Converging Clean Architecture with Normalized Systems

Gerco Koks
Antwerpen Management School, Alumini
Centric Netherlands BV, Chief Architect
Zundert, Netherlands
email:gerco.koks@outlook.com

Geert Haerens
Antwerpen Management School, Lector
Engie, Enterprise Architect
Haacht, Belgium
email:geert.haerens@engie.com

***Abstract*—This paper explores the convergence between Clean Architecture and Normalized Systems principles and design elements, highlighting their synergistic potential to enhance software design and evolvability. The paper draws upon the research described in the thesis of "On the Convergence of Clean Architecture with the Normalized Systems Theorems" from G. Koks through a comparative analysis. It demonstrates how each paradigm contributes to modular, maintainable, and evolvable software design and how integrating both approaches can lead to a more widely spread adoption and an improved software design.**

Keywords-Software; Architecture; Evolvability; Modularity; Stability.

I. INTRODUCTION

In the evolving landscape of software architecture, the software development paradigms of Clean Architecture (CA) and Normalized Systems (NS) have emerged as pivotal in addressing the multifaceted challenges of software design, particularly in managing stability, modularity, and evolvability to achieve resiliency in software. This paper delves into the synergy between these two paradigms, each contributing significantly to the contemporary discourse on software architectural complexity.

Tracing the historical underpinnings of these concepts reveals the works of pioneers like D. McIlroy [2], who championed modular programming, and Lehman [3], who underscored the importance of software evolution. Contributions from Dijkstra [4] on structured programming and Parnas [5] on software modularity further cemented the foundation for CA and NS. These historical insights contextualize the evolution of software engineering principles and underscore the relevance of fostering maintainable and evolvable software systems.

The foundation of this paper is an exploration of findings from extensive research on the convergence of CA and NS [1]. This research provides a nuanced perspective on integrating these distinct yet harmonious frameworks to enhance software design. It meticulously examines the core principles and elements of both CA and NS, presenting a scientifically robust synthesis that addresses critical challenges in software architecture.

This paper outlines the insights from G. Koks' research, exploring the significant benefits and practical implications of integrating the strengths of CA and NS within the dynamic field of software development.

The introduction is intended to set the stage and articulate the goal of this paper. Section 2 lays out the theoretical background, zooming in on the specific principles and elements of each Software Design Paradigm while also highlighting their unified concepts. In Section 3, we analyze the similarities and differences of their principles and elements and their effect on the evolvability of software constructs. The paper summarizes the conclusions in Section 4.

II. THEORETICAL BACKGROUND

This Section explores the theoretical background of both CA and NS frameworks in software engineering. It focuses on the synergetic concepts, underlying principles, and architectural building blocks of both approaches and paradigms, providing the foundation for the comparative analysis.

A. *Unified concepts*

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

1) Modularity: The original material of Martin [6, 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.* [7, 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) Cohesion: Mannaert *et al.* [7, p. 22] consider cohesion as modules that exist out of connected or interrelated parts of a hierarchical structure. On the other hand, Martin [6, p. 118] discusses cohesion in the context of components. He attributes the three component cohesion principles as crucial to grouping classes or functions into cohesive components. Cohesion is a complex and dynamic process, as the level of cohesiveness might evolve as requirements change over time.

3) Coupling: Coupling is an essential concept in software engineering that is related to the degree of interdependence among various software constructs. High coupling between components indicates the strength of their relationship, creating an interdependent relationship between them. Conversely, low coupling signifies a weaker relationship, where modifications in one part 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.* [7, p. 23] and Martin [6, p. 130] agree to achieve as much decoupling as possible.

B. Fundamentals of NS theory

Software architectures should be able to evolve as business requirements change over time. In NS theory, evolvability is measured by the lack of Combinatorial Effects. When the impact of a change depends not only on the type of the change but also on the size of the system it affects, we talk about a Combinatorial Effect. The NS theory assumes that software undergoes unlimited evolution (i.e., new and changed requirements over time, so Combinatorial Effects are very harmful to software evolvability. Indeed, suppose changes to a system depend on the size of the growing system. In that case, these changes become more challenging to handle (i.e., requiring more work and lowering the system's evolvability.

