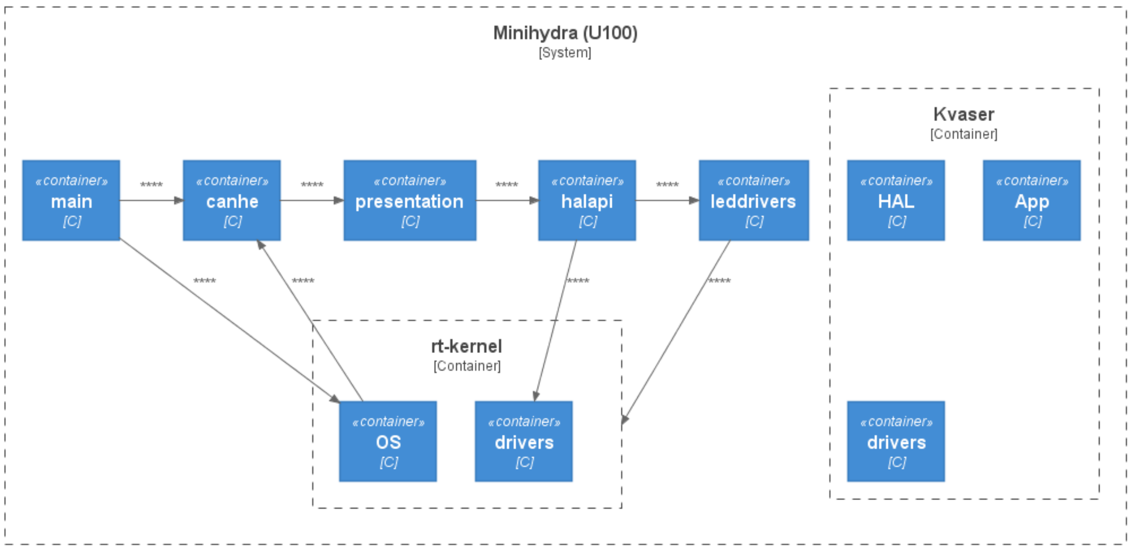

Figure 7.1: The Minihydra system described by one of the developers.

on, namely the LED lights on the product.

Some of the containers in the diagram can be mapped towards containers in the reverse-engineered container diagram, featured in Figure 6.3. They both have a *main* (responsible for startup and configuration), *CAN* (responsible for CAN-communication) and *Presentation* (responsible for LED lights on the product) container, however that is where the similarities ends. In the reverse-engineered diagram, the containers represent areas of responsibility, but the diagram in Figure 7.1 instead represents a flow of information between different files, all the way from start-up to the driver. In the reverse-engineered diagram, the drivers and hardware access layer are both omitted due to abstraction, and tucked into the *Presentation* container.

The developer has acquired a local architectural view of the system, centered around the LEDs of the product. The developer proved to have plenty of knowledge regarding how the LEDs on the product are controlled, but quite limited knowledge of how the architecture of the system is outside of this local view. Working on a system with only limited knowledge about its architecture may entail bad solutions from an architectural point of view, due to a lack of overall perspective.

7.5 Participant feedback

Both during and after the interviews we received feedback from the participants regarding the study. Most participants said that they had learned new things about the system and that some of their misconceptions had been corrected. One thing that was found particularly beneficial was the high-level architecture diagrams that had been created through reverse-engineering, several developers even asked for copies. However, some concerns were raised that our method of finding gaps might not be efficient enough to be used in the development process. Reverse-engineering an architecture and eliciting interpreted architectures from developers takes a considerable amount of time and if it is done during development the risk is quite high that the architecture descriptions gathered quickly becomes outdated. Therefore, we suggest that our method should not be used in the development phase of a project but instead at a later stage to discover any overlooked software architectural gaps and to gain knowledge about their causes and effects that can be used in upcoming projects.

7.6 Validity threats

There are several threats to validity regarding a study like this, and in this section foreseen threats are discussed, as well as the different procedures used to minimize them where it was possible.

