Antwerp Management School

On the convergence of Clean Architecture with the Normalized Systems Theorems

A Design Science approach of stability, evolvability and modularity on a C# software artifact.

Author:
Gerco Koks

Supervisor:
Prof. Dr. Ing. Hans Mulder

Promotor:
Dr. ir. Geert Haerens

Co-Promotor:
Frans Verstreken, MSc

A thesis submitted in fulfillment of the requirements
for the degree of Master of Enterprise IT Architecture

April 30, 2023

"The essence of science is that it is always
willing to abandon a given idea for a better one."
— *Albert Einstein* —

"Life is a series of natural and spontaneous changes.
Don't resist them; that only creates sorrow. Let reality
be reality. Let things flow naturally forward in
whatever way they like."
— *Lao Tzu* —

"The secret of change is to focus all of your energy
not on fighting the old, but on building the new."
— *Socrates* —

"Change is the only constant in life."
— *Heraclitus* —

Information

Title:	On the convergence of Clean Architecture with the Normalized Systems Theorems
Subtitle:	A Design Science approach of stability, evolvability and modularity on a C# software artifact.
Submission date:	15 May 2022
Language:	US English
Reference Style:	APA 7th Edition
Copyright	© 2023 G.C. Koks
License:	©①② This work is licensed under a CC-BY-SA 4.0 license.

Repositories

Thesis:	https://github.com/liquidvisions/master-thesis/
artifacts:	https://github.com/liquidvisions/liquidvisions.pantharhei/

Author

Name:	Gerco Koks
Email:	gerco.koks@outlook.com
LinkedIn:	https://www.linkedin.com/in/gercokoks/

Supervisor

Name:	Prof. Dr. Ing. Hans Mulder
Email:	Hans.Mulder@ams.ac.be
LinkedIn:	https://www.linkedin.com/in/jbfmulder/

Promotor

Name:	Dr. ir. Geert Haerens
Email:	geert.haerens@uantwerpen.be
LinkedIn:	https://www.linkedin.com/in/geerthaerens/

Co-Promotor

Name:	Frans Verstreken, MSc
Email:	frans.verstreken@nsx.normalizedsystems.org
LinkedIn:	https://www.linkedin.com/in/fransverstreken/

On the convergence of Clean Architecture with the Normalized Systems Theorems

A Design Science approach of stability, evolvability and modularity on a C# software artifact.

Gerco Koks

Abstract

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

— Gerco

Table of Contents

Appendix D. Component Cohesion Principles 67

1. Introduction

After my bachelor's degree in 2009, I started to work as a junior software engineer. I was confident I was willing to accept any technical challenge as my experience up until that point led me to believe that creating some software was not that difficult. I could not have been more wrong. I quickly discovered it was a real challenge to apply new requirements to existing pieces of (legacy) software or explain my craftings to the more mature engineers. The craftsmanship of software engineering was enormously challenging to me.

Determined to overcome the difficulties, I started reading and investigating and immediately recognized the Law of Increasing Complexity of Lehman (1980), where he explained the balance between the forces driving new requirements and those that slow down progress. Other pioneers in software have recognized these challenges in engineering also.

D. McIlroy (1968) proposed a vision for systematically reusing software building blocks that should lead to more reuse. D. McIlroy (1968) stated, "The real hero of programming is the one who writes negative code," i.e., when a change in a program source makes the number of lines of code decrease ('negative' code), while its overall quality, readability or speed improves (Wikipedia, 2023). Perhaps very early concepts of modular software constructs?

Dijkstra (1968) argued against using unstructured control flow in programming and advocated for using structured programming constructs to improve the clarity and maintainability of the source code. In addition, Dijkstra advocated structured programming techniques that improved the modularity and evolvability of software artifacts.

Parnas (1972) continued with the principle of information hiding. Parnas stated that design decisions used multiple times by a software artifact should be modularized to reduce complexity.

Over the years, I got introduced to various software design principles and philosophies like Clean Architecture (CA), increasing my knowledge and craftsmanship. My career moved more toward architecture and product management. Nevertheless, I have always retained my passion for Software Engineering.

My obsession got re-ignited during the Master's degree introduction days at the Priory of Corsendock. Jan Verelst introduced me to Normalized Systems (NS), and I was intrigued by software stability and evolvability. It was fascinating to learn that there is now empirical scientific evidence for a quest I have been on for almost a decade.

NS Had to be the topic of my research. I was curious to compare what I knew (CA) with what science offered (NS). In early investigations, I found overlapping characteristics. Nevertheless, there were also a couple of differences. Could these design approaches be used in conjunction with each other?

Java SE has primarily been used for case studies in order to develop the Normalized Systems Theory (De Bruyn et al., 2018; Oorts et al., 2014). Although sufficient in Java, I was pleased to read that both software design approaches have formulated modular structures independent of any programming technology (Mannaert & Verelst, 2009; Robert C. Martin, 2018). So I could use my favorite programming language C#, to create a software artifact that supported my research.

Based on early investigations, I instinctively found that many applications of CA are a specialization of the NS Theorems. Consequently, I hypothesized that CA and NS could be used to achieve a modular, evolvable, and stable software artifact.
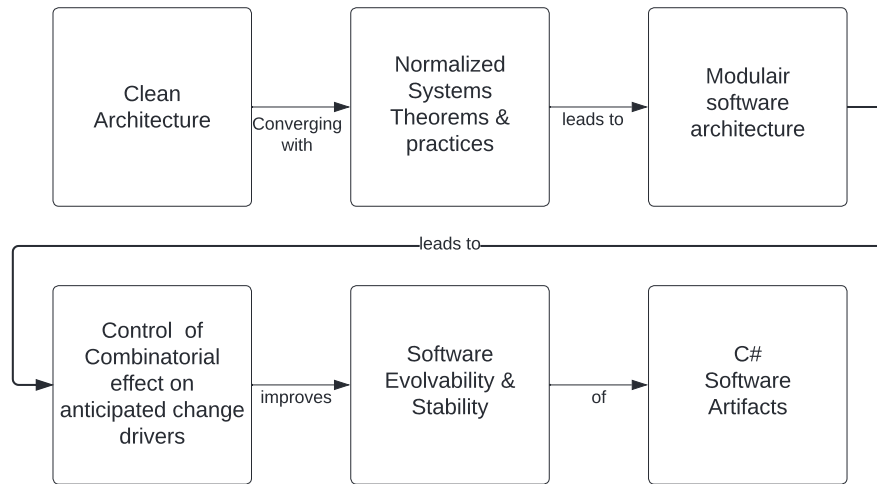
Figure 1.1.: The hypothesis

Since this research is investigating the convergence of glsca and NS, it is relevant to introduce them and discuss the concepts mentioned in the following sections.

1.1. Research method

This research is a Design Science Method and relies on the Engineering Cycles as described by Wieringa (2014). The engineering cycle provides a structured approach to developing the required artifacts to analyze the design problem.
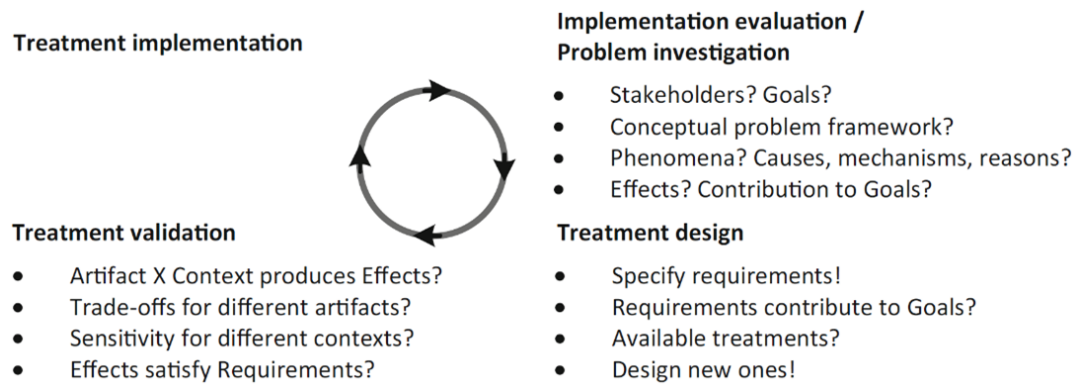
Figure 1.2.: The Engineering Cycle of Wieringa (2014)

In the context of this research, the artifacts described in chapters 4.1 and 4.2 are considered information systems. Hevner et al. proposed a framework for research in information systems by introducing the interacting relevance and rigor cycles.

Figure 1.3 depicts a specialization of the Design Science Framework of Hevner et al. (2004). The rigor cycle comprises the theories and knowledge from NS and CA, supplemented by the rigorous knowledge of modularity, evolvability, and stability of software systems. The relevance cycle represents the business needs of the stakeholders. The research requirements are described as research objectives.
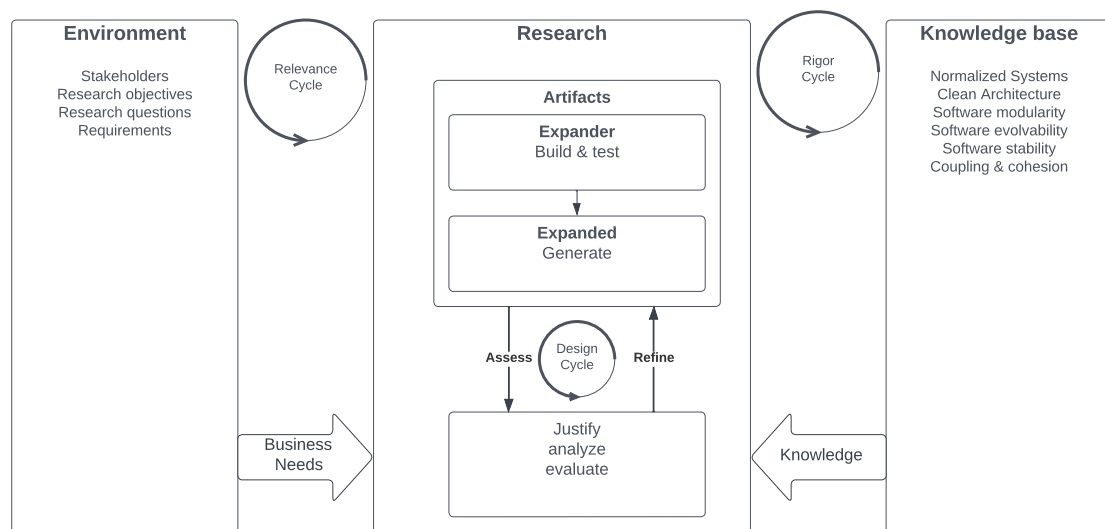
Figure 1.3.: The Design Science Framework for IS Research

1.2. Research Objectives

In this Design Science Research, we will shift the focus from Research Questions to Research Objects. The primary goal of this research is to determine the degree of convergence of CA with the NS Theory. In order to achieve this goal, the research is divided into the following objectives:

1. **Literature Analysis**
   Conduct a literature review of CA and NS, focusing on their fundamental elements, principles, and real-world case studies. This review will provide a solid foundation for understanding the underlying concepts and their practical implications.

2. **Architectural Desing**
   Create an Architectural Design fully and solely based on CA. Implement the findings of the Literature Review in the Design. This design will be the basis for the Artifact Development.

3. **Artifact Development**
   Construct two artifacts that facilitates the study on the convergence between CA and NS Theories.

   3.1. **Expander framework & Clean Architecture Expander**
   These two components will be designed and implemented based on the CA design. The Clean Architecture Expander will enable the parameterized instantiation of software systems that adhere to the principles and design of CA. The Expander framework serves as a supporting system for the expander, loading and orchestrating dependencies and models and executing the expander.

   3.2. **Expanded Clean Architecture artifact**
   The expanded artifact will facilitate the analysis of a RESTful API implementation and its alignment with the CA principles and design.

4. **Analysis of combinatorics**
   Analyze the artifacts to determine if any combinatorial effects occur due to following the principles and architectural approach of CA.