NS theory is built on classic system engineering and statistical entropy principles. In classic system engineering, a system is stable if it has BIBO – Bounded Input leading to Bounded Output. NS theory applies this idea to software design as a limited change in functionality should cause a limited change in the software. In classic system engineering, stability is measured at infinity. NS theory considers infinitely large systems that will go through infinitely many changes. A system is stable for NS, if it does not have CE, meaning that the effect of change only depends on the kind of change and not on the system size.

NS theory suggests four theorems and five extendable elements as the basis for creating evolvable software through pattern expansion of the elements. The theorems are proved formally, and they give a set of required conditions that must be followed strictly to avoid Combinatorial Effects. The NS theorems have been applied in NS elements. These elements offer a set of predefined higher-level structures, patterns, or "building blocks" that provide a clear blueprint for implementing the core functionalities of realistic information systems, following the four theorems.

1) NS Theorems: NS theory proposes four theorems, which have been proven, to dictate the necessary conditions for software to be free of Combinatorial Effects.

- Separation of Concerns
- Data Version Transparency
- Action Version Transparency
- Separation of States

Violation of any of these 4 theorems will lead to Combinatorial Effects and, thus, non-evolvable software under change.

2) NS Elements: Consistently adhering to the four NS theorems is very challenging for developers. First, following the NS theorems leads to a fine-grained software structure. Creating such a structure introduces some development overhead that may slow the development process. Secondly, the rules must be followed constantly and robotically, as a violation will introduce Combinatorial Effects. Humans are not well suited for this kind of work. Thirdly, the accidental introduction of Combinatorial Effects results in an exponential increase of rework that needs to be done.

Five expandable elements—data, action, workflow, connector, and trigger — were proposed to make the realization of NS applications more feasible. These carefully engineered patterns comply with the four NS theorems and can be used as essential building blocks for a wide variety of applications.

- **Data Element**: the structured composition of software constructs to encapsulate a data construct into an isolated module (including get- and set methods, persistency, exhibiting version transparency,etc.).
- **Action Elements**: the structured composition of software constructs to encapsulate an action construct into an isolated module.
- **Workflow Element**: the structured composition of software constructs describing the sequence in which action elements should be performed to fulfil a flow into an isolated module.
- **Connector Element**: the structured composition of software constructs into an isolated module, allowing external systems to interact with the NS system without statelessly calling components.
- **Trigger Element**: the structured composition of software constructs into an isolated module that controls the system states and checks whether any action element should be triggered accordingly.

The element provides core functionalities (data, actions, etc.) and addresses the cross-cutting concerns that each core functionality requires to function properly. As cross-cutting concerns cut through every element, they require careful implementation to avoid introducing Combinatorial Effects.

3) Element Expansion: An application is composed of a set of data, action, workflow, connector, and trigger elements that define its requirements. The NS expander is a technology that will generate code instances of high-level patterns for the specific application. The expanded code will provide generic functionalities specified in the application definition and will be a fine-grained modular structure that follows the NS theorems (see Figure 1).

The application's business logic is now manually programmed inside the expanded modules at pre-defined locations. The result is an application that implements a certain required business logic and has a fine-grained modular structure. As the code's generated structure is NS compliant, we know that the code is evolvable for all anticipated change drivers corresponding to the underlying NS elements. The only location where Combinatorial Effects can be introduced is in the customized code.

4) Harvesting and Software Rejuvenation: The expanded code has some pre-defined places where changes can be made. To prevent these changes from being lost when the application is expanded again, the expander can gather them and return them when it is re-expanded. Gathering and returning the changes is called harvesting and injection.

The application can be re-expanded for different reasons. For example, the code templates of the elements are improved

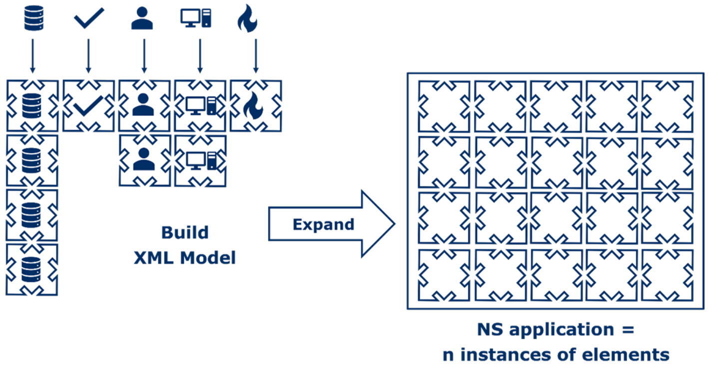

Figure 1. Requirements expressed in XML description file, used on input for element expansion.

