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

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

May 7, 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:	©(i)(o) 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/

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

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, 2023b). 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.
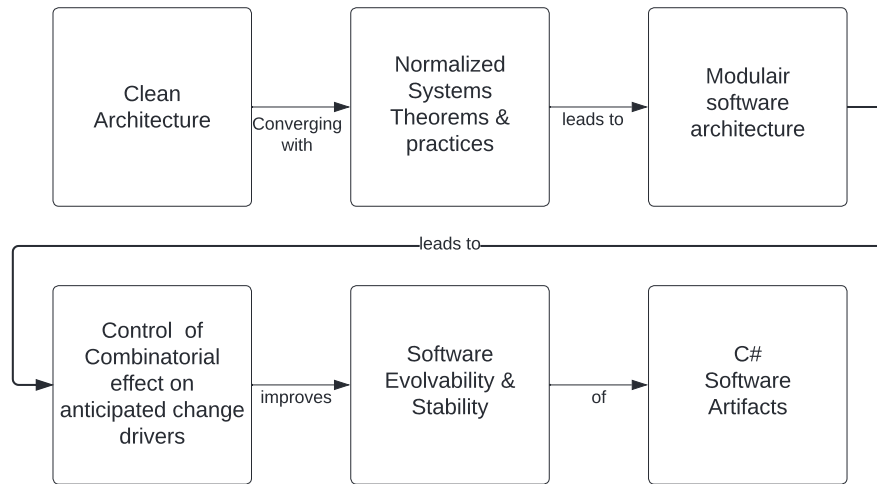
Figure 1.1.: The hypothesis

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.
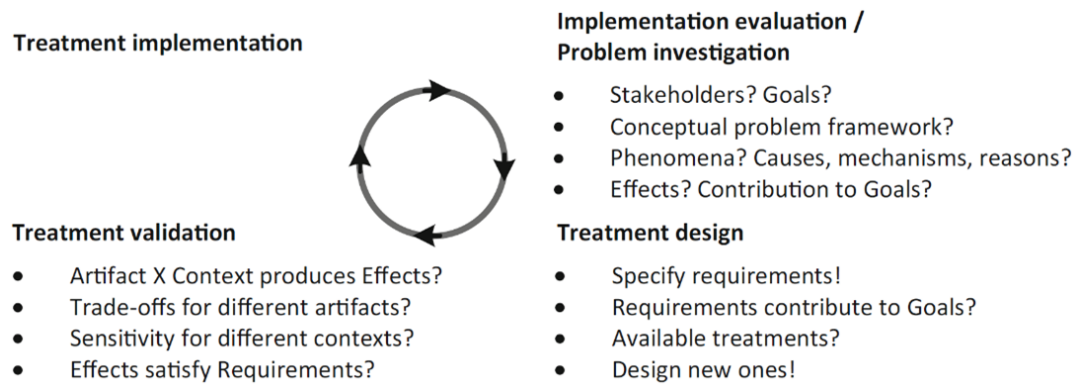
Figure 1.2.: The Engineering Cycle of Wieringa (2014)

In the context of this research, the artifacts described in chapters 4 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.
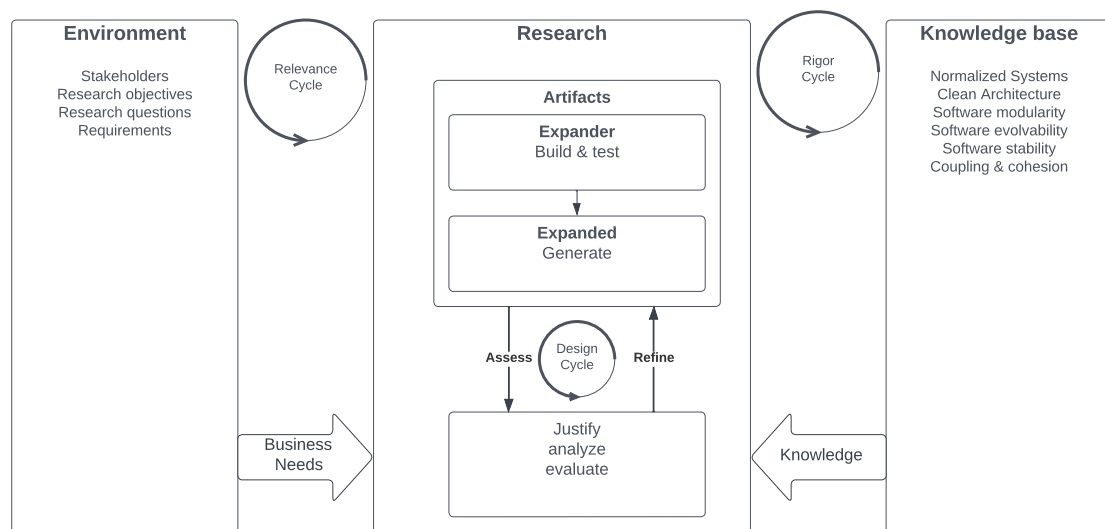
Figure 1.3.: The Design Science Framework for IS Research

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 thesis is organized into seven main chapters, beginning with the introduction. The introduction provides an overview of the study's research method and objectives and includes this Section of the Thesis Outline.

Chapter 2 focuses on the study's theoretical background, covering CA, NS, and some generic concepts. Chapter 3 is dedicated to the requirements for Software Transformation and the artifacts built as part of this study. Chapter 4 focuses on specific Artifact Design Decisions where Chapter 5 discusses the evaluation results of this study. We conclude with the conclusion in 6 and a personal reflection on the journey of this research in 6.3.

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 inter-related 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 E 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.
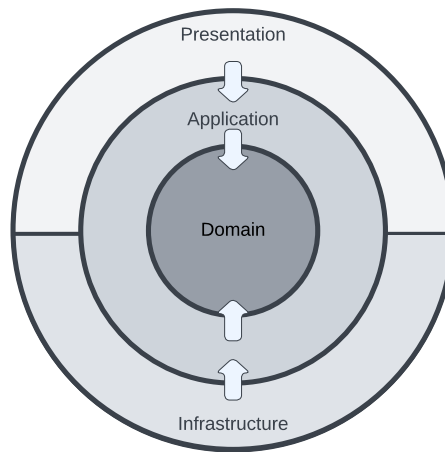
Figure 2.1.: Flow of control

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.	The Component Architecture Requirements
1.1	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
1.2	The Visual Studio projects representing the component layers comply with the naming conventions outlined in the appendix D.1 Component Layer Naming Conventions
1.3	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 Expander Requirements
2.1	The Domain and Application layers have no dependencies on any infrastructure technologies, like web- or database technologies.
2.2	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.
2.3	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.
2.4	All Component Layers utilize the C# programming language, explicitly targeting the .NET 7.0 framework.
2.5	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.: Technology Expander Requirements

15

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 D.2. In addition, the following tables detail the requirements specific to each element.
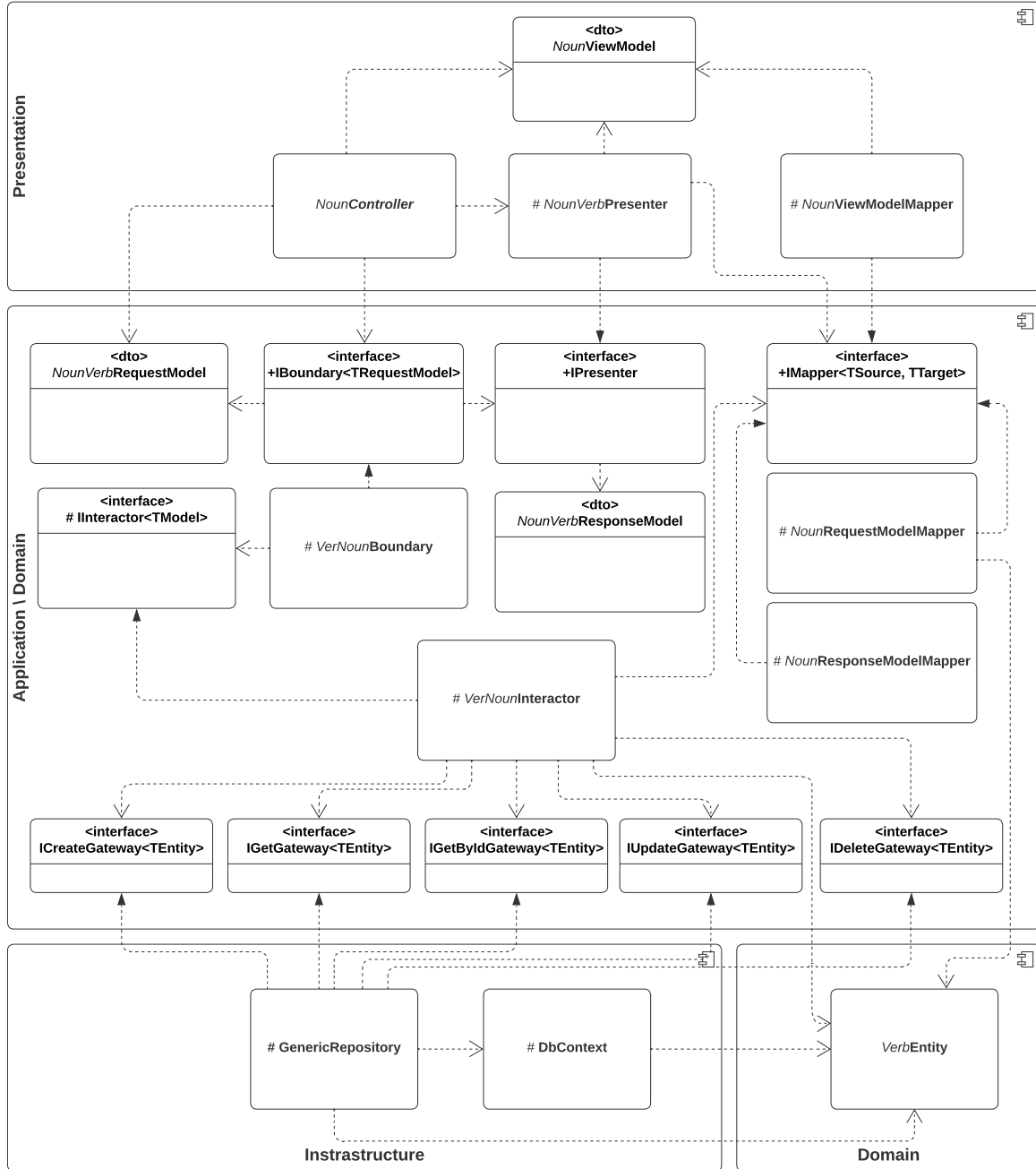

