Converging Clean Architecture with Normalized Systems

Dhr. ir. Geert Haerens
Organisation
City, Country
email:geert.haerens@uantwerpen.be

Gerco Koks
Antwerpen Management School, Alumini
Zundert, Netherlands
email:gerco.koks@outlook.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. Normalized Systems

NS in software engineering revolves around stable and evolvable information systems, drawing from System Theory and Statistical Entropy from Thermodynamics. NS is rooted in software engineering but applies to other domains, such as Enterprise Engineering [8], Hardware configurations like TCP-IP firewalls [9], and Business Process Modeling [10].

The NS theory emphasizes stability as a crucial property derived from the concept of Bounded Input leading to Bounded Output (BIBO). Stability in NS means that a bounded functional change must result in a bounded amount of work, regardless of the system's size. Instabilities, also referred to as combinatorial effects, occur when the number of changes depends on the system size, negatively impacting its evolvability.
In the following list, we will describe the design Theorems of NS, first presented by Mannaert and Verelst [11].

- ***Separation Of Concerns (SoC)***: A processing function containing only a single task to achieve stability.
- ***Data Version Transparency (DvT)***: A data structure passed through a processing function's interface must exhibit version transparency to achieve stability.
- ***Action Version Transparency (AvT)***: A processing function that is called by another processing function needs to exhibit version transparency to achieve stability.
- ***Separation of State (SoS)***: Calling a processing function within another processing function must exhibit state-keeping to achieve stability.

NS aims 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, the following normalized elements have been proposed [7, pp. 363–398]. It is essential to recognize that different programming languages may necessitate alternative constructs [7, p. 364].

The following list describes each element using the definition from Mannaert *et al.* [12, p. 102]

- ***Data Element***: Based on DvT, data elements have "get" and "set" methods for wide-sense data version transparency or marshal -and parse- methods for strict-sense DvT. Supporting tasks can be added in a way that is consistent with the principles of SoC and DvT.
- ***Task Element***: Based on SoC, the core action entity can only contain a single functional task, 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 so that a separate action entity wraps the action entities representing task versions. Supporting tasks can be added in a way that is consistent with SoC and AvT.
- ***Workflow Element***: 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 an argument.
- ***Connector Element***: 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 that consistent with SoC and AvT.
- ***Trigger Element***: 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 that is consistent with SoC and AvT.

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.

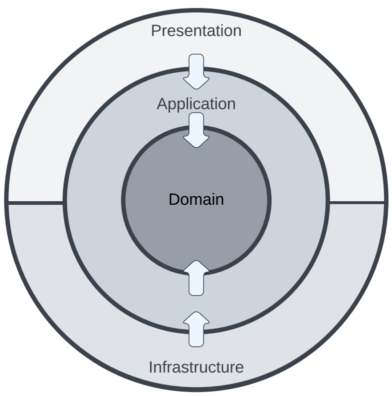

Figure 1. Flow of control

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 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.

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 [13] 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-1 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-A 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.

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.* [7, p. 251] by using the following five proposed Functional Requirements Specifications. Mannaert *et al.* [7, 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

A. 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 Architecture Requirements

The following requirements are applied to the component architecture of both the Generator artifact and the Generated artifact.

1. The component architecture requirement

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 (((FIX FULLREF))).

1.2 The Visual Studio projects representing the component layers comply with the naming conventions outlined in Appendix (((FIX FULLREF))).

1.3 The dependencies between the component layers must follow an inward direction towards the higher-level components as illustrated in Figure 1 schematically, and cannot skip layers.

2. The technology requirement

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.

2) Software Architecture Requirements

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

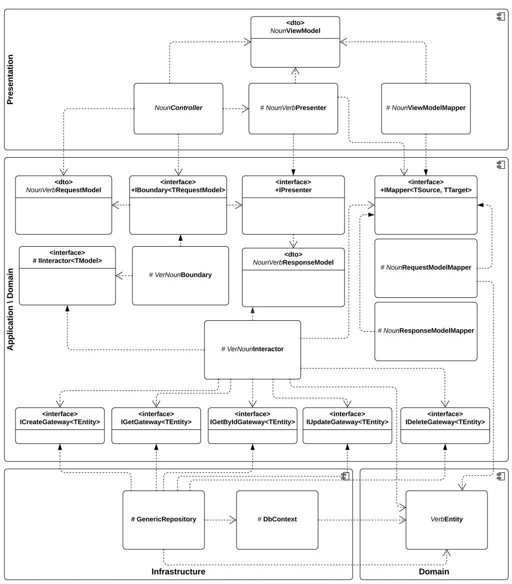

Figure 2. The Generic architecture of the artifacts

requirementThe ViewModel

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

3. The Presenter requirement

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

4. The ViewModelMapper requirement

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

5. The Controller requirement

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

6. The IBoundary requirement

6.1 The IBoundary interface establishes the contract for its derived Boundary implementations.
6.2 The IBoundary interface has public scope within the system.

7. The Boundary Implementation requirement

7.1 A Boundary implementation is derived from the IBoundary interface and follows