(fix bugs, make faster, etc., include a new cross-cutting concern (add a new logging feature, or change the technology (use a new persistence framework.

Software rejuvenation aims to routinely carry out the harvesting and injection process to ensure that the constant enhancements to the element code templates are incorporated into the application.

Code expansion produces more than 80% of the code of the application. The expanded code can be called boiler-plate-code, but it is more complex than what is usually meant by that term because it deals with Cross-Cutting Concerns. Manually producing this code takes a lot of time. Using NS expansion, this time can now be spent on the constant improvement of the code templates, the development of new code templates that make the elements compatible with the latest technologies, and the meticulous coding of the business logic. The changes in the elements can be applied to all expanded applications, giving the concept of code reuse a new meaning. All developers can use a modification on a code template by one developer on all their applications with minimal impact, thanks to the rejuvenation process. Figure 2 summarizes the NS development process.

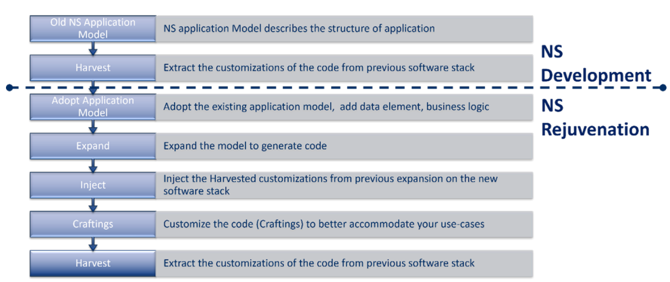

Figure 2. The NS development process.

5) Dimensions of Change: Element expansion, harvesting, rejuvenation and injection protect against CE from four change dimensions. The first dimension is the addition of new instances of data, task, flow, trigger and connector elements. These types of changes originate from new functionalities. The second dimension is the changes to the element code templates due to the introduction of new cross-cutting concerns or the overall improvement of the code of the templates. The third dimension is technology-induced changes, handled by the cross-cutting concerns and thus via the element templates. The fourth and last dimension represents the custom code, the crafting, which can be harvested and reinjected.

C. Clean Architecture

CA is a software design approach emphasizing code organization into independent, modular layers with distinct responsibilities. This approach aims to create a more flexible, maintainable, and testable software system by enforcing the separation of concerns and minimizing dependencies between components. CA aims 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 [6].

CA organizes its components into distinct layers. This architecture promotes the separation of concerns, maintainability, testability, and adaptability. The following list briefly describes each layer [6]. By organizing code into these layers and adhering to the principles of CA, developers can create more flexible, maintainable, and testable software with well-defined boundaries and a separation of concerns.

- ***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 coordinates the data flow 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.

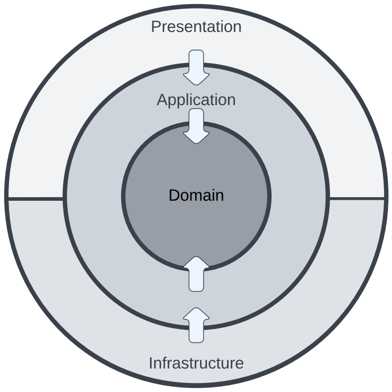

Figure 3. Flow of control

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 Dependency Inversion Principle (DIP) and can be represented schematically as concentric circles containing all the described components. The arrows in Figure 3 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.

Martin [6, p. 78] argues that software can quickly become a well-intended mess of bricks and building blocks without rigorous 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 [6, p. 79]. In 2004, the principles were established to form the acronym SOLID.

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

- ***Single Responsibility Principle (SRP)***: This principle has undergone 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" Martin [6, 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 Martin [6, 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. Martin [6, p. 81] acknowledges the slightly inappropriate name of the principle, as many interpreted it, that a module should do just one thing.
- ***Open/Closed Principle (OCP)***: Meyer [8] first mentioned the 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. [6, p. 94].
- ***Liskov Substitution Principle (LSP)***: The 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)
- ***Interface Segregation Principle (ISP)***: The 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 [6, p. 104].
- ***DIP***: The 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 defining a contract for how they should interact [6, 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 of the system over time, as changes can be made to specific modules without affecting the rest of the system.

Martin [6] proposes the following elements to achieve the goal of "Clean Architecture."

