Towards automatically generating explanations of software systems

Generating explanations of a web-template system in different abstraction levels

Alex Tao and Mahsa Roodbari

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Towards automatically generating explanations of software systems

Generating explanations of a web-template system in different abstraction levels

ALEX TAO
MAHSA ROODBARI

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer science and engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Towards automatically generating explanations of software systems
Generating explanations of a web-template system in different abstraction levels
ALEX TAO, MAHSA ROODBARI

Supervisor: Michel RV Chaudron, Department of Computer Science and Engineering
Examiner: Robert Feldt, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-421 39 Gothenburg
Telephone +46 72 333 1722

Typeset in LaTeX
Gothenburg, Sweden 2019

Towards automatically generated explanations of software systems
Generating explanations of a web-template system in different abstraction levels

ALEX TAO
MAHSA ROODBARI

Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Traditional software documentation is often challenging to manage as its content grow. For example, information becomes scattered, information become hard to retrieve and the documentation has to satisfy multiple different stakeholders. The aim of this study was to find ways to address these challenges. It followed a design science research approach, with one case used for demonstration and evaluation. A tool, System Explanation Composer (SEC), was built to explore how software artifacts and architectural knowledge (AK) could be interrelated and how different aspects of the case system could be explained and presented. A meta-model which presents software artifacts and AK was designed. This meta-model interrelates knowledge in the following four categories; requirements, architecture, rationale, and implementation. SEC generates explanations to a selected set of commonly asked development related questions using this meta-model. These explanations were evaluated using a quantitative and a qualitative approach. Although the sample size of the evaluation was too small for making any generalizable statements, the quantitative results strongly indicate that participants in this study solved development-related tasks faster, more accurately and more independently using SEC as opposed to using file-based documentation. The qualitative results also support this statement and indicate that SEC could further assist developers in performing their daily tasks more efficiently after improving two main aspects: the behavior and rationale sections of the meta-model and the query system of SEC.

Keywords: software, explanation, architectural view, requirement, architecture, rationale, implementation, ontology, artifacts, architectural knowledge

Acknowledgements, Mahsa Roodbari

Firstly I would like to express my sincere gratitude to our supervisor, Professor Michel R.V Chaudron for his continued support and guidance throughout our master thesis study, for his motivation and valuable input and knowledge. Besides our supervisor, I would like to thank his Ph.D. assistants Truong Ho Quang and Rodi Jolak and our opponent Humberto Linero for their constructive feedback and reviewing our work. My sincere appreciation also goes to all those friends and colleagues who generously spent their time to participate in our interviews and provided us with a helpful evaluation. Finally, I would like to extend my gratitude to Professor Robert Feldt, our examiner, for his insightful comments and the hard questions which encouraged us to improve our research from various perspectives.

Mahsa Roodbari, Gothenburg, February 2019

Acknowledgements, Alex Tao

I would first and foremost like to extend my gratitude towards my academic supervisor, professor Michel R.V Chaudron and his Ph.D. assistants, Truong Ho Quang and Rodi Jolak. Their guidance and insight into the research area have been of great value for our work. Secondly, I would like to thank our thesis opponent Humberto Linero for his feedback on our work and for participating in evaluating our tool. Thirdly, I extend my thanks to the other twelve participants that tested and provided valuable feedback for our tool. We have promised to leave them anonymous and thus I will not mention any names. Lastly, I would like to show my appreciation to our examiner Robert Feldt, who has provided us with tough, but fair constructive criticism throughout the term, making sure the study and writing would uphold a higher quality and standard.

Alex Tao, Gothenburg, February 2019

Contents

List of Figures

List of Tables

Glossary

Software artifact Software artifacts are tangible products of a software system. These products are objects such as requirements, class diagrams, use cases, and design documents. Here's a list of the object we refer to as software artifacts in this study:

- Requirements
- UML models (Class diagrams, deployment diagrams, sequence diagrams, etc)
- Use cases
- User stories
- Features
- Software architecture documentation
- Requirements documentation

Architectural knowledge (AK) Architectural knowledge refers to knowledge that explains what the architecture is, how it is implemented and why it was chosen to be implemented. The "why" part is a bit special because it is important but often left out in documentation. It is also often referred to as the rationale.

Ontology Ontologies are taxonomies that categorize and relate real-world objects to one another. In this paper, the ontology is often referred to a designed taxonomy representing software artifacts and architectural knowledge.

SparQL SPARQL is a querying language which can be used to query and manipulate data stored in RDF languages. To query an ontology specified in OWL, the OWL file is first converted to RDF.

System explanation composer (SEC) System explanation composer is the tool that was implemented in this project. Its main features are to present explanations to developers questions by retrieving and presenting software artifacts, AK and their relationships.

System Explanations System explanations are information presented by SEC to developers in response to their development-related questions. These explanations are presented in both graphical structures and textual formats. They present software artifacts, their descriptions, and relationships from different points of views.

1

Introduction

Large-scale software systems tend to be complex and can require teams of developers to build. This makes it unlikely that a single person knows such a system in its entirety. As time progresses, important information of the system can be difficult to access and sometimes lost. The reasons can range from information being forgotten to developers moving on to new projects. To make sure large-scale systems can be maintained and understood without being dependent on its original developers, it is of vital importance that existing documentation is correct and sufficient. However, practical issues make documentation not only difficult to write but also difficult to correctly interpret. This can make writing documentation seem like a high effort and low-value process and can deter developers and architects from writing or reading documentation altogether. The goal of this study is to find a way to address the challenges related to the interpretation of software documentation.

The following sections briefly introduce current challenges in documenting software, current CASE tools, the challenges we intend to resolve and the approach we take.

1.1 Current documentation and its issues

In order to address stakeholders information needs, techniques used for documenting different aspects of software systems have been developed over the years. Currently, technical documentation of software systems can commonly be categorized by three different document types; requirements, architecture/design and implementation. Each document type uses its own set of artifacts to document system knowledge. The requirements can be expressed through functional requirements, non-functional requirements, use cases or user stories. The architecture can be expressed through different views containing component diagrams, class diagrams, sequence diagrams, deployment diagrams, data-flow diagrams, and with each of them often an accompanying rationale. The implementation level system documentation is often expressed through comments in the code, commonly describing the inputs, functionality and/or outputs of methods or classes.

Software documentation is used by different stakeholders for different purposes. A new developer may need to learn more about a system's architecture or find dependencies between requirements and software components to figure out how to implement a feature. A project manager may need to talk about different aspects of a system to a customer and could benefit from being able to see how the system's features map to the architecture of the system. An architect may need to restructure a system and understand the drivers and reasoning behind the current system design before making any changes. The list of use cases for software documentation goes on, and it has become evident that documentation is a very important resource for most stakeholders of a software system.

However, due to the generally large and diverse audience of software documentation, they become difficult to write. To match the information needs of stakeholders, information has to be repeated, restructured and expressed differently. This tends to generate many different documents, each targeting different groups of stakeholders. When information is repeated often and spread out over different documents, it means that a significant change in the software will require updating its correspondent documentation and repeated instances of this documentation. As such software documentation become difficult to maintain.

In most cases, software documentation is stored in files, and their contents are linearly structured inside as documents. While this kind of content structure is good for many purposes, it's not quite optimal for expressing relationships between software artifacts. Software artifacts tend to form networks of dependencies that are difficult to express through linear text. The common solution to this problem has been to distribute unique IDs to different artifacts and link them together by referring these IDs to each other via tables or hyperlinks. However, as a software system increases in size, it forms complex networks between components. Information becomes spread over the different documents and collecting all information related to a subject may require extensive search and navigation[5], [8], [10]. This way of collecting information causes several risks and problems; users may miss important information and dependencies, users may ignore documentation if it's too time-consuming to find or understand, users are more prone to make errors in comprehension and interpretation due to the noise surrounding the sought information and it may take much time to find all the needed information. In other words, linearly structured documentation for software are simply inefficient in large and complex systems.

1.2 Features current commonly used CASE tools lack

This section highlights some findings and discussions from studies regarding what features current commonly used CASE tools lack. These studies seem to support a general statement; that developers need better methods to retrieve information that supports their workflow [5], [8], [10].

In a study on how developers seek, relate and use information [8], developers were tasked to fix bugs in a system they did not know very well. Even though they had access to any resource they needed (documentation, tools, source code, etc.) to find information, it was shown that developers spent a lot of time looking in irrelevant parts of the code. It was found that when searching for information, 88% of the searches they conducted yielded irrelevant results and the average time developers spent looking at irrelevant parts of the code base was measured to be around 36% of the total time. Based on their findings, the authors discuss that there's a need for tools that can (1) reduce the navigational overhead when searching for information that satisfies developers' needs, (2) provide more cues of the context before developers have to read information in full, (3) help developers relate information and (4) assist developers in collecting information. While these suggestions were mainly addressing source code, these points also apply for software documentation.

There is a lack of tools that answer high-level questions. In many cases, it requires developers to break down their original questions to low-level questions, use multiple different tools to find low-level answers and then build up a final answer to their original question [10]. However, during this process, the information found is scattered over different documents and tools. To make sense of the information developers need ways to collect information, either relying on the developers own memory or external tools. Memorizing content is often unreliable and difficult and tools often do not efficiently support important features developers need (such as tracing information back to its source or tracking information to other closely related information). As a result, building up answers for high-level questions is cumbersome and error-prone.

The current specification and requirement documentations are big, complex and of varied structure, making them difficult to maintain [5]. The results from Lethbridge et al [5] suggest that software engineers tend to use simple and yet powerful documentation and tools while ignoring the complex and time-consuming ones. They discuss that greater documentation relevance could be found by recording and presenting documentation differently and making them easier to update.

To summarize this section, we can categorize the current lack of tool support in five areas: (1) recording and presenting documentation, (2) navigating documentation, (3) searching for information, (4) relating information and (5) memorizing/collecting information.

1.3 Interrelating software artifacts and AK

The challenges regarding documentation and the lack of tool support regarding information retrieval for developers has given rise to research regarding how to store software documentation to make retrieval of related software artifacts more efficient[11], [13], [15], [17]. Artifacts such as requirements, design decisions, software components and diagram elements are represented as entities, linked to one another and forming a logical linked knowledge structure. By interlinking structures that can represent relations between different types of software artifacts and AK, the idea is that such interrelations can represent system knowledge. This knowledge can then be retrieved to support developers' information needs during development and system maintenance. The research in this area is still in its infancy and its full potential is unexplored. The largest obstacle in applying this concept to the industry is concerns about its cost to implement.

1.4 Statement of the problem

From the discussions of the previous sections, it should now be evident that the current challenges with documentation of software systems are plentiful. Table 1.1 summarizes the mentioned challenges.

ID	Challenge	Description
C1	**Size and complexity**	Analyzing a system becomes more difficult when the system increases in size and complexity and reading software documentation may lead to different interpretations between individuals.
C2	**Scattered information**	Documented knowledge tend to be spread out over different chapters and documents, which may cause different individuals to have different versions of documents and locating some knowledge can become cumbersome.
C3	**Limited approaches for information retrieval**	Current information retrieval techniques used in documentation are usually limited to common text/keyword search. In large documents, this may cause finding relevant information difficult.
C4	**Multiple audiences**	Often times, writers must specifically tailor their text to match the interests of each audience. This makes writing documentation difficult and redundant.
C5	**Excessive or redundant information**	Repeating information in documentation is a source for inconsistencies. Modifying information in one place means all repetitions has to be modified as well. As such documentation is often difficult to maintain.

Table 1.1: Summary of current challenges with software documentation

The main problems this study addresses are the following four challenges; (C1) Size and complexity, (C2) Scattered information, (C3) Limited approaches for information retrieval and (C4) Multiple audiences. However, the solutions to these challenges may as a side-effect also partially resolve the last challenge; (C5) Excessive or redundant information.

As a basis for this study, two research questions that aim to find partial answers for the five mentioned challenges were set up:

RQ1: How can existing software artifacts be presented and explained in a way that increases learnability of the structure and behavior of a software system?

RQ2: Do such explanations help developers find and interpret knowledge of the system?

1.5 Purpose of the study

This study will demonstrate some of the benefits of using an interrelated documentation structure. By demonstrating these benefits, the goal is to put the implementation costs of interrelating software artifacts and AK into perspective (Graaf, Liang, Tang Vliet discuss these costs in their article [15]). The benefits will be demonstrated by performing a design science research study, where a proof of concept will be built and evaluated. This proof of concept will be referred to as Software Explanation Composer (SEC). SEC will provide navigation and tracing between software artifacts and AK of a case system and its documentation.

We will address the challenges listed in table 1.1 using two steps. First, by designing a proof of concept system that retrieves existing software knowledge and generates explanations to answer developer questions this is SEC. Second, by investigating how developers can benefit from using this system in solving development-related tasks. Following the two steps above, a case system will be selected and SEC will be designed, built and evaluated.

1.6 Scope

To be able to fit the study within the given time frame of 30 weeks, only a single open-source software and its documentation will be used as a case study. The chosen case system is an e-commerce template, Snowflake [14]. With this case as a base, a new system, SEC, that generates software explanations will be built. The primary concern of this system is not its generalizability but instead demonstrating the possible ways of relating software artifacts and using these relations to generate software explanations for the case.

It is taken into account that not all design choices, decisions, requirements, etc. of the chosen case system, Snowflake, are presented in the given case documentation. This means it is also expected that relationships between some components won't be expressed. The generated explanations will be based on information that is available in the documentation.

Another important issue that should be raised is whether software artifacts and relations from the case system documentation will be automatically identified and populated. While this functionality is desired, current technology does not have the capability to do so reliably. Since the main purpose of this study is not to create or investigate techniques to automatically identify software artifacts and their relations, this process will be performed manually.

Lastly, to save time and effort from building the new system from scratch, multiple out of the box open-source libraries will be used as a base.

1.7 A brief summary of the approach

This section gives a brief overview of SEC in order to early on give the reader of this paper a better grasp of what SEC is and to set the following sections and chapters into perspective.

There are five commonly recurring concepts mentioned throughout the paper.

- Ontology
- Case documentation and source code
- Commonly asked development questions
- Explanation

To structure and form relationships between software artifacts and architectural knowledge (AK), we used an **ontology**. Ontologies were used for mainly two reasons: Firstly, they provide a rich syntax for defining classes and relationships. This is used to build a meta-model that expresses relationships between software artifacts and AK. Secondly, they provide the ability to store individuals of these defined classes and therefore they also function as knowledge bases.

The **documentation and source code of the chosen case**, Snowflake, was used to populate the ontology. This would serve as a demonstration of how explanations from retrieved knowledge can be generated. It is important to note that the process of populating knowledge into the ontology was entirely manual.

A group of **commonly asked development questions** was selected from existing studies. We refer to these as developer questions. These questions were used as a basis for SEC, as each explanation is tied to a different developers question. SEC generates explanations by querying knowledge from the ontology using a query language called SparQL.

Once knowledge has been queried from the knowledge base, SEC **generates explanations**. The explanations are composed of two items: One, an interactive visualization of individuals (software artifacts and AK) and their relationships. Two, a textual description for each visualized individual.

To learn more about the methodology used to design the system, see Methods chapter (chapter 3). To see a demonstration of the tool and learn about the actual design of and reasoning behind the ontology and system, see the chapter dedicated to design and implementation (chapter 4).

1.8 Where our study fits in the literature

There is existing research in improving tool support for fulfilling information needs of developers that use a similar approach as this study, but they are not rigorously generalized (To see a similarities/differences comparison between SEC and these tools, see section 6.5). Current tools cover the following areas:

1. The design of ontologies for software artifacts and AK [13], [17]
2. Tools for retrieving information using interrelated software artifacts and AK for one or two cases [11], [15]
3. Tools that support updating/adding new information and forming interrelations between them [11]

Our study stands on the shoulders of these works and aims to further the research area by studying yet another case and exploring the explanatory power of using a more specific ontology design, how interrelated information can be used to reduce navigational overhead, how relevant knowledge can be collected and how different aspects of a system can be explained.

2

Background

This chapter provides a background to our study and the literature most relevant to it. The first part of this chapter, section 2.1, briefly describes common techniques used for documenting object-oriented software. The second part, section 2.2, discusses the existing state of the art solutions to some challenges of software documentation, current similar studies and explains the gap in the literature this study aims to fill.

2.1 Current software documentation

This section provides a brief introduction to some common techniques used for documenting software and how the artifacts resulting from these processes relate.

2.1.1 Document types for object-oriented software

For object-oriented software, there are four common types of documentation: requirements specification, software architecture documentation, technical documentation, and test documentation. Each of these documentation types concerns a different process of software engineering. In this study, we set our focus on requirements specification and software architecture documentation. This section describes the two documentation types in a very simplified manner.

According to IEEE 29148-2011 [12], the purpose of a requirements specification is to address the following topics: (1) Software functionality, (2) stakeholders needs and context of use (3) how the software interacts with people, hardware and other software, (4) performance of the software, (5) attributes of the software and (6) design constraints. Software functionality is commonly expressed using functional requirements. They specify what features and functionality the software has to implement. Stakeholders needs and context of use are recorded by documenting the business purpose, goals, business model and operational scenarios. How software interacts with people, hardware and other software are commonly expressed using use cases, user stories, scenarios, etc. These describe the high-level behavior of the system, and how it interfaces with external objects. The performance, attributes, and constraints are specified using non-functional requirements. Non-functional requirements specify the quality of the software from many different perspectives, such as performance, maintainability, security, testability, usability and many more.

Software architecture documentation is used to document the high-level design of the system. They address the technologies used, how different subsystems and software components communicate and interrelate and provide means to reason about high-level design. According to Dewayne E. Perry and Alexander L. Wolf [3], software architecture can be summarized as the following: "...architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design". What is often left out in software architecture is the reasoning behind the architecture design, which is referred to as the rationale. This is also an important piece of the architecture.

2.1.2 4+1 architectural view model

The 4+1 architectural view model can be traced back to 1995 where it was first introduced by Phillip Kruchten. The general idea of this model is that software systems tend to become difficult to grasp and analyze all at once. By viewing a system from different perspectives, one can analyze the system based on its different properties, making the process more manageable and structured. The main four architectural views are; (1) the logical view, (2) the development view, (3) the process view, and (5) the physical view. The extra view, which is referred to as the "+1" view, is called the scenarios and use cases.

The logical view shows information from a system functionality perspective as presented to end-users. The main stakeholders of this view are, obviously the end-users, but developers, functional testers and architects may find this view equally useful.

The development view shows how the system implementation is designed and structured and is concerned with different software modules, packages, classes, etc. The main stakeholders for this view are developers and architects.

The process view shows the system in terms of processes and threads. It is used to analyze concurrency. The main stakeholders of this view are developers and testers.

The physical view shows how the system is decomposed physically. It shows the physical devices that the software runs on, in other words; how the system is deployed. This view is targeted mainly for system engineers and developers.

The last view, the scenarios, shows how the system behaves. Scenarios explain different ways the system can be used and how the system behaves in these scenarios. The main stakeholders of this view are both end-users and developers. However, testers, as well as architects, may also find this view useful.

2.1.3 Software structure and behavior

Software can commonly be explained by its structure and behavior, which are often expressed using diagrams. Structural diagrams describe the skeleton of a system. They express the composition and structure of components/classes/packages/etc. Such diagrams give developers information about the properties of components and how they inherit from, relate to and/or depend on one another. Some examples of structural UML diagrams are:

1. Class diagrams
2. Component diagrams
3. Package diagrams
4. Deployment diagrams

Behavioral diagrams describe, as the name suggests, the behavior of the system. Often, these diagrams refer to the same components that are found in the structure but explain the system from a behavioral standpoint. Such diagrams describe how the components interact and communicate to complete specific tasks. Examples for behavioral UML diagrams are:

1. Activity diagrams
2. Sequence diagrams
3. State machines

2.1.4 Design rationale

Designing good software, whether it be on the software level or on the architectural level, requires careful decisions based on reasoning and thought. For example, one way to represent the design rationale was proposed by A.Tang et al **Tang2006** who identified a comprehensive list of generic design rationales. This list captures most of the reasoning that decisions are based on during design. The list is the following: (1) Design constraints, (2) design assumptions, (3) weakness, (4) benefit, (5) cost, (6) complexity, (7) certainty of design, (8) certainty of implementation and (9) trade-offs.

In this study, we will use a simplified representation of the design rationale. It will focus on three categories: (1) Assumptions, describing assessments that are based on beliefs due to factors being unknown. (2) Arguments, providing motivation and discussion of a design. (3) Constraints, describing constraints in the system that affect the design and the constraints that result from the design.

2.2 Related work

This section aims to give the reader a view of prior work in the area and to demonstrate the void this study fills in the current literature. It describes some similar studies, previous use of ontologies for software documentation and research that serve as a basis for motivating this study.

2.2.1 Similar studies

This section describes some studies that are similar to SEC and some key aspects SEC differs from them. At the end of this section, a summary of the key contributions of SEC will be highlighted.

2.2.1.1 Whyline

This system was designed and implemented by Andrew J. Ko and Brad A Myers [6]. Its purpose was to assist developers in debugging their code, doing so by answering a set of "why" questions about the code. Unlike the tool of this study (SEC), it answers questions at a program level rather than at the documentation level. The questions Whyline answers are in the lines of "Why did pac resize to 0.5" or "Why didn't pac resize to 0.5" and shows why (or why not) the resize statement had been called. The tool allows the user to play the program execution both forward and backward, analyze its callgraph and view variable values.

2.2.1.2 A knowledge-based software information system, LaSSiE

Lassie is a tool designed by P. Devanbu et al back in 1989 [2]. It is a tool that allows users to express detailed relations between software artifacts and source code. The user interfaces with the tool by natural language query input, and the tool outputs search results to the user. One weakness with this tool was that the relations between artifacts were very detailed, and constructing the knowledge-base was very labor expensive.

2.2.1.3 The Knowledge Architect

Jansen et al [11] created a tool called The Knowledge Architect whose goal was to solve six different challenges with software architecture documentation (See table 2.1). Their tool is a complete package, from capturing AK from existing documentation to visualizing AK and its relations. When evaluating the Knowledge architect with two industrial cases, they found significant evidence that using their tool instead of documentation would increase the quality of architecture reviews. However, it failed to prove that using the tool would make reviewing architecture more efficient.

2.2.1.4 PAKME

The purpose of PAKME [7] is to help architects, developers and software maintainers to analyze software systems. Its purpose and features are very similar to the Knowledge Architect. PAKME has four main features:

1. Knowledge acquisition. Provides tools to edit or add new knowledge.
2. Knowledge maintenance. Provides features to modify, delete and instantiate artifacts.
3. Knowledge retrieval. Provides basic and advances search functionality to find desired AK.

ID	Challenge	Description
1	**Understandability**	The understandability of a document becomes lower when the system increases in size and complexity. Simply reading the SAD usually leads to different interpretations between individuals.
2	**Locating relevant AK**	Documented knowledge is often informally spread around, this causes the risk that different individuals have different versions of the knowledge. The other problem is locating AK may be difficult when they are large and spread out.
3	**Traceability**	Documents have limited ability to relate AK such as between requirements and software architecture. It is often not clear how they relate to one another.
4	**Change impact analysis**	Without reliable ways to locate and trace between Architectural knowledge (AK), it becomes difficult to analyze the impact of changes.
5	**Design maturity assessment**	Current architecture documentations have no overview of the status of the qualities of the architecture, such as conceptual integrity, correctness, completeness and buildability. This makes it difficult to assess it.
6	**Trust**	In large systems, changes occur frequently and the cost of updating the architecture can be high. Thus the documentation can quickly become outdated and its reliability becomes compromised.

Table 2.1: Summary of current problems with software documentation

4. Knowledge presentation. Provides support for generating different views of AK residing in the knowledge base.

2.2.2 Work that contributes to the general vision

This section highlights the work that serves as a groundwork to this study. They form underlying the concepts, theories, and motivation for conducting this study and building SEC.

2.2.3 The use of ontologies

The research from Klaas Andries de Graaf et al, [13] showed that using ontologies to structure, store and retrieve the relations between software architecture, such as design decisions, requirements and architectural elements, improved developers understanding of the architecture of a system. It provides means to trace between requirements, design decisions and architectural structure of the system. The architecture section of their ontology design is simplified and very similar to a component diagram. Elements of the architecture are stored as components that reside within subsystems. Subsystems and components in turn offer interfaces for communication between one another. This makes the model more general and entering data into the ontology less complicated. However, there's a trade-off with a simplified ontology design, as it affects the power of search results. We found this study to be interesting, and used their ontology as an inspiration when building ours.

In a further study, they compared how efficient it was to retrieve information using an ontology-based architecture documentation in comparison to a file-based documentation [15]. Their ontology had been improved to a small extent but was mostly the same. They performed experiments with two companies, asking developers to find dependencies between decisions, requirements, components, and other architectural artifacts. The developers used ontologies that were populated with architecture documentation of software they worked with in their company. The researchers then compared how efficient this was against retrieving the same information using the original file-based architecture documentation. The results showed that when developers used the ontology to search for answers for the given questions, the results were on average more accurate and faster to collect. The goal of this study was to measure the feasibility of using ontologies for storing architectural knowledge (AK). While this study, just like our project, investigates the effectiveness of categorizing and relating software artifacts and AK, the two studies have different approaches. The difference is that our objective is to take this one step further by investigating solutions to the same problems with a more presentation-based focus on how software systems can be explained using an underlying ontological structure.

Mohamed Soliman et al [17] identified the difficulty in finding architectural knowledge among programming knowledge in online developer communities, and developed a new search approach for browsing online developer communities (in this case

Stack Overflow[1]) for architecture related posts. Just as we do, the authors of this study acknowledge the importance of the higher level documentation of software systems. Their approach to answering the problem was to explore techniques for retrieving high-level architecture related knowledge using search in online developer communities. Using their proposed approach, they built a system on top of an ontology which models architectural knowledge. The ontology was designed based on analyzing annotated, unstructured architecture related posts of their case system in Stack Overflow to find structure and categorization of architectural elements. To measure the effectiveness of their search approach, they performed experiments with 16 practitioners, comparing their tool to traditional search methods. They found that practitioners performed significantly better using their tool.

2.2.4 Developer documentation on demand

A community of researchers advocates for a new vision for how to satisfy information needs of developers[16]. The goal is to find methods to automatically generate high-quality documentation from the given user query, minimizing the manual work of collecting information from different sources such as from experts, QA forums and documentation. They propose establishing precise links between software artifacts, constructing software that can automatically infer undocumented properties of systems, finding new ways for developers to request system information and new ways to generate, select and present software documentation based on the user request.

The vision underlying our study stems from this article. Their vision to retrieve, infer, generate and present documentation from different abstraction levels of software from the system level to the implementation, clearly requires multiple specialized studies to fulfill. Our intent is to contribute to this vision by specializing in interrelation, presentation, and explanation of software artifacts at the system level.

2.2.5 Commonly asked questions during programming tasks

One of the elements of SEC is the questions SEC generates explanations for. Most of the questions were chosen from the set of questions identified by Jonathan Sillito in his paper "Asking and Answering Questions during a Programming Change Task" [10]. The key elements from this paper that were useful for our study were the 44 identified types of questions and the discussion of current tool support gap in answering given questions.

2.2.6 Program comprehension

Back in 2005, Margaret-Anne Storey [18] performed a survey on program comprehension tools, theories, and methods. The mentioned theories and methods align quite well with the objectives of this study. There are three especially notable points

[1] Stack Overflow is a forum for developers where they can ask questions and share knowledge about different software systems

she discusses which we aim to address with our tool; the top-down and bottom-up comprehension theories, and that programming environments should provide different views of visualizing programs.

The top-down approach assumes that a programmer understands a piece of software by first trying to comprehend the software at a high abstraction level and steadily moving downwards toward implementation. The bottom-up approach assumes that the developer learns by analyzing a system by starting at a low level (such as implementation) and building up a higher level understanding by grouping or chunking knowledge.

Margaret also mentions that other works have shown that comprehension depends on the specific task, the individual and the program characteristics (such as how well it is documented and what language the program uses). Any of these two methods can be used differently, by different individuals and in different situations. The tool presented in this paper aims to support these methods by providing the ability to trace and navigate from high-level system knowledge into low-level system knowledge, and the other way around.

Margaret later writes the following "...Programming environments should provide different ways of visualizing programs. One view could show the message call graph providing insight into the programming plans, while another view could show a representation of the classes and relationships between them to show an object-centric or data-centric view of the program. These orthogonal views, if easily accessible, can facilitate comprehension, especially when combined"[18]. While the tool presented in this paper will not be generating call graphs, the point still stands: Developers comprehension of software can be improved when they can view a software system from different perspectives. To support this aspect, the tool presents system knowledge by different architectural views[4], by structural elements, by behavioral elements and across different abstraction levels.

3
Methods

This study has two main objectives. Firstly to design and implement a proof of concept to demonstrate a possible approach to automatically generating system explanations. And secondly to evaluate the usefulness of such generated explanations.

Figure 3.1 shows the methods used and the order they are performed. The first phase of the study began with a research phase where relevant work and common developer questions were collected and explanation templates were designed based on these questions. The second phase was about designing and building the ontology and populating it with software artifacts and AK. Next, we built a system that uses the ontology to present explanations to the developer questions. Lastly, we evaluated our system by gathering quantitative and qualitative data from usability tests and interviews.

The rest of this chapter will describe the methods for research, design, implementation, and evaluation of the built system, System Explanation Composer (SEC).

3.1 Research method

This study is a design science research and the methodology is different from natural science research in one key aspect. Natural science concerns observing existing phenomena, while design science concerns creating things that serve human purposes [1]. According to Ken Peffers et al [9], design science research is performed using six steps.

1. Problem identification and motivation
2. Define objectives for a solution
3. Design and development
4. Demonstration
5. Evaluation
6. Communication

Problem identification and motivation were performed by browsing existing research in the problem area and identifying problems that currently exist for documentation. This process is reflected in the introduction and related works. Based on the results of the problem identification, an objective for the research was then defined, which

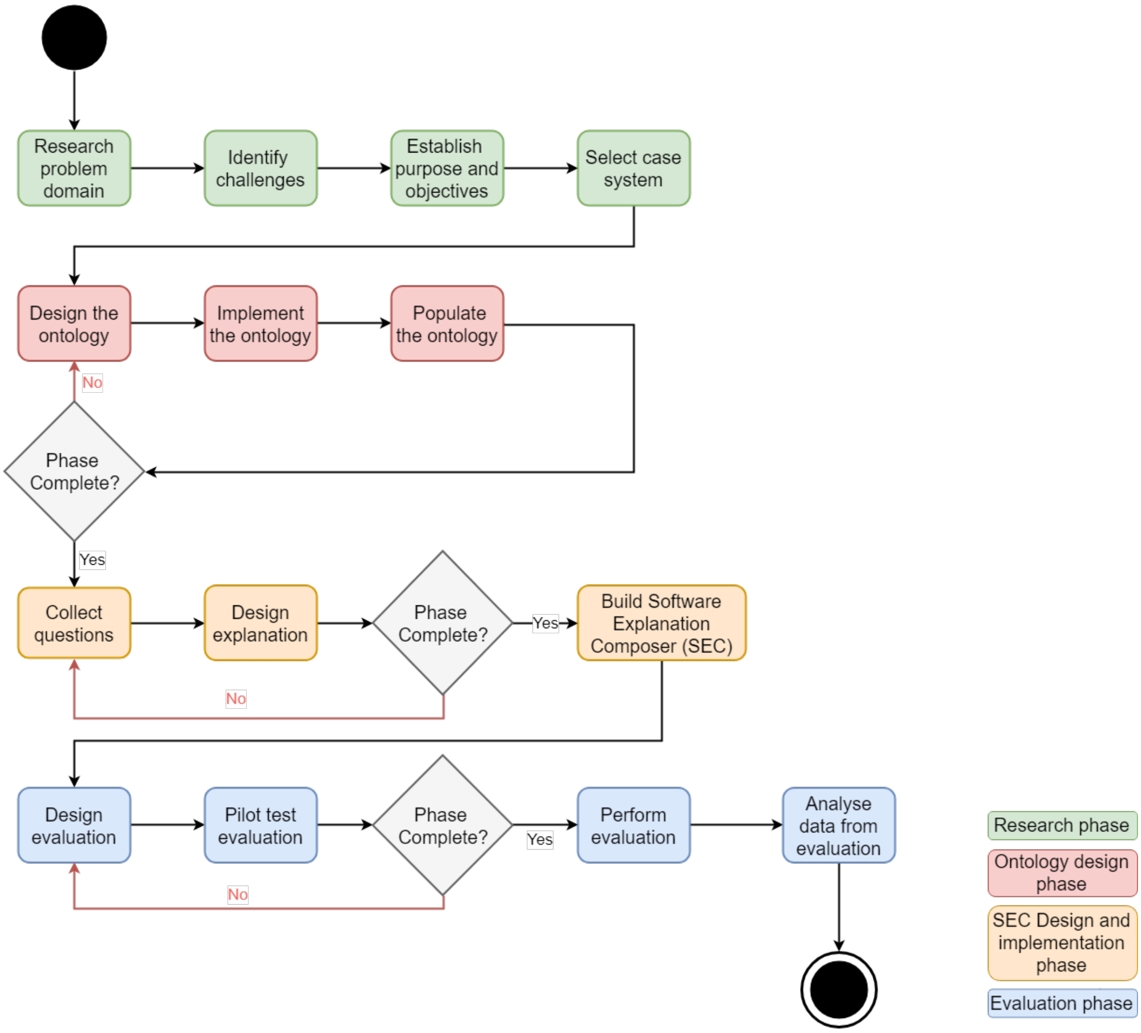


Figure 3.1: An overview of the method

is reflected by the statement of the problem. The design and development phase constituted of developing the underlying ontology and SEC, which is explained in chapter 4. The demonstration phase is tied to the case, Snowflake, which we use to populate the ontology with. An evaluation was also performed, where both quantitative metrics and qualitative feedback were collected. And lastly, communicating the details of the project resulted in this thesis report.

3.2 Selecting case system

In order to select a case system that has enough documentation, while also fitting within the boundaries of the project scope, a few simple criteria were set up:

- It should have documentation related to requirement, design, architecture, and rationale.
- The code has to be available (i.e., open source).
- The size of the system should be relatively small, so we can quickly learn it and enter all its documentation into SEC.

The selected case system is an e-commerce template called Snowflake [14]. It is a web application of 100 classes with detailed documentation for requirements, architecture, rationale and system design which matched the criteria mentioned above.

3.3 Overall system design

To generate explanations, three system components are built and used (See figure 3.2). The first component is the ontology. The ontology serves as a knowledge base to model and store architectural knowledge. The ontology is hosted in an Apache Jena Fuseki Server, allowing outside applications to fetch information from the ontology via SparQL queries. The second component is a set of common questions which developers frequently ask during development tasks. SEC is designed to answer these questions. The third and last component is the explanation generator which makes use of all the previous components. The explanation generator queries the ontology to fetch links, entities, textual descriptions, etc. and builds both textual and visual explanations for the developer questions. Each of these components is in further explained in the following sections.

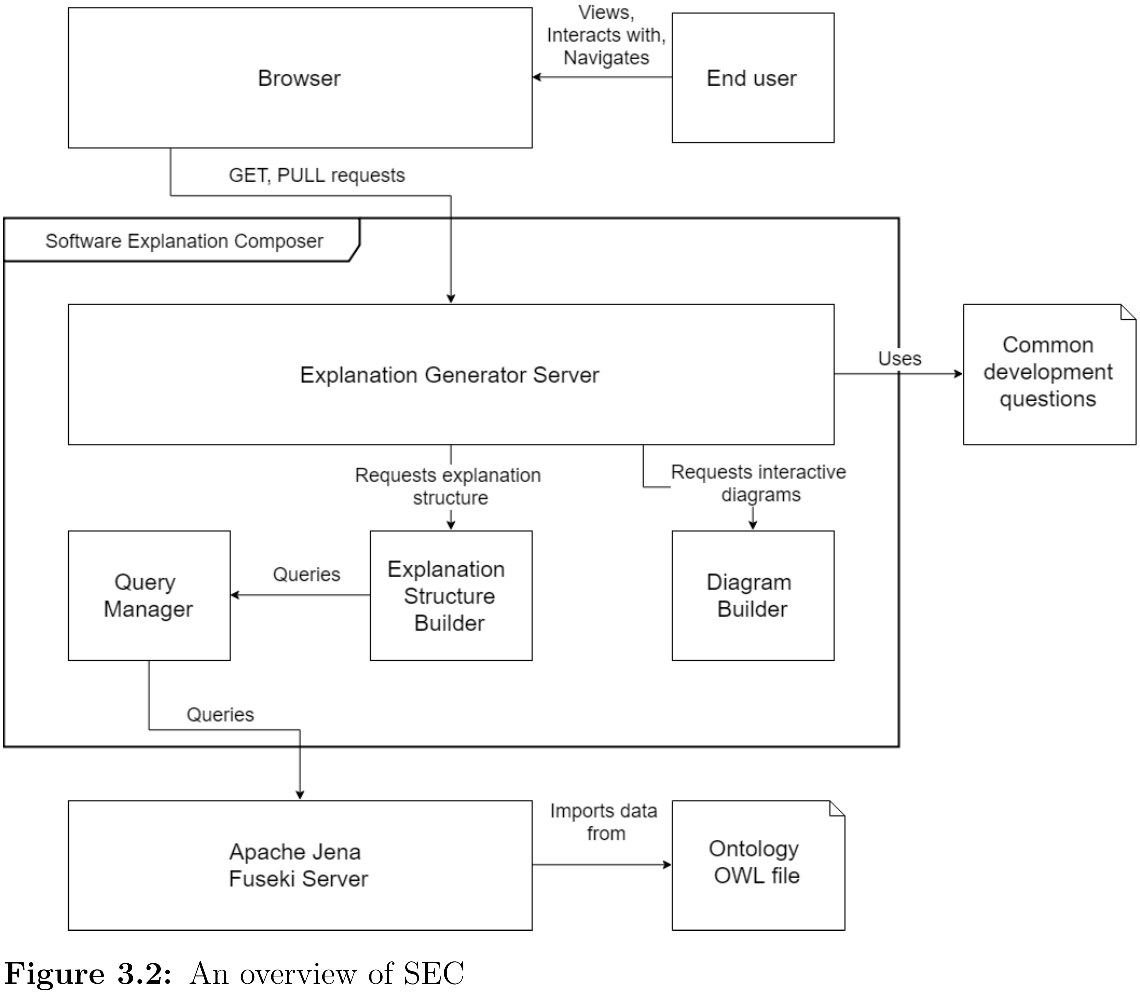

Browser
Views,
Interacts with,
Navigates
End user
GET, PULL requests
Software Explanation Composer
Explanation Generator Server
Uses
Common development questions
Requests explanation structure
Requests interactive diagrams
Query Manager
Queries
Explanation Structure Builder
Diagram Builder
Queries
Apache Jena Fuseki Server
Imports data from
Ontology OWL file

Figure 3.2: An overview of SEC

3.4 Ontology

An ontology was used as a database to store software artifacts and AK found in the documentation. Although designing software ontologies were not part of the main purpose of the system, it was required as a base to build SEC, which in turn was required to answer the research questions. As a result, an ontology was designed, implemented and populated during the thesis.