1.3. Thesis outline

The structure of this thesis reflects the research methodology described in the previous section 1.1. Chapter 2 presents the theoretical backgrounds of both NS and CA, discussing important characteristics and requirements of software stability, as well as the principles and architectures proposed by both development approaches. Chapter 3 focuses on the requirements relevant to this research and is divided into two sections. Section **??** outlines the research requirements, describing the requirements necessary for conducting the research. In contrast, section 3.2 details the artifact requirements, laying out the requirements relevant to the artifacts contributing to this research. Chapter 4 describes all the characteristics of the design artifacts. Chapter 5

evaluates the research results, discussing the impact of using CA on NS. Finally, the conclusion of this research is presented in the final chapter, Chapter 6.

2. Theoretical background

This thesis aims to study the convergence of CA with NS. Therefore, it is essential to understand the key concepts and principles, design elements, and characteristics of both software design approaches. This chapter starts with concepts that apply to both CA and NS, followed by a reference to the essential concepts, principles, and design elements of NS. We will conclude with the concepts, principles, and elements of CA.

2.1. Generic Concepts

The following sections will explore the fundamental concepts of modularity, cohesion, and coupling. These concepts are incorporated by both CA and NS and are essential to study evolvability of software artifacts.

2.1.1. Modularity

Both CA and NS use a slightly different definitions for the concept of modularity. Robert C. Martin (2018, p. 82) describes a module as a piece of code encapsulated in a source file with a cohesive set of functions and data structures. According to Mannaert et al. (2016, p. 22), modularity is a hierarchical or recursive concept that should exhibits a high degree of cohesion. While both design approaches agree on the cohesiveness of a module's internal parts, there seems to be a slight difference in granularity in their definitions.

2.1.2. High Cohesion

Mannaert et al. (2016, p. 22) consider cohesion as modules that exist out of connected or interrelated parts of a hierarchical structure. On the other hand, Robert C. Martin (2018, p. 118) discusses cohesion in the context of components. He attributes the three component cohesion principles as crucial to grouping classes or functions into cohesive components. Cohesion is a complex and dynamic process, as the level of cohesiveness might evolve as requirements change over time. The component cohesion principles are further described in *Appendix D Component Cohesion Principles*, and the beneficiary impact of applying cohesion on this research's artifacts.

2.1.3. Low Coupling

Coupling is an essential concept in software engineering related to the degree of interdependence among software modules and components. High coupling between modules indicates the strength of their relationship, whereby a high level of coupling implies a significant degree of interdependence. Conversely, low coupling signifies a weaker relationship between modules, where modifications in one module are less likely to impact others. Although not always possible, the level of coupling between the various modules of the system should be kept to a bare minimum. Both Mannaert et al. (2016, p. 23) and Robert C. Martin (2018, p. 130) agree with the idea that modules should be coupled as loosely as possible

2.2. Normalized Systems Theory

NS is a computer science approach that aims to create stable and evolvable systems. NS utilizes scientific insights from System Theory and Entropy from Statistical Thermodynamics. This Thesis acknowledges the software engineering background of NS, as it consistently refers to software Artifacts and Information Systems. However, the underlying Theory of NS can be applied to various other domains, such as Enterprise Engineering (Huysmans & Verelst, 2013), Business Process Modeling (van Nuffel, 2011), and the application in TCP-IP based firewall rule base (Haerens, 2021), emphasizing the impact of NS Theory. In the following sections, we will highlight the concepts of NS Theory that has impacted this study.

2.2.1. Stability

NS Theory considers stability a crucial property. Mannaert et al. (2016, pp. 269–270) describe that stable software is not excessively sensitive to small changes. The number of changes required for new versions of the system is not dependent on the size of that system. Conversely, instabilities occur when the total number of changes relies on the size of the system. The bigger the size of the system, the more changes are required to implement the requirement.Mannaert et al. (2016, p. 271) refer to these instabilities as Combinatorial Effects.

2.2.2. Evolvability

In section 2.2.1, it was highlighted that stability in a system is achieved when it exhibits low sensitivity to minor changes, regardless of the system's size. On the other hand, evolvability refers to a system's ability to adapt and adjust to changing requirements continuously and the absence of Combinatorial Effects.

2.2.3. Combinatorial effects versus Ripple effects

Combinatorial Effects and Ripple Effects are characteristics often used in NS Theory. Both are closely related to the challenges of maintaining evolvable systems but are certainly not interchangeable. A Ripple Effect is reffered to indicate the total amount of changes that are required to adhere to new requirements.

Mannaert et al. (2016, pp. 271–272) describe Combinatorial Effects as: "instabilities in the evolution of an information system where the number of additional software primitives is not only dependent on the amount of additional functional requirements, but also on the set of existing software primitives —the size of the system— at that point in time. These dependencies on the size of the system are caused by the dimensions of variability due to the various versions. They are combinatorial effects between the additional functional requirements, and the various existing versions of software primitives."

2.2.4. Expansion and code generation

Creating and maintaining a stable and evolvable system is, according to Mannaert et al. (2016, p. 403), a particularly challenging, repetitive, and meticulous engineering job. Developers must have a sound knowledge of NS while implementing new requirements in an always consistent manner. Mannaert et al. (2016, p. 403) propose to automate the instantiation process of software structures by using code generation for recurring tasks

2.2.5. Harvesting and Rejuvenation

Expansion and code generation should embrace manually added craftings on parts of the system where automation is not possible or desirable. These craftings are preserved after each expansion by a method that is called harvesting and rejuvenation (Mannaert et al., 2016, pp. 405–406).

2.2.6. The Design Theorems

In the following table we will describe the Design Theorems of NS, firstly presented by Mannaert and Verelst (2009, pp. 111–119). They are known as Separation Of Concerns (SoC), Data Version Transparency (DvT), Action Version Transparency (AvT) and Separation of State (SoS).

Principle	Definition
SoC	The latest definition of SoC has been defined by Mannaert et al. (2016, p. 274) as: A processing function can only contain a single task to achieve stability.
DvT	A data structure that is passed through the interface of a processing function needs to exhibit version transparency to achieve stability.
AvT	A processing function that is called by another processing function, needs to exhibit version transparency to achieve stability.
SoS	Calling a processing function within another processing function, needs to exhibit state keeping to achieve stability.

Table 2.1.: The Design Theorems of Normalized Systems.

2.2.7. Normalized Elements

In the context of the NS Theory approach, the goal is to design evolvable software, independent of the underlying technology. Nevertheless, when implementing the software and its components, a particular technology must be chosen. For Object Oriented Programming Languages like Java, the following Normalized Elements have been proposed (Mannaert et al., 2016, pp. 363–398).It is essential to recognize that different programming languages may necessitate alternative constructs (Mannaert et al., 2016, p. 364).

Element	Description
Data	This object represents a piece of data in the system. Data elements are used to pass information between processing functions and other objects. In NS, data elements are typically standardized to ensure consistency across the system.
Task	This object represents a specific task or action in the system. Tasks can be composed of one or more processing functions and can be used to represent complex operations within the system.
Connector	This object is used to connect different parts of the system. Connectors can link processing functions, data elements, and other objects to work together seamlessly.
Flow	This object represents the flow of control through the system. It determines the order in which processing functions are executed and can be used to handle error conditions or other exceptional cases.
Trigger	A trigger represents an object that reacts to specific events or changes in the system by executing predefined actions.

Table 2.2.: The Elements proposed by Normalized Systems Theory

2.3. Clean architecture

CA is a software design approach that emphasizes the organization of code into independent, modular layers with distinct responsibilities. This approach aims to create more flexible, maintainable, and testable software systems by enforcing the separation of concerns and minimizing dependencies between components. The goal of clean architecture is to provide a solid foundation for software development, allowing developers to build applications that can adapt to changing requirements, scale effectively, and remain resilient against the introduction of bugs (Robert C. Martin, 2018).

2.3.1. The Design principles

Robert C. Martin (2018, p. 78) argues that software can quickly become a well-intended mess of bricks and building blocks without a rigorous set of design principles. So, from the early 1980s, he began to assemble a set of software design principles as guidelines to create software structures that tolerate change and are easy to understand. The principles are intended to promote modular and component-level software structure (Robert C. Martin, 2018, p. 79). In 2004 the arrangement of the principles was definitively arranged to form the acronym SOLID.

The following sections will provide an overview of each of the SOLID principles.

The Single Responsibility Principle

This principle has gone through several iterations of the formal definition. The final definition of the Single Responsibility Principle (SRP) is: *a module should be responsible to one, and only one, actor* (Robert C. Martin, 2018, p. 82). The word actor in this statement refers to all the users and stakeholders represented by the (functional) requirements. The modularity concept in this definition is described by Robert C. Martin (2018, p. 82) as a cohesive set of functions and data structures.

In conclusion, this principle allows for modules with multiple tasks as long as they cohesively belong together. Robert C. Martin (2018, p. 81) acknowledges the slightly inappropriate name of the principle, as many interpreted it that a module should do just one thing.

The Open-Closed Principle

Meyer (1988) first mentioned the Open/Closed Principle (OCP) and formulated the following definition: *A module should be open for extension but closed for modification.* The software architecture should be designed such that the behavior of a module can be extended without modifying existing source code. The OCP promotes the use of abstraction and polymorphism to achieve this goal. The OCP is one of the driving forces behind the software architecture of systems making it rellatively easy to apply new requirements. (Robert C. Martin, 2018, p. 94).

The Liskov Substitution Principle

TheLiskov Substitution Principle (LSP) is named after Barbara Liskov, who first introduced the principle in a paper she co-authored in 1987. Barbara Liskov wrote the following statement to define subtypes (Robert C. Martin, 2018, p. 95).

If for each object o1 of type S, there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.1. Or in simpler terms: To build Software from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted for an other (Robert C. Martin, 2018, p. 80)

The Interface Segregation Principle

The Interface Segregation Principle (ISP) suggests that software components should have narrow, specific interfaces rather than broad, general-purpose ones. In addition, the ISP states that consumer code should not be allowed to depend on methods it does not use. In other words, interfaces should be designed to be as small and focused as possible, containing only the methods relevant to the consumer code using them. This allows for the consumer code to use only the needed methods without being forced to implement or depend on unnecessary methods (Robert C. Martin, 2018, p. 104).

The Dependency Inversion Principle

The Dependency Inversion Principle (DIP) prescribes that high-level modules should not depend on low-level modules and that both should depend on abstractions. The principle emphasizes that the architecture should be designed so that the flow of control between the different objects, layers, and components is always from higher-level implementations to lower-level details. In other words, high-level implementations, like business rules, should not be concerned about low-level implementations, such as how the data is stored or presented to the end user. Additionally, both the high-level and low-level implementations should only depend on abstractions or interfaces that define a contract for how they should interact with each other (Robert C. Martin, 2018, p. 91).

This approach allows for great flexibility and a modular architecture. Modifications in the low-level implementations will not affect the high-level implementations as long as they still adhere to the contract defined by the abstractions and interfaces. Similarly, changes to the high-level modules will not affect the low-level modules as long as they still fulfill the contract. This reduces coupling and ensures the evolvability system over time, as changes can be made to specific modules without affecting the rest of the system.

2.3.2. Component architecture

CA organizes their components into distinct layers. This architecture promotes the separation of concerns, maintainability, testability, and adaptability. The following section is a short description of each layer (Robert C. Martin, 2018).

Domain layer

This layer contains the application's core business objects, rules, and domain logic. Entities represent the fundamental concepts and relationships in the problem domain and are independent of any specific technology or framework. The domain layer focuses on encapsulating the essential complexity of the system and should be kept as pure as possible.

Application layer

This layer contains the use cases or application-specific business rules orchestrating the interaction between entities and external systems. Use cases define the application's behavior regarding the actions users can perform and the expected outcomes. This layer is responsible for coordinating the flow of data between the domain layer and the presentation or infrastructure layers while remaining agnostic to the specifics of the user interface or external dependencies.

Presentation layer

This layer translates data and interactions between the use cases and external actors, such as users or external systems. Interface adapters include controllers, view models, presenters, and data mappers, which handle user input, format data for display, and convert data between internal and external representations. The presentation layer should be as thin as possible, focusing on the mechanics of user interaction and deferring application logic to the use cases.

Infrastructure layer

This layer contains the technical implementations of external systems and dependencies, such as databases, web services, file systems, or third-party libraries. The infrastructure layer provides concrete implementations of the interfaces and abstractions defined in the other layers, allowing the core application to remain decoupled from specific technologies or frameworks. This layer is also responsible for configuration or initialization code to set up the system's runtime environment.