- ***Entities***: 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.
- ***RequestModels***: RequestModels represent the input data required by a specific interactor.
- ***ResponseModel***: ResponseModel represents the output data required by a specific interactor.
- ***ViewModels***: ViewModels are responsible for managing the data and behavior of the user interface.
- ***Controllers***: Controllers are responsible for handling requests from the user interface and routing them to the appropriate Interactor.
- ***Presenters***: Presenters are responsible for formatting and the data for the user interface.
- ***Gateways***: 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.

III. Requirements

This chapter outlines the requirements for this Design Science Research study, where we focus on the stability and evolvability of software artifacts. Section III-A begins by discussing software Transformation Requirements proposed by Mannaert *et al.* [7], which serve as a foundation for assessing the stability & evolvability of the artifacts. Next, Section III-B 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.

A. Software Transformation Requirements

In order to study stability and evolvability within cellular automata (CA) artifacts, we will apply certain principles from the Functional-Construction software transformation, as outlined by Mannaert *et al.* [7, p. 251]. The Functional Requirements Specifications, as proposed by Mannaert *et al.* [7, pp. 254–261], offer a structured framework that can be applied during the implementation phase to investigate combinatorial effects in CA artifacts.

First, an information system must be capable of representing instances of data entities. These data entities are composed of multiple data fields, which may either represent a basic value or serve as a reference to another data entity. This capability ensures that the system can efficiently store and manage complex data relationships.

Second, the system must support the execution of processing actions on instances of these data entities. A processing action typically consists of a series of tasks, which may either be basic units of processing, capable of independent change, or calls to other processing actions. This ensures flexibility in how processes are managed and executed within the system.

Third, the information system should facilitate input and output of data entity instances through defined connectors. This requirement emphasizes the need for seamless integration and interaction with external systems or components, allowing for data exchange in a structured and efficient manner.

Furthermore, it is essential for an existing information system to accommodate updates to data entities. Specifically, the system should be able to represent new versions of data entities that include additional fields or even entirely new entities. This allows for adaptability as the system evolves over time.

Finally, the system must be able to update its processing actions. This includes providing new versions of processing tasks or actions, which may be mandatory for the system to use, as well as the ability to introduce additional tasks or actions. This ensures that the system can expand and adjust its processing capabilities in response to new requirements or changes in the environment.

By adhering to these Functional Requirements Specifications, the system gains both the stability to handle existing operations and the evolvability to adapt to future changes and requirements.

B. Artifact Requirements

Chapter (((FIX REF))) 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.

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.*

TABLE I
NAMING CONVENTION COMPONENT LAYERS

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

IV. 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.*

TABLE II
NAMING CONVENTION OF RECURRING ELEMENTS

Layer	Element	Type	Convention
Presentation	Controller	class	[*Noun*]Controller
	ViewModel-Mapper	class	[*Noun*]ViewModel-Mapper
	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
	RequestModel-Mapper	class	[*Verb*][*Noun*]Request-ModelMapper
	Presenter	interface	IPresenter
	Validator	interface	IValidator
	Validator	class	[*Verb*][*Noun*]Validator
Infrastructure	Gateway	class	[*Noun*]Repository
Domain	Data Entity	class	[*Noun*]

1) Component Architecture Requirements: The following requirements apply to the component architecture of both the Generator artifact and the Generated artifact.

The component architecture is organized into separate Visual Studio projects for the Domain, Application, Infrastructure, and Presentation layers. A detailed description of these layers can be found in Section (((FIX FULLREF))). Each of these projects adheres to the naming conventions described in Appendix (((FIX FULLREF))). Importantly, the dependencies between component layers must follow an inward direction, aligning with higher-level components as schematically illustrated in Figure 3. The dependencies cannot skip layers, ensuring a clear hierarchical structure.

In terms of technology, the Domain and Application layers are designed to be independent of any infrastructure technologies, such as web or database technologies. In contrast, the Presentation Layer relies on various infrastructure technologies to facilitate interaction with end-users. These technologies include Command Line Interfaces (CLIs), RESTful APIs, and web-based solutions. Each dependency within the Presentation Layer is isolated and managed in separate Visual Studio projects to ensure the system's stability and evolvability.