Figure 3.1.: The Generic architecture of the artifacts

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

Table 3.3.: Technology Requirements

Nr.	Presenter Requirements
4.1	The Presenter Implementation is derived from the IPresenter interface and follows the specified implementation. The IPresenter interface can be found in the Application Layer.
4.2	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.
4.3	When required, the Presenter utilizes the IMapper interface without depending on specific implementations of the IMapper interface.
4.4	The Presenter has an internal scope and cannot be instantiated outside of the Presentation layer.

Table 3.4.: Presenter Requirements

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

Table 3.5.: ViewModelMapper Requirements

Nr.	Controller Requirements
6.1	The Controller is responsible for receiving external requests and forwarding the request to the appropriate Boundary within the Application Layer.
6.2	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
7.1	The IBoundary interface establishes the contract for its derived Boundary implementations.
7.2	The IBoundary interface has public scope within the system.

Table 3.7.: IBoundary Requirements

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

Table 3.8.: Boundary Implementation Requirements

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

Table 3.9.: IInteractor Requirements

Nr.	Interactor Implementation Requirements
10.1	An Interactor implementation is derived from the IInteractor interface and follows the specified implementation.
10.2	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.
10.3	Depending on the Task, the Interactor implementation orchestrates the mapping from RequestModels to Entities, or from Entities to ResponseModels, utilizing the IMapper interface.
10.4	Interactor implementations have an internal scope and cannot be implemented outside the Application Layer.

Table 3.10.: Interactor Implementation Requirements

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

Table 3.11.: IMapper Requirements

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

Table 3.12.: RequestModelMapper Requirements

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

Table 3.13.: ResponseModelMapper Requirements

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

Table 3.14.: IPresenter Requirements

19

Nr.	Gateway Requirements
15.1	The Domain and Application layers have no dependencies on any infrastructure technologies, like web- or database technologies.
15.2	The *[Verb]Gateway* interface establishes the contract for its derived Gateway implementations, which are typically implemented in the Infrastructure Layer.
15.3	The *[Verb]*Gateway interface has a public scope within the system.
15.4	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.: Gateway Requirements

Nr.	ResponseModel Requirements
16.1	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.
16.2	The ResponseModel does not depend on external objects within the architecture.

Table 3.16.: ResponseModel Requirements

Nr.	RequestModel Requirements
17.1	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.
17.2	The RequestModel does not depend on external objects within the architecture.

Table 3.17.: RequestModel Requirements

Domain Layer

Nr.	Data Entity Requirements
18.1	The Data Entity consists solely of attributes representing the corresponding data fields.
18.2	The Data Entity does not rely on external objects within the architecture.
18.3	The Application Layer is the only layer that utilizes the Data Entity.

Table 3.18.: Data Entity Requirements

The Infrastructure Layer

Nr.	Gateway Implementation Requirements
19.1	The [*Verb*]Gateway Implementation derives from the I[*Verb*]Gateway interface and adheres to the specified implementation.
19.2	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).
19.3	The [*Verb*]Gateway Implementation has an internal scope and cannot be instantiated outside of the Layer.

Table 3.19.: Gateway Implementation Requirements

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 Framework & Clean Architecture 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
20.1	The Expander Framework enables interaction with the Clean Architecture Expander via a Command Line Interface (CLI). The CLI is implemented in the Presentation Layer of the Expander Framework.
20.2	The Expander Framework retrieves the model from a Micrsoft SQL Database (MSSQL) using the EntityFramework ORM technology. The EntityFramework technology is implemented in the Infrastructure Layer of the Expander Framework.
20.3	The Expander Framework loads and executes the configured Expanders. In the case of this research, only the Clean Architecture Expander is applied.
20.4	The Expander Framework supports generic harvesting and rejuvenation, which can be used or extended by the Expanders using the OCP principle.
20.5	The Expander Framework supports generic template handling, which can be used or extended by the Expanders using the OCP principle.
20.6	The Expander framework adheres to this chapter's Component and Software Requirements specified in Sections 3.2.1 and 3.2.2.

Table 3.20.: Expander Framework Requirements

Nr.	Clean Architecture Expander Requirements
21.1	The Clean Architecture Expander generates a C# net7.0 RESTful service that provides an HTTP interface on top of the metamodel of the Expander Framework, allowing the basic CRUD operations.
21.2	The Clean Architecture Expander consists solely of an Application Layer and reuses the Domain Layer of the Expander Framework.
21.3	The Clean Architecture Expander adheres to this chapter's Component and Software Requirements specified in Sections 3.2.1 and 3.2.2.

Table 3.21.: Clean Architecture Expander Requirements

3.2.4. Generated Artifact Requirements

Nr.	The Generated Artifact Requirements
22.1	The Generated artifact adheres to this chapter's Component and Software Requirements specified in Sections 3.2.1 and 3.2.2.

Table 3.22.: The Generated Artifact Requirements

4. Artifact Design Decisions.

Chapter 3 Requirements outlines all the requirements of the artifacts, which aim to ensure compliance and adherence to the Clean Architecture Design and SOLID principles. This chapter will discuss specific design decisions made to meet the required functionality while adhering to the requirements outlined in Chapter 3.

4.1. The Artifact name and use

The name of the Expander Framework, Pantha Rhei, was inspired by the Greek philosopher *Heraclitus*, who famously stated that "life is flux". The name reflects the artifact's ability to cope with constant change in a stable and evolvable manner. The name is also reflected in the use of the "flux" command in the CLI, which allows users to interact with the application.

To install Pantha Rhei, interested readers can follow the instructions provided in the appendix B Installing & using Pantha Rhei.

4.2. The meta-model and model

The meta-model is a blueprint that describes a software system's structure, entities, relationships, and expanders. The model is an instantiation of the meta-model, representing a specific software system with unique characteristics.

In this research, the model represents the elements, relationships, and characteristics of the meta-model. The aim is to achieve meta-circularity, although it is not fully implemented according to NS Theory. However, by combining the model with the Clean Architecture Expander, users can modify and extend the model and the meta-model to adapt to changing requirements.

Figure 4.1 illustrated the version of the meta-model used for this research. A detailed description of each of the elements can be found in Appendix C The Entity Relationship Diagram of the Meta Mode
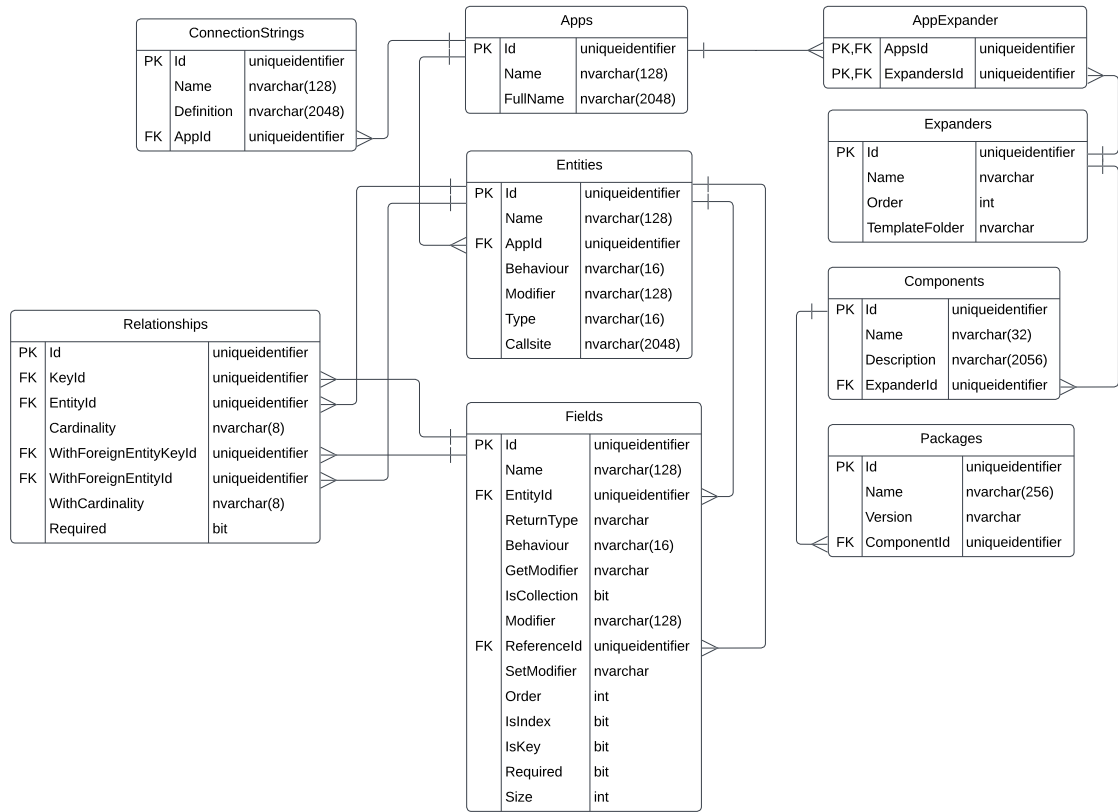
ConnectionStrings