7.6.1 Reverse-engineering the system

Reproducing an architectural model of the system from its source code involved a lot of manual work: reading code, keeping track of register values, documenting findings, abstracting findings, and finally presenting the completed model. As with all manual work, mistakes can occur, but to mitigate this risk we chose to work with one of the simplest products running the Minihydra system, limiting the size of the analyzed software system. Working with a smaller system allowed for a more detailed and thorough investigation The two of us could, on complex parts of the system, analyze the code in parallel in order to reduce the risk of missing any tasks or messages. What HEs the product was running was also verified using debug printouts, ensuring that no running HE had been missed during the reverse-engineering stage.

Abstraction is also based partly on preference and could be done in many other ways. To produce clean and readable diagrams we decided to include only separate tasks and their respective communication in the component diagrams, and group them according to responsibility to form containers. How the developers choose to abstract their diagrams was not taken into account when looking for gaps as it is just a matter of perspective, and there is no right or wrong way to abstract the architecture. However, introducing the C4-model encouraged a more unanimous way of abstracting the system.

The diagrams that were produced of the Minihydra systems architecture through reverse-engineering are however not representative of the system as a whole, as we only analyzed the parts forming the firmware for the particular product under investigation. The truly implemented architecture of Minihydra is thus more complex than our diagrams show, however as the other products are not within the scope of this study, the representation produced is deemed sufficient for our purpose.

7.6.2 Interviews and elicitation

When conducting interviews with the developers, care was taken not to ask leading questions and guiding their answers in order to minimize the risk that our own ideas were influencing their answers. During the first interview where they were asked to describe the architecture of Minihydra, they were not informed of which product was under investigation in this thesis. Not informing the interviewee of what product was under investigation may seem like an unnatural decision to make, but the products running Minihydra all have great similarities (they are all CAN-interfaces, based on the same architectural ideas), and in the worst case the diagram would have a few extra containers or components due to that we are working with a simpler product. At the same time, not telling them which product was investigated allowed them to include everything they wanted in the diagram, not hesitating if it was included in the specific product or not. The thought was to rather gain too much information than too little. For a developer to double check if a specific file is included or not in a product's firmware requires little effort, but doing so by the whiteboard was less convenient. Thus extra tasks or HEs on their diagrams are not considered to be gaps.

Before letting the interviewees draw their own view of the system's architecture, they were first shown an example system to teach them the basics in using the C4-model. The example system was unrelated to Minihydra and depicted a much larger system, more precisely an internet banking system, which can be seen in Figure 2.2, 2.3, and 2.4. We do not think that showing them an unrelated example system and asking them to use the C4-model pushed them into thinking about the Minihydra system differently. Establishing a common way of depicting software architecture was simply needed in order to be able to compare the diagrams in a just and reliable manner. The C4-model was specifically selected because it is easy to understand and get started with even for someone not used to working with architecture, and apart from its four levels of abstraction it allows for much freedom in how the diagrams are created.

In order to ensure construct validity during the interviews, we tried to avoid misunderstandings by asking the interviewees to further explain or clarify if something was not fully clear or if doubt arose that we and the interviewees interpreted the questions in the same way. The interviews were also recorded in order to be able to listen in again in case uncertainties arose after the interviews.

7.6.3 Reproducibility of the study

Since this study was performed on a proprietary software system (owned and distributed by Kvaser) and in close collaboration with the developers of that system the reproducibility of this exact case study would be tricky. Even if given access to the source code under analysis in this study, the interviewees have still learned about the system through our interviews, changing their interpretation of the system. As previously mentioned in Section 7.6.1, abstraction during the reverse-engineering process is partly depending on the preference, hence two people may come up with two different diagrams based on the same information. To mitigate this effect, different levels of abstraction were not considered to be a gap in thesis.

7.7 Future work

By following the same method as us, a similar case study could be reproduced on another software system where the developers are willing to partake in such a study. While some of the questions asked by us during the first interview could be used directly on another system, other questions were specific for the Minihydra system, such as questions regarding the initial hardware configuration of the products. Since software architecture erosion is inevitable [12], differences between intended and implemented architecture will eventually arise, and since interpreted software architecture is a mix between the two, a study like this one can be conducted on almost any system.