By organizing code into these layers and adhering to the principles of CA, developers can create software systems that are more flexible, maintainable, and testable, with well-defined boundaries and separation of concerns

2.3.3. The Design Elements

Robert C. Martin (2018) proposes the following elements to achieve the goal of "Clean Architecture".

Element	Description
Entity	Entities are the core business objects, representing the domain's fundamental data.
Interactor	Interactors encapsulate business logic and represent specific actions that the system can perform.
RequestModel	RequestModels are used to represent the input data required by a specific interactor.
ViewModel	ViewModels are responsible for managing the data and behaviour of the user interface.
Controller	Controllers are responsible for handling requests from the user interface and routing them to the appropriate Interactor.
Presenter	Presenters are responsible for formatting and the data for the user interface.
Gateway	A Gateway provides an abstraction layer between the application and its external dependencies, such as databases, web services, or other external systems.
Boundary	Boundaries are used to separate the the different layers of the component.

Table 2.3.: The Elements proposed by Clean Architecture

2.3.4. The Dependency rule

An essential aspect is described as the dependency rule. The rule states that *source code dependencies must point only inward toward higher-level policies* (Robert C. Martin, 2018, p. 206). This 'flow of control' is designed following the DIP and can be represented schematically as concentric circles containing all the components described in section 2.3.2 Component architecture The arrows in Figure 2.1 clearly show that the dependencies flow from the outer layers to the inner layers. Most outer layers are historically subjected to large-scale refactorings due to technology changes and innovation. Separating the layers and adhering to the dependency rule ensures that the domain logic can evolve independently from external dependencies or certain specific technologies.
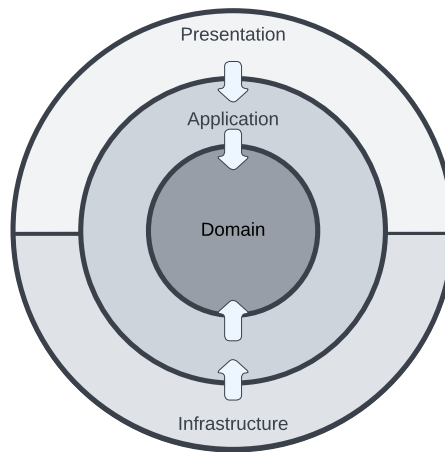
Figure 2.1.: Flow of control

2.3.5. Screaming Architecture

Robert C. Martin adopts this concept of Screaming Architecture from Jacobson (1992), who points out that Software Architecture are structures that supports Use Cases of a Software System. Robert C. Martin (2018, p. 195) builds on that idea describing that Software Architectures should emphasize the intent, theme and purpose of the System, rather than being dictated by frameworks, technology, and delivery mechanisms.

3. Requirements

This Chapter outlines the requirements for this Design Science Research study, where we focus on the stability a& evolvability of Software Artifacts. Section 3.1 begins by discussing Software Transformation Requirements proposed by Mannaert et al. (2016), which serve as a foundation for assessing the stability & evolvability of the Artifacts. Next, Section 3.2 details the specific requirements of the Artifacts used in this study. These requirements will help ensure that the Artifacts are suitable for evaluating the stability & evolvability of Software Artifacts designed based on Clean Architecture and SOLID Principles.

3.1. Software Transformation Requirements

We study stability and evolvability by investigating potential Combinatorial Effects in CA artifacts. Therefore, during the implementation, we will apply parts of the Functional-Construction Software Transformation from Mannaert et al. (2016, p. 251) by using the following five proposed Functional Requirements Specifications. Mannaert et al. (2016, pp. 254–261) have defnined them as followed.

1. An information system needs to be able to represent instances of data entities. A data entity consists of several data fields. Such a field may be a basic data field representing a value of a reference to another data entity.

2. An information system needs to be able to execute processing actions on instances of data entities. A processing action consists of several consecutive processing tasks. Such a task may be a basic task, i.e., a unit of processing that can change independently, or an invocation of another processing action.

3. An information system needs to be able to input or output values of instances of data entities through connectors.

4. An existing information system representing a set of data entities, needs to be able to represent a new version of a data entity that corresponds to including an additional data field and an additional data entity.

5. An existing information system providing a set of processing actions, needs to be able to provide a new version of a processing task, whose use may be mandatory, a new version of a processing action, whose use may be mandatory, an additional processing task and an additional processing action

3.2. Artifact requirements

Chapter 1.2 Research Objectives outlines the construction of two artifacts. Both of these artifacts will be meticulously designed and developed in accordance with the design philosophy and principles of CA, by strict adherence to the following requirements.

3.2.1. Component Architecture Requirements

The following requirements are applied to the Component Architecture of both the Generator Artifact and the Generated Artifact.

Nr.	Requirement
1	The solution is organized into separate Visual Studio projects for the Domain, Application, Infrastructure, and Presentation layers of the component. A detailed description of these layers can be found in Section 2.3.2 Component architecture.
2	The Visual Studio projects representing the component layers comply with the naming conventions outlined in the appendix C.1 Component Layer Naming Conventions
3	The dependencies between the component layers must follow an inward direction towards the higher-level components as illustrated in Figure 2.1 schematically, and cannot skip layers.

Table 3.1.: The Component Architecture Requirements

Nr.	Technology Requirement
1	The Domain and Application layers have no dependencies on any infrastructure technologies, like web- or database technologies.
2	The Presentation Layer relies on various infrastructure technologies for facilitating end-user interaction. Examples of such technologies include Command Line Interfaces (CLIs), RESTful APIs, and web-based solutions. Each dependency is isolated and managed in separate Visual Studio Projects to ensure the stability and evolvability of the system.
3	The Infrastructure Layer may rely on other infrastructure components, such as databases or filesystems. Each infrastructure dependency is isolated and managed in separate Visual Studio Projects to promote stability and evolvability.
4	All Component Layers utilize the C# programming language, explicitly targeting the .NET 7.0 framework.
5	Reusing existing functionality or technology (packages) is permitted only when adhering to the LSP and utilizing the open-source package manager, Nuget.org.

Table 3.2.: Application Layer Technology Requirements

3.2.2. Software Architecture Requirements

Figure 3.1 illustrates the generic Software Architecture of the Artifacts. Each instantiated element adheres to the Element Naming Convention outlined in Appendix C.2. In addition, the following tables detail the requirements specific to each element.
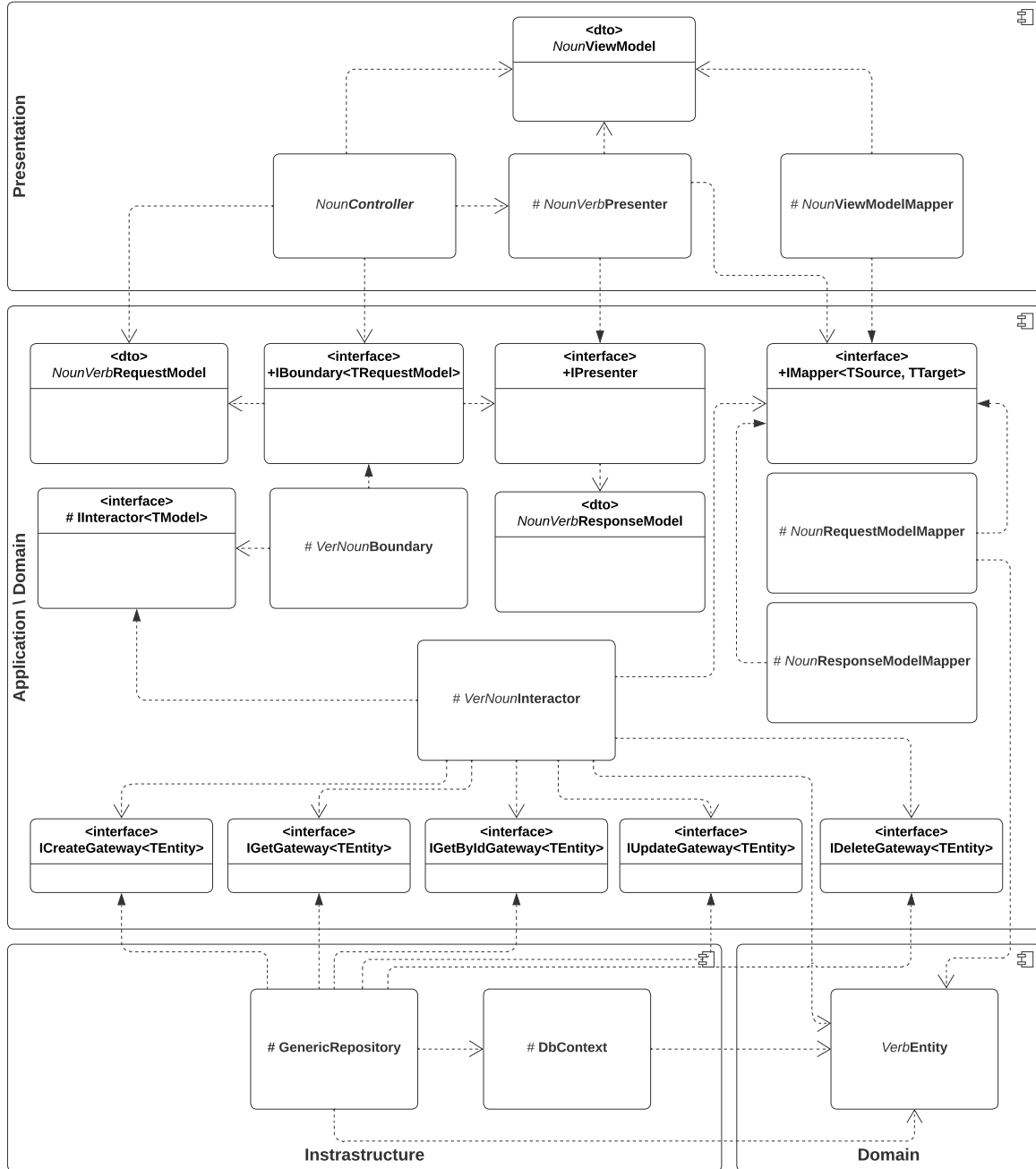

Figure 3.1.: The Generic architecture of the artifacts

Presentation Layer

Nr.	ViewModel Requirement
1	The ViewModel consists of data attributes representing fields from the corresponding Entity. In addition, it may contain information specific to the user interface.
2	The ViewModel has no external dependencies on other objects within the architecture.

Table 3.3.: ViewModel Requirements

Nr.	Presenter Requirement
1	The Presenter Implementation is derived from the IPresenter interface and follows the specified implementation. The IPresenter interface can be found in the Application Layer.
2	The Presenter is responsible for creating the Controller's Response by instantiating the ViewModel, constructing the HTTP Response message, or combining both elements as needed.
3	When required, the Presenter utilizes the IMapper interface without depending on specific implementations of the IMapper interface.
4	The Presenter has an internal scope and cannot be instantiated outside of the Presentation layer.

Table 3.4.: Presenter Requirements

Nr.	ViewModelMapper Requirement
1	The ViewModelMapper is derived from the IMapper interface and follows the specified implementation. The IMapper interface can be found in the Application Layer.
2	The ViewModelMapper is responsible for mapping the values of the necessary data attributes from the ResponseModel to the ViewModel.
3	The ViewModelMapper has an internal scope and cannot be instantiated outside of the Presentation layer.

Table 3.5.: ViewModelMapper Requirements

Nr.	Controller Requirement
1	The Controller is responsible for receiving external requests and forwarding the request to the appropriate Boundary within the Application Layer.
2	The Controller relies on the IBoundary interface without depending on specific implementations of the IBoundary interface.

Table 3.6.: Controller Requirements

Application Layer

Nr.	IBoundary Requirements
1	The IBoundary interface establishes the contract for its derived Boundary implementations.
2	The IBoundary interface has public scope within the system.

Table 3.7.: IBoundary Requirements

Nr.	Boundary Implementation Requirements
1	A Boundary implementation is derived from the IBoundary interface and follows the specified implementation.
2	The Boundary implementation serves as a separation between the internal aspects of the Application Layer and the other layers within the Component.
3	Each Boundary implementation handles a single task, which is then executed using the IInteractor interface.
4	Boundary implementations have an internal scope and cannot be instantiated outside the Application Layer.