PK	Id	uniqueidentifier
	Name	nvarchar(128)
	Definition	nvarchar(2048)
FK	AppId	uniqueidentifier

Apps

PK	Id	uniqueidentifier
	Name	nvarchar(128)
	FullName	nvarchar(2048)

AppExpander

PK,FK	AppsId	uniqueidentifier
PK,FK	ExpandersId	uniqueidentifier

Expanders

PK	Id	uniqueidentifier
	Name	nvarchar
	Order	int
	TemplateFolder	nvarchar

Entities

PK	Id	uniqueidentifier
	Name	nvarchar(128)
FK	AppId	uniqueidentifier
	Behaviour	nvarchar(16)
	Modifier	nvarchar(128)
	Type	nvarchar(16)
	Callsite	nvarchar(2048)

Components

PK	Id	uniqueidentifier
	Name	nvarchar(32)
	Description	nvarchar(2056)
FK	ExpanderId	uniqueidentifier

Relationships

PK	Id	uniqueidentifier
FK	KeyId	uniqueidentifier
FK	EntityId	uniqueidentifier
	Cardinality	nvarchar(8)
FK	WithForeignEntityKeyId	uniqueidentifier
FK	WithForeignEntityId	uniqueidentifier
	WithCardinality	nvarchar(8)
	Required	bit

Packages

PK	Id	uniqueidentifier
	Name	nvarchar(256)
	Version	nvarchar
FK	ComponentId	uniqueidentifier

Fields

PK	Id	uniqueidentifier
	Name	nvarchar(128)
FK	EntityId	uniqueidentifier
	ReturnType	nvarchar
	Behaviour	nvarchar(16)
	GetModifier	nvarchar
	IsCollection	bit
	Modifier	nvarchar(128)
FK	ReferenceId	uniqueidentifier
	SetModifier	nvarchar
	Order	int
	IsIndex	bit
	IsKey	bit
	Required	bit
	Size	int

Figure 4.1.: The meta-model represented as an Entity Relationship Diagram

4.3. Plugin Architecture

The Expander Framework Artifact is responsible for loading and bootstrapping Expanders and initiating the generation process. Expanders are dynamically loaded at runtime through a dotnet capability called assembly binding, allowing the architecture illustrated in the following image (Koks, 2023l).
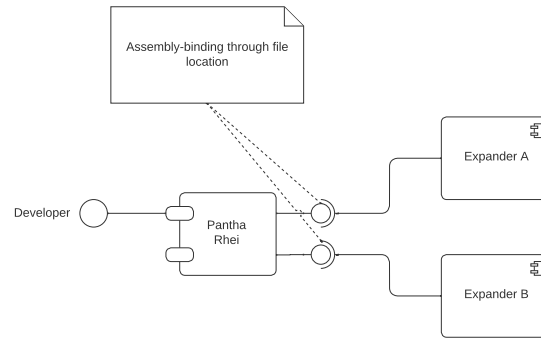
Figure 4.2.: Expanders are considered plugins

This plugin design adheres to several principles of SOLID. The Single Responsibility Principle (SRP) principle is implemented by ensuring that an expander generates one and only one construct. The OCP principle is applied by allowing the creation of new expanders in addition to the already existing ones. The LSP principle is respected by enabling the addition or replacement of expanders without modifying the internal workings of the Expander Framework.

More details can be found in the Appendix A.1 The ExpanderPluginLoaderInteractor

4.4. Expanders

check this

The requirement for an expander is to have an implementation of the IExpanderInteractor (Koks, 2023s) interface, as displayed in Figure 4.3. Although it is not mandatory, it is recommended to use the abstract AbstractExpander (Koks, 2023a) class, which provides a full Expander experience.
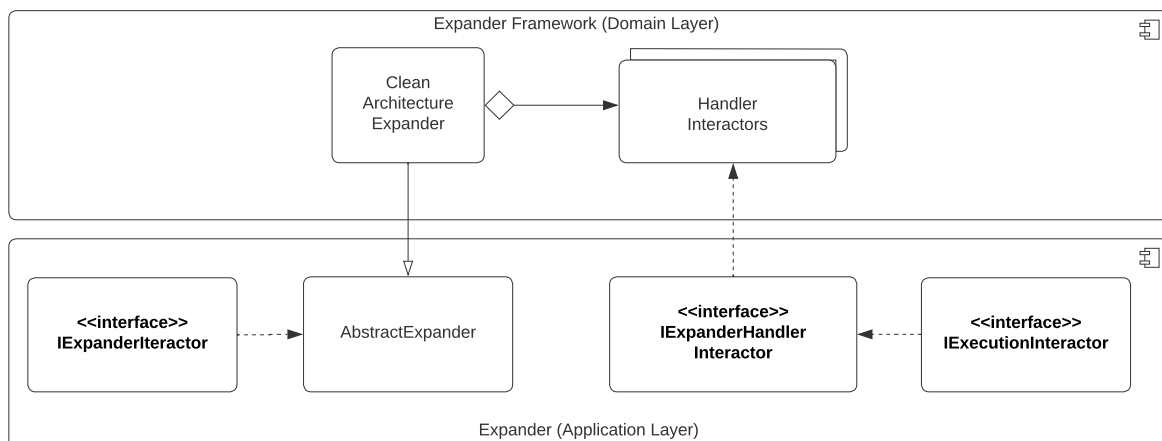

Figure 4.3.: The Design of an Expander

4.5. The IExecutionInteractor command

An exciting implementation that facilitates a high degree of cohesion while maintaining low coupling is the utilization of the *IExecutionInteractor* interface (Koks, 2023r). This interface allows for the execution of various derived types responsible for specific tasks, such as executing Handlers, Harvesters, and Rejuvenators (Koks, 2023k, 2023x, 2023y). The implementation promotes decoupling by adhering to both OCP and LSP.
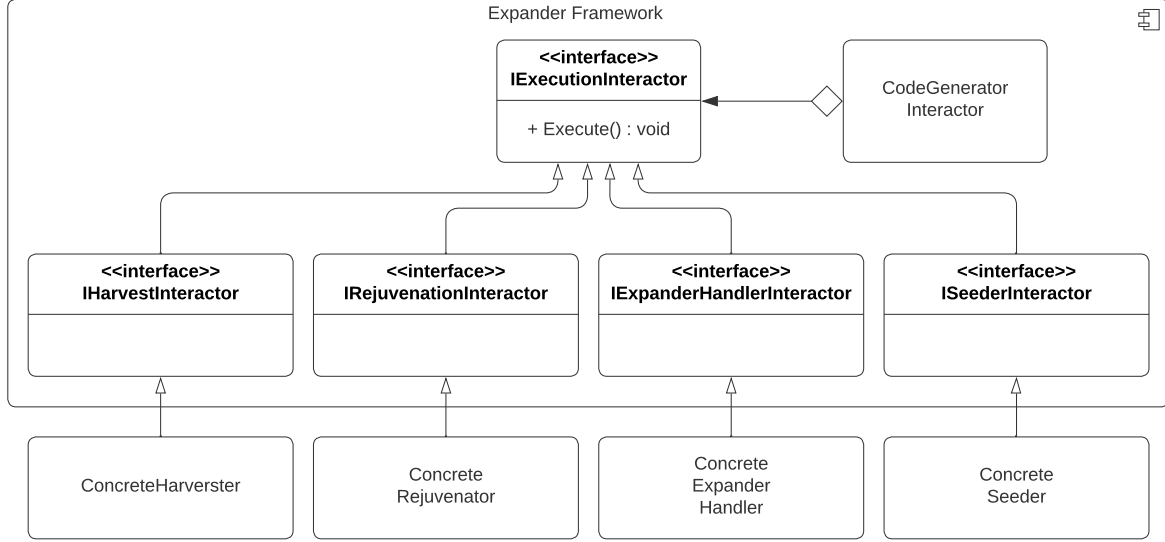

Figure 4.4.: Low coupling with *IExecutionInteractor*

Figure 4.4 illustrates that the required interfaces are placed in the Domain Layer of the Expander Framework. In contrast, the concrete classes also can be implemented as part of the internal scope of the Clean Architecture Expander (Koks, 2023w). Code listing A.3 The ExpandEntitiesHandlerInteractor illustrates an implementation example of this interface. Finally, the code listing A.2 The CodeGeneratorInteractor illustrates the aggregation of the execution, which allows for a graceful cohesion of the execution Tasks (Koks, 2023b).

