The present work was submitted to
the Research Group
Software Construction

of the Faculty of Mathematics,
Computer Science, and
Natural Sciences

Bachelor Thesis

Mapping and tracing security design solutions in architectural descriptions

presented by

Nicolas van Bellen

Aachen, August 15, 2024

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Alex Sabau, M.Sc.

Acknowledgment

First of all, I would like to thank Prof. Dr. rer. nat. Horst Lichter for the opportunity to write my bachelor thesis at his chair. I would also like to thank him and Prof. Dr. rer. nat. Bernhard Rumpe for evaluating my thesis.
I would also like to thank my supervisor M.Sc. Alex Sabau for the opportunity to write with him, his assistance, ongoing support and constructive criticism during the work.
I would also like to thank my friends who helped me with the revision. Also my family for always supporting me emotionally, morally and financially during my studies. I would especially like to thank my girlfriend for her patience and motivating support.

Nicolas van Bellen

Abstract

To improve security in the design of software system architectures, a new approach of security by design has been developed. As part of this concept, which aims to integrate security requirements with the software system architecture, a mapping and tracing between non-functional requirements (NFR) and security design solutions (SDS) is to be developed. For this purpose, a proof of concept (POC) was designed and implemented. This demonstrates a possible way to address the problem. In addition to the POC, two manipulation concepts were tested, aiming to make it easier for users to understand the SDS for the mapped NFR.

Contents

List of Tables

List of Figures

1 Introduction

Contents

The importance of software security has grown significantly due to the rapid growth of software development, which means that the risk of cyber attacks is increasing[McG04]. Negligent software security leads to non-negligible risks that affect individuals and companies. Software security is one of the most important quality attributes today[Mat17]. Despite the recognition of the importance of software security, there are difficulties in adopting security practices into the software development cycle[KK18]. Many methods do not directly incorporate software security into the development phase through requirements engineering[KK18]. One way to solve the problem is security by design. It ensures ground-laying security starting in the early days of the software lifecycle [Tum+20]. By integrating security from the beginning, potential vulnerabilities can be addressed proactively rather than reactively. The established approach to maintaining security architectures is independent of the overall architecture. This has several disadvantages, such as an increase in redundancies in the design documents, synchronization errors due to simultaneous changes to separate models and the expertise required to make security decisions. A new approach to security by design by Alex Sabau proposes to eliminate the development of separate security analyses by unifying security requirements with software system architectures. The unification requires traceability of security requirements that can be mapped to modeling elements in the architecture description. [Sab24]

In this thesis, an application of the mapping and tracing concept will be designed and implemented for the first time in a proof of concept. The focus will be on *Non-functional requirements*, which are to be mapped to a software architecture so that the *NFR* can subsequently be traced, thereby making the *Security Design Solution* traceable to the requirement.

1.1 Motivaton and Problem Statement

Due to the growing relevance of software security, it is prudent to incorporate security concepts during the planning phase of software development. This ensures a high level

of software security at an early stage in the software lifecycle. Initial planning also reduces the emergence of security vulnerabilities during implementation, as security decisions have already been made, thereby alleviating some of the responsibility from the developer. However, the problem is that separate security architectures are prone to errors. Additionally, as the size of the project grows, security documentation can become very confusing. This problem can be addressed by mapping *Non-functional requirement* to *Security Design Solution*. Consequently, implemented solutions can be traced back to security requirements, making them comprehensible in the long term. The goal is to adapt the concept developed by Sabau[Sab24] to the mapping and tracing of security design solutions and subsequently implement it. This is intended to create added value for the original concept and contribute to the further development of the approach.

1.2 Research questions

Consequently, two research questions have emerged.

- How can a proof of concept be designed and implemented so that the mapping and tracing between a non-functional requirement and security design solutions are realized?
- How can the traced security design solution be presented to the user in a manner that is as comprehensible and clear as possible?

1.3 Structure of this thesis

The work is divided into three logical sections.

First, the *Background* chapter 2 introduces the Background to the work. They are intended to create a basic knowledge of architecture models, requirements tracing and Non-functional requirements. The *Related Work* chapter 3 introduces similar general requirement tools and a software modeling tool. This should provide a rough outline of where this thesis starts.

The second section begins with an introduction to the developed *Concept* 4. This is intended to provide a theoretical understanding and present a graphical modeling of the concept in addition to the foundations of the concept. The *Design* 5 is then presented with the design of the software architecture.

The third section of the thesis is the *Implementation* of the Proof of Concept in chapter 6. Finally, the *Evaluation* 7 follows and the *Conclusion* 8 summarizes the work once again and presents possibilities for improving the proof of concept developed in this thesis.

2 Background

Digitalization is based on models, which is evident because digital data represents a model that depicts a part of the real world.

Prof. Dr. rer. nat.
Bernhard Rumpe

Contents

In this chapter, the foundations for understanding the rest of the thesis are established. First, a general overview of software architectures2.1 is provided. Next, general requirement tracing 2.2 is introduced, and finally, non-functional requirements 2.3 are explained.

2.1 Software Architectures

Software architecture is a crucial field in software development that describes the structure and behavior of software systems [Che+10]. It defines components and their relationships, guiding the evolution of these systems. Important factors include modularity, maintainability, reusability, scalability, and security [CMC15], with security becoming increasingly prominent today.

To implement these factors, fundamental concepts exist within a software architecture. A software architecture consists of components, each fulfilling specific tasks. Interfaces define how components communicate with each other. Every Software architecture can also be divided into layers, with each layer having its own responsibilities, such as the presentation layer, business logic layer, and data layer [VT16].

Generally, the process of software architecture can be divided into six steps: requirements formulation, architectural design, component design, implementation, testing, and maintenance [PB11]. For this thesis, the requirements and architectural design are particularly relevant. Requirements specify what the software must accomplish and can be functional or non-functional. *Non-functional requirement* are explained in Chapter 2.3. In the architectural design phase, *NFR* will later be mapped and traced.

2.2 Requirement Tracing

Requirement tracing is the process of linking high-level requirements to low-level design elements and software artifacts throughout the software development lifecycle [ZK11]. This ensures that all requirements are correctly implemented from capture through development to verification. Consequently, relationships between various requirements and other development factors, such as design or code, can be documented and tracked. Surplus it supports change management and quality assurance. Requirement tracing becomes difficult with increasing complexity due to various levels of abstraction[MRA15]. Maintaining traceability links is also a challenge, especially when requirements change. Requirement tracing involves following and identifying the life of requirements in forward or backward directions during the software development lifecycle [MZ22]. Specialized requirements management tools help maintain an overview and manage the linking of requirements in practice. In Chapter 3.2.1, such a tool is introduced.

2.3 Non-Functional Requirements

Quality, performance, and operating conditions of a system are the fundamental attributes to which a *Non-functional requirement* pertains. There are six different categories of requirements [SL07]:

- Performance requirements, such as response time.
- Reliability requirements, including fault tolerance.
- Security requirements, encompassing authentication.
- Usability requirements, with efficiency.
- Maintainability requirements, including readability.
- Portability requirements, such as compatibility.

For software development, non-functional requirements are essential. They define the quality attributes of a software product.[Ali+22]. The context of this thesis revolves around security requirements. *NFRs* can have intricate relationships with one another or may even be mutually exclusive, so they must be carefully formulated [SL07].

3 Related Work

Contents

In this chapter, two requirement tracing tools are first introduced, followed by a software modeling tool.

3.1 Requirement Tracing Tools

3.1.1 Jira

Jira is a requirement tracing tool that manages requirements and tasks. Tracing in Jira works by collecting tickets, which automatically creates documentation. The tickets can be accessed at any time, allowing past decisions to be reviewed. Additionally, Jira traces the current progress of a project by creating a workflow. This workflow is a path defined in Jira that tracks a task from creation to completion.

3.1.2 Jama Software

Jama is a tool that offers requirements management and traceability tools. With this tool, you can monitor in real-time how requirements progress through the development process. This ensures that the requirements are fulfilled as planned. The progress is traced, allowing for quick intervention and correction of any erroneous changes to the requirements.

3.2 Software Modelling Tool

3.2.1 Enterprise Architect

"Enterprise architecture (EA) is the definition and representation of a high-level view of an enterprise's business processes and IT systems, their interrelationships, and the extent to which these processes and systems are shared by different parts of the enterprise." [Tam+11] The goal is to provide an organization with planning capabilities. Besides that EA is to align IT with the business to ensure that the business is supported and enabled by IT [MVV18]. It supports the modeling of software and business systems. EA offers a variety of modeling languages, such as UML and BPMN. It also includes lifecycle

management, covering the software development lifecycle, requirements analysis, design, and implementation.

EA is a powerful tool for enterprises and developers who need a robust and flexible solution for modeling and managing their software and system architectures.

4 Concept

Contents

In the chapter 4.1, the foundation of the concept will be explained first and then the concept of mapping and tracing security design solutions in architectural descriptions will be introduced in chapter 4.2. Afterwards the Process in chapter 4.3 and the possible mapping cases in chapter 4.3.1 will be explained. The modeling elements and semantics of UML class diagrams are used to represent the concept models and also later to represent the design.

4.1 Conceptual Foundations

The concept of this thesis is based on the paper "A Guided Modeling Approach for Secure System Design" [Sab24]. To create a basic understanding of the concept of this thesis, I will briefly outline the idea of the paper and then introduce the definitions used in the underlying paper that are relevant to the concept of this thesis.

4.1.1 Idea

Designing security architectures independently of the overall architecture is a well-researched area. However, previous approaches have disadvantages in the documentation of security architectures, such as the increase in redundancy in design documents, synchronization errors due to simultaneous changes to separate models, and the fact that expert knowledge is required to develop security architectures. However, experts with the necessary knowledge are rare, creating a bottleneck. [Sab24]

To overcome these drawbacks, the paper proposes an approach that eliminates the need for multiple security architectures and supports architects in modeling secure software systems. [Sab24] The goal is to develop a tool that maps security requirements to elements of the architectural description. This aims to eliminate the division between security architectures and software system architectures. Consequently, security requirements can be directly mapped to the architecture. Through tracing, it is possible to identify which elements of the architecture constitute the solution to a security requirement. Furthermore, this approach aims to provide recommendations for improving

the security of the existing software system architecture. This will make designing software architectures easier and more accessible. In the following chapter, the conceptual foundations on which the *POC* is based are explained.

4.1.2 Architecture Structure

Figure 4.1 shows the structure introduced in the paper for a Software Systems architecture on which the concept of this thesis is based.