The Infrastructure Layer may rely on additional components, such as databases or filesystems, but similar to the Presentation Layer, each infrastructure dependency is isolated and managed in its own Visual Studio project to maintain system stability and evolvability. All layers within the component architecture utilize the C# programming language, explicitly targeting the .NET 7.0 framework.

Furthermore, the reuse of existing functionality or technology, such as packages, is permitted only when it complies with the Liskov Substitution Principle (LSP) and makes use of the NuGet open-source package manager. This ensures that any reused components align with the overall design principles and maintain the flexibility and integrity of the system.

By adhering to these requirements, the component architecture remains well-structured, maintainable, and capable of evolving over time.

2) Software Architecture Requirements: Figure 4 illustrates the generic software architecture of the artifacts. Each instantiated element adheres to the Element Naming Convention outlined in Appendix (((FIX APPENDIX REF))). The following sections detail the requirements specific to each element.

The ViewModel consists of data attributes representing fields from the corresponding Entity and may also contain information specific to the user interface. It is important to note that the ViewModel has no external dependencies on other objects within the architecture.

The Presenter is derived from the IPresenter interface and adheres to the specified implementation, which is located in the Application layer. Its main responsibility is to create the Controller's Response by instantiating the ViewModel, constructing the HTTP Response message, or combining both as necessary. When needed, the Presenter utilizes the IMapper interface without depending on specific implementations of IMapper. The Presenter has an internal scope and cannot be instantiated outside the Presentation layer.

The ViewModelMapper, derived from the IMapper interface, follows the specified implementation found in the Application layer. Its primary role is to map the necessary data attributes from the ResponseModel to the ViewModel. The ViewModelMapper also has an internal scope, ensuring it cannot be instantiated outside the Presentation layer.

The Controller is responsible for receiving external requests and forwarding them to the appropriate Boundary within the Application layer. It relies on the IBoundary interface without depending on specific implementations of this interface.

The IBoundary interface establishes the contract for its derived Boundary implementations, and it has public scope within the system. Boundary implementations, derived from the IBoundary interface, ensure separation between the internal aspects of the Application Layer and the other layers. Each Boundary implementation handles a single task, executed using the IInteractor interface. These implementations also have an internal scope and cannot be instantiated outside the Application layer.

The IInteractor interface defines the contract for its derived Interactor implementations. Like Boundary implementations, Interactors have an internal scope and are limited to the Application layer. Interactor implementations execute single tasks or orchestrate a series of tasks. Tasks dependent on infrastructure components, such as databases, are handled through a Gateway. Additionally, Interactor implementations utilize the IMapper interface to handle mapping between RequestModels, Entities, and ResponseModels.

The IMapper interface establishes the contract for Mapper implementations and has public scope within the system. Derived from IMapper, the RequestModelMapper is responsible for mapping the necessary data attributes from the RequestModel to an Entity. The RequestModelMapper has internal scope and cannot be instantiated outside the Application layer.

Similarly, the ResponseModelMapper is responsible for mapping data attributes from the ResponseModel and follows

