ANTWERP MANAGEMENT SCHOOL

MASTER'S CURRICULUM PROGRAM 2021 - 2023

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 (MSc)

June 12, 2023

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
DOI:	10.5281/zenodo.8029972
Copyright:	© 2023 G.C. Koks
License:	🅭🅯🄎 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
ORCID:	0009-0005-5287-0459
Email:	gerco.koks@outlook.com
LinkedIn:	https://www.linkedin.com/in/gercokoks/

Supervisor

Name:	Prof. Dr. Ing. Hans Mulder
ORCID:	0000-0002-3304-9711
Email:	Hans.Mulder@ams.ac.be
LinkedIn:	https://www.linkedin.com/in/jbfmulder/

Promotor

Name:	Dr. ir. Geert Haerens
ORCID:	0000-0001-6425-7911
Email:	geert.haerens@uantwerpen.be
LinkedIn:	https://www.linkedin.com/in/geerthaerens/

"Mistakes are part of the process
because the only person who does not makes a mistake
is the person who does nothing."
— *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* —

In loving dedication to my cherished family members. To my parents, who are both facing a devastating illness and have been admitted to a care home during the final stages of my research, my thoughts are always with you. I am eternally grateful to my brother and sister for their understanding and patience as I poured my heart into this pursuit. And to the brightest star in my life, my dear daughter Emma. To all of you, thank you for your unwavering love, support, and belief in me. I love you all!

Acknowledgments

“*With unwavering commitment, heartfelt encouragement,
and steadfast support, the impossible becomes possible.*”

Looking back, I never imagined I would embark on this academic journey. As a youngling, I struggled with a focusing disorder, which led me to attend a school specializing in helping kids with similar challenges. Despite this, I discovered I could excel by dedicating myself to enjoyable activities. Early in my life, this manifested primarily in sports.

I knew I had to provide for myself as an adult, so I pursued an education in what I loved most: Sports. Consequently, the first decade of my professional career revolved around being a sports instructor. Eventually, my interests shifted toward the technical challenges of software programming. I decided to take the plunge and pursue it professionally. So, I embarked on my first educational endeavor, my bachelor’s degree in Computer Science, completed in 2012. In hindsight, it was the best decision I have made. It caused a giant leap forward from a career perspective but also for my personal growth. My decision has brought me to this point where I am close to completing my Master’s degree, which makes me incredibly proud.

It would have been a much more difficult journey without the encouragement and support of many people. First, Rene Bliekendaal encouraged and convinced me to embark on this journey. Thank you! I also greatly respect my employer for facilitating this pursuit while I continued to work full-time. Your flexibility has played an essential role in making it a reality.

I sincerely thank my supervisor, Hans Mulder, and promoter, Geert Haerens, for their invaluable guidance, feedback, and mentorship throughout my research. Your expertise and insights have been instrumental in shaping my work and ensuring this success. Also, a special mention goes to Frans Vertreken, who stepped up to guiding me in the early stages of research when no one else was available. Your knowledge and encouragement have been inspiring and laid the basis for my result. I extend my heartfelt gratitude to Antwerp Management School, which provided me with an excellent platform to pursue my personal and scholarly growth. To everyone else who has played a role in this journey, no matter how big or small, I offer my heartfelt thanks. This achievement would not have been possible without each one of you.

So, what is next for me? First, I fully intend to repay my employer’s continued support and encouragement by sharing my knowledge and planning to bring additional value. Then, in my free time, I will spend some well-deserved and needed time focusing on my health and my closest family members. After that? I am not sure yet, but I want to explore more on the subject of software stability and evolvability, as this is where my true passion and potential lie.

— Gerco

Table of Contents

List of Tables

List of Figures

1. Introduction

Shortly after starting my bachelor's degree in 2008, I started to work as a junior software engineer. I was confident and 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 had to offer (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 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. Therefore, CA could be used to achieve a modular, evolvable, and stable software artifact as defined by NS.

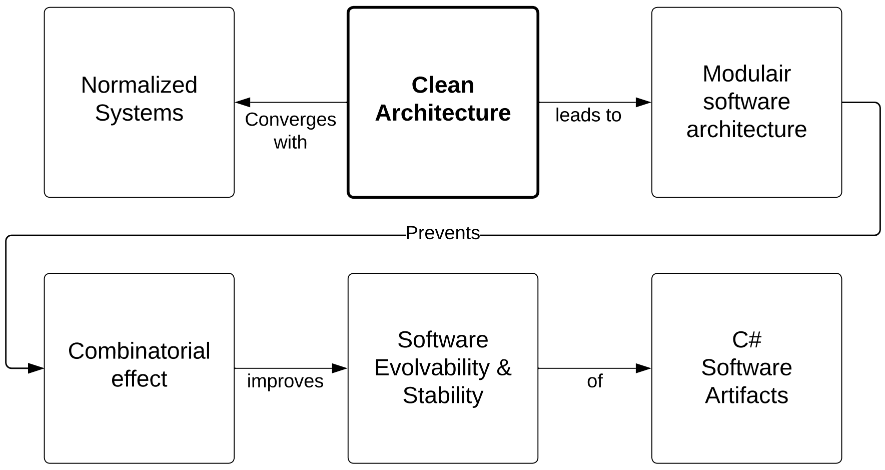


Figure 1.1.: The hypothesis

Since this research investigates the convergence of CA 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 described by Wieringa (2014). The engineering cycle provides a structured approach to developing the required artifacts to analyze the design problem.

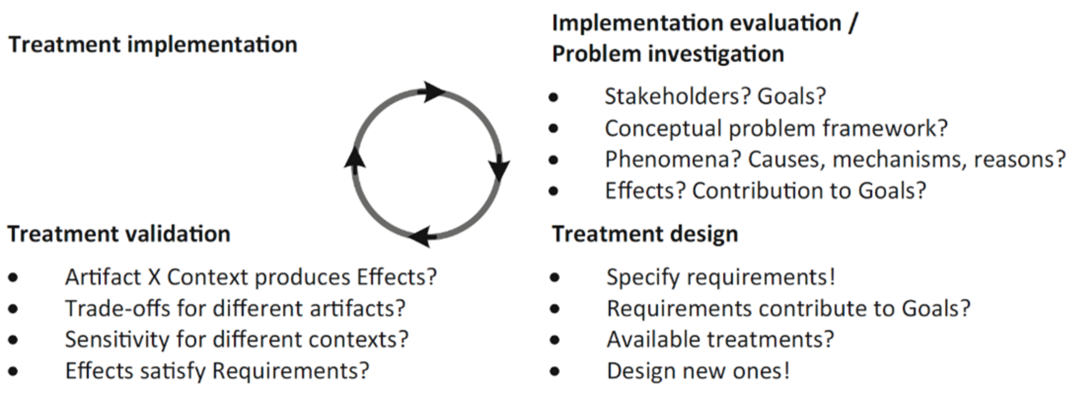

Treatment implementation
Implementation evaluation / Problem investigation
Stakeholders? Goals?
Conceptual problem framework?
Phenomena? Causes, mechanisms, reasons?
Effects? Contribution to Goals?
Treatment validation
Artifact X Context produces Effects?
Trade-offs for different artifacts?
Sensitivity for different contexts?
Effects satisfy Requirements?
Treatment design
Specify requirements!
Requirements contribute to Goals?
Available treatments?
Design new ones!

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

In this research, a significant component has been the development of a tangible software artifact, providing a real-world illustration of the convergence between Clean Architecture and Normalized Systems. Hevner et al. (2004) 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.

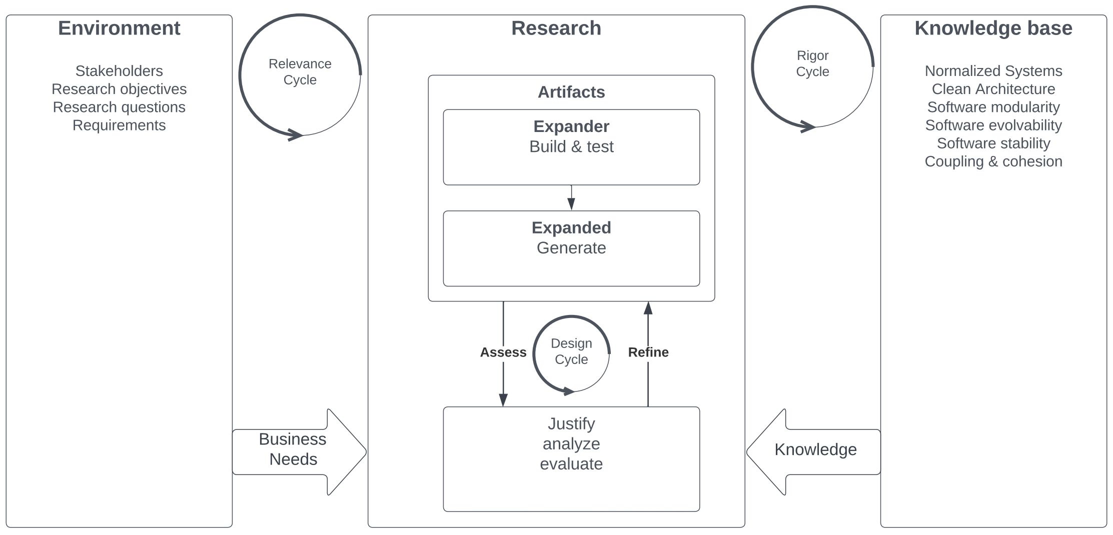


Figure 1.3.: The design Science Framework for IS Research (Hevner et al., 2004)

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 facilitate the study of 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 convergence with the CA principles and design.
4. **Analysis of combinatorics**
   Examine the artifacts for actual or potential combinatorial effects to determine whether CA and NS converge. The fundamental principles and architecture of CA are considered while conducting the analysis.

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 6.4.

2. Exploring the Concepts of Clean Architecture and Normalized Systems

This research aims to study the convergence of CA with NS. This chapter will describe the key concepts and principles, design elements, and characteristics of both software design approaches that affect the research conclusion. 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, followed by the concepts, principles, and elements of CA. We will briefly discuss each subject with little detail on the theory, as most concepts have been thoroughly documented in the literature.

2.1. Generic Concepts

In the following sections, we will examine concepts related to both CA and NS. Understanding these concepts is crucial for executing the research and interpreting its results.

2.1.1. Modularity

The original material of 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 exhibit high cohesion. While both design approaches agree on the cohesiveness of a module's internal parts, there is a slight difference in granularity in their definitions.

2.1.2. High Cohesion

Mannaert et al. (2016, p. 22) consider cohesion as modules that exist out of connected or interrelated parts of a hierarchical structure. On the other hand, Robert C. Martin (2018, p. 118) discusses cohesion in the context of components. He attributes the three component cohesion principles as crucial to group 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

The Theory of NS revolves around modular structures in information systems and their behavior when modified over time. NS uses scientific insights from System Theory and Statistical Entropy from Thermodynamics. NS has a background in software engineering. 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). This emphasizes the impact that NS Theory has. In the following sections, we will highlight the concepts of NS Theory that have impacted this study.