3.4.1 Designing the ontology

The ontology designed in this study was originally inspired by the ontology demonstrated by Klaas de graaf et al [13]. However, after some analysis of their ontology, it was found that it was not designed to represent two key aspects required for this project. Firstly, it has a limited ability to differentiate between different perspectives/abstraction levels of software architecture. Secondly, it lacked representation of some software artifacts which were found in the case documentation. It was concluded that ignoring these two factors could limit the expressiveness of the ontology and the explanations created from it.

In response to the above, minor modifications were made to specifically contain the architectural knowledge found in the case documentation. While analyzing the content of the documentation, it was found that the 4+1 architecture view model [4] provided a good fit to store the existing architectural knowledge. This would provide a more comprehensive model which could express different views and software artifacts. The trade-offs of using a more comprehensive ontology are that firstly more work to populate the classified software artifacts will be required, and secondly that the ontology model may not be an as good fit for different types of projects. However, since different software system designs and implementations can differ from one another by a large margin, creating a single ontology that fits every type of software system while being expressive but also simple enough, may prove to be a very difficult task. It could be argued that there is no ideal general solution to this problem, that instead different variations of ontologies should be used for different system types.

3.4.2 Implementing the ontology

The ontology was built using Protégé[1] as an ontology editor. It was used for both building the meta-model and for populating the ontology. A few simple patterns and conventions were followed when implementing the ontology which is discussed next.

[1] https://protege.stanford.edu/

3.4.3 Patterns and conventions

While designing and implementing the ontology, the following general patterns and conventions were followed to make the ontology easy to reuse, read, populate and manage.

Upper level ontology: This pattern means that general terms that are the same over different domains are modeled in a hierarchical structure. The upper level of the ontology should be general across domains while becoming increasingly more domain-specific farther down the hierarchy. This pattern provides some semantic interoperability to the ontology.

Hierarchy untangling: This pattern means that there should be no poly-hierarchies (e.g no entity with more than a single parent) in the ontology. This pattern was used to simplify the model and minimize ontology management problems.

Explicitly stating unrequired relations: Since ontologies use an open world assumption, everything that is not modeled in the ontology is assumed possible. In other words, modeling an ontology only serves to constrain the model. This means some types of relations do not need to be explicitly expressed between entities (e.g relations with cardinality 0 to many). However, as a convention to make the ontology more readable and self-documenting, such relations were expressed in the model anyway.

3.4.4 Populating the ontology

The ideal scenario for populating the ontology with data would be to find ways to automatically do so. However, there is no previous documented approach where this was done successfully with high enough precision and recall. As inventing a method (or teaching a machine) to do so would require an entire or possibly multiple studies of its own, no such system was created. Instead, software artifacts and AK from the case documentation were manually identified and inserted into the ontology.

Data insertion was performed using an iterative process. Populating the ontology started from one of the case system's features. The ontology was then traversed to fill out other entities also related to this feature until all entities, both directly and indirectly, related to the feature had been populated. We call this a "slice". As one such "slice" was completed, new knowledge and insight about the ontology were discovered. The ontology design was then updated as needed. A new iteration for inserting next "slice" began once the ontology design had been properly updated.

3.5 Explanation design phase

In the explanation design phase, a set of high-level questions which developers tend to ask during development tasks were selected from various studies [5], [8], [10].

These questions served as a basis for the explanations generated by SEC. Once questions were selected, the explanations were designed. The explanations aim to answer the chosen questions by fetching information from the ontology. The process of these two steps are in further detail explained in the following two sections.

3.5.1 Collecting questions

The goal of the system explanations was to answer a set of high-level questions based on given software artifacts. Later we could determine whether the answer to these questions was perceived as useful by developers. The process of designing the questions was performed as follows:
Firstly, a set of questions were written based on the authors' own experiences in learning or understanding software systems. This phase served mostly as a warm-up.

Secondly, relevant questions were collected from other research [5], [8], [10]. They were selected based on the following criteria:

- The questions should relate to common development tasks.
- There should be evidence that these questions are commonly asked by developers.
- The questions should touch the architecture, rationale, requirements or implementation.
- There should be questions that touch the structural aspects of the system.
- There should be questions that touch the behavioral aspects of the system.

Since the collected questions were gathered from different sources, they were also originally based within different contexts. To make sure the questions were coherent, they were modified to fit a single context.

Lastly, the questions were validated using two steps. The first step was to pinpoint what kind of software artifacts held the answer to each question. In many cases, it was found that the answers to questions could easily be found by combining different types of software artifacts. The second step was to verify this theory, which was done by answering the questions by hand using the identified software artifacts. Paragraphs in the given case documentation which were identified as holding part of the answer to the question were found and mapped to a software artifact type.

3.5.2 Design of explanations

The issue of designing the explanations was open for creativity as long as the solution addressed the previously mentioned challenges shown in table 1.1. No formal idea generation process was used to find the solutions, rather ideas were discussed between the authors and supervisors of the project. Potential ideas were expressed and presented using sketches and digital mock-ups and were changed or improved upon until they were satisfactory for further prototyping. If they were not satisfactory, they were simply discarded. The final selection of ideas had to conform to at least two of the following points:

1. Implementable as a prototype within the time-frame
2. Addresses one or more of the challenges listed in table 1.1
3. Explores new areas that have not been covered by previous research

The ideas that were satisfactory for prototyping were lastly built into SEC.

3.6 Construction of SEC

The two main purposes of SEC were to use it as a proof of concept to experiment with solutions and to evaluate whether these solutions would help developers to be more efficient in learning and understanding the chosen case system. As such, the main concerns for the implementation of SEC was to be able to quickly prototype and test solutions and to provide a somewhat intuitive user interface. Since it was important to quickly prototype and test the ideas, the speed of implementing SEC was the primary concern when choosing technologies and structuring the system. This means no formal system design was planned beforehand.

The system was implemented using three layers, the presentation layer, the application layer, and the data layer (see figure 3.3) and this section will briefly describe each layer and the technologies used.

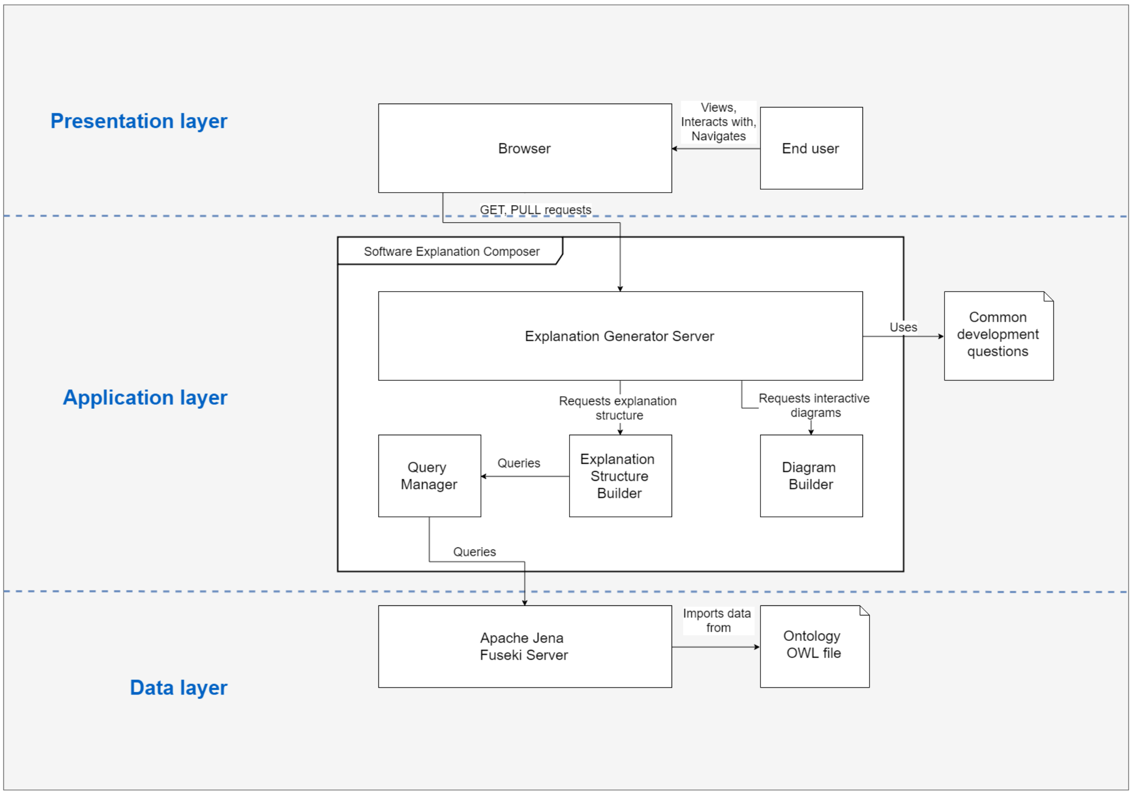

Presentation layer
Browser
Views, Interacts with, Navigates
End user
GET, PULL requests
Software Explanation Composer
Explanation Generator Server
Uses
Common development questions
Application layer
Requests explanation structure
Requests interactive diagrams
Query Manager
Queries
Explanation Structure Builder
Diagram Builder
Queries
Apache Jena Fuseki Server
Imports data from
Ontology OWL file
Data layer

Figure 3.3: The layers of SEC

The responsibility of the presentation layer is to provide a graphical user interface to the user. Its purpose is to draw the user interface and all the visualizations based on given HTML, CSS and JavaScript source code. In this case, we used the Google Chrome browser for this purpose.

The application layer which holds all the system logic was built using Python, HTML, CSS, and JavaScript. This layer was built on a Flask server which fetches data from the data layer to build explanations. Given a developer question, SEC builds the structure behind the visualizations by fetching entities and relations that make up the explanations. Then it structures and lays out descriptions which are later passed on to the presentation layer for rendering. Lastly, it is responsible for serving the layouted HTML and JavaScript code to the presentation layer for rendering. The languages used were chosen due to how easy they were to use and learn, and the sheer amount of open source content that was available for them, making UI layouting and visualization very simple. See table 3.1 for a list and summary of each library used for each programming language.

Library	**Language**	**Summary**
D3 [2]	JavaScript	Used to visualize given data structures
Dagre [3]	JavaScript	Used to translate data structures into a graph
DagreD3 [4]	Javascript	Used to draw and automatically layout dagre graphs
JQuery [5]	Javascript	Used to make http POST and GET requests to application server
TippyJS [6]	Javascript	Used to create tool tips
Flask	Python	Used as a simple framework for hosting the application server
SPARQL Wrapper	Python	Used to perform SparQL queries to the data layer
Jinja	HTML	Used to build reusable HTML templates
JSON	JSON	Used to translate python dictionary to JSON

Table 3.1: Libraries used in the application layer

The data layer was built using an Apache Jena Fuseki [7] server in which the ontology is hosted in. This server has three important features which are specifically good for this project; (1) it serves as an endpoint for querying data from the ontology, (2) it has a library of different inference engines and (3) the chosen inference engine is simple but fast, which reduces querying time significantly. In this case, we used

[7] https://jena.apache.org/documentation/fuseki2/

only some basic features of the inference system, and thus a simple version of the inference was sufficient.

3.7 Evaluation

In order to assess whether the explanations provided by SEC assists developers in solving development-related tasks, we conducted supervised usability tests and semi-structured interviews. In this section, we will discuss the design of each of these methods; the data collection and the participants of our evaluations.

3.7.1 Design of evaluation

The evaluation consisted of 3 segments, the first segment was a questionnaire regarding personal experience, the second was a usability test and the third was a semi-structured interview. We will explain the design of each of these segments in the following sections (to see the evaluation guide see appendix D).

Personal experience questionnaire:
At the beginning of the evaluation, the participants were asked to fill in a questionnaire where they indicated their level of experience in software development, in web development (due to our case system being a website), software architecture and UML diagrams.

Design of Usability tests:
Usability tests were performed in 2 main phases. Phase 1 was initiated with a short demo of SEC followed by a hands-on exploration of the tool by the participants (took around 10 minutes). Once relatively familiar with the tool, participants were then given a scenario and asked to complete 6 tasks regarding that scenario by using SEC. In phase 2, participants were given a different scenario and were asked to use the using the case system's file-based documentation to complete some 6 tasks regarding that scenario.

The scenarios in these phases were not identical but very similar in nature and level of difficulty. They were designed to be practical and to resemble a common development scenario that developers are familiar with to make it more intuitive for them to perform the tasks.

The 6 tasks in phase 1 were each mapped to the other 6 tasks in phase 2. These mapped tasks were not identical in their phrasing since the terms of certain artifacts varied between the tool and the documentation. However, they referred to the same artifacts and had the same level of difficulty. There was an exception made in the case of the task related to implementation. Since the documentation had no information regarding the implementation, this task had to be replaced with another task regarding architecture for phase 2. This was done to keep format, scoring, and timing of both phases similar.

The usability test was designed to require the participants to utilize all features of SEC. This was done to achieve an evaluation of all parts of the system at the same level. Each task was in the format of several questions, and the correct answer to those questions would result in the completion of the task. Tasks were assigned general scores based on the users' performance and a weight based on the level of difficulty of the task (to see the weights see appendix B).
The scoring of the tasks was designed as followed:

- Score 3: Correct answer without the moderator's intervention.
- Score 2: Correct answer with intervention or partly correct answer without intervention
- Score 1: Partly correct answer with intervention
- Score 0: Out of time or wrong answer

When performing one phase after another, participants could gain knowledge of the case system. This could affect how efficiently they complete the second phase. To cancel out advantages caused by learning due to the order of the phases, the order of phase 1 and 2 was swapped for every other participant.

Design of the semi-structured interview:
Open-ended semi-structured interviews were conducted to obtain participants' opinion about SEC after having used it to perform the usability tests. The questions in these interviews were open-ended because such questions usually result in broader answers and give the participants a chance to express their own opinions. This would make their answers less prone to respondent errors and less biased since they themselves chose to express a certain opinion or topic. The questions are based on the first two questions of the I Like, What if, I wish method [8]. The questions encouraged participants to provide feedback in three basic forms: First, they were asked to explain what they liked about using SEC in comparison to using the case system's documentation. Second, they were asked to imagine using SEC to assist their current work and think of changes or additions of functions that would be useful for them. Lastly, they had an opportunity to speak freely about their opinions of SEC.

The interview questions were designed this way to encourage the participants to give honest feedback. The first question simply asked what they liked when comparing using SEC against using the case documentation. This question is relatively easy to answer and gives us feedback of which parts of SEC were especially valuable to the participants. The second question has a bit more thought behind it. While planning the evaluation, we made the assumption that people, in general, tend to be conservative about expressing negative opinions and feedback, and that negative feedback generally isn't all that useful on its own anyway. As such, instead of asking what they did not like about SEC, we asked them whether SEC lacked features that they thought would be useful for them and whether there were any areas they would have liked to see improved. This question allowed them to use their imagination to suggest new ideas, while also providing us with feedback of areas of SEC that would

[8] https://ilikeiwish.org/

likely need to be improved. Lastly, we gave them free room to express whatever they wanted about the tool, in case they had other comments in mind, this gave the participants some room to express opinions not covered by the interview questions.

3.7.2 Data collection

This section describes the data that was collected during the evaluations. There are two types of data: quantitative and qualitative. Their format is explained below.

3.7.2.1 Quantitative data

Two types of quantitative data were collected during the usability tests:

Task time: The time it takes for completion of a task.

Task score: The quality of performing a task. In other words, how correctly and independently (without the help of the moderator) did the participants solve the tasks.

From the above data, we can get a sense of efficiency, which is a combined measure of how fast the participants solved the tasks and how well they scored. Analyzing these measurements to see if SEC can improve developers efficiency in performing development tasks, will partly answer our second research question. This will be discussed in depth in the results section.

3.7.2.2 Qualitative data

These were mainly the data gathered from the interviews. The qualitative data contained what participants liked about SEC, what they wish existed in the tool and other comments. The analysis of this data also partly answers our second research question from the perspective of a developer's perception. This will also be discussed further in the results section.

3.7.3 Participants

Since we wanted feedback from different points of views, we sent out participation requests to researchers and other master students from Chalmers/Gothenburg university as well as software developers currently working in the industry.

A total of 13 people volunteered to take part in the evaluations; out of which two only participated in pilot evaluations due to their prior interaction with SEC. The main purpose of the pilots was to get feedback on the evaluation design. Since the evaluation design was slightly changed after the pilot evaluations, we excluded these results from the quantitative calculations. However, the qualitative feedback from these participants was useful and was thus kept as part of the results.
Of the remaining 11 participants who performed the actual version of the evaluation, 7 had an industrial background and 4 had an academic background (for the full list

of participants' personal experience and background, see appendix C). During one of the evaluation sessions, the task times of one participant was lost. The quantitative analysis of the task times thus had a sample size of 10 while the same analysis for the scores had a sample size of 11.

3.7.4 Performing the evaluation

Each participant attended one session of evaluation which lasted about 60-90 minutes. There were two moderators present throughout the session. One of them was mainly responsible for guiding the participants' through the evaluation and the other took the main role of a scribe and kept track of time. However, both took notes of answers, personal observation and kept track of whenever participants received tips for the tasks.

Since it was the first time participants had seen the tool, the moderators would answer participants' questions as long as the answers wouldn't directly help them solve the tasks. For example; they could get help finding the correct part of the tool or get some tips about the chapters they may find things they were looking for, but the moderators would not help them analyze the content. Every task had a time limit of 5 minutes to keep the sessions within 90 minutes, but whenever there was extra time, giving participants some additional time to solve their problems was prioritized over staying within time limits. This was applied to both documentation and SEC.

4
Design and implementation

This section explains how we approached the first research question: "How can existing software artifacts be presented and explained in a way that increases learnability of the structure and behavior of a software system?". Here we explain and discuss the ontology design, the explanation design and lastly demonstrate the functionalities of SEC.

4.1 Relating software artifacts using an ontology

Software systems are commonly expressed using collections of design, specification and software artifacts which together form a network of relations and dependencies. To be able to present and explain these networks, it is first necessary to model and store knowledge of how the artifacts are categorized and interrelate. However, the act of storing and expressing knowledge has some special needs that traditional table-based and document-based databases are not very good at fulfilling:

- Expressing knowledge of a domain
- Have a rich language that expresses how elements relate to one another
- Combining different knowledge models to aid collaboration and improvements; in other words, knowledge models that are extendable, reusable and modifiable

Ontologies are specifically designed to fulfill the points above and thus they make a great match for the database needs of this project. On top of providing the needs mentioned above, when comparing ontologies to traditional file-based documents; since they differ in how information are stored, they also differ in the way information can be retrieved (See figure 4.1, 4.2 and 4.3).

When responding to queries, traditional information retrieval methods match the words and analyzes surrounding text in documents to retrieve likely related paragraphs, pages or chapters (See figure 4.3). Using ontologies to find information on the other hand, enables retrieval of relevant information across different levels of abstractions and their relationships in a meaningful context. This allows the information retriever to a certain extent understand the information to answer high-level questions (See figure 4.2).

Based on the above reasoning, we chose to use an ontology as our database. Our goal was to design an ontology meta-model that includes the important software artifacts of the case system, expresses the needs of different stakeholders and is easy enough to populate and understand.

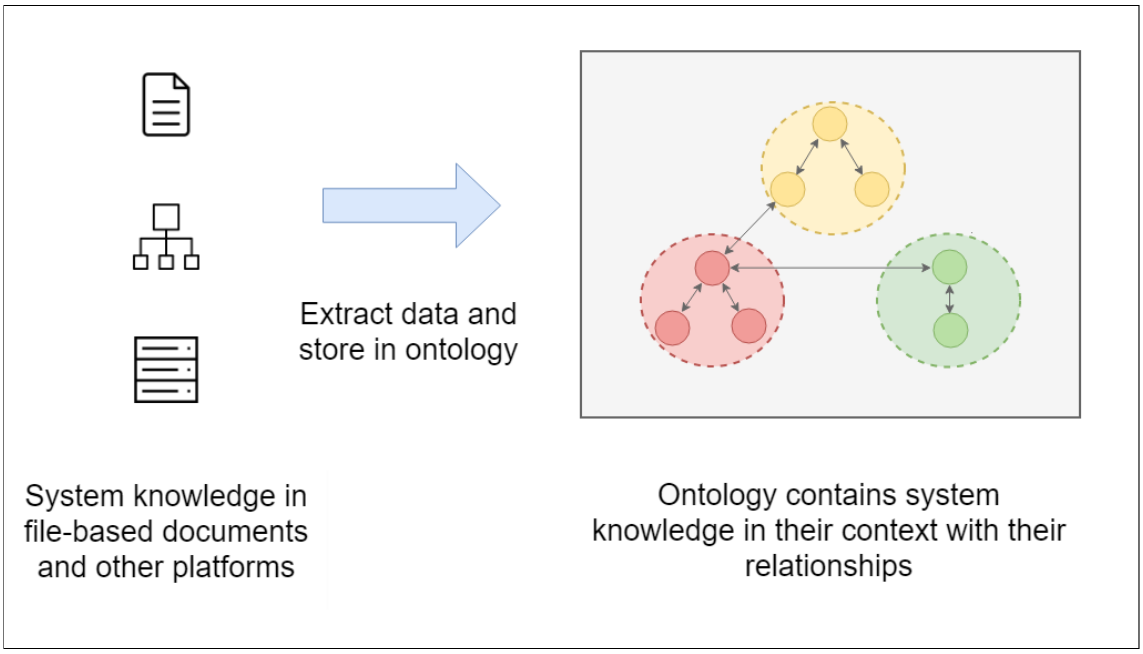

Extract data and store in ontology
System knowledge in file-based documents and other platforms
Ontology contains system knowledge in their context with their relationships

Figure 4.1: Storing identified software artefacts and AK from file-based documentation information into the ontology

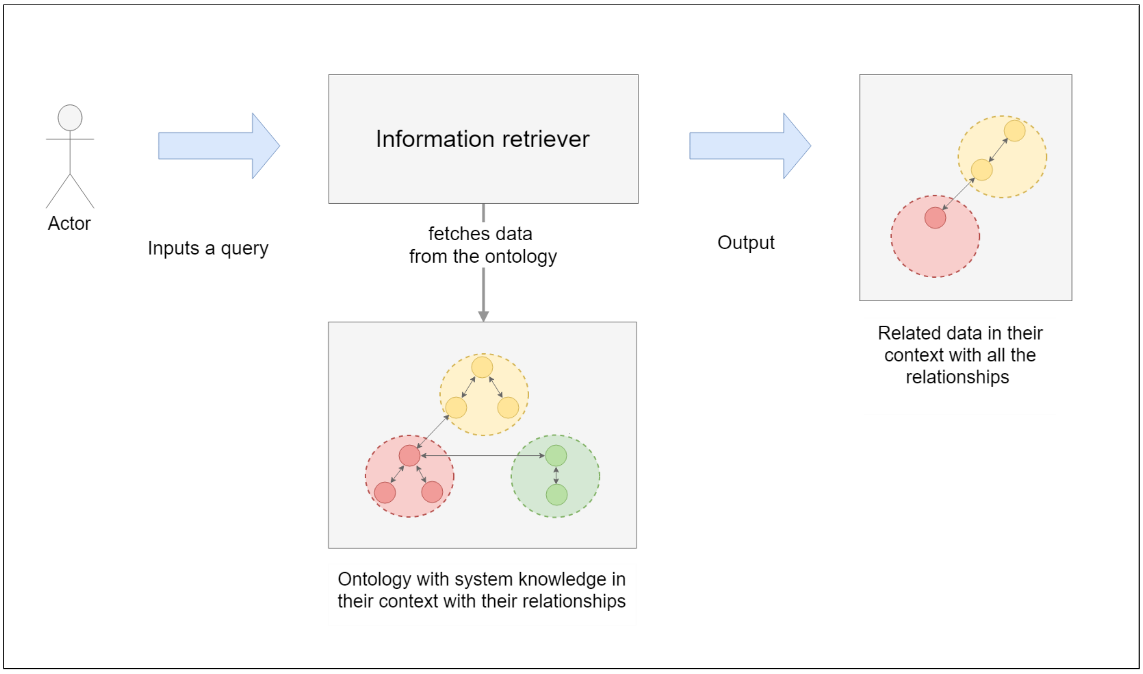

Actor
Inputs a query
Information retriever
fetches data from the ontology
Output
Related data in their context with all the relationships
Ontology with system knowledge in their context with their relationships

Figure 4.2: Information retrieval using an ontology

To make sure the ontology meta-model includes the most important software artifacts of the case system, the case system was analyzed and the specific artifacts being used were identified. For example; some projects may use class diagrams to show their system design, while others use component diagrams instead. Others may use both or something entirely different. As such, when modeling the meta-model, generic interfaces were used to represent the same type of models. Each of

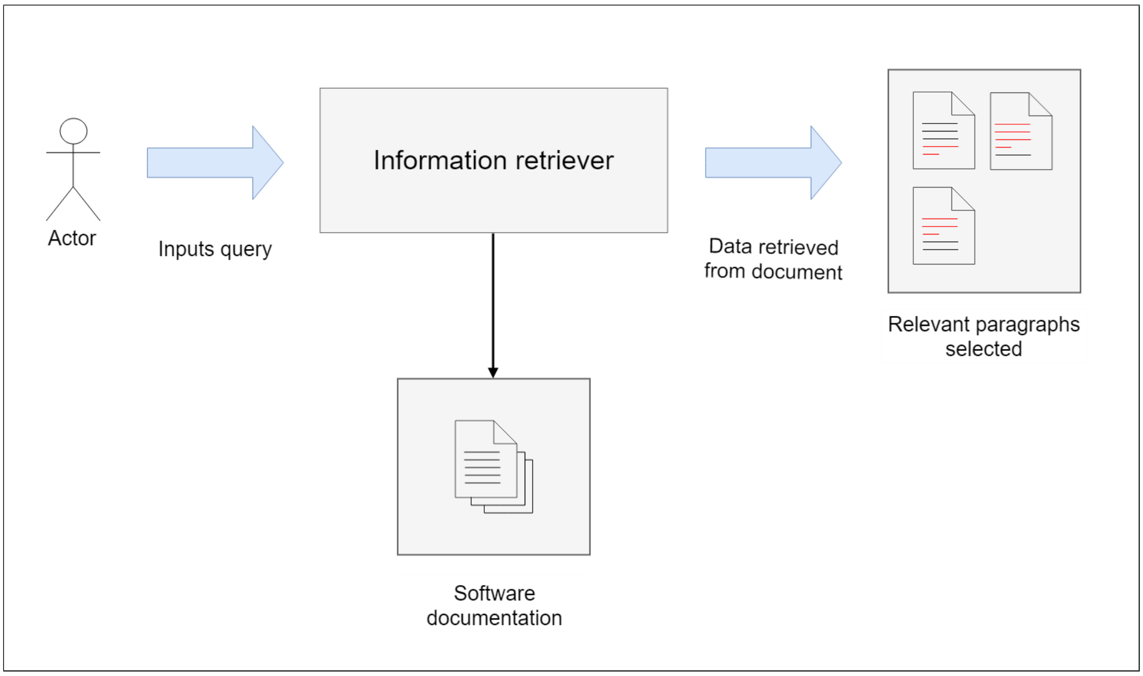


Figure 4.3: Traditional information retrieval

these generic interfaces was designed to be extendable and specialized to satisfy the needs of different software systems.

The information needs of different stakeholders differ between projects, which means there is no simple way to model ontologies that satisfy them all. One way to approach this issue is by analyzing the needs of the specific stakeholders for the case system and build the meta-model around that analysis. However, this would make the model specialized for a specific system and less reusable. It is important to note that one of the key aspects of ontology modeling is enabling the ability to extend and reuse it. As such, we made the choice to go for a direction which supports extension and modification of the model by using the following three methods; (1) by using common interfaces for ontology classes as explained in the paragraph just above, (2) dividing the ontology into different sections, which touches different types of software artefacts each and (3) by using a slightly adapted version of the 4+1 architectural view model [4].

The four main sections this ontology consists of are requirements, architecture, rationale, and implementation (See figure 4.4). There are three reasons for dividing the ontology into these four different sections. First, these four sections can represent most artifacts of common software systems. Second, it enables extension and modification within each section without affecting other sections of the ontology, which improves the extensibility and modifiability of the model. Third, it improves the understandability of the meta-model by making it easier to visually comprehend and analyze.

The architectural views of the 4+1 architectural view model are different perspec-

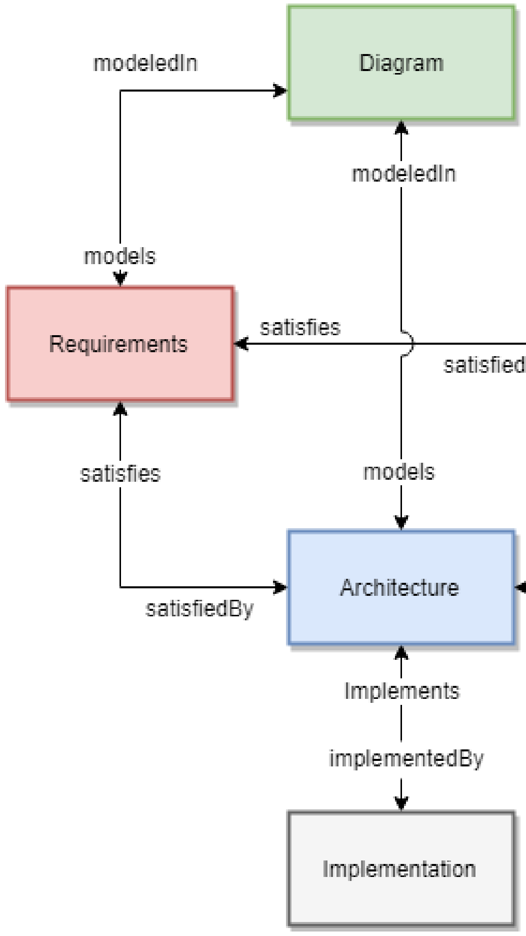


Figure 4.4: An abstraction of the ontology showing the four sections plus diagrams

tives to view the architecture of a system. Typically four different views are used to describe a software system: the physical view, the development view, the logical view, and the process view. In addition to these views, there are also scenarios and use cases which represents the "+1” part of the view model.

The original 4+1 architectural view model was not fully adhered to in this case, since this case-system is a web-based e-commerce system where the User Interface (UI) components are vital, the less prominent process view (based on the given case documentation) was replaced with a UI view (shown in figure 4.5 and 4.7). Each of these views consists of structures and some of them also contain behavior. The behavior components are deconstructed from scenarios (the +1 component of the view model). Structures express the structural artifacts of the case system from the perspective of the given view. For example; structures of the development view are structural artifacts such as classes from class diagrams (see figure 4.8). They express the structure of the system’s architecture from the development point of view, while structures of the physical view show how different components of the case system are deployed 4.7. Some architectural views also have behaviors. Behaviors of a given view describe how the structural artifact of the same view behaves. They are usually visualized using sequence diagrams, activity diagrams and/or state machines.