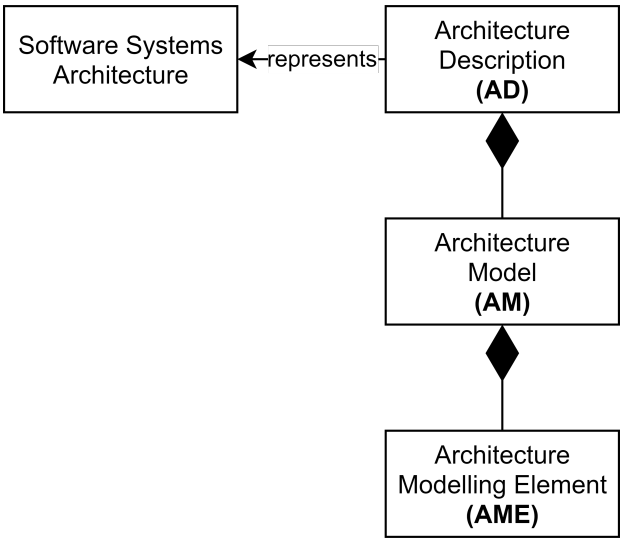


Figure 4.1: Architecture Structure

- An ***Architecture Description*** (***AD***) refers to the set of design documents describing a software system's architecture. The design documents can take any form that helps to describe a software system.
- An ***Architectural Model*** (***AM***) is a subset of an *AD* and is a set of model elements. It represents exactly one model, that models certain aspects of the architecture.
- An ***Architectural Modelling Element*** (***AME***) is an atomic element of an *AM* and an *AM* is composed of a set of *AMEs*.
- A ***Security Design Solution*** (***SDS***) is the set of *AMs* and *AMEs* that fulfil a given requirement.

[Sab24]

4.2 Concept Model

The goal is to create a unidirectional mapping between an *Non-functional requirement* and the designed solution contained in the *Architecture Description* for the given *NFR*. In order to model the concept for this thesis, the original concept model from Alex Sabau's work [Sab24] was simplified for ease of understanding and adapted to the use case.

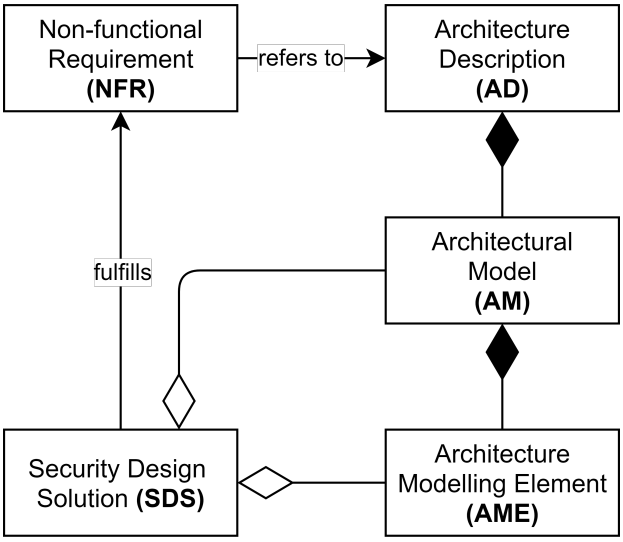


Figure 4.2: Concept Model

The structure of the *AD*, *AM* and *AME* works as explained in chapter 4.1.2. The *SDS* is also defined as in chapter 4.1.2. It is the set of *AMs* and *AMEs* that fulfill the given *NFR*. One *NFR* refers to only one *AD*.

With the *NFR* as the selector and the *AD* as the source the challenge is to realize the mapping between both and form an *SDS*.

4.3 Process Model

As you can see in Figure 4.3, the architect gives a set of *Non-functional requirements* as input. Those are to be mapped to the parts of the *Architecture Description* in which the requirements are implemented. On this basis, *Architectural Models* are formed from an existing *Architecture Description* that describes the architecture: It is assumed that an *AD* consists of a set of graphs. Each graph is an *AM*. Consequently, all components of a graph are *Architectural Modelling Elements*. In order to be able to represent *AMEs*,

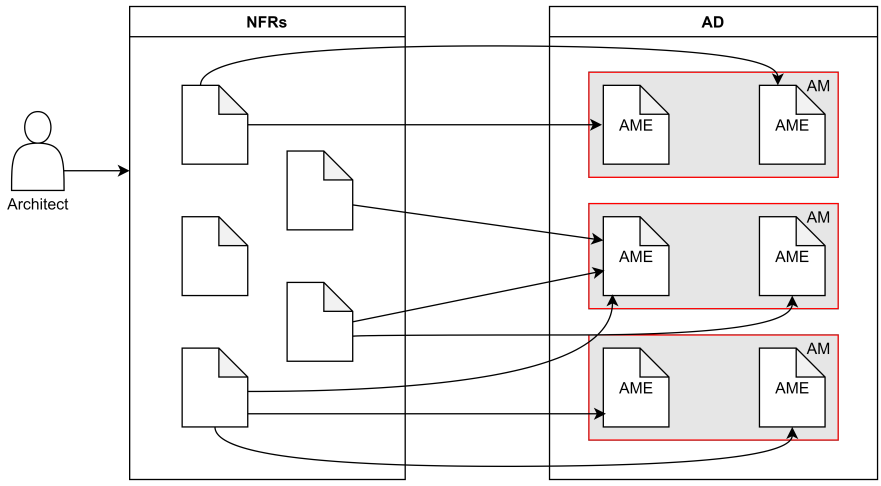


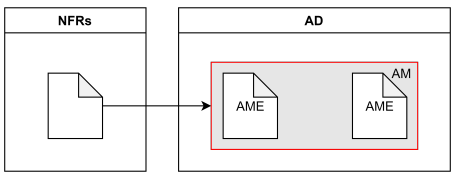
Figure 4.3: Process Model

the graphs must be able to be manipulated. The challenge now is to filter out exactly those graphs or subgraphs that can be mapped to a specific *NFR*. Only one *NFR* can be traced to an *SDS* at one time. Once the mapping has been created, the relation can be traced and the subset belonging to an *NFR* can be visualized. This subset is an *Security Design Solution*. With this *SDS* the user is able to understand how the requirement was solved and implemented. As one can see in Figure 4.3 different mapping cases can occur during the process.

4.3.1 Mapping Cases

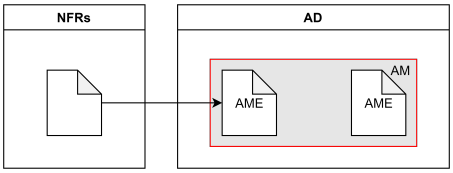
Five different cases can occur during the tracing, which are explained in this section.

1. A *NFR* refers to one *AM*



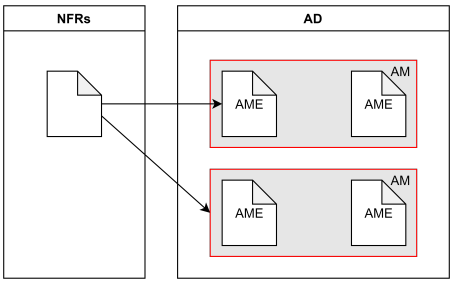
Let D and A be a set of *AMs*.
The *AM* is the minimal subset A of the *AD* D. The given set A fulfills the requirement of the *NFR*.

2. A *NFR* refers to one *AME*



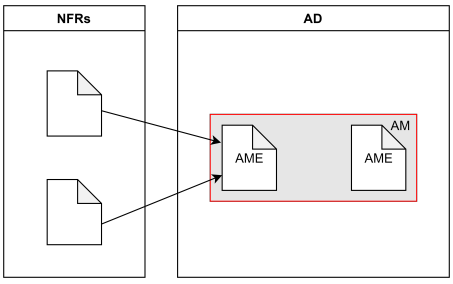
Let D, A and B be a set of *AMEs*.
The *AM* is a finite subset A of D, representing the *AD*. Furthermore is the *AME* also the minimal subset B from the set A. The given set B fulfills the requirement of the *NFR*.

3. A *NFR* refers to several *AMs* or *AMEs* from different *AMs*



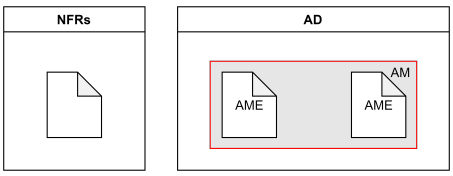
Let D, A_1, A_2, B_1 and B_2 be a set of *AMEs*. With $A_1 \neq A_2$.
The *AMs* A_1 and A_2 are finite subsets of D, representing the *AD*. The *AMEs* are the minimal subsets B_1 from the set A_1 and B_2 from the set A_2. The union of B_1 and B_2 fulfills the requirement of the *NFR*.

4. Several *NFRs* refer to one *AM*/*AME*



Let D, A and B be a set of *AMEs*.
The *AM* is a finite subset A of D, representing the *AD*. Furthermore is the *AME* also the minimal subset B from the set A. The given set B fulfills the requirements of both given *NFRs*.

5. A *NFR* does not refer to any *AM*/*AME*



Let D be a set of *AMs*.
No subset of D that represents the *AD* fulfills the *NFR*.

Conclusion

The 5 cases presented are fundamental for the correct implementation of the *POC* and to develop a functioning tracing. The logic for tracing, which is explained in more detail in chapter 6, is based on them. The following section presents two approaches for visualizing the traced data.

4.3.2 Manipulation Concepts

Two different approaches have been explored for returning the *SDS* to the user. The two figures 4.4 and 4.5 are based on the previously shown process model 4.3. Both concepts aim to achieve clarity and comprehensibility for the user.

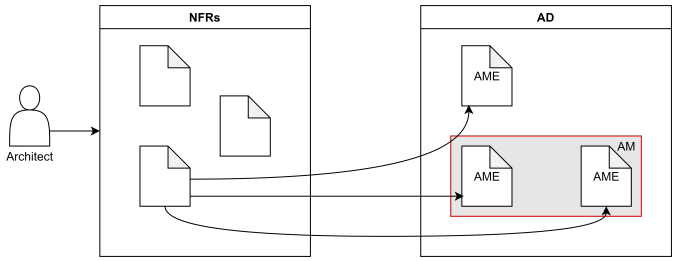


Figure 4.4: Example for Manipulating Concept A

A: Delete all non-relevant model elements When displaying the *SDS* of an *NFR*, only the *AMs* or *AMEs* relevant to fulfill the requirement are shown.

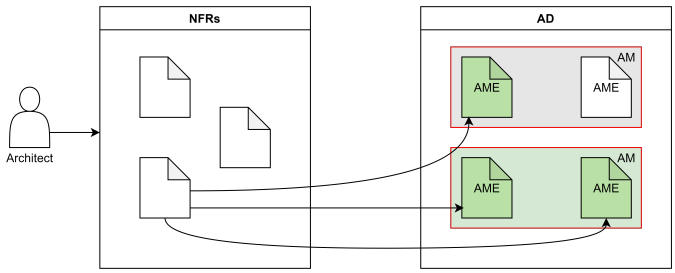


Figure 4.5: Example for Manipulating Concept B

B: Color highlighting of relevant model elements When displaying the *SDS* of an *NFR*, only the *AMs* which contain at least one *AME* are displayed. Additionally the relevant *AMs* or *AMEs* are highlighted in color.