Conducting a study like this on another type of system may provide additional insights such as new types of gaps with different causes and effects. Since the Minihydra system contains some hardware aspects, it is possible that this study may have resulted in the discovery of additional or other types gaps compared to if the study would have been conducted on, for example, a pure software system. Selecting a system to analyze written in a language well supported by reverse-engineering tools could allow for analysis of a larger system or a more in-depth investigation in the allocated time frame.

In this study, the differences between the implemented architecture and the interpreted architecture were studied. If studying a system with a well-documented architecture, the differences between how the developers think the system should be implemented and the documented architecture could instead be investigated. Software architecture is prone to interpretation [23], and conducting such a study could result in concrete examples of misinterpretations as well as their causes and effects.

8

Conclusion

This case study aimed to answer two different research questions. The first question was if there were any software architectural gaps present in the Minihydra software system when comparing the implemented architecture against the developer's interpreted architectures. The second question was to gather insights from the developers of said system about the causes and effects that any of the discovered gaps might have. The first challenge was to reverse-engineer the implemented architecture of the Minihydra system. Several different software tools were used to gain knowledge about the system but in the end, all reverse-engineered diagrams were made manually through code inspection. The reasoning behind this decision is discussed in Section 7.1. Next, the interpreted software architecture of the system was elicited from developers of the system via interviews. Software architecture can be portrayed in many different ways so in order to get comparable diagrams the developers were first introduced to the C4-model, which is the model that was used to create all architectural diagrams in this study. By comparing the implemented architecture with the interpreted, several software architectural gaps were discovered, answering RQ1.

The discovered gaps were used as points of discussion in follow-up interviews with the developers where they shared their insights about possible causes and effects of these gaps. Found causes and effects include the following:

Causes:

- **Misinterpretations due to naming** - Poorly named software artifacts can cause developers to misinterpret their functionality and thereby implement solutions that violate the software architecture.

- **Lazy solutions** - Solutions written by developers that violate the system's architecture. Either because the developer lacked architectural knowledge or that the developer chose to implement a quick and dirty solution instead of a proper one.

- **Lost architectural knowledge** - Instances where architectural knowledge has been lost. For example, when employees that had architectural knowledge no longer remain within the organization or that new solutions and implementations are not documented.

- **Lack of high-level documentation** - Without a high-level documentation it is hard for developers to grasp the software system's architecture, increasing the risk of misinterpretations.

Effects:

- **Jeopardizing performance** - Insufficient architectural knowledge may lead to slow and ill-performing solutions being implemented.
- **Less modular system** - Gaps may lead to bad implementations, creating undesired dependencies and thereby threaten the modularity of individual modules.
- **Increasing implementation time** - Not possessing the required knowledge regarding the system can increase implementation time, especially if the information has to be accumulated by analyzing the source code.
- **Redundant solutions** - Not knowing that a particular problem has already been solved can lead to the implementation of redundant solutions.

Bibliography

[1] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *2007 Working IEEE/IFIP conference on software architecture (WICSA'07)*, IEEE, 2007.

[2] L. De Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, 2012.

[3] S. Brown, "The c4-model for visualising software architecture." https://c4model.com/. Accessed: 2021-01-16.

[4] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Towards recovering the software architecture of microservice-based systems," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, IEEE, 2017.

[5] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice.* Addison-Wesley Professional, 2003.

[6] M. De Silva and I. Perera, "Preventing software architecture erosion through static architecture conformance checking," in *2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS)*, IEEE, 2015.

[7] M. Shaw and P. Clements, "The golden age of software architecture," *IEEE software*, vol. 23, no. 2, pp. 31–39, 2006.

[8] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[9] L. Hochstein and M. Lindvall, "Combating architectural degeneration: a survey," *Information and Software Technology*, vol. 47, no. 10, pp. 643–656, 2005.

[10] J. Bosch, "Software architecture: The next step," in *European Workshop on Software Architecture*, pp. 194–199, Springer, 2004.

[11] N. Medvidovic, A. Egyed, and P. Gruenbacher, "Stemming architectural erosion by coupling architectural discovery and recovery.," in *STRAW*, vol. 3, 2003.