4.6. Dependency management

Dependency management is an extremely valuable aspect of achieving stability and evolvability. Dependency management can be achieved by using Dependency Injection. This research acknowledges the statement of Mannaert et al. (2016, p. 215) that Dependency Injection does not solve coupling between classes. Working on the Artifact has shown that combinatorial effects can occur when not careful. Nevertheless, Dependency Injection is a widely used pattern in building the Artifact. In order to achieve stability and evolvability, the Dependency Injection pattern <u>must</u> be combined with various other principles of both CA and NS.

The goal is to centralize the management of dependencies and remove unwanted manual object instantiations in the code. Al this while respecting the DIP principle so that each Component Layer is responsible for managing its dependencies. The Artifact achieves this by using extension methods as illustrated in Code Listing A.9 (Koks, 2023h). Additionally, and quite significantly, implementations primarily rely on abstractions or contracts (interfaces) instead of the details of concrete implementations.

Traditionally, Dependency Injection injects instantiations through constructor parameters or class properties. Although there are benefits in this approach, doing so will eventually lead to combinatorial effects, breaking the stability of a Software Artifact. In order to solve this problem, the Artifact used the Service Locator pattern, a central registry responsible for resolving dependencies (Wikipedia, 2023a). Many frameworks are available from Nuget.org, but the Artifact uses the Service Registry that is part of the .NET framework. This service registry is considered a cross-cutting concern. The dependency on this technology is reduced by applying the principles of the LSP and ISP. The Artifact creates and uses separate interfaces to register (Koks, 2023q) and resolve (Koks, 2023p) dependencies. As illustrated in Code Listing A.10, the framework technology dependency is abstracted behind implementing those interfaces (Koks, 2023i).

Practically every class gets the IDependencyFactoryInteractor (Koks, 2023p) injected, on which further resolving is responsible for that class's inner workings. Code Listing A.11 illustrates how this is done in the AbstractExpander (Koks, 2023a) class. Finally, all the dependencies are bootstrapped on application bootup, depicted in Code Listing A.12.

The approach described here has many advantages in managing the stability and evolvability of the Software Artifact. However, as for most things, there are also some drawbacks. For example, a good amount of experience is required for developers to understand code that incorporates abstractions, contracts, and Dependency Injection. Another drawback is that dependency errors are detected in runtime rather than compile time. The benefits of the Artifacts, however, outweigh the drawbacks.

5. Analysis results

This chapter will analyze the two development approaches, CA and NS. We will examine how these approaches align and affect software architecture on the Artifact. First, in Section 5.1, we will compare the principles of CA with the principles of NS. Then, we will compare the design elements in section 5.2. We will showcase real-world examples from the Artifacts to illustrate their practical manifestations.

5.1. An Analysis of Priniples

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

Strong alignment	✚✚	This indicates that the principles of CA and NS are highly aligned. Both have a similar impact on the design and implementation of the artifact.
Supports alignment	✚	The CA principle supports in implementing the NS principle through specific design choices. However, applying the CA principle does not inherently ensure adherence to the corresponding NS principle.
No or weak alignment	➖	The principles are not aligned or have no significant similarities in terms of their purpose, goals, or architectural supports

5.1.1. Single Responsibility Principle

SoC	✚✚	The main goal of both SRP and SoC is to promote and encourage modularity, low coupling, and high cohesion. While the definition has some differences, the two principles can be regarded as practically interchangeable. Many examples in the Artifacts show a strong alignment between SRP and SoC. To name one, an Expander should be able to can perform multiple Tasks to complete the full instantiation of the Model. Each of those Tasks can be implemented separately from each other. Figure 5.1 illustrated some of the Tasks that are implemented in the Clean Architecture Expander Artifact. The Code Listing A.4 is an example of one implementation of such a Task ExpandEntitiesHandlerInteractor (Koks, 2023k). To sum up, SRP and SoC share the goal of organizing a software system into modular components with specific 'responsibilities' or 'concerns'. This convergence highlights the importance of encapsulating these responsibilities within a software system.
DvT	✚	Although using SRP does not implicitly guarantees DvT, it does support DvT by directing certain design choices. For example, both CA and NS assign specific DTO objects to support specific use cases (Interactors or Tasks) or to transfer (parts of) Data between architectural layers. CA specifically assigned DTOs and guidelines on where and when to use them. These are also applied in the Artifact of this study as ResponseModels, RequestModels, and ViewModels (Koks, 2023z, 2023ab) (TODO: Response citation toevoegen). The separation of data structures specific to Use Cases minimizes the impact of data structure changes by preferring stamp coupling over data coupling. However, SRP is not a guaranteed measure for DvT.
AvT	✚	While SRP emphasizes limiting the responsibility of each module, it does not explicitly require handling specific versions of use cases. Nevertheless, adhering to glssrp can still indirectly contribute to achieving AvT. One way to achieve this is by separating versions of Actions into separate contracts, objects, or methods, enabling Action Version transparency to some degree. Although not yet available in the Artifact, the Code Listing A.5 shows that API versioning is a common standard practice and fully supported by the open API specifcation and the .net core framework (Github, 2023a; OAS, 2023). Manifestations in the Artifact can be located in the Logger (Code Listing A.6), amongst others (2023v).
SoS	➖	Following SRP might lead to separate modules that manage their state, indirectly contributing to SoS. However, the alignment is very weak, and no manifestations are found in the artifacts.

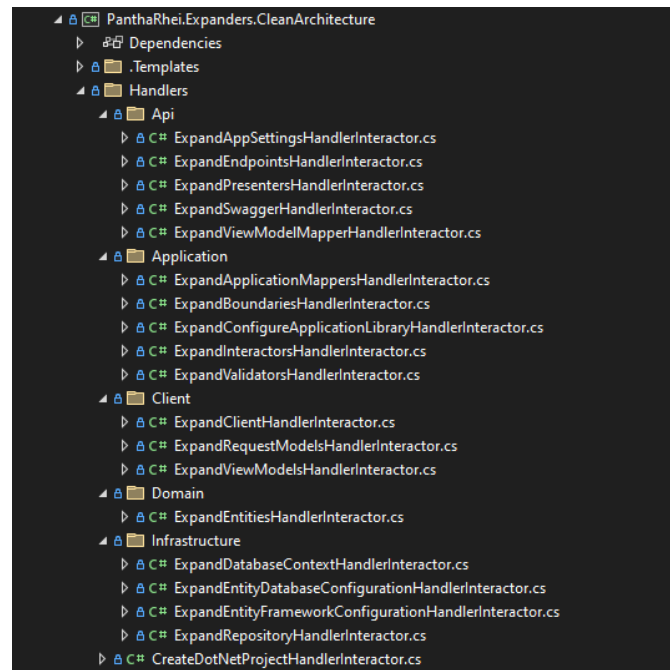Table 5.1.: The alignment of SRP with the NS principles

Figure 5.1.: Each of the handlers handles an isolated part of the expanding process.

5.1.2. Open/Closed Principle

SoC	✚✚	The OCP is strongly aligned with the SoC principle of NS. OCP states that software architectures should be open for extension but closed for modification. When applying OCP correctly, the architecture supports new requirements built as an extension, affecting as few existing implementations as possible. Conversely, adhering to SoC does not guarantee the adherence of OCP, as SoC focuses on modularization and encapsulation rather than the extensibility of functionality. The same example with the Tasks provided in sub-section 5.1.1 is also an excellent manifestation of this principle.
DvT	✚	While DvT aims to handle changes in data structures without impacting the system, OCP focuses on the extensibility of software architectures. Although OCP does not explicitly address data version transparency, OCP promotes architectures that are more adaptable to changes in data structures. Separating DTOs specific for a use case is a great example and manifestation in the Artifact (Koks, 2023z, 2023ab). However, there is no complete alignment because the use of OCP does not guarantee DvT.
AvT	✚✚	The OCP is strongly aligned with the AvT principle of NS, as both principles emphasize the importance of allowing changes or extensions to actions without affecting existing implementations. OCP is also closely related to SRP. Besides SRP, OCP have the most manifestations in the Artifact, some of which are already mentioned in previous examples.
SoS	━	The OCP indirectly relates to the SoS principle. The alignment of both principles is weak, and no manifestations are found in the artifacts.

Table 5.2.: The alignment of OCP with the NS principles

5.1.3. Liskov Substitution Principle

SoC	✚✚	LSP states that objects of a derived class should be able to replace objects of the base class without affecting the program negatively. Replacing objects can only be achieved by separating them, aligning the principles inherritly. A good example is the implementation of the ITemplateInteractor (Koks, 2023t) where the template engine Scriban (Github, 2023b) is used to generate code instantiations as a result of the Expanding the Model (Koks, 2023aa). We could easily replace the Scriban template engine for an other engine with only impacting the Dependency Injection Register.
DvT	➖	The alignment between LSP and DvT is weak, and no manifestations are found in the artifacts.
AvT	✚	The LSP supports the AvT principle. Both principles emphasize the importance of allowing the extensibility of the software system. By adhering to LSP, the architecture allows for class hierarchies that can be easily extended to accommodate new (versions of) actions, which can contribute to achieving AvT. However, adhering to LSP alone may not guarantee full adherence with AvT. Considder ICreateGateway (Koks, 2023o) in Code Listing A.7. The artifact contains multiple implementations of this interface. Each implementation could be considered a different version applied to the interface.
SoS	➖	The LSP does not relate to the SoS principle. The alignment of both principles is weak, and no manifestations are found in the artifacts.