Table 3.8.: Boundary Implementation Requirements

Nr.	IInteractor Requirements
1	The IInteractor interface establishes the contract for its derived Interactor implementations.
2	The IInteractor has an internal scope and cannot be implemented outside the Application Layer.

Table 3.9.: IInteractor Requirements

Nr.	Interactor Implementation Requirements
1	An Interactor implementation is derived from the IInteractor interface and follows the specified implementation.
2	The Interactor implementation executes a single task or orchestrates a series of tasks. Each of these tasks is implemented in separate Interactors. Alternatively, a Gateway is used for Tasks with Infrastructure dependencies, such as data persistence in a database.
3	Depending on the Task, the Interactor implementation orchestrates the mapping from RequestModels to Entities, or from Entities to ResponseModels, utilizing the IMapper interface.
4	Interactor implementations have an internal scope and cannot be implemented outside the Application Layer.

Table 3.10.: Interactor Implementation Requirements

Nr.	IMapper Requirements
1	The IMapper interface establishes the contract for its derived Mapper implementations.
2	The IMapper interface has a public scope within the system.

Table 3.11.: IMapper Requirements

Nr.	RequestModelMapper Requirement
1	The RequestModelMapper is derived from the IMapper interface and follows the specified implementation.
2	The RequestModelMapper is responsible for mapping the values of the necessary data attributes from the RequestModel to an Entity.
3	The RequestModelMapper has an internal scope and cannot be implemented outside the Application Layer.

Table 3.12.: RequestModelMapper Requirements

Nr.	ResponseModelMapper Requirement
1	The RequestModelMapper is derived from the IMapper interface and follows the specified implementation.
2	The RequestModelMapper is responsible for mapping the values of the necessary data attributes from the RequestModel to an Entity.
3	The RequestModelMapper has an internal scope and cannot be implemented outside the Application Layer.

Table 3.13.: ResponseModelMapper Requirements

Nr.	IPresenter Requirements
1	The IPresenter interface establishes the contract for its derived Presenter implementations, typically implemented as part of the Presentation Layer.
2	The IPresenter interface has a public scope within the system.

Table 3.14.: IPresenter Requirements

Nr.	I*[Verb]*Gateway Requirements
1	The *[Verb]Gateway* interface establishes the contract for its derived Gateway implementations, which are typically implemented in the Infrastructure Layer.
2	The *[Verb]*Gateway interface has a public scope within the system.
3	Each task is represented in the naming convention of the interface. As an example, the basic CRUD actions result in a total of five IGateway interfaces: ICreateGateway, IGetGateway, IGetByIdGateway, IUpdateGateway, and IDeleteGateway.

Table 3.15.: *[Verb]*Gateway Requirements

Nr.	ResponseModel Requirement
1	The ResponseModel consists primarily of data attributes representing the fields of the corresponding Entity. Additionally, the ResponseModel may contain data specific to the output of the Interactor.
2	The ResponseModel does not depend on external objects within the architecture.

Nr.	RequestModel Requirement
1	The RequestModel consists primarily of data attributes representing the fields of the corresponding Entity. Additionally, the RequestModel may contain data specific to the input of the Interactor.
2	The RequestModel does not depend on external objects within the architecture.

Domain Layer

Nr.	Data Entity Requirement
1	The Data Entity consists solely of attributes representing the corresponding data fields.
2	The Data Entity does not rely on external objects within the architecture.
3	The Application Layer is the only layer that utilizes the Data Entity.

The Infrastructure Layer

Nr.	[*Verb*]Gateway Implementation Requirement
1	The [*Verb*]Gateway Implementation derives from the I[*Verb*]Gateway interface and adheres to the specified implementation.
2	The [*Verb*]Gateway Implementation is responsible for the interaction associated with the specific task, utilizing the infrastructure technology of the specific layer (e.g., a SQL database or a filesystem). The [*Verb*]Gateway Implementation has an internal scope and cannot be instantiated outside of the Layer.

Design Principles compliancy

Each architectural pattern adheres to at least one of the SOLID principles to ensure that none of the implementations violate these principles.

3.2.3. Expander Requirements

In addition to the more generic requirements of previous sections, the following requirements are specific for Clean Architecture Exander & Expander Framework Artifact.

Nr. Expander Framework Requirements

Table 3.20.: Expander Framework Requirements

Nr. Clean Architecture Expander Requirements

Table 3.21.: Clean Architecture Expander Requirements

3.2.4. Generated Artifact Requirements

4. Designing artifacts.

4.1. Designing and Creating the generator artifact

In the previous chapters, we discussed the importance of software stability and evolvability in order to cope with the continuously changing business and technological requirements. Although considered from a much broader perspective, the Greek philosopher *Heraclitus* described this state of constant *flux* with his famous statement *Pantha Rhei*. His statement was the main inspiration for the name of the generator artifact.

The following sections discuss the design, implementations and functionality of the Pantha Rhei, the generator artifact. The artifact is created in fulfillment of this research. Whereas the name of the artifact was inspired by Heraclitus, the functional idea behind the artifact was inspired by the theory behind NS and by the Prime Radiant application of the company NSX [1]. The concept of code generation in order to achieve stable and evolvable software is one of the important aspects of NS.

For the interested readers of this thesis, it is rather simple to install the Pantha Rhei application by following the instruction in the appendix A Installing & using Pantha Rhei.

4.1.1. Expanders

As described in section 4.1, and portraited in figure 4.1, expanders are used as plugins by the Pantha Rhei expander artifact. There are a couple of prerequisites applicable before the expander can be dynamically loaded as a plugin at runtime.

Prerequisite 1: Project dependency

The expander should have a dependency on the Dynamic Link Library (DLL) of the project "PanthaRheiGeneratorDomain" (Koks, 2023j).

---

[1]https://normalizedsystems.org/prime-radiant/

Prerequisite 2: Implements *"IExpanderInteractor"*

In order to behave like an expander, one should behave like an expander. This is done by implementing the "IExpanderInteractor" (Koks, 2023f) interface. Although not required, it is strongly recommended to use the abstract "AbstractExpander" (Koks, 2023a) as it contains all the routines required implementations of "IExecutionInteractor" (Koks, 2023d) like Harvesters, Rejuvenators, Pre-, and PostProcessors.

Prerequisite 3: Implements *"AbstractExpanderDependencyManagerInteractor"*

The Dependency Injection pattern is an exciting and beneficiary pattern that entirely adheres to the DIP principle. By implementing the abstract "AbstractExpanderDependencyManagerInteractor" (Koks, 2023b) all, for the expander specific implementations of the following interfaces will automatically be registered and made available for runtime processing.

- The expander which should be an implementation of "IExpanderInteractor" (Koks, 2023f) or "AbstractExpander" (Koks, 2023a)

- When applicable, the expander handers, which should be an implementation of "IExpanderHandlerInteractor" (Koks, 2023e)

- The default "RegionHarvesterInteractor" (Koks, 2023m) will automatically be executed during the generation process.

- Specific implementations of Harversters, which should be an implementation of "IHarvesterInteractor" (Koks, 2023g)

- The default "RegionRejuvenatorInteractor" (Koks, 2023n) will automatically be executed during the generation process.

- Specific implementations of Rejuvenators, which should be an implementation of "IRejuvenatorInteractor" (Koks, 2023i)

- The default PostProcessor "InstallDotNetTemplateInteractor" (Koks, 2023h) will automatically be executed during the generation process.

- Specific implementation of PostProcessors, which should be an implementation of "PostProcessorInteractor" (Koks, 2023k)

- The default PreProcessor "UnInstallDotNetTemplateInteractor" (Koks, 2023o) will automatically be executed during the generation process.

- Specific implementation of PreProcessors, which should be an implementation of "PreProcessorInteractor" (Koks, 2023l)

4.1.2. Plugin Architecture

When all preconditions are met and the expander is compiled, the expander consists of a DLL and a set of templates. The Generator artifact considers the expanders as optional plugins, which are dynamically loaded at runtime, through a method called assembly-binding. See section 4.1.1 Expanders for a full explanation of the required preconditions.
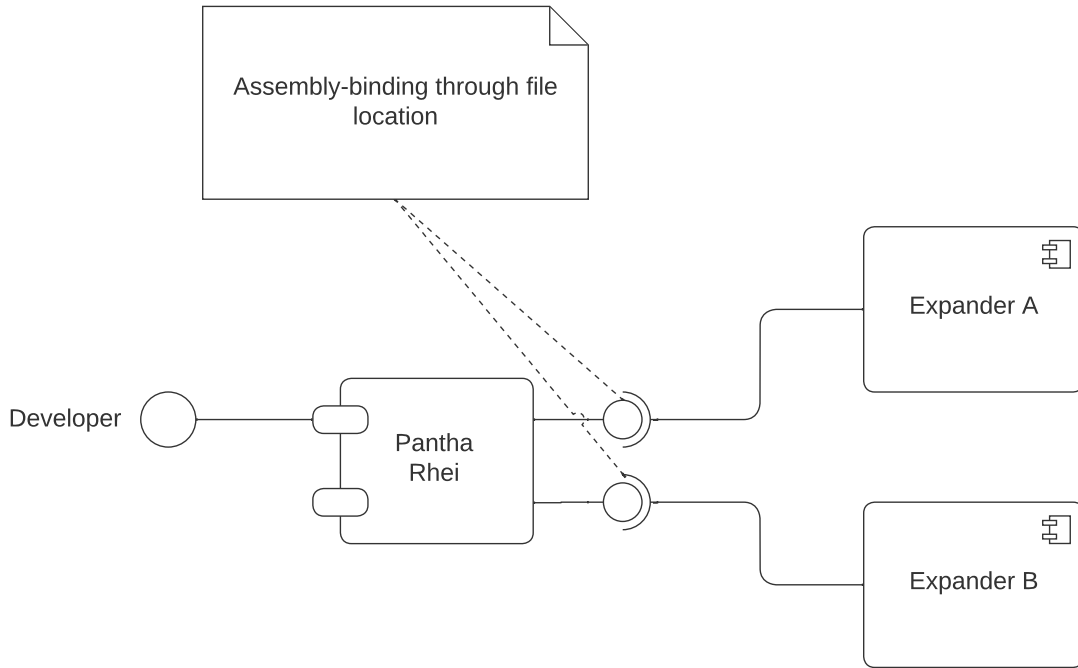

Figure 4.1.: Expanders are considered plugins

By implementing the expanders as plugins, the design adheres to several Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle (DIP) principles. First and foremost, Single Responsibility Principle (SRP) is respected because an expander should generate one, and precisely one construct (Mannaert et al., 2016, p. 403). In the case of this research, this construct is an application following the design and principles of the design approach of CA with a Restful API interface. Furthermore, it supports the OCP principles because developers can introduce a new version of the expander as a separate plugin if this is required. LSP. By extension, this also adheres to the LSP principle, as expanders can be replaced with other implementations of expanders, without affecting the rest of the application.