[12] J. Van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of systems and software*, vol. 61, no. 2, 2002.

[13] T. Haitzer, E. Navarro, and U. Zdun, "Reconciling software architecture and source code in support of software evolution," *Journal of Systems and Software*, vol. 123, 2017.

[14] A. Baabad, H. B. Zulzalil, S. B. Baharom, *et al.*, "Software architecture degradation in open source software: A systematic literature review," *IEEE Access*, vol. 8, pp. 173681–173709, 2020.

[15] B. Meyer, *Agile! Bertrand MeyerThe Good, the Hype and the Ugl.* Springer, 2014.

[16] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, "Recommending refactorings to reverse software architecture erosion," in *2012 16th European Conference on Software Maintenance and Reengineering*, IEEE, 2012.

[17] A. Albert *et al.*, "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems," *Embedded world*, vol. 2004, pp. 235–252, 2004.

[18] B. Randell, *Software engineering in 1968.* Citeseer, 1979.

[19] D. L. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*, pp. 279–287, IEEE, 1994.

[20] F. A. Fontana, R. Roveda, M. Zanoni, C. Raibulet, and R. Capilla, "An experience report on detecting and repairing software architecture erosion," in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE, 2016.

[21] J. Rosik, A. Le Gear, J. Buckley, M. A. Babar, and D. Connolly, "Assessing architectural drift in commercial software development: a case study," *Software: Practice and Experience*, vol. 41, no. 1, pp. 63–86, 2011.

[22] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, 2001.

[23] P. Bengtsson, "Design and evaluation of software architecture," Software Engineering and Computer Science, University of Karlskrona/Ronneby, 1999.

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

[25] S. R. Tilley, S. Paul, and D. B. Smith, "Towards a framework for program understanding," in *WPC'96. 4th Workshop on Program Comprehension*, IEEE,

1996.

[26] G. Y. Guo, J. M. Atlee, and R. Kazman, "A software architecture reconstruction method," in *Working Conference on Software Architecture*, Springer, 1999.

[27] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[28] S. E. Hove and B. Anda, "Experiences from conducting semi-structured interviews in empirical software engineering research," in *11th IEEE International Software Metrics Symposium (METRICS'05)*, pp. 10–pp, IEEE, 2005.

[29] K. Charmaz, *Constructing grounded theory: A practical guide through qualitative analysis.* sage, 2006.

[30] J. Saldaña, *The coding manual for qualitative researchers.* sage, 2021.

[31] Enterprise Architect. [Online]. Available: https://sparxsystems.us/software/enterprise-architect/. [Accessed: 08-Aug-2021].

[32] srcML. [Online]. Available: https://www.srcml.org/. [Accessed: 06-Aug-2021].

[33] srcUML. [Online]. Available: https://github.com/srcML/srcUML. [Accessed: 06-Aug-2021].

[34] StarUML. [Online]. Available: https://staruml.io/. [Accessed: 08-Aug-2021].