5 Design

Contents

This chapter presents the design of the software architecture of the *Proof of Concept*. The description follows a top-down approach. Section 5.1.1 presents the individual layers into which the *Proof of Concept* has been divided. Section 5.1.2 analyzes the components that make up the layers. Section 5.2 then presents the architecture from a dynamic perspective. The communication between the components and modules at runtime for the two manipulation concepts is presented in this chapter by sequence diagrams.

5.1 Static View

5.1.1 Layer

First, an overview of the software architecture should be created. Figure 5.1 shows the architecture from a software perspective in the form of its 3-layer architecture. It can be seen that the view layer is formed by the *Frontend*, the *Business-Logic* layer by the *Backend* and the *Persistence* by the *Data Access Layer*. The *Backend* provides the *Frontend* with an interface.

The *Backend* was designed according to a strict layer hierarchy to easily integrate into the existing 3-layer architecture of the software, as shown in Figure 5.2. Each individual layer was designed in such a way that it can only access the underlying layer, thus preventing any layers from being skipped. This creates a clear hierarchy in which each layer has its own area of responsibility [LL13]. A component is a part of software that provides a set of services to its environment via well-defined interfaces. By adhering to a layer hierarchy, the associated functional encapsulation achieves a low level of inter-dependency and a high level of cohesion in the architecture [LL13]. This concept improves the maintainability and comprehensibility of the components.

The design principle was also based on the architecture pattern of the *Model-View-Controller (MVC)*. The *MVC* divides the software architecture into three components: A model, one or more views and a controller [LL13]. The model component implements the technical functionality of the application, a view presents the data to the user using

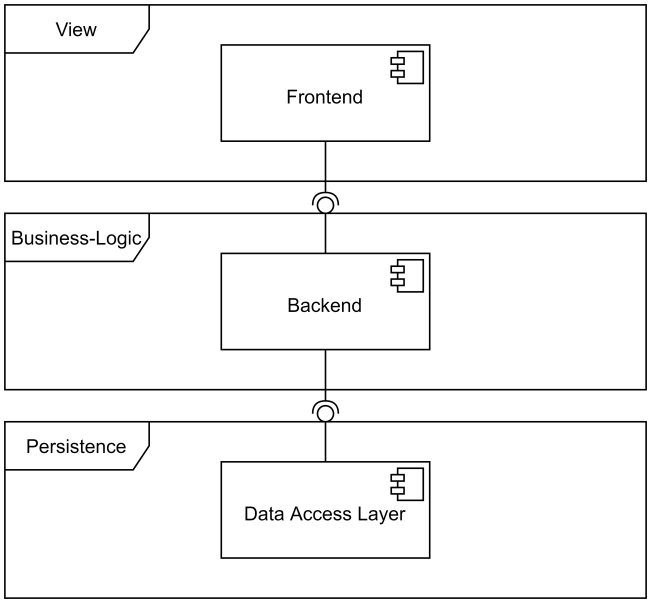


Figure 5.1: Layer model as an overview

the access functions of the model. Each view is assigned to a controller that receives the user's input and, depending on the user's input, informs the view of changes or calls up the model's application functions.

The design of the architecture according to the *MVC* architecture pattern results in a clear separation of responsibilities [LL13]. Each component has its own responsibility. This means that the maintenance and further development of the individual components is promoted, as the components are kept lean and uncomplicated [LL13]. The software architecture was divided into three layers, as you can see in Figure 5.2, described from the bottom to the top.

Persistence

The persistence layer forms the lowest layer of the software architecture and is realized by a repository in the *Backend*. For this *Proof of Concept* The data is stored as in-memory storage in the *Data Access Layer*. The data will be accessed by the logic layer.

Logic

Any access to the persistence layer is only made by the logic layer. This is the case because the *Proof of Concept* should be kept as small as possible due to the limited time available for the thesis. Therefore, there is no database but only a repository in

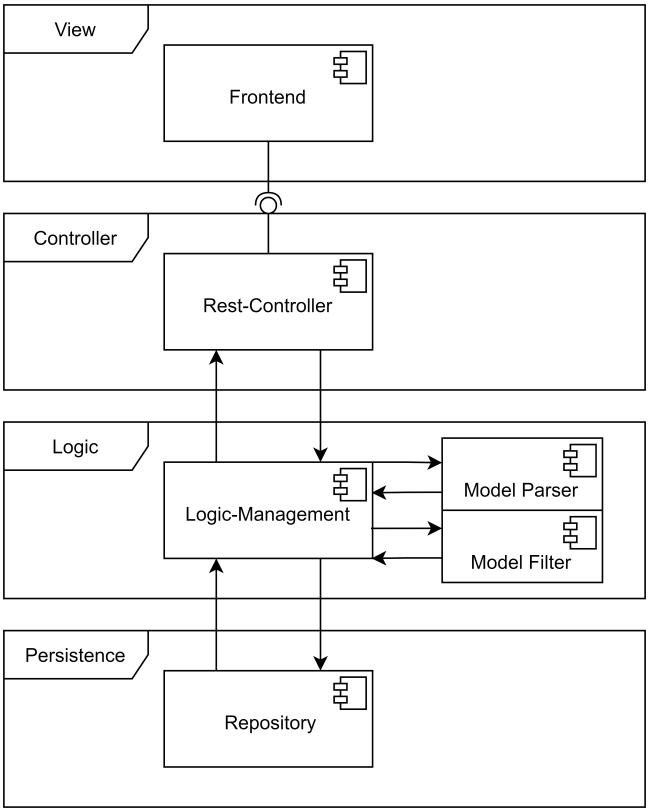


Figure 5.2: Layer model in detail

the *Backend*. The model parser and the model filter, seen in the figure, form functional modules and are used directly in the logic management. A more detailed description can be found in chapter 6. The logic layer also realizes the functional separation of responsibilities within the layers. The functions for manipulating the *AD* and creating the *SDS* for the given *NFR* are performed in the logic layer. The logic layer requests the *AD* from the persistence layer. The parsed *SDS* is passed on to the next higher layer, the controller.

Controller

The controller is responsible for the communication between the *Backend* and the *Frontend*, it receives the data from the *Frontend* and sends the data processed in the logic layer back to the higher view layer as a response.

View

The view layer is the top layer in the software architecture. It communicates with the controller as the interface to the *Backend*. The *Frontend* visualizes the *NFRs* and the associated *SDSs* as graphs. To do this, it sends a request to the controller and receives the *SDS* in response. The *Frontend* of the *Proof of Concept* is kept as simple as possible due to the limited time available for the thesis and due to limited *Frontend* programming experience. By designing in a strict layered architecture, a software architecture was created that has a high degree of modularity. This means that the software was structured into different components, which can be further developed as independently of each other as possible without restricting functionality [LL13].

5.1.2 Components and Interfaces

The high level of modularity at layer level was also realized at component level. Figure 5.3 shows the *Proof of Concept* at component level. The View layer is represented by the *Frontend*, which consists of only one component. The Controller layer is represented by the RestController. The Logic layer consists of the Logic-Management and the Utils. The Logic-Management consists of the SdsModelBuilder and the Utils contain two components. The Persistence layer consists of the repository, which has two components. The Controller layer, the Logic layer and the Persistence layer are located in the *Backend*.

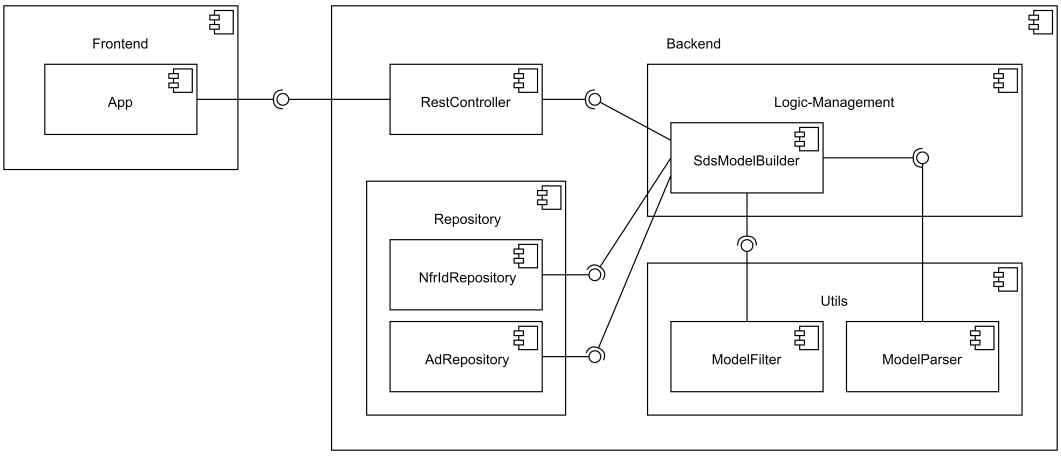


Figure 5.3: Overview of the components in the component diagram

App

The App implements the interface to the RestController from the *Backend* in the form of a REST-API. The REST-API will be explained further in chapter 6.1. It also implements

a list of *NFRs* and the option of displaying an *SDS* as a graph from the data transferred by the RestController. Therefore it sends a request with the ID of the *NFR* via the REST-API to the Controller. The App represents the View of the *MVC* pattern.

RestController

The RestController was designed according to the controller of the *MVC* pattern. It implements the interface to the *Frontend* in the form of a REST-API. The requests from the *Frontend* are received here and parameters are processed so that they can be passed on to the Logic-Management.

Logic-Management

The Logic-Management consists of the SdsModelBuilder. It is designed according to the model of the *MVC* pattern. This implements the logic that is required for the mapping. It receives the required parameters from the Controller. The SdsModelBuilder retrieves the *AD* and other required parameters from the Repository. A more in-depth explanation is given in chapter 6.3.3. The model filter is used to filter the required subset from the AD. The SdsModelBuilder then creates the *AM* and *AME* models from the returned subset of the *AD*. The filtered *AM* and *AME* are grouped as an *SDS* model. The SdsModelBuilder then uses the ModelParser to parse the *SDS* model so that it can be returned to the controller.

Utils

The Utils consist of the ModelFilter and the ModelParser. After a request from Logic-Management, the ModelFilter filters out the components of the *AD* required for the *NFR*. Depending on which visual approach (chapter 4.3.2) was selected by the user, it returns the result to Logic-Management. The ModelParser receives an *SDS* model from Logic-Management and parses it so that it can be forwarded to the RestController, which can then hand it over to the App.js.

Repository

The Repository consists of the NfrIdRepository and the AdRepository. The NfrIdRepository contains IDs of *NFRs*, with the associated information about which *AD* and which subset of the *AD* belongs to the *NFR*. If the ID does not exist but is requested by the Logic-Management, an error is returned. If the requested ID does exist, Logic-Management receives the dataset. The AdRepository contains all stored *ADs* and returns the required *AD* on request from Logic-Management. If the requested *AD* does not exist, an error is also returned.

Now that the layers and components have been introduced, the entities still need to be introduced.