2.2.1. The study of Stability, Evolvability, and Combinatorics

The NS Theory considers stability a crucial property derived from the concept BIBO: A bounded functional change must result in a bounded amount of work, independent of the size of the system. 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 new requirement. Mannaert et al. (2016, p. 271) refer to these 'instabilities' as combinatorial effects. Conversely, stability is achieved when a system is free from these so-called combinatorial effects. Based on the concept of stability, Mannaert et al. (2012) require information systems to be stable with respect to a set of anticipated changes to exhibit high evolvability.

2.2.2. Expansion

According to Mannaert et al. (2016, p. 403), creating and maintaining a stable and evolvable system is 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 automating software structure instantiation by using code generation for recurring tasks. This process is referred to as expansion.

2.2.3. Craftings

Craftings are manually inserted code snippets added after the automated instantiation process to comply with the functional requirements. Craftings are only inserted when automation instantiation is not possible or desired. These craftings should be preserved after each expansion. The craftings are preserved by a process called harvesting and injection (Mannaert et al., 2016, pp. 405–406).

2.2.4. 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.

2.2.5. Normalized Elements

In the context of the NS Theory approach, the goal is to design evolvable software independent of the underlying technology. Nevertheless, a particular technology must be chosen when implementing the software and its components. 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). The table describing each element uses the definition from Mannaert et al. (2012).

Element	Description
Data	Based on DvT, data elements have get- and set-methods for wide-sense data version transparency, or marshal -and parse- methods for strict-sense data version transparency. Supporting tasks can be added in a way which is consistent with the principles SoC and DvT.
Task	Based on SoC, the core action entity can only contain a single functional task, and not multiple tasks. Based on AvT, arguments and parameters must be encapsulated data entities. Based on SoC and SoS, workflows need to be separated from action entities, and will therefore be encapsulated in a workflow element. Based on AvT, tasks need to be encapsulated in such a way that a separate action entity wraps the action entities representing task versions. Supporting tasks can be added in a way which is consistent with SoC and AvT.
Workflow	Based on SoC, workflow elements cannot contain other functional tasks, as they are generally considered a separate change driver, often implemented in an external technology. Based on SoS, workflow elements must be stateful. This state is required for every instance of use of the action element, and therefore needs to be part of, or linked to, the instance of the data element that serves as argument.
Connector	Based on Theorem SoS, connector elements must ensure that external systems can interact with data elements, but that they cannot call an action element in a stateless way. Supporting tasks can be added in a way which is consistent with SoC and AvT.
Trigger	Based on SoC, trigger elements need to control the separated —both error and non- errorstates, and check whether an action element has to be triggered. Supporting tasks can be added in a way which is consistent with SoC and AvT.

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 relatively easy to apply new requirements. (Robert C. Martin, 2018, p. 94).

The Liskov Substitution Principle

The Liskov 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 another (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 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 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, 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 its components into distinct layers. This architecture promotes the separation of concerns, maintainability, testability, and adaptability. The following section briefly describes 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.
ResponseModel	ResponseModel are used to represent the output data required by a specific interactor.
ViewModel	ViewModels are responsible for managing the data and behavior 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 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 technological 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.

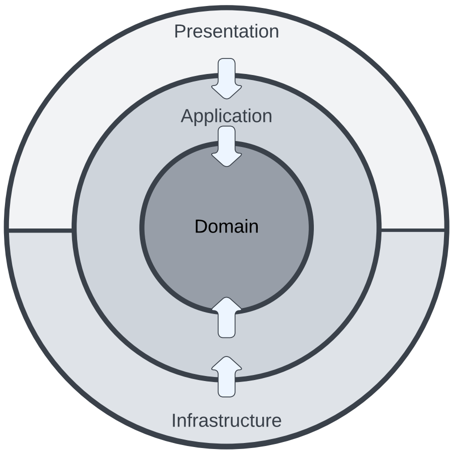


Figure 2.1.: Flow of control

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 the idea that Software architectures should emphasize the system's intent, theme, and purpose 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 and 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 defined them as follows.

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 with 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 Appendix C.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.	The technology requirements
2.1	The Domain and Application layers have no dependencies on 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.: The technology requirements

3.2.2. Software Architecture Requirements

Figure 3.1 illustrates the generic software architecture of the artifacts. Each instantiated element adheres to the Element Naming Convention outlined in Appendix C.2. The following tables detail the requirements specific to each element.

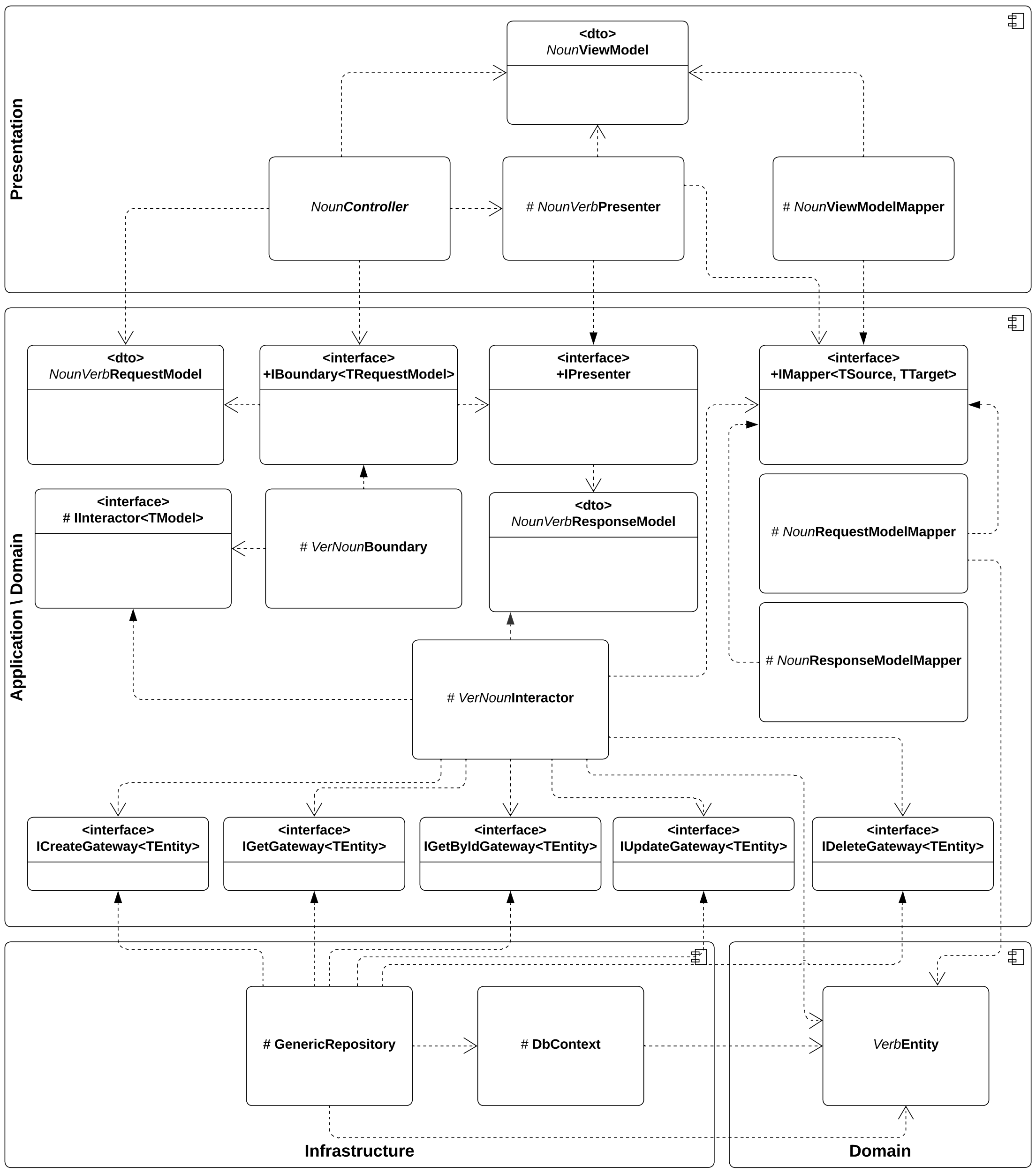


Figure 3.1.: The Generic architecture of the artifacts

Nr.	The ViewModel 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.: The ViewModel requirements

Nr.	The 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 the Presentation layer.

Table 3.4.: The Presenter requirements

Nr.	The 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 the Presentation layer.

Table 3.5.: The ViewModelMapper requirements

Nr.	The 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.: The Controller requirements

The Application Layer

Nr.	The 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.: The IBoundary requirements

Nr.	The Boundary Implementation requirements
8.1	A Boundary implementation is derived from the IBoundary interface and follows the specified implementation.
8.2	The Boundary implementation separates the internal aspects of the Application Layer and the other layers within the component.
8.3	Each Boundary implementation handles a single task, executed using the IInteractor interface.
8.4	Boundary implementations have an internal scope and cannot be instantiated outside the Application layer.

Table 3.8.: The Boundary Implementation requirements

Nr.	The 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.: The IInteractor requirements

Nr.	The 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.: The Interactor Implementation requirements

Nr.	The 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.: The IMapper requirements

Nr.	The 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.: The RequestModelMapper requirements

Nr.	The 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.: The ResponseModelMapper requirements

Nr.	The 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.: The IPresenter requirements

Nr.	The Gateway requirements
15.1	The Domain and Application layers have no dependencies on infrastructure technologies, like web- or database technologies.
15.2	The *[Verb]Gateway* interface establishes the contract for its derived Gateway implementations, 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. For example, the basic CRUD actions result in five IGateway interfaces: ICreateGateway, IGetGateway, IGetByIdGateway, IUpdateGateway, and IDeleteGateway.

Table 3.15.: The Gateway requirements

Nr.	The 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.: The ResponseModel requirements

Nr.	The 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.: The RequestModel requirements

The Domain Layer

Nr.	The 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.: The Data Entity requirements

The Infrastructure Layer

Nr.	The 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 the layer.

Table 3.19.: The Gateway Implementation requirements

Design Principles Compliance

Nr.	The Design Principles requirements
20.1	Each architectural pattern adheres to at least one of the SOLID principles to ensure that none of the implementations violate these principles.

Table 3.20.: The Design Principles requirements

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.	The Expander Framework requirements
21.1	The Expander Framework enables interaction with the Clean Architecture Expander via a CLI. The CLI is implemented in the Presentation layer of the Expander Framework.
21.2	The Expander Framework retrieves the model from an MSSQL using the EntityFramework ORM technology. The EntityFramework technology is implemented in the Infrastructure layer of the Expander Framework.
21.3	The Expander Framework loads and executes the configured Expanders. In the case of this research, only the Clean Architecture Expander is applied.
21.4	The Expander Framework supports generic harvesting and injection, which can be used or extended by the Expanders using the OCP principle.
21.5	The Expander Framework supports generic template handling, which can be used or extended by the Expanders using the OCP principle.
21.6	The Expander framework adheres to this chapter's component and software Requirements specified in Sections 3.2.1 and 3.2.2.