A

Appendix 1

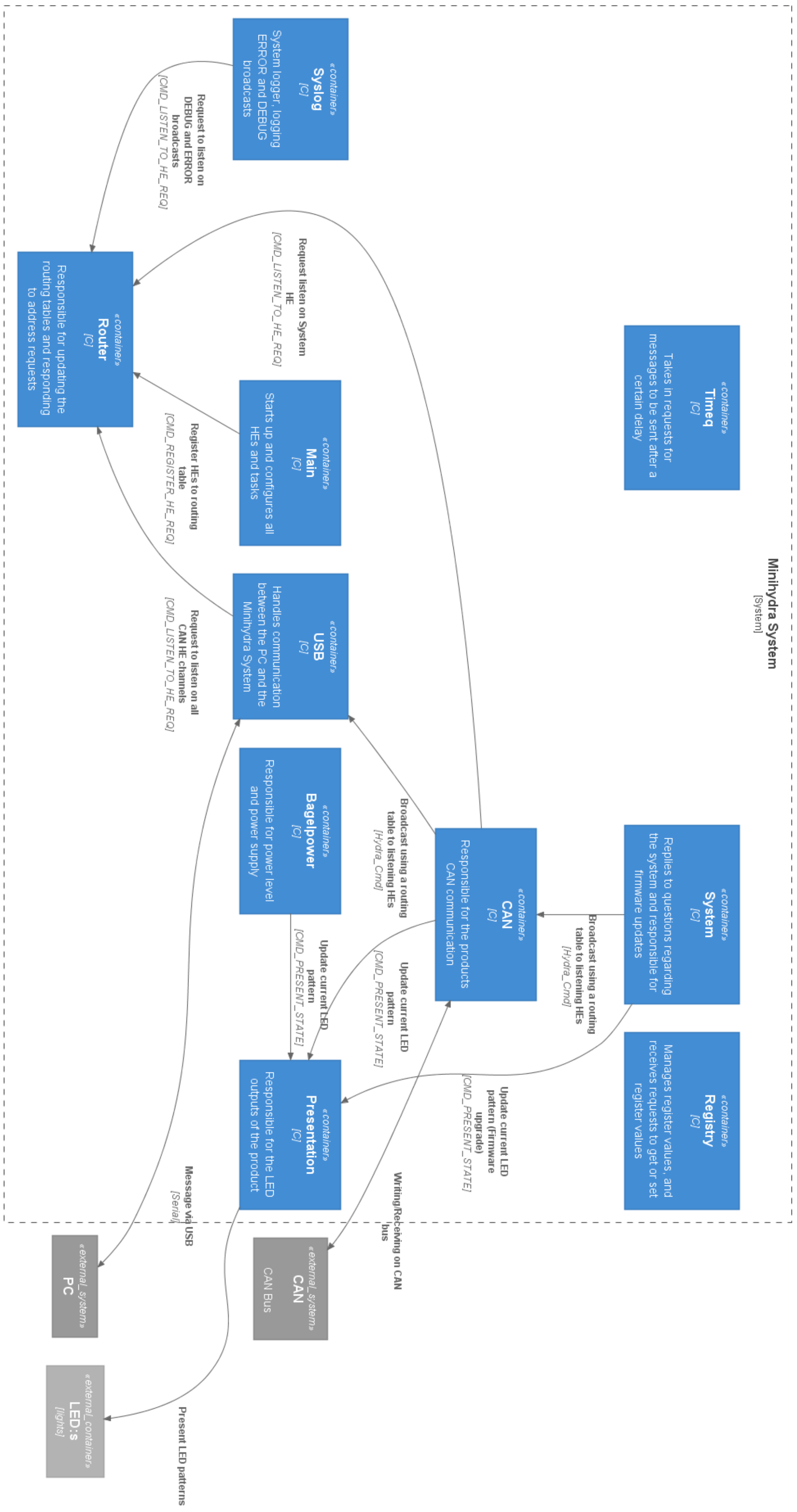
Minihydra System
[System]
«container»
Timeq
[C]
Takes in requests for messages to be sent after a certain delay
«container»
Syslog
[C]
System logger, logging ERROR and DEBUG broadcasts
Request to listen on DEBUG and ERROR broadcasts
[CMD_LISTEN_TO_HE_REQ]
«container»
Router
[C]
Responsible for updating the routing tables and responding to address requests
Request listen on System HE
[CMD_LISTEN_TO_HE_REQ]
«container»
Main
[C]
Starts up and configures all HEs and tasks
Register HEs to routing table
[CMD_REGISTER_HE_REQ]
«container»
USB
[C]
Handles communication between the PC and the Minihydra System
Request to listen on all CAN HE channels
[CMD_LISTEN_TO_HE_REQ]
«container»
Bagelpower
[C]
Responsible for power level and power supply
Broadcast using a routing table to listening HEs
[Hydra_Cmd]
«container»
CAN
[C]
Responsible for the products CAN communication
«container»
System
[C]
Replies to questions regarding the system and responsible for firmware updates
Broadcast using a routing table to listening HEs
[Hydra_Cmd]
Update current LED pattern
[CMD_PRESENT_STATE]
Update current LED pattern
[CMD_PRESENT_STATE]
«container»
Presentation
[C]
Responsible for the LED outputs of the product
Update current LED pattern (Firmware upgrade)
[CMD_PRESENT_STATE]
«container»
Registry
[C]
Manages register values, and receives requests to get or set register values
Message via USB
[Serial]
Writing/Receiving on CAN bus
«external_system»
PC
«external_system»
CAN
CAN Bus
«external_container»
LED:s
[lights]
Present LED patterns

Figure A.1: Container diagram showing the implemented software architecture of Minihydra.