Table 5.3.: The alignment of LSP with the NS principles

5.1.4. Interface Segregation Principle

SoC	✚✚	The ISP strongly aligns with the SoC principle, as both 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. In Listing A.8, you can see that each CRUD operation has its own interface (Koks, 2023f).
DvT	➖	The ISP does not relate to the DvT principle. The alignment of both principles is weak, and no manifestations are found in the artifacts.
AvT	✚	The alignment between ISP and AvT arises from the emphasis of ISP on creating targeted interfaces that are tailored to specific needs. Smaller interfaces can enhance modularity and minimize unwanted side effects when modifying Actions in the software system, positively impacting the implementation of the AvT. For example, modifications in Actions are likely to have a limited impact. However, adhering to ISP is not a guarantee for AvT.
SoS	➖	The ISP does not relate to the SoS principle. The alignment of both principles is weak, and no manifestations are found in the artifacts.

Table 5.4.: The alignment of ISP with the NS principles

5.1.5. Dependency Inversion Principle

SoC	✚	DIP states that high-level modules should not depend on low-level modules. By adhering to DIP correctly, the architecture promotes modular architectures and the use of component layers, as described in 2.3.4 The Dependency rule (Koks, 2023u). Managing Dependencies inheritly promotes SoC, therefore DIP aligns with SoC to some extend. However, adhering to SoC does not guarantee SoC.
DvT	➖	The DIP does not relate to the DvT principle. The alignment of both principles is weak, and no manifestations are found in the artifacts.
AvT	✚	The DIP can support the AvT principle. The AvT principle emphasizes the importance of isolating actions or operations within a system. By adhering to DIP, the architecture simplifies the dependency management of those isolated versions of actions, which may contribute to achieving AvT. The artifact's handling of this is already described in Chapter 2.3.1 The Dependency Inversion Principle. However, the alignment between DIP and AvT is not so strong as with SoC, and adhering to DIP alone will not guarantee a system that entirely complies to AvT.
SoS	➖	The DIP does not relate to the SoS principle. The alignment of both principles is weak, and no manifestations are found in the artifacts.

Table 5.5.: The alignment of DIP with the NS principles

5.2. An analysis of Elements

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

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

5.2.1. 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 Element of NS may also be tailored to serve a specific set of information that is required for a single task or use case. In CA these type of Data Elements are specified explicitly as ViewModels, RequestModels or ResponseModels. Code Listing A.13 illustrates the similarities of between an CAs Entity and the Data Element from NS
Task	➖	The Entity has no alignment with the Task element of NS. However, the Tasks element might operate on entities to perform business logic.
Flow	➖	The Entity and Flow elements are not aligned, as the Flow element represents the control between Tasks in NS, while the Entity in CA represents domain objects.
Connector	➖	The Entity and Connector element from NS are not aligned, as the Connector element in NS is involved in the communication separation of 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 alignment of the Element Entity with the elements of NS

5.2.2. The Interactor Element

Data	➖	The Interactor and Data element do not align, although an Interactor 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. Code Listing A.14 illustrates an implementation of a Task, with a Single execution of a business rule (2023e).
Flow	➕➕	The Interactor has a strong alignment with the Flow element of NS as the both elements can orchestrates the flow of execution for a use case, which can involve multiple Tasks in NS. This is clearly illustrated in Code Listing A.15, where the Interactor handles Validation, mapping and persistance of the Entity (2023c).
Connector	➖	There is no alignment between the Interactor and the Connector Element of NS.
Trigger	➖	There is no alignment between the Interactor and the Trigger Element of NS.

Table 5.7.: The alignment of the Element Interactor with the elements of NS

5.2.3. 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. As illustrated in Code Listing A.16, they can 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 (Koks, 2023g).
Task	➖	There is no alignment between the RequestModel and the Task Element of NS.
Flow	➖	There is no alignment between the RequestModel and the Flow Element of NS.
Connector	➖	There is no alignment between the RequestModel and the Connector Element of NS.
Trigger	➖	There is no alignment between the RequestModel and the Trigger Element of NS.

Table 5.8.: The alignment of the Element RequestModel with the elements of NS

35

5.2.4. 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. They can 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.
Task	━	There is no alignment between the ResponseModel and the Task Element of NS.
Flow	━	There is no alignment between the ResponseModel and the Flow Element of NS.
Connector	━	There is no alignment between the ResponseModel and the Connector Element of NS.
Trigger	━	There is no alignment between the ResponseModel and the Trigger Element of NS.

Table 5.9.: The alignment of the Element ResponseModel with the elements of NS

5.2.5. 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 alignment of the Element ViewModel with the elements of NS

36

5.2.6. 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 Interactor.
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 alignment of the Element Controller with the elements of NS

5.2.7. 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 alignment of the Element Gateway with the elements of NS

5.2.8. The Presenter Element

Data	━	There is no alignment between the Presenter and the Task Element of NS.
Task	✚	The Presenter is responsible for in preparing the Response and View-Model on behalf of the Controller and can be considdered a Task Element with a very specialized job. Code Listing A.17 illustrated the inner workings of a Presenter (Koks, 2023d)
Flow	✚	Presenters can handle multiple Tasks when this is required. in this case there is also some alignment between the Presenter Element of CA with the Flow Element of NS.
Connector	━	There is no alignment between the Presenter and the Connector Element of NS.
Trigger	━	There is no alignment between the Presenter and the Trigger Element of NS.

Table 5.13.: The alignment of the Element Presenter with the elements of NS

5.2.9. 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 alignment of the Element Boundary with the elements of NS

Evaluating the findings

	SoC	DVT	AVT	SoS
SRP	++	+	+	−
OCP	++	−	++	−
LSP	++	−	+	−
ISP	++	+	+	−
DIP	++	+	+	−

Table 5.15.: Convergence between the SOLID and NS principles

	Data	Task	Flow	Connector	Trigger
Entity	++	−	−	−	−
Interactor	−	++	+	−	−
RequestModel	+	−	−	−	−
ResponseModel	+	−	−	−	−
ViewModel	+	−	−	−	−
Controller	−	−	−	+	+
Gateway	−	−	−	++	−
Presenter	−	−	−	−	−
Boundary	−	−	+	++	−

Table 5.16.: Convergence between the SOLID and NS elements

6. Conclusions and discussion

This thesis culminates a multifaceted exploration into the convergence of CA with NS. We have drawn upon the author's firsthand experience in designing Software Architectures used rigorous theoretical research and created a practical and working Software Artifact. The primary objective was to investigate the alignment between CA and NS, by analyzing their principles and design elements through theory and practice. This Chapter will summarize the findings into a research conclusion.

6.1. Conclusion

A noteworthy distinction between NS and CA lies in their foundational roots. NS is a product of computer science research built upon formal theories and principles derived from rigorous scientific investigation. Although, throughout this thesis, NS is referred to as a development approach, it is a part of Computer Science.

Stability and evolvability are concepts not directly referenced in the literature of CA, but very much aligns with the goal of Mannaert et al. (2016, p. 31). The attentive reader surely observes the shared emphasis on modularity and the separation of concerns. Both approaches attempt to achieve low coupling and high cohesion. In addition, CA add the dimensions of dependency management as an essential measure to improve maintainability and manage modularity.

A critical difference between CA and NS lies in their approach to handling state. CA does not explicitly address state management in its principles or design elements. At the same time, NS provides the principle of Separation of State, ensuring that state changes within a Software System are stable and evolvable. This principle can be crucial in developing scalable and high-performance systems, as it isolates state changes from the rest of the system, reducing the impact of state-related dependencies and side effects. This finding leads to the first conclusion of this thesis.

1. The alignment between CA and NS is incomplete because CA needs specific state management principles. As a result, CA cannot fully ensure stable and evolvable software artifacts as defined by NS.

6.2. Discussion

Nevertheless, NS and CA can be effectively applied to enhance the stability and evolvability of Software Systems. This synergetic relationship stems from the fact that NS contributes a solid

theoretical foundation for understanding and addressing the combinatorial effects of system changes. At the same time, CA offers a robust, practical framework for architectural design in software development.

By integrating the theoretical underpinnings of NS with the pragmatic principles of CA, Software Architectures can harness both approaches' strengths to create stable software systems and adaptable to changing requirements. In addition, the convergence of these methodologies allows for a comprehensive understanding of the factors contributing to reducing the complexity of the Software Architecture, enabling developers to make informed design decisions prioritizing modularity, scalability, and extensibility, leading to the last conclusion of this thesis.

The synergistic application of NS and CA can significantly improve software systems' stability and evolvability. It leverages the advantages of both approaches—combining NSs rigorous theoretical basis with CAs practical guidelines for architectural design. This fusion of methodologies offers a powerful, holistic approach to software development that addresses the challenges of system complexity and fosters adaptability in an ever-evolving technological landscape.

2. The alignment between CA and NS is incomplete because CA needs specific state management principles. As a result, CA cannot fully ensure stable and evolvable software artifacts as defined by NS.