5.1.3 Entities

AM

An *AM* entity is an object that contains all the information needed to represent the *AM* as a graph. It can be created using the AmFactory, which is based on the factory pattern. The attributes can be manipulated according to requirements.

AME

An *AME* entity is an object that contains all the information needed to represent the *AME* as a graph. It can be created using the AmeFactory, which is based on the factory pattern. The attributes can be manipulated according to requirements.

SDS

The *SDS* entity is an object that consists of an *AM* or *AME* object. The *SDS* object is used in a list that represents the set of *AMs* and *AMEs* that fulfill an *NFR*.

Now that the *Proof of Concept* has been presented from a static perspective, the dynamic view of the architecture follows.

5.2 Dynamic View

The dynamic view considers the *Proof of Concept* and its components at runtime, i.e. how the components work together at runtime [LL13]. In section 5.2.1, the communication of the individual components at runtime is visualized with the help of sequence diagrams.

5.2.1 Communication

According to Manipulating Concept A and B, there are two different processes for communication between the individual components. These are explained in more detail in this chapter and the differences identified. Figure 5.4 shows the process of Visualization Approach A and Figure 5.5 the one of B.

In both concepts, the process begins with a GET request via the REST-API from the App.js to the RestController, in which the NfrId and the VABool are provided. The NfrId is an atomic ID that uniquely identifies an *NFR*. The VABool is a Boolean that is true for Manipulating Concept A, for example, then it is false for B or vice versa. The RestController passes the two parameters to the Logic-Management. This requests the dataset that belongs to the *NFR* from the repository with the NfrId. As soon as it receives the dataset, the *AD* is requested using the *NFR* dataset.

Approach A From now on, the two approaches differ via the set VABool. As you can see in Figure 5.4, the filter method with the *AD* and a components list as parameters is now passed for approach A. The components list comes from the *NFR* data set and contains all component IDs that are relevant to the *NFR*. It is used to filter the *AD* and delete all non-relevant model elements. Once the modelList has been returned to Logic-Management, the *AMs* and *AMEs* are created and saved in an *SDS* list.

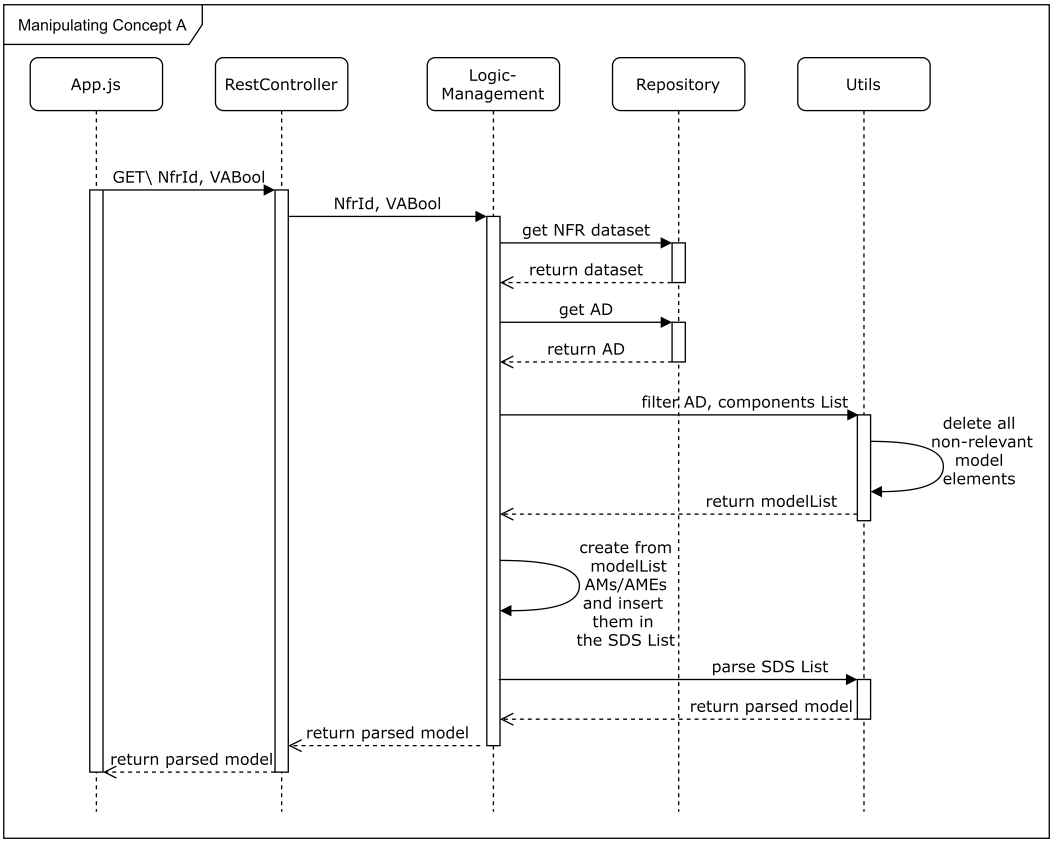


Figure 5.4: Communication with the visual approach A (chapter 4.3.2) in a sequence diagram

Approach B The second approach, as shown in Figure 5.5, creates the atomically associated *AMEs* from the complete *AD* and packs them into the *SDS* list so that the relevant *AMEs* can then be colored by ID from the component list provided by the *NFR* dataset.

The *SDS* list is now passed to the parser, which parses it and then it can be passed first to the Logic-Management, then to the RestController, and then to the App.js.

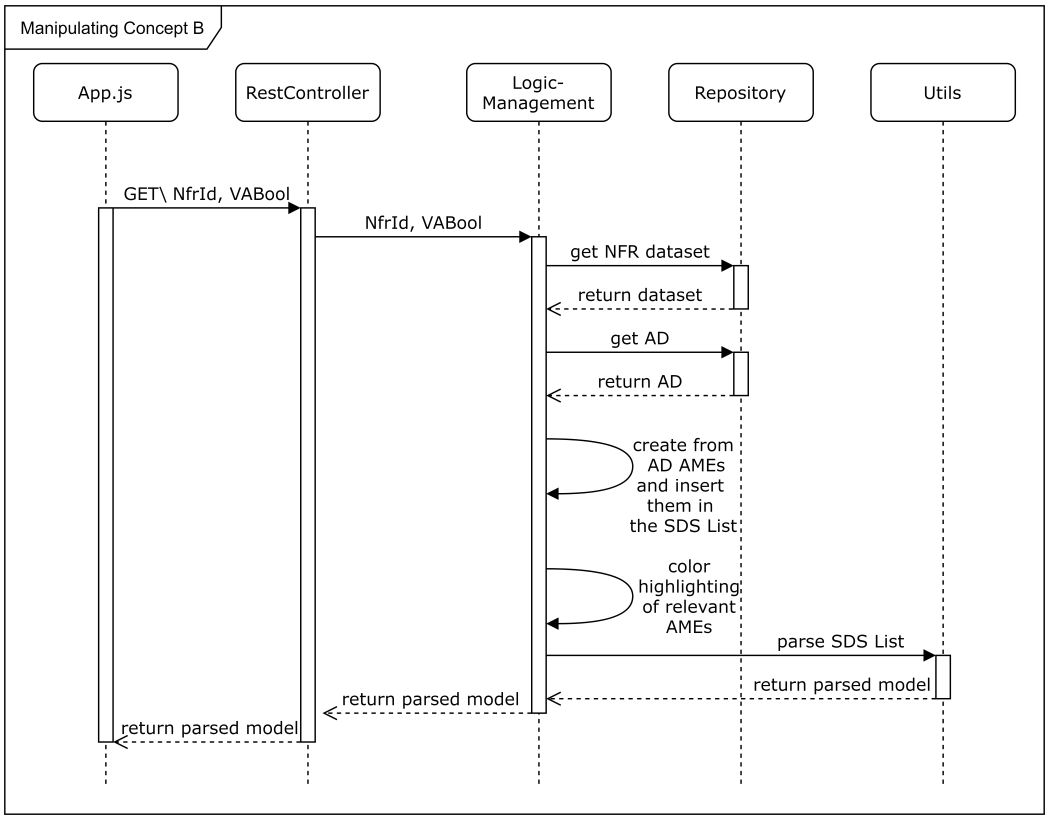


Figure 5.5: Communication with the visual approach B (chapter 4.3.2) in a sequence diagram

5.3 Conclusion

In this chapter, the designed software architecture was described from a static and dynamic perspective.

From a static perspective, the *Proof of Concept* was designed using a strict layer hierarchy and based on the *Model-View-Controller* architecture pattern. This leads to a clear separation of responsibilities within the individual components and a high degree of cohesion.

The following chapter describes how the software design was implemented. has been implemented.

6 Implementation

Contents

This chapter describes how the software architecture presented in Design 5 was implemented. As shown in Figure 5.2, the components have clearly defined layers and follow the *Model-View-Controller* pattern. First of all, chapter 6.1 explains the implementation basics that are necessary for understanding the implementation. Design decisions are also explained. The technical basics are then addressed in chapter 6.2. The application is then presented in chapter 6.3.

6.1 Implementation Foundation

Before the exact implementation is presented, a few technical basics are on which the implementation is based.

When researching how an *AD* can be modeled, which lays the data basis in the *Proof of Concept*, the Draw.io library[1] stood out. The Nasdanika Draw.io sounded promising for the implementation of the prototype in the Java *Backend*. The challenge is, that most visualization toolkits and programming libraries in Java are not inherently designed to be displayed in a .js *Frontend*, which limits their compatibility with JavaScript-based *Frontends* [Han+21] [IBS04]. Making it hard to find tools that can be effectively used to realize the *POC*.

6.1.1 Nasdanika.org

The Nasdanika Draw.io offers two maven modules to edit Draw.io diagrams. It contains an API and a model and is based on the Nasdanika Graph[2] data structure.

[1] https://docs.nasdanika.org/core/drawio/index.html
[2] https://docs.nasdanika.org/core/graph/index.html

However, the documentation[3] of the library is poor and therefore using the API to manipulate .drawio files via the library is almost impossible. It also offers no way to display the graph created and manipulated in the *Backend* in the *Frontend*. A complex parser would have had to be developed in order to visualize the Nasdanika Graph Model.

After Nasdanika Draw.io turned out to be not fitting for the *POC*, the mxGraphModel, on which Draw.io is based on, was used.

6.1.2 MxGraph

The mxGraph Library is an open-source JavaScript and Java Swing library used for graph visualization and information management. It is used to create and edit graphs in web applications [24]. Often it is used to model graphical diagrams such as UML, flowcharts, networks, BPMN diagrams and similar visualizations [24]. For the prototype, it is required that only UML-like diagrams are used. The use of MxGraph allows for a common integrated representation of all kinds of documents in a concise manner, which is simultaneously formal, visualizable, and efficiently processable [Ebe08]. The main use of the two libraries is either only in web applications or only in desktop applications [24]. The library was chosen as the data structure because it was implemented with Jgraph for the *Frontend* and JgraphX for the *Backend*. It is also used by Draw.io[4], which was used to design the underlying *AMs* and *AMEs*. The structure is briefly explained in the following using an example.