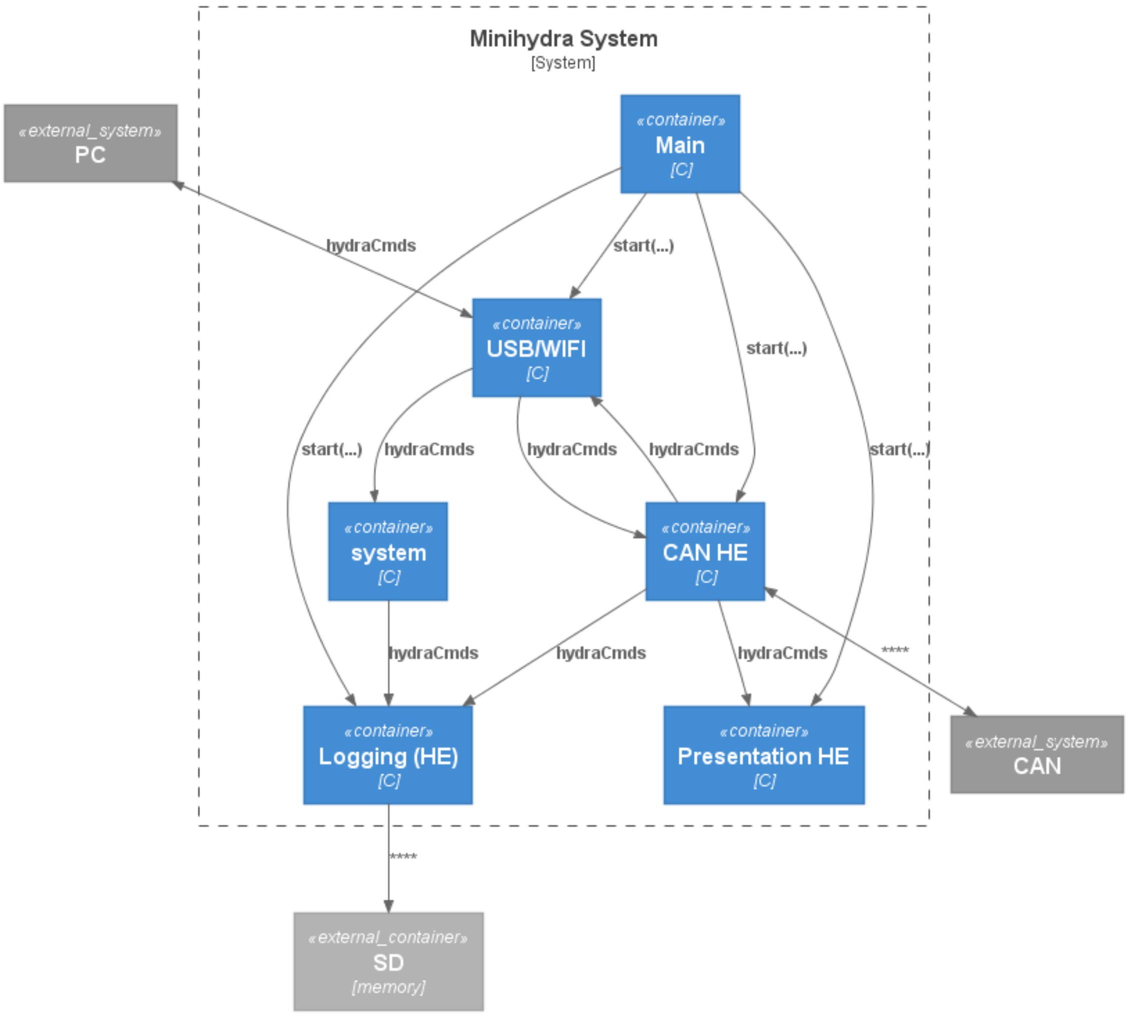

Figure A.2: Container diagram showing an interpreted software architecture of Minihydra.