Table 3.21.: The Expander Framework requirements

Nr.	The Clean Architecture Expander requirements
22.1	The Clean Architecture Expander generates a C# net7.0 RESTful service that provides an HTTP interface on top of the meta-model of the Expander Framework, allowing the basic CRUD operations.
22.2	The Clean Architecture Expander consists solely of an Application layer and reuses the Domain layer of the Expander Framework.
22.3	The Clean Architecture Expander adheres to this chapter's component and software Requirements specified in Sections 3.2.1 and 3.2.2.

Table 3.22.: The Clean Architecture Expander requirements

3.2.4. Generated Artifact Requirements

Nr.	The generated artifact requirements
23.1	The generated artifact adheres to this chapter's component and software Requirements specified in Sections 3.2.1 and 3.2.2.

Table 3.23.: The generated artifact requirements

4. Artifact Design

This chapter will discuss specific design decisions made to meet the required functionality while adhering to the requirements outlined in chapter 3. Two different artifacts are used to support this study. Figure 4.1 is a schematical overview of both these artifacts.

4.1. The Research Artifacts

The first artifact consists of two main components: the Clean Architecture Expander and the Expander framework. 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 perceived ability to cope with constant change in a stable and evolvable manner. Users can interact with the Expander Framework using the CLI command 'flux' in combination with several parameters. Appendix D, yprovides a comprehensive guide on using this command, including all available options and parameters.

As illustrated in Figure 4.1, the main task of the first artifact or 'expand' the second artifact. By entering the correct command, the Expander Framework loads the model being instantiated during the expansion process. Then, the required expanders are prepared based on information available through the model. In the case of this study, the Clean Architecture Expander. The Clean Architecture Expander consists of a set of tasks and templates. When the Expander Framework executes the Clean Architecture Expander, the model is instantiated into the generated artifact with the aid of the templates.

The model is an instance of the meta-model. Consequently, the model can represent any application as long as the meta-model is respected. In the case of this study, the model represents the entities, attributes, relationships, and other characteristics of the meta-model.

As a result, the second artifact (artifact II) allows a user to modify or maintain the model used by the Expander Framework by exposing a Restful interface. This method approaches the meta-circularity process, where an expansion process is used to update the meta-model. Although not fully compliant with the theory of NS, the Expander Framework consists of the required tasks to update its own meta-model. This is illustrated in Figure 4.1 by the 'updates' arrow.

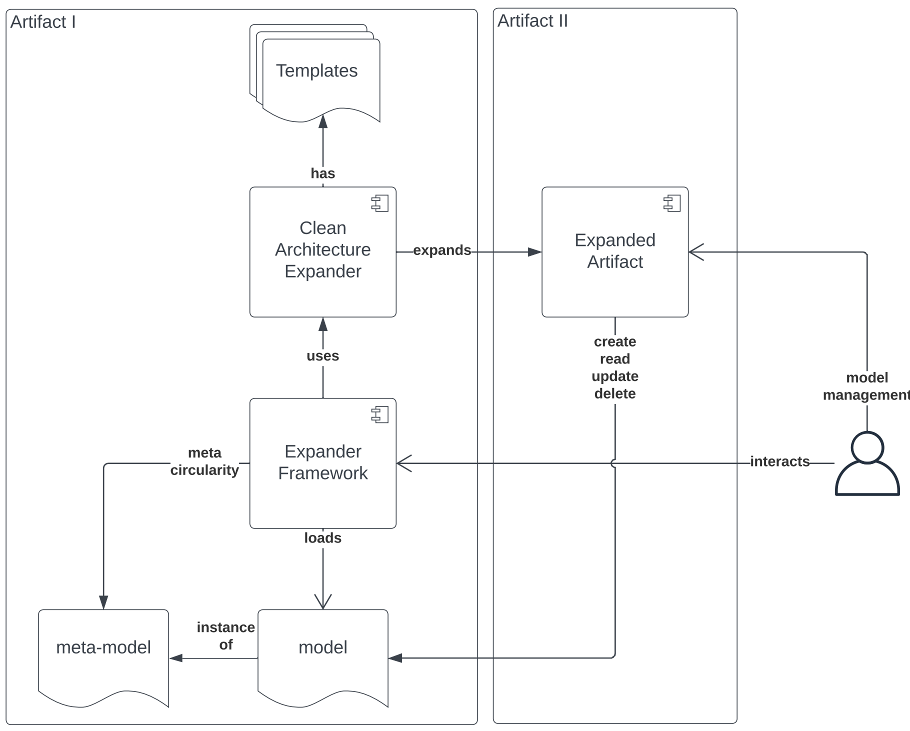


Figure 4.1.: Schematic overview of the artifacts

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.

Figure 4.2 illustrates the version of the meta-model used for this research. A detailed description of each of the elements can be found in Appendix B, The Entity Relationship Diagram of the Meta-Model

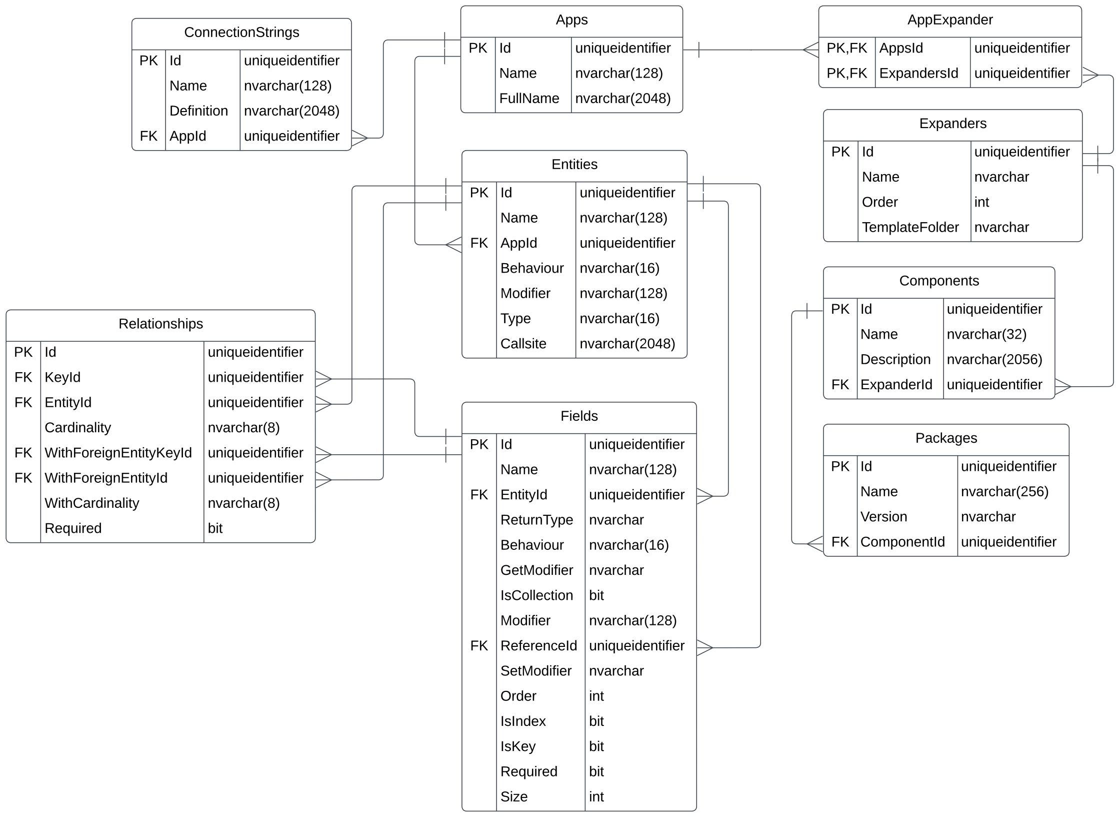


Figure 4.2.: The meta-model represented as an Entity Relationship Diagram

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, 2023n).

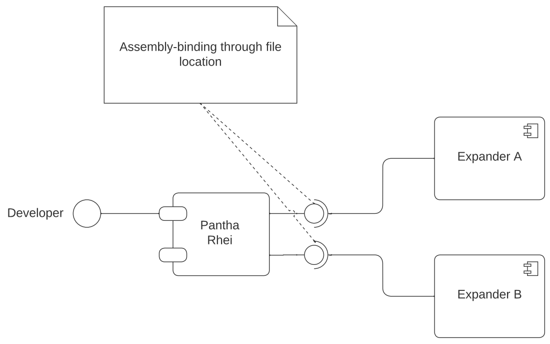


Figure 4.3.: Expanders are considered plugins

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 Appendix A.1, The ExpanderPluginLoaderInteractor

4.4. Expanders

The Exander Framework allows for the miscellaneous execution of expanders of any type. The Expander Framework is independent of any of the details of Expanders, fully adhering to the principle of DIP. Conversely, an Expander is required to implement several interfaces to ensure execution and dependency management are available during runtime. The Expander Framework also consists of a set of default tasks, such as the execution of the expansion tasks known as ExpanderHandlerInteractors IExpanderHandlerInteractor (Koks, 2023u), logging, bootstrapping dependencies, and tasks to execute harvestings and injections. Except for the use of the IExpanderInteractor, non of which are required.

Figure 4.4 illustrates the dependencies between the domain layer of the Expander Framework. The Clean Architecture Expander is considered an application layer containing specific tasks bounded to a particular application or process. In this case, the Expansion process.

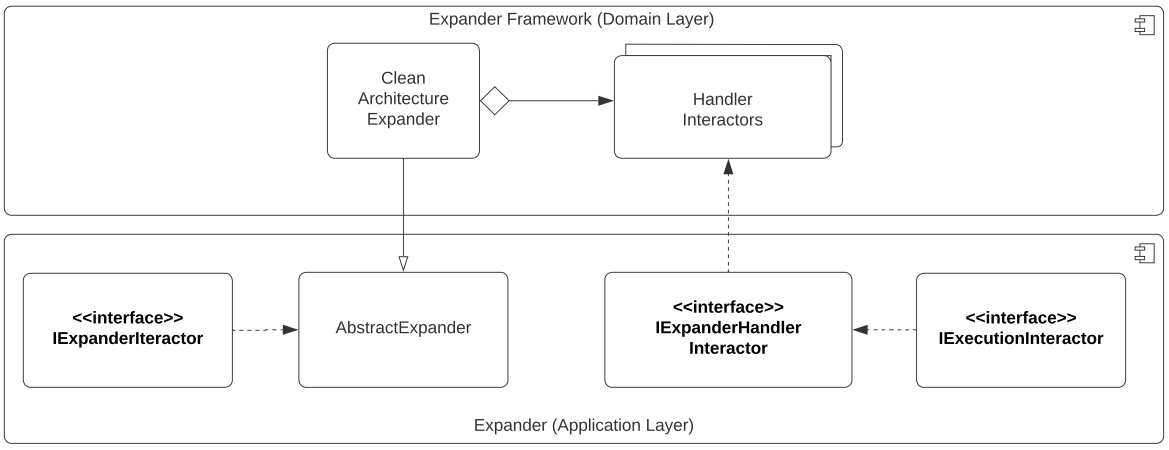

Expander Framework (Domain Layer)
Clean Architecture Expander
Handler Interactors
<<interface>> IExpanderIteractor
AbstractExpander
<<interface>> IExpanderHandler Interactor
<<interface>> IExecutionInteractor
Expander (Application Layer)

Figure 4.4.: The design of an Expander

4.5. Executing Commands

An exciting implementation that facilitates a high degree of cohesion while maintaining low coupling is the utilization of the *IExecutionInteractor* interface (Koks, 2023t). This interface allows for the execution of various derived types responsible for various tasks, such as executing Handlers, Harvesters, and Rejuvenators[1] (Koks, 2023m, 2023z, 2023aa). The implementation promotes decoupling by adhering to both OCP and LSP.