6.3. Bridging the Gap of doing and learning

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.
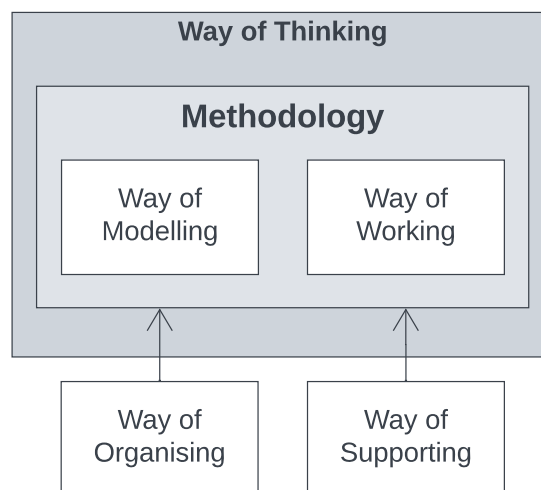

Figure 6.1.: The Five Way Framework, inspired by Dietz and Mulder (2020)

6.3.1. 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.

6.3.2. Way of Managing

6.3.3. 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.

6.3.4. 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.

6.3.5. 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.

Bibliography

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

Dietz, J. L. G., & Mulder, H. B. F. (2020). *Enterprise ontology: A human-centric approach to understanding the essence of organisation*. Springer International Publishing. https://doi.org/10.1007/978-3-030-38854-6

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

Web References

Github. (2023a). *Aspnet-api-versioning/Program.cs at main · dotnet/aspnet-api-versioning*. Retrieved May 5, 2023, from https://github.com/dotnet/aspnet-api-versioning/blob/main/examples/AspNetCore/WebApi/MinimalApiExample/Program.cs

Github. (2023b). *Scriban*. https://github.com/scriban/scriban

OAS. (2023). *Versioning an API*. Google Cloud. Retrieved May 5, 2023, from https://cloud.google.com/endpoints/docs/openapi/versioning-an-api

Wikipedia. (2023a). Service locator pattern. In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Service_locator_pattern&oldid=1152702717

Wikipedia. (2023b, 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.

Code Samples

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

Koks, G. (2023b). *CodeGeneratorInteractor*. GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Application/Interactors/Generators/CodeGeneratorInteractor.cs

Koks, G. (2023c). *CreateEntityInteractor*. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/PanthaRhei.Output/Output/6c6984a1-c87a-429b-b91f-2a976adb3c0e/LiquidVisions.PanthaRhei.Generated/src/Application/Interactors/Entities/CreateEntityInteractor.cs

Koks, G. (2023d). *CreateEntityPresenter*. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/PanthaRhei.Output/Output/6c6984a1-c87a-429b-b91f-2a976adb3c0e/LiquidVisions.PanthaRhei.Generated/src/Presentation.Api/Presenters/Entities/CreateEntityPresenter.cs

Koks, G. (2023e). *CreateEntityValidator*. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/PanthaRhei.Output/Output/6c6984a1-c87a-429b-b91f-2a976adb3c0e/LiquidVisions.PanthaRhei.Generated/src/Application/Validators/Entities/CreateEntityValidator.cs

Koks, G. (2023f). *CRUDGateways*. GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/tree/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Domain/Gateways

Koks, G. (2023g). *DeleteEntityRequestModel*. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/PanthaRhei.Output/Output/6c6984a1-c87a-429b-b91f-2a976adb3c0e/LiquidVisions.PanthaRhei.Generated/src/Application/RequestModels/Entities/DeleteEntityRequestModel.cs

Koks, G. (2023h). *DependencyInjectionExtension*. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Application/DependencyInjectionExtension.cs

Koks, G. (2023i). *DependencyManagerInteractor*. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Domain/Interactors/Dependencies/DependencyManagerInteractor.cs

Koks, G. (2023j). *Entity*. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/PanthaRhei.Output/Output/6c6984a1-c87a-

429b-b91f-2a976adb3c0e/LiquidVisions.PanthaRhei.Generated/src/Domain/Entities/Entity.cs

Koks, G. (2023k). *ExpandEntitiesHandlerInteractor*. GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Expanders/src/PanthaRhei.Expanders.CleanArchitecture/Handlers/Domain/ExpandEntitiesHandlerInteractor.cs

Koks, G. (2023l). *ExpanderPluginLoaderInteractor*. Retrieved May 1, 2023, from https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Application/Interactors/Initializers/ExpanderPluginLoaderInteractor.cs

Koks, G. (2023m). *GenericRepository*. GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Infrastructure.EntityFramework/Repositories/GenericRepository.cs

Koks, G. (2023n). *HarvestRepository*. GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Infrastructure/HarvestRepository.cs

Koks, G. (2023o). *ICreateGateway*. GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Gateways/ICreateGateway.cs

Koks, G. (2023p). *IDependencyFactoryInteractor*. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Domain/Interactors/Dependencies/IDependencyFactoryInteractor.cs

Koks, G. (2023q). *IDependencyManagerInteractor*. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Domain/Interactors/Dependencies/IDependencyManagerInteractor.cs

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

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

Koks, G. (2023t). *ITemplateInteractor*. Retrieved May 5, 2023, from https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Templates/ITemplateInteractor.cs#L6

Koks, G. (2023u). *Layers of the CodeGenerator*. GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/tree/master-thesis-artifact/Generator/src

Koks, G. (2023v). *Logger*. GitHub. https://github.com/LiquidVisions/LiquidVisions. PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src /PanthaRhei.Generator.Infrastructure/Logging/Logger.cs

Koks, G. (2023w). *MigrationHarvesterInteractor*. https://github.com/LiquidVisions/ LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Expanders/src/Pantha Rhei.Expanders.CleanArchitecture/Harvesters/MigrationHarvesterInteractor.cs

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

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

Koks, G. (2023z). *RequestModels*. https://github.com/LiquidVisions/LiquidVisions. PanthaRhei/tree/master-thesis-artifact/PanthaRhei.Output/Output/6c6984a1- c87a-429b-b91f-2a976adb3c0e/LiquidVisions.PanthaRhei.Generated/src/Applic ation/RequestModels/Apps

Koks, G. (2023aa). *ScribanTemplateInteractor*. https://github.com/LiquidVisions/ LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/ Generator/src/PanthaRhei.Generator.Application/Interactors/Templates/Scrib anTemplateInteractor.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..

DTO DTO stands for Data Transfer Object. It is a design pattern used in software development that involves simple objects for transferring data between layers or processes within an application. They are often lightweight, and have no business logic, serving primarily as a container for data to be transferred..

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..

SOLID An acronym that stands for a set of design principles composed by Robert C. Martin. The five principles that comprise SOLID are: Single Responsibility Principle, Open/-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle.

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.

CLI Command Line Interface.

CRP The Common Reuse Principle.

DIP Dependency Inversion Principle.

DvT Data Version Transparency.

ISP Interface Segregation Principle.

LSP Liskov Substitution Principle.

MDD Model Driven Development.

MSSQL Micrsoft SQL Database.

NS Normalized Systems.

OCP Open/Closed Principle.

REP The Resuse/Release Equivalence Principle.

SoC Separation Of Concerns.

SoS Separation of State.

SRP Single Responsibility Principle.

VSCode Visual Studio Code.

[title=Glossary of Terms]

A. Code listings

```
1  internal class ExpanderPluginLoaderInteractor : IExpanderPluginLoaderInteractor
2  {
3      \\...other code
4
5      /// <inheritdoc/>
6      public void LoadAllRegisteredPluginsAndBootstrap(App app)
7      {
8          foreach (Expander expander in app.Expanders)
9          {
10             string rootDirectory = Path.Combine(expandRequestModel.ExpandersFolder,
                   expander.Name);
11             string[] files = directoryService.GetFiles(rootDirectory, searchPattern,
                   SearchOption.TopDirectoryOnly);
12             if (!files.Any())
13             {
14                 throw new InitializationException($"No plugin assembly detected in '{
                       rootDirectory}'. The plugin assembly should match the following '{
                       searchPattern}' pattern");
15             }
16
17             LoadPlugins(files)
18                 .ForEach(assembly => BootstrapPlugin(expander, assembly));
19
20         }
21     }
22
23     \\...other code
24
25     private List<Assembly> LoadPlugins(string[] assemblyFiles)
26     {
27         List<Assembly> plugins = new();
28
29         foreach (string assemblyFile in assemblyFiles)
30         {
31             try
32             {
33                 Assembly assembly = LoadPlugin(assemblyFile);
34                 plugins.Add(assembly);
35             }
36             catch (Exception innerException)
37             {
38                 throw new InitializationException($"Failed to load plugin '{
                       assemblyFile}'.", innerException);
39             }
40         }
41
42         return plugins;
43     }
44
45     private Assembly LoadPlugin(string assemblyFile)
46     {
47         Assembly assembly = assemblyContext.Load(assemblyFile);
48         logger.Trace($"Plugin context {assemblyFile} has been successfully loaded...");
49         return assembly;
50     }
```

```
51
52      private void BootstrapPlugin(Expander expander, Assembly assembly)
53      {
54          Type bootstrapperType = assembly.GetExportedTypes()
55              .Where(x => x.IsClass && !x.IsAbstract)
56              .Single(x => x.GetInterfaces()
57              .Contains(typeof(IExpanderDependencyManagerInteractor)));
58
59          IExpanderDependencyManagerInteractor expanderDependencyManager = (
60              IExpanderDependencyManagerInteractor)activator
61              .CreateInstance(bootstrapperType, expander, dependencyManager);
62          expanderDependencyManager.Register();
63      }
64  }
```