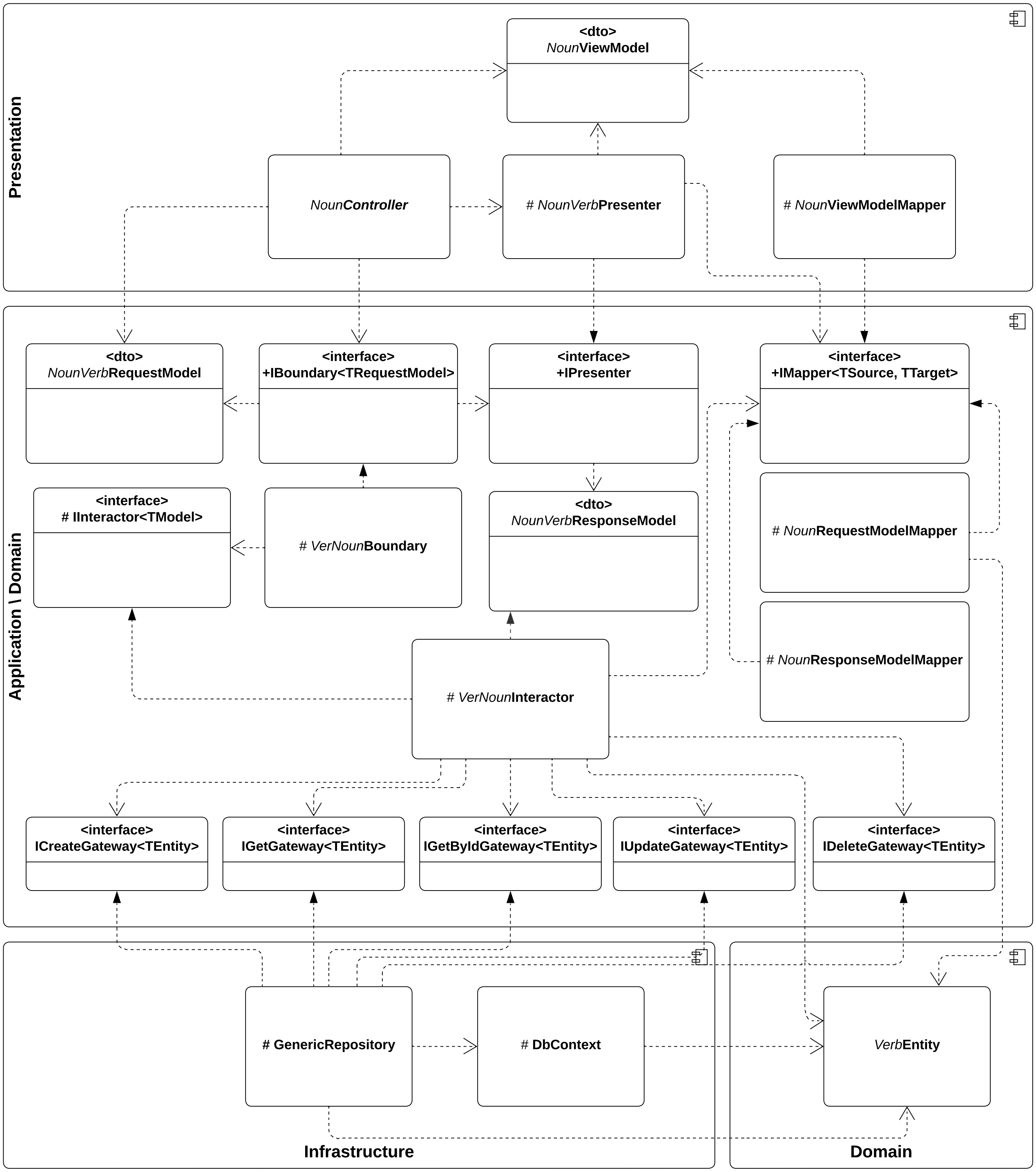

Figure 4. The Generic architecture of the artifacts

the same implementation and scope restrictions as the RequestModelMapper.

The IPresenter interface establishes the contract for Presenter implementations, typically within the Presentation layer. It has public scope and ensures consistency in Presenter behavior throughout the system.

The Gateway establishes the contract for interaction with infrastructure technologies such as databases or filesystems.

Each Gateway follows a specific naming convention, with interfaces like ICreateGateway, IGetGateway, IGetByIdGateway, IUpdateGateway, and IDeleteGateway representing different CRUD operations. Gateway implementations are derived from these interfaces and are responsible for task-specific interactions with infrastructure components. These implementations have internal scope and cannot be instantiated outside their respective layers.

The ResponseModel consists of data attributes representing fields from the corresponding Entity and may include output-specific data for the Interactor. The ResponseModel does not depend on external objects within the architecture.

The RequestModel is similarly structured, consisting of data attributes from the corresponding Entity and input-specific data for the Interactor. It, too, does not depend on external objects within the architecture.

Data Entities represent corresponding data fields and do not rely on external objects. They are only utilized by the Application layer.

The Gateway Implementation derives from the corresponding Gateway interface and adheres to the specified implementation. It is responsible for handling tasks associated with its infrastructure technology, such as interaction with a SQL database or filesystem. Gateway Implementations have internal scope and cannot be instantiated outside their respective layers.

Lastly, each architectural pattern adheres to at least one of the SOLID principles, ensuring compliance and avoiding violations of these design principles.

3) Expander Framework & Clean Architecture Expander Requirements: In addition to the more generic requirements outlined in previous sections, the following requirements are specific to the Clean Architecture Expander and Expander Framework artifact.