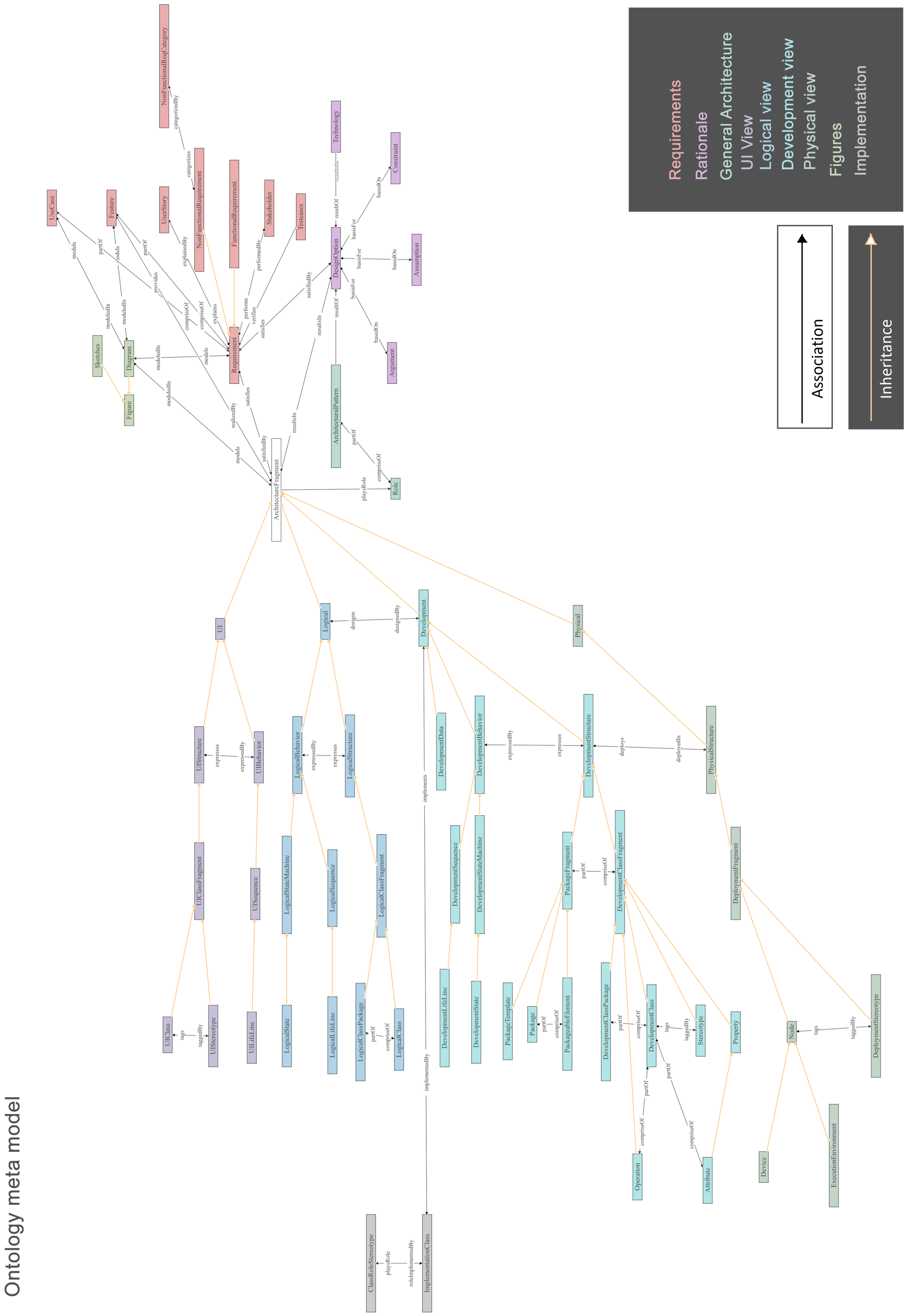


Figure 4.5: A complete view of the ontology

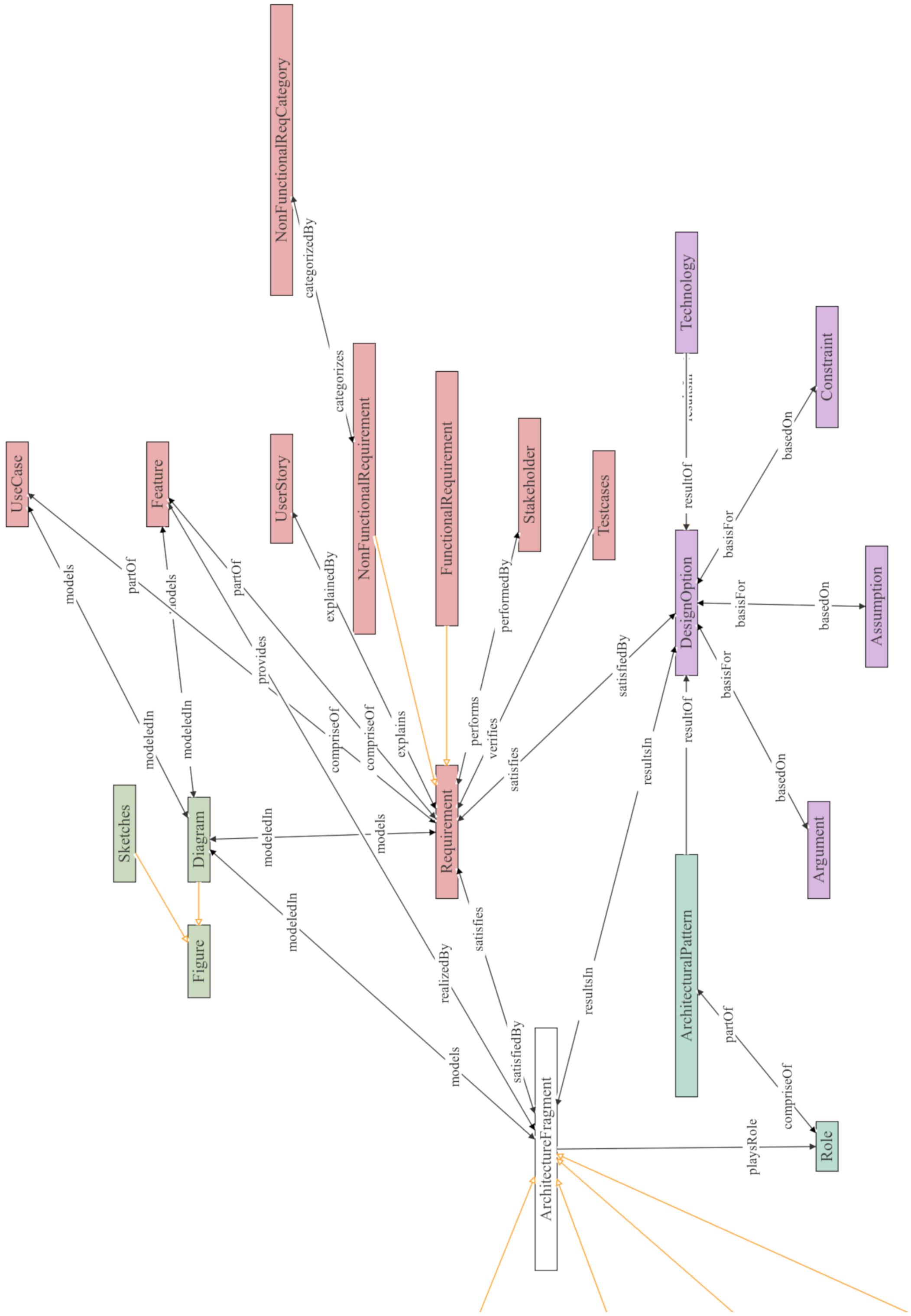


Figure 4.6: A closer look on the right part of the ontology

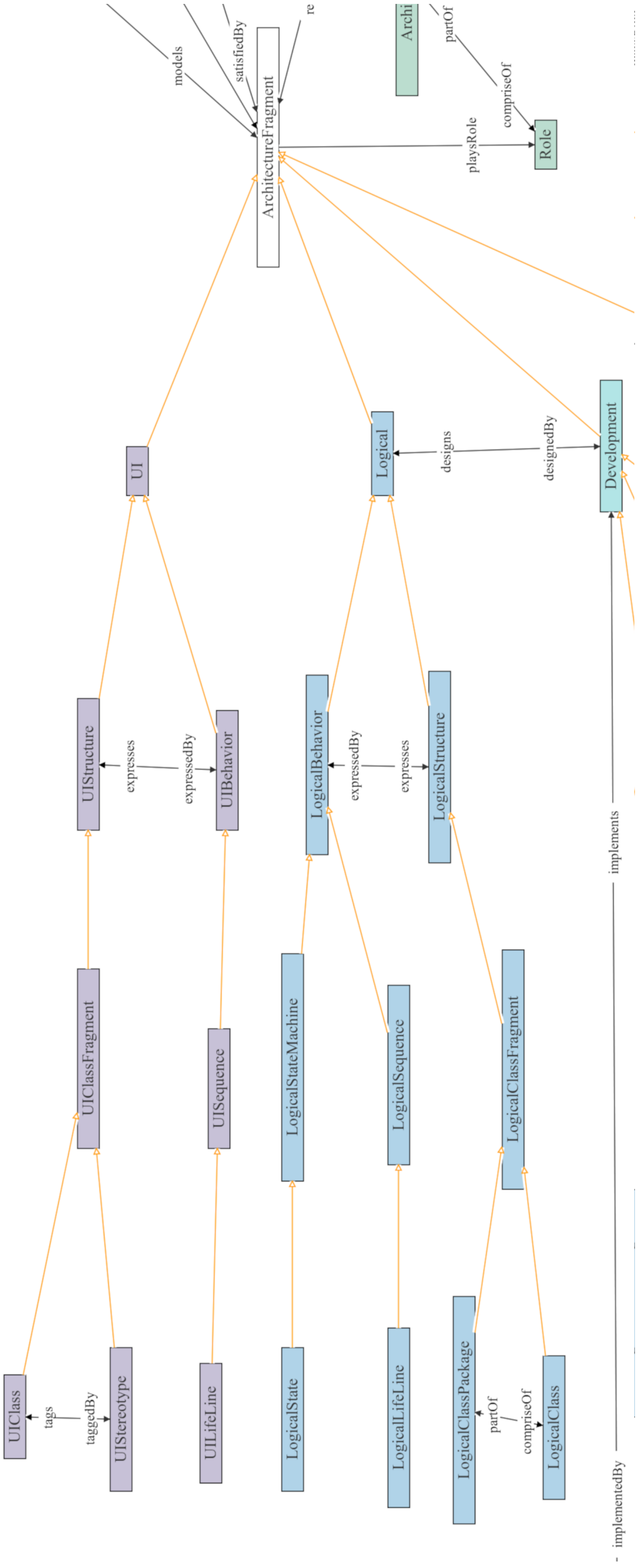


Figure 4.7: A closer look on the top-left part of the ontology

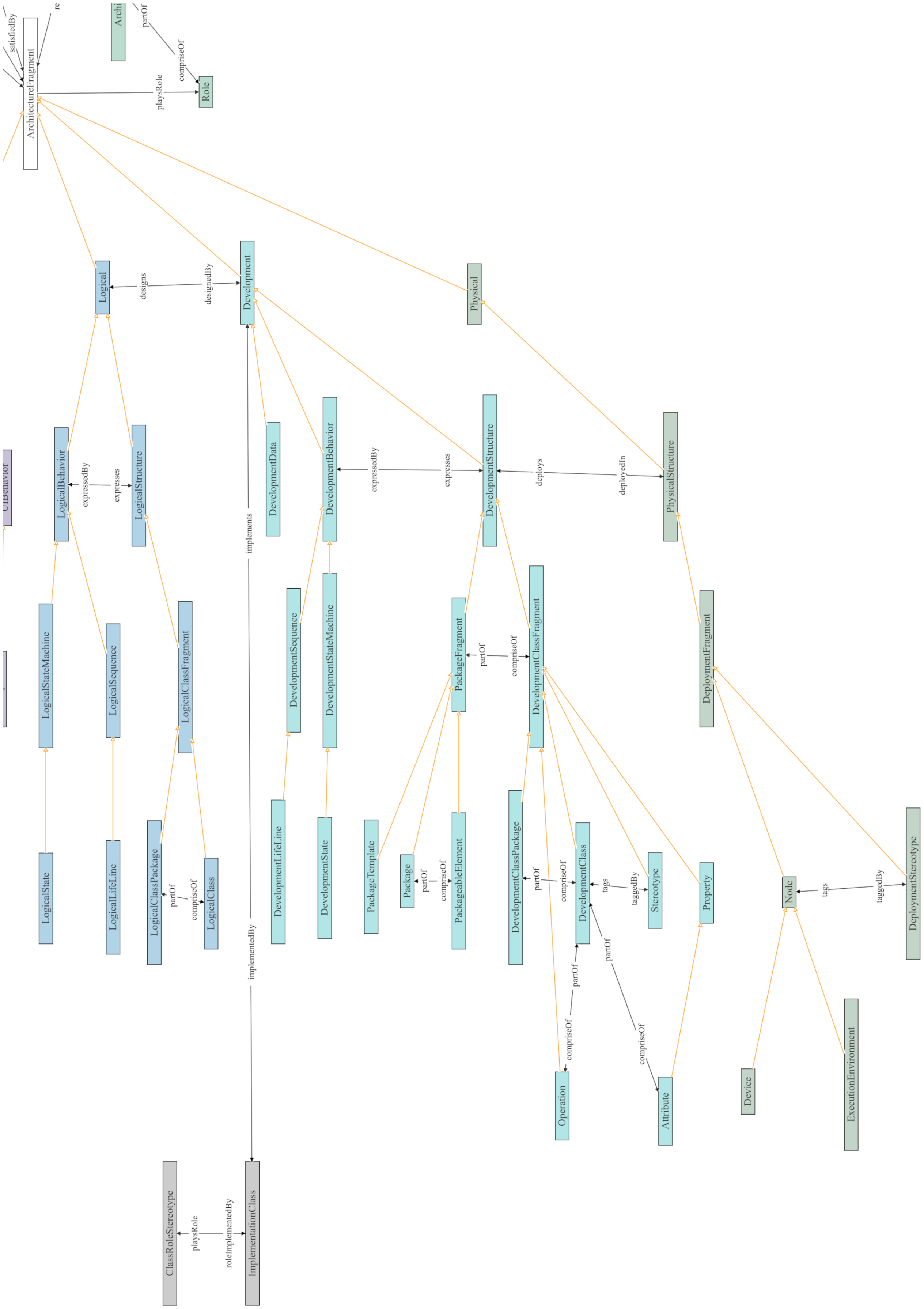


Figure 4.8: A closer look on the bottom-left part of the ontology

4.1.1 Structure and Behavior

In the ontology, each of the architectural views have structures (see figure 4.7 and 4.8). The structures for a specific view expresses how the system is structured from this viewpoint. These "structure" classes (i.e DevelopmentStructure, LogicalStructure, PhysicalStructure and UIStructure) in the ontology are not meant to be instantiated. Instead, they serve two main functions. First, they are used as interfaces that enable extensibility and reusability of the ontology. And second, they make structural and behavioral entities easier to distinguish. In general, structures are commonly expressed using structural diagrams such as:

- Class diagrams
- Component diagrams
- Package diagrams
- Deployment diagrams
- Object diagrams
- Composite structure diagrams

Recall that to simplify the ontology, only diagrams that were found in the case system of the project were modeled in the ontology. For this case, the deployment diagram was modeled in the physical view. Class diagrams were used in the logical, UI and development views. Package diagram was used in the development view.

The UI, Logical and Development views also have "behavior" classes. The behavior of a specific view expresses how the structural components of that view behave. These behaviors are commonly expressed using the following behavioral diagrams:

- Sequence diagrams
- Activity diagrams
- Use case diagrams
- State machines

In the ontology, sequence diagrams were modeled in all three of the views (Logical, Development, and UI) and state machines were modeled only in the Logical view.

4.1.2 Rationale

The rationale of the ontology consists of two main types of artifacts; design options and technology. Design options are alternatives to design considered for building the architecture of the system. To differentiate between design options that are implemented in the system from design options that are not, the implemented design options are marked as chosen (using a class variable). The reasoning behind the design options is expressed by arguments, constraints, and assumptions.

The technology class in the ontology represents libraries, languages, frameworks and other software technologies that were used to implement the case system. They are specializations of design options, meaning they inherit the properties of design options. Technologies can also be marked as chosen, and their reasoning is also expressed by arguments, constraints, and assumptions.

4.1.3 Relations between different sections of the ontology

Recall that the physical view shows the physical nodes and devices of the system. The development view and shows the actual modularization of the software. It shows how the system is decomposed to subsystems and components and how they relate to one another. The physical and development views are related since the software components of the system has to reside within some physical device. The relations between the physical and development view are modeled via the "deploys" and "deployedBy" relations between the DevelopmentStructure and the PhysicalStructure.

The logical view which focuses on the end-user and shows an abstraction of the system based on the problem domain also has a connection to the development view. The logical view functions as an intermediate that allows tracing from the structural aspects of the development view onto the features and requirements. In other words, the link between the logical view and the development view allows tracing and navigation from the software modules to the requirements and functionalities of the system. This relation is expressed using the "designs" and "designedBy" relations which reside between the "Logical" and "Development" classes of the ontology.

4.2 Design of explanations

To make sure the SEC can answer questions that developers tend to ask, a few frequently asked developer questions [10] were selected from previous studies (see section 3.5.1). All questions are listed in table 4.1. The questions that were inspired by previous studies are the questions DQ2 to DQ5. DQ1 was necessary to include since it gives an overview of the system functionality. Although this question is not specifically mentioned in previous studies, we deemed it useful for providing an overview of the system. DQ6 and DQ7 are useful for architects, but also in some cases for developers. They may not be commonly asked, but they are important questions that can be difficult to answer using common file-based documentation.

ID	**Question**
DQ1	Which functionalities exist in the system?
DQ2	Which architectural patterns exist in the system?
DQ3	What is the role of this feature?
DQ4	How is this feature mapped to its implementation?
DQ5	What is the behavior of this feature?
DQ6	What is the rationale behind the choice of this architectural pattern?
DQ7	How is this architectural pattern implemented?

Table 4.1: Selected developer questions

Based on one of the questions from the list above, explanations are composed by selecting "sub-graphs" of the ontology. For example; to explain which functionalities exist in the system, a subset of the ontology classes and relations are selected. In this case, they are features, requirements, user stories, use cases, and their interrelations (see figure 4.9). When it comes to more complicated questions, such as the question regarding the feature behavior, different answers can be composed since there are multiple different possible compositions of "sub-graphs" that answer the question. Each different composition answers the question from a different perspective or abstraction level. For this problem, we have chosen to display the compositions that we think are useful for developers (see figure 4.10).

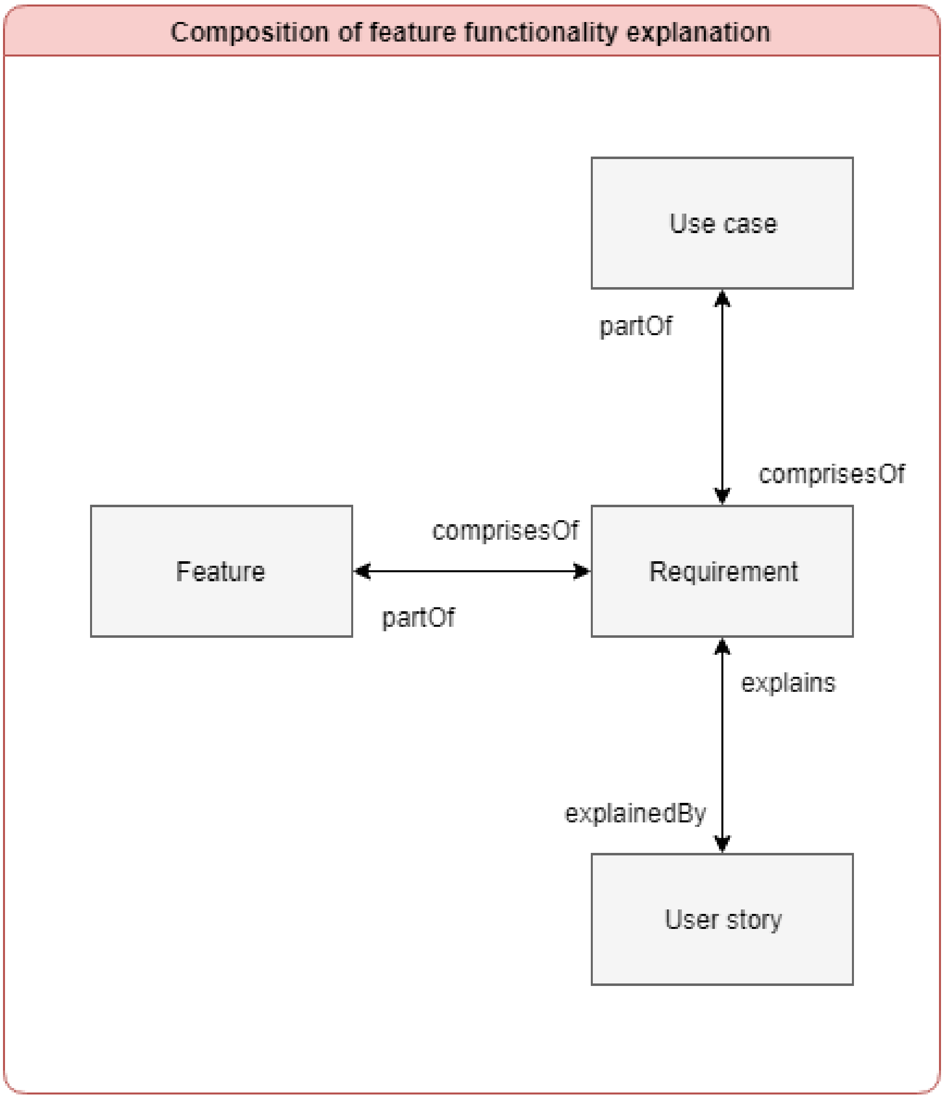


Figure 4.9: Composition of feature functionality

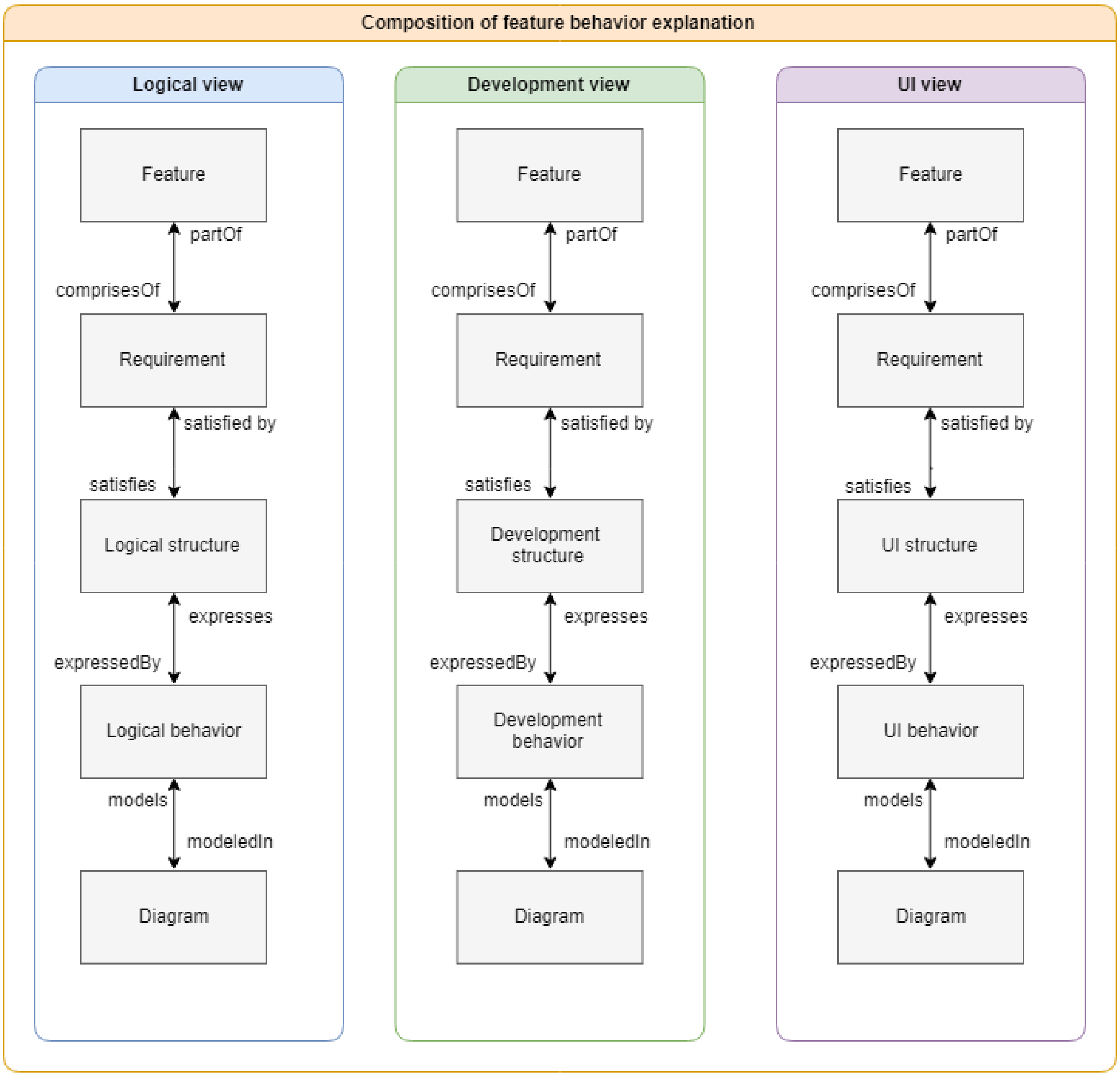
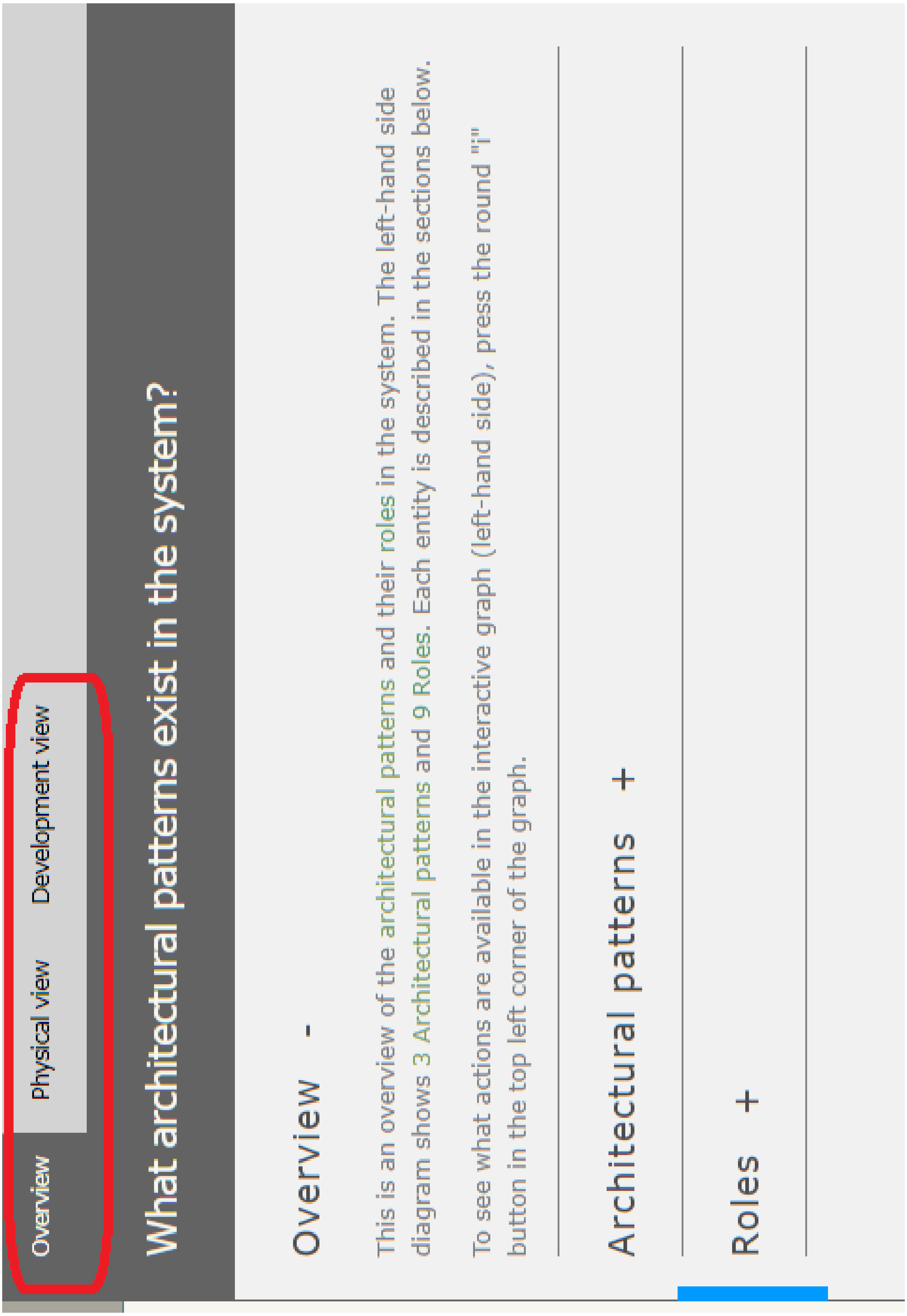


Figure 4.10: Composition of feature behavior

The resulting explanation design consists of two connected main components; a graphical component (interactive visualization of software artifact relations) and a textual component (descriptions accompanying the visualization). Some questions touch multiple different views of the architecture and to make the visualizations easier to interpret and to some extent minimize cluttering in the graph, different tabs are used to display each different view (see figure 4.11). Although when needed, information from different views can also be combined to be displayed in a single tab.



What architectural patterns exist in the system?

Overview -

This is an overview of the architectural patterns and their roles in the system. The left-hand side diagram shows 3 Architectural patterns and 9 Roles. Each entity is described in the sections below.

To see what actions are available in the interactive graph (left-hand side), press the round "i" button in the top left corner of the graph.

Architectural patterns +

Roles +

Figure 4.11: Tabs shown in red selection

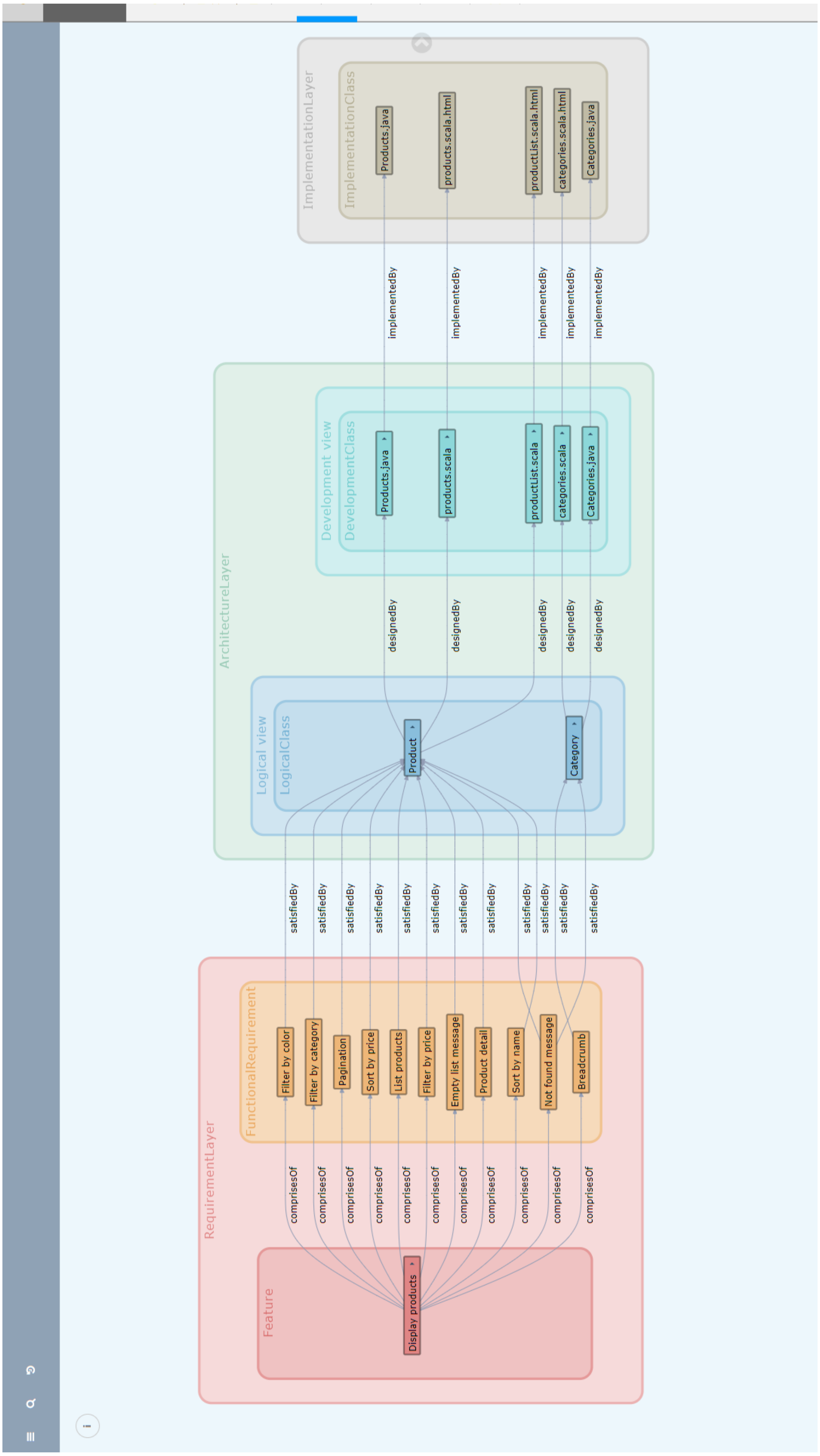
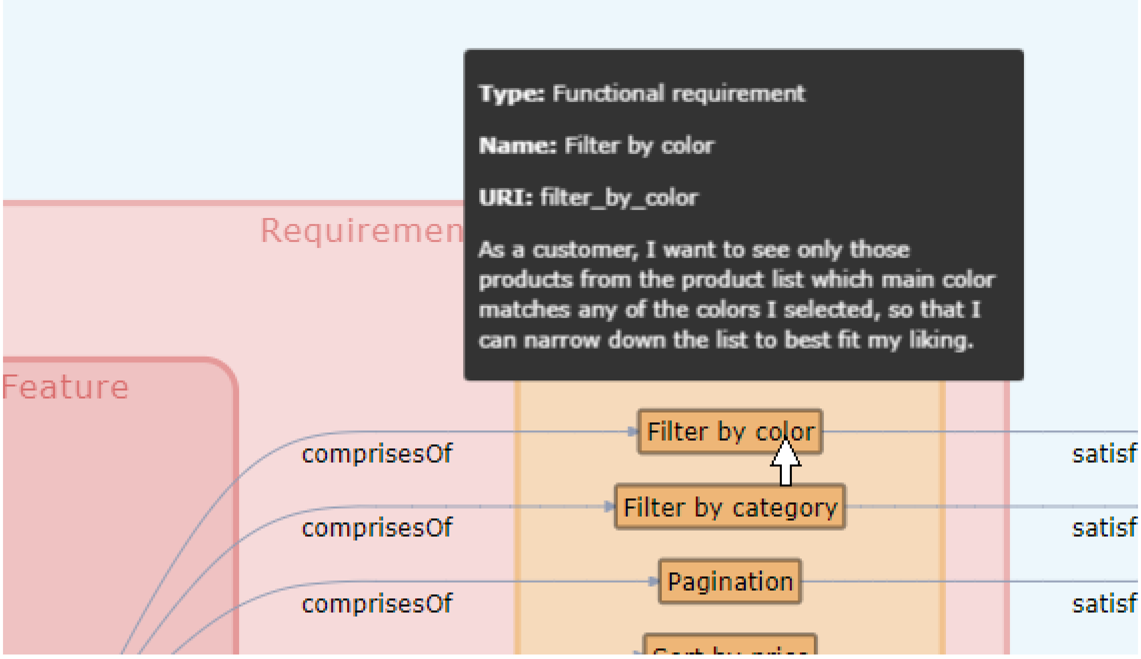


Figure 4.12: The interactive visualization



Type: Functional requirement

Name: Filter by color

URI: filter_by_color

As a customer, I want to see only those products from the product list which main color matches any of the colors I selected, so that I can narrow down the list to best fit my liking.

Figure 4.13: Hovering an entity shows a summary

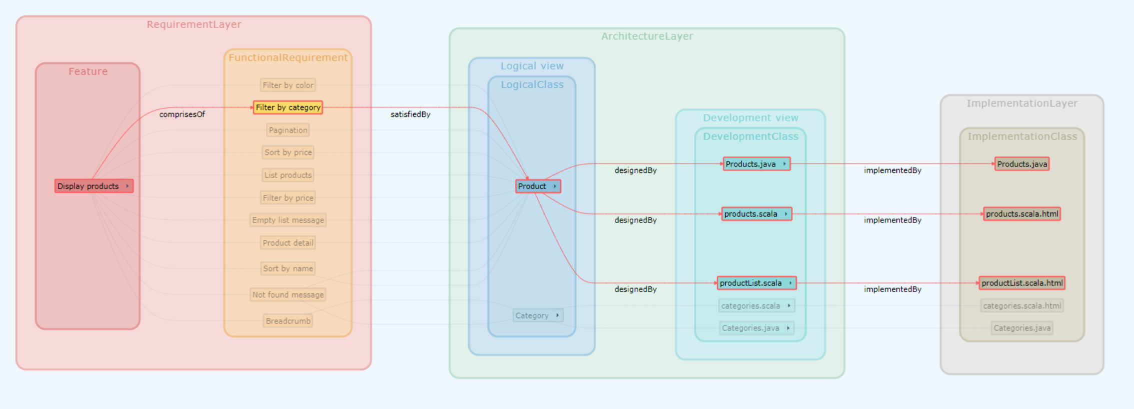


Figure 4.14: Clicking an entity highlights both directly and indirectly related entities and links

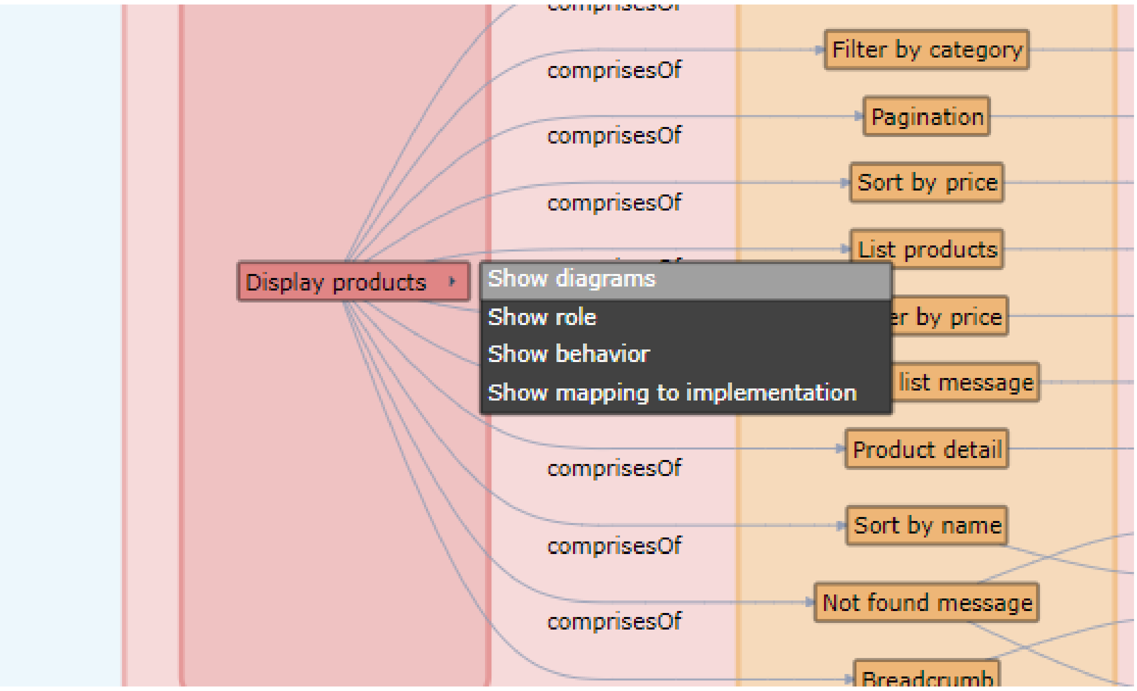


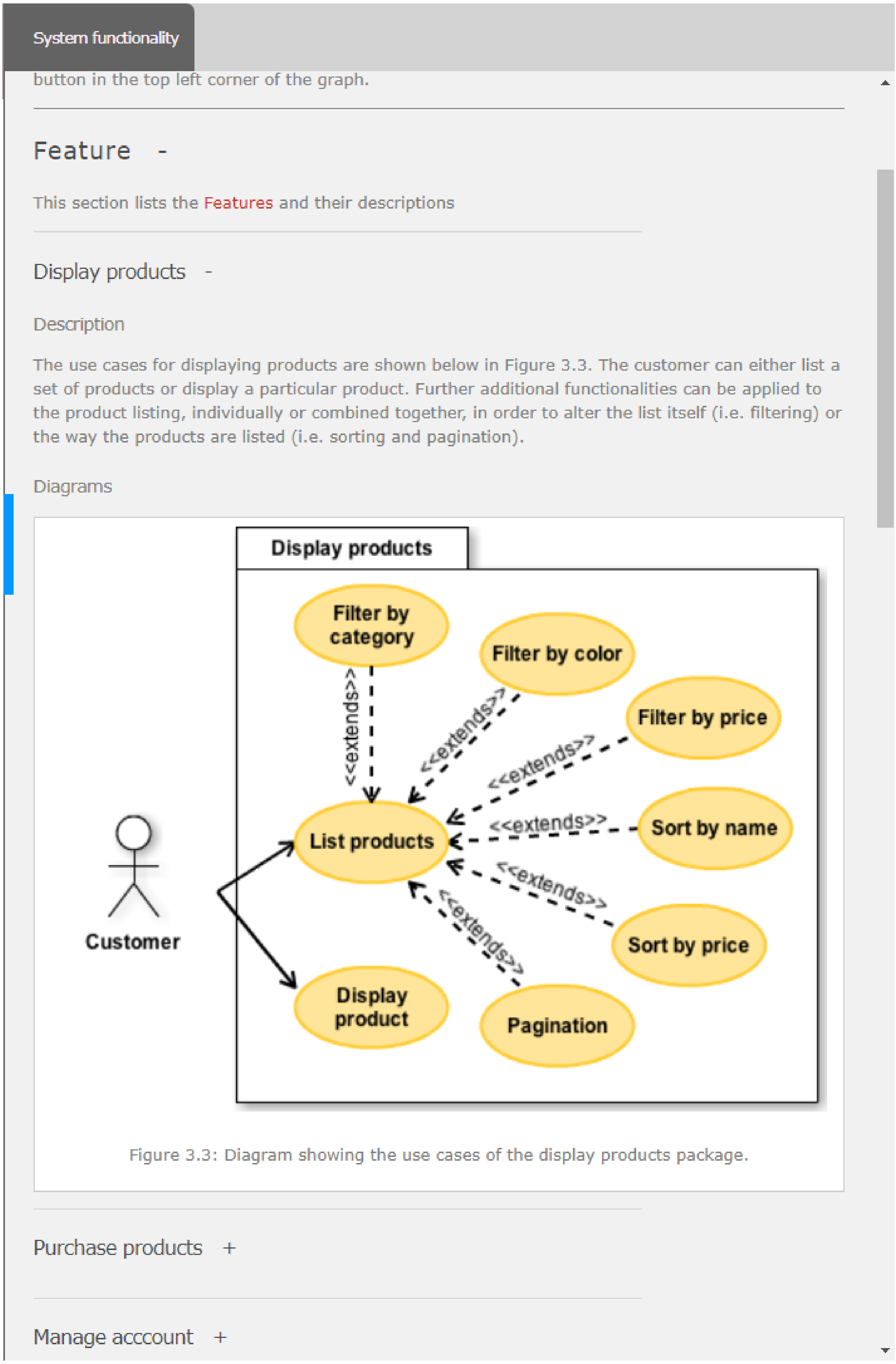
Figure 4.15: Selecting an entity allows the user to navigate to different explanations of it

In order to generate interactive visualizations, input from the user is required. The input is one of the developer questions, and if the question requires it, also one or more accompanying software artifacts. After input has been given, the system queries data from the ontology. Using this data, it constructs a visual explanation of the given question. The visual consists of the concerning software artifacts and their relations. As per the design of the ontology, there are five main types of software artifacts that can be visualized: requirements, architecture, rationale, diagrams, and implementation. The visualization serves as a means to explain overall structure and relations within and between these artifact types. To further support navigating, tracing and comprehending different sections of the system, it has three features listed in table 4.2.