Listing A.1: The ExpanderPluginLoaderInteractor

```
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 A.2: The CodeGeneratorInteractor

```
1   public class ExpandEntitiesHandlerInteractor
2       : IExpanderHandlerInteractor<CleanArchitectureExpander>
3   {
4       // ... other code
5
6       /// <inheritdoc/>
7       public void Execute()
8       {
9           directory.Create(entitiesFolder);
10
11          foreach (var entity in app.Entities)
12          {
13              string fullSavePath = Path.Combine(
14                  entitiesFolder,
15                  $"{entity.Name}.cs"
16              );
17
18              templateService.RenderAndSave(
19                  pathToTemplate,
```

```
20                    new { entity },
21                    fullSavePath
22                );
23            }
24        }
25 }
```

Listing A.3: The ExpandEntitiesHandlerInteractor

```
1  public class ExpandEntitiesHandlerInteractor
2      : IExpanderHandlerInteractor<CleanArchitectureExpander>
3  {
4      // ... other code
5
6      /// <inheritdoc/>
7      public void Execute()
8      {
9          directory.Create(entitiesFolder);
10
11         foreach (var entity in app.Entities)
12         {
13             string fullSavePath = Path.Combine(
14                 entitiesFolder,
15                 $"{entity.Name}.cs"
16             );
17
18             templateService.RenderAndSave(
19                 pathToTemplate,
20                 new { entity },
21                 fullSavePath
22             );
23         }
24     }
25 }
```

Listing A.4: The ExpandEntitiesHandlerInteractor

```
1  var forecast = app.NewVersionedApi();
2
3  // GET /weatherforecast?api-version=1.0
4  forecast.MapGet( "/weatherforecast", () =>
5          {
6              return Enumerable.Range( 1, 5 ).Select( index =>
7                  new WeatherForecast
8                  (
9                      DateTime.Now.AddDays( index ),
10                     Random.Shared.Next( -20, 55 ),
11                     summaries[Random.Shared.Next( summaries.Length )]
12                 ) );
13         } )
14         .HasApiVersion( 1.0 );
15
16 // GET /weatherforecast?api-version=2.0
17 var v2 = forecast.MapGroup( "/weatherforecast" )
18                  .HasApiVersion( 2.0 );
```

Listing A.5: The Aspnet-Api-Versioning/Program.Cs at Main · Dotnet/Aspnet-Api-Versioning

```
1  internal class Logger : ILogger
2  {
3      /// <summary>
4      /// Writes the message at the trace level.
5      /// </summary>
6      /// <param name="message">The message that needs to be logged.</param>
7      public void Trace(string message)
```

```
 8      {
 9          Trace(message, null);
10      }
11
12      /// <summary>
13      /// Writes the diagnostic message at the Trace level using the specified
           expandRequestModel.
14      /// </summary>
15      /// <param name="message">A string containing format items.</param>
16      /// <param name="args">Arguments to format.</param>
17      public void Trace(string message, params object[] args)
18      {
19          internalLogger.Trace(message, args);
20      }
21 }
```

Listing A.6: The Logger

```
 1 internal class HarvestRepository : ICreateGateway<Harvest>
 2 {
 3      // other code
 4
 5      public bool Create(Harvest entity)
 6      {
 7          if(string.IsNullOrEmpty(entity.HarvestType))
 8          {
 9              throw new InvalidProgramException("Expected harvest type.");
10          }
11
12          string fullPath = Path.Combine(
13              expandRequestModel.HarvestFolder,
14              app.FullName,
15              $"{file.GetFileNameWithoutExtension(entity.Path)}.{entity.HarvestType}");
16
17          serializer.Serialize(entity, fullPath);
18
19          return true;
20      }
21
22      // other code..
23 }
24
25 internal class GenericRepository<TEntity> : ICreateGateway<TEntity>
26 {
27      // other code ...
28
29      public bool Create(TEntity entity)
30      {
31          context.Set<TEntity>().Add(entity);
32          context.Entry(entity).State = EntityState.Added;
33
34          return context.SaveChanges() >= 0;
35      }
36
37      // other code ...
38 }
```

Listing A.7: Partial examples of implementations of the ICreateGateway (Koks, 2023o) in the GenericRepository (Koks, 2023m) and the HarvestRepository (Koks, 2023n)

```
 1 // Create
 2 public interface ICreateGateway<in TEntity>
 3      where TEntity : class
 4 {
 5      bool Create(TEntity entity);
 6 }
```

```
 7
 8  // Read
 9  public interface IGetGateway<out TEntity>
10      where TEntity : class
11  {
12      IEnumerable<TEntity> GetAll();
13
14      TEntity GetById(object id);
15  }
16
17  // Update
18  public interface IUpdateGateway<in TEntity>
19      where TEntity : class
20  {
21      bool Update(TEntity entity);
22  }
23
24  // Delete
25  public interface IDeleteGateway<in TEntity>
26      where TEntity : class
27  {
28      bool Delete(TEntity entity);
29
30      bool DeleteAll();
31
32      bool DeleteById(object id);
33  }
34
35  internal class AppSeederInteractor : IEntitySeederInteractor<App>
36  {
37      private readonly ICreateGateway<App> createGateway;
38      private readonly IDeleteGateway<App> deleteGateway;
39      private readonly Parameters parameters;
40
41      public AppSeederInteractor(IDependencyFactoryInteractor dependencyFactory)
42      {
43          createGateway = dependencyFactory.Get<ICreateGateway<App>>();
44          deleteGateway = dependencyFactory.Get<IDeleteGateway<App>>();
45          parameters = dependencyFactory.Get<Parameters>();
46      }
47
48      public int SeedOrder => 1;
49
50      public int ResetOrder => 1;
51
52      public void Seed(App app)
53      {
54          app.Id = parameters.AppId;
55          app.Name = "PanthaRhei.Generated";
56          app.FullName = "LiquidVisions.PanthaRhei.Generated";
57
58          createGateway.Create(app);
59      }
60
61      public void Reset() => deleteGateway.DeleteAll();
62  }
```

Listing A.8: The Gateways for Create, Read, Update, Delete operations

```
1  public static class DependencyInjectionExtension
2  {
3      /// <summary>
4      /// Adds the dependencies of the project to the dependency inversion object.
5      /// </summary>
6      /// <param name="services"><seealso cref="IServiceCollection"/></param>
7      /// <returns>An instance of <seealso cref="IServiceCollection"/>.</returns>
```

```
8    public static IServiceCollection AddApplicationLayer(this IServiceCollection
         services)
9    {
10       return services.AddTransient<ICodeGeneratorBuilderInteractor,
             CodeGeneratorBuilderInteractor>()
11           .AddTransient<IEntitiesToSeedGateway, EntitiesToSeedGateway>()
12           .AddTransient<ICodeGeneratorInteractor, CodeGeneratorInteractor>()
13           .AddInitializers()
14           .AddSeedersInteractors()
15           .AddBoundaries()
16           .AddTemplateInteractors();
17   }
18
19   private static IServiceCollection AddTemplateInteractors(this IServiceCollection
         services)
20   {
21       services.AddTransient<ITemplateInteractor, ScribanTemplateInteractor>()
22           .AddTransient<ITemplateLoaderInteractor, TemplateLoaderInteractor>();
23
24       return services;
25   }
26
27   private static IServiceCollection AddInitializers(this IServiceCollection services)
28   {
29       return services.AddTransient<IExpanderPluginLoaderInteractor,
             ExpanderPluginLoaderInteractor>()
30           .AddTransient<IAssemblyContextInteractor, AssemblyContextInteractor>()
31           .AddTransient<IAssemblyContextInteractor, AssemblyContextInteractor>()
32           .AddTransient<IExpanderPluginLoaderInteractor,
                 ExpanderPluginLoaderInteractor>()
33           .AddTransient<IObjectActivatorInteractor, ObjectActivatorInteractor>();
34   }
35
36   private static IServiceCollection AddBoundaries(this IServiceCollection services)
37   {
38       return services.AddTransient<IExpandBoundary, ExpandBoundary>()
39           .AddTransient<ISeederInteractor, SeederInteractor>();
40   }
41
42   private static IServiceCollection AddSeedersInteractors(this IServiceCollection
         services)
43   {
44       services.AddTransient<IEntitySeederInteractor<App>, AppSeederInteractor>()
45           .AddTransient<IEntitySeederInteractor<App>, ExpanderSeederInteractor>()
46           .AddTransient<IEntitySeederInteractor<App>, EntitySeederInteractor>()
47           //.AddTransient<ISeederInteractor<App>, PackageSeederInteractor>()
48           .AddTransient<IEntitySeederInteractor<App>, FieldSeederInteractor>()
49           .AddTransient<IEntitySeederInteractor<App>, ComponentSeederInteractor>()
50           .AddTransient<IEntitySeederInteractor<App>,
                 ConnectionStringsSeederInteractor>()
51           .AddTransient<IEntitySeederInteractor<App>, RelationshipSeederInteractor>()
                 ;
52
53       return services;
54   }
55 }
```

Listing A.9: Bootstrapping dependencies in DependencyInjectionExtension (Koks, 2023h)

```
1 /// <summary>
2 /// The <see cref="IServiceCollection">dependency container</see>.
3 /// </summary>
4 internal class DependencyManagerInteractor : IDependencyFactoryInteractor,
     IDependencyManagerInteractor
5 {
6     private readonly IServiceCollection serviceCollection;
```