Structure

The mxGraph Library has also been developed in the MVC architecture, with the mxGraphModel representing the model.

Some parameters have been omitted for the sake of clarity. As you can see, each mxCell of the mxGraphModel has an ID, a parent, a style, a value and a mxGeometry. The structure is tree-like similar to the structure of an XML.

6.1.3 XML

The XML format is a standard procedure developed by the W3C (World Wide Web Consortium) for exchanging data between incompatible systems[5]. XML serves as a data model for databases of XML documents and applications beyond traditional data models, supporting hierarchical and recursive structures [Pok07]. It provides a flexible and extensible data model for structured data, making it suitable for a wide range of applications [Chi+05].

[3] https://javadoc.io/doc/org.nasdanika.core/drawio/latest/org.nasdanika.drawio/module-summary.html
[4] https://www.drawio.com/
[5] https://www.w3schools.com/xml/xml_whatis.asp

```
<mxGraphModel>
  <root>
    <mxCell id="0" />
    <mxCell id="1" parent="0" />
    <mxCell id="9T3OI0znlgV2pwNcInLf-5"
        style="edgeStyle=orthogonalEdgeStyle" parent="1"
        source="9T3OI0znlgV2pwNcInLf-3"
        target="9T3OI0znlgV2pwNcInLf-1" edge="1">
      <mxGeometry relative="1" as="geometry" />
    </mxCell>
    <mxCell id="9T3OI0znlgV2pwNcInLf-3" value="&lt;p
        style=&quot;text-align: center ; margin: 6px 0px
        0px&quot;&gt;&lt;br&gt;&lt;/p&gt;&lt;p
        style=&quot;text-align: center ; margin: 6px 0px
        0px&quot;&gt;LoginPage&lt;/p&gt;"
        style="align=left;overflow=fill;html=1;dropTarget=0;"
        parent="1" vertex="1">
      <mxGeometry x="40" y="40" width="130" height="60"
          as="geometry" />
    </mxCell>
  </root>
</mxGraphModel>
```

Figure 6.1: Reduced mxGraphModel

Structure

Like the mxGraphModel, the structure is tree-like. XML allows many modeling options to be represented in the data structure thanks to its many string, numeric, date, Boolean, binary and URI data types. It is also readable for humans and machines. However, the data records can become very large very quickly due to the structure. It is one of the supported data formats according to the REST principles and is therefore used in this *POC* for communication between the *Frontend* and *Backend*. Because the two structures are so similar, XML is also required as the data type for storing the mxGraphModel in the *POC*.

6.1.4 REST

Representational State Transfer, or REST for short, is an architectural paradigm, according to which web services can be implemented. Web services that are implemented according to the REST approach are also called RESTful Web Services [III12]. RESTful APIs define a series of resources distinguished by uniform resource identifiers (URIs) and can be manipulated with actions based on the semantics of HTTP verbs such as GET, POST, DELETE, and PUT [Arc21]. In the *POC*, a GET request is used to request data from the *Backend* to the *Frontend*. It sends parameters to the *Backend* via the URI and receives a response back using the data sent and the *Backend* logic. RESTful services

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="note">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="to" type="xs:string"/>
                <xs:element name="from" type="xs:string"/>
                <xs:element name="heading" type="xs:string"/>
                <xs:element name="body" type="xs:string"/>
            </xs:sequence>
            <xs:attribute name="date" type="xs:date"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Figure 6.2: XML Example

can bring together discrete data from different services using XML-based messaging, enabling the creation of meaningful data sets [SGL07].

6.1.5 Java

Java[6] is both a programming language and a computing platform that supports object-oriented programming. Java is used for low latency applications [Van+06]. It is platform-independent through research that has been focused on overcoming the limitations of Java for real-time technology, indicating its widespread use and continuous development [Ter12].

Java Spring Spring Framework[7] is a powerful and versatile framework for the development of Java applications. It offers comprehensive infrastructure support for the development of robust and maintainable Java-based software. It has many powerful features, such as Spring Boot. This allows you to start a Java Spring application with minimal configuration effort.

Maven With Java, Maven is used as a build management and project management tool. It accumulates all versions of published libraries, allowing applications to choose any version for dependency [DPM16] making it easy to start the built app on any platform regardless of the needed dependencies. Maven has a standardized structure that makes it easier to set up and understand new projects. The project can also be automatically compiled, tested, packaged and executed [DPM16].

The combination of Spring and maven simplifies setting up the application. Since it is the programming language in which I have the most experience, I chose Java Spring to

[6] https://www.java.com/
[7] https://spring.io/projects/spring-boot

implement my *Backend*, described in Figure 5.3. The version used for the *POC* is Java 21, Spring Boot 3.2.4 and Maven 3.9.0.

6.1.6 React

React[8] is an open-source JavaScript library. It is used to create user interfaces (UIs), particularly for single-page applications (SPAs). React enables developers to create reusable UI components that control the state and behavior of an application. React is user-friendly and can be easily learned by developers with basic knowledge of JavaScript, making it accessible for a wide range of developers [Kan+24]. The reason is to minimize time spend when learning a new language. The version used for the *POC* is React 18.2.0.

6.2 Technical Entities

At this point, four basic technical foundations are clarified to better understand the implementation. The AmeModel, AmModel, SdsModel and ModelWithFlag objects are each instantiated by their factory according to the factory design pattern[9].

6.2.1 AmeModel

As described in chapter 4.1, the AmeModel is a subset of an AmModel. The AmeModel has the parameters id, value, geometry and style. These parameters can be traced back to the structure of the mxGraphModel, explained in chapter 6.1.2. To manipulate the model there exists is a setter and getter for each parameter. If a new instance of AmeModel is created, each parameter must also be provided.

AmeModelFactory The AmModelFactory receives a mxCell, which is shown in Figure 6.1, in the constructor and instantiates a new AmeModel with the parameters of the mxCell.

6.2.2 AmModel

The AmModel consists of a set of AmeModels, as I explained in chapter 4.1. Therefore, the AmModel only contains a list of AmeModels and a method that returns the list.

AmeModelFactory To instantiate an AmModel, a mxCell must be passed to the factory. The factory now automatically creates and fills the AmeModels that make up the AmModel from the parameters it contains.

[8] https://react.dev/
[9] https://www.baeldung.com/java-factory-pattern

6.2.3 SdsModel

The SdsModel is the interface between AmModel and AmeModel. This means that an SdsModel can be an AmeModel or an AmModel. As will be explained in chapter 6.3, the SdsModel is mainly used as a list.

SdsFactory The factory expects a mxCell as a parameter for the createModel() method.

6.2.4 ModelWithFlag

The ModelWithFlag object has two parameters: a mxGraphModel and a Boolean isAmModel. This object is used as a list to be able to distinguish after filtering whether an AmModel must be instantiated for the mxGraphModel or one or more AmeModels. The process is explained in more detail later in chapter 6.3.5 and chapter 6.3.3.

ModelWithFlagFactory The factory expects a mxGraphModel and a Boolean as input to create a new ModelWithFlag object.

6.3 Application

This chapter covers the implementation of the application. The explanation roughly follows the process flow shown in figures 5.4 and 5.5. First, the Frontend is explained and then the Controller. Afterward, the Logic-Management and the Repository are explained in more detail. Finally, the Util classes are discussed.

6.3.1 Frontend

The *Frontend* was implemented using the *App.js* class. It forms the interaction with the user and is the top layer of the layered architecture. It implements the REST-API for communication with the RestController. A GET request is made that contains the parameters shown in table 6.1 A table filled with *NFRs* is displayed for the user. If the user

Type	**Name**	**Description**
Long	nfrId	Identifier of the NFR
Boolean	vaBool	Boolean that toggles whether Visual Approach A or B (chapter 4.3.2) is being used for the request

Table 6.1: REST-API parameters

clicks on an *NFR*, the request is sent to the RestController with the parameters shown in table 6.1. After the App.js has received a *text/xml* response from the RestController, it must be parsed again from XML to a mxGraphModel. This is done in the component *GraphContainer.js*.

However, there occurs a problem that is going to be explained in the evaluation, chapter 7. The mxGraphModel must then be parsed into an mxGraph again and this will then be displayed to the user in the container.

6.3.2 RestController

The next lower layer is the RestController, which forms the Controller of the *MVC* pattern. The GET-request is received here. It first checks whether the requested Nfr element from the Repository is empty. If it is, a *ResponseEntity.badRequest()* is sent as a response to the App.js. If not, the *SdsModelBuilder.java* is injected with the Repository parameters and the *sdsModelBuilder.builder(nfrId, vaBool)* is called.

The repository had to be initiated in the RestController because the injection dependency required by maven did not allow the repository to be initiated in *SdsModelBuilder*. injection dependency

After the LogicManagement has transferred the parsed *String xmlFile* to the controller, this is returned to the App.js via the REST-API with *ResponseEntity.ok(xmlFile)*.

6.3.3 Logic-Management

The *SdsModelBuilder.java* is the most complex class of the backend. This is where the repository is accessed and the *ModelParser.java* and *MxGraphModelFilter.java* utility functions are used. Calling the *builder(Long nfrId, Boolean vaBool)* function starts the

```
Nfr Case3 = new Nfr();
Case3.setId(3L);
Case3.setCellIds(List.of("dLUkl0QVlCv7WK-rCqbg-31",
"9T3OI0znlgV2pwNcInLf-6"));
Case3.setFileNames(List.of("Example1.xml",
"Example2.xml"));
nfrService.saveNfr(Case3);
```

Figure 6.3: Nfr object example

Logic-Management process as described in chapter 6.3.3. The corresponding Nfr object is retrieved from the repository with the help of *nfrId*. An example of an Nfr object is shown in Figure 6.3. The associated *AMs* are now read from this. These have the form of a .xml file, as can be seen in 6.1. The *AMs* are then passed to the parser using the method *parseXmlFilesToMxGraphModels(amList)*. The parser returns a list with the type mxGraphModel. Now the case distinction between Visual Approach A and B takes place, described in 4.3.2.

A: If A was selected, the *mxGraphModelFilter.java* is called again. This now filters out the *AM* and *AME* based on the IDs that were not transferred from the Nfr object. These are then deleted. The list returned to the SdsModelBuilder now has the type ModelWithFlag. As explained in chapter 6.2, *AMs* and *AMEs* are now created and saved in an SdsModel list. As is displayed in Figure 6.4

B: If B is selected, all objects from the mxGraphModels list are instantiated as AMEs. All AMEs whose IDs are stored in the Nfr object are then colored red. As one can see in Figure 6.5

The SdsModel list is then passed to the *ModelParser.java* and this parses the *AMs* and *AMEs* into a *String xmlFile*. This is then returned to the controller.