The visualization is also accompanied by an extendable side-bar containing textual descriptions of the visualized entities (see figure 4.16). The first section in the side-bar briefly describes what the interactive graph is showing. The other sections show descriptions and diagrams of each visualized entity. The side-bar has several attributes which are designed to complement the visualization which is described in table 4.3.

4.3 Demonstration of SEC

This section will demonstrate the explanations to each developer question. As just described in the previous section, in order to generate explanations from SEC, a starting input has to be provided. This input is formed by providing a developer



System functionality

button in the top left corner of the graph.

Feature -

This section lists the Features and their descriptions

Display products -

Description

The use cases for displaying products are shown below in Figure 3.3. The customer can either list a set of products or display a particular product. Further additional functionalities can be applied to the product listing, individually or combined together, in order to alter the list itself (i.e. filtering) or the way the products are listed (i.e. sorting and pagination).

Diagrams

Figure 3.3: Diagram showing the use cases of the display products package.

Purchase products +

Manage acccount +

Figure 4.16: The sidebar with textual descriptions

Type	Description	Figure
Interaction	Summarized information of the entity can be revealed by hover	figure 4.13
	It has the ability to highlight paths and entities relevant to a specific entity	figure 4.14
	Navigating links is directly accessible by selecting entities of the visualization	figure 4.15
	It has basic functionalities to make the graph more readable (i.e zooming and panning).	
Consistent coloring	Each entity of the visualization is consistently differently colored by their type across explanations	figure 4.12
Automatic layouting	Entities are automatically clustered by their artefact type	figure 4.12
	Entities are automatically laid out in separate levels depending on their relations	figure 4.12

Table 4.2: Features of the interactive visualization

question (see section 4.2), and if the question requires it, also an entity (see figure 4.17). SEC uses this input to query structure and descriptions related to the question and creates explanations based on the queries.

4.3.1 DQ1: Which functionalities exist in the system?

System functionality corresponds to the features and requirements in the ontology, the classes are features, requirements, user stories, and use cases. Creating an explanation to this question means fetching all entities and the relations among them from the ontology and displaying them visually and textually (See figure 4.18). As can be seen in figure 4.19, all features are displayed on the left-hand side and each of them connects to one or more functional requirement(s). These functional requirements, in turn, connect to use cases and user stories. In this case, no non-functional requirements were related to the features and were thus not displayed. In terms of explanation, the requirements and user stories present the functionalities which exist in the system and the use cases present how these functionalities are used.

This visualization, while very high level and not very detailed, can be useful to different stakeholders such as new developers, functional testers and other non-developers (such as end-users or managers). For example; new developers can use the view to quickly identify system requirements and use cases, functional testers can use it to identify what requirements, user stories, and use cases needs to be

Name	Description	Figure
Current question	The current question in focus is located at the top of the extendable side-bar	
Automatic scrolling	Whenever an entity is selected in the interactive visualization, the right-hand side automatically scrolls down to the section with the description of that entity.	
Retractable side-bar	It is extendable and retractable, allowing the user to make more room for the visualization if needed.	
Grouped by direct class	Entities are grouped by their direct class types (i.e all functional requirements are grouped together).	
Expandable/collapsible sections	Each text section is extendable and collapsible, making navigation of large amounts of text a bit more manageable.	

Table 4.3: Attributes of the textual descriptions side-bar

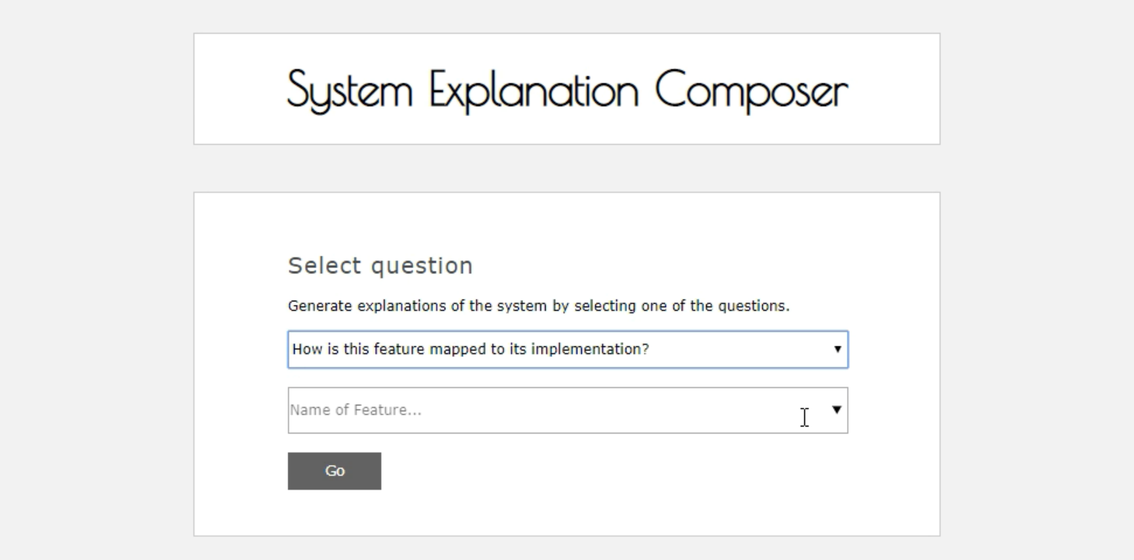
validated and verified, and non-technical stakeholders can use it to learn more about the system without diving into technical details. Lastly, it can also be used as a tool that assists developers or architects in maintaining the documentation. From the visualization, it is easy to see whether requirements or use cases are missing for a certain feature.

4.3.2 DQ2: Which architectural patterns exist in the system?

All existing architectural patterns that exist in the system, and how they are traced to implementation are shown in this explanation. The explanation consists of three tabs, an overview, the physical view, and the development view.

The overview tab shows all architectural patterns in the system and the roles that together make up the pattern. For example, a client-server pattern would consist of two roles; the client and the server. This view simply serves as a means to quickly get a glance over the existing patterns in the system (See figure 4.20).

The physical view tab shows the architectural patterns and roles in the system (same as the ones from the overview), connecting to the physical view, development view and down to the implementation classes (see figure 4.22). It shows in which physical devices different parts of the software components are located. For example; by selecting the web application server role, paths to the physical devices which implement this role, and implementation classes that reside within these physical



System Explanation Composer

Select question

Generate explanations of the system by selecting one of the questions.

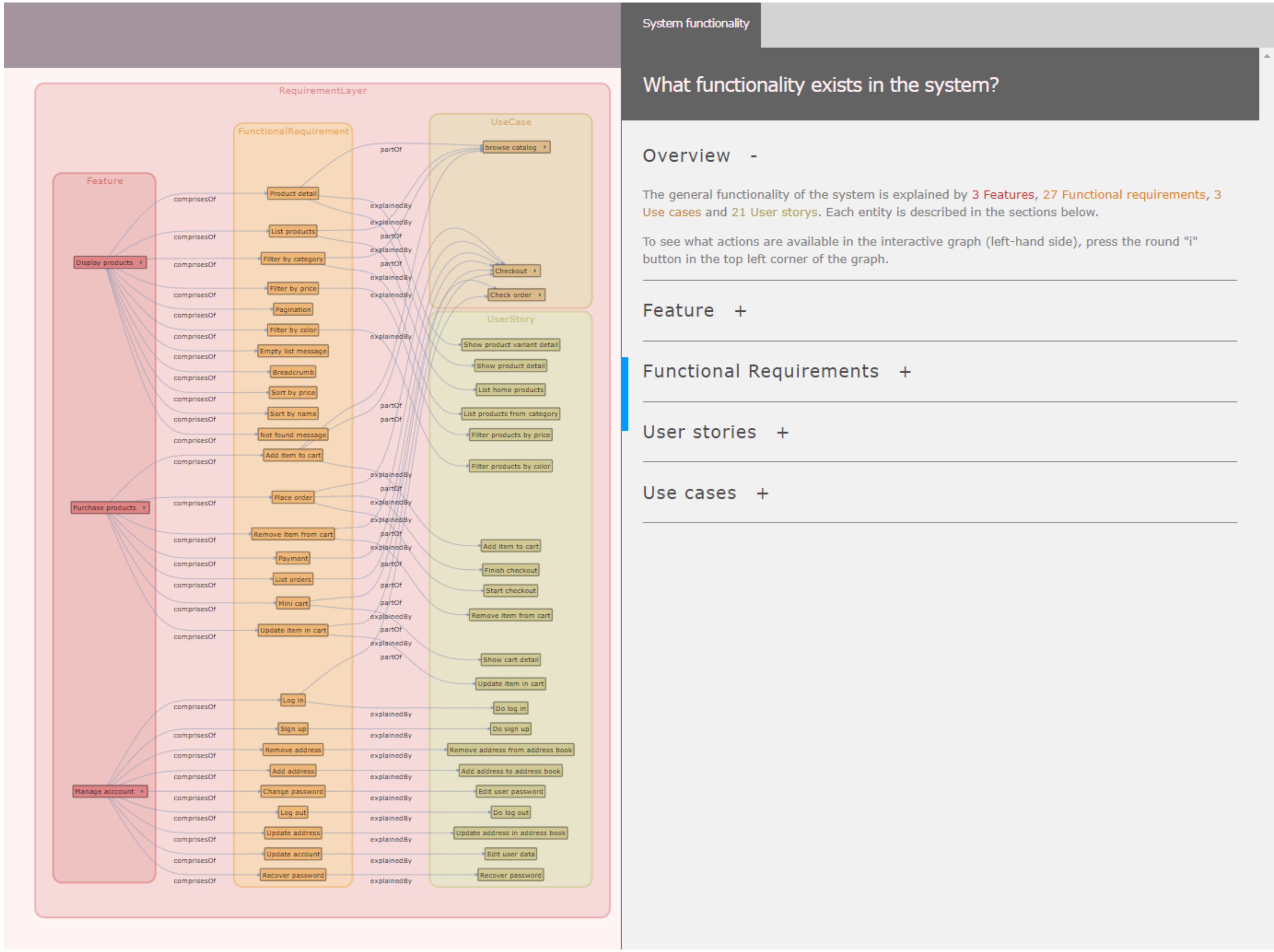
How is this feature mapped to its implementation?

Figure 4.17: The input to SEC

devices, are highlighted. It can also be used the other way around; by selecting an implementation class, we can highlight the physical device it resides in and the architectural pattern(s) and role(s) it is part of.

The development view tab shows the same architectural patterns as the previous two views, but bypasses the physical layer, instead, the roles connect directly to the development view (see figure 4.21). Much like the physical view tab, it enables tracing from patterns to implementation classes but from a software perspective. This view focuses mainly on providing information to developers, but can also be useful for architects. Here are some examples of a few ways this view can be used.

- It can be used to see the architectural responsibility of a specific implementation class. This is done by highlighting an implementation class, and tracing back to the architectural patterns.
- It can be used to see all classes that together serve a responsibility of an architectural pattern. This is done by highlighting one of the roles and tracing the roles to the implementation classes.
- If a system design has been made before implementation, this tab can be used as a means to see which parts of the system design has not been implemented yet.
- It can be used to check whether certain implementation classes have taken on more responsibility than they should. E.g classes should in most cases not implement both model and view functionality.
- It can be used to see which classes are affected by specific architectural patterns. This knowledge can be useful if the system architecture is in need of changes.



System functionality

What functionality exists in the system?

Overview -

The general functionality of the system is explained by 3 Features, 27 Functional requirements, 3 Use cases and 21 User storys. Each entity is described in the sections below.

To see what actions are available in the interactive graph (left-hand side), press the round "i" button in the top left corner of the graph.

Feature +

Functional Requirements +

User stories +

Use cases +

Figure 4.18: The generated explanation for system functionality

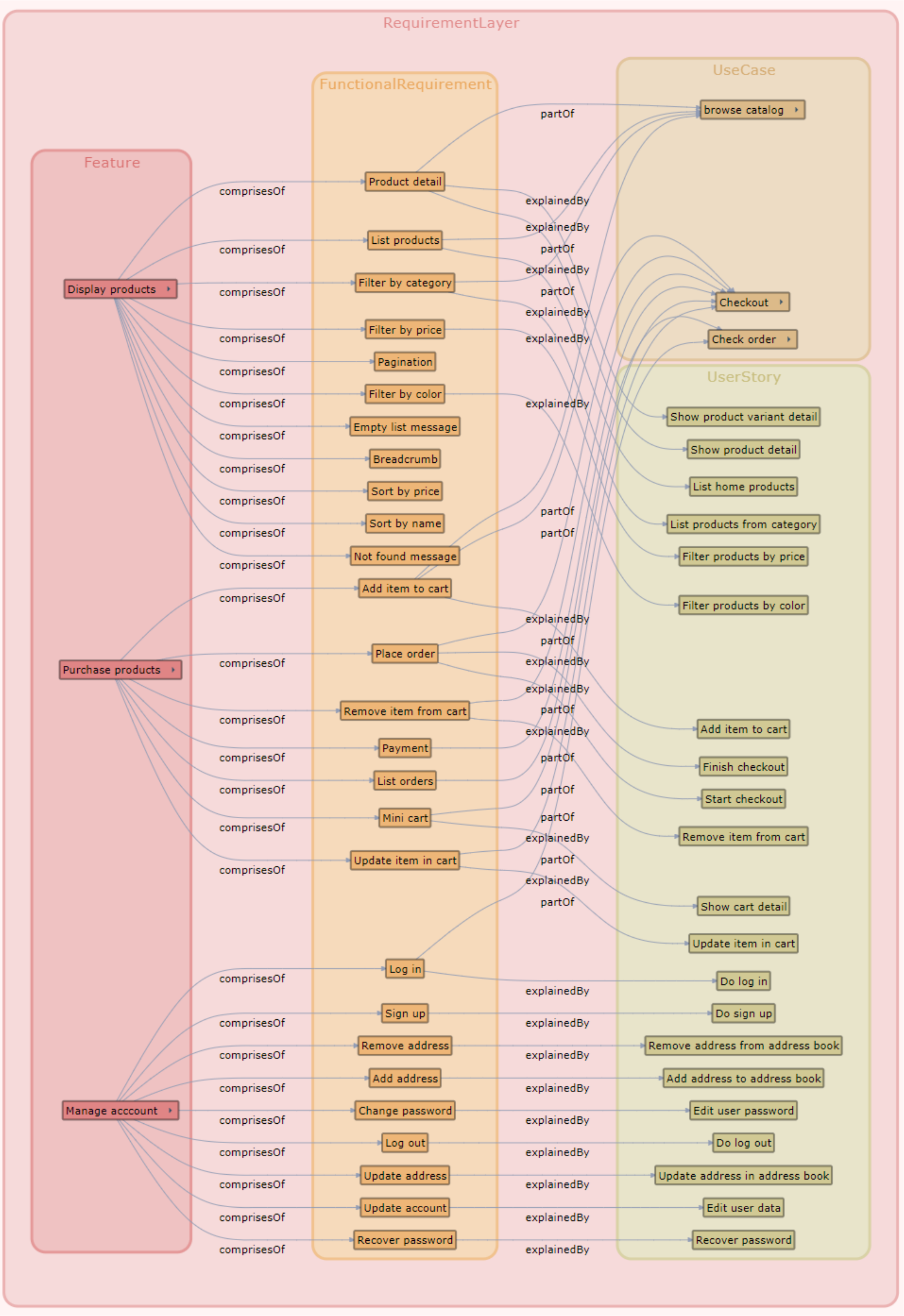
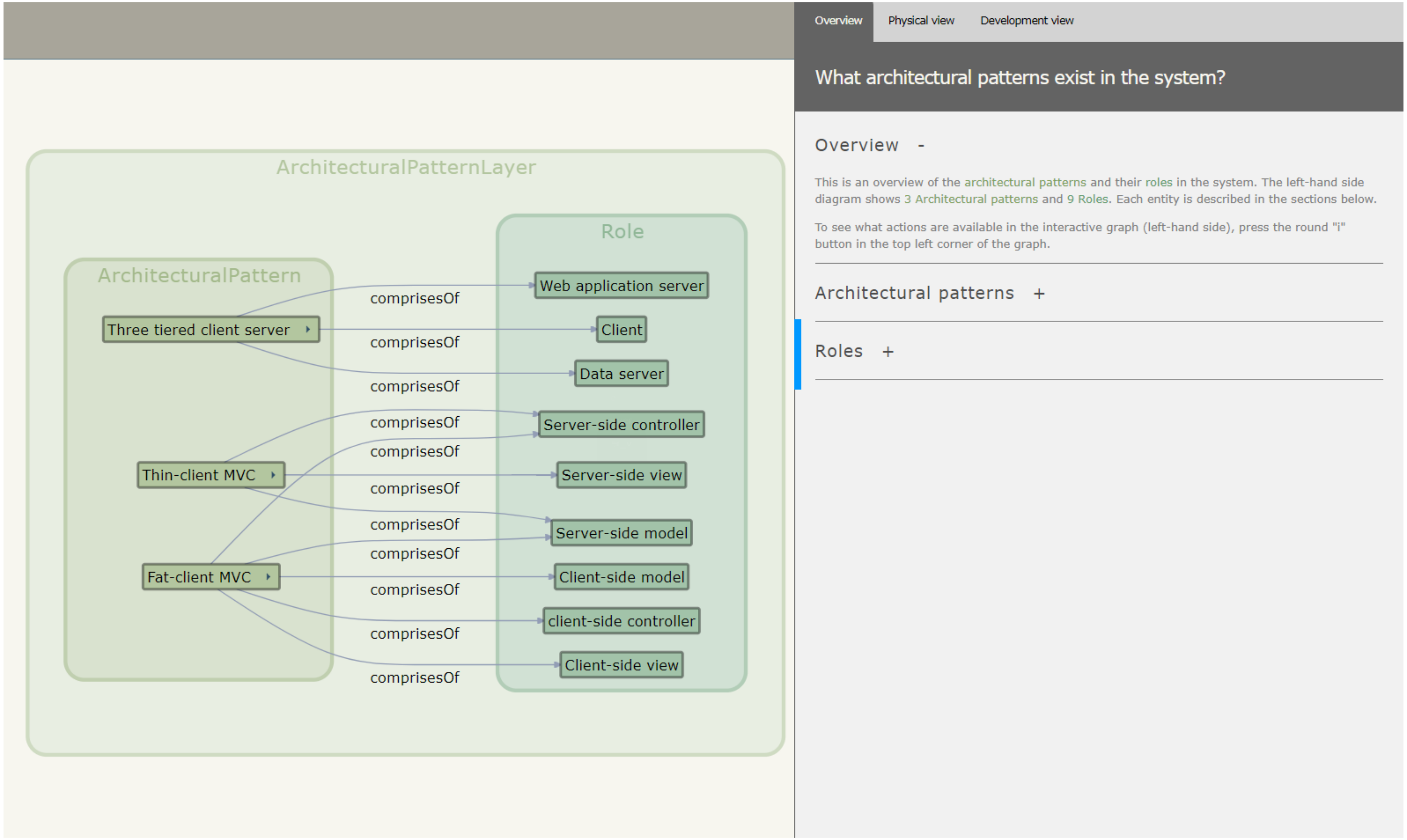


Figure 4.19: (Q1) A closer view of the system functionality visualization.



What architectural patterns exist in the system?

Overview -

This is an overview of the architectural patterns and their roles in the system. The left-hand side diagram shows 3 Architectural patterns and 9 Roles. Each entity is described in the sections below.

To see what actions are available in the interactive graph (left-hand side), press the round "i" button in the top left corner of the graph.

Architectural patterns +

Roles +

Figure 4.20: (Q2) An overview of the architectural patterns

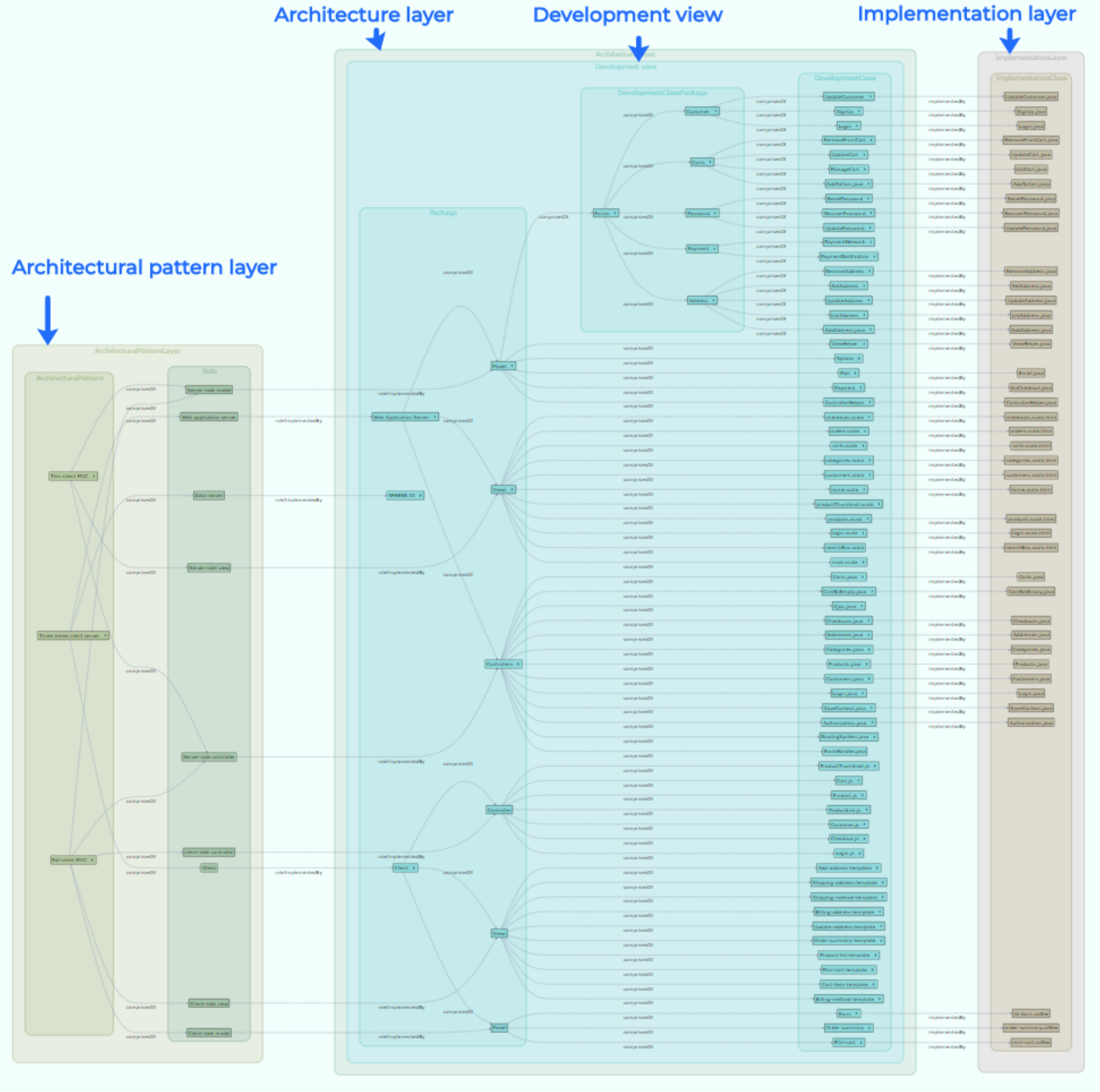


Figure 4.21: (Q2) A closer view of the architecture and implementation via the development view

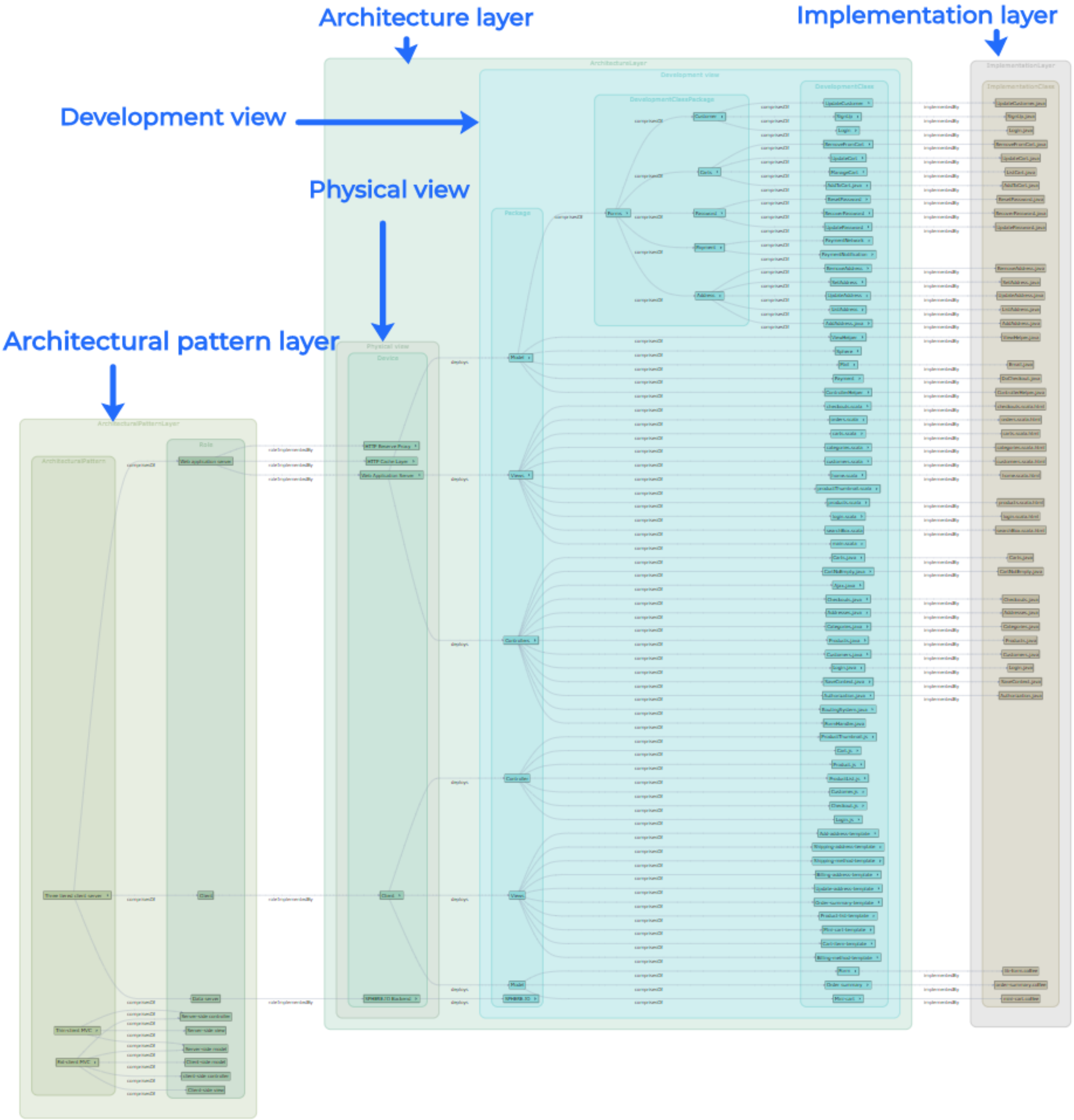


Figure 4.22: (Q2) A closer view of architecture and implementation via physical layer

4.3.3 DQ3: What is the role of this feature?

The role of a feature can be interpreted as, what functionality does the feature provide, and what is its responsibility in the system? The answer to this question is almost the same as "DQ1: Which functionalities exist in the system?", except it focuses on a single feature. Once the functionality of the system evolves and increases in volume, this question can be useful to filter out some noise.

4.3.4 DQ4: How is this feature mapped to its implementation?

This question targets the mapping between the problem domain to the implementation level. The problem domain consists of features, requirements and logical view while the implementation level consists of development view and implementation classes. Constructing answers to this question thus concerns querying the entities and relations of these ontology classes. The explanation to this question consists of three tabs; overview, detailed view and pattern view. They are more elaborately described below.

The overview tab in this explanation shows a brief answer to the selected question (see figure 4.23). This can be verified by looking at the top of the side-bar, which displays the developer question and by reading the summary, which mentions the feature in focus. The explanation shows the entity and all requirements related to it. It then shows that the requirements relate to some development classes (which are abstracted away to hide unneeded details) in the architecture. Lastly, these development classes relate to implementation classes. In a broad sense, the overview tab is showing what required functionality and what implementation classes are related to the selected feature "Purchase products". What is useful from this view, is that all implementation classes related to "Purchase products" can easily be identified, which makes it easier to find implementation classes. This can be useful for many tasks, such as for maintenance (i.e re-factoring, change tasks) or learning the system.

The detailed view tab, much like the overview, also shows how the selected feature is implemented. What differs from the overview is that it explains the relations in more detail (see figure 4.24). The architecture is no longer abstracted away, and the specific logical and development classes in the architecture cluster are shown. This enables the ability to more specifically trace implementations of individual functionalities of the feature.

The pattern view tab does not directly answer the developer question "How is this feature mapped to its implementation". Instead, it is a complementary tab that can be useful to better understand how the implementation classes are architecturally structured (see figure 4.25). This tab has shifted focus away from the feature entity and instead uses the implementation classes generated from the other two views, i.e the implementation classes that are related to "Purchase products", as input. From this input, SEC queries the ontology to find out how they are mapped to architectural patterns. As can be seen from figure 4.25, the implementation classes map to the development view, much like the other views, and further into the roles the classes play within each architectural pattern. This view can be useful for developers and architects alike to gain insight to the roles of the classes (e.g do they belong to the client or the server) which can assist developers in writing code that properly follows the architecture of the system or architects in identifying which parts of the systems will be affected by proposed architectural changes.

4.3.5 DQ5: What is the behavior of this feature?

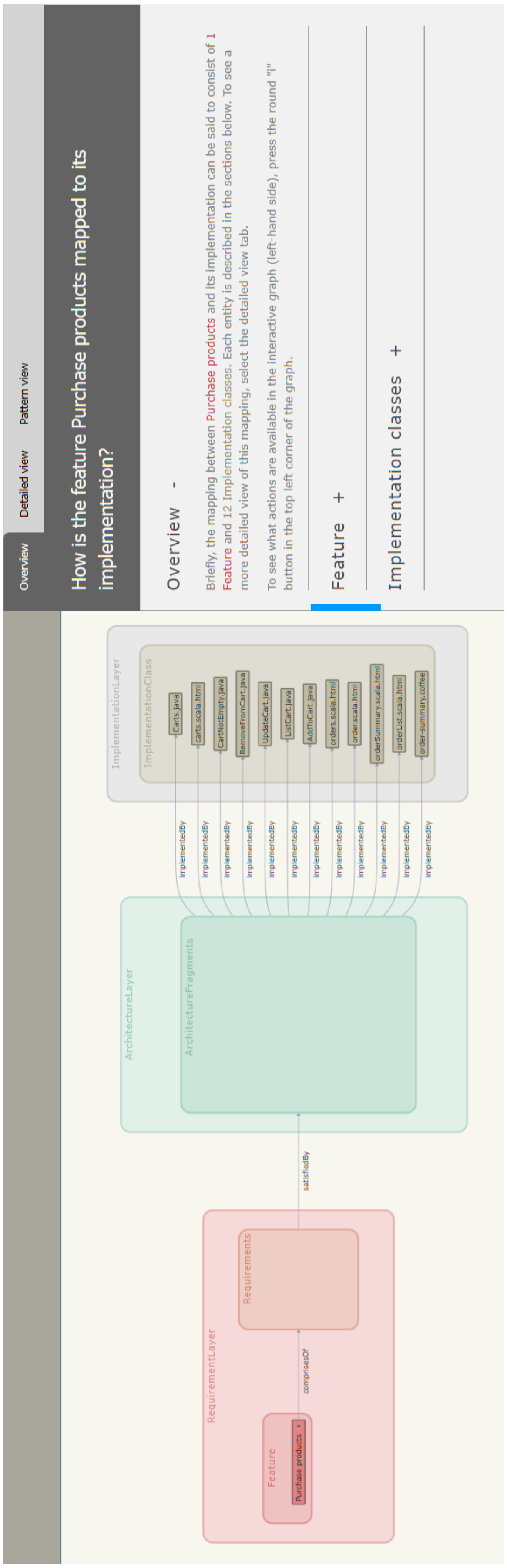
Explaining system behavior is a bit different from explaining system structure. Since the current iteration of the ontology lacks the vocabulary for expressing system behavior, a workaround has been made to express it. For this purpose, SEC relies on existing behavioral diagrams. SEC shows the system structure that relates to the chosen feature, and then the behavioral diagrams that in turn relate to the system structure.

As with all tabs in this explanation, **the functional viewpoint tab** starts with the feature in focus, "Purchase products", positioned on the leftmost part of the visualization (see figure 4.26). Other functional entities and their relations to "Purchase products" are structured, just as in DQ5. In this view, what can be counted as behavior are mainly the use cases which explain system behavior for specific cases. Since the ontology lacks the vocabulary to actually express the behavior itself, SEC instead visualizes diagrams that are related to the use cases since these diagrams would with high probability express the case behavior.

The logical viewpoint tab shows how "Purchase products" and its requirements relate to the structures in the logical view which in turn are expressed by behavioral entities (see figure 4.27). In this case, the behavioral entities are states. The ontology does not model the exact behavior and relations between the shown states. It only models the structural entities these behavioral states express. To show system behavior, SEC relies instead on existing diagrams to show it. As such, these state entities are related to diagrams, which in turn explain how these states work together. The main use of this tab is that it in detail shows tracing between structural and behavioral entities. This allows the user to ask questions such as; "what behaviors does the cart have?", or "which logical entities are affected by this behavior?".