4.1.3. The executer object

A vital implementation that facilitates a high degree of cohesion, whilst maintaining low coupling and adhering to the SRP principle is the use of the "IExecutionInteractor" (Koks, 2023d)

interface. The generation process is designed to execute "tasks" in a predefined order. By using the *"IExecutionInteractor"* it is possible to design each of the tasks as separate classes, entirely complying, or enabling all of the SOLID principles. The
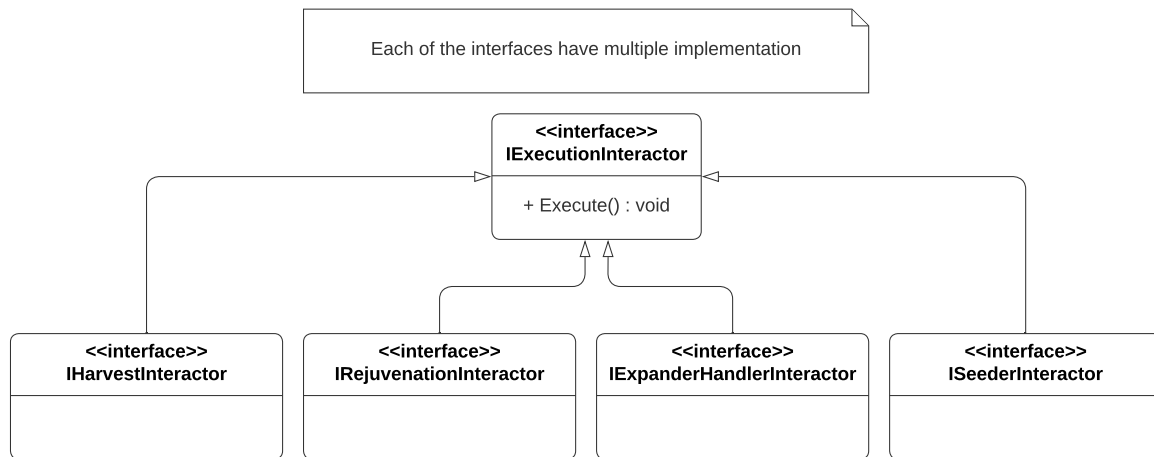
Figure 4.2.: Both high cohesion and low coupling by using the *"IExecutionInteractor"*

As depicted in listing 4.1, this design leads to a cohesive design where all tasks are gracefully executed from a single point in the application.

```
1  /// <summary>
2  /// Implements the contract <seealso cref="ICodeGeneratorInteractor"/>.
3  /// </summary>
4  internal sealed class CodeGeneratorInteractor : ICodeGeneratorInteractor
5  {
6      // ... other code
7
8      /// <inheritdoc/>
9      public void Execute()
10     {
11         foreach (IExpanderInteractor expander in expanders
12             .OrderBy(x => x.Model.Order))
13         {
14             expander.Harvest();
15
16             Clean();
17
18             expander.PreProcess();
19             expander.Expand();
20             expander.Rejuvenate();
21             expander.PostProcess();
22         }
23     }
24
25     // ... other code
26 }
```

Listing 4.1: The "CodeGeneratorInteractor"

4.1.4. The meta-model

The meta-model is a higher-level abstraction of the model that is used to describe the characteristics of a software system in the context of this research. It contains the essential characteristics that are needed in order to generate software applications and components. It contains the structures, entities and relationships within the Generated Artifact. Having a meta-model ensures a standardized way to describe and define the software components and their interactions, enabling the design and analysis of modular, evolvable, and scalable software systems.

In this context, the meta-model serves as a blueprint for the software system, describing its components, such as entities, fields, relationships, and expanders, along with their attributes and constraints. The meta-model is used to generate the Generated Artifact (4.2). Moreover, it is potentially reusable for other types of applications, which should lead to the same characteristics of stability and evolvability as the one used for this research.

The following sections describe all the entities that are part of the meta-model. These entities are the basis of the Entity Relation Diagram (ERD) depicted in Appendix B. The ERD is the official model representing the meta-model.

The App entity

The App entity represents an application and is regarded as the highest entry point for the Generator artifact. The App Entity and subsequent entities contain all the information needed to generate the Generated Artifact described in section 4.2.

Name	DataType	Description
Id	Guid	Unique identifier of the application
Name	string	Name of the application
FullName	string	Full name of the application
Expanders	List of Expanders	The Expanders that will be used during the generation process.
Entities	List of Entities	The Entities that are applicable for the Generated artifact.
ConnectionStrings	List of Connection-Strings	The ConnectionString to the database that is used by the Generator Artifact.

Table 4.1.: The fields of the App entity

The Component entity

The Component entity represents a software component that can be part of an application. Based on this entity the Generator Artifact can make design time on where to place certain elements

27

Name	DataType	Description
Id	Guid	Unique identifier of the component
Name	string	Name of the component
Description	string	Description of the component
Packages	List of Package	The Packages that should be applied to the component.
Expander	Expander	Navigation property to the Expander entity.

Table 4.2.: The fields of the Component entity

The ConnectionString entity

The ConnectionString entity represents a ConnectionString used by an application to connect to a database or other external system.

Name	DataType	Description
Id	Guid	Unique identifier of the ConnectionString
Name	string	Name of the ConnectionString
Definition	string	Definition of the ConnectionString
App	App	Navigation property to the App entity

Table 4.3.: The fields of the ConnectionString entity

The Entity entity

The Entity entity represents an entity in the application's data model.

Name	DataType	Description
Id	Guid	Unique identifier of the entity
Name	string	Name of the entity
Callsite	string	The source code location where the entity is defined. In the case of a C# artifact, this is to determine the name of the namespace.
Type	string	Type of the entity
Modifier	string	Modifier of the entity (e.g. public, private)
Behavior	string	The behavior of the entity (e.g. abstract, virtual)
App	App	Navigation property to the App entity.
Fields	List of Fields	The Fields property represents a collection of the fields that make up the entity.
ReferencedIn	List of Fields	Represents a navigation property to a Field that uses the current entity as a return type.
Relations	List of Relationships	List of relationships involving this entity
IsForeignEntityOf	List of Relationships	List of relationships where this entity is the foreign entity

Table 4.4.: The fields of the Entity entity

The Expander entity

The Expander entity represents an expander, which is responsible for generating code for an application. The Generator Artifact attempts to execute all expanders that are related to the selected App.

Name	DataType	Description
Id	Guid	Unique identifier of the expander
Name	string	Name of the expander
TemplateFolder	string	relative path to the templates that are used by the expander.
Order	int	The order in which the expander is executed
Apps	List of Apps	List of applications associated with the expander.
Components	List of Components	List of components associated with the expander

Table 4.5.: The fields of the Expander entity

The Field entity

The Field entity represents a field or property of an entity in an application's data model. Each field has a unique ID, name, and other properties such as its return type, modifiers, and behavior. It can be associated with an entity and can have relationships with other entities.

The IsKey and IsIndex properties indicate whether the field is part of the primary key or an index of the entity, respectively.

Name	DataType	Description
Id	Guid	Unique identifier of the field
Name	string	Name of the field
ReturnType	string	Return type of the field
IsCollection	bool	Whether the field is a collection or not
Modifier	string	Modifier of the field (e.g. public, private)
GetModifier	string	Modifier of the get accessor for the field
SetModifier	string	Modifier of the set accessor for the field
Behavior	string	The behavior of the field (e.g. abstract, virtual)
Order	int	The order of the field within its entity
Size	int?	The size of the field
Required	bool	Whether the field is required or not
Reference	Entity	The entity that this field refers to
Entity	Entity	A navigation property to the parent entity
IsKey	bool	Indicates whether the field is part of the primary key
IsIndex	bool	Indicates whether the field is part of an index
RelationshipKeys	List of Relationships	A List of entities that are defined as relations.
IsForeignEntityKeyOf	List of Relationships	List of relationships to the field that is the foreign key

Table 4.6.: The fields of the Field entity

The Package entity

The Package entity represents a software package that can be used by a component. This could either be a Nuget package in the case of .NET projects, or for example npm packages for web projects.

Name	DataType	Description
Id	Guid	Unique identifier of the package
Name	string	Name of the package
Version	string	Version of the package used
Component	Component	Component associated with the package

Table 4.7.: The fields of the Package entity

The Relationship entity

The Relationship entity represents a relationship between two entities in the App's data model. The Relationship entity has proper cardinality support. Relationships are bidirectional and can be navigated from either entity.

Name	DataType	Description
Id	Guid	Unique identifier of the relationship
Key	Field	The key field of the relationship
Entity	Entity	Navigation property to the parent Entity
Cardinality	string	The cardinality of the relationship
WithForeignEntityKey	Field	The foreign key field of the relationship, pointing to a Field entity.
WithForeignEntity	Entity	The entity associated with the foreign key field
WithCardinality	string	The cardinality of the relationship with the foreign entity
Required	bool	indicates whether the relationship is required or not

Table 4.8.: The fields of the Relationship entity

4.2. Desigingin and generating a generated artifact

4.2.1. The model

4.3. Research Design Decision

In accordance to chapter **??**, the following design decisions have been made to be able to analyze the stability and the evolvability of both of the artifacts.

4.3.1. Fulfilling Research Requirement 1

The following Data entities will be created as part of the initial versions of both artifacts. A full description of the data entities and their relations towards each other can be reviewed in chapter 4.1.4 The meta-model. A description of all attributes are included. TODO:toevoegen dat de entity, task entiteiten een relaties moet krijgen met component entitiet zodat plaatsen van entiteiten in componenten niet meer in designtime moet.

5. Evaluation results

This Chapter focuses on the comparison of CA and NS and explores the convergence between the two architectural design approaches. We will start by comparing the principles of CA with the theories of NS and analyze how they converge in their approach and the effect on the software design. In section 5.2, we will compare the design elements of the two architectural approaches. We will highlight the similarities and differences in their implementation, in order to determine the further convergence of the two discussed architectural design approaches.

To conduct this comparison, we used a combination of artifacts and a literature study. We analyzed real-world examples of software artifacts that were designed using CA, and reviewed the relevant literature to gain a comprehensive understanding of both approaches.

Throughout this evaluation chapter, we will explore the strengths and weaknesses of each approach, as well as the potential benefits of their convergence. By examining the principles and design elements of CA and NS, we hope to provide a clear understanding of the effects of using the different approaches in conjunction with each other.

5.1. Converging principles

In this section, we will apply a systematic cross-referencing approach to assess the level of convergence between each of the SOLID principles of CA and the NS theories. Along with a brief explanation, the level of convergence is denoted as followed:

Fully converges	✚✚	This indicates a high degree of alignment between the respective SOLID principle and NS theorem. The application of either principle or theorem results in a similar impact on the software design.
Supports convergence	✚	In this case, the SOLID principle assists in implementing the NS theorem through specific design choices. However, it is essential to note that applying the principle does not inherently ensure adherence to the corresponding theorem.
No convergence	▬	This denotation signifies a lack of alignment between the SOLID principle and the corresponding theorem.

5.1.1. Converging the Single Responsibility Principle

SoC	✚✚	SRP and SoC share a common objective: facilitating evolvable software systems through the promotion of modularity, low coupling, and high cohesion. While there may be some differences in granularity when applying both principles according to the original definition of SOC, the more stringent definition of Separation of Concerns, offered by the NS Theorems minimizes these differences. As a result, the two principles can be regarded as practically interchangeable. In conclusion, SRP and SOC exhibit full convergence, as they both emphasize encapsulating 'responsibilities' or 'concerns' within modular components of a software system.
DvT	✚	While not immediately apparent, SRP offers supports for the DvT theorem. While SRP emphasizes limiting the responsibility of each module, it does not explicitly require handling changes in data structures. However, following glssrp can still indirectly contribute to achieving DvT by promoting the Law of Demeter [1], which encourages modules to interact with each other only through well-defined interfaces. This approach can minimize the impact of data structure changes, although it does not guarantee full convergence with DvT.
AvT	✚	Although not that apparent, SRP supports the DvT theorem. While SRP emphasizes limiting the responsibility of each module, it does not explicitly require handling changes in data structures. However, following glssrp can still indirectly contribute to achieving DvT by promoting the Law of Demeter [2], which encourages modules to interact with each other only through well-defined interfaces. This approach can minimize the impact of data structure changes, although it does not guarantee full convergence with DvT.
SoS	➖	The convergence between SRP and the SoS theorem is not as direct as with other theorems. SRP focuses on assigning a single responsibility to each module but does not explicitly address state management. Nevertheless, by following SRP, developers can create modules that manage their state, which indirectly contributes to SoS.

Table 5.1.: Converge SRP with NS

5.1.2. Converging the Open/Closed Principle

SoC	✚✚	The OCP converges with the SoC theorem. OCP states that software implementations should be open for extension but closed for modification. When applying OCP correctly, modifications are separated from the original implementations. For example by creating a new implementation of an interface or a base class. Conversely, adhering to SoC does not guarantee the fulfillment of OCP, as SoC focuses on modularization and encapsulation, rather than the extensibility of modules.
DvT	✚	The OCP supports the DvT theorem. While DvT aims to handle changes in data structures without impacting the system, OCP focuses on the extensibility of software entities. OCP does not explicitly address data versioning, and thus does not guarantee full convergence with DvT. However, by designing modules that follow OCP, developers can create components that are more adaptable to changes in data structures.
AvT	✚✚	The OCP converges with the AvT theorem. Both principles emphasize the importance of allowing changes or extensions to actions or operations without modifying existing implementations. By adhering to OCP, developers can create modules that can be extended to accommodate new actions or changes in existing ones, effectively achieving AvT.
SoS	━	The OCP has an indirect relationship with the Separation of States (SoSt) theorem. However, not on a level where we can speak of convergence. SoS emphasizes isolating different states within a system. Adhering to OCP alone does not guarantee full separation of states.