The Expander Framework facilitates interaction with the Clean Architecture Expander via a command-line interface (CLI), which is implemented in the Presentation layer of the framework. Additionally, the Expander Framework retrieves models from a Microsoft SQL Server (MSSQL) database using EntityFramework ORM technology, integrated within the Infrastructure layer. The framework also supports loading and executing configured Expanders, though in this particular research, only the Clean Architecture Expander is applied.

Moreover, the Expander Framework supports generic harvesting and injection functionalities, which can be extended or used by the Expanders in accordance with the Open-Closed Principle (OCP). This extensibility is further enhanced by the framework's support for generic template handling, also designed to be extended by the Expanders following the OCP. The framework adheres to the component and software requirements outlined in Sections IV-1 and IV-2 of this chapter.

The Clean Architecture Expander specifically generates a C# .NET 7.0 RESTful service, which provides an HTTP interface atop the Expander Framework's meta-model, enabling basic Create, Read, Update, Delete (CRUD) operations. This expander consists solely of an Application layer and reuses the Domain layer provided by the Expander Framework. Additionally, the Clean Architecture Expander adheres to the component and software requirements set forth in Sections IV-1 and IV-2 of this chapter.

By adhering to these requirements, both the Expander Framework and the Clean Architecture Expander align with the overall architecture goals while maintaining flexibility and extensibility.

4) Generated Artifact Requirements: ***1. The generated artifact requirement***

1.1 The generated artifact adheres to this chapter's component and software requirements specified in Sections IV-1 and IV-2.

V. The Analysis

This Section delves into the convergence of CA and NS, exploring their convergence and application in software design. The discussion is anchored in the results of the research "On the Convergence of Clean Architecture with the Normalized Systems Theorems" [1], which meticulously examines the principles and design elements of both CA and NS mentioned in previous chapters. By aligning the theoretical constructs of both paradigms, the thesis provides a perspective on achieving modular, evolvable, and stable software architectures. This convergence reinforces the robustness of software systems and enhances their evolvability and longevity in the face of future requirements. The subsequent sections will summarize the key components of their convergence by highlighting the practical implications and the potential for evolvable software design.

A. The converging principles

The main goal of both the SRP and Separation Of Concerns (SoC) is to promote and encourage modularity, low coupling, and high cohesion. While their definitions have minor nuances, the two principles are practically interchangeable. Even though SRP does not implicitly guarantee Data Version Transparency (DvT) or Action Version Transparency (AvT), it supports those theorems by directing design choices in a certain way. One example lies in separating data models for requests, responses, and views and respective versions of these models.

The OCP and its relation to NS theory emphasize the importance of designing software entities that are open for extension but closed for modification. This principle aligns with the NS approach to evolvability, advocating for structures that can adapt to new requirements without altering existing code, thus minimizing the impact of changes. An example of this synergy can be seen in the use of expanders within NS, which allow for introducing new functionality or data elements without disrupting the core architecture, cohesively supporting the OCP principle goal of extendibility and maintainability.

The LSP emphasizes that objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program. This principle strongly aligns with the emphasis on modular and replaceable components in NS, advocating for flexibility and the seamless integration of new functionalities. Applying this principle within NS is evident in designing tailored interfaces specific to a particular version. This ensures system evolution without compromising existing functionality, thereby upholding the LSP directive for substitutability and system integrity.

The ISP advocates for creating specific consumer interfaces rather than one general-purpose interface, aligning with NS principles to enhance system evolvability and maintainability. This alignment is evident in the modular and decoupled design strategies advocated by both NS and ISP, where the focus is on minimizing unnecessary dependencies and promoting high cohesion within systems. By applying ISP, developers can ensure that system components only depend on the interfaces they use, which mirrors the approach in NS to create evolvable systems by reducing the impact of changes across modules.

The DIP and its alignment with NS are centered on inverting the conventional dependency structure to reduce rigidity and fragility in software systems. DIP promotes high-level module independence from low-level modules by introducing abstractions that both can depend on, thereby facilitating a more modular and evolvable design. This principle mirrors the emphasis on minimizing dependencies to enhance system evolvability in the NS paradigm. Examples from the thesis demonstrate how leveraging DIP in conjunction with NS principles leads to systems that are more adaptable to change, showcasing the practical application of these combined approaches in achieving resilient software architectures. Designers should also be aware of the potential pitfalls of using DIP as faulty implementations can increase combinatorial effects.