The development viewpoint tab shows the system behavior of "Purchase products" in more detail, explaining the behavior of structural classes from the development standpoint (see figure 4.28). The system functionality is connected to classes and packages of the development view, which in turn are connected to the behavioral entities and lastly to the diagrams. In this case, these behavioral entities are lifelines from sequence diagrams. Same as with the logical view, the ontology does not express lifeline relations, instead, SEC leaves this task to the existing diagrams.

The UI viewpoint works the same way as above except describing the behavior



Overview Detailed view Pattern view

How is the feature Purchase products mapped to its implementation?

Overview -

Briefly, the mapping between Purchase products and its implementation can be said to consist of 1 Feature and 12 Implementation classes. Each entity is described in the sections below. To see a more detailed view of this mapping, select the detailed view tab.

To see what actions are available in the interactive graph (left-hand side), press the round "i" button in the top left corner of the graph.

Feature +

Implementation classes +

Figure 4.23: (Q4) An overview of mapping between feature and implementation

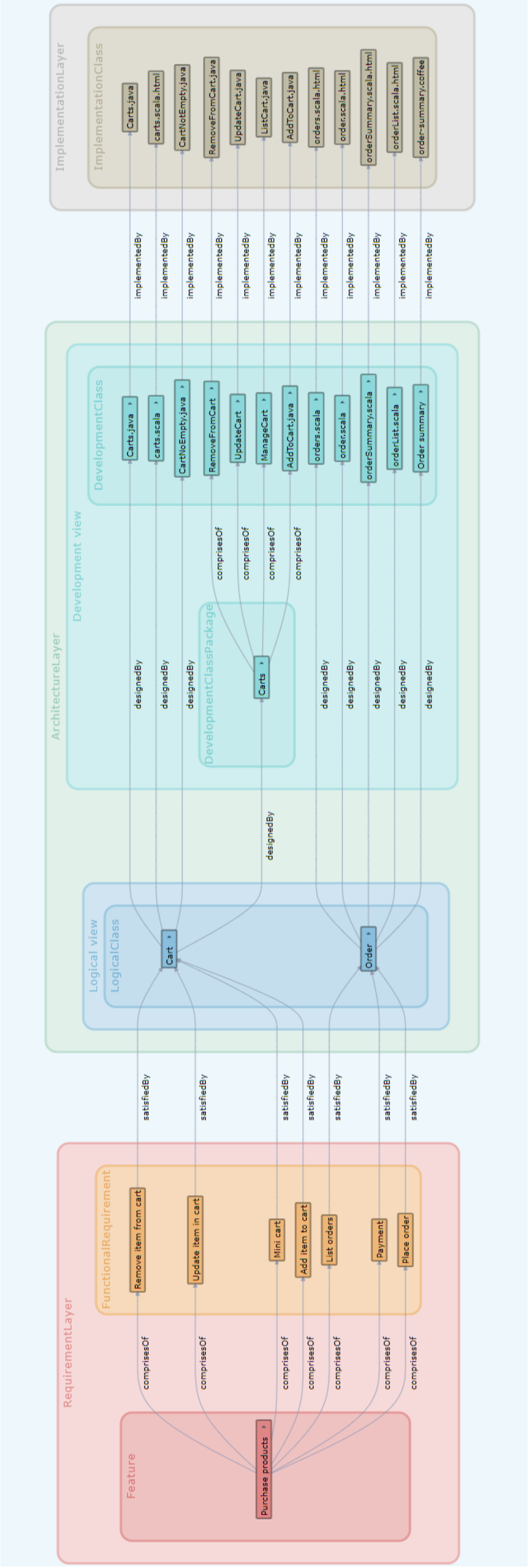


Figure 4.24: (Q4) A detailed view of the mapping between feature and implementation

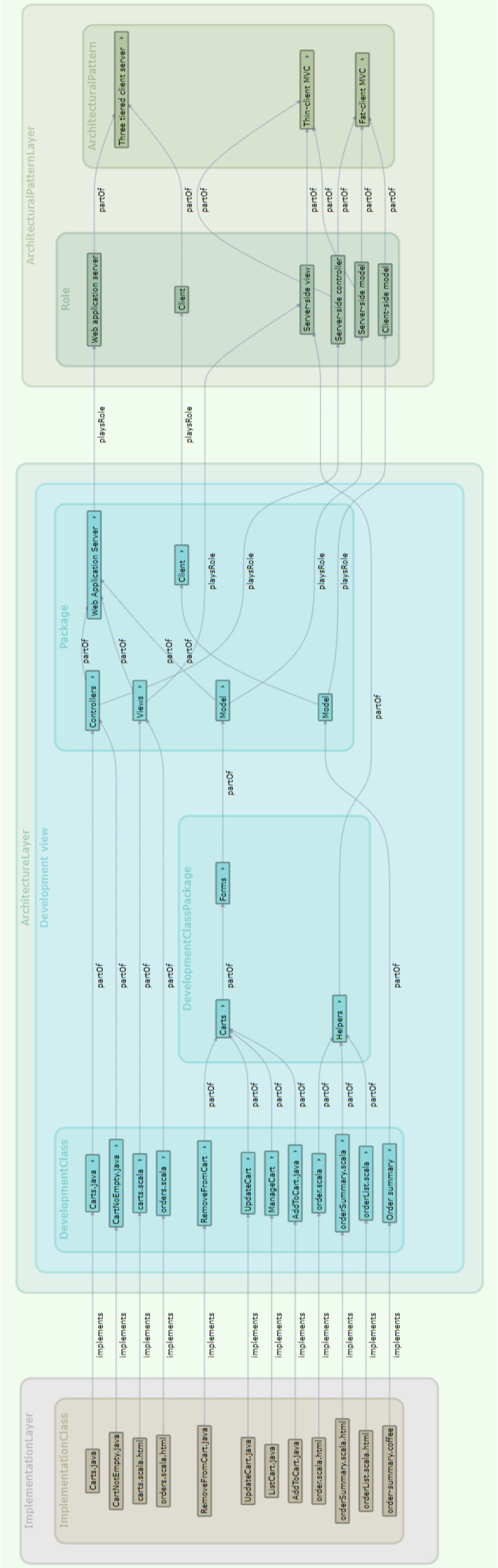


Figure 4.25: (Q4) A detailed view of how the implementation classes maps to architectural patterns

of the UI classes (see figure 4.29).

4.3.6 DQ6: What is the rationale behind the choice of this architectural pattern?

The pattern we have chosen to demonstrate for this question is "Thin-client MVC" (see figure 4.30). This explanation consists of one tab showing how "Thin-client MVC" relates to design options, which in turn relates to other design options, non-functional requirements, constraints, arguments, and assumptions. For this pattern, there are no defined assumptions and thus they are not visualized.

Choosing to use architectural pattern "Thin-client MVC" is a result of the design option "Choice of using mixed fat and thin client MVC". This design not only results in using the "Thin-client MVC" but also the "Fat-client MVC", as the name states, it uses a mix of both. Furthermore, the choice of using this design is motivated by other design options, requirements, arguments, constraints or assumptions.

For this specific case, the design option "Choice of using mixed fat and thin client MVC" has three types of artifacts that motivate it.

1. Different arguments, each of them arguing for why this design was chosen.
2. Previous design options that were already made.
3. Non-functional requirements of the system

These together motivate for why this design option was made for the system, and in turn, motivates why the architectural pattern "Thin-client MVC" was implemented in the system.

4.3.7 DQ7: How is this architectural pattern implemented?

This question is similar to DQ1. The only difference is that it is specific for one architectural pattern. It is useful when the user chooses to study a single architectural pattern as it hides information from the other patterns.

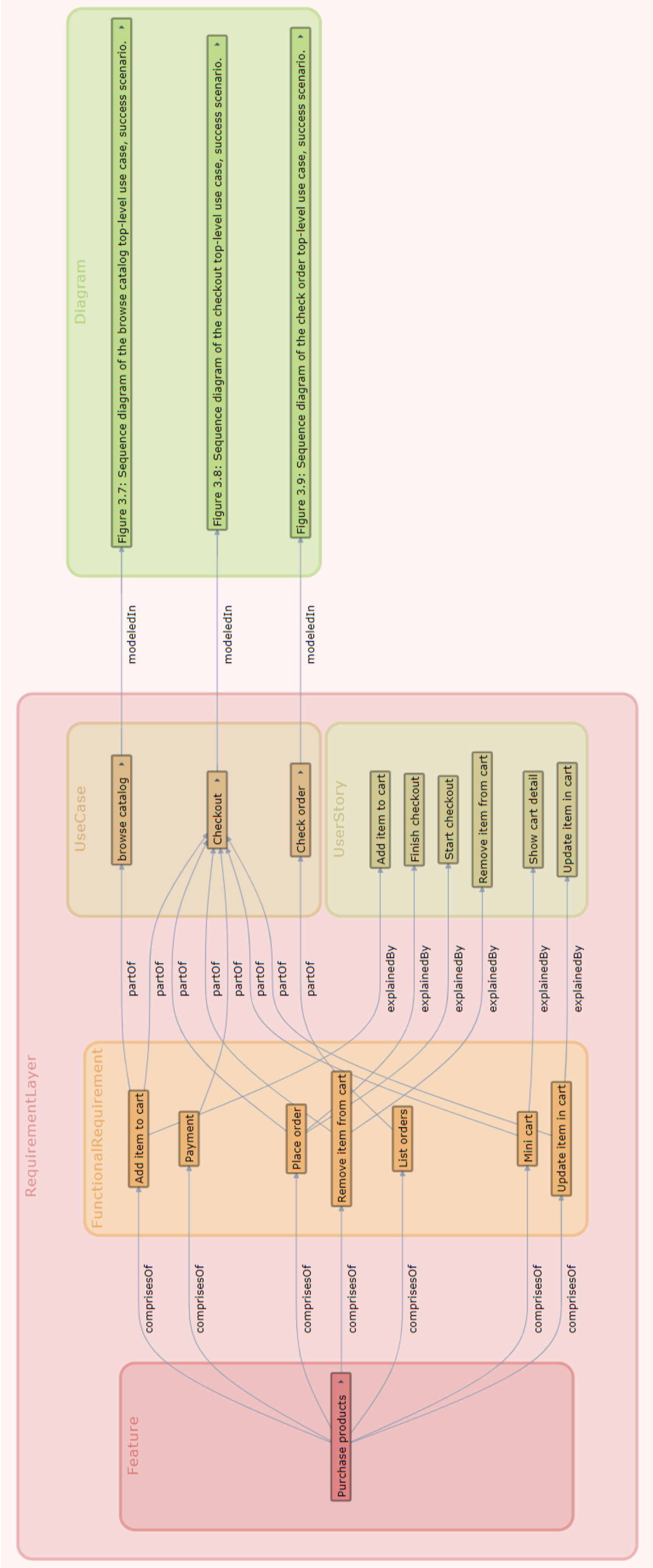


Figure 4.26: (Q5) The behavior from a functional perspective

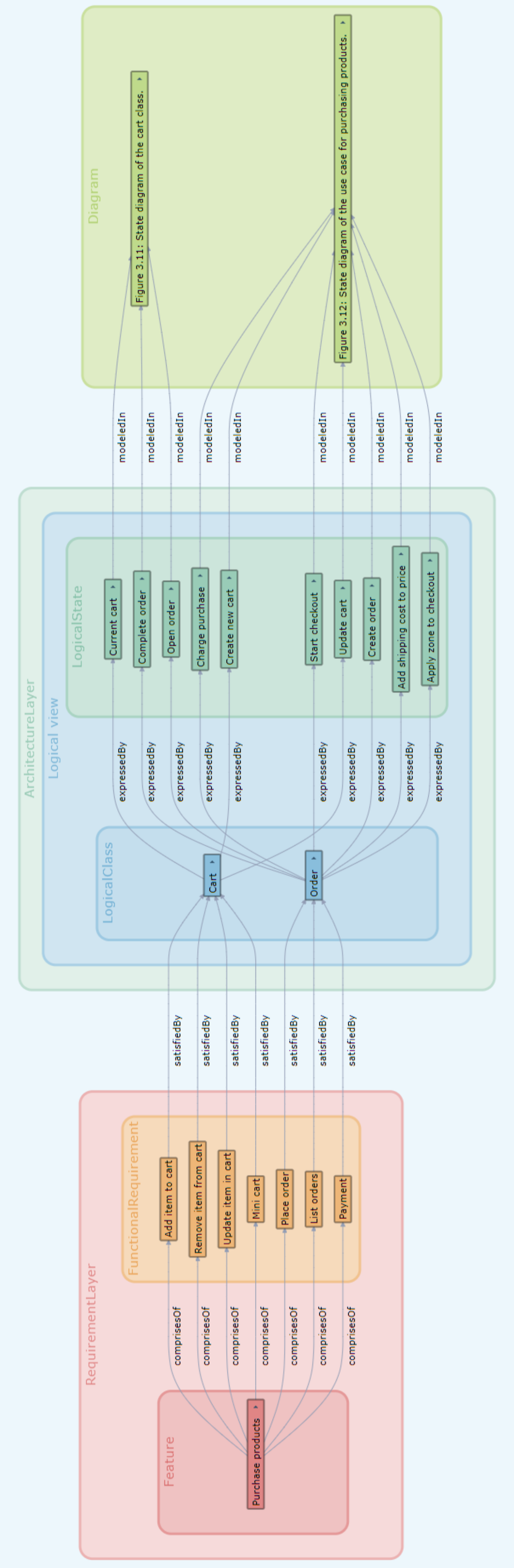


Figure 4.27: (Q5) The behavior from a logical perspective

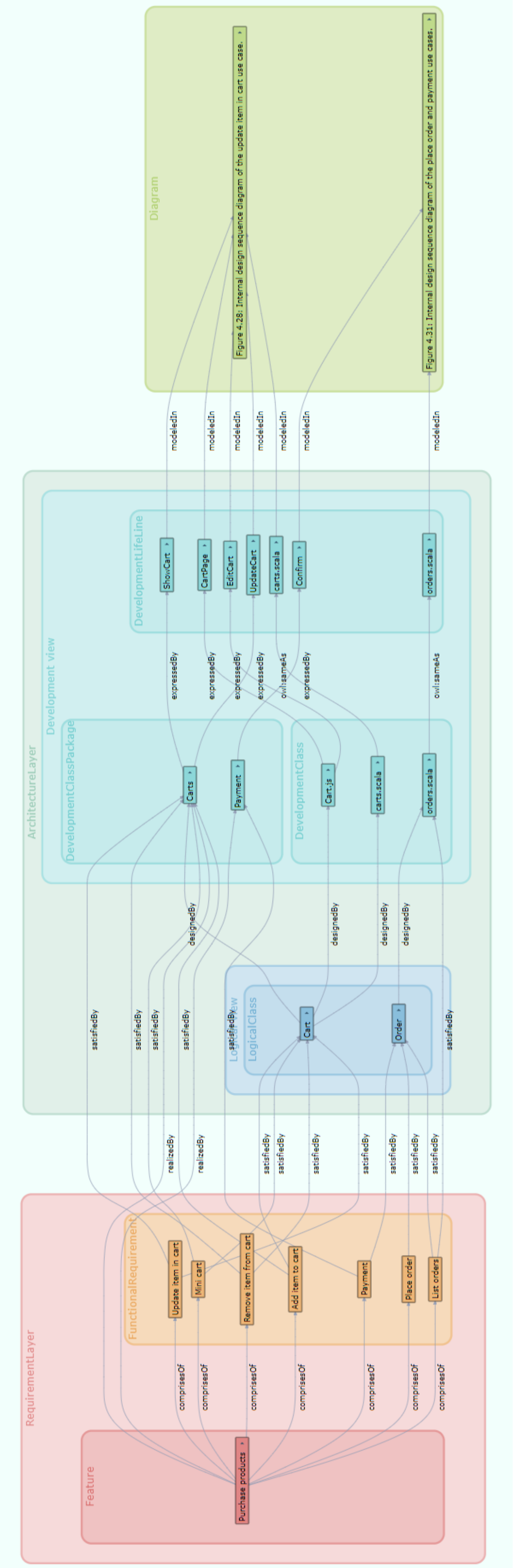


Figure 4.28: (Q5) The behavior from a development perspective

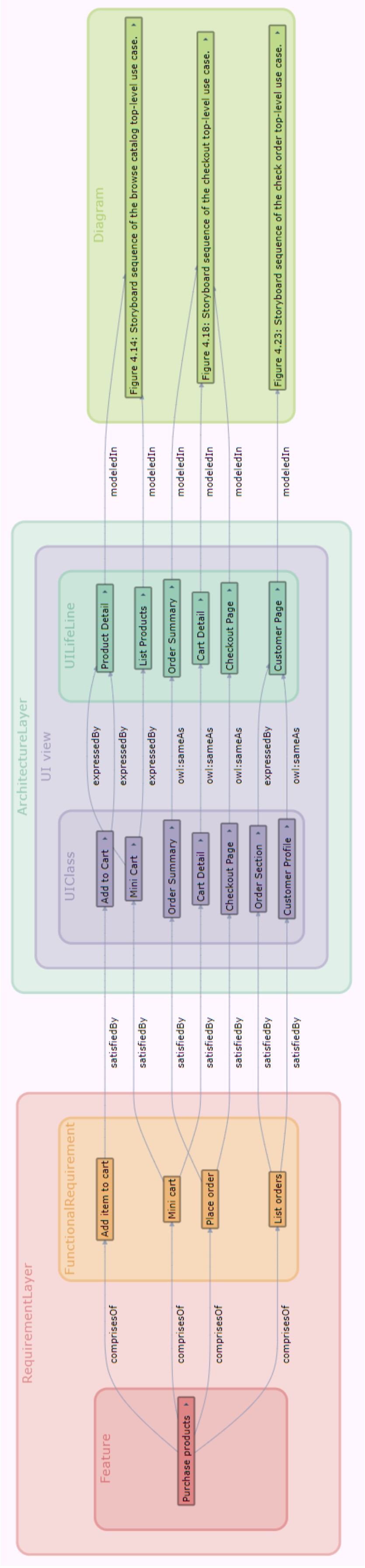


Figure 4.29: (Q5) The behavior from a UI perspective

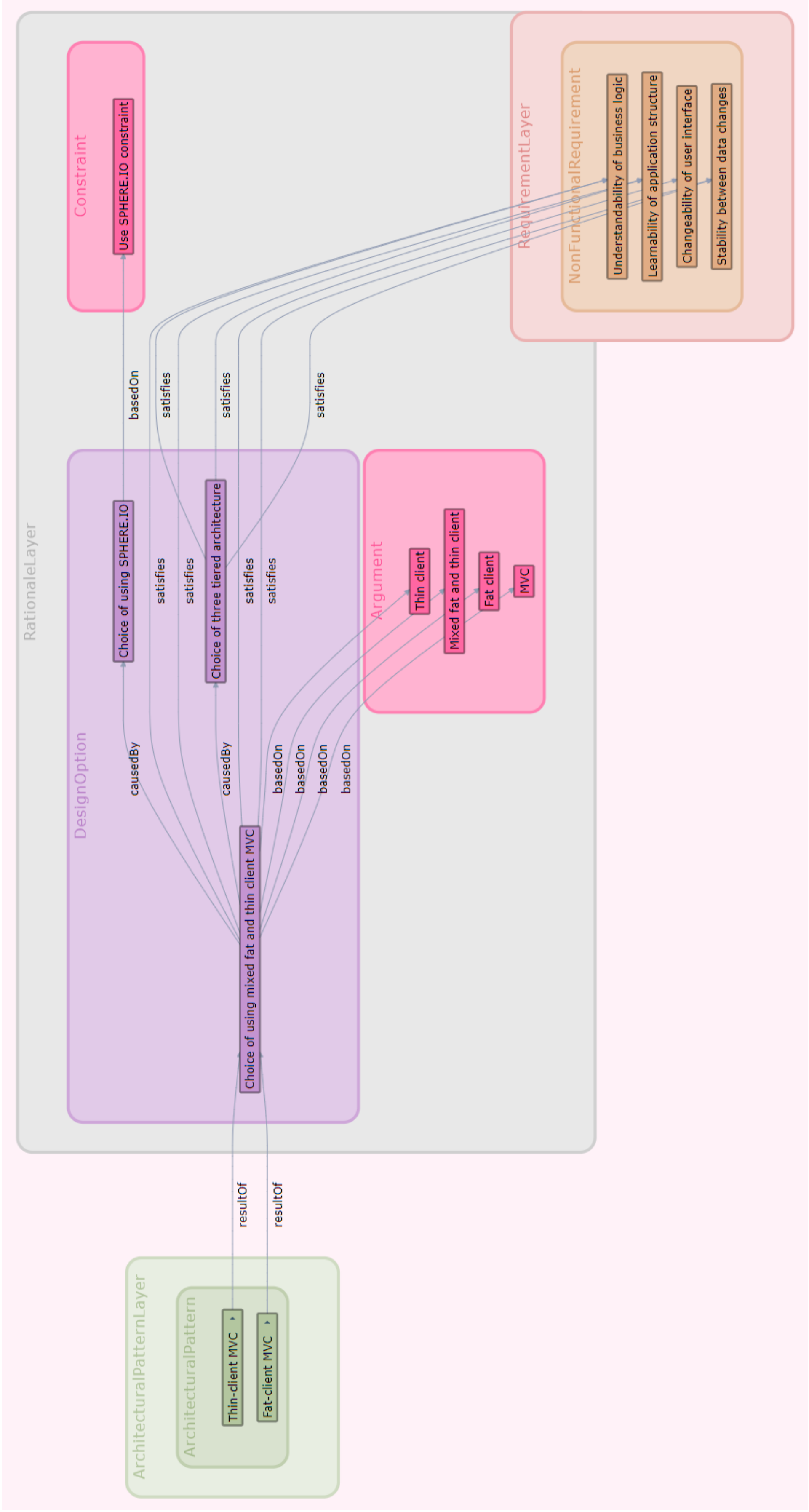


Figure 4.30: (Q6) Rationale for the architectural pattern "Thin-client MVC"

5

Results

This chapter will present the result of the evaluations and some analysis of these results to address our second research question; whether explanations and presentation of existing software artifacts are useful for developers. We answer two sub-questions: 1) Do the explanations presented by SEC tool increase developers' efficiency in performing development related tasks? 2) How are these explanations perceived by developers subjectively?

5.0.1 Results from usability tests

Usability tests were conducted to determine whether explanations provided by SEC increase developers' efficiency in performing development related tasks (for more information about the design of the evaluation, refer to section 3.7).

Efficiency is a combined measure between how fast the participants solved the tasks and how well they scored. As such, to measure whether participants were more efficient in solving development-related tasks using the tool than using the documentation, participants of the evaluation were both timed and scored for all tasks. Also recall that one observation for task times was lost, and thus the sample size for the task times is 10 while the sample size of the task scores is 11. The sections below will in more detail present the results of the timing and scoring.

5.0.1.1 Task times

Each task was timed separately, both when the participants used the documentation and when they used the tool. Since the tasks for documentation and the tasks for the tool could be mapped close to a 1 to 1 ratio, this made it possible to compare time efficiency between them in three ways: (1) Comparing their the total time between the two phases, (2) comparing the total time per task type and (3) performing a t-test to check the significance of the results. See table 5.1 for results.

There are two ways to view the total time comparison. The first way to view it is using a bar chart comparing time differences between using the tool and using documentation. From figure 5.1, we can state the following:

When comparing the total time of solving tasks using tool versus using documentation, all participants completed their tasks faster using the tool.

Participant	Tool time	Documentation time
P1	1361	1532
P2	1628	2157
P3	813	1408
P4	1184	1354
P5	703	1012
P6	1002	1523
P7	1040	1760
P8	943	2084
P9	860	1604
P13	1149	1504

Table 5.1: Tool time and documentation time results

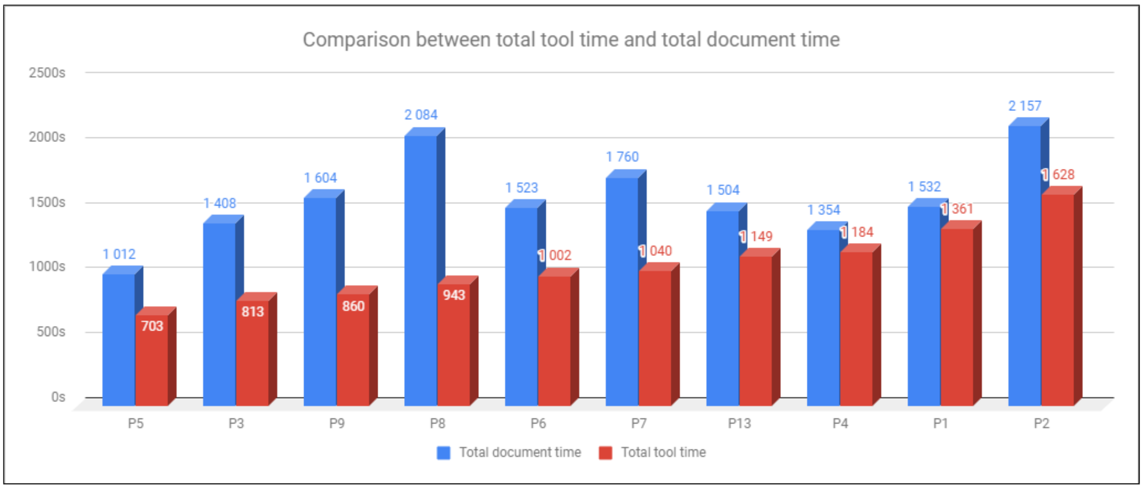


Figure 5.1: Comparison between documentation time and tool time

The second way to view this data is to plot the documentation time on one axis, and the tool time on the other. This is visualized in figure 5.1. In the figure, several lines have been drawn, and the section between the lines are colored. Each section marks a certain time ratio of using the tool compares to using documentation. The sections of the figure can be interpreted in the way shown in table 5.2. From this graph, we can observe that:

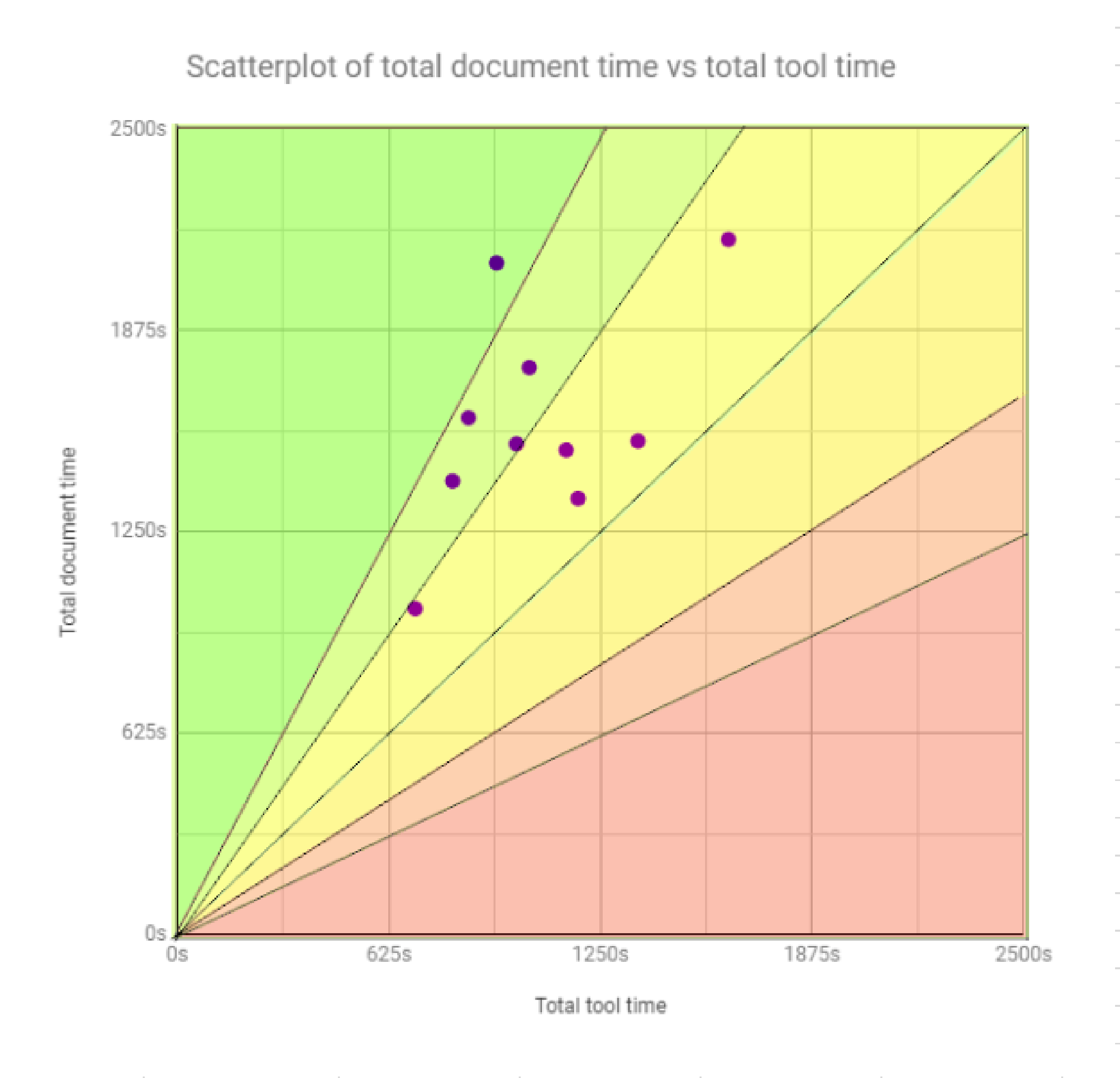


Figure 5.2: Scatter-plot showing tool time versus documentation time

Color	Tool to documentation time ratio
Green	2+
Lime	1.5-2
Yellow	1.5-0.75
Orange	0.75-0.5
Red	0.5-

Table 5.2: Colored sections of figure 5.2

Half of the participants were more than 1.5 times faster to solve their given tasks using the tool.

The other way to compare the time efficiency is time comparison by task type, requirements, architecture, behavior, and rationale (see figure 5.3). This comparison shows how each task type compares to another, averaged over all participants. It can show which areas the participants perform better when using the tools, and which areas they perform worse. It also shows the magnitude of the difference. Figure 5.3 shows the following:

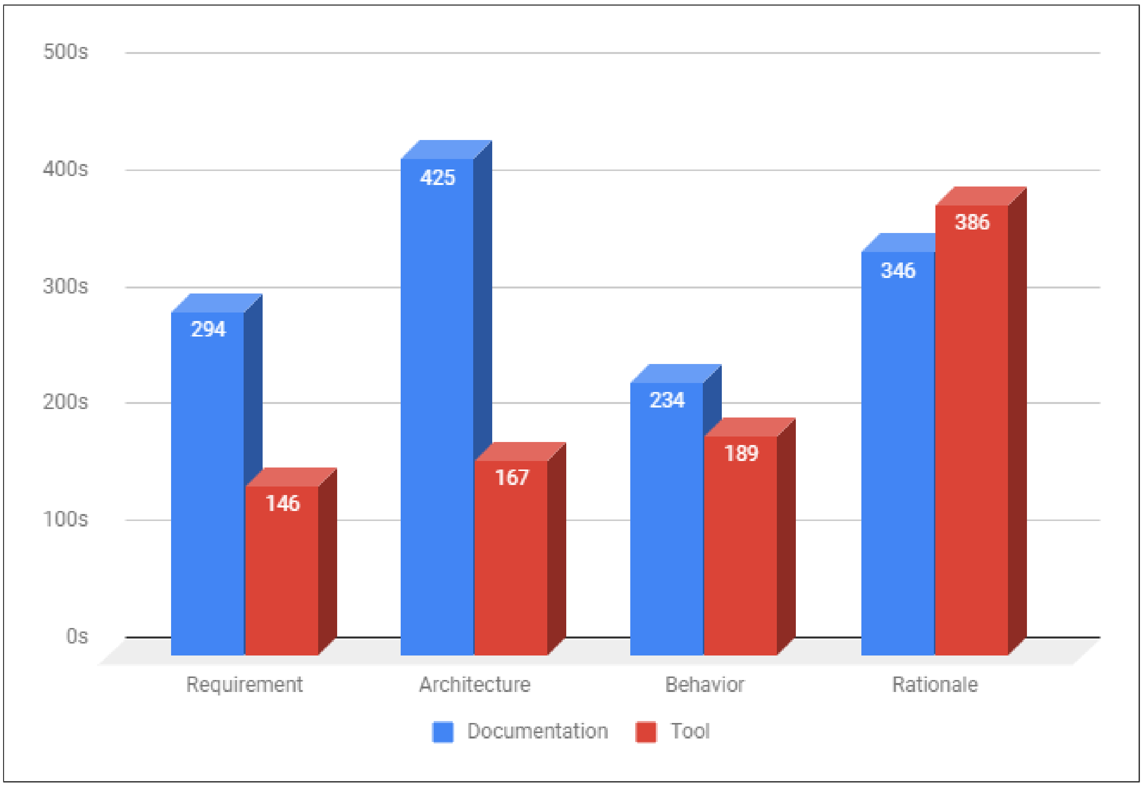


Figure 5.3: Time average by task type

On average, the participants completed the requirements and architecture tasks twice as fast using the tool, while the performance differences in the behavior and rationale sections were very small. The participants perform slightly better using SEC in the behavior section, and slightly worse in the rationale section

Lastly, a **t-test** was used to check whether the participants' time differences between using the tool versus using the documentation were significant (see appendix B for data). However, before conducting the t-test, we checked for normality using a Shapiro-Wilk test and QQ-plot and checked for equal variances using the F-test. The results showed that both data sets follow a normal distribution and their variances are equal (see appendix A). As such the following hypotheses were set up

H0: The participants solved tasks faster using the documentation
H1: The participants solved tasks faster using the tool

Statistic	**Result (seconds)**
M_{tool}	1068.30
M_{doc}	1593.80
SD_{tool}	275.23
$SD_{documentation}$	338.73
SD_{pooled}	308.62
Single-tailed Welch t-test p-value	0.00065
Hedges' g	1.55

Table 5.3: Table containing the results of the t-test and Hedge's g

With this knowledge in mind, the significance level was set to 0.05 and a single-tailed T-test was used to test the significance of **H0**.

The t-test (see table 5.3) shows that the probability that the participants solved tasks faster using the documentation is 0.065 percent. This strongly suggests that the participants, in fact, solved the tasks faster using SEC. However, the significance of the results often does not provide the entire picture. Another way to view the results is how much faster participants solved the tasks using SEC. To gauge the power of the effect, the effect size was calculated using Hedges' g, with correction for small sample sizes using the following formula:

$$g = \frac{M_{tool} - M_{doc}}{SD_{pooled}} * \frac{N-3}{N-2.25} * \sqrt{\frac{N-2}{N}}$$

where M_{tool} and M_{doc} are the means of the two samples, SD_{pooled} is the pooled standard deviation for the two samples and N is the combined sample size. SD_{pooled} was calculated according to Cohen's "correct" pooled standard deviation formula in the following manner:

$$SD_{pooled} = \sqrt{\frac{(n_{tool}-1) * SD_{tool} + (n_{doc}-1) * SD_{doc}}{n_{tool} + n_{doc} - 2}}$$

where n_{tool} and n_{doc} are the sizes of the two samples and SD_{tool} and SD_{doc} are standard deviations of the two samples. According to general guidelines, the value of Hedges' g can be interpreted using a specific rule of thumb; 0.2 or lower for a small effect, around 0.5 for a medium effect, 0.8 or higher for a large effect. The calculation of Hedges' g resulted in the value 1.55 (see table 5.3), which is a large effect.

To summarize the task times, both the individual data analysis and the t-test show that participants solved their given tasks faster using SEC and Hedge's g shows that power of the difference between the two samples

is large. With these results in mind and without discussing threats to validity (which we will do later in the discussion) we can conclude that participants solved their given tasks faster using SEC than using the case documentation.

5.0.1.2 Task scores

In addition to timing, the tasks were also scored. The scoring can be viewed in the following way: (1) compare total score between the two phases (see figure 5.4 and 5.5), (2) comparing the score by task type (see figure 5.7) and (3) comparing the score level distribution. What differs from the timing, is that in this case, we chose not to perform a t-test, since normality of the samples could not be assumed.