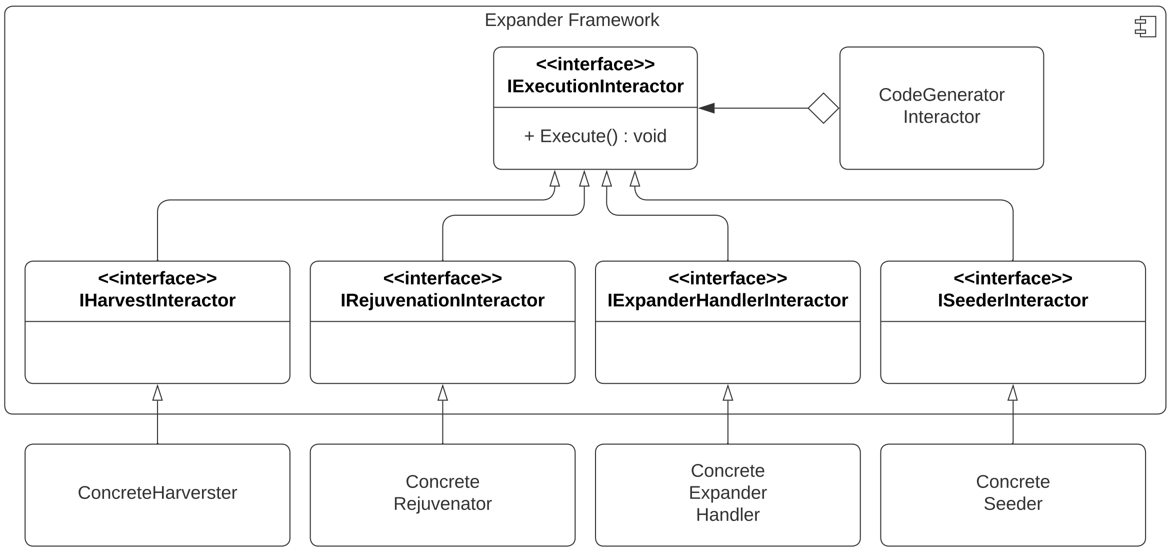

Expander Framework
<<interface>> IExecutionInteractor
+ Execute() : void
CodeGenerator Interactor
<<interface>> IHarvestInteractor
<<interface>> IRejuvenationInteractor
<<interface>> IExpanderHandlerInteractor
<<interface>> ISeederInteractor
ConcreteHarverster
Concrete Rejuvenator
Concrete Expander Handler
Concrete Seeder

Figure 4.5.: Low coupling with *IExecutionInteractor*

[1]It is important to note that the Rejuvenation objects in this version of the artifact are capable of performing injections and not the entire Rejuvenation process.

Figure 4.5 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, 2023y). 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 Mannaert et al. (2016, p. 215) statement 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 must 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.8 (Koks, 2023i). 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, which 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, 2023s) and resolve (Koks, 2023r) dependencies. As illustrated in Code Listing A.9, the framework technology dependency is abstracted behind implementing those interfaces (Koks, 2023j).

Practically every class gets the IDependencyFactoryInteractor (Koks, 2023r) injected, on which further resolving is responsible for that class's inner workings. Code Listing A.10 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.11.

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

In this chapter we will analyze the two approaches, CA and NS. We will examine how these approaches converge 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. Additionally, we will showcase real-world examples from the artifacts to illustrate their practical manifestations.

We will finalize this chapter by analyzing combinatorial effects on the various change dimensions of the studied artifacts.

5.1. An Analysis of Principles

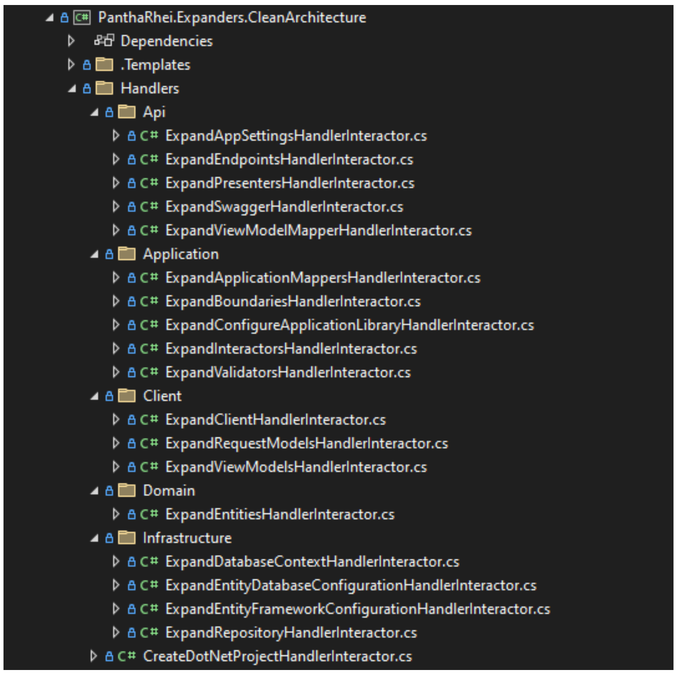
In this section, we will apply a systematic cross-referencing approach to each of the principles of CA with NS. By cross-referencing these principles, we aim to uncover the degree of convergence between CA and NS from a theoretical perspective supported by examples from the artifacts. Along this explanation, the level of convergence is denoted as follows:

Strong convergence	**++**	This indicates that the principles of CA and NS are highly converged. Both have a similar impact on the design and implementation of the artifact.
Supports convergence	**+**	The CA principle supports 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 convergence	**−**	The principles 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 are practically interchangeable. Many examples in the artifacts show a strong convergence 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 illustrates some of the Tasks implemented in the Clean Architecture Expander artifact. The Code Listing A.3 is an example of one implementation of such a task ExpandEntitiesHandlerInteractor (Koks, 2023m).
DvT	+	Although using SRP does not implicitly guarantee DvT, it supports DvT by directing certain design choices.For example, CA and NS assign specific DTO objects to support specific use cases (Interactors or Tasks) or 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, 2023ab, 2023ad). 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 SRP 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, Code Listing A.4 shows that API versioning is a common standard practice fully supported by the open API specification and the .net core framework (Github, 2023b; OAS, 2023). Manifestations in the artifact can be located in the Logger (Code Listing A.5), amongst others (Koks, 2023x).
SoS	−	Following SRP might lead to separate modules that manage their state, indirectly contributing to SoS. However, the convergence is very weak, and no manifestations are found in the artifacts.

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



```
PanthaRhei.Expanders.CleanArchitecture
  Dependencies
  .Templates
  Handlers
    Api
      ExpandAppSettingsHandlerInteractor.cs
      ExpandEndpointsHandlerInteractor.cs
      ExpandPresentersHandlerInteractor.cs
      ExpandSwaggerHandlerInteractor.cs
      ExpandViewModelMapperHandlerInteractor.cs
    Application
      ExpandApplicationMappersHandlerInteractor.cs
      ExpandBoundariesHandlerInteractor.cs
      ExpandConfigureApplicationLibraryHandlerInteractor.cs
      ExpandInteractorsHandlerInteractor.cs
      ExpandValidatorsHandlerInteractor.cs
    Client
      ExpandClientHandlerInteractor.cs
      ExpandRequestModelsHandlerInteractor.cs
      ExpandViewModelsHandlerInteractor.cs
    Domain
      ExpandEntitiesHandlerInteractor.cs
    Infrastructure
      ExpandDatabaseContextHandlerInteractor.cs
      ExpandEntityDatabaseConfigurationHandlerInteractor.cs
      ExpandEntityFrameworkConfigurationHandlerInteractor.cs
      ExpandRepositoryHandlerInteractor.cs
    CreateDotNetProjectHandlerInteractor.cs
```

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

5.1.2. Open/Closed Principle

SoC	++	The OCP strongly converges 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 adherence to OCP, as SoC focuses on modularization and encapsulation rather than the extensibility of functionality. The same example with the Tasks provided in subsection 5.1.1 is also an excellent manifestation of this principle.
DvT	–	The OCP indirectly relates to the DvT principle. The convergence of both principles is weak, and no manifestations are found in the artifacts.
AvT	++	The OCP strongly converges 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 has 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 convergence of both principles is weak, and no manifestations are found in the artifacts.

Table 5.2.: The convergence 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 and converging the principles inherently. A good example is the implementation of the ITemplateInteractor (Koks, 2023v), where the template engine Scriban (Github, 2023c) is used to generate code instantiations as a result of the Expanding the model (Koks, 2023ac). We could easily replace the Scriban template engine with another engine, only impacting the Dependency Injection Register.
DvT	−	The convergence 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 to AvT. Consider ICreateGateway (Koks, 2023q) in Code Listing A.6. 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 convergence of both principles is weak, and no manifestations are found in the artifacts.

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

5.1.4. Interface Segregation Principle

SoC	++	The ISP strongly converges 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. Listing A.7 shows that each CRUD operation has its own interface (Koks, 2023g).
DvT	−	The ISP does not relate to the DvT principle. The convergence of both principles is weak, and no manifestations are found in the artifacts.
AvT	+	The convergence between ISP and AvT arises from the emphasis of ISP on creating targeted interfaces 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 convergence of both principles is weak, and no manifestations are found in the artifacts.

Table 5.4.: The convergence 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 section 2.3.4, The Dependency Rule (Koks, 2023w). Managing Dependencies inheritly promotes SoC, therefore DIP converges with SoC to some extend. However, adhering to SoC does not guarantee SoC.
DvT	−	The DIP does not relate to the DvT principle. The convergence 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. However, the convergence between DIP and AvT is not so strong as with SoC, and adhering to DIP alone will not guarantee a system that entirely complies with AvT.
SoS	−	The DIP does not relate to the SoS principle. The convergence of both principles is weak, and no manifestations are found in the artifacts.