6.3.4 Repository

As you can see from the graphic 5.1.2, the repository consists of two parts: the NfrIdRepository and the AdRepository.

The NfrIdRepository stores *NFRs* as objects whose parameters are then accessed by Logic-Management. The Nfr repository was designed according to the repository pattern[10]. As table 6.2 shows, it stores the attributes of an *NFR* that enable mapping. How a new Nfr object is then instantiated is shown in 6.3.

Type	**Name**	**Description**
Long	id	Identifier of the Nfr object.
List<String>	cellIds	List of mxCell IDs that belong to the given *NFR*. This is needed for creating *AMs* and *AMEs* in the Logic-Management.
List<String>	fileNames	List of file names, where each file represents an *AM* and the complete list forms an *AM*.

Table 6.2: Repository parameters

The AdRepository consists of several .xml files where the UML-like mxGraphModels are stored. The files are managed via *FilesStorageUtil.java*.

[10]https://java-design-patterns.com/patterns/repository/

6.3.5 Utils

The utilities shown in 5.1.2 include the two classes *ModelParser* and *MxGraphModelFilter*.

The ModelParser contains three functions: The *parseXmlFilesToMxGraphModels()* parses the .xml files from the given Nfr object into a mxGraphModel. The *parseSdsToMxGraphModel()* receives a list consisting of SdsModels and parses them into an mxGraphModel. Finally, there is the *mxGraphModelToXml()* method, which transforms an mxGraphModel using the mxCodec class provided by mxGraph.

The *MxGraphModelFilter* only has the *FilterModel()* method. This receives a parsed *AM* list and the cellIds are mapped to the *NFR* according to the procedure described in chapter 6.3.3. It now filters out all unneeded mxCells and passes a ModelWithFlag list to the Logic-Management.

6.4 Conclusion

The modular architecture designed in chapter 5 was implemented by encapsulating the Logic-Management functionality in individual components. A mapping between an *NFR* and an *AD*, with its components, was created and this mapping can be traced.

```
for (ModelWithFlag mxGraphModel : modelList) {
            mxGraph graph = new mxGraph();
            graph.setModel(mxGraphModel.getModel());
            if (mxGraphModel.isAmModel()) {
                // Retrieve all cells from the graph
                Object[] amCells =
                    graph.getChildCells(graph.getDefaultParent(),
                    true, true);
                logger.info("amCells: " + amCells.toString());
                // Create a new parent node
                mxCell newParent = new mxCell();
                newParent.setId("newParent");

                // Add amCells to the new parent node
                for (Object cell : amCells) {
                    if (cell instanceof mxCell) {
                        mxCell mxCell = (mxCell) cell;
                        newParent.insert(mxCell);
                    }
                }
                    // Create a new mxGraph and set the model
                    models.add(AmModelFactory.createModel(newParent));
            } else if(!mxGraphModel.isAmModel()) {
                // Retrieve all cells from the graph
                Object[] ameCells =
                    graph.getChildCells(graph.getDefaultParent(),
                    true, true);
                for (Object cell : ameCells) {
                    if (cell instanceof mxCell) {
                        mxCell mxCell = (mxCell) cell;
                        logger.info("Creating AME Model Instance
                            for cell: " + mxCell.getId());
                        models.add(AmeModelFactory.createModel(mxCell));
                    }
                }
            }
```

Figure 6.4: Remove all cells that are irrelevant

```
// alle cells werden als AME dargestellt
    for (mxGraphModel model : parsedAmList) {
        // Create a new mxGraph and set the model
        mxGraph graph = new mxGraph();
        graph.setModel(model);

        // Retrieve all cells from the graph
        Object[] cells =
           graph.getChildCells(graph.getDefaultParent(),
        true, true);

        // Iterate over each cell
        for (Object cell : cells) {
            if (cell instanceof mxCell) {
                mxCell mxCell = (mxCell) cell;
                // Create an AME object for each cell
                models.add(AmeModelFactory.createModel(mxCell));
                }
            }
        }

        colorHighlightModels(models, nfrObject);
```

Figure 6.5: Filtering *AME* and coloring them

7 Evaluation

Contents

In Chapter 7.1, a problem within the *POC* is first identified and then the functionality of the *POC* is explained. In Chapter 7.2, the results of the two manipulation concepts are presented, as well as the five mapping cases that were developed in the process model 4.3. Subsequently, the results of this thesis are discussed.

7.1 Proof of Concept

As already explained in chapter 6.1, it is a challenge to find a library for creating and manipulating diagrams that offer libraries for JavaScript and Java. This is the prerequisite for ensuring the compatibility of the graphs in *Frontend* and *Backend*. For this reason and due to the good documentation[1], the mxGraph library was used to implement the *POC*.

The from the *Backend* logic manipulated models must be parsed in graphs and these must be transferred from the *Backend* to the *Frontend*. Since the REST API does not allow the graph to be sent directly, but requires a data type like XML, JSON or similar data types as a response, this must be parsed beforehand. However, mxGraph only supports the XML data type with its mxCodec class. As XML, as you can see in 7.1, the graph is passed to the *Frontend* in the correct structure. To display the graph again, it must be decoded again with the mxCodec class. However, an error occurs here if you want to parse from XML to the mxGraph datatype with the provided method you can see in Figure 7.2 in line 6. This piece of code is from the *GraphContainer.js*, which was mentioned in chapter 6.3.1 before. If you look at Figure 7.3 and compare the output of the console at point A with the XML passed from the *Backend*, the XML file passed is the same. This shows that the code works without any problems up to this point.

If you then look at Figure 7.4, which shows the output of line 7 of the code 7.2, you will notice that the mxGraphModel no longer has any structure. There are no other children under the mxCell with the id: “1”. Style and value are undefined. If you compare these

[1] https://jgraph.github.io/mxgraph/

```
<mxGraphModel>
    <root>
        <mxCell id="0"/>
        <mxCell id="1" parent="0"/>
        <mxCell id="dLUkl0QVlCv7WK-rCqbg-13" parent="1"
           style="align=left;overflow=fill;html=1;dropTarget=0;"
           value="&lt;p style=&quot;text-align: center ; margin: 6px
           0px 0px&quot;&gt;&lt;br&gt;&lt;/p&gt;&lt;p
           style=&quot;text-align: center ; margin: 6px 0px
           0px&quot;&gt;Frontend&lt;/p&gt;">
            <mxGeometry as="geometry" height="60.0" width="130.0"
               x="40.0" y="40.0"/>
        </mxCell>
    </root>
</mxGraphModel>
```

Figure 7.1: Backend response

```
    const graph = new mxGraph(containerRef.current);
    const xmlDocument = mxUtils.parseXml(mxGraphModel);
    const codec = new mxCodec(xmlDocument);
    const node = xmlDocument.documentElement;
    console.log('XML Document element:', node);             //A
    codec.decode(node, graph.getModel());
    console.log('Graph model decoded:', graph.getModel());  //B
```

Figure 7.2: XML to mxGraph parser

with the parameters of a mxCell listed in chapter 6.1.2, you will notice that mxGeometry is missing. This is a fatal bug that has also been discussed frequently on GitHub[2][3] and Stackoverflow[4][5]. As the mxGraph library has been deprecated[6] since 09.11.2020, no solution will be implemented for this problem. Consequently, it is unfortunately not possible to display the graph in the *Frontend* with the *POC*, as the *decode()* method belongs to the library and therefore cannot be fixed. A new parser for decoding and encoding could be implemented next. This idea is explained further in chapter 8.2.

Nevertheless, the *POC* delivered a good result, because the mapping between an Nfr object, as described in 6.3.3, and a sub-graph was successfully implemented. It follows that traceability has also been achieved. Which will be shown in chapter 7.2. The

[2]https://github.com/jgraph/mxgraph/issues/165
[3]https://github.com/jgraph/mxgraph/issues/301
[4]https://stackoverflow.com/questions/62824431/mxcodec-doesnt-decode-xml-correctly
[5]https://stackoverflow.com/questions/46872779/mxgraph-codec-decode-not-adding-cells-to-the-actual-graph
[6]https://github.com/jgraph/mxgraph

```
innerHTML: '<root><mxCell id="0"/><mxCell id="1"
   parent="0"/><mxCell id="dLUkl0QVlCv7WK-rCqbg-13" parent="1"
   style="align=left;overflow=fill;html=1;dropTarget=0;"
   value="&lt;p style=&quot;text-align: center ; margin: 6px 0px
   0px&quot;&gt;&lt;br&gt;&lt;/p&gt;&lt;p
   style=&quot;text-align: center ; margin: 6px 0px
   0px&quot;&gt;Frontend&lt;/p&gt;"><mxGeometry as="geometry"
   height="60.0" width="130.0" x="40.0"
   y="40.0"/></mxCell></root>'
```

Figure 7.3: Console output at position A

```
root: Object { value: undefined, id: "0", mxObjectId: "mxCell#4"}
    children: Array [ …{} ]
        0: Object { value: undefined, id: "1", mxObjectId:
           "mxCell#5"}
        geometry: undefined
        id: "1"
        mxObjectId: "mxCell#5"
        parent: Object { value: undefined, id: "0", mxObjectId:
           "mxCell#4"}
        style: undefined
        value: undefined
```

Figure 7.4: Console output at position B

parameters can be changed via the underlying data models, the *AM* and the *AME*, and thus the complete or partial graphs can be manipulated. For example, you can change the color as shown in the *POC* in chapter 6.3.3 and in the following chapter. A user can select an *NFR* and depending on the Manipulation Concept used, the mxGraphModel is returned. This is shown in more detail in the following chapter.

7.2 Manipulation Concept

This chapter shows that the mapping works in the *POC*. A sub-graph that forms an *SDS* can be traced via an *NFR*.

For display reasons, as the mxGraphModel can quickly become very large, as already explained in chapter 6.1.3, the results are displayed as graphs. Due to the non-functioning decoder, as described in chapter 7.1, reconstructed graphs will be shown. To prove the authenticity of the graphs, the mxGraphModel from Figure 7.1 can be compared with Case 2 with Manipulation Concept A.

As shown in Figure 6.3, an *Example1.xml* and an *Example2.xml* were created to display the use cases. The two *AMs* do not claim to be correct in terms of functionality and are

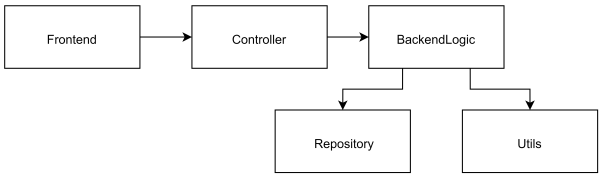


Figure 7.5: Example1.xml

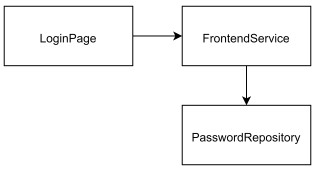


Figure 7.6: Example2.xml

Figure 7.7: The given AD for the showcase

for demonstration purposes only.