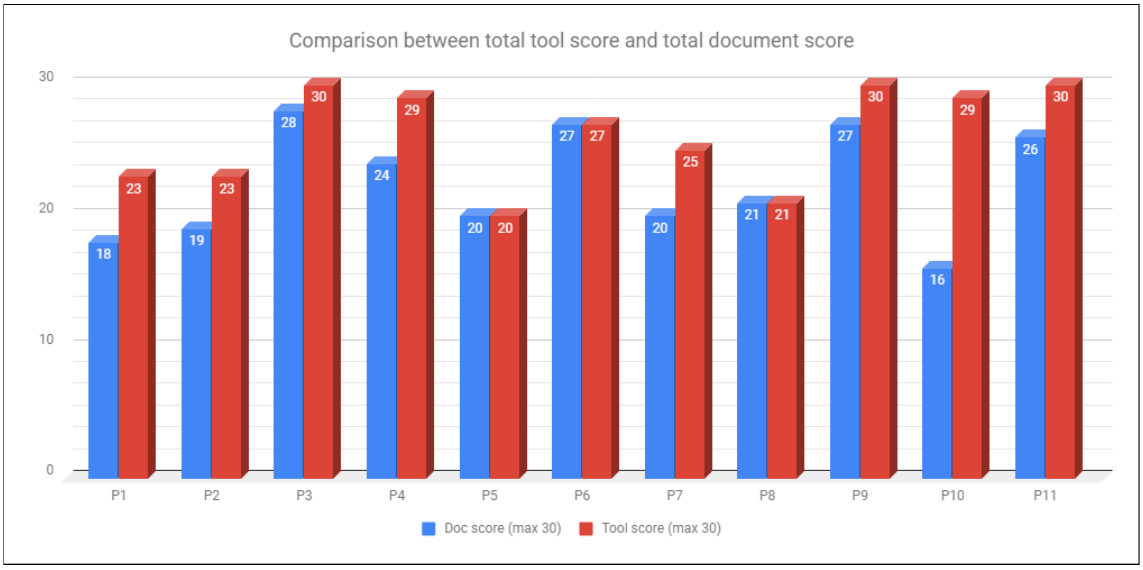


Figure 5.4: Comparison between documentation score and tool score

From figure 5.5, it can be seen that when participants used the tool, they scored higher than or equal to their scores when using documentation in every instance. Also, in no instance did the participants score full points using the documentation, while three participants scored full points using the tool.

Figure 5.5 also shows that most participants scored from 1 to 1.5 times better using the tool. One participant did exceptionally well, and the rest had the same score on both tool and documentation.

Just as with the time comparison by task type, the task types for the scores are requirements, architecture, behavior, and rationale When comparing the score per task types (see figure 5.7), on average the participants scored higher using SEC for every task type. When comparing SEC score to the case documentation score, participants scored rather well on requirement and moderately on architecture and rationale while the behavior had the lowest score differences. The scoring difference between tool and documentation is noticeable, although not quite as large as the

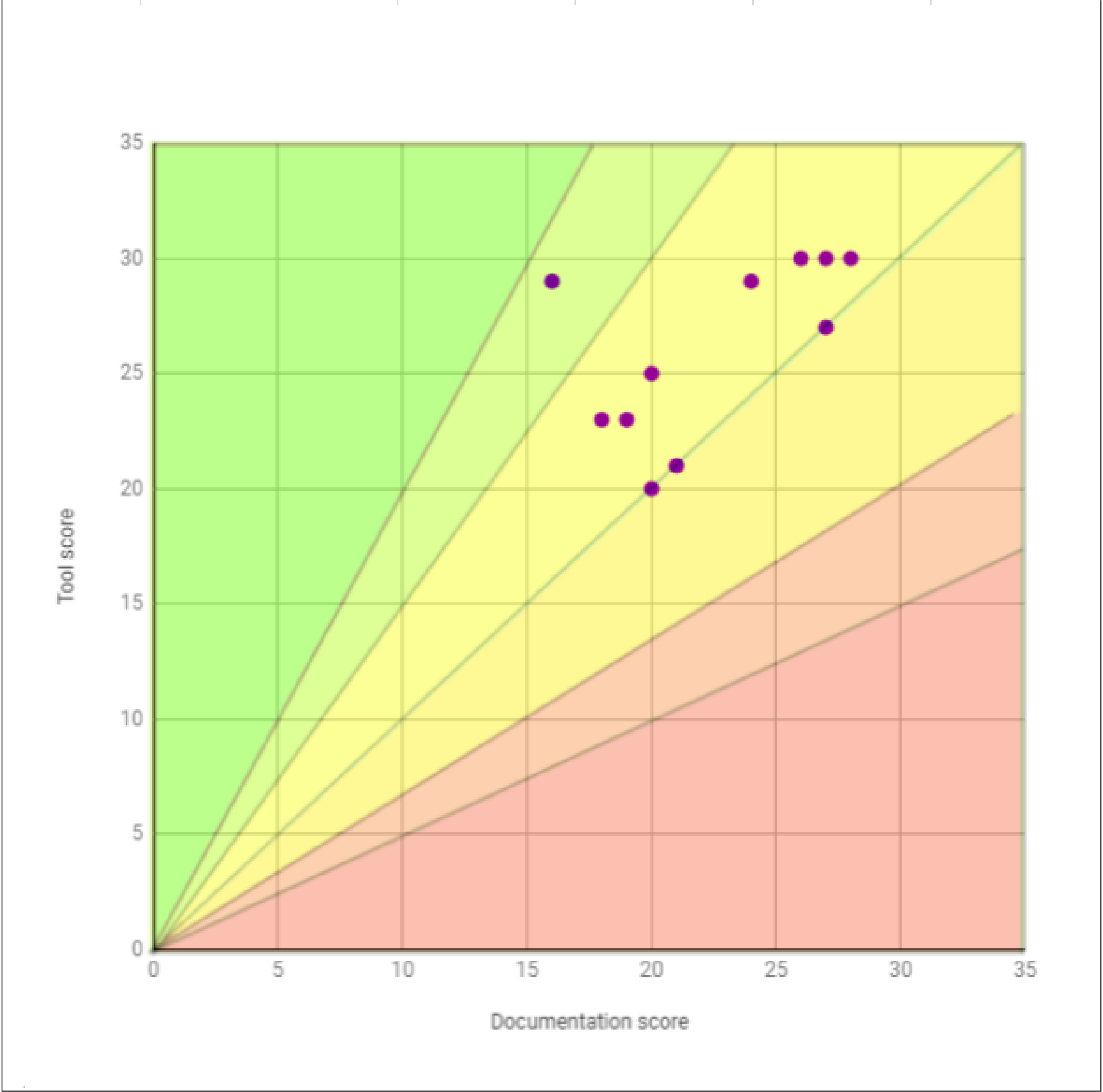


Figure 5.5: Scatter-plot showing tool score versus documentation score

timing results (see section 5.0.1.1).

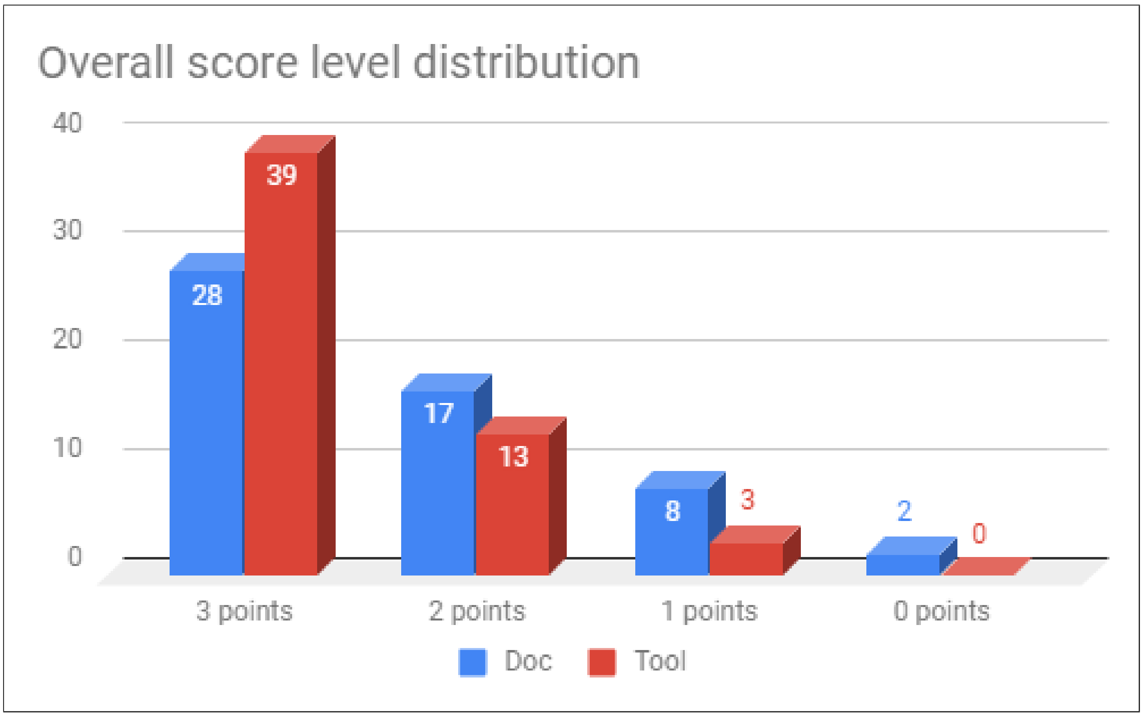


Figure 5.6: Score distribution of tool versus documentation

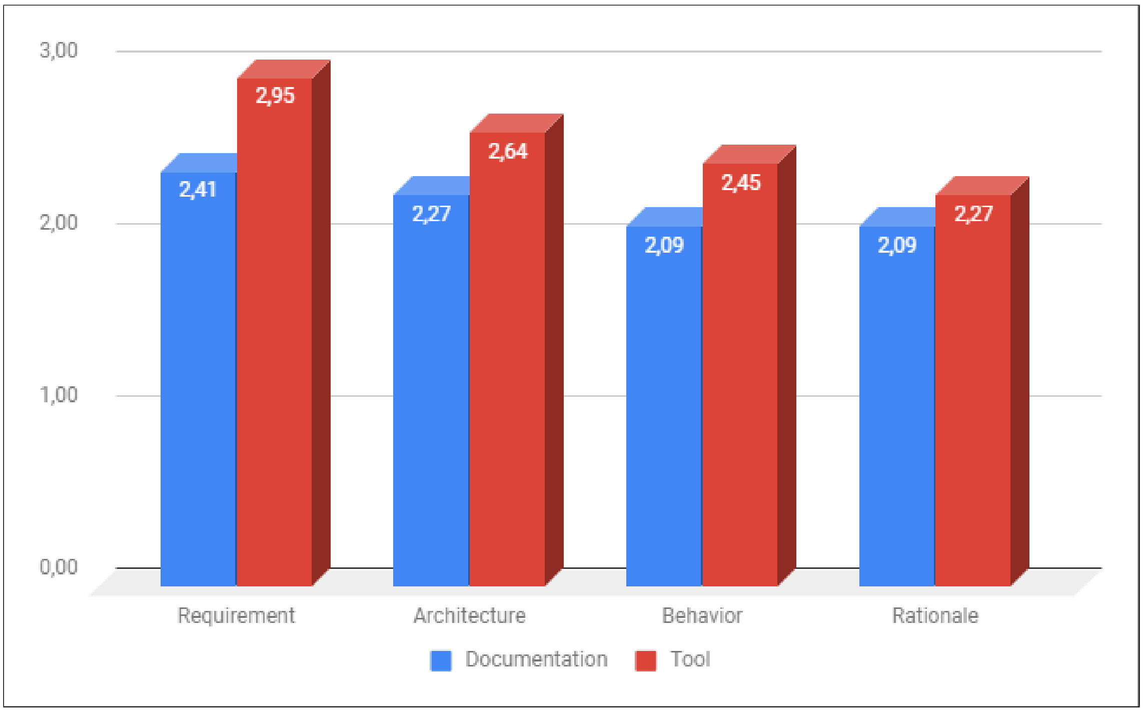


Figure 5.7: Score average by task type

Comparing the score level distribution is useful since it shows how independently the participants could perform overall. This is visualized in figure 5.6. The scoring distribution for the tool shows that 39 of the total tasks were given a score of 3. The ratio is around 70 percent since the total amount of tasks are 55. The scoring

Topic	**Summary**
Traceability	Being able to trace from the problem domain to the system design and implementation was an attribute that many participants found useful.
Analysis	The ability to trace and navigate the visualization was useful for analyzing the system.
Look and feel	The look and feel of the system was good
Relevancy	The explanations were relevant to answering the specified question.
Visuals and text	The connection between the visualization and text was useful for learning and comprehending the system.
Meta model design	The design of the meta model seems to properly model the domain
Navigation	Navigating the visualization is useful for finding information about the system
Learnability	SEC makes the system easy to comprehend and learn.
Overview	SEC gives a good overview of the system.

Table 5.4: Summary of system attributes/functionalities the participants liked

distribution for the documentation shows that 28 of the total tasks were given a score of 3. This gives us just a bit over 50 percent. As for the lower scores, the documentation has higher numbers on every score level. To summarize, it shows that the participants, in general, could more independently and correctly solve their given tasks by using the tool.

The results of the task scoring shows that in general, when participants used SEC instead of the case documentation; (1) all of them scored higher using SEC, (2) the scores mostly were in the 1 to 1.5 ratio range and (3) they more often had a perfect score for the tasks. With these results in mind, we can conclude that participants seem to score equal to or better using SEC.

5.0.2 Results from qualitative interviews

We performed semi-structured interviews after the usability tests were done to find out how developers perceived SEC tool (for more information about the evaluation design, see section 3.7).

In these interviews, participants were asked to provide feedback in three basic forms. First, they were asked to explain what they liked about using SEC in comparison to using the case documentation. Second, they were asked to imagine using SEC to assist their current work and think of changes or additions of functions that would be useful for them. Lastly, they had the opportunity to speak freely about their opinions of SEC. The collected answers are summarized to the topics shown in table 5.4, 5.5 and 5.6 (See appendix E to see the data). Figures 5.8 and 5.9 shows how many participants mentioned each topic during the interview.

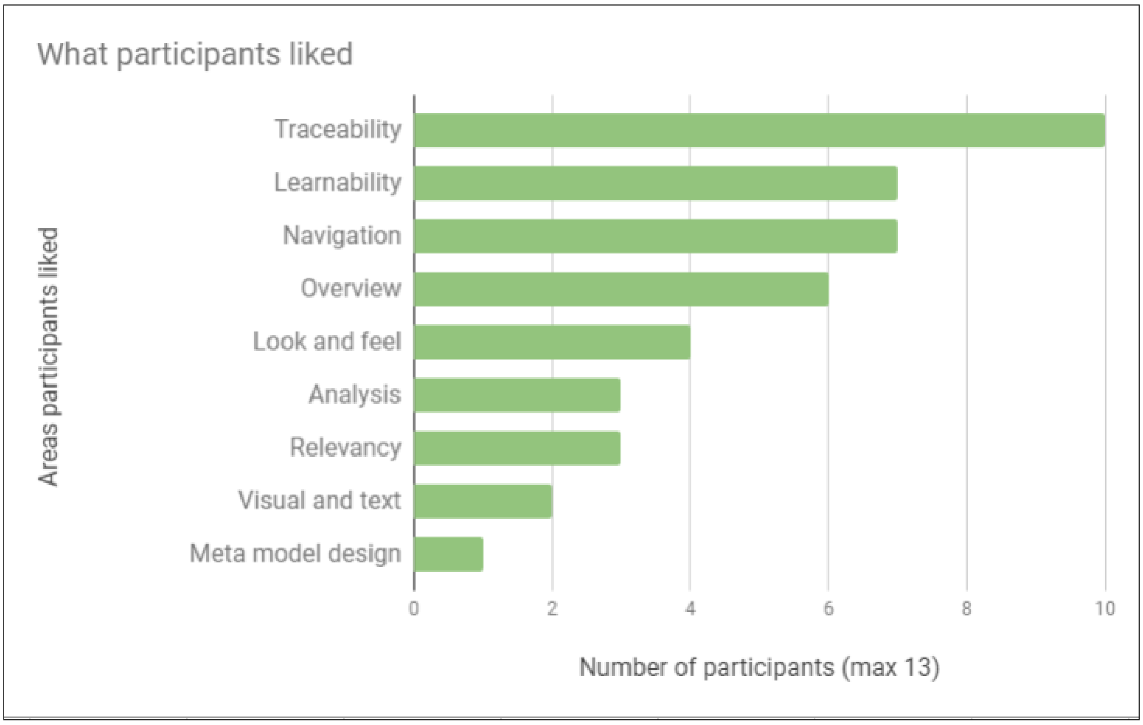


Figure 5.8: A summary of which areas of SEC the participants liked

The results from what participants liked about SEC (shown in figure 5.8) shows that most participants liked: (1) the aspects of traceability between software artifacts and AK, (2) how SEC assisted them in learning the case system and (3) how SEC could be used to navigate between software artifacts and AK. Some participants liked: (1) the overview SEC provides, (2) the look and feel of the UI, (3) that SEC helped them analyze the case system and (4) that the textual descriptions of the software artifacts were close to the visualization.

The results from suggested improvements (see figure 5.9) shows that the most requested features were: (1) a querying system that allows participants to perform customized queries to fetch information, (2) the developer questions to be summarized and (3) the ontology to be automatically populated. Some participants suggested: (1) that SEC should appeal to more use cases and stakeholders, (2) to make improvements to the UI to make it more intuitive, (3) to allow access directly to the source code in SEC and (3) to abstract the visualizations to improve scalability.

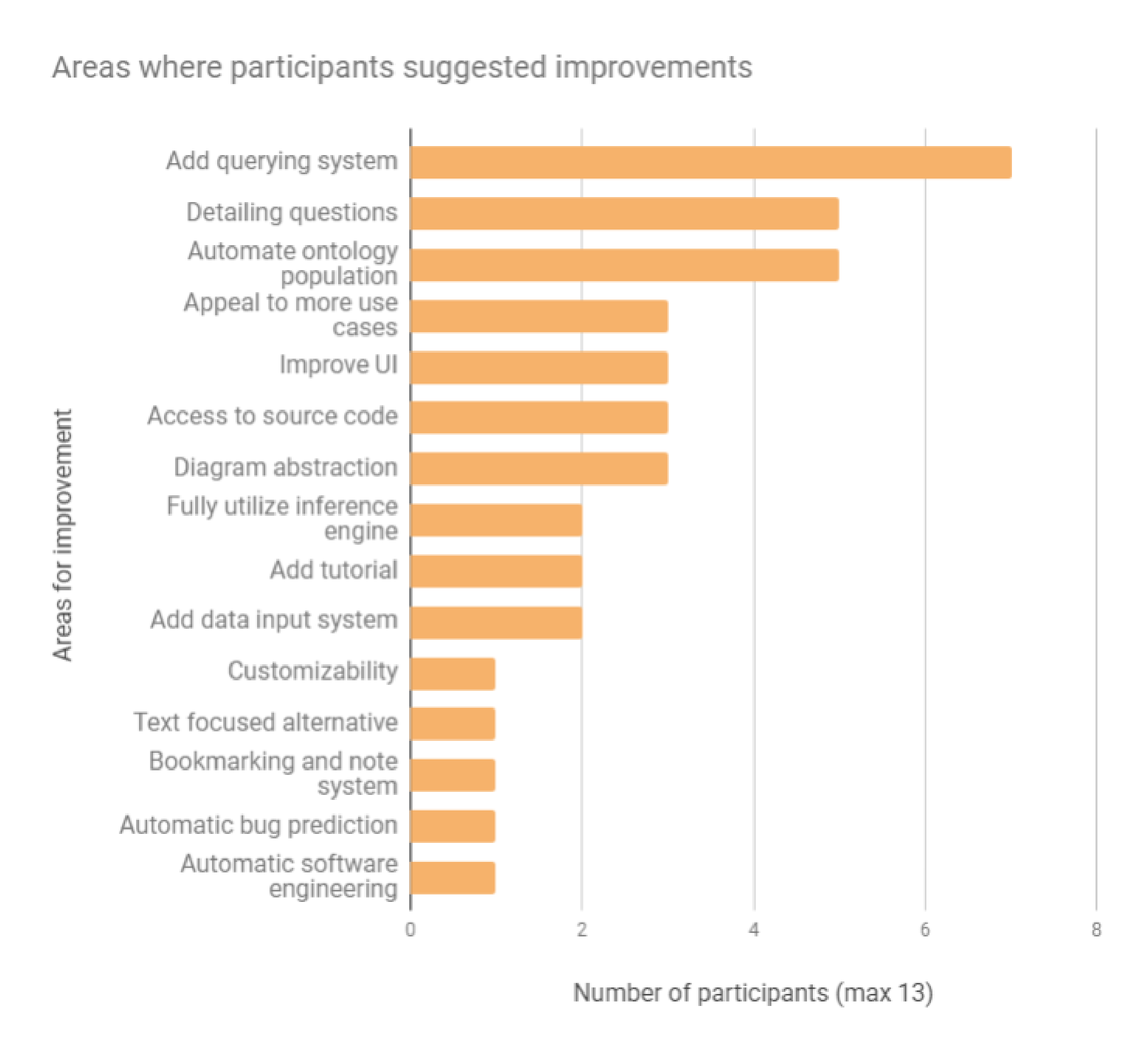


Figure 5.9: A summary of improvements participants suggested

Topic	**Summary**
Add tutorial	The participants may need some time to familiarize with the tool. Adding some tutorials would help with this process.
Add querying system	The current iteration of the tool only has a question/answer format. It would be useful to be able to make customized queries to fetch information from the system.
Diagram abstraction	In bigger systems, the diagrams will be very cluttered. Finding a way to group entities into bigger chunks, or abstract away unnecessary information would be quite useful.
Detailing questions	It is a bit challenging to predict what the questions will answer. Some kind of a summary showing how the answers to each question will be structured could solve this problem.
Fully utilize inference engine	Inference engines for ontologies can be used to infer "new" knowledge of the system. It could be a useful method to provide interesting information about the system.
Automate ontology population	Populating the ontology is one of the main challenges with tools such as this. Finding an approach to automatically populate the ontology would make implementing solutions like this much more attractive.
Appeal to more use cases	The use cases for SEC can be extended to support more task types and stakeholders.
Improve UI	The UI can be improved to make use of the tool more efficient and intuitive.
Access to source code	Being able to browse the source code directly from SEC makes it easier to learn more about the system.
Customizability	The tool should be customizable to fit for different stakeholders.
Text-focused alternative	Sometimes showing text may be more useful and intuitive than showing a visualization. This especially applies to the rationale.
Bookmarking and note system	SEC can be useful as a tool for communication between teammates. Functions to mark certain elements, write notes and share them would be useful.

Table 5.5: (Part 1) Summary of suggested improvements

Topic	**Summary**
Add data input system	SEC currently lacks the ability to add new system knowledge in the UI. There should be a UI that allows easy input of such knowledge.
Automatic bug prediction	SEC could be used to predict bugs
Automatic software engineering	The tasks that the participants solved during the evaluation could be automated.

Table 5.6: (Part 2) Summary of suggested improvements

6

Discussion

The results of the evaluations suggest that participants who use SEC to solve development-related tasks are generally more efficient than they are when using the case documentation. The goal of this chapter is to provide a brief discussion about the design of SEC and the results from the evaluation. First, we discuss the key SEC design features and some of the challenges and implications related to that. Second, we discuss the results of the evaluation and how they relate to the current challenges with software documentation that this study focuses on. Third, we discuss the similarities and differences between SEC and other similar tools. Lastly, we discuss the threats to validity in the results.

6.1 Ontology design, its challenges, and implications

The first research question was to study how software artifacts and AK of a system can be presented and explained. A part of our approach to tackling this issue was to store relationships between software artifacts and AK. We did this by building an ontology which served as a meta-model for storing and accessing software artifacts and AK in SEC.

6.1.1 The four sections of the ontology

In contrast to most similar studies and tools that also use ontologies [11], [13], [17], the ontology designed in this study consists of software artifacts among four different major sections; requirements, architecture, rationale, and implementation. This means the ontology in our study covers a more diverse range of software artifacts than most other ontologies made in the research area. The existence of these sections also made it easier to present software artifacts and their relationships from different areas of concern which helps developers answer high-level questions using the information from across these sections.
The main challenge of modeling the ontology this way (see figure 4.5 for a complete view of the ontology) was to find the correct abstraction level to build it. It is easy to make the ontology too detailed or too simple. The tricky part is to find a good balance between the two. The border that was set in this study lies between being able to trace from each software artifact to other related artifacts and staying on a high level without going into specific details of how they communicate and interact.

For example, many of the ontology classes relate to one another via the "comprisesOf" and "partOf" relations. The idea of only showing these relations is that they provide a meaningful way to view the structure of the system without revealing all the exact details of how different software components interact. To express the exact interaction between components, the ontology relies on external diagram images and descriptions. Designing the ontology this way provides a range of benefits such as minimizing cluttering of the visualization and making insertion of instance data less complex.

The result from the evaluation showed that the participants, in general, solved their given tasks faster using SEC. Even though the positive result from the evaluation is also affected by other factors than the ontology structure alone, it seems to indicate that this way of storing software artifacts was successful in assisting developers to navigate, trace, reason about the system and answer high-level questions. However, to find out whether this is specific ontology is optimal for the purposes stated above would require further research in the matter.

6.1.2 The 4+1 architectural view

The design of the architecture section of the ontology was based on the 4+1 architectural view model [4], with some modifications. This way of modeling the architecture component was useful for categorizing different software artifacts. The categorization made different abstraction levels of the architecture section of the ontology distinguishable. This resulted in separating concerns, reducing room for misinterpretation and increasing the expressiveness of the ontology. For example; without each view of the architecture, it becomes hard to express and interpret whether two architectural artifacts belong to the same abstraction level or not. By utilizing the architectural views, a general rule can be applied. If the two architectural artifacts belong to the same view, or to the views that are closely related, they are likely on the same abstraction level, otherwise, they are not. The 4+1 view not only affected the dynamics of the ontology but also affected how SEC presents software artifacts. The views provided good ways to cluster entities and explain the same developer-question from different abstraction levels.

As with most design, the expressiveness of the ontology also comes with a cost. Integrating the 4+1 view model into the ontology means each entity instance added to the ontology needs to be classified more elaborately. If a system that uses machine learning to learn how to populate ontologies was to be used for this case, just identifying architectural artifacts would not be enough, it would also need to classify which views these artifacts belong to. This adds another level of challenge for moving towards automatically populating ontologies.

6.1.3 System behavior

Another notable challenge of modeling the ontology was finding out how to represent system behavior. The behavior of a system is usually modeled differently from the system structure since they describe the system from different perspectives. System structure concerns aspects such as class hierarchy and code structure while system behavior concerns execution paths and interaction between classes. Modeling such interactions or execution paths in the ontology would have made it too complicated and detailed. The way we approached this problem was modeling the behavior in the ontology by storing a replica of all entities affected by the behavior and linking these entities to existing diagrams and their structural counterparts. This means that the ontology itself cannot explain the behavior. Instead, it relies on existing diagram images and diagram descriptions from the case documentation to do so. The responsibility of the ontology is to merely map the structural entities to diagrams that express their behavior. The idea is that this way, it is possible to trace from structural entities to their behavior without modeling the exact details of the behavior into the ontology.

From the results of the evaluation, it seems that modeling the behavior using this method did not help participants solve tasks related to system behavior any faster. This suggests that the behavioral section of the ontology may need improvements. However, as with their structural counterparts, these results are not only affected by the ontology design, but also by how the behavior was presented in SEC. We suspect that it is actually the presentation of the behavior which holds the main responsibility for this result (discussed in the next section).

6.2 Presentation of SEC, challenges and implications

The first research question asks how can SEC can present and express software artifacts in such a way that it increases learnability of the structure and behavior of the software system. The presentation of the artifacts was done by showing a graph and a textual description. The graph presented the structure of the artifacts and their relationships.

As you may recall, the results for behavior and rationale showed that participants did not necessarily perform better in these areas. We previously explained that we believe one of the significant causes behind this effect is the way system behavior and rationale was presented.

Designing the presentation of system behavior was particularly challenging since the ontology does not model behavioral interaction between software artifacts. One solution to this challenge could be to directly integrate the actual images of the diagrams (such as sequence diagrams, state machines and other UML diagrams that represent system behavior) into the structural graph of the artifacts. Using this method, the entities in the structural graph would be visually connected to their counterparts in the diagram image. This way both the structure of entities and their relationships as well as the behavioral interactions among the entities will be displayed in one view. However, at this time, this process is technically challenging and requires a tool that can visually translate images containing UML models to a format computers can understand (such as coordinates, names, size, etc.).

Rationale type tasks had the poorest performance compared to the other tasks. As with system behavior, designing the presentation for rationale is also a problem that could be further investigated. In this case, for the sake of consistency, we chose to show the rationale visually, like all other explanations. During the evaluation, we could observe that participants found it difficult to solve rationale related tasks. This was later also confirmed by the quantitative results. Some participants suggested that for the rationale type tasks specifically, textual descriptions could be more useful than the structural graph. The reason being that it is more intuitive to read rationale textually than to view it as a structural graph.

6.3 Discussion of results from the usability test

The results of the usability tests show that overall, all participants were faster and more precise when solving their given tasks using SEC. When looking into the different task types, we saw that SEC was particularly effective in requirements and architecture related questions.

In the grand scheme of things, the results from the usability test seem to align with the results gathered by other studies [11], [15]. Although their tools and evaluations differ from ours, they also found that in general, developers were more effective in solving development related tasks using tools than using traditional documentation. The results of this study mainly differ from theirs in that participants did not seem to complete tasks related to rationale and behavioral aspects more efficiently using SEC.

We believe that the reason rationale and behavior sections of SEC performed worse was due to a combination of three main factors: (a) First, the approach used to visualize the behavior and rationale was not what the participants expected to see. For example; as we previously mentioned, one participant commented that for the rationale, they preferred a textual explanation rather than a graphical one. The reason being that argumentation is easier to understand when expressed in words than when visualized. Regarding the behavior, we suspect that participants had difficulties interpreting how the visualizations expressed behavior since the behavioral diagrams were not directly shown in the interactive visualization. (b) The second factor is that most of the participants saw SEC for the first time. Some participants actually expressed that they probably would have performed better if they had more time to look into and use SEC. (c) The third factor is that the user interface was not fully developed, and some features and interactions that would have been useful for improving participants' efficiency were not present. For example, at the time of the usability test, the visualization was not "synced" with the textural description in SEC. Many participants expected that the side-bar at the right-hand side would automatically scroll down to the correct entity description when they selected an entity in the visual representation. Since it did not, they had to both search for the entity they were looking for in the left-hand side visualization, while also in the right-hand side, scroll to find the text that belongs to that entity. Some participants found this cumbersome and sometimes would, therefore, choose not to look for information from the textual descriptions at all.

Based on what we have discussed regarding the rationale and behavior related parts of SEC, we believe that there are obvious improvements that can be made to SEC to improve performance in these areas. The explanations related to rationale should be more text-focused, the explanations related to behavior could include behavioral diagrams directly in the visualization and the user interface can be improved to make SEC more intuitive for the user.

6.4 Discussion of results from the interviews

During the open-ended semi-structured interview, participants expressed what they liked about SEC. What is worth discussing is that per the design of the evaluation, the moderator did not lead the participant to express their opinions about specific parts of SEC. Rather participants were asked what they liked about using SEC in comparison to their experience of using the case documentation. As such, not all topics were discussed by all participants which could have led to a lower total count per topic. However, what is valuable with this format is that it enabled us, without introducing bias by specifically asking the participants about the topics, to check whether SEC actually addressed the identified documentation challenges listed at the beginning of the paper (see table 1.1). The qualitative results show that traceability, learnability, navigation, overview and look and feel were the top five areas of SEC which participants liked. These correspond quite well with some of the identified challenges, as can be seen in table 6.1 which shows a mapping between the challenges and the areas of SEC the participants liked.

Challenge	**What participants liked**
(C1) Size and complexity	Learnability, Navigation
(C2) Scattered information	Navigation, Traceability
(C3) Limited approaches for information retrieval	Navigation, Traceability
(C4) Multiple audiences	-
(C5) Excessive or redundant information	-

Table 6.1: Mapping between top five "I like" feedback to challenges

The first three challenges correspond quite well to the areas of SEC participants liked. What about suggested improvements? Areas that participants suggested improvements for are areas where they most likely believed were weak. Table D.1 shows the mapping between the challenges and the top five suggested areas of improvements.

Challenge	**Suggested improvements**
(C1) Size and complexity	-
(C2) Scattered information	-
(C3) Limited approaches for information retrieval	Add querying system
(C4) Multiple audiences	Appeal to more use cases
(C5) Excessive or redundant information	-

Table 6.2: Mapping between top five "What if" feedback to challenges

According to comments from the participants, SEC needs some improvements to fully resolve the challenges C3 and C4. What should be taken into account is that at the time of the evaluation, SEC was not a fully developed system. As such it lacked many core features which would have been important to have in an industrial setting. Regarding C3, the only method for participants to find information about the case system was via the developer questions. As such it is not surprising that a more customizable and flexible information retrieval system was desirable. The fact that participants both expressed that they liked the existing format and suggested improvements upon it, shows that the implemented solution seems to partly address the challenge but more traditional methods (text search, some query language, etc.) should be implemented as well. Regarding C4, 5 participants suggested that the SEC could adhere to more use cases. What is worth to mention is that most of the comments suggested new ideas for improving the range of use cases. The same argument could be made here since SEC was not complete at the time of the evaluation, it did obviously not support most use cases. However, in the way the ontology is modeled and how the system is structured, we have attempted to make the system easy to extend for implementing more use cases and adhere to more stakeholders in the future.

6.5 Similarities and differences to related work

SEC is composed of several components that are both similar and different from previous work in the field of research. This section is dedicated to highlighting and discussing these similarities and differences.

SEC is similar to other existing tools in two ways. First is that it uses the same technique to achieve a certain outcome (such as using an ontology to interrelate artifacts). Second is that it tries to achieve the same underlying goals (such as generating explanations), but uses another method or does so in a different abstraction level.

The tools that are most similar to SEC are tools that interrelate software artifacts, support tracing, navigation and provide user interfaces for retrieving knowledge, they are: *The Knowledge* Architect, *ArchiMind*, *LaSSiE*, and *PAKME*. SEC is similar to these tools in the following aspects:

1. Interrelates software artifacts
2. Uses ontology for storing domain model and artifact instances
3. Supports querying knowledge from the ontology
4. Supports tracing and navigation between artifacts
5. Has very similar objectives

In addition to being similar to the above tools, SEC is also similar to *Whyline*. They are similar in the way that both tools have the underlying idea of generating explanations to software systems based on given questions. All the mentioned similarities are core elements of SEC, but they are also expected to be similar since the purpose of SEC was to use these existing technologies and ideas to explore further possibilities, extending the boundary of knowledge in the area. This leads to the topic of how SEC differs from the aforementioned tools, which are highlighted below.

1. SEC does not inherently support maintenance of knowledge as there are existing tools that demonstrate this ability.
2. In contrast to most of the aforementioned tools, the ontology of SEC covers a larger range of software artifacts, from architecture and requirements to implementation.
3. In comparison to other tools, the ontology of SEC has very simplistic structural relations between artifacts. Its function is to coarsely relate artifacts for navigation and traceability rather than expressing exact dependency and communication patterns. These kinds of expressions are left to be presented through images and descriptions.
4. SEC puts focus on aspects of collection, presentation, explanation, and navigation of software artifacts.
5. In contrast to other tools, SEC does not implement a keyword search system to retrieve AK.
6. SEC instead presents knowledge by composing explanations to a selection of predefined high-level questions. It is the first tool (to the best of our knowledge) in this research area that generates explanations using this technique.
7. The explanations differ from the ones generated by Whyline in abstraction level. SEC answers high-level questions related to architecture and specification while Whyline answers implementation level questions.

6.6 Threats to validity

This section describes threats to the validity of this research and how they have been addressed.

6.6.1 Construct validity

The most prominent threat to construct validity is that SEC is better designed to answer the kind of tasks given in the evaluation than the case documentation is. It is thus expected that the results would be somewhat biased towards participants being more efficient when using SEC. In an attempt to mitigate this problem, annotations to the documentation index were made to point out sections which may be especially relevant to architecture, requirements, user stories, etc. While the pilot tests show that the annotations provided a noticeable improvement in participants performance when solving the tasks using the documentation, it not clear whether it provided too much or too little support. However, in our opinion, having an annotated index for the case documentation definitely makes the playground fairer than without.
One common threat to the construct validity is called mono-method bias. This happens when only one method of measurement is used. To mitigate such threat, we used both a usability test that mainly provided us with quantitative data as well as an interview with open ended-questions that provided the qualitative data. This makes it possible to check whether the results of the two methods align or not.

The last construct validity is about learning and ordering of the usability test. Recall that the usability test consisted of two parts. In the first part, the participant solved development related tasks using SEC. In the second part, they solved similar tasks using the documentation. These two parts were very similar, and since both parts used the same case system, participants could have gained some knowledge after completing part one and therefore could apply this knowledge to the second part. This could have helped them to solve the second part more efficiently. To mitigate this risk, the order of these parts were swapped for every other participant. As a result, half of the participants used the documentation first and the other half used the SEC tool first to answer the questions.

6.6.2 Internal validity

The result of the quantitative section of the evaluation strongly suggested that participants solved development-related tasks more efficiently when using SEC than they did using the case documentation. However, as with all results, it is worth discussing whether it is valid or not.

The main limitation that caused internal validity threats during the quantitative section of the evaluation was the time constraint. Time was constrained for mainly two reasons; (1) participants were not paid to participate, and as such would not agree to participate if the sessions were too long and (2) too long sessions tend to tire out the participants which may affect the results.

The strategy to mitigate the issue of time limitation consisted of the following four steps. First, a time limit was set on each task to adhere to the time limit. Second, since the main focus was not to evaluate the intuitiveness and learnability of the UI or the case documentation, but rather whether the tool helped participants solve their tasks more efficiently or not, participants were allowed to ask the moderator of the evaluation for pointers and directions when they felt lost (both when using the tool and documentation). Third, in order to help participants move forward with the tasks, whenever the participants failed to complete a task within the time limit, help was provided to find the solution (this was reflected in their task score).