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

5.1.6. The Principles Convergence Overview

In this section, we will apply a systematic cross-referencing approach to each of the principles of CA with NS. By cross-referencing these principles, we aim to uncover the degree of convergence between CA and NS from a theoretical perspective supported by examples from the artifacts. Along with this explanation, the level of convergence is denoted as follows:

Clean Architecture / **Normalized Systems**	Separation Of Concerns	Data Version Transparency	Action Version Transparency	Separation of State
Single Responsibility Principle	++	+	+	−
Open/Closed Principle	++	−	++	−
Liskov Substitution Principle	++	−	+	−
Interface Segregation Principle	++	−	+	−
Dependency Inversion Principle	++	−	+	−

Table 5.6.: An overview of the convergence of all CA and NS principles

5.2. An Analysis of Elements

In this section, we will apply a systematic cross-referencing approach to each of the elements of CA with NS. By cross-referencing these elements, we aim to uncover the degree of convergence between CA and NS from a theoretical perspective supported by examples from the artifacts. Along with this explanation, the level of convergence is denoted as follows:

Strong convergence	++	Both elements have a high level of similarity or are closely related in terms of their purpose, structure, or functionality.
Supports convergence	+	Both elements have similarities or share certain aspects in their purpose, structure, or functionality, but they are not identical or directly interchangeable.
No or weak convergence	−	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 complete set of attributes and relationships, the Data Element of NS may also be tailored to serve a specific set of information required for a single task or use case. In CA, these type of Data Elements are explicitly specified as ViewModels, RequestModels, or Response-Models. Code Listing A.12 illustrates an example of an Entity in a way that is very similar to the Data Element from NS (Koks, 2023k)
Task	−	There is no convergence between the Entity element of CA and the Connector element of NS, and no manifestations are found in the artifact.
Flow	−	There is no convergence between the Entity element of CA and the Flow element of NS, and no manifestations are found in the artifact.
Connector	−	There is no convergence between the Entity element of CA and the Connector element of NS, and no manifestations are found in the artifact.
Trigger	−	There is no convergence between the Entity element of CA and the Trigger element of NS, and no manifestations are found in the artifact.

Table 5.7.: The convergence of the element Entity with the elements of NS

5.2.2. The Interactor Element

Data	**−**	There is no convergence between the Interactor element of CA and the Data element of NS, and no manifestations are found in the artifact.
Task	**++**	The Interactor element of CA has a strong convergence with the task element of NS, as both encapsulate the execution of business rules. This is illustrated in Code Listing A.13, which converges with an implementation of a task, having a single execution of a business rule (Koks, 2023f).
Flow	**++**	The Interactor strongly converges with the Flow element of NS, as both elements can orchestrate the flow of execution for a use case, which can involve multiple Tasks in NS. This is clearly illustrated in Code Listing A.14, where the Interactor handles validation, mapping, and persistence of the Entity (2023d).
Connector	**−**	There is no convergence between the Interactor element of CA and the Connector element of NS, and no manifestations are found in the artifact.
Trigger	**−**	There is no convergence between the Interactor element of CA and the Trigger element of NS, and no manifestations are found in the artifact.

Table 5.8.: The convergence 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. While both elements can contain an aggregated or subset of data attributes representing the ontology, the RequestModel element is specifically targeting the needs of the input on behalf of a particular Use Case. This is illustrated in Code Example A.15, where the available data attributes are tailored to the Use Case of deleting an Entity record (Koks, 2023h).
Task	−	There is no convergence between the RequestModel element of CA and the task element of NS, and no manifestations are found in the artifact.
Flow	−	There is no convergence between the RequestModel element of CA and the Flow element of NS, and no manifestations are found in the artifact.
Connector	−	There is no convergence between the RequestModel element of CA and the Connector element of NS, and no manifestations are found in the artifact.
Trigger	−	There is no convergence between the RequestModel element of CA and the Trigger element of NS, and no manifestations are found in the artifact.

Table 5.9.: The convergence of the element RequestModel with the elements of NS

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. While both elements can contain an aggregated or subset of data attributes representing the ontology, the ResponseModel element is specifically targets the needs of the output on behalf of a particular Use Case.
Task	−	There is no convergence between the ResponseModel element of CA and the task element of NS, and no manifestations are found in the artifact.
Flow	−	There is no convergence between the ResponseModel element of CA and the Flow element of NS, and no manifestations are found in the artifact.
Connector	−	There is no convergence between the ResponseModel element of CA and the Connector element of NS, and no manifestations are found in the artifact.
Trigger	−	There is no convergence between the ResponseModel element of CA and the Trigger element of NS, and no manifestations are found in the artifact.

Table 5.10.: The convergence of the element ResponseModel with the elements of NS

5.2.5. The ViewModel 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 elements can contain an aggregated or subset of data attributes representing the ontology, the ViewModel element is specifically targets the needs on behalf of a particular (user)interface of the application.
Task	–	There is no convergence between the ViewModel element of CA and the task element of NS, and no manifestations are found in the artifact.
Flow	–	There is no convergence between the ViewModel element of CA and the Flow element of NS, and no manifestations are found in the artifact.
Connector	–	There is no convergence between the ViewModel element of CA and the Connector element of NS, and no manifestations are found in the artifact.
Trigger	–	There is no convergence between the ViewModel element of CA and the Trigger element of NS, and no manifestations are found in the artifact.

Table 5.11.: The convergence of the element ViewModel with the elements of NS

5.2.6. The Controller Element

Data	−	There is no convergence between the Controller element of CA and the Data element of NS, and no manifestations are found in the artifact.
Task	−	There is no convergence between the Controller element of CA and the task element of NS, and no manifestations are found in the artifact.
Flow	−	There is no convergence between the Controller element of CA and the Flow element of NS, and no manifestations are found in the artifact.
Connector	+	Although the Controller of CA supports the intent of the Connector element of NS, they are only partially interchangeable. While both elements are involved in the interaction between components, the Controller element from CA primarily intends to interact with external systems using a specific protocol or technology involving user or web interfaces. An example of such a Controller is illustrated in Code Listing A.17. (Koks, 2023l). In this example, the Controller exposes a Restful interface.
Trigger	+	Although the Controller of CA supports the intent of the Trigger element of NS, they are only partially interchangeable. While both elements are involved in receiving events from external systems, a Controller is also able to initiate communication with the same external systems using specific protocols or technologies involving user or web interfaces.

Table 5.12.: The convergence of the element Controller with the elements of NS

5.2.7. The Gateway Element

Data	−	There is no convergence between the Gateway element of CA and the Data element of NS and no manifestations are found in the artifact.
Task	−	There is no convergence between the Gateway element of CA and the task element of NS, and no manifestations are found in the artifact
Flow	−	There is no convergence between the Gateway element of CA and the Flow element of NS, and no manifestations are found in the artifact
Connector	++	The Gateway element of CA has a strong convergence with the Connector element of NS, as both are involved in communication between components and provide interfaces for accessing external resources or systems. Code Listing A.6 is an example of two different types of Gateways. The HarvestRepository (Koks, 2023p) interacts with the File system of the operating system, while GenericRepository (Koks, 2023o) interacts with a SQL Database.
Trigger	−	There is no convergence between the Gateway element of CA and the Trigger element of NS, and no manifestations are found in the artifact

Table 5.13.: The convergence of the element Gateway with the elements of NS

5.2.8. The Presenter Element

Data	−	There is no convergence between the Presenter element of CA and the Data element of NS, and no manifestations are found in the artifact.
Task	+	The Presenter is responsible for preparing the ViewModel on behalf of the Controller and can be considered a task Element with a narrow scope. Because of this narrow scope, the elements are not fully interchangeable. Code Listing A.16 illustrated the inner workings of a Presenter (Koks, 2023e).
Flow	+	Presenters can handle multiple Tasks when this is required. In this case, there is also some convergence between the Presenter Element of CA with the Flow Element of NS.
Connector	−	There is no convergence between the Presenter element of CA and the Connector element of NS, and no manifestations are found in the artifact.
Trigger	−	There is no convergence between the Presenter element of CA and the Trigger element of NS, and no manifestations are found in the artifact.

Table 5.14.: The convergence of the element Presenter with the elements of NS

5.2.9. The Boundary Element

Data	−	There is no convergence between the Boundary element of CA and the Data element of NS, and no manifestations are found in the artifact.
Task	−	There is no convergence between the Boundary element of CA and the task element of NS, and no manifestations are found in the artifact.
Flow	−	There is no convergence between the Boundary element of CA and the Flow element of NS, and no manifestations are found in the artifact.
Connector	++	The Boundary element of CA has a strong convergence with the Connector element of NS, as both are involved in communication between components and help ensure loose coupling between these components. However, the Boundary element's scope seems narrower, as this element usually separates architectural boundaries within the application or component. In the Code Listing Example A.18, we can notice that the primary purpose of the Boundary is to separate the inner parts of the Application layer from the Presentation layer, which converges with the goal of the Connector Element of NS
. Trigger	−	There is no convergence between the Boundary element of CA and the task element of NS, and no manifestations are found in the artifact.

Table 5.15.: The convergence of the element Boundary with the elements of NS

5.2.10. The Elements Convergence Overview

The following table offers a brief overview of the convergence of all elements from both CA and NS.

Clean Architecture / **Normalized Systems**	Data Elements	Task Element	Flow Element	Connector Element	Trigger Element
Entity Element	++	−	−	−	−
Interactor Element	−	++	++	−	−
RequestModel Element	++	−	−	−	−
ResponseModel Element	++	−	−	−	−
ViewModel Element	++	−	−	−	−
Controller Element	−	−	−	+	+
Gateway Element	−	−	−	++	−
Presenter Element	−	+	+	−	−
Boundary Element	−	−	−	++	−

Table 5.16.: An overview of the convergence of all CA and NS elements

5.3. Analysis of Combinatorial Effects

Besides the theoretical analysis by comparing the principles in section 5.1 and elements in section 5.2, we have also analyzed the combinatorial effects on the artifacts. To ensure clarity, we have divided the analysis into the following change dimension, which we will describe in successive sections.

5.3.1. The Mirror World

Mannaert et al. (2016, p. 137) use the term 'Mirror world' as an analogy that refers to the activation of the technology of the information system.

The analysis of both artifacts did not show immediate combinatorial effects, aside from the analysis described in the next section of Combinatorics in the templates. However, table 5.6 clearly shows that SoS is not represented by any of the design principles of CA. Therefore, artifacts solely based on the CA principles will potentially lack stability and evolvability when implementing stateful solutions. Nevertheless, we could not detect any combinatorial effects due to the underrepresentation of Separation of State in CA, which might have been influenced due to the absence of complex stateful implementations, aside from Interactors handling multiple actions similarly as how this is prescribed by the Workflow element of NS.

As indicated in Table 5.16, there also seems to be a lack of a strong foundation for receiving external triggers in the design philosophy of CA. The Controller element partially represents this. This feature is typically utilized for web-based platforms like websites and Restful APIs. However, this approach may not be as thorough regarding receiving external triggers from different technologies or systems. , here we could not detect any combined effect which might have been influenced due to the absence of handling external triggers.