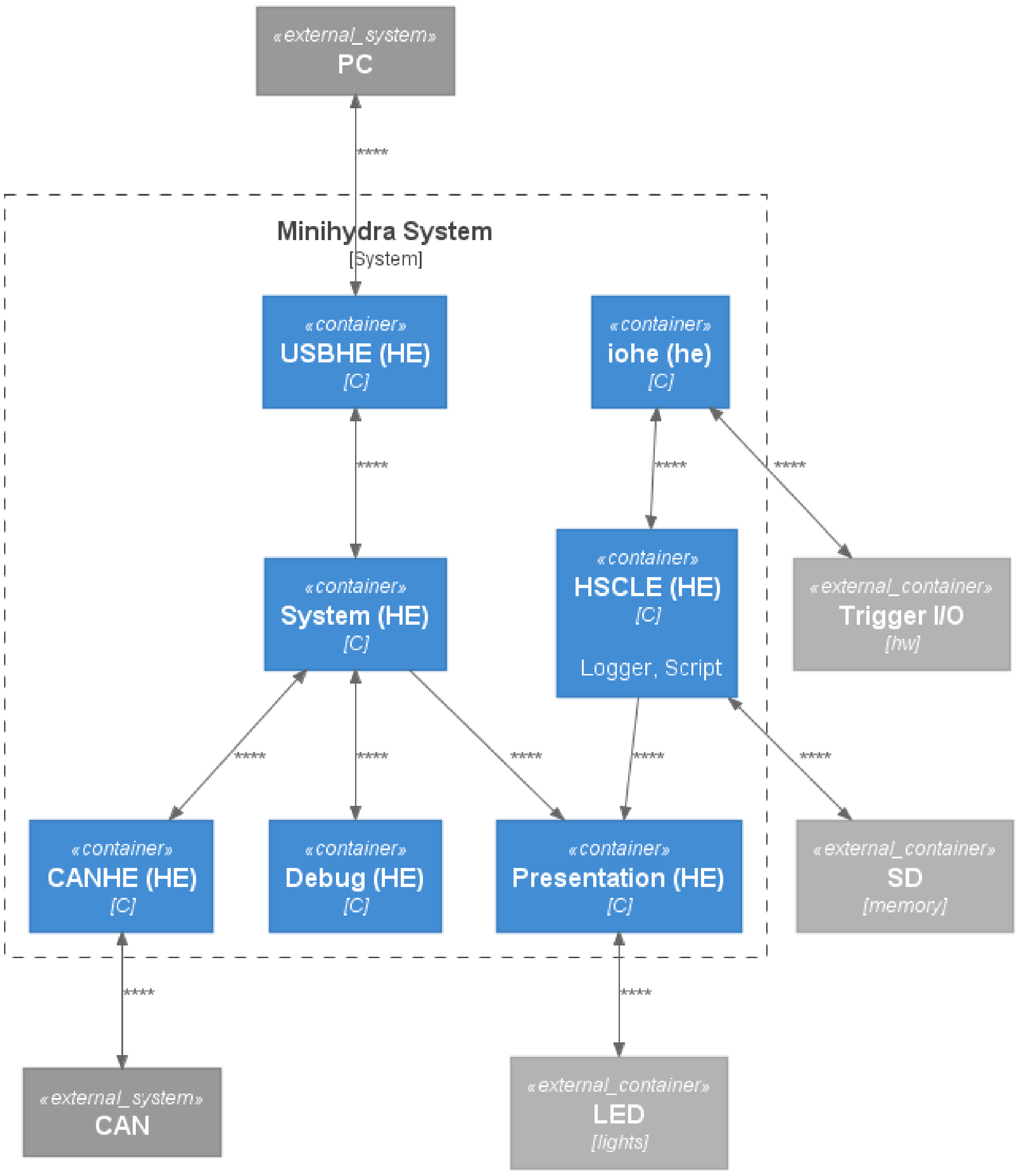

Figure A.3: Container diagram showing an interpreted software architecture of Minihydra.