While these validity threats may have caused the results to be biased, we did attempt to minimize the damage as much as possible. First, pointers were provided both when participants were using SEC and when they were using the documentation. Second, when providing pointers, the questions were never directly answered for them. Instead, the evaluation moderator merely explained where certain features were located or pointed to chapters where they may find their answers. Third, since the documentation was much larger and harder to learn and browse than SEC, we provided a few simple annotations on the index page to point them into the right direction (e.g which chapters were related to requirements, architecture, etc). And lastly, to mitigate the fact that almost none of the participants (except 1) had seen SEC before, they were allowed to explore the tool for 10 minutes where the moderator explained all its features, how the visualizations and textual descriptions were connected and how they could be interpreted.

With regard to these internal validity threats, we believe the bias would lean more towards causing the participants to solve tasks faster using the documentation than the other way around. Despite this, the results still turned out positive that participants were more efficient using SEC. In our opinion, this strengthens the result that participants solved their given tasks more efficiently when using SEC.

Another common internal validity threat is the bias in the selection of participants. The results could, for example, be affected depending on whether: (a) participants were very familiar with using the case documentation or SEC, (b) participants were not familiar with using any kind of documentation all, or (c) if they were all from industrial or academic backgrounds. Of course, some of these biases are very difficult to mitigate. In this case, the actions taken to mitigate this kind of bias were to make sure to select the participants that are representative of the population, software developers and to make sure to select participants with certain background diversity. The selected participants all knew how to code, and had at least seen or used SAD to perform software development related tasks before. Half of the participants had an industrial background and the other half had an academic background.

The last validity threat we will discuss is the sample size. The sample size of the quantitative data we measured is 10. This is very small, and the results can obviously not be generalized to the general population of software developers. As such it could be argued that due to this large risk for error, the quantitative data are not of any use. However, since our main purpose of the evaluation was not to use the data to make general statements about SEC, but rather aimed at collecting further insight regarding whether the participants solved their given tasks more efficiently or not. The quantitative data here is mainly used as a validity check to see whether the qualitative results from the interviews hold or not. For example, a participant can express that they think that SEC helped them perform much better, but in reality, perhaps it did not. Using the quantitative data in this scenario allows us to check the validity of that statement. If the two data contradict, then there is obviously a conflict to reflect on, if not, the two results can strengthen each other. Now, we did use the data to perform a t-test and calculate the effect size. And while these results aren't useful for predicting how future evaluations will perform, they do provide insight into how well these specific participants performed and provide pointers to the areas of SEC that may need improvements. We also believe that the fact that most participants performed significantly better using SEC than the case documentation, warrants a further investigation in the topic, and indicates potential.

6.6.3 External validity

The external validity concerns to what extent the results of the findings of this study can be generalized. The external validity threats in this study are due to constraints in both time and resources, only a single case system was used as a reference to design the ontology, to populate it and to evaluate the tool. This obviously severely limits the ability to generalize the results of this study. By the design of the study, generalizability was never a primary concern, rather the primary purpose of the study was to explore how a tool can explain and present software artifacts of a case system in such a way that increases understandability and traceability. However, as previously discussed in this paper, creating a generalized tool that can explain any kind of software system is a difficult task. The results from this study could be used as a lesson or inspiration for future work, whether that be another case or towards engineering a more generalized tool.

7
Conclusion

The current format of software documentation is commonly wiki-based or file-based, both of which have their shortcomings; they provide limited traceability and navigation between software artifacts especially as documents grow in size and complexity. Also, each software system often has multiple documentations to meet the needs of different stakeholders and therefore information tends to become scattered over different documents and platforms.

The purpose of this study was to explore ways to address these shortcomings. It is a design science research studying one case, where the design issue was to examine (a) how to represent relationships that can properly express the interrelations between different software artifacts and AK of the case and (b) how to compose and present information from such relationships to the user. The end goal with this tool was to provide assistance to developers in navigating, understanding and tracing artifacts of software systems. To achieve this goal, a case system was selected and a tool was developed. The tool is called System Explanation Composer (SEC) which was built using four parts: (1) an ontology to structure and store software knowledge, (2) a case system to populate the ontology with, (3) a selection of frequently asked questions by developers, which is also referred to as developer questions, (4) an explanation generator that creates explanations to the developer questions

The ontology design has three distinct attributes that are unique from other similar research in the area. The first attribute is that the ontology is designed to store and interrelate software artifacts among four different categories.

1. Requirements, which holds information about the problem domain.
2. Architecture, which expresses how the system is designed.
3. Rationale, which holds the reasoning behind the design of the architecture.
4. Implementation, which holds information about the source code.

The second unique feature is the architecture section of the ontology. It is based on the 4+1 architectural view model [4] which targets the information needs of different stakeholders by providing guidelines to express several abstraction levels of system design. These two attributes made the ontology more expressive and gave SEC the ability to provide tracing from features to implementation. It also has the ability to answer high-level questions by combining knowledge from different areas of a software system.

The final unique feature to SEC is that it generates explanations to high-level questions about the system. Unlike most other research in the field that focuses on

insertion and retrieval of information[2], [7], [11], [13], [17], SEC instead demonstrates how retrieved information from ontologies can be composed and presented to form explanations to high-level questions. The other studies provided the groundwork, while SEC further explores and demonstrates the potential value of structuring software documentation in ontologies.
With the ontology as a base, explanations to the chosen developer questions could be generated. The explanations presented a structural graph and textual descriptions to the entities of the structural graphs side by side. This made it possible for SEC to show how certain software artifacts were structured while providing descriptions of them at once.

To find out whether SEC would prove to be useful for developers, a two-part evaluation was performed. The first part was a usability test, which had a sample size of 10. The second part was a semi-structured interview, which was performed after the usability test, with a sample size of 13. The results from the usability tests show that in general, participants were significantly more efficient in solving given tasks using SEC in comparison to using the file-based case documentation. This was especially true with tasks related to requirements and architecture. In the case of behavioral and rationale related tasks, there seemed to be no noticeable improvements nor recessions in performance when using SEC. However, there are good indications that the low performing areas of SEC can be greatly improved in the future.

After the usability tests, participants were also interviewed using a semi-structured format with open-ended questions. Unprompted, almost all participants expressed that they liked the traceability aspect of SEC. More than half of the participants also expressed that they noticed improvements in learnability, navigation, and overview when using SEC. The areas most participants suggested improvements for were including a more customizable querying system, providing short summaries to explanations and finding techniques to automate ontology population. These results suggest that participants believe that SEC addresses some challenges of software documentation and has much room for growth.

Despite the fact that we had a small sample size and could not generalize these results to the whole population of software developers, we still believe our results show potential and are useful for promoting further research within this area.

Future work should aim at automatically populating ontologies as it would pave the way for a tool such as SEC to be used in the industry. However, finding a method to do this is obviously not a simple task. Other interesting areas to look into are; (1) finding ways to better model and/or present the behavior and rationale sections of the ontology and (2) providing a customizable querying system alongside the question/answer format, (3) finding ways to make manual population of the ontology user-friendly and effortless, (4) abstracting entities to reduce graph cluttering and (5) connecting SEC to the IDE for closer access to source code. If solutions to these problems were developed, despite the lack of automation of ontology population, we believe tools like SEC will prove to be a valuable asset to software development processes.

Bibliography

[1] H. A. Simon, “The sciences of the artificial”, *Cambridge, MA*, 1969.

[2] P. Devanbu, R. Brachman, and P. G. Selfridge, “Lassie: A knowledge-based software information system”, *Communications of the ACM*, vol. 34, no. 5, pp. 34–49, Jan. 1991. DOI: 10.1145/103167.103172.

[3] D. E. Perry and A. L. Wolf, “Foundations for the study of software architecture”, *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, Jan. 1992. DOI: 10.1145/141874.141884.

[4] P. Kruchten, “Architectural Blueprints — The “ 4 + 1 ” View Model of Software Architecture”, vol. 12, no. November, pp. 42–50, 1995.

[5] T. C. Lethbridge, J. Singer, A. Forward, and D. Consulting, “How Software Engineers use Documentation : The State of the Practice Documentation ”, 2003.

[6] A. J. Ko and B. A. Myers, “Designing the whyline”, *Proceedings of the 2004 conference on Human factors in computing systems - CHI 04*, 2004. DOI: 10.1145/985692.985712.

[7] M. A. Babar, X. Wang, and I. Gorton, “Pakme: A tool for capturing and using architecture design knowledge”, *2005 Pakistan Section Multitopic Conference*, 2005. DOI: 10.1109/inmic.2005.334419.

[8] A. J. Ko, B. A. Myers, S. Member, M. J. Coblenz, and H. H. Aung, “An Exploratory Study of How Developers Seek , Relate , and Collect Relevant Information during Software Maintenance Tasks”, vol. 32, no. 12, pp. 971–987, 2006.

[9] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research”, *Journal of management information systems*, vol. 24, no. 3, pp. 45–77, 2007.

[10] J. Sillito, G. C. Murphy, and K. D. Volder, “Asking and Answering Questions during a Programming Change Task”, vol. 34, no. 4, pp. 434–451, 2008.

[11] A. Jansen, P. Avgeriou, and J. S. V. D. Ven, “Enriching software architecture documentation”, *Journal of Systems and Software*, vol. 82, no. 8, pp. 1232–1248, 2009. DOI: 10.1016/j.jss.2009.04.052.

[12] *Systems and software engineering - Life cycle processes - Requirements engineering*, IEEE 29148, 2011.

[13] K. A. D. Graaf and A. Tang, “Ontology-based Software Architecture Documentation”, 2012. DOI: 10.1109/WICSA-ECSA.212.20.

[14] L. L. Escoriza, “Analysis , design and development of a web-shop template using SPHERE . IO e-commerce platform”, no. January, 2014.

[15] K. A. D. Graaf, P. Liang, A. Tang, and H. V. Vliet, “Science of Computer Programming How organisation of architecture documentation affects architectural knowledge retrieval”, *Science of Computer Programming*, vol. 121, pp. 75–99, 2016, ISSN: 0167-6423. DOI: 10.1016/j.scico.2015.10.014. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2015.10.014.

[16] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, *et al.*, “On-demand developer documentation”, in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, IEEE, 2017, pp. 479–483.

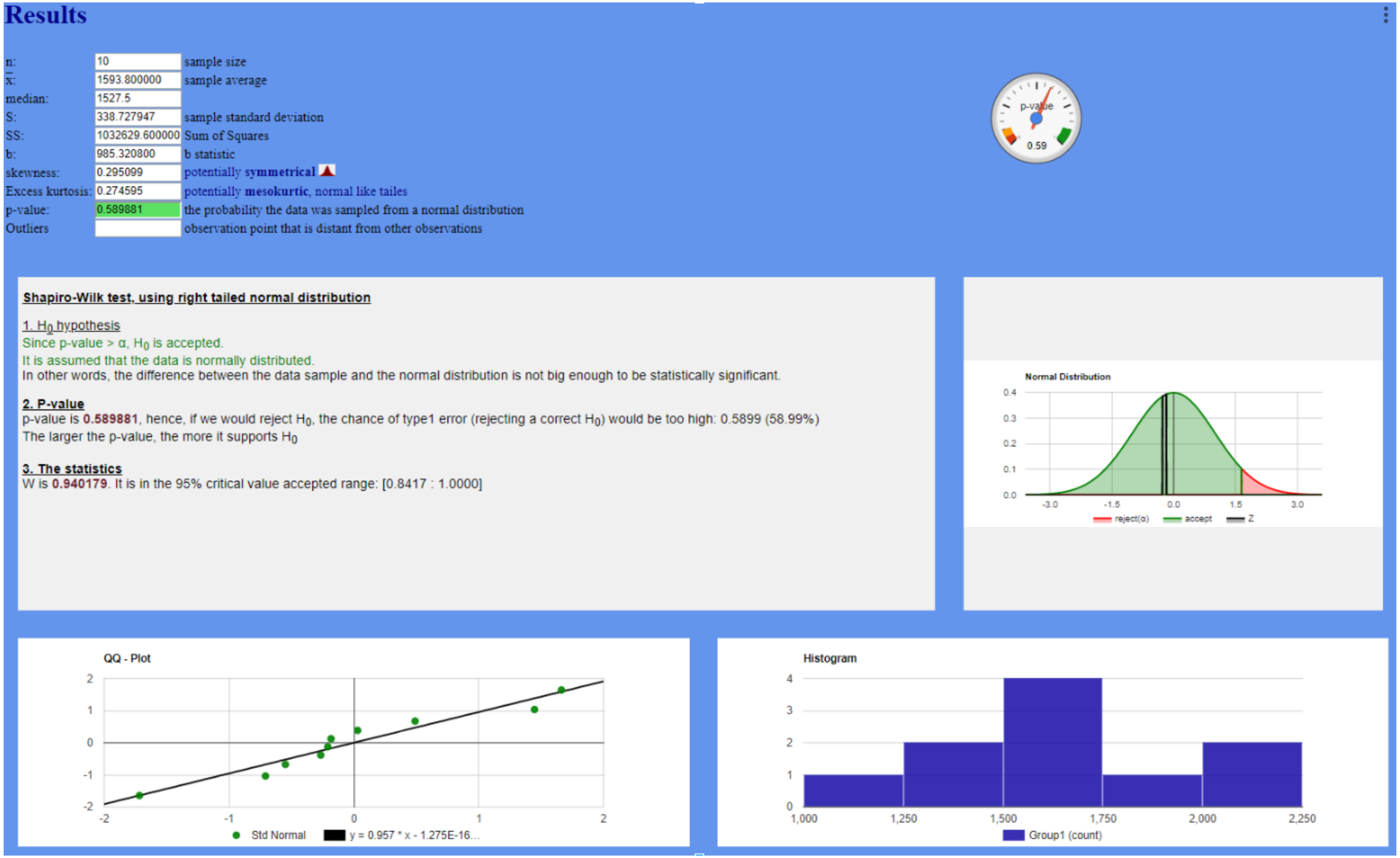
[17] M. Soliman, A. R. Salama, M. Galster, O. Zimmermann, and M. Riebisch, “Improving the search for architecture knowledge in online developer communities”, *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018. DOI: 10.1109/icsa.2018.00028.

[18] M.-A. Storey, “Theories, methods and tools in program comprehension: Past, present and future”, *13th International Workshop on Program Comprehension (IWPC05)*, DOI: 10.1109/wpc.2005.38.

A

Appendix A

This appendix shows the test for normality for tool time (figure A.1) and documentation time(figure A.2). It also shows the F-test that checks whether the variances between the two are equal (figure A.3).



Results

n:	10	sample size
$\bar{x}$:	1593.800000	sample average
median:	1527.5	
S:	338.727947	sample standard deviation
SS:	1032629.600000	Sum of Squares
b:	985.320800	b statistic
skewness:	0.295099	potentially **symmetrical**
Excess kurtosis:	0.274595	potentially **mesokurtic**, normal like tailes
p-value:	0.589881	the probability the data was sampled from a normal distribution
Outliers		observation point that is distant from other observations

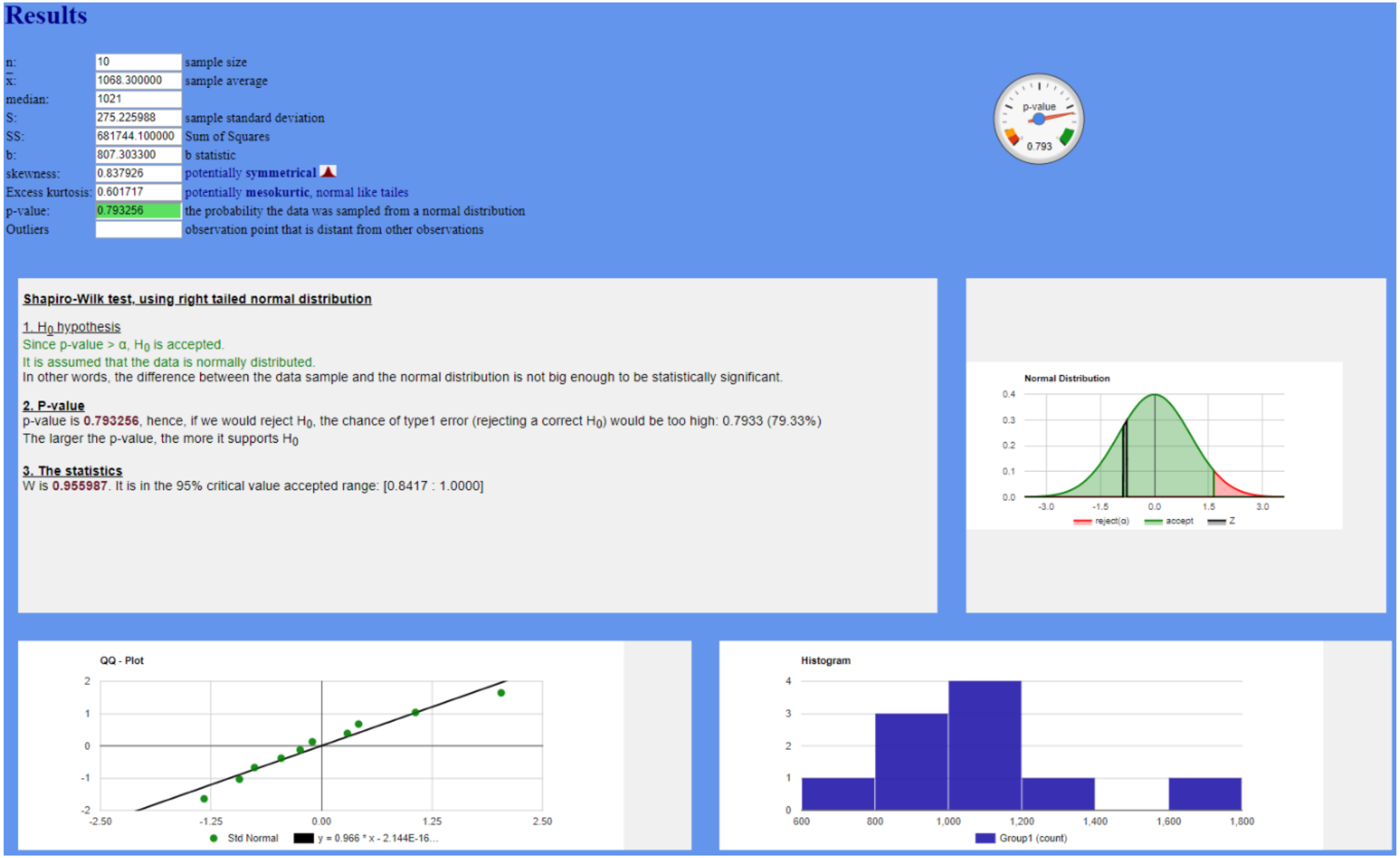
Shapiro-Wilk test, using right tailed normal distribution

1. H_0 hypothesis
Since p-value > α, H_0 is accepted.
It is assumed that the data is normally distributed.
In other words, the difference between the data sample and the normal distribution is not big enough to be statistically significant.

2. P-value
p-value is **0.589881**, hence, if we would reject H_0, the chance of type1 error (rejecting a correct H_0) would be too high: 0.5899 (58.99%)
The larger the p-value, the more it supports H_0

3. The statistics
W is **0.940179**. It is in the 95% critical value accepted range: [0.8417 : 1.0000]

Figure A.1: Check for normality, tool time



Results

n:	10	sample size
$\bar{x}$:	1068.300000	sample average
median:	1021	
S:	275.225988	sample standard deviation
SS:	681744.100000	Sum of Squares
b:	807.303300	b statistic
skewness:	0.837926	potentially **symmetrical**
Excess kurtosis:	0.601717	potentially **mesokurtic**, normal like tailes
p-value:	0.793256	the probability the data was sampled from a normal distribution
Outliers		observation point that is distant from other observations

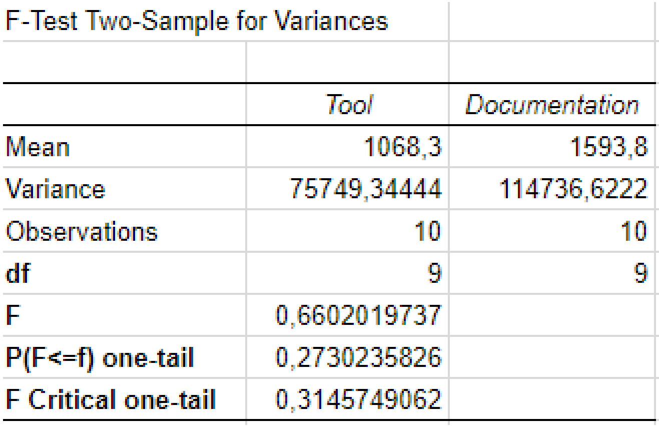
Shapiro-Wilk test, using right tailed normal distribution

1. H_0 hypothesis
Since p-value > α, H_0 is accepted.
It is assumed that the data is normally distributed.
In other words, the difference between the data sample and the normal distribution is not big enough to be statistically significant.

2. P-value
p-value is **0.793256**, hence, if we would reject H_0, the chance of type1 error (rejecting a correct H_0) would be too high: 0.7933 (79.33%)
The larger the p-value, the more it supports H_0

3. The statistics
W is **0.955987**. It is in the 95% critical value accepted range: [0.8417 : 1.0000]

Figure A.2: Check for normality, documentation time



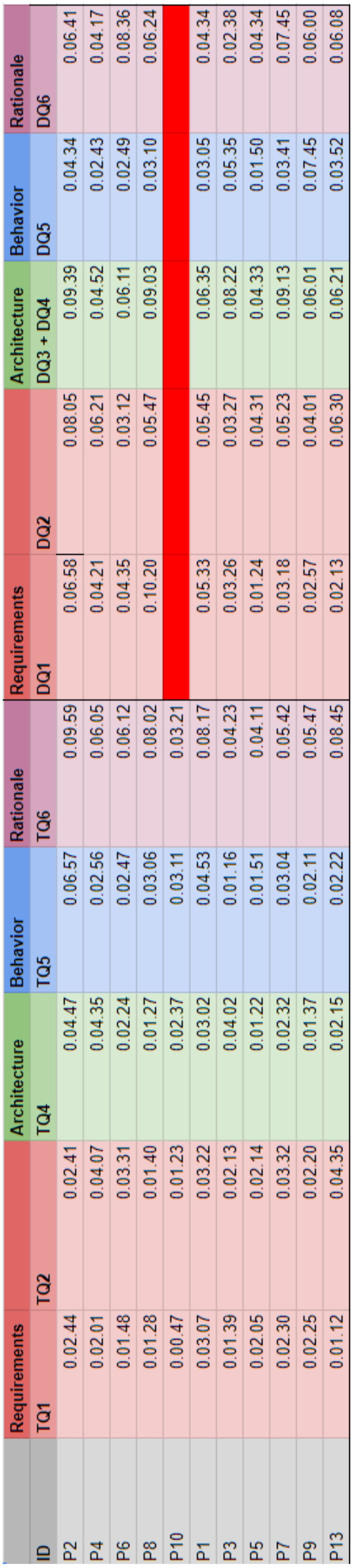
F-Test Two-Sample for Variances

	Tool	*Documentation*
Mean	1068,3	1593,8
Variance	75749,34444	114736,6222
Observations	10	10
df	9	9
F	0,6602019737	
P(F<=f) one-tail	0,2730235826	
F Critical one-tail	0,3145749062	

Figure A.3: F-test, check for equal variances between tool and documentation time

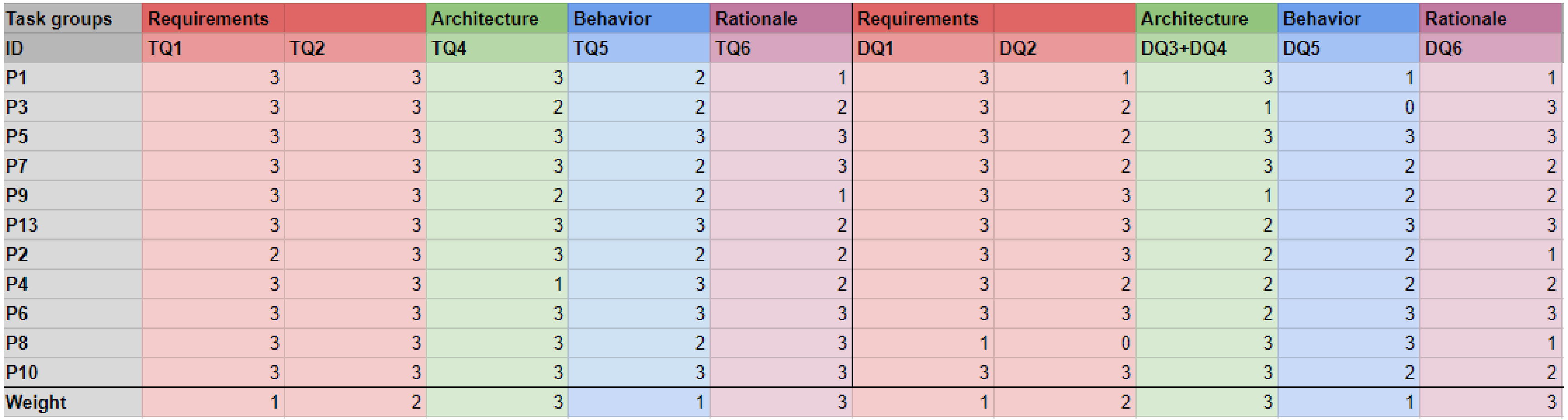
B
Appendix B

This appendix shows the raw time data (figure B.1) and score data (figure B.2). TQ stands for Tool Question and DQ stands for Documentaition Question.



	Requirements		Architecture	Behavior	Rationale	Requirements		Architecture	Behavior	Rationale
ID	TQ1	TQ2	TQ4	TQ5	TQ6	DQ1	DQ2	DQ3 + DQ4	DQ5	DQ6
P2	0.02.44	0.02.41	0.04.47	0.06.57	0.09.59	0.06.58	0.08.05	0.09.39	0.04.34	0.06.41
P4	0.02.01	0.04.07	0.04.35	0.02.56	0.06.05	0.04.21	0.06.21	0.04.52	0.02.43	0.04.17
P6	0.01.48	0.03.31	0.02.24	0.02.47	0.06.12	0.04.35	0.03.12	0.06.11	0.02.49	0.08.36
P8	0.01.28	0.01.40	0.01.27	0.03.06	0.08.02	0.10.20	0.05.47	0.09.03	0.03.10	0.06.24
P10	0.00.47	0.01.23	0.02.37	0.03.11	0.03.21					
P1	0.03.07	0.03.22	0.03.02	0.04.53	0.08.17	0.05.33	0.05.45	0.06.35	0.03.05	0.04.34
P3	0.01.39	0.02.13	0.04.02	0.01.16	0.04.23	0.03.26	0.03.27	0.08.22	0.05.35	0.02.38
P5	0.02.05	0.02.14	0.01.22	0.01.51	0.04.11	0.01.24	0.04.31	0.04.33	0.01.50	0.04.34
P7	0.02.30	0.03.32	0.02.32	0.03.04	0.05.42	0.03.18	0.05.23	0.09.13	0.03.41	0.07.45
P9	0.02.25	0.02.20	0.01.37	0.02.11	0.05.47	0.02.57	0.04.01	0.06.01	0.07.45	0.06.00
P13	0.01.12	0.04.35	0.02.15	0.02.22	0.08.45	0.02.13	0.06.30	0.06.21	0.03.52	0.06.08

Figure B.1: Left side, labeled TQ1 to TQ6 are tool times. Right side, labeled DQ1 to DQ6 are documentation times



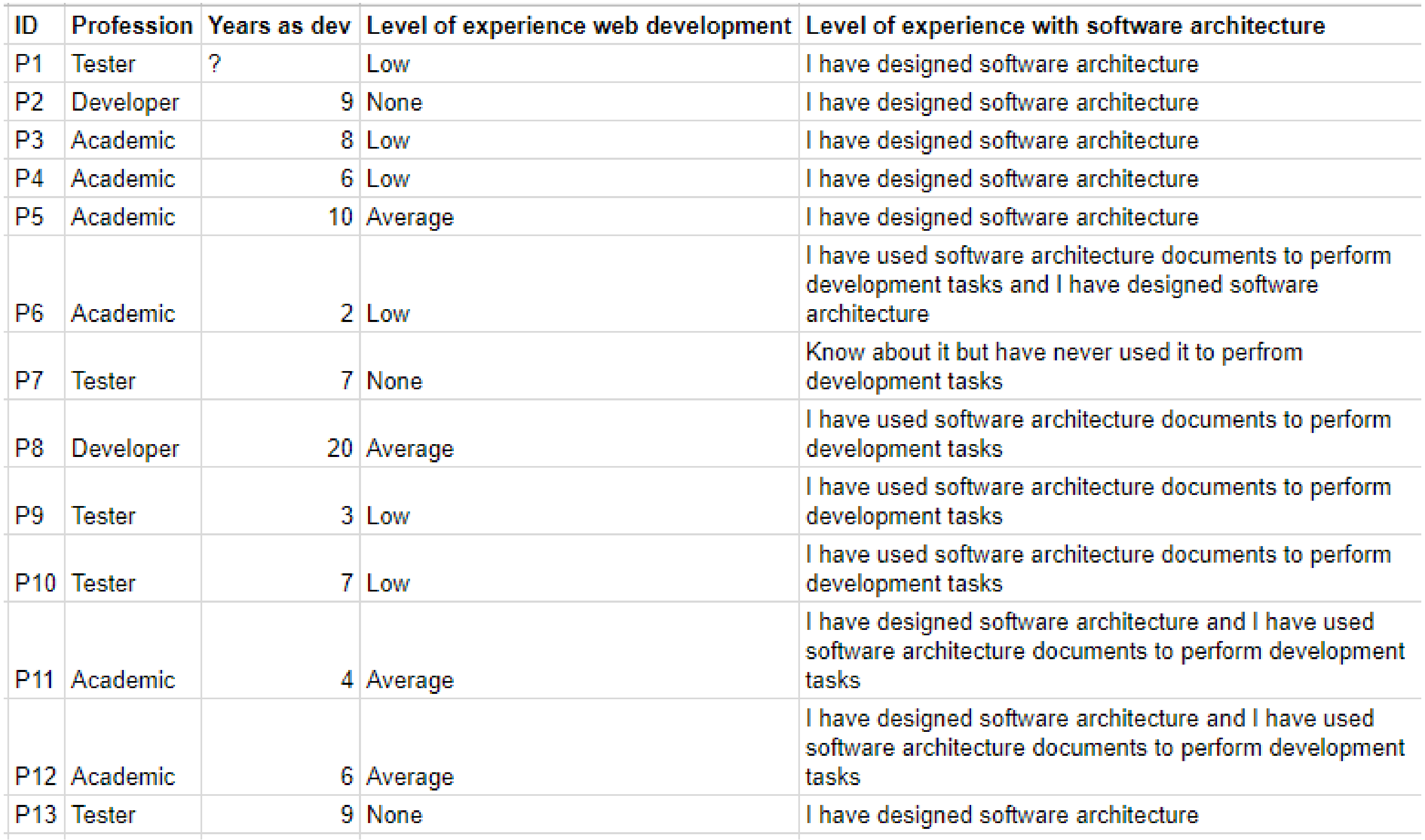
Task groups	Requirements		Architecture	Behavior	Rationale	Requirements		Architecture	Behavior	Rationale
ID	TQ1	TQ2	TQ4	TQ5	TQ6	DQ1	DQ2	DQ3+DQ4	DQ5	DQ6
P1	3	3	3	2	1	3	1	3	1	1
P3	3	3	2	2	2	3	2	1	0	3
P5	3	3	3	3	3	3	2	3	3	3
P7	3	3	3	2	3	3	2	3	2	2
P9	3	3	2	2	1	3	3	1	2	2
P13	3	3	3	3	2	3	3	2	3	3
P2	2	3	3	2	2	3	3	2	2	1
P4	3	3	1	3	2	3	2	2	2	2
P6	3	3	3	3	3	3	3	2	3	3
P8	3	3	3	2	3	1	0	3	3	1
P10	3	3	3	3	3	3	3	3	2	2
Weight	1	2	3	1	3	1	2	3	1	3

Figure B.2: Left side, labeled TQ1 to TQ6 are tool scores. Right side, labeled DQ1 to DQ6 are documentation scores

C

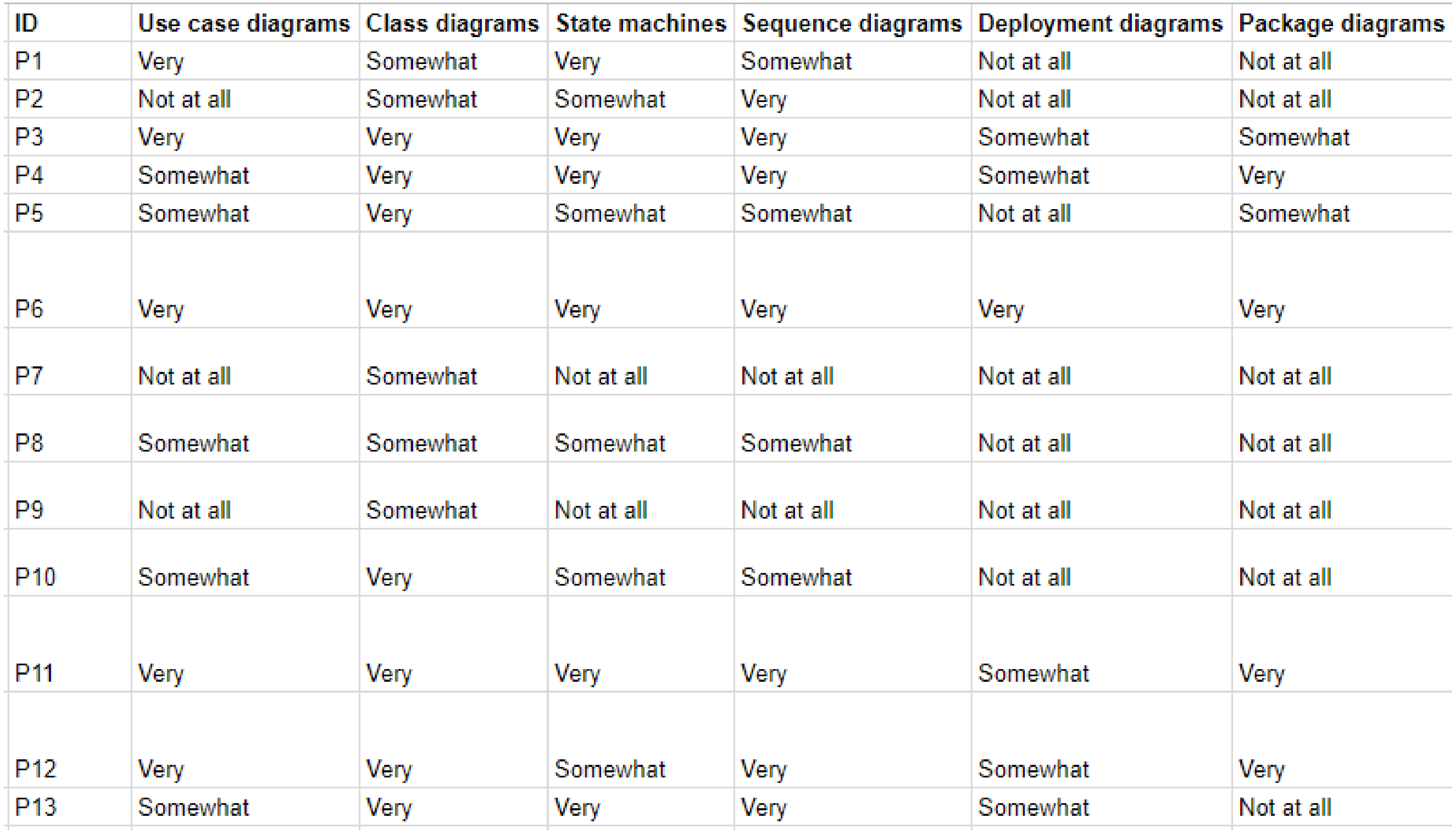
Appendix C

This appendix shows personal development experience (figure C.1), and UML model familiarity (figure C.2) for each participant.



ID	Profession	Years as dev	Level of experience web development	Level of experience with software architecture
P1	Tester	?	Low	I have designed software architecture
P2	Developer	9	None	I have designed software architecture
P3	Academic	8	Low	I have designed software architecture
P4	Academic	6	Low	I have designed software architecture
P5	Academic	10	Average	I have designed software architecture
P6	Academic	2	Low	I have used software architecture documents to perform development tasks and I have designed software architecture
P7	Tester	7	None	Know about it but have never used it to perfrom development tasks
P8	Developer	20	Average	I have used software architecture documents to perform development tasks
P9	Tester	3	Low	I have used software architecture documents to perform development tasks
P10	Tester	7	Low	I have used software architecture documents to perform development tasks
P11	Academic	4	Average	I have designed software architecture and I have used software architecture documents to perform development tasks
P12	Academic	6	Average	I have designed software architecture and I have used software architecture documents to perform development tasks
P13	Tester	9	None	I have designed software architecture

Figure C.1: Personal experience for each participant



ID	Use case diagrams	Class diagrams	State machines	Sequence diagrams	Deployment diagrams	Package diagrams
P1	Very	Somewhat	Very	Somewhat	Not at all	Not at all
P2	Not at all	Somewhat	Somewhat	Very	Not at all	Not at all
P3	Very	Very	Very	Very	Somewhat	Somewhat
P4	Somewhat	Very	Very	Very	Somewhat	Very
P5	Somewhat	Very	Somewhat	Somewhat	Not at all	Somewhat
P6	Very	Very	Very	Very	Very	Very
P7	Not at all	Somewhat	Not at all	Not at all	Not at all	Not at all
P8	Somewhat	Somewhat	Somewhat	Somewhat	Not at all	Not at all
P9	Not at all	Somewhat	Not at all	Not at all	Not at all	Not at all
P10	Somewhat	Very	Somewhat	Somewhat	Not at all	Not at all
P11	Very	Very	Very	Very	Somewhat	Very
P12	Very	Very	Somewhat	Very	Somewhat	Very
P13	Somewhat	Very	Very	Very	Somewhat	Not at all

Figure C.2: How familiar participants were with UML models

D
Appendix D

This appendix shows the evaluation guide including the Personal experience questionnaire, Usability test and Semi-structured interviews.