Table 5.2.: Converge OCP with NS

5.1.3. Converging the Liskov Substitution Principle

SoC	✚✚	Adhering to LSP in the software design leads to a more modular design and separation of specific concerns. Therefore we state that LSP converges with SoC. LSP states that objects of a derived class should be able to replace objects of the base class without affecting the correctness of the program. This can only be achieved by a strict separation of concerns in combination with Action version Transparent implementations of the signature.
DvT	➖	LSP has a limited alignment with the DvT theorem. While LSP focuses on the substitutability of objects in class hierarchies, DvT aims to handle changes in data structures without impacting the system. By following LSP, developers can ensure that derived classes can be substituted for their base classes, which may help reduce the impact of data structure changes on the system. However, LSP does not explicitly address data versioning and thus does not guarantee full convergence with DvT.
AvT	✚	The LSP supports the AvT theorem. Both principles emphasize the importance of allowing the extensibility of the system, without negatively impacting the desired requirements. By adhering to LSP, developers can create class hierarchies that can be easily extended to accommodate new actions or changes in existing ones, which may contribute to achieving AvT. However, adhering to LSP alone may not guarantee full convergence with AvT.
SoS	➖	By designing class hierarchies according to LSP, developers can create components that are less prone to side effects caused by shared states. However, the alignment between LSP and SoS is very weak, and adhering to LSP alone may not guarantee full separation of states.

Table 5.3.: Converge LSP with NS

5.1.4. Converging the Interface Segregation Principle

SoC	✚✚	The ISP converges with the SoC theorem, as both principles emphasize the importance of modularity and the separation of concerns. ISP states that clients should not be forced to depend on implementation they do not use, promoting the creation of smaller, focused interfaces. By adhering to SoC and designing target interfaces, inherently support SoC, leading to more evolvable software systems.
DvT	✚	The ISP has an indirect relationship with the DvT theorem. When adhering to the ISP, a developer can create a specific interface supporting a specific version of data entities. Although this approach can help minimize the impact of data structure changes on the system, it does not guarantee full DvT information system.
AvT	✚	The ISP supports the AvT theorem. Both principles emphasize the importance of separating actions within a system. ISP promotes the creation of interfaces for each (version of an) action, which aligns with the core concept of AvT. This alignment allows for a more manageable system, where changes in one action do not lead to ripple effects throughout the entire system. Although the adherence to ISP does not guarantee full compliance with the AvT
SoS	➖	There is no alignment between ISP and SoS. Mostly because ISP focuses on the separation of interfaces and abstract classes, while SoS emphasizes isolating different states within a system. This is an implementation concern. By no means the application of ISP could lead to a separation of state.

Table 5.4.: Converge ISP with NS

5.1.5. Converging the Interface Segregation Principle

SoC	✚✚	This principle converges with the SoC theorem. DIP states that high-level modules should not depend on low-level modules. When unavoidable, high-level modules should depend on abstractions of low-level modules and abstractions should not depend on details. By adhering to DIP correctly, developers can create modular and decoupled software systems, which aligns to break down a system into evolvable components. This convergence enables developers to create maintainable, scalable, and adaptable software systems that effectively manage complexity. Dependency Injection is a valuable (but not the only or mandatory) aspect of DIP. We have observed that the claim that the technique of Dependency Injection solves coupling between classes in an application is dangerous and in some cases wrong (Mannaert et al., 2016, p. 215). Nevertheless, this technique, when applied correctly (see 2.3.1), the artifact have pointed out that it has been a great asset in creating evolvable software, especially in the aspect of SoC.
DvT	✚	Adhering to the DIP can indirectly support the DvT theorem. By adhering to the DIP, developers can promote implementations that encourage modules to interact with each other only through well-defined interfaces or abstractions. This approach can help minimize the impact of data structure changes on the system but does not guarantee full compliance with DvT.
AvT	✚	The DIP can support the AvT theorem. Both principles emphasize the importance of isolating actions or operations within a system. By adhering to DIP, developers can create modular components that interact through abstractions, which may contribute to achieving AvT. However, the alignment between DIP and AvT less strong than with SoC, and adhering to DIP alone will not guarantee a system that entirely complies to AvT.
SoS	➖	There is a very weak alignment between DIP and SoS. Although Developers can create components that are less prone to side effects caused by a shared state, this can hardly be contributed to the DIP. Therefore it is stated that there is no convergence between the two principles.

Table 5.5.: Converge DIP with NS

5.2. Converging elements

In this section, we will apply a systematic cross-referencing approach to assess the level of convergence between each of the elements of both CA and NS. Along with a brief explanation, the level of convergence is denoted as followed:

Strong convergence	✚✚	Both elements have a high level of similarity or are closely related in terms of their purpose, structure, or functionality.
Some convergence	✚	Both elements have some similarities or share certain aspects in their purpose, structure, or functionality, but they are not identical or directly interchangeable.
No convergence	➖	The elements are not related or have no significant similarities in terms of their purpose, structure, or functionality.

5.2.1. Converging the Entity element

Data	✚	Both elements represent data objects that are part of the ontology or data schema of the application, and typically include attributes and relationship information. While both can contain a full set of attributes and relationships, the Data entity of NS may also include a specific set of information that is required for a single task or use case.
Task	➖	The Entity is not convergent with the Task element of NS. However, the Tasks element might operate on entities to perform business logic.
Flow	➖	The Entity and Flow are not convergent, as the Flow element represents the control between Tasks in NS, while the Entity in CA represents domain objects.
Connector	➖	The Entity element and Connector element are not convergent, as the Connector element in NS is involved in between components, while the Entity in CA represents domain objects.
Trigger	➖	The Entity element and Trigger element are not convergent, as the Trigger element in NS is about event-based execution of Tasks, while the Entity in CA represents domain objects.

Table 5.6.: The convergence of the Entity element

5.2.2. Converging the Interactor element

Data	−	The Interactor and Data elements are not convergent. However, Interactors might use Data elements as input and output during the execution of business logic.
Task	╋╋	The Task element in NS is very closely related to the Interactor element of CA, as both encapsulate the execution of business logic.
Flow	╋	The Interactor and Flow elements are partly convergent, as the Interactor orchestrates the flow of execution for a use case, which can involve multiple Tasks in NS.
Connector	−	The Interactor and Connector elements are not convergent. However, the Interactor might rely on connectors to communicate with other components in the system.
Trigger	−	The Interactor and Trigger elements are not convergent. However, the Interactors can be triggered by events or external requests, similar to NS Trigger elements.

Table 5.7.: The convergence of the Interactor element

5.2.3. Converging the RequestModel element

Data	✚	Both elements represent data objects that are part of the ontology or data schema of the application, and typically include attributes and relationship information. While they may contain a specific set of information as input for a Task or use case, both elements can also contain a full set of attributes and relationships. However, unlike the Data entity in NS, which may include only a subset of information necessary for a specific Task or use case, it may also include the full set of information required for Tasks other purposes.
Task	▬	The RequestModel is not convergent with the Task element of NS. However, the Tasks element might operate on RequestModels as input parameters to perform business logic.
Flow	▬	The RequestModel and Flow are not convergent, as the Flow element represents the control between Tasks in NS, while the RequestModel in CA represents (parts of) domain objects.
Connector	▬	The RequestModel element and Connector element are not convergent, as the Connector element in NS is involved in the communication between components, whilst the RequestModel in CA represents (parts of) domain objects.
Trigger	▬	The RequestModel element and Trigger element are not convergent, as the Trigger element in NS is about event-based execution of Tasks, while the RequestModel in CA represents (parts of) domain objects.

Table 5.8.: The convergence of the RequestModel element

5.2.4. Converging the ResponseModel element

Data	✚	Both elements represent data objects that are part of the ontology or data schema of the application, and typically include attributes and relationship information. While they may contain a specific set of information as output for a Task or use case, both elements can also contain a full set of attributes and relationships. However, unlike the Data entity in NS, which may include only a subset of information necessary for a specific Task or use case, it may also include the full set of information required for Tasks other purposes.
Task	▬	The ResponseModel is not convergent with the Task element of NS. However, the Tasks element might operate on RequestModels as input parameters to perform business logic.
Flow	▬	The ResponseModel and Flow are not convergent, as the Flow element represents the control between Tasks in NS, while the ResponseModel in CA represents (parts of) domain objects.
Connector	▬	The ResponseModel element and Connector element are not convergent, as the Connector element in NS is involved in communication between components, whilst the ResponseModel in CA represents (parts of) domain objects.
Trigger	▬	The ResponseModel element and Trigger element are not convergent, as the Trigger element in NS is about event-based execution of Tasks, while the ResponseModel in CA represents (parts of) domain objects.

Table 5.9.: The convergence of the ResponseModel element

5.2.5. Converging the ViewModel element

Data	✚	The ViewModel and Data element of NS is convergent to some degree. Both are involved in defining the structure of data used in the system. This could include required information about attributes and relationships. Additionally, the ViewModel could also represent information that is specifically intended for the representation of behavior for a user interface.
Task	▬	The ViewModel is not convergent with the Task element of NS. The ViewModel is focused on presenting information, whilst the Task element is concerned with executing business logic.
Flow	▬	The ViewModel and Flow are not convergent, as the Flow element represents the control between Tasks in NS and is not directly involved in the presentation of information.
Connector	▬	The ViewModel element and Connector element are not convergent, as the Connector element in NS is involved in the communication between components, whilst the ViewModel in CA is involved in the presentation of information.
Trigger	▬	The ViewModel element and Trigger element are not convergent, as the Trigger element in NS is responsible for the event-based execution of Tasks, whilst the ViewModel in CA is involved in the presentation of information.

Table 5.10.: The convergence of the ResponseModel element

5.2.6. Converging the Controller element

Data	−	The Controllers and Data elements are not directly convergent, as the Controller element is focused on handling input/output from external systems, while Data elements represent domain objects.
Task	−	The Controllers and Task elements are not convergent. However, controllers might initiate Task elements when handling incoming requests.
Flow	−	The Controllers and Flow elements are not convergent, as the Controller element is focused on handling input/output from external systems, whilst the Flow element is concerned with the orchestration of Tasks.
Connector	+	The Controller and Connector element are convergent to some degree. Both elements are involved in communication between components. The use of the Controller is a bit more strict it strictly defines communication from external parts of the systems, involving specific Interacto.
Trigger	+	The Controller and the Trigger element of NS are convergent to some degree as they both can initiate actions based on external events or requests. A Controller is primarily involved in receiving events or reqeusts from external sources, followed by the invocation of the appropriate interactor.

Table 5.11.: The convergence of the Controller element

5.2.7. Converging the Gateway element

Data	‑	The Gateway and Data element are not convergent. Nevertheless, the Gateway element might interact with data entities when providing access to external resources or systems.
Task	‑	The Gateway and Task element are not convergent. Nevertheless, the Task elements might use gateways when interacting with external resources or systems during the execution of business logic.
Flow	‑	The Gateway and Flow element are not directly related, as the Flow element represents the orchestration between Tasks in NS, whilst the Gateway element in CA provide access to external resources or systems.
Connector	╋╋	The Gateway and Connector element have a strong convergence, as both are involved in communication between components and provide interfaces for accessing external resources or systems.
Trigger	‑	The Gateway and Trigger element are not convergent, as the Trigger in NS is about event-based execution of Tasks, whilst the Gateway in CA provide access to external resources or systems.