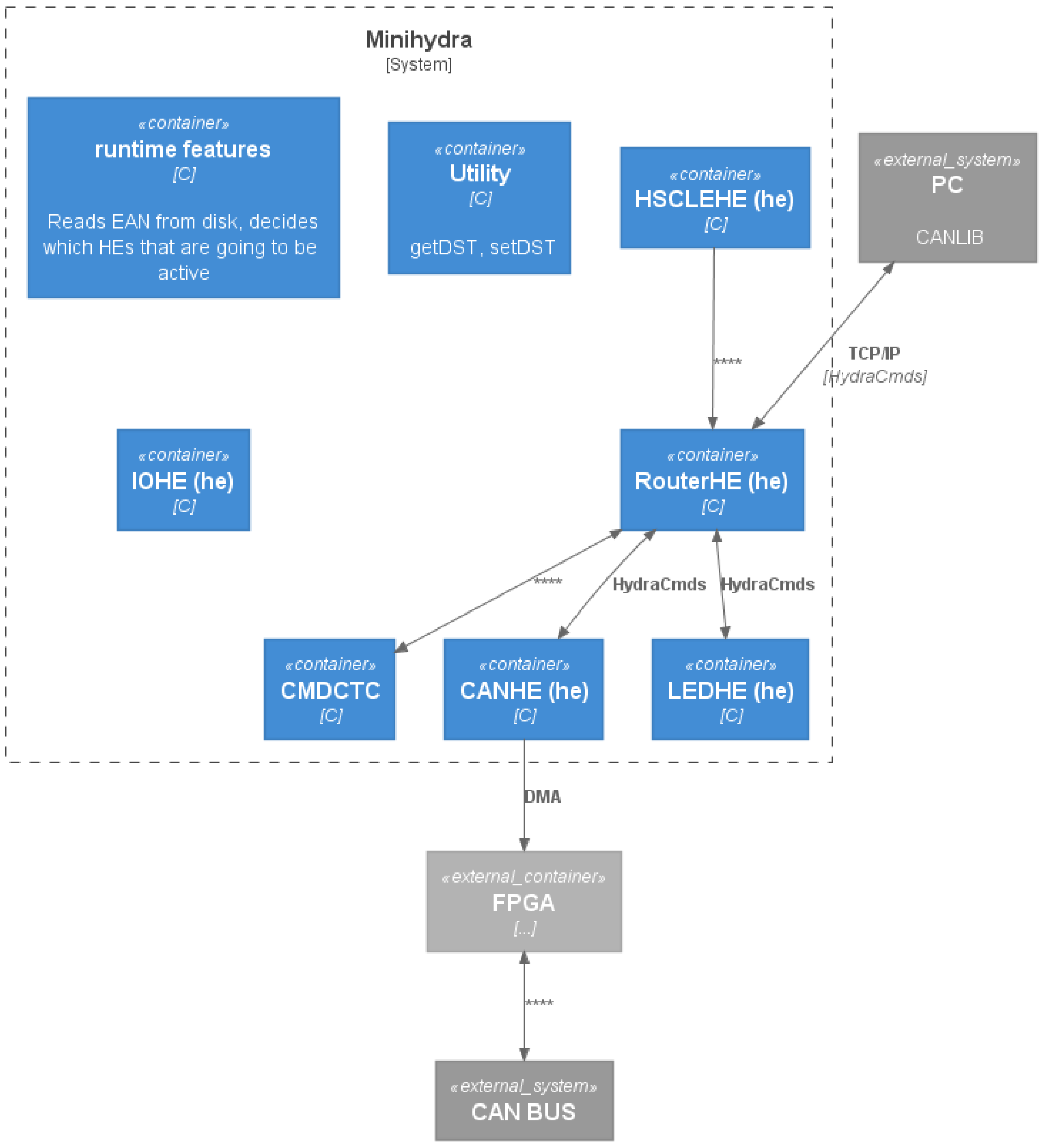

Figure A.4: Container diagram showing an interpreted software architecture of Minihydra.