D.1 Introduction

Good morning. Thank you for participating in this evaluation.

The purpose of this session is to evaluate our tool, called the System Knowledge Composer, in terms of how well it helps comprehension of requirements, architecture and rationale of a software system, in this case, an open source e-commerce system.

The evaluation will take around 1.5 hours and consists of 3 main parts:

You will be given a few tasks to solve using the System Knowledge Composer. You will solve similar tasks, but this time using the written documentation instead. We'll ask you for your opinions and feedback of the tool.

During the session, we will be of assistance for questions you have regarding navigation on the kpage or just understanding what certain elements are. However, we cannot answer the tasks for you.

Since we have a limited amount of time, we do not expect you to try to understand the system in-depth. This means you can select several potential options for any question if you're not entirely sure of which one is correct.

We would also make it clear that it is the tool we are evaluating, this means whatever your answer is for the tasks, there is no right nor wrong answer, we only require that you try to answer to the best of your ability using the given information. We are mostly interested in how you find the answers and how you interpret the information you find.

Lastly, note that the system is still just a proof of concept, so you will likely find bugs and issues with the page navigation. Try not focus too much on that. The main focus of the evaluation is on the graphs and the presentation of the information you find.

D.2 Personal experience questionnaire

Question 1
Please indicate how many years of experience you have as a software developer?

Question 2
Please indicate your level of experience within web development?

- None
- Low
- Average
- High

Question 3
Please indicate which ones of the items below best describes your experience with software architecture.

- Never worked with it
- Know about it but have never used it to perform development tasks
- I have used software architecture documents to perform development tasks
- I have designed software architecture

Question 4
Please indicate how familiar you are with the following UML diagrams.

Not at all	**Somewhat**	**Very**
Use case diagrams		
Class diagrams		
State machines		
Sequence diagrams		
Deployment diagrams		
Package diagrams		

Table D.1: Level of familiarity with UML diagrams

D.3 Usability test

D.3.1 Explore System Knowledge Composer

Spend 10 minutes exploring System Knowledge Composer. We will guide you through some of the views.

D.4 Part 1 - Perform a task using System Knowledge Composer

Considering the following user story and using the Software Knowledge Composer, please answer the 6 questions on the following pages.
User story
Name: Add size filter
Description: As a customer
I want to be able to filter products based on available sizes So I can only see the products that are available in my size

Question 1:
Looking at the functionalities that exist in the system

- Which feature is more relevant/suitable to include this new “Add size filter” functionality?
- Explain briefly why you chose this feature.

Question 2:
Considering the feature **Display Products**

- Are there any additions or changes to use cases, requirements and/or user stories related to this feature that you think should be made for the new “Add size filter” functionality? If so, which are affected? Please elaborate briefly.

Question 3:
Considering the feature **Display Products**

- Can you find potential implementation classes you’ll have to make changes to, to include the new “Add size filter” functionality? Please elaborate briefly.

Question 4:
Considering the feature **Display Products**

- Can you find the class diagram of the logical classes involved in the architecture layer of this feature?
- In the architecture layer, can you name some of the development classes?
- Which development classes do think you you’ll have to modify to include the new “Add size filter” functionality? Please elaborate briefly.

Question 5:
Considering the feature **Purchase products**

- Can you find the behavior of this feature? Can you see sequence diagrams or state machine diagrams related to this feature?

Question 6:
Which architectural patterns are followed in the system?

- Can you find descriptions for these patterns?
- Different packages play different roles in this architectural pattern you chose. Can you name one of the packages and its role?
- Can you explain why the MVC pattern has been chosen?

D.4.1 Part 2 - Perform a task using software documentation

Considering the following user story, please use the software documentation to answer the following questions.
User story
Name: Shipping address same as billing address
Description: As a customer
I want to be able to choose if I want the shipping address to be filled with my billing address So that I don't have to fill in my address twice if the shipping and billing address are the same

Question 1:
Looking at the **features/functionalities** that exist in the system:

- Which existing feature/functionality is more relevant/suitable to include the new "Shipping address same as billing address" functionality?
- Explain briefly why you chose this feature/functionality.

Question 2:
Considering the feature/functionality for **account management** (alternative terms: user management or manage account):

- Are there any additions or changes to use cases, requirements and/or user stories related to this feature/functionality that you think should be made for the new "Shipping address same as billing address" functionality? If so, which are affected? Please elaborate.

Question 3:
Considering the feature/functionality for **account management** (alternative terms: user management or manage account)

- Can you find the class diagram of the conceptual classes involved in the architecture layer of this feature?
- Which potential conceptual classes do you think you'll have to modify to include the new "Shipping address same as billing address" functionality? Please elaborate.

Question 4:
Considering the feature/functionality for **account management** (alternative terms: user management or manage account)

- Looking at the internal design of this feature/functionality, can you name the classes in the server-side Model component that you think might need to be modified to include the new “Shipping address same as billing address” functionality?

Question 5:
Considering the feature/functionality for **account management** (alternative terms: user management or manage account)

- What is the behavior of this feature/functionality? Can you see sequence diagrams or state machine diagrams related to it?

Question 6:
Which **architectural patterns** are followed in the system?

- Which architectural patterns are followed in the system?

- Can you find descriptions for these patterns?

- Can you name and explain the 3 main components MVC pattern is divided into?

- Can you explain why a mix of thin-client and fat-client has been chosen for the system architecture?

D.5 Interviews

D.5.1 Part 3 - Feedback survey

Fill in the blanks for each of the following topics. You are allowed to name multiple items per topic. Note that the purpose of this evaluation is to learn the strengths and weaknesses of our tool. As such, we are interested in your honest opinions and welcome any opinions of the tool, whether they are positive or negative.

In comparison to using normal documentation, I like...
....
....
....
....
....

If I was to use this tool in practice, what if these features were available...
....

. . . .
. . . .
. . . .
. . . .

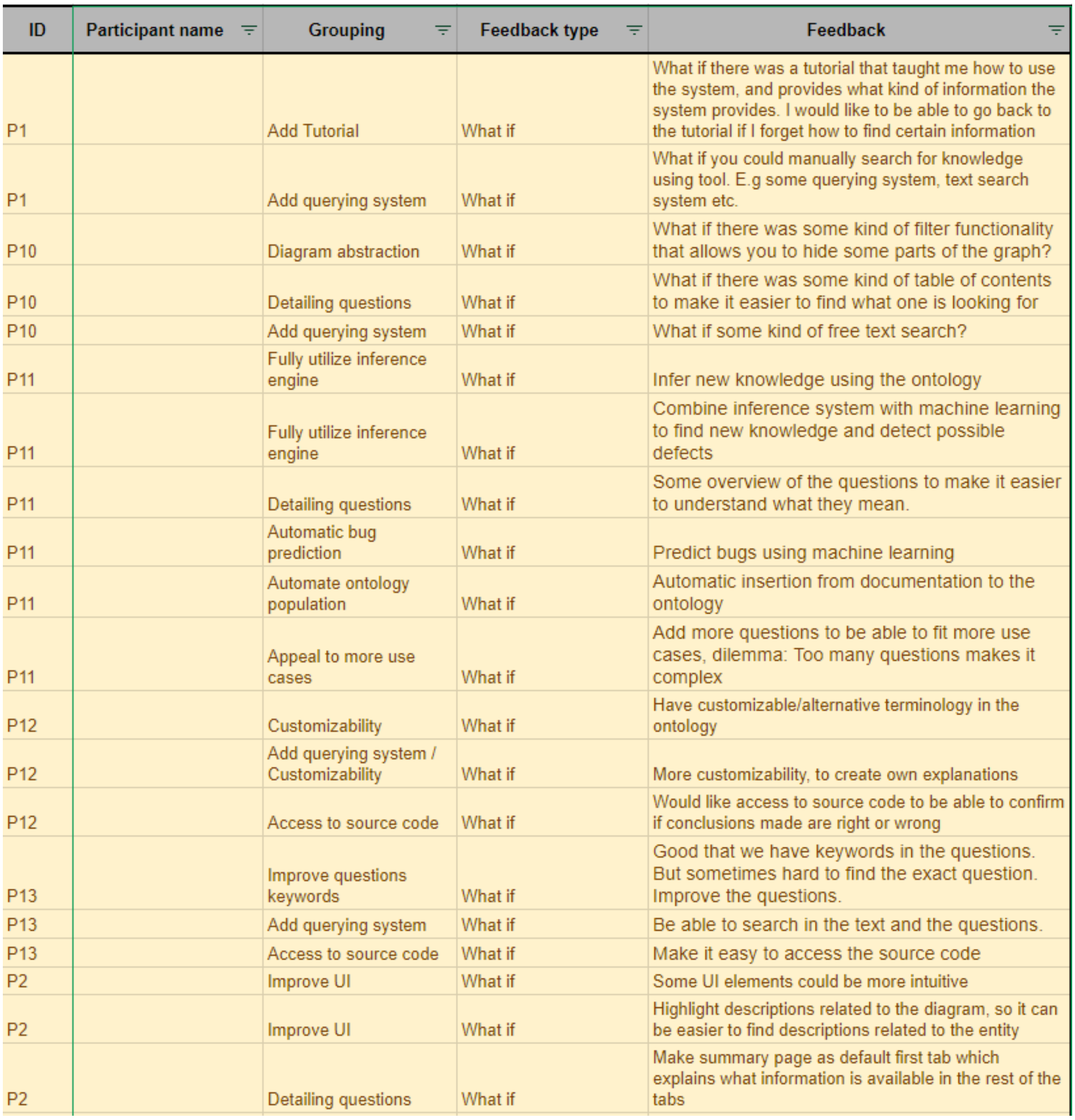
Without regard to resources, time or boundaries of current technology, I wish this was possible...
. . . .
. . . .
. . . .
. . . .
. . . .

Aside from the topics above, here's additional comments I'd like to make...
. . . .
. . . .
. . . .
. . . .
. . . .

E

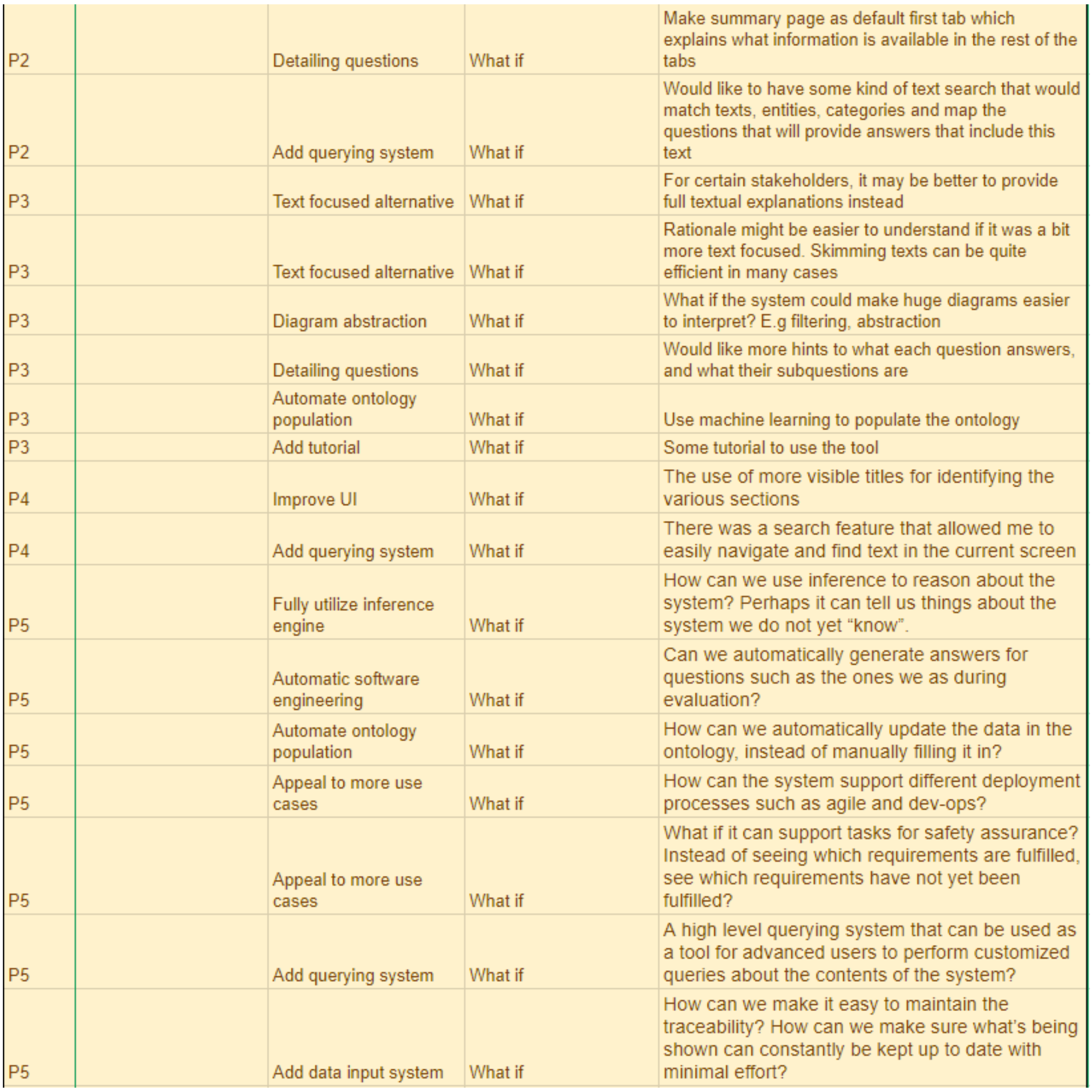
Appendix E

This appendix shows the qualitative results from the evaluations. There are two categories; categorization by the question that was asked (i.e I like, What if and Other) and by the answer that was provided which we call grouping.



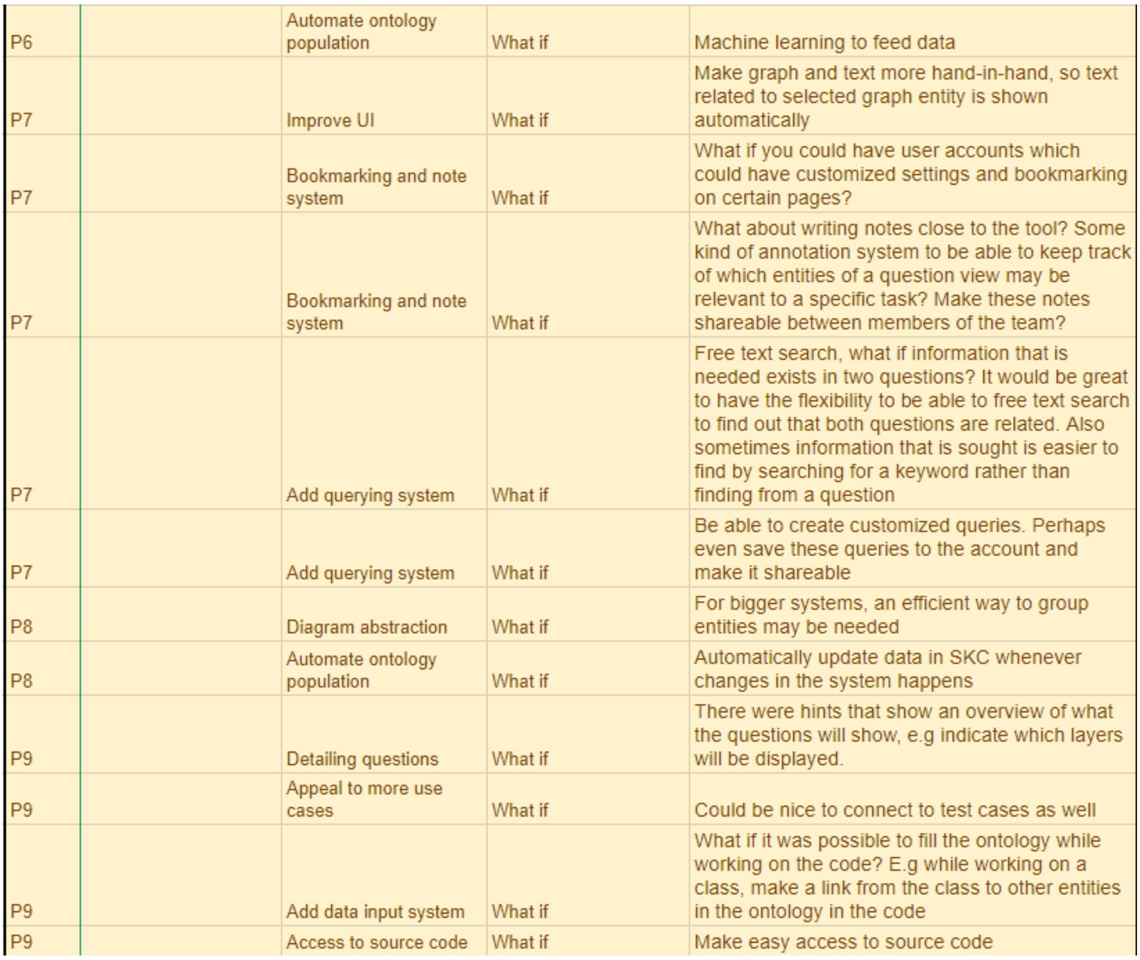
ID	Participant name	Grouping	Feedback type	Feedback
P1		Add Tutorial	What if	What if there was a tutorial that taught me how to use the system, and provides what kind of information the system provides. I would like to be able to go back to the tutorial if I forget how to find certain information
P1		Add querying system	What if	What if you could manually search for knowledge using tool. E.g some querying system, text search system etc.
P10		Diagram abstraction	What if	What if there was some kind of filter functionality that allows you to hide some parts of the graph?
P10		Detailing questions	What if	What if there was some kind of table of contents to make it easier to find what one is looking for
P10		Add querying system	What if	What if some kind of free text search?
P11		Fully utilize inference engine	What if	Infer new knowledge using the ontology
P11		Fully utilize inference engine	What if	Combine inference system with machine learning to find new knowledge and detect possible defects
P11		Detailing questions	What if	Some overview of the questions to make it easier to understand what they mean.
P11		Automatic bug prediction	What if	Predict bugs using machine learning
P11		Automate ontology population	What if	Automatic insertion from documentation to the ontology
P11		Appeal to more use cases	What if	Add more questions to be able to fit more use cases, dilemma: Too many questions makes it complex
P12		Customizability	What if	Have customizable/alternative terminology in the ontology
P12		Add querying system / Customizability	What if	More customizability, to create own explanations
P12		Access to source code	What if	Would like access to source code to be able to confirm if conclusions made are right or wrong
P13		Improve questions keywords	What if	Good that we have keywords in the questions. But sometimes hard to find the exact question. Improve the questions.
P13		Add querying system	What if	Be able to search in the text and the questions.
P13		Access to source code	What if	Make it easy to access the source code
P2		Improve UI	What if	Some UI elements could be more intuitive
P2		Improve UI	What if	Highlight descriptions related to the diagram, so it can be easier to find descriptions related to the entity
P2		Detailing questions	What if	Make summary page as default first tab which explains what information is available in the rest of the tabs

Figure E.1: What if, part 1



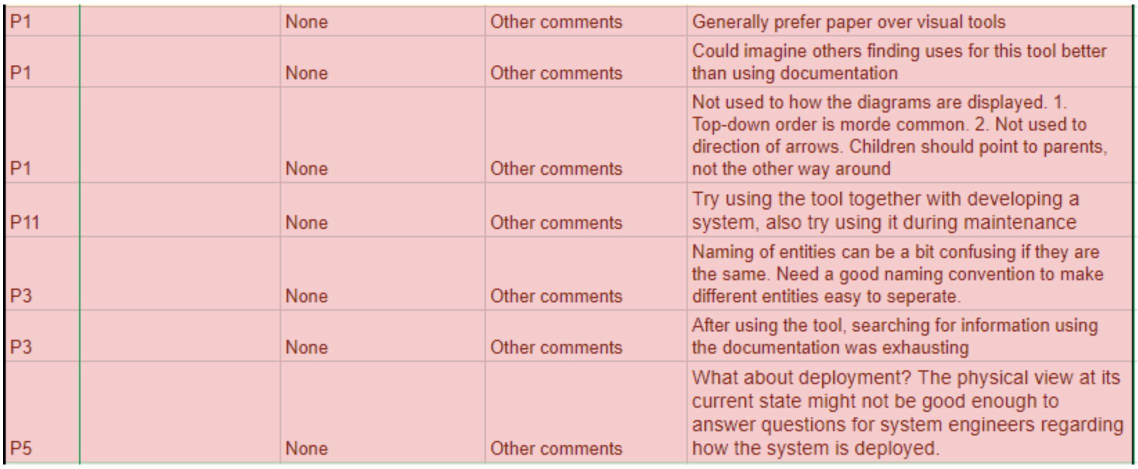
P2		Detailing questions	What if	Make summary page as default first tab which explains what information is available in the rest of the tabs
P2		Add querying system	What if	Would like to have some kind of text search that would match texts, entities, categories and map the questions that will provide answers that include this text
P3		Text focused alternative	What if	For certain stakeholders, it may be better to provide full textual explanations instead
P3		Text focused alternative	What if	Rationale might be easier to understand if it was a bit more text focused. Skimming texts can be quite efficient in many cases
P3		Diagram abstraction	What if	What if the system could make huge diagrams easier to interpret? E.g filtering, abstraction
P3		Detailing questions	What if	Would like more hints to what each question answers, and what their subquestions are
P3		Automate ontology population	What if	Use machine learning to populate the ontology
P3		Add tutorial	What if	Some tutorial to use the tool
P4		Improve UI	What if	The use of more visible titles for identifying the various sections
P4		Add querying system	What if	There was a search feature that allowed me to easily navigate and find text in the current screen
P5		Fully utilize inference engine	What if	How can we use inference to reason about the system? Perhaps it can tell us things about the system we do not yet “know”.
P5		Automatic software engineering	What if	Can we automatically generate answers for questions such as the ones we as during evaluation?
P5		Automate ontology population	What if	How can we automatically update the data in the ontology, instead of manually filling it in?
P5		Appeal to more use cases	What if	How can the system support different deployment processes such as agile and dev-ops?
P5		Appeal to more use cases	What if	What if it can support tasks for safety assurance? Instead of seeing which requirements are fulfilled, see which requirements have not yet been fulfilled?
P5		Add querying system	What if	A high level querying system that can be used as a tool for advanced users to perform customized queries about the contents of the system?
P5		Add data input system	What if	How can we make it easy to maintain the traceability? How can we make sure what’s being shown can constantly be kept up to date with minimal effort?

Figure E.2: What if, part 2



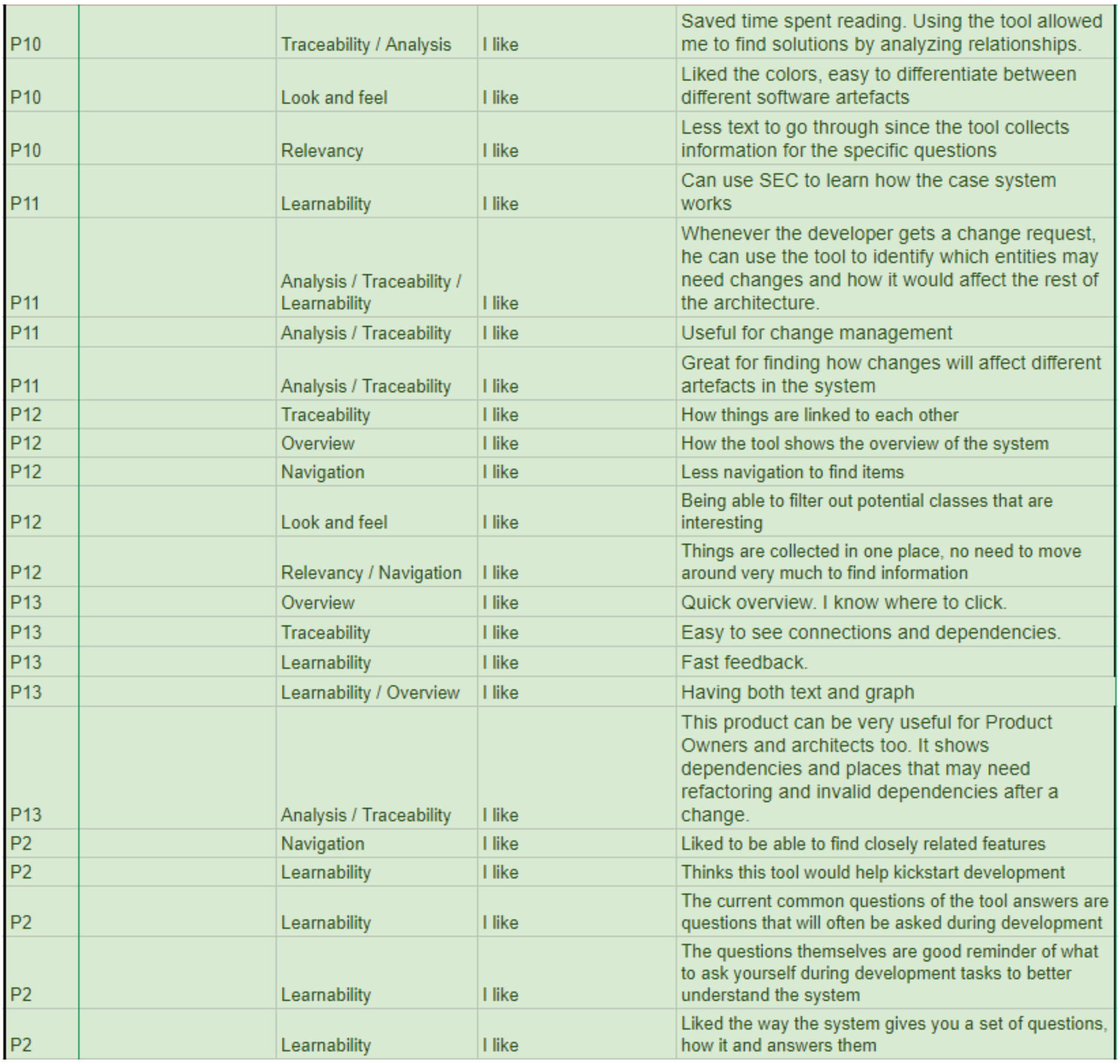
P6		Automate ontology population	What if	Machine learning to feed data
P7		Improve UI	What if	Make graph and text more hand-in-hand, so text related to selected graph entity is shown automatically
P7		Bookmarking and note system	What if	What if you could have user accounts which could have customized settings and bookmarking on certain pages?
P7		Bookmarking and note system	What if	What about writing notes close to the tool? Some kind of annotation system to be able to keep track of which entities of a question view may be relevant to a specific task? Make these notes shareable between members of the team?
P7		Add querying system	What if	Free text search, what if information that is needed exists in two questions? It would be great to have the flexibility to be able to free text search to find out that both questions are related. Also sometimes information that is sought is easier to find by searching for a keyword rather than finding from a question
P7		Add querying system	What if	Be able to create customized queries. Perhaps even save these queries to the account and make it shareable
P8		Diagram abstraction	What if	For bigger systems, an efficient way to group entities may be needed
P8		Automate ontology population	What if	Automatically update data in SKC whenever changes in the system happens
P9		Detailing questions	What if	There were hints that show an overview of what the questions will show, e.g indicate which layers will be displayed.
P9		Appeal to more use cases	What if	Could be nice to connect to test cases as well
P9		Add data input system	What if	What if it was possible to fill the ontology while working on the code? E.g while working on a class, make a link from the class to other entities in the ontology in the code
P9		Access to source code	What if	Make easy access to source code

Figure E.3: What if, part 3



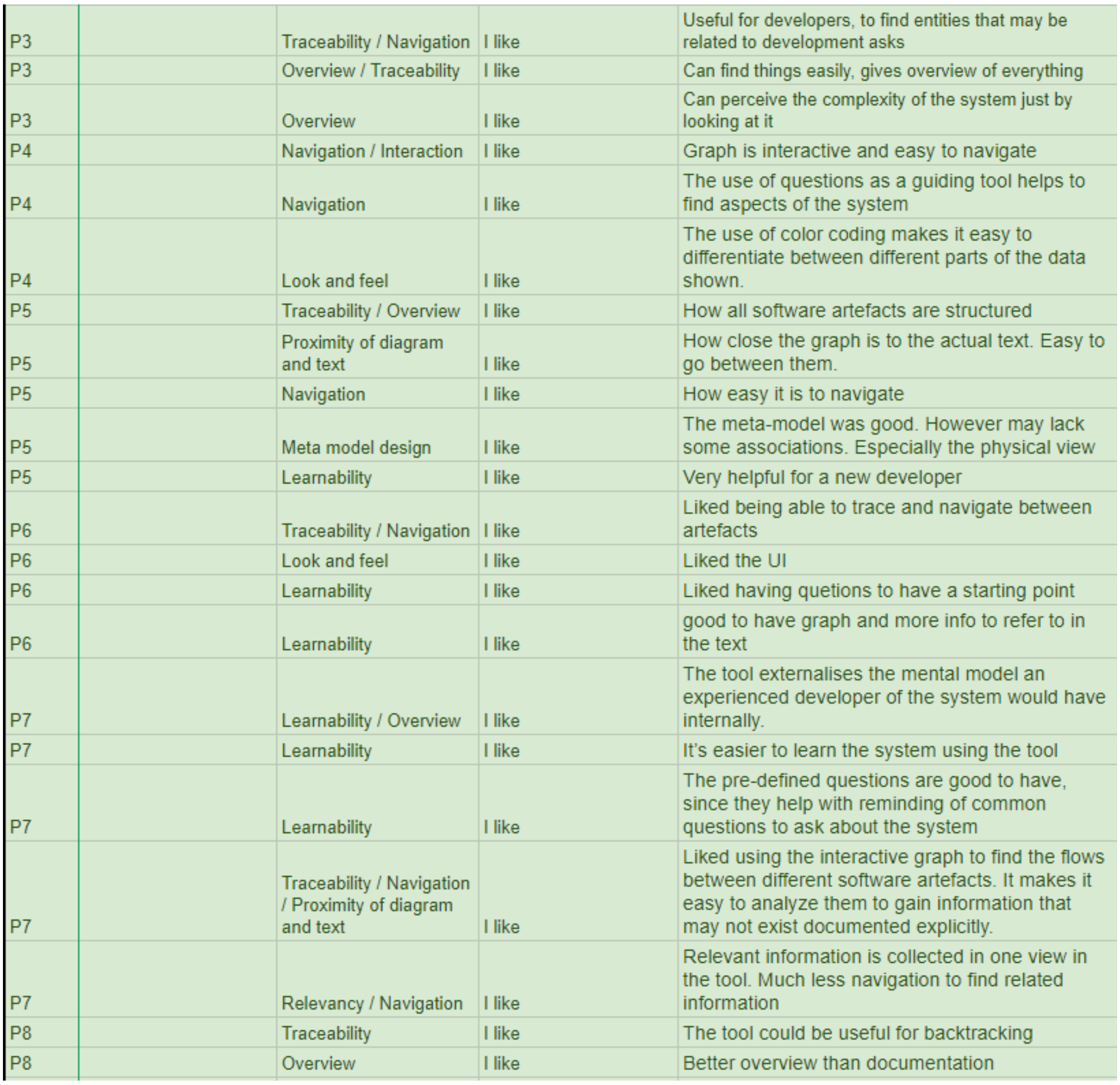
P1		None	Other comments	Generally prefer paper over visual tools
P1		None	Other comments	Could imagine others finding uses for this tool better than using documentation
P1		None	Other comments	Not used to how the diagrams are displayed. 1. Top-down order is morde common. 2. Not used to direction of arrows. Children should point to parents, not the other way around
P11		None	Other comments	Try using the tool together with developing a system, also try using it during maintenance
P3		None	Other comments	Naming of entities can be a bit confusing if they are the same. Need a good naming convention to make different entities easy to seperate.
P3		None	Other comments	After using the tool, searching for information using the documentation was exhausting
P5		None	Other comments	What about deployment? The physical view at its current state might not be good enough to answer questions for system engineers regarding how the system is deployed.

Figure E.4: Other



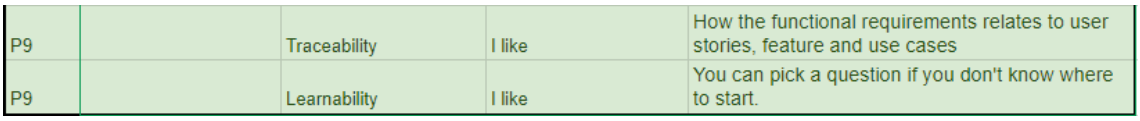
P10		Traceability / Analysis	I like	Saved time spent reading. Using the tool allowed me to find solutions by analyzing relationships.
P10		Look and feel	I like	Liked the colors, easy to differentiate between different software artefacts
P10		Relevancy	I like	Less text to go through since the tool collects information for the specific questions
P11		Learnability	I like	Can use SEC to learn how the case system works
P11		Analysis / Traceability / Learnability	I like	Whenever the developer gets a change request, he can use the tool to identify which entities may need changes and how it would affect the rest of the architecture.
P11		Analysis / Traceability	I like	Useful for change management
P11		Analysis / Traceability	I like	Great for finding how changes will affect different artefacts in the system
P12		Traceability	I like	How things are linked to each other
P12		Overview	I like	How the tool shows the overview of the system
P12		Navigation	I like	Less navigation to find items
P12		Look and feel	I like	Being able to filter out potential classes that are interesting
P12		Relevancy / Navigation	I like	Things are collected in one place, no need to move around very much to find information
P13		Overview	I like	Quick overview. I know where to click.
P13		Traceability	I like	Easy to see connections and dependencies.
P13		Learnability	I like	Fast feedback.
P13		Learnability / Overview	I like	Having both text and graph
P13		Analysis / Traceability	I like	This product can be very useful for Product Owners and architects too. It shows dependencies and places that may need refactoring and invalid dependencies after a change.
P2		Navigation	I like	Liked to be able to find closely related features
P2		Learnability	I like	Thinks this tool would help kickstart development
P2		Learnability	I like	The current common questions of the tool answers are questions that will often be asked during development
P2		Learnability	I like	The questions themselves are good reminder of what to ask yourself during development tasks to better understand the system
P2		Learnability	I like	Liked the way the system gives you a set of questions, how it and answers them

Figure E.5: I like, part 1



P3		Traceability / Navigation	I like	Useful for developers, to find entities that may be related to development asks
P3		Overview / Traceability	I like	Can find things easily, gives overview of everything
P3		Overview	I like	Can perceive the complexity of the system just by looking at it
P4		Navigation / Interaction	I like	Graph is interactive and easy to navigate
P4		Navigation	I like	The use of questions as a guiding tool helps to find aspects of the system
P4		Look and feel	I like	The use of color coding makes it easy to differentiate between different parts of the data shown.
P5		Traceability / Overview	I like	How all software artefacts are structured
P5		Proximity of diagram and text	I like	How close the graph is to the actual text. Easy to go between them.
P5		Navigation	I like	How easy it is to navigate
P5		Meta model design	I like	The meta-model was good. However may lack some associations. Especially the physical view
P5		Learnability	I like	Very helpful for a new developer
P6		Traceability / Navigation	I like	Liked being able to trace and navigate between artefacts
P6		Look and feel	I like	Liked the UI
P6		Learnability	I like	Liked having quetions to have a starting point
P6		Learnability	I like	good to have graph and more info to refer to in the text
P7		Learnability / Overview	I like	The tool externalises the mental model an experienced developer of the system would have internally.
P7		Learnability	I like	It's easier to learn the system using the tool
P7		Learnability	I like	The pre-defined questions are good to have, since they help with reminding of common questions to ask about the system
P7		Traceability / Navigation / Proximity of diagram and text	I like	Liked using the interactive graph to find the flows between different software artefacts. It makes it easy to analyze them to gain information that may not exist documented explicitly.
P7		Relevancy / Navigation	I like	Relevant information is collected in one view in the tool. Much less navigation to find related information
P8		Traceability	I like	The tool could be useful for backtracking
P8		Overview	I like	Better overview than documentation

Figure E.6: I like, part 2



P9		Traceability	I like	How the functional requirements relates to user stories, feature and use cases
P9		Learnability	I like	You can pick a question if you don't know where to start.

Figure E.7: I like, part 3