5.3.2. The Templates

Using templates in the Clean Architecture Expander has led to some notable combinatorial effects when changing the names of Entities, Attributes, and Namespaces or naming conventions of certain pre- and postfixes. These combinatorial effects are attributed to the lack of support for the Data Version Transparency principle in the CA principles.

5.3.3. Technologies and Frameworks

We did not observe or find any combinatorial effects using frameworks and technologies that are part of the functionality. The artifacts uses several frameworks for data persistence (EntityFramework with Microsoft Azure SQL), Logging (NLog), and Template rendering engine (Scriban). Each of these technologies is implemented adhering to the LSP principle. We have found that replacing them is an anticipated change and a relatively simple task when adhering to the contracts that separate the implementation of the technology from its use.

However, we observe a combinatorial effect when requirements dictate that the programming language is replaced, for example, using Java instead of C#. When the requirement only applies to the generated artifact, a new expander should be created, impacting the uses of frameworks and templates. In this case, the impact of combinatorial effects is moved from the generated artifact to the expander.

5.3.4. The Craftings

Implementing the Harvesting and Injection process has led to some minor instabilities. Currently, there is a lack of support for re-injecting craftings on elements moved to a different target folder. In addition, changing the names of placeholders in the templates also leads to failures when re-injecting craftings. These combinatorial effects are attributed to the lack of support for the Data Version Transparency principle in the CA principles.

6. Conclusions and discussion

This thesis embodies a multidimensional exploration of the convergence of CA with NS. We have drawn upon the author's firsthand experience designing software architectures using rigorous theoretical research. Additionally we created practical and working software artifacts. The primary objective was to study the convergence 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 actually a part of Computer Science.

Stability and evolvability are concepts not directly referenced in the literature on CA, but this design approach aligns with the goal of Mannaert et al. (2016, p. 31). As depicted in Table 5.6, the attentive reader surely observes the shared emphasis on modularity and the separation of concerns, as all SOLID principles have a strong convergence with SoC. Both approaches attempt to achieve low coupling and high cohesion. In addition, CA adds the dimensions of dependency management as useful measures to improve maintainability and manage dependencies in a modular architecture.

The Transparency of Data versions appears to be underrepresented in the SOLID principles of CA. DvT is primarily supported by the SRP of CA, as evidenced by the presence of ViewModels, RequestModels, ResponseModels, and Entities in the artifact. It is worth noting that this separation of concerns on an ontological level is an integral part of the design elements of CA. While CA does address DvT through the SRP, a more comprehensive representation of the underlying idea of DvT within the principles of CA will likely improve the convergence of CA with NS, potentially improving the stability and evolvability of software Systems based on CA.

As described in section 5.3, the underrepresentation of DvT has led to significant combinatorial effects in some parts of the artifacts. These combinatorial effects are also be attributed to the author's inexperience in creating systems that enable code generation through expansion while maintaining stability on templates and craftings. When Data Version Transparency was better represented in the principles of CA, the severity of the combinatorial effects would have most likely been less.

As indicated in Table 5.16, CA lacks a strong foundation for receiving external triggers in its design philosophy. This is partially represented by the Controller element. However, this element tends to be used for web-enabled environments like websites and Restful APIs. This may result in a less comprehensive approach to receiving external triggers across various technologies or systems.

The most notable difference between CA and NS is 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.

The findings can only lead to the conclusion that the convergence 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

In this research, the convergence between CA and NS has been thoroughly investigated. While it has been demonstrated that the convergence between these two approaches is incomplete, combining both methodologies is highly beneficial for both NS and CA for various reasons. The primary advantage of this convergence lies in the complementary nature of CA with NS, where each approach provides strengths that can be leveraged to address a strong architectural design.

Clean Architecture offers a well-defined, practical, and modular structure for software development. Its principles, such as SOLID, guide developers in creating maintainable, testable, and scalable systems. This architectural design approach is highly suitable for various applications and can be easily integrated with the theoretical foundations provided by NS.

Conversely, the NS approach offers a more comprehensive theoretical understanding of achieving stable and evolvable systems. Furthermore, the popularity and widespread adoption of Clean Architecture in the software development community can benefit Normalized Systems. As more developers already adopting Clean Architecture become more familiar with Normalized Systems and recognize their value to software design. Combining both approaches will likely lead to increased adoption of Normalized Systems.

6.3. Limitations

In this research, the artifacts created to demonstrate the convergence between CA and NS have shown promising results. However, it is essential to recognize these artifacts' limitations, particularly in implementing NS principles such as the Separation of State Principle and the Trigger Element. These limitations must be acknowledged to guide future work and refinement of the combined architectural design approaches.

One of the primary limitations of the artifacts lies in their incomplete representation of the Separation of State principle. This principle is crucial in NS to ensure proper handling of state changes while achieving stability and evolvability. While the artifacts incorporate some aspects of state management, they fall short of fully implementing the Separation of State principle as NS prescribes.

The other limitation of the artifacts is their lack of a comprehensive Trigger Element, an essential element of NS. The Trigger Element manages external triggers while ensuring that software remains stable and evolvable. In the artifacts, incorporating the Trigger Element is limited, primarily relying on the Controller element from CA. While this approach may be sufficient for web-enabled environments such as websites and RESTful APIs, it may not be adequate for a broader range of requirements.

6.4. Reflections

In this section, we will discuss the valuable experiences and lessons we gained while working on this research and thesis. The approach was inspired by the Enterprise Engineering (EE) master class. EE encourages using grounded methodologies and theories to comprehend the inner workings of an enterprise (Dietz & Mulder, 2020, p. 262). The Five Way Framework of Wijers et al. (1989) is often applied in EE to discuss design methodologies. A method is commonly understood as a systematic procedure for accomplishing a task—for example, methods to build artifacts, methods to study, and methods to write a thesis. Next to 'method,' there is the concept of 'methodology,' commonly understood as a system of methods used in a particular area of study or activity. The original meaning is doctrine or principles of methods, specifically concerning scientific research. In this reflection, we will adopt the first meaning while acknowledging that methodology is firmly rooted in theoretical foundations.

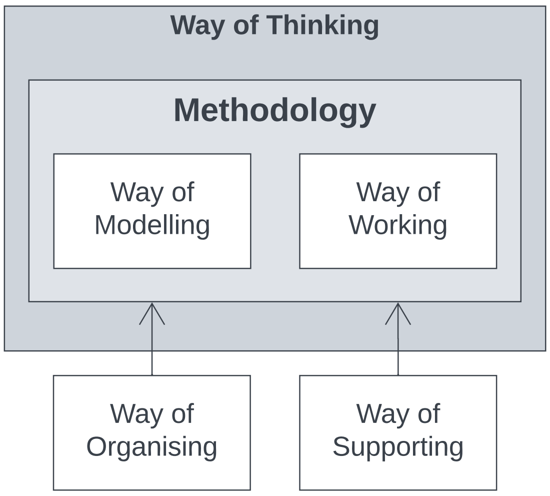


Figure 6.1.: The so-called Five Ways Framework is presented as an aid to discussing methodologies. It is an adapted version of the one that is discussed above

6.4.1. Way of Thinking

Early in my career, I became obsessed with software Quality and Maintainability. The NS theory shifted the obsession toward Stability and evolvability. Gaining a thorough understanding of the essential principles of this theory has boosted my confidence in making informed decisions regarding all aspects of architecture, not just limited to software.

As a Domain Architect, my job involved creating software products using the MDD paradigm. Initially, I was skeptical about this approach based on my early experiences. The theory of NS taught me to understand better 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 gain a clearer vision and helped me push the roadmap of the MDD framework in the right direction.

6.4.2. Way of Modeling

To explain the implementation concepts of the artifact, I looked into various modeling languages. Archimate was the first option I considered. However, during one of the EE masterclasses, I learned the hard way that Archimate is not always the best choice for communicating your models to a broad audience. I thought about using basic "boxes and arrows" but decided to use the UML2 standard because it is a formal modeling language.

6.4.3. Way of Working

I very much enjoyed designing and creating the C# artifacts. In hindsight, I enjoyed it so much that I put in way too much effort than was needed. I was very curious about the aspects of code generation, the effect of code generation on stable and evolvable artifacts, and meta-circularity characteristics. I am confident I could have arrived at the same conclusions presented in this thesis using a manually built Restful C# artifact. However, the insights I gained on the subjects of code expansion are of invaluable value to me. Therefore, I am very pleased and satisfied that I took the effort to build the Code Expander as a primary artifact.

The use of a testing framework like ArchUnitNET (2023a) could have benefitted the development process in testing the adherence to the artifact requirements described in chapter 3.2

The NS theorems are formulated clearly and abstractly, making them applicable outside the software engineering field. During the masterclasses, we learned about the application of NS in the domains of Firewalls, Document management systems, and Evolvable Business Processes. I also experienced benefits in structuring and maintaining my thesis document using VSCode and Latex by applying the principle of SoC in managing the various chapters and sections.

6.4.4. Way of Organizing

I was one of the first with a research topic and started working on my artifact in the first month when starting this journey. The artifact was almost complete before the first-year summer break. My promoter and I considered submitting the thesis in August/October 2022, but we ultimately decided to prioritize improving the quality of the research and thesis.

The review process seemed difficult and sometimes even problematic for a couple of reasons. Next time I will ensure having the proper tools and agree on procedures to improve reviews from multiple proofreaders. Secondly, I noticed that having multiple proofreaders sometimes steers in opposite directions. This sometimes affected my ability to make decisions and negatively affected my confidence. Having a joined review document, where all proofreaders can leave comments, will significantly improve this experience for me and my proofreaders. Then there is the personal aspect of sometimes taking things too personally, grounded in a lack of self-confidence. However, this experience improved my self-confidence.

6.4.5. Way of Supporting

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 Expansion as a primary artifact.

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 rejected the idea of relying on online tooling for writing my thesis. 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, being able to use all the other benefits that come with VSCode. In my next project, I will likely use the VSCode Latex editor in my next project again.

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.

Mannaert, H., Verelst, J., & Ven, K. (2012). Towards evolvable software architectures based on systems theoretic stability. *Software: Practice and Experience*, *42*(1), 89–116. https://doi.org/10.1002/spe.1051

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, DL. (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). *ArchUnitNET*. https://github.com/TNG/ArchUnitNET

Github. (2023b). *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. (2023c). *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. C. (2023a). *AbstractExpander*. GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Expanders/AbstractExpander.cs

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

Koks, G. C. (2023c). *CreateEntityBoundary*. Retrieved May 7, 2023, from https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/PanthaRhei.Output/Output/6c6984a1-c87a-429b-b91f-2a976adb3c0e/LiquidVisions.PanthaRhei.Generated/src/Application/Boundaries/Entities/CreateEntityBoundary.cs

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

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

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

Koks, G. C. (2023k). *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. C. (2023l). *EntityController*. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/PanthaRhei.Output/Output/6c6984a1-c87a-429b-b91f-2a976adb3c0e/LiquidVisions.PanthaRhei.Generated/src/Presentation.Api/Controllers/EntityControllers.cs

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

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

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

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

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

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

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

Koks, G. C. (2023u). *IExpanderHandlerInteractor*. GitHub. https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536c b90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators /IExpanderHandlerInteractor.cs

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

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

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

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

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

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

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

Glossary

BIBO Bounded Input Bounded Output: A concept used in Normalized Systems in combination with stability, derived from other scientific fields.