7.2.1 Delete all non-relevant model elements

In this manipulation concept, all irrelevant *AMEs* are removed from the *AD*. The process is shown in 5.4. An explanation of the implementation was given in chapter 6.3. Only the *AMEs* that fulfill the *NFR* are displayed to the user.

Case 1

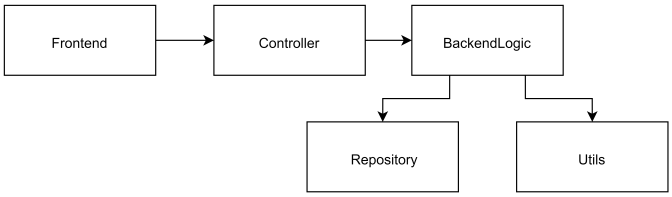


Figure 7.8: One *NFR* refers to one *AM*

In this case, a complete *AM* was the solution for the given requirement. The second *AM*, which is part of the *AD*, was filtered out.

Case 2

Figure 7.9: One *NFR* refers to one *AME*

The solution for this *NFR* lies in only one *AME*, which is why all other *AMEs* belonging to the *AD* are removed from the model.

Case 3

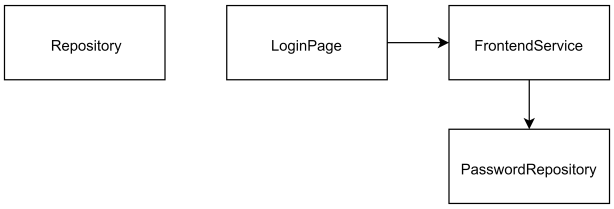


Figure 7.10: One *NFR* refers to several *AMs* or *AMEs*

Both *AMs* are required for this *NFR*, but only one is displayed in full and all *AMEs* except the repository are removed from the other.

Case 4

Several *NFRs* refer to at least one AME This case cannot be displayed directly with the *POC* created for this thesis, as only one *NFR* can be displayed at a time 4.3. However, this case with the condition that only the solution of one *NFR* can be displayed in parallel is the same as case 2. The display of several *NFRs* at the same time can be a future development for this *POC*. Similarly, in the case where two *NFRs* are dependent on at least one shared *AME*, backtracking must be implemented. These ideas are discussed in more detail in chapter 8.2.

Case 5

A *NFR* does not refer to any AME As there is no mapping between the *NFR* and a subset of the *AD*, the user is shown an empty graph.

7.2.2 Color highlighting of relevant model elements

In this manipulation concept, all relevant *AMEs* from the affected AMs are colored red. The user is only shown the *AMs* in which there is at least one *AME* that partially fulfills the *NFR*. This use case proves that the manipulation of *AMs* and *AMEs* works. Furthermore, it shows that the underlying concept based on the work of Alex Sabau [Sab24] has been implemented.

Case 1

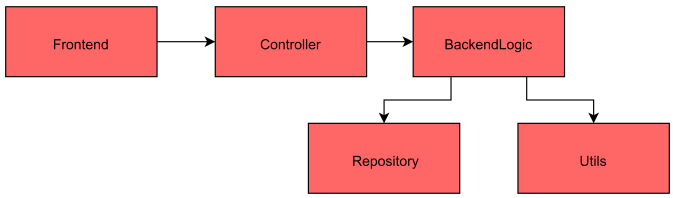


Figure 7.11: One *NFR* refers to one *AM*

As only one *AM* belongs to the solution of the *NFR*, only this is displayed. However, as the complete *AM* forms the *SDS*, all *AMEs* are highlighted.

Case 2

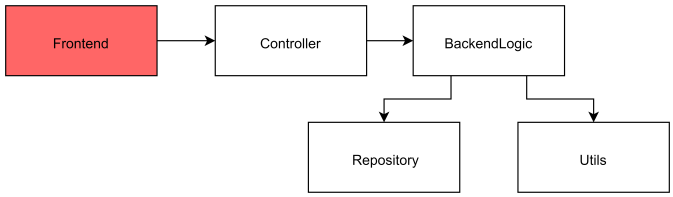


Figure 7.12: One *NFR* refers to one *AME*

An *AME* from the *AM Example1.xml* forms the solution of the *NFR*, therefore the complete *AM* is displayed so that the user gets the context to the relevant *AME* and thus has a better classification of how the context around the *AME* is. Only the *AME* is highlighted.

Case 3

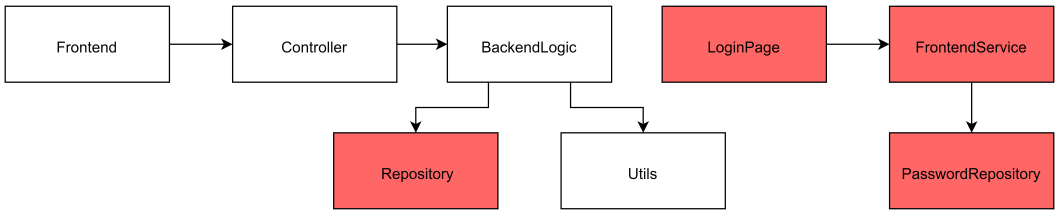


Figure 7.13: One *NFR* refers to several *AMs* or *AMEs*

AMEs are required from both *AMs* to represent the *SDS* for the *NFR*. Consequently, both AMs are displayed, but only the relevant *AMEs* are highlighted in color.

Case 4

Several *NFRs* refer to at least one AME According to the reasoning in chapter 7.2.1, exactly the same argument applies to this case.

Case 5

A *NFR* does not refer to any AME Due to the prerequisite that the sub-graph of an AD is only displayed if at least one *AME* forms the solution for the *NFR*, the same applies as in chapter 7.2.1 that no graph is displayed.

7.3 Discussion

Chapter 7.3.1 discusses whether the POC has implemented the process presented in the Process Model chapter 4.3 and whether the Concept Model developed in chapter 4 has been successfully implemented. Chapter 7.3.2 then argues which manipulation concept is better suited to the real application and makes a recommendation.

7.3.1 Proof of Concept

Despite the bug described in chapter 7.1, the *POC* has implemented the mapping. This was achieved by mapping the nfrId from the *Frontend* with the Nfr object id, as explained in chapter 6.3. The Nfr object described in Figure 6.3 has all the parameters required to create the *AMs* and *AMEs* that form an *SDS*. If the Nfr object is adapted, the *SDS* can also be changed. For each *NFR* that has been mapped to an Nfr object, tracing to the *SDS* is possible.

The tracing is implemented in the Management-Logic. The sequence of the process can be seen in the figures 5.4 and 5.5. The implementation of the complete *POC* was also explained in detail in chapter 6. The figures 6.1 and 7.1 prove that the implemented logic is correct, as the structure is the same in both cases. In figure 6.1 a raw, unmanipulated mxGraphModel is shown and in figure 7.1 the structure after it has been manipulated and passed to the *Frontend*. The results from chapter 7.2 also showed the functionality in the real-life application. *SDSs* could be traced for cases 1-3 using both approaches. Case 4 cannot yet be covered by the current POC due to the requirement from chapter 4.3 that only one *NFR* can be traced at a time. This case would be a possible further development in the future. Case 5 is the trivial case, as no mapping exists and therefore no tracing is possible.

Consequently, the developed process model from chapter 4.3 was successfully implemented. The concept model developed in chapter 4.2 was realized with the *POC*.

7.3.2 Manipulation Concepts

The manipulation concepts were introduced in chapter 4.3.2 and implemented in chapter 7.2. This chapter discusses the pros and cons and comes to a conclusion as to which of the two concepts could be used in the future.

All in all, Manipulation Concept B "Color highlighting of relevant model elements" is the better concept. Concept A creates smaller graphs because fewer model elements are displayed. However, even with larger scaling, it becomes very confusing. The more individual *AMEs* from different *AMs* are contained in the *SDS*, the more individual model elements are displayed without context. This reduces the comprehensibility of large amounts of data.

Concept A, however, retains a high level of comprehensibility, as the complete AM is displayed to the user, giving them more context, which makes the solution of the NFR easier to understand. At the same time, it is still clear which *AMEs* are relevant for the *NFR* being searched for, despite the possibility of a larger number of *AMs* being passed to the user.

7.4 Conclusion

In this chapter, the *Proof of Concept* was tested using the first three cases and both manipulation concepts. Additionally, the raw input of a mxGraphModel was compared with the modeled mxGraphModel, which was transferred from the *Backend* to the *Frontend*. However, the results could be further improved by developing a functional parser. The discussion demonstrated that, despite the flawed mxGraph encoding method, the *POC* still fulfills the first research question (see section 1.2. The *POC* illustrates a possible implementation of the developed concept model. Since the concept model does not

significantly deviate from the fundamental concept model [Sab24], the basic principle of this *POC* can also be used for further research.

In addition to designing and developing the *POC*, the second research question was also addressed. Between the two applied manipulation concepts, the concept of color highlighting proved to be superior due to its clarity and the high comprehensibility provided by the additional context.

By combining these two approaches, one can counter the statement made in Chapter 2.2 that requirement tracing becomes difficult with increasing complexity due to various levels of abstraction. Regardless of how large the *AD* becomes, the complexity of tracing remains constant through clear mapping and graph reduction.

8 Conclusion

Contents

In this chapter, a summary of the work and its results is presented. Subsequently, possible improvements to the *POC* are proposed.

8.1 Summary

Mapping and tracing between *Security Design Solution* in *Architecture Description* is an innovative concept for mapping security requirements to software architectures. The advantage of integrating the security architecture with the system architecture is a promising approach to enhancing software security and making documentation more manageable. The *Proof of Concept* has provided initial experiences that can contribute to the further development of the underlying concept.

Similar to Enterprise Architect3.2.1, the concept can manage the design of software architectures and define requirements. *Proof of Concept* goes a step further by mapping the requirements directly to a *SDS*. This not only allows for the definition and management of requirements but also enables tracing them to their respective solutions. Original requirement tracing tools like Jira or Jama primarily perform passive tracing. They accompany the process and guide it, ensuring that developers do not deviate from the correct path.

To plan and implement an initial *POC*, an approach was found to map *NFR* and *SDS*. This mapping enables effective tracing, which will facilitate the creation and management of documentation in the future. In addition to the *POC*, two exploratory manipulation concepts were also developed and applied to the previously identified possible mapping cases. The color highlighting concept stood out as it reduces the loaded *AMEs* while still providing the user with context for the *SDS*, thus enhancing the comprehensibility of the implementation.

The *POC* forms a foundation for further research, improvements, and ideas. Suggestions for enhancements are presented in the next section.

8.2 Future Work

8.2.1 Implement a new XML decoder

Due to a bug in the mxCodec decoder, it is not possible to decode a mxGraph in the *Frontend* from an XML file. To effectively utilize the mxGraph library and ensure that the user receives the graph, it is necessary to implement a new XML decoder.