Table 5.12.: The convergence of the Gateway element

5.2.8. Converging the Presenter element

Data	▬	The Presenter and Data element are not convergent. However, Data elements might be transformed into a suitable format for the user inter interface by CA's Presenter element.
Task	▬	The Presenter and Task elements are not convergent. The Presenter element is focused on transforming output data to the user interface, while the Task element of NS executes business logic.
Flow	▬	The Presenter and Flow element are not convergent. The Flow element of NS represents the orchestration between Tasks, while the Presenter element in CA is responsible for transforming output data to the user interface.
Connector	▬	The Presenter and Connector elements are not convergent. Although the Presenter element of CA might rely on Connector elements of NS to communicate with other components in the system.
Trigger	▬	The Presenter and Trigger elements are not convergent, as the Triggers element in NS is about event-based execution of Tasks, while presenters in CA are responsible for transforming output data on behalf of the user interface.

Table 5.13.: The convergence of the Presenter element

5.2.9. Converging the Boundary element

Data	▬	The Boundary and Data elements are not convergent, as the Boundary element is focused on separating concerns between components, while the Data element of NS represents a domain object.
Task	▬	The Boundary and Flow Task element are not convergent, However, the Boundary element can be used in a Task element to ensure a clear separation of concerns between different modules.
Flow	✚	The Boundary and Flow element are not convergent, However, the Boundary element can be used in a Flow element to ensure a clear separation of concerns between different modules.
Connector	✚✚	The Boundary and Connector elements have a strong convergence, as both are involved in communication between components and help ensure loose coupling between these components.
Trigger	▬	The Boundary and Trigger element are not convergent, as the Trigger in NS is about event-based execution of Tasks, whilst the Boundary in CA ensures the separation of concerns between components of the system.

Table 5.14.: The convergence of the Boundary element

6. Conclusions

1. **Literature Review**

   - Ca offers structure, principles and guidelines on how to build something. On top of that, NST also offers guidelines in order to apply actual changes.
     glsca has a strong emphasis on testability of code. Coupling is an important aspect on this.

   - SRP differs fundamentally from SoC in definition, although in the artifact not that much (further explain). Komt vooral door verschillende definities in granulariteit.

2. **Architectural Desing**

   -

3. **Artifact Development**

   -

   3.1. **The Code Generator and Clean Architecture Expander**

      -

   3.2. **Expanded Clean Architecture artifact**

      -

4. **Convergence Analysis:**

   -

   4.1. An analysis per principle of CA, compared with each of the principles of NS, indicating each level of convergence per principle

   4.2. An analysis per element of CA, compared with each of the elements of NS, indicating each level of convergence per principle

7. Reflections

In this Chapter, I will discuss my experiences and learnings from working with NS and CA during my research. I will use the "5-ways" framework to structure my reflections, providing insights into my thinking, managing, modeling, working, and supporting aspects of the developing approaches that were the topic of my research. Through this chapter, I hope to demonstrate the value of NS and its contribution to my knowledge in software engineering.

Way of Thinking

One of the aspects of NS is the characteristics of code generation. In my Job as a Domain Architect, I was involved in the development of software products based on the Model Driven Development (MDD) paradigm. My early experiences made me very sceptical about this approach. The theory of NS taught me to better understand the reasoning and characteristics of code generation, on which I then realized that my skepticism was more about the process caused as an effect on the implementation of the MDD. The knowledge of NS helped me to gain a more clear vision. This currently helps me push the roadmap on the MDD framework in the right direction.

Way of Managing

Way of Modeling

I considered multiple modeling languages in order to explain the implementation concepts of the artifact. One of which was the idea of using Archimate, but decided otherwise as I wanted to use a language that could be interpreted by a broader audience. I even considered just using *boxes and arrows*, but eventually decided on the UML2 standard as it is an official modeling language.

Way of Working

The topic of NS re-ignited my passion for software engineering and the aspects of designing and creating, -what I would previously mention as maintainable and qualitative. NS Taught me that it was in fact about software evolvability and stability. And on top of that, NS contributed greatly to my knowledge in doing so.

I very much enjoyed designing and creating the C# artifacts. In hindsight, I enjoyed it so much that I probably put in much more effort than was needed. This was also because I was very curious about the aspects of code generation, the effect of code generation on stable and evolvable artifacts and the characteristics of meta-circularity. I'm confident to say that I could have reached the same conclusions that are described in this Thesis, with a single, manually built Restful C# artifact.

The NS theorems are so nicely and abstractly formulated, that it ascends the domain of software engineering. During the masterclasses, we learned about applications of NS in the areas of Firewalls, Document management systems and Evolvable Business Processes. I experienced also benefits in structuring and maintaining my Thesis document using Visual Studio Code (VSCode) and Latex.

Way of Suporting

At the beginning of my research, I received a thorough introduction to the NS Theories and the Prime Radiant tooling from an employer at NSX. This introduction was extremely helpful in gaining a better understanding of the fundamentals of NS. It also inspired me to consider the code-generation aspects of the methodology, as well as the use of expanders, which are valuable for consistently delivering software artifacts with great precision. One thing and another has led to the decision to create the artifacts as described in this thesis.

For the writing of the Thesis, I decided to use Latex. I quickly discovered that Overleaf was one of the most popular editors. Nevertheless, I continued my search since I did not like the idea of being dependent on online tooling for writing my thesis. Although offline working is possible, doing so is very rudimentary without having the complete experience. At some point, I decided to experiment with my favorite code editor VSCode, and with the help of a latex package manager and some VSCode plugins I was able to create a fully-fledged Latex Editor in VSCode, with all it's benefits. There was even a plugin available that allowed me to use the spellchecker Grammarly while writing and modifying the .tex files.

Using Latex was a real eye-opener, and very relatable to my research as it allowed me to adhere to most of the NS principles while writing and maintaining my Thesis.

References

D. McIlroy. (1968). NATO SOFTWARE ENGINEERING CONFERENCE 1968.

De Bruyn, P., Mannaert, H., Verelst, J., & Huysmans, P. (2018). Enabling Normalized Systems in Practice – Exploring a Modeling Approach. *Business & Information Systems Engineering*, *60*(1), 55–67. https://doi.org/10.1007/s12599-017-0510-4

Dijkstra, E. (1968). Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, *11*(3), 147–148. https://doi.org/10.1145/362929.362947

Haerens, G. (2021). On the Evolvability of the TCP-IP Based Network Firewall Rule Base.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research.

Huysmans, P., & Verelst, J. (2013). Towards an Engineering-Based Research Approach for Enterprise Architecture: Lessons Learned from Normalized Systems Theory. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications* (pp. 58–72, Vol. 8827). Springer International Publishing. https://doi.org/10.1007/978-3-642-38490-5_5

Jacobson, I. (1992). *Object-oriented software engineering: A use case driven approach.* ACM Press ; Addison-Wesley Pub.

Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, *68*(9), 1060–1076. https://doi.org/10.1109/PROC.1980.11805

Mannaert, H., & Verelst, J. (2009). *Normalized systems re-creating information technology based on laws for software evolvability.* Koppa.
OCLC: 1073467550.

Mannaert, H., Verelst, J., & De Bruyn, P. (2016). *Normalized systems theory: From foundations for evolvable software toward a general theory for evolvable design.* nsi-Press powered bei Koppa.

Meyer, B. (1988). *Object-oriented software construction* (1st ed). Prentice Hall PTR.

Oorts, G., Huysmans, P., De Bruyn, P., Mannaert, H., Verelst, J., & Oost, A. (2014). Building Evolvable Software Using Normalized Systems Theory: A Case Study. *2014 47th Hawaii International Conference on System Sciences*, 4760–4769. https://doi.org/10.1109/HICSS.2014.585

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058. https://doi.org/10.1145/361598.361623

Robert C. Martin. (2018). *Clean architecture: A craftsman's guide to software structure and design.* Prentice Hall.
OCLC: on1004983973.

van Nuffel, D. (2011). Towards Designing Modular and Evolvable Business Processes, 424.

Wieringa, R. J. (2014). *Design Science Methodology for Information Systems and Software Engineering.* Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-43839-8

Wikipedia. (2023, March 25). Douglas McIlroy. In *Wikipedia*. Retrieved April 21, 2023, from https://en.wikipedia.org/w/index.php?title=Douglas_McIlroy&oldid=1146587956

Page Version ID: 1146587956.

References to Code Samples

Koks, G. (2023a). *AbstractExpander*. GitHub. https://github.com/LiquidVisions/Liquid
    idVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Gen
    erator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Expanders/
    AbstractExpander.cs

Koks, G. (2023b). *AbstractExpanderDependencyManagerInteractor*. https://github.com
    /LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b
    1b0536cb90c12c / Generator / src / PanthaRhei . Generator . Domain / Interactors /
    Generators/Expanders/AbstractExpanderDependencyManagerInteractor.cs

Koks, G. (2023c). *CodeGeneratorInteractor*. GitHub. https://github.com/LiquidVisio
    ns/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c
    12c/Generator/src/PanthaRhei.Generator.Application/Interactors/Generators/
    CodeGeneratorInteractor.cs

Koks, G. (2023d). *IExecutionInteractor*. GitHub. https://github.com/LiquidVisions/
    LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/
    Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/IExecuti
    onInteractor.cs

Koks, G. (2023e). *IExpanderHandlerInteractor*. GitHub. https://github.com/LiquidVi
    sions / LiquidVisions . PanthaRhei / blob / 9687c96eb368d96201d4baa66b1b0536cb
    90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/
    IExpanderHandlerInteractor.cs

Koks, G. (2023f). *IExpanderInteractor*. GitHub. https : / / github . com / LiquidVisions /
    LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/
    Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Expande
    rs/IExpanderInteractor.cs

Koks, G. (2023g). *IHarvesterInteractor*. https://github.com/LiquidVisions/LiquidVisio
    ns.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/
    src/PanthaRhei.Generator.Domain/Interactors/Generators/Harvesters/IHarves
    terInteractor.cs

Koks, G. (2023h). *InstallDotNetTemplateInteractor*. https://github.com/LiquidVisions/
    LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/
    Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Preproc
    essors/InstallDotNetTemplateInteractor.cs

Koks, G. (2023i). *IRejuvenatorInteractor*. https://github.com/LiquidVisions/LiquidVi
    sions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generat
    or / src / PanthaRhei . Generator . Domain / Interactors / Generators / Rejuvenator /
    IRejuvenatorInteractor.cs

Koks, G. (2023j). *PanthaRheiGeneratorDomain.* https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/tree/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Domain

Koks, G. (2023k). *PostProcessorInteractor.* GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/PostProcessors/PostProcessorInteractor.cs

Koks, G. (2023l). *PreProcessorInteractor.* GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Preprocessors/PreProcessorInteractor.cs

Koks, G. (2023m). *RegionHarvesterInteractor.* GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Harvesters/RegionHarvesterInteractor.cs

Koks, G. (2023n). *RegionRejuvenatorInteractor.* GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Rejuvenator/RegionRejuvenatorInteractor.cs

Koks, G. (2023o). *UnInstallDotNetTemplateInteractor.* https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/PostProcessors/UnInstallDotNetTemplateInteractor.cs

List of Figures

List of Tables

Glossary

comp The name of the Company that is considdered the owner of the software. If there is no company involved, this can be left blank..

CRUD An acronym that stands for Create, Read, Update, and Delete. It represents the basic operations required to manage persistent data in a database or software system..

Noun The primary subject or object that that class or interface is assosiated with..

Nuget.org NuGet is a free and open-source package manager for the Microsoft development platform, primarily targeting the .NET Framework. It unilizes third-party libraries into projects by providing a centralized platform for discovering, downloading, and managing dependencies..

prod The name of the product of the software..

tech The primary technology that is used by the component layer..

Verb The primary action that that class or interface is assosiated with..

Acronyms

AvT Action Version Transparency.

CA Clean Architecture.

CCP The Common Closure Principle.

CRP The Common Reuse Principle.

DIP Dependency Inversion Principle.

DLL Dynamic Link Library.

DvT Data Version Transparency.

ERD Entity Relation Diagram.

ISP Interface Segregation Principle.

LSP Liskov Substitution Principle.

MDD Model Driven Development.

NS Normalized Systems.

OCP Open/Closed Principle.

REP The Resuse/Release Equivalence Principle.

SoC Separation Of Concerns.

SOLID Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle.

SoS Separation of State.

SRP Single Responsibility Principle.

VSCode Visual Studio Code.

[title=Glossary of Terms]

A. Installing & using Pantha Rhei

A.1. Installation instructions

Step 1: Create output folder

Create an output folder so that the applications...