In the following table, we summarize the analysis in a tabular overview using the following denotation:

- ***Strong convergence*** (**++**): This indicates that the principles of CA and NS are highly converged. Both have a similar impact on the design and implementation.
- ***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.
- ***Weak or no convergence*** (**−**): The principles have no significant similarities in terms of their purpose, goals, or architectural supports.

TABLE III
THE CONVERGENCE BETWEEN CA AND NS PRINCIPLES.

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	++	−	+	−

B. The converging elements

The Data Element from NS and the Entity Element from CA represent data objects of the ontology or data schema, typically including 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 types of Data Elements are explicitly specified as ViewModels, RequestModels, or Response Models.

The Interactor element of CA and the Task and WorkFlow elements of NS are all responsible for encapsulating business rules. NS has a more strict approach to encapsulating the execution of business rules in Task Elements, as it is only allowed to have a single execution of a business rule. Additionally, the WorkFlow element is responsible for executing multiple tasks statefully and is highly convergable with the Interactor element of CA.

The convergence of the Controller element from CA with NS is highlighted by its partial interchangeability with the Connector and Trigger elements in NS. The Controller Element is primarily responsible for interaction using protocols and technologies involving the user interface, while the Connector and Trigger elements are also intended to interact with other types of external systems.

The Gateway element of CA and the Connector element of NS communicate between components by providing Data Version Transparent interfaces to provide Action Version Transparency between these components.

The Presenter is responsible for preparing the ViewModel on the controller's behalf and can be considered a Task or Workflow Element in the theories of NS.

The Boundary element of CA strongly converges 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 more specific, as this element usually separates architectural boundaries within the application or component.

In the following table, we summarize the analysis in a tabular overview using the same denotation used in Section V-A.

TABLE IV
THE CONVERGENCE BETWEEN CA AND NS ELEMENTS.

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	–	–	–	++	–

VI. CONCLUSION

The primary objective of G. Koks was to study the convergence between CA and NS by analyzing their principles and design elements through theory and practice. This Section will summarize the findings into a research conclusion.

Stability and evolvability are concepts not directly referenced in the literature on CA, but this design approach aligns with the goal of NS. The attentive reader can observe the shared emphasis on modularity and the separation of concerns, as all SOLID principles strongly converge 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 by rigorously managing dependencies in the Software Architecture.

The DvT appears to be underrepresented in the SOLID principles of CA. DvT is primarily supported by the SRP of CA, as evidenced by ViewModels, RequestModels, ResponseModels, and Entities as software elements. It is worth noting that this application of Data Version Transparency 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.

CA Lacks a strong foundation for receiving external triggers in its design philosophy. This is partially represented by the Controller element. However, this element is described as being used for web-enabled environments and might 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. NS Provides the principle of Separation of State (SoS), 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. Consequently, CA cannot fully ensure stable and evolvable software artifacts as NS has defined them.

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

CA 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.

To conclude, the popularity and widespread adoption of CA in the software development community can benefit NS. As more developers adopt CA, they become more familiar with NS and recognize their value to software design. Synergizing both approaches will likely lead to increased adoption of NS.

BIBLIOGRAPHY

[1] G. Koks, "On the Convergence of Clean Architecture with the Normalized Systems Theorems," en, Ph.D. dissertation, 2023-06. [Online]. Available: https://zenodo.org/record/8029971 (visited on 2024-03-24).

[2] D. McIlroy, "NATO Software Engineering Conference," en, 1968.

[3] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980, ISSN: 0018-9219. DOI: 10.1109/PROC.1980.11805.

[4] E. Dijkstra, "Letters to the editor: Go to statement considered harmful," en, *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968-03, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/362929.362947.

[5] D. Parnas, "On the criteria to be used in decomposing systems into modules," en, *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972-12, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/361598.361623. (visited on 2023-03-19).

[6] R. C. Martin, *Clean architecture: a craftsman's guide to software structure and design* (Robert C. Martin series). London, England: Prentice Hall, 2018, OCLC: on1004983973, ISBN: 978-0-13-449416-6.

[7] H. Mannaert, J. Verelst, and P. De Bruyn, *Normalized systems theory: from foundations for evolvable software toward a general theory for evolvable design*, eng. Kermt: nsi-Press powered bei Koppa, 2016, ISBN: 978-90-77160-09-1.
[8] B. Meyer, *Object-oriented software construction*, 1st ed. Upper Saddle River, N.J: Prentice Hall PTR, 1988, ISBN: 978-0-13-629155-8.