```
 7      private IServiceProvider provider;
 8
 9      /// <summary>
10      /// Initializes a new instance of the <see cref="DependencyManagerInteractor"/>
            class.
11      /// </summary>
12      /// <param name="serviceCollection">The <see cref="IServiceCollection"/>.</param>
13      public DependencyManagerInteractor(IServiceCollection serviceCollection)
14      {
15          this.serviceCollection = serviceCollection;
16      }
17
18      /// <inheritdoc/>
19      public void AddTransient(Type serviceType, Type implementationType)
20      {
21          serviceCollection.AddTransient(serviceType, implementationType);
22      }
23
24      /// <inheritdoc/>
25      public IDependencyFactoryInteractor Build()
26      {
27          provider = serviceCollection.BuildServiceProvider();
28
29          return this;
30      }
31
32      /// <inheritdoc/>
33      public IEnumerable<T> GetAll<T>()
34      {
35          if (provider == null)
36          {
37              Build();
38          }
39
40          return provider.GetServices<T>();
41      }
42
43      /// <inheritdoc/>
44      public T Get<T>()
45      {
46          if (provider == null)
47          {
48              Build();
49          }
50
51          return provider.GetRequiredService<T>();
52      }
53
54      /// <inheritdoc/>
55      public void AddSingleton<T>(T singletonObject)
56          where T : class => serviceCollection.AddSingleton(singletonObject);
57
58      /// <inheritdoc/>
59      public void AddSingleton(Type serviceType, Type implementationType)
60      {
61          serviceCollection.AddSingleton(serviceType, implementationType);
62      }
63 }
```

Listing A.10: The DependencyManagerInteractor (Koks, 2023i) as an abstraction on external technology dependencies

```
1 public abstract class AbstractExpander<TExpander> : IExpanderInteractor
2     where TExpander : class, IExpanderInteractor
3 {
4     private readonly ILogger logger;
5     private readonly IDependencyFactoryInteractor dependencyFactory;
```

```
6       private readonly App model;
7
8       /// <summary>
9       /// Initializes a new instance of the <see cref="AbstractExpander{TExpander}"/>
           class.
10      /// </summary>
11      /// <param name="dependencyFactory"><seealso cref="IDependencyFactoryInteractor
           "/></param>
12      protected AbstractExpander(IDependencyFactoryInteractor dependencyFactory)
13      {
14          this.dependencyFactory = dependencyFactory;
15
16          logger = this.dependencyFactory.Get<ILogger>();
17          model = dependencyFactory.Get<App>()
18              .Expanders
19              .Single(x => x.Name == Name);
20      }
21 }
```

Listing A.11: An example of resolving dependencies as part of the AbstractExpander (Koks, 2023a)

```
1  // ... other code
2  cmd.OnExecute(() =>
3  {
4      var provider = new ServiceCollection()
5          .AddConsole()
6          .AddDomainLayer()
7          .AddApplicationLayer()
8          .AddEntityFrameworkLayer()
9          .AddInfrastructureLayer()
10         .BuildServiceProvider();
11
12     // ... other code
13 });
14
15 // ... other code
```

Listing A.12: Bootstrapping the dependencies of each component layer of the generator artifact.

```
1  public class Entity
2  {
3      public virtual Guid Id { get; set; }
4      public virtual string Name { get; set; }
5      public virtual string Callsite { get; set; }
6      public virtual string Type { get; set; }
7      public virtual string Modifier { get; set; }
8      public virtual string Behaviour { get; set; }
9      public virtual App App { get; set; }
10     public virtual List<Field> Fields { get; set; }
11     public virtual List<Field> ReferencedIn { get; set; }
12     public virtual List<Relationship> Relations { get; set; }
13     public virtual List<Relationship> IsForeignEntityOf { get; set; }
14 }
```

Listing A.13: An Entity in CA is practically the same as Data Elements in NS (Koks, 2023j).

```
1  internal class CreateEntityValidator : AbstractValidator<CreateEntityCommand>,
       IValidator<CreateEntityCommand>
2  {
3      public CreateEntityValidator()
4      {
5          #region ns-custom-validations
6
7          RuleFor(x => x.Name).NotEmpty();
8
```

```
 9          #endregion ns-custom-validations
10      }
11
12      public new Response Validate(CreateEntityCommand objectToValidate) =>
13          base.Validate(objectToValidate)
14              .ToResponse();
15 }
```

Listing A.14: An Interactor performing a single Task as it is intented for the Task Element of NS (Koks, 2023e).

```
 1 internal class CreateEntityInteractor : IInteractor<CreateEntityRequestModel>
 2 {
 3     //...
 4
 5     public async Task<Response> ExecuteUseCase(CreateEntityRequestModel requestModel)
 6     {
 7         Response result = validator.Validate(requestModel);
 8         if (result.IsValid)
 9         {
10             try
11             {
12                 Entity entity = mapper.Map(requestModel);
13                 entity.Id = Guid.NewGuid();
14
15                 result.SetParameter(entity);
16
17                 int repositoryResult = await repository.Create(entity);
18                 if (repositoryResult != 1)
19                 {
20                     result.AddError(ErrorCodes.InternalServerError, $"Failed to create
                         {nameof(Entity)}.");
21                     return result;
22                 }
23             }
24             catch (Exception exception)
25             {
26                 result.AddError(ErrorCodes.InternalServerError, exception.Message);
27             }
28         }
29
30         return result;
31     }
32 }
```

Listing A.15: An Interactor orchestrating multiple Tasks as it is inteded by the Flow Element of NS (Koks, 2023c).

```
 1     public class DeleteEntityRequestModel : RequestModel
 2     {
 3         public Guid Id { get; set; }
 4     }
```

Listing A.16: The DeleteEntityRequestModel (Koks, 2023g)

```
 1 public class CreateEntityPresenter : ICreateEntityPresenter
 2 {
 3     private readonly IMapper<Entity, EntityViewModel> mapper;
 4
 5     public CreateEntityPresenter(IMapper<Entity, EntityViewModel> mapper)
 6     {
 7         this.mapper = mapper;
 8     }
 9
10     public Response Response { get; set; }
11
12     public IResult GetResult(HttpRequest request = null)
```

```
13      {
14          return Response.IsValid ?
15              Results.Created($"//{mapper.Map(Response.GetParameter<Entity>()).Id}",
16                  mapper.Map(Response.GetParameter<Entity>())) :
                Response.ToWebApiResult(request);
17      }
18 }
```

Listing A.17: The CreateEntityPresenter (Koks, 2023d)

B. Installing & using Pantha Rhei

B.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:

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <packageSources>
        <clear />
        <add key="github" value="https://nuget.pkg.github.com/liquidvisions/index.json"
            />
    </packageSources>
        <packageSourceCredentials>
        <github>
            <add key="Username" value="" />
            <add key="ClearTextPassword" value="{email me for a valid access token @
                gerco.koks@outlook.com} " />
        </github>
    </packageSourceCredentials>
</configuration>
```

Listing B.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.

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

Listing B.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 B.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 B.1

```
flux --root "{folder}"
     --mode Default
     --app "{uniqueid}"
     --db "{connectionstring}"
```

Listing B.3: Example command executing Pantha Rhei

–root	A mandatory parameter that should contain the full path to the output directory B 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 B.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 B.1.: The *flux* command line parameters

RunModes are available to isolate the execution of the ExpanderHandler. It requires a current implementation shown in listing B.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 B.2.: The available *Generation modes*

```
1 public class ExpandDatabaseContextHandlerInteractor
2     : IExpanderHandlerInteractor<CleanArchitectureExpander>
3 {
4     // ... other code
5     public bool CanExecute =>
```

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

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

C. The Entity Relationship Diagram of the Meta Mode

C.1. The App entity

The App entity represents the application and is regarded as the entry point for the model. The App Entity and the subsequent entities contain all the information required to perform the expandsion of a software system.

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 C.1.: The fields of the App entity

C.2. 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

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 C.2.: The fields of the Component entity

C.3. 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 C.3.: The fields of the ConnectionString entity

C.4. 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 C.4.: The fields of the Entity entity

C.5. 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 C.5.: The fields of the Expander entity

C.6. 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 C.6.: The fields of the Field entity

C.7. 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 C.7.: The fields of the Package entity

C.8. 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 C.8.: The fields of the Relationship entity

D. Designs & Architecture

D.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 D.1.: Naming convention component layers

D.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 D.2.: Naming convention of recurring elements

D.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.
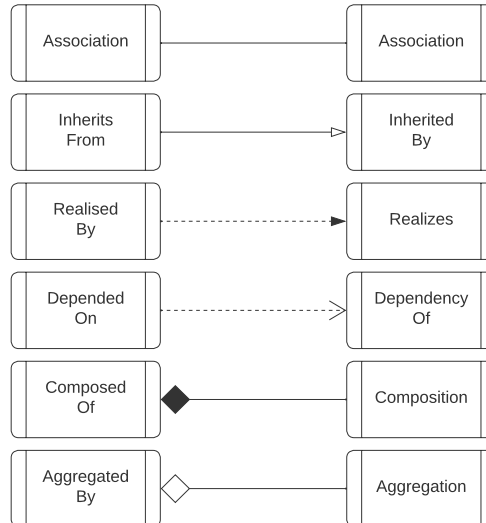

Figure D.1.: UML notation

E. 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 E.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.