- ...has a location where to find the required expanders.

- ...has a location where the log files are stored.

- ...has a location where the result of the generation processes can be stored.

The location of the output folder is irrelevant.

Step 2: Create the Nuget configuration file

Add a configuration file named *nuget.config* file to the output folder with the the following content:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3      <packageSources>
4          <clear />
5          <add key="github" value="https://nuget.pkg.github.com/liquidvisions/index.json"
                />
6      </packageSources>
7          <packageSourceCredentials>
8          <github>
9              <add key="Username" value="" />
10             <add key="ClearTextPassword" value="{email me for a valid access token @
                    gerco.koks@outlook.com} " />
11         </github>
12     </packageSourceCredentials>
13 </configuration>
```

Listing A.1: The content of the Nuget configuration file

This config file is needed for the following step where the Pantha Rhei application is installed. The file contains the information to the private feed where the Pantha Rhei application can be downloaded and installed.

Step 3: Installing Pantha Rhei

Open a console in on the location where the Nuget configuration file is stored. The following command will download the package, and start the installation process which is executed in the background.

```
1 dotnet tool install LiquidVisions.PanthaRhei.Flux -g
```

Listing A.2: The install command

Step 4: Download & Install the Expanders

By clicking on the following link, an archived folder will be presented as a download by your browser. Download the archived folder and extract it on completion. Store the extracted folder, in a subfolder called *Expanders* in the root of the output folder. By doing so, the following folder structure should be available:

```
PanthaRhei.Output
├── Expanders
│     └── .Templates
└── nuget.config
```

The Pantha Rhei application is now ready for use.

Step 5: Setup a SQL database

Currently, Pantha Rhei is working with an MS SQL database for storing the model of the applications. Set up a SQL Database. This can either be a licensed version of SQL Server, the free-to-use SQL Express or an Azure SQL instance. See https://www.microsoft.com/en/sql-server/sql-server-downloads for more information. Make sure to have a valid connection string to the SQL server instance that is needed in step A.1 Step 6: Execute the command.

Step 6: Execute the command

Pantha Rhei is used by executing the *flux* command with the parameters as described in table A.1

```
1 flux --root "{folder}"
2      --mode Default
3      --app "{uniqueid}"
4      --db "{connectionstring}"
```

Listing A.3: Example command executing Pantha Rhei

–root	A mandatory parameter that should contain the full path to the output directory A Installing & using Pantha Rhei.
–db	A mandatory parameter that contains the connection string to the database.
–app	A mandatory parameter indicating the unique identifier of the application that should be generated.
–mode	An optional parameter that determines if a handler should be executed. *Default* is the default fallback mode (see A.2).
–reseed	An optional parameter that bypasses the expanding process. The model will be thoroughly cleaned and reseeded based on the entities of the expander artifact. This enables to a certain extent the meta-circularity and enables the expander artifact to generate itself.

Table A.1.: The *flux* command line parameters

RunModes are available to isolate the execution of the ExpanderHandler. It requires a current implementation shown in listing A.4. The following RunModes are available.

Default	This is the default generation mode that executes all configured handlers of the CleanArchitectureExpander. This will also install the required Visual Studio templates which are needed for scaffolding the Solution and C# Project files. Furthermore, it also executes the Harvest and Rejuvenation handlers. This mode will clean up the entire output folder prior after the Harvesting process is finished prior to the execution of the handlers.
Extend	This mode will skip the installation of the Visual Studio templates and the project scaffolding. It will not clean up the output folder but will overwrite any files handled. This mode is often less time-consuming and can be used in scenarios to quickly check the result of a part of the generation process.
Deploy	An optional mode that allows for expander handlers to run deployments in isolation. For example, when a developer wants to deploy the output to an Azure App Service.
Migrate	An optional mode that allows for expander handlers to run migrations in isolation. For example, this currently updates the database schema by running the Entity Framework Commandline Interface (see https://learn.microsoft.com/en-us/ef/core/cli/dotnet).

Table A.2.: The available *Generation modes*

```csharp
public class ExpandDatabaseContextHandlerInteractor
    : IExpanderHandlerInteractor<CleanArchitectureExpander>
{
    // ... other code
    public bool CanExecute =>
```

```
 6            parameters
 7                .GenerationMode
 8                .HasFlag(GenerationModes.Default) ||
 9            parameters
10                .GenerationMode
11                .HasFlag(GenerationModes.Extend);
12        // ... other code
13 }
```

Listing A.4: Example on how an expander handler can adhere to the RunMode parameters

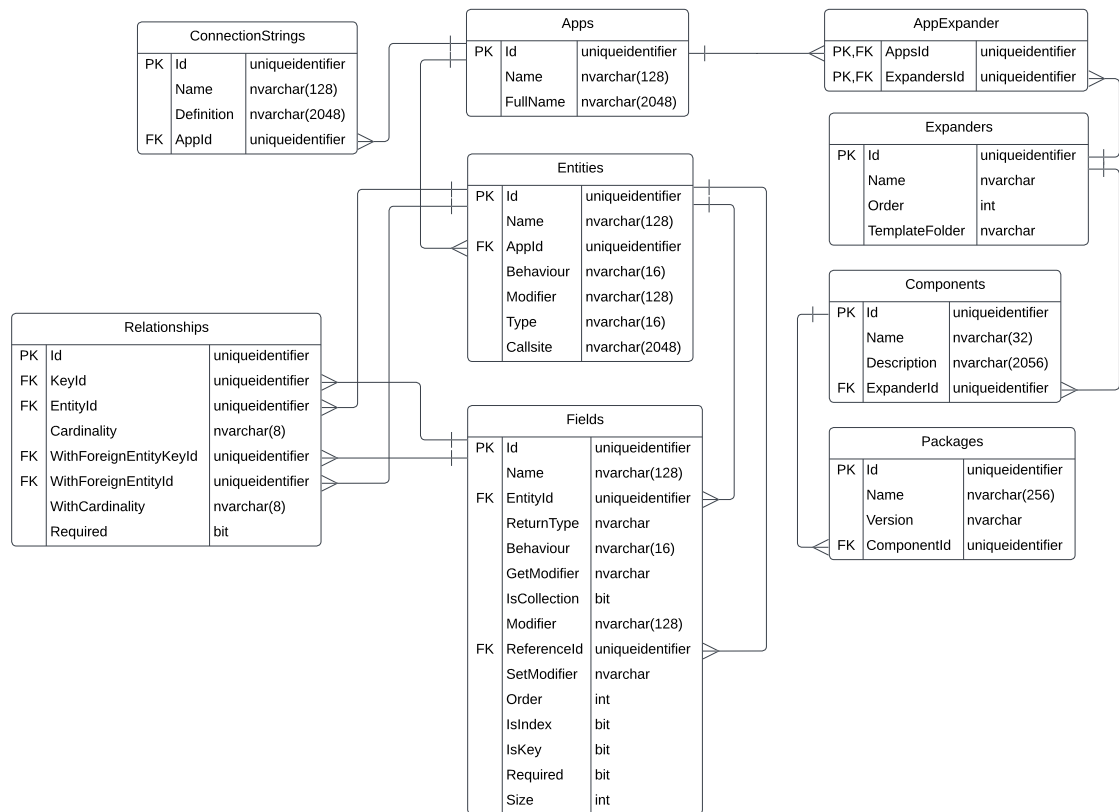# B. The Entity Relationship Diagram of the Meta Mode

Figure B.1.: Entity Relationship Diagram of the MetaModel

C. Designs & Architecture

C.1. Component Layer Naming Conventions

[**prod**] is defined as *The name of the product of the software.*
[**comp**] is defined as *The name of the Company that is considdered the owner of the software.*
If there is no company involved, this can be left blank.
[**tech**] is defined as *The primary technology that is used by the component layer.*

Layer	Project name	Package name
Domain	[prod].Domain	[comp].[prod].Domain
Application	[prod].Application	[comp].[prod].Application
Presentation	[prod].Presentation.[tech]	[comp].[prod].Presentation.[tech]
Infrastructure	[prod].Infrastructure.[tech]	[comp].[prod].Infrastructure.[tech]

Table C.1.: Naming convention component layers

C.2. Element Naming Conventions

[**Verb**] is defined as *The primary action that that class or interface is assosiated with.*
[**Noun**] is defined as *The primary subject or object that that class or interface is assosiated with.*

Layer name	Element	Type	Naming Convention
Presentation	Controller	class	[*Noun*]Controller
	ViewModelMapper	class	[*Noun*]ViewModelMapper
	Presenter	class	[*Verb*][*Noun*]Presenter
	ViewModel	class	[*Noun*]ViewModel
Application	Boundary	class	[*VerbNoun*]Boundary
	Boundary	interface	IBoundary
	Gateway	interface	I[*Verb*]Gateway
	Interactor	interface	I[*Verb*]Interactor
	Interactor	class	[*Verb*][*Noun*]Interactor
	Mapper	interface	IMapper
	RequestModelMapper	class	[*Verb*][*Noun*]RequestModelMapper
	Presenter	interface	IPresenter
	Validator	interface	IValidator
	Validator	class	[*Verb*][*Noun*]Validator
Infrastructure	Gateway	class	[*Noun*]Repository
Domain	Data Entity	class	[*Noun*]

Table C.2.: Naming convention of recurring elements

C.3. UML2 Notation Legenda

In order to visualize the designs of the artifact, a standard UML notation is used. The designs containing relationships adhere to the following definitions.
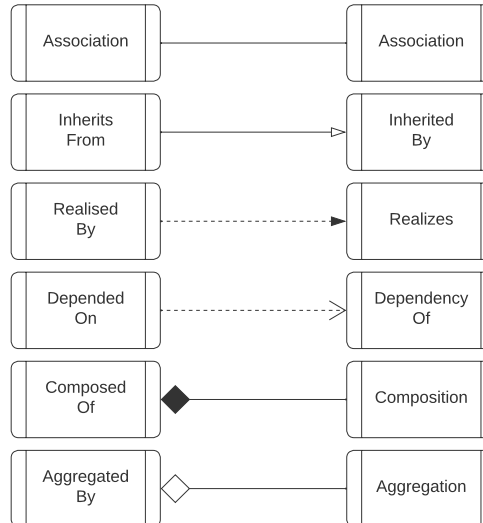

Figure C.1.: UML notation

D. Component Cohesion Principles

Name	Description
The Resuse/Release Equivalence Principle	REP is a concept related to software development that refers to the balance between reusing existing software components and releasing new ones to ensure the efficient use of resources and time (Robert C. Martin, 2018, p. 104).
The Common Closure Principle	In the context of Clean Architecture, the CCP states that classes or components that change together should be packaged together. In other words, if a group of classes is likely to be affected by the same kind of change, they should be grouped into the same package or module. This approach enhances the maintainability and modularity of the software (Robert C. Martin, 2018, p. 105).
The Common Reuse Principle	CRP states that classes or components that are reused together should be packaged together. It means that if a group of classes tends to be used together or has a high level of cohesion, they should be grouped into the same package or module. This approach aims to make it easier for developers to reuse components and understand their relationships (Robert C. Martin, 2018, p. 107).

Table D.1.: The Component Cohesion Principles

Cohesion facilitates the reduction of complexity and interdependence among the components of a system, thereby contributing to a more efficient, maintainable, and reliable system. By organizing components around a shared purpose or function or by standardizing their interfaces, data structures, and protocols, cohesion can offer the following benefits:

- **Reduce redundancy and duplication of effort**:
  Cohesion ensures that components are arranged around a common purpose or function, reducing duplicates or redundant code. This simplifies system comprehension, maintenance, and modification.

- **Promoting code reuse:**
  Cohesion facilitates code reuse by making it easier to extract and reuse components designed for specific functions. This saves time and effort during development and enhances overall system quality.

- **Enhance maintainability:**
  Cohesion decreases the complexity and interdependence of system components, making it easier to identify and rectify bugs or errors in the code. This improves system maintainability and reduces the risk of introducing new errors during maintenance.

- **Increase scalability:**
  Cohesion improves a system's scalability by enabling it to be extended or modified effortlessly to accommodate changing requirements or conditions. By designing well-organized and well-defined components, developers can easily add or modify functionality as needed without disrupting the rest of the system.