8.2.2 Implement Backtracking for an SDS

As described in Case 4 7.2.1, multiple *NFRs* can depend on a single *AME*. If a requirement changes, thereby altering the *SDS*, it is necessary to verify whether the affected *AME* is also part of the *SDS* for another *NFR*. If so, the compatibility between the changes and the *NFRs* must be considered, and a decision should be made if necessary. This prevents the unintentional invalidation of requirements when another requirement seeks to modify the *AME*.

8.2.3 Implementing a Database

If the tool is to be used in the long term, a database for persistent data storage will be required.

8.2.4 A possibility of Adding new AMs

The addition of new *AMs* or *AMEs* depends on the implementation of a database. If a database is implemented, it should be possible to send the mxGraphModel as XML to the *Backend* via a REST API, where it can then be managed in the database.

8.2.5 Mapping new NFR to the AD

Mapping newly created *NFRs* would also be an interesting feature. For this to work, however, the encoder must function correctly. Then, it should be possible to visualize the graph in the *Frontend* and mark the desired *AMEs* that belong to the *NFR*. The information about which mxCell IDs belong to which *NFR* ID can then be easily passed to the *Backend* and managed in the necessary database.

Bibliography

[24] *mxGraph*. https://jgraph.github.io/mxgraph/. access on: 29. April 2024. 2024 (cit. on p. 22).

[Ali+22] A. Ali et al. "Role of Non-functional Requirements in projects' success." In: 2022. DOI: 10.1109/ICoDT255437.2022.9787463. URL: https://www.scopus.com/inward/record.uri?%20%5C%5C%20eid=2-2.0-85133203295&doi=10.1109%2fICoDT25%20%5C%5C%205437.2022.9787463&partnerID=40&md5=%20%5C%5C%20fba60648e1185d74c5771edd732a988d (cit. on p. 4).

[Arc21] A. Arcuri. "Automated Black- And White-Box Testing of RESTful APIs with EvoMaster." In: *IEEE Software* 38.3 (2021). Cited by: 28, pp. 72–78. DOI: 10.1109/MS.2020.3013820 (cit. on p. 23).

[Che+10] Y. Chen et al. "A ten-year survey of software architecture." In: 2010, pp. 729–733. DOI: 10.1109/ICSESS.2010.5552456. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.%20%5C%5C%200-77957851965&doi=10.1109%2fICSESS.2010.5552456&%20%5C%5C%20partnerID=40&md5=d28dcdb07d174645771721498f374d1e (cit. on p. 3).

[Chi+05] K. Chiu et al. "A binary XML for scientific applications." In: vol. 2005. 2005, pp. 336–343. DOI: 10.1109/E-SCIENCE.2005.1 (cit. on p. 22).

[CMC15] P. Chavan, M. Murugan, and P. Chavan. "A review on software architecture styles with layered robotic software architecture." In: 2015, pp. 827–831. DOI: 10.1109/ICCUBEA.2015.165. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84943329051&doi=10.1109%2fICCUBEA.2015.165&partnerID=40&%20%5C%5C%20md5=b041dd3c1f41fee01030db0f2bf0891e (cit. on p. 3).

[DPM16] C. Désarmeaux, A. Pecatikov, and S. Mcintosh. "The dispersion of Build maintenance activity across Maven lifecycle phases." In: 2016, pp. 492–495. DOI: 10.1145/2901739.2903498 (cit. on p. 24).

[Ebe08] J. Ebert. "Metamodels taken seriously: The TGraph approach." In: 2008, p. 2. DOI: 10.1109/CSMR.2008.4493294 (cit. on p. 22).

[Han+21] D. Han et al. "NetV.js: A web-based library for high-efficiency visualization of large-scale graphs and networks." In: *Visual Informatics* 5.1 (2021). All Open Access, Gold Open Access, pp. 61–66. DOI: 10.1016/j.visinf.2021.01.002 (cit. on p. 21).

[IBS04] R. Ian Bull, C. Best, and M.-A. Storey. "Advanced widgets for eclipse." In: 2004, pp. 6–11. DOI: 10.1145/1066129.1066131 (cit. on p. 21).

[III12] W. III. "Service design patterns: fundamental design solutions for SOAP/WSDL and RESTful web services by Robert Daigneau." In: *ACM Sigsoft Software Engineering Notes* 37 (July 2012). DOI: 10.1145/2237796.2237821 (cit. on p. 23).

[Kan+24] M. Kanniga Parameshwari et al. "CRUD Application Using ReactJS Hooks." In: *EAI Endorsed Transactions on Internet of Things* 10 (2024). All Open Access, Gold Open Access, pp. 1–10. DOI: 10.4108/eetiot.5298 (cit. on p. 25).

[KK18] R. A. Khan and S. U. Khan. "A preliminary structure of software security assurance model." In: 2018, pp. 137–140. DOI: 10.1145/3196369.3196385. URL: https://www.scopus.com/inward/record.uri?eid=2%20%5C%5C%20-s2.0-85051552000&doi=10.1145%2f3196369.3196385%5C%5C%20&partnerID=40&md5=f162d2c9dbe17c88c175173a07d7a082 (cit. on p. 1).

[LL13] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken.* dpunkt.verlag
, 2013. ISBN: 978-3-89864-268-2 (cit. on pp. 13, 14, 16, 18).

[Mat17] R. Matulevičius. *Fundamentals of Secure System Modelling.* Aug. 2017. ISBN: 978-3-319-61716-9. DOI: 10.1007/978-3-319-61717-6 (cit. on p. 1).

[McG04] G. McGraw. "Software Security." In: *IEEE Security and Privacy* 2.2 (2004), pp. 80–83. DOI: 10.1109/MSECP.2004.1281254. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.%20%5C%5C%200-2342469291&doi=10.1109%2fMSECP.2004.%20%5C%5C%201281254&partnerID=40&md5=bc2b87142a3%20%5C%5C%200c142a20036f39aafb5c8 (cit. on p. 1).

[MRA15] A. Marques, F. Ramalho, and W. L. Andrade. "TRL - A Traceability Representation Language." In: vol. 13-17-April-2015. 2015, pp. 1358–1363. DOI: 10.1145/2695664.2695745. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84955505708&doi=10.1145%2f2695664.2695745&partner%20%5C%5C%20ID=40&md5=0b5a868990110a620083fff68bc781c0 (cit. on p. 4).

[MVV18] K. Mapingire, J. Van Deventer, and A. Van Der Merwe. "Competencies and skills of enterprise architects: A study in a South African telecommunications company." In: 2018, pp. 154–163. DOI: 10.1145/3278681.3278700. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85058631798&doi=10.1145%2f3278681.3278700&partnerID=40&md5=94343f4b49b011481c8b33d6aa802de1 (cit. on p. 5).

[MZ22] A. A. Madaki and W. M. N. W. Zainon. "A Review on Tools and Techniques for Visualizing Software Requirement Traceability." In: *Lecture Notes in Electrical Engineering* 829 LNEE (2022), pp. 39–44. DOI: 10.1007/978-981-16-8129-5_7. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85125263597&doi=10.1007%2f978-981-16-8129-5_7&partne%20%5C%5C%20rID=40&md5=f8c110f95b9cd%20%5C%5C%20c891eb8f48ef6c14ff7 (cit. on p. 4).

[PB11] P. Petrov and U. Buy. "A systemic methodology for software architecture analysis and design." In: 2011, pp. 196–200. DOI: 10.1109/ITNG.2011.41. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-80051510583&doi=10.1109%2fITNG.2011.41&partnerID=40&md5=74ab53%5C%5C%20325baca105d7bbf834516f8906 (cit. on p. 3).

[Pok07] J. Pokorný. "XML databases: Principles and usage." In: 2007, pp. 37–38 (cit. on p. 22).

[Sab24] A. R. Sabau. "A Guided Modeling Approach for Secure System Design." In: *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*. To appear. Los Alamitos, CA, USA: IEEE Computer Society, June 2024 (cit. on pp. 1, 2, 7–9, 38, 41).

[SGL07] A. P. Sheth, K. Gomadam, and J. Lathem. "SA-REST: Semantically interoperable and easier-to-use services and mashups." In: *IEEE Internet Computing* 11.6 (2007). Cited by: 138; All Open Access, Green Open Access, pp. 91–94. DOI: 10.1109/MIC.2007.133 (cit. on p. 24).

[SL07] V. Sadana and X. F. Liu. "Analysis of conflicts among non-functional requirements using integrated analysis of functional and non-functional requirements." In: vol. 1. 2007, pp. 215–218. DOI: 10.1109/COMPSAC.2007.73. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-37349059904&doi=10.1109%2fCOMPSAC.2007.73&partner%5C%5C%20ID=40&md5=cc73f%20%5C%5C%2085c8ded65aa623b3bd619aacbe0 (cit. on p. 4).

[Tam+11] T. Tamm et al. "How does enterprise architecture add value to organisations?" In: *Communications of the Association for Information Systems* 28.1 (2011). All Open Access, Bronze Open Access, pp. 141–168. DOI: 10.17705/1cais.02810. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-79953002068&doi=10.17705%2f1cais.02810&partnerID=40&md5=c15acce960501e52605ec5796a9f97ca (cit. on p. 5).

[Ter12] M. Teresa Higuera-Toledano. "About 15 years of real-time java." In: 2012, pp. 34–43. DOI: 10.1145/2388936.2388943 (cit. on p. 24).

[Tum+20] K. Tuma et al. "Automating the early detection of security design flaws." In: 2020, pp. 332–342. DOI: 10.1145/3365438.3410954. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.%20%5C%5C%200-85097000085&doi=10.1145%2f3365438.%20%5C%5C%203410954&

partnerID=40&md5=49417de5323%20%5C%5C%20a711b8adffabebae35723 (cit. on p. 1).

[Van+06] B. Van Den Bossche et al. "J2EE-based middleware for low latency service enabling platforms." In: 2006. DOI: 10.1109/GLOCOM.2006.181 (cit. on p. 24).

[VT16] H. van Vliet and A. Tang. "Decision making in software architecture." In: *Journal of Systems and Software* 117 (2016), pp. 638–644. DOI: 10.1016/j.jss.2016.01.017. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84956623197&doi=10.1016%2fj.jss.2016.01.017&partnerID=40%5C%5C%20&md5=8a06ea5ab015ff3e0576015892d0d3c9 (cit. on p. 3).

[ZK11] C. Ziftci and I. Krueger. "Tracing requirements to tests with high precision and recall." In: 2011, pp. 472–475. DOI: 10.1109/ASE.2011.6100102. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84855427383&doi=10.1109%2fASE.2011.6100102&partn%20%5C%5C%20erID=40&md5=64ed%20%5C%5C%20e1baef5d1506377d02e63126d62b (cit. on p. 4).

Glossary

AD Architecture Description

AM Architectural Model

AME Architectural Modelling Element

BE Backend

DAL Data Access Layer

FE Frontend

MVC Model-View-Controller

NFR Non-functional requirement

POC Proof of Concept

SDS Security Design Solution