CLI Command Line Interface: A means of interacting with a computer program through text-based commands entered in a command-line interpreter.

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.

MDD Model Driven Development: An approach to software development that emphasizes the use of models and transformations to automatically generate code and other artifacts.

MSSQL Microsoft SQL Database: A relational database management system developed by Microsoft, commonly used for storing and retrieving data in software applications.

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

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, and Dependency Inversion Principle.

VSCode Visual Studio Code: A popular source code editor developed by Microsoft, known for its versatility and extensive plugin ecosystem.

Acronyms

AvT	Action Version Transparency.
CA	Clean Architecture.
CCP	The Common Closure Principle.
CRP	The Common Reuse Principle.
DIP	Dependency Inversion Principle.
DvT	Data Version Transparency.
EE	Enterprise Engineering.
ISP	Interface Segregation Principle.
LSP	Liskov Substitution Principle.
NS	Normalized Systems.
OCP	Open/Closed Principle.
REP	The Reuse/Release Equivalence Principle.
SoC	Separation Of Concerns.
SoS	Separation of State.
SRP	Single Responsibility Principle.

A. Code listings

A.1. The ExpanderPluginLoaderInteractor

```
internal class ExpanderPluginLoaderInteractor : IExpanderPluginLoaderInteractor
{
    \\...other code

    /// <inheritdoc/>
    public void LoadAllRegisteredPluginsAndBootstrap(App app)
    {
        foreach (Expander expander in app.Expanders)
        {
            string rootDirectory = Path.Combine(expandRequestModel.ExpandersFolder,
                expander.Name);
            string[] files = directoryService.GetFiles(rootDirectory, searchPattern,
                SearchOption.TopDirectoryOnly);
            if (!files.Any())
            {
                throw new InitializationException($"No plugin assembly detected in '{
                    rootDirectory}'. The plugin assembly should match the following '{
                    searchPattern}' pattern");
            }

            LoadPlugins(files)
                .ForEach(assembly => BootstrapPlugin(expander, assembly));

        }
    }

    \\...other code

    private List<Assembly> LoadPlugins(string[] assemblyFiles)
    {
        List<Assembly> plugins = new();

        foreach (string assemblyFile in assemblyFiles)
        {
            try
            {
                Assembly assembly = LoadPlugin(assemblyFile);
                plugins.Add(assembly);
            }
            catch (Exception innerException)
            {
                throw new InitializationException($"Failed to load plugin '{
                    assemblyFile}'.", innerException);
            }
        }

        return plugins;
    }

    private Assembly LoadPlugin(string assemblyFile)
    {
        Assembly assembly = assemblyContext.Load(assemblyFile);
        logger.Trace($"Plugin context {assemblyFile} has been successfully loaded...");
        return assembly;
    }

    private void BootstrapPlugin(Expander expander, Assembly assembly)
    {
        Type bootstrapperType = assembly.GetExportedTypes()
            .Where(x => x.IsClass && !x.IsAbstract)
            .Single(x => x.GetInterfaces()
            .Contains(typeof(IExpanderDependencyManagerInteractor)));

        IExpanderDependencyManagerInteractor expanderDependencyManager = (
            IExpanderDependencyManagerInteractor)activator
            .CreateInstance(bootstrapperType, expander, dependencyManager);

        expanderDependencyManager.Register();
    }
}
```

Listing A.1: The ExpanderPluginLoaderInteractor.

A.2. The CodeGeneratorInteractor

```
/// <summary>
/// Implements the contract <seealso cref="ICodeGeneratorInteractor"/>.
/// </summary>
internal sealed class CodeGeneratorInteractor : ICodeGeneratorInteractor
{
    // ... other code

    /// <inheritdoc/>
    public void Execute()
    {
        foreach (IExpanderInteractor expander in expanders
            .OrderBy(x => x.Model.Order))
        {
            expander.Harvest();

            Clean();

            expander.PreProcess();
            expander.Expand();
            expander.Rejuvenate();
            expander.PostProcess();
        }
    }

    // ... other code
}
```

Listing A.2: The CodeGeneratorInteractor.

A.3. The ExpandEntitiesHandlerInteractor

```
public class ExpandEntitiesHandlerInteractor
    : IExpanderHandlerInteractor<CleanArchitectureExpander>
{
    // ... other code

    /// <inheritdoc/>
    public void Execute()
    {
        directory.Create(entitiesFolder);

        foreach (var entity in app.Entities)
        {
            string fullSavePath = Path.Combine(
                entitiesFolder,
                $"{entity.Name}.cs"
            );

            templateService.RenderAndSave(
                pathToTemplate,
                new { entity },
                fullSavePath
            );
        }
    }
}
```

Listing A.3: The ExpandEntitiesHandlerInteractor.

A.4. An API Versioning example

```
var forecast = app.NewVersionedApi();

// GET /weatherforecast?api-version=1.0
forecast.MapGet( "/weatherforecast", () =>
         {
             return Enumerable.Range( 1, 5 ).Select( index =>
                 new WeatherForecast
                 (
                     DateTime.Now.AddDays( index ),
                     Random.Shared.Next( -20, 55 ),
                     summaries[Random.Shared.Next( summaries.Length )]
                 ) );
         } )
        .HasApiVersion( 1.0 );

// GET /weatherforecast?api-version=2.0
var v2 = forecast.MapGroup( "/weatherforecast" )
                 .HasApiVersion( 2.0 );
```

Listing A.4: An API Versioning example

A.5. The Logger

```
internal class Logger : ILogger
{
    /// <summary>
    /// Writes the message at the trace level.
    /// </summary>
    /// <param name="message">The message that needs to be logged.</param>
    public void Trace(string message)
    {
        Trace(message, null);
    }

    /// <summary>
    /// Writes the diagnostic message at the Trace level using the specified
        expandRequestModel.
    /// </summary>
    /// <param name="message">A string containing format items.</param>
    /// <param name="args">Arguments to format.</param>
    public void Trace(string message, params object[] args)
    {
        internalLogger.Trace(message, args);
    }
}
```

Listing A.5: The Logger.

A.6. Implementations of the ICreateGateway

```
internal class HarvestRepository : ICreateGateway<Harvest>
{
    // other code

    public bool Create(Harvest entity)
    {
        if(string.IsNullOrEmpty(entity.HarvestType))
        {
            throw new InvalidProgramException("Expected harvest type.");
        }

        string fullPath = Path.Combine(
            expandRequestModel.HarvestFolder,
            app.FullName,
            $"{file.GetFileNameWithoutExtension(entity.Path)}.{entity.HarvestType}");

        serializer.Serialize(entity, fullPath);

        return true;
    }

    // other code..
}

internal class GenericRepository<TEntity> : ICreateGateway<TEntity>
{
   // other code ...

    public bool Create(TEntity entity)
    {
        context.Set<TEntity>().Add(entity);
        context.Entry(entity).State = EntityState.Added;

        return context.SaveChanges() >= 0;
    }

    // other code ...
}
```

Listing A.6: Implementations of the ICreateGateway (Koks, 2023q) in the GenericRepository (Koks, 2023o) and the HarvestRepository (Koks, 2023p).

A.7. The Gateways for Create, Read, Update and Delete

```
// Create
public interface ICreateGateway<in TEntity>
```

```
    where TEntity : class
{
    bool Create(TEntity entity);
}

// Read
public interface IGetGateway<out TEntity>
    where TEntity : class
{
    IEnumerable<TEntity> GetAll();

    TEntity GetById(object id);
}

// Update
public interface IUpdateGateway<in TEntity>
    where TEntity : class
{
    bool Update(TEntity entity);
}

// Delete
public interface IDeleteGateway<in TEntity>
    where TEntity : class
{
    bool Delete(TEntity entity);

    bool DeleteAll();

    bool DeleteById(object id);
}

internal class AppSeederInteractor : IEntitySeederInteractor<App>
{
    private readonly ICreateGateway<App> createGateway;
    private readonly IDeleteGateway<App> deleteGateway;
    private readonly Parameters parameters;

    public AppSeederInteractor(IDependencyFactoryInteractor dependencyFactory)
    {
        createGateway = dependencyFactory.Get<ICreateGateway<App>>();
        deleteGateway = dependencyFactory.Get<IDeleteGateway<App>>();
        parameters = dependencyFactory.Get<Parameters>();
    }

    public int SeedOrder => 1;

    public int ResetOrder => 1;

    public void Seed(App app)
    {
        app.Id = parameters.AppId;
        app.Name = "PanthaRhei.Generated";
        app.FullName = "LiquidVisions.PanthaRhei.Generated";

        createGateway.Create(app);
    }

    public void Reset() => deleteGateway.DeleteAll();
}
```

Listing A.7: The Gateways for Create, Read, Update and Delete.

A.8. The DependencyInjectionExtension

```
public static class DependencyInjectionExtension
{
    /// <summary>
    /// Adds the dependencies of the project to the dependency inversion object.
    /// </summary>
    /// <param name="services"><seealso cref="IServiceCollection"/></param>
    /// <returns>An instance of <seealso cref="IServiceCollection"/>.</returns>
    public static IServiceCollection AddApplicationLayer(this IServiceCollection
        services)
    {
        return services.AddTransient<ICodeGeneratorBuilderInteractor,
            CodeGeneratorBuilderInteractor>()
            .AddTransient<IEntitiesToSeedGateway, EntitiesToSeedGateway>()
            .AddTransient<ICodeGeneratorInteractor, CodeGeneratorInteractor>()
            .AddInitializers()
            .AddSeedersInteractors()
            .AddBoundaries()
            .AddTemplateInteractors();
    }

    private static IServiceCollection AddTemplateInteractors(this IServiceCollection
        services)
    {
        services.AddTransient<ITemplateInteractor, ScribanTemplateInteractor>()
            .AddTransient<ITemplateLoaderInteractor, TemplateLoaderInteractor>();

        return services;
    }

    private static IServiceCollection AddInitializers(this IServiceCollection services)
    {
        return services.AddTransient<IExpanderPluginLoaderInteractor,
            ExpanderPluginLoaderInteractor>()
            .AddTransient<IAssemblyContextInteractor, AssemblyContextInteractor>()
            .AddTransient<IAssemblyContextInteractor, AssemblyContextInteractor>()
            .AddTransient<IExpanderPluginLoaderInteractor,
                ExpanderPluginLoaderInteractor>()
            .AddTransient<IObjectActivatorInteractor, ObjectActivatorInteractor>();
    }

    private static IServiceCollection AddBoundaries(this IServiceCollection services)
    {
        return services.AddTransient<IExpandBoundary, ExpandBoundary>()
            .AddTransient<ISeederInteractor, SeederInteractor>();
    }

    private static IServiceCollection AddSeedersInteractors(this IServiceCollection
        services)
    {
        services.AddTransient<IEntitySeederInteractor<App>, AppSeederInteractor>()
            .AddTransient<IEntitySeederInteractor<App>, ExpanderSeederInteractor>()
            .AddTransient<IEntitySeederInteractor<App>, EntitySeederInteractor>()
            //.AddTransient<ISeederInteractor<App>, PackageSeederInteractor>()
            .AddTransient<IEntitySeederInteractor<App>, FieldSeederInteractor>()
            .AddTransient<IEntitySeederInteractor<App>, ComponentSeederInteractor>()
            .AddTransient<IEntitySeederInteractor<App>,
                ConnectionStringsSeederInteractor>()
            .AddTransient<IEntitySeederInteractor<App>, RelationshipSeederInteractor>()
                ;

        return services;
    }
}
```

Listing A.8: Bootstrapping dependencies in DependencyInjectionExtension (Koks, 2023i).

A.9. The DependencyManagerInteractor

```
/// <summary>
/// The <see cref="IServiceCollection">dependency container</see>.
/// </summary>
internal class DependencyManagerInteractor : IDependencyFactoryInteractor,
    IDependencyManagerInteractor
{
    private readonly IServiceCollection serviceCollection;
    private IServiceProvider provider;

    /// <summary>
    /// Initializes a new instance of the <see cref="DependencyManagerInteractor"/>
        class.
    /// </summary>
    /// <param name="serviceCollection">The <see cref="IServiceCollection"/>.</param>
    public DependencyManagerInteractor(IServiceCollection serviceCollection)
    {
        this.serviceCollection = serviceCollection;
    }

    /// <inheritdoc/>
    public void AddTransient(Type serviceType, Type implementationType)
    {
        serviceCollection.AddTransient(serviceType, implementationType);
    }

    /// <inheritdoc/>
    public IDependencyFactoryInteractor Build()
    {
        provider = serviceCollection.BuildServiceProvider();

        return this;
    }

    /// <inheritdoc/>
    public IEnumerable<T> GetAll<T>()
    {
        if (provider == null)
        {
            Build();
        }

        return provider.GetServices<T>();
    }

    /// <inheritdoc/>
    public T Get<T>()
    {
        if (provider == null)
        {
            Build();
        }

        return provider.GetRequiredService<T>();
    }

    /// <inheritdoc/>
    public void AddSingleton<T>(T singletonObject)
        where T : class => serviceCollection.AddSingleton(singletonObject);

    /// <inheritdoc/>
    public void AddSingleton(Type serviceType, Type implementationType)
    {
        serviceCollection.AddSingleton(serviceType, implementationType);
    }
}
```

Listing A.9: The DependencyManagerInteractor (Koks, 2023j) as an abstraction on external technology dependencies.

A.10. Resolving Dependencies

```
public abstract class AbstractExpander<TExpander> : IExpanderInteractor
    where TExpander : class, IExpanderInteractor
{
    private readonly ILogger logger;
    private readonly IDependencyFactoryInteractor dependencyFactory;
    private readonly App model;

    /// <summary>
    /// Initializes a new instance of the <see cref="AbstractExpander{TExpander}"/>
        class.
    /// </summary>
    /// <param name="dependencyFactory"><seealso cref="IDependencyFactoryInteractor
        "/></param>
    protected AbstractExpander(IDependencyFactoryInteractor dependencyFactory)
    {
        this.dependencyFactory = dependencyFactory;

        logger = this.dependencyFactory.Get<ILogger>();
        model = dependencyFactory.Get<App>()
            .Expanders
            .Single(x => x.Name == Name);
    }
}
```

Listing A.10: An example of resolving dependencies as part of the AbstractExpander (Koks, 2023a).

A.11. Bootstrapping Dependencies

```
// ... other code
cmd.OnExecute(() =>
{
    var provider = new ServiceCollection()
        .AddConsole()
        .AddDomainLayer()
        .AddApplicationLayer()
        .AddEntityFrameworkLayer()
        .AddInfrastructureLayer()
        .BuildServiceProvider();

    // ... other code
});

// ... other code
```

Listing A.11: Bootstrapping the dependencies of each component layer of the generator artifact.

A.12. An Entity as a Data Element

```
public class Entity
{
    public virtual Guid Id { get; set; }
    public virtual string Name { get; set; }
    public virtual string Callsite { get; set; }
    public virtual string Type { get; set; }
    public virtual string Modifier { get; set; }
    public virtual string Behaviour { get; set; }
    public virtual App App { get; set; }
    public virtual List<Field> Fields { get; set; }
    public virtual List<Field> ReferencedIn { get; set; }
    public virtual List<Relationship> Relations { get; set; }
    public virtual List<Relationship> IsForeignEntityOf { get; set; }
}
```

Listing A.12: An Entity in CA is practically the same as Data Elements in NS (Koks, 2023k).

A.13. An Interactor as a Task Element

```
internal class CreateEntityValidator : AbstractValidator<CreateEntityCommand>,
    IValidator<CreateEntityCommand>
{
    public CreateEntityValidator()
    {
        #region ns-custom-validations

        RuleFor(x => x.Name).NotEmpty();

        #endregion ns-custom-validations
    }

    public new Response Validate(CreateEntityCommand objectToValidate) =>
        base.Validate(objectToValidate)
            .ToResponse();
}
```

Listing A.13: An Interactor performing a single Task as it is intented for the task Element of NS (Koks, 2023f).

A.14. An Interactor as a Flow Element

```
internal class CreateEntityInteractor : IInteractor<CreateEntityRequestModel>
{
    //...

    public async Task<Response> ExecuteUseCase(CreateEntityRequestModel requestModel)
    {
        Response result = validator.Validate(requestModel);
        if (result.IsValid)
        {
            try
            {
                Entity entity = mapper.Map(requestModel);
                entity.Id = Guid.NewGuid();

                result.SetParameter(entity);

                int repositoryResult = await repository.Create(entity);
                if (repositoryResult != 1)
                {
                    result.AddError(ErrorCodes.InternalServerError, $"Failed to create
                        {nameof(Entity)}.");
                    return result;
                }
            }
            catch (Exception exception)
            {
                result.AddError(ErrorCodes.InternalServerError, exception.Message);
            }
        }

        return result;
    }
}
```

Listing A.14: An Interactor orchestrating multiple Tasks as it is inteded by the Flow Element of NS (Koks, 2023d).

A.15. A RequestModel as a Data Element

```
    public class DeleteEntityRequestModel : RequestModel
    {
        public Guid Id { get; set; }
    }
```

Listing A.15: The DeleteEntityRequestModel (Koks, 2023h).

A.16. A Presenter as a Task Element

```
public class CreateEntityPresenter : ICreateEntityPresenter
{
    private readonly IMapper<Entity, EntityViewModel> mapper;

    public CreateEntityPresenter(IMapper<Entity, EntityViewModel> mapper)
    {
        this.mapper = mapper;
    }

    public Response Response { get; set; }

    public IResult GetResult(HttpRequest request = null)
    {
        return Response.IsValid ?
            Results.Created($"//{mapper.Map(Response.GetParameter<Entity>()).Id}",
                mapper.Map(Response.GetParameter<Entity>())) :
            Response.ToWebApiResult(request);
    }
}
```

Listing A.16: The CreateEntityPresenter (Koks, 2023e).

A.17. A Controller as a Connector Element

```
public static class EntityController
{
    // ...

    private static WebApplication MapCreateEntity(this WebApplication app)
    {
        RouteHandlerBuilder builder =  app.MapPost(endpointTemplate, async (
            CreateEntityRequestModel model, IBoundary<CreateEntityRequestModel>
            boundary, ICreateEntityPresenter presenter, HttpRequest request) =>
        {
            await boundary.Execute(model, presenter);
            return presenter.GetResult(request);
        });

        builder.Produces(StatusCodes.Status201Created, typeof(EntityViewModel));
        builder.Produces(StatusCodes.Status500InternalServerError, typeof(
            ErrorViewModel));
        builder.Produces(StatusCodes.Status400BadRequest, typeof(ErrorViewModel));
        builder.WithTags("Entities");

        return app;
    }

    // ...
}
```

Listing A.17: The EntityController (Koks, 2023l).

A.18. A Boundary as a Connector Element

```
internal class CreateEntityBoundary : IBoundary<CreateEntityRequestModel>
{
    private readonly IInteractor<CreateEntityRequestModel> interactor;

    public CreateEntityBoundary(IInteractor<CreateEntityRequestModel> interactor)
    {
        this.interactor = interactor;
    }

    public async Task Execute(CreateEntityRequestModel requestModel, IPresenter
        presenter) =>
        presenter.Response = await interactor.ExecuteUseCase(requestModel);
}
```

Listing A.18: The CreateEntityBoundary (Koks, 2023c).

B. The Entity Relationship Diagram of the Meta-Model

B.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 B.1.: The fields of the App entity

B.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 B.2.: The fields of the Component entity

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

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

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

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

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

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

C. Designs & architecture

C.1. Component Layer Naming Conventions

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

Layer	**Project name**	**Package name**
Domain	[PROD].Domain	[COMP].[PROD].Domain
Application	[PROD].Application	[COMP].[PROD].Application
Presentation	[PROD].Presentation.[TECH]	[COMP].[PROD].Presentation.[TECH]
Infrastructure	[PROD].Infrastructure.[TECH]	[COMP].[PROD].Infrastructure.[TECH]

Table C.1.: Naming convention component layers

C.2. Element Naming Conventions

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

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

Table C.2.: Naming convention of recurring elements

C.3. UML2 Notation Legenda

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

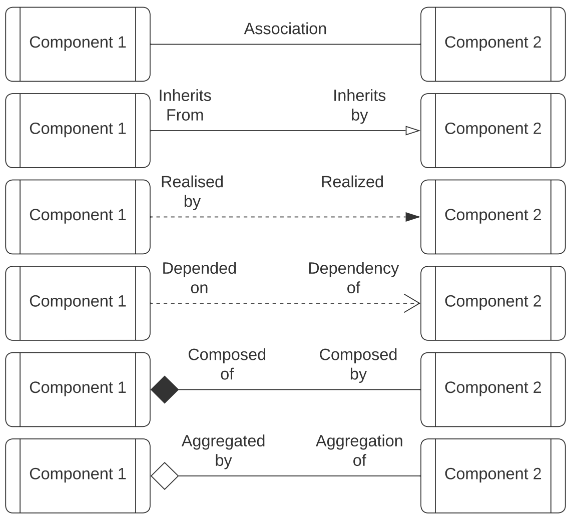


Figure C.1.: UML notation

D. Using Pantha Rhei

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

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

Listing D.1: Example command executing Pantha Rhei

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

The following RunModes are available to isolate execution tasks.

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, this mode will also initiate the processes of Harvesting and Injection. 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 D.2.: The available Generation Modes

E. Component Cohesion Principles

Name	**Description**
The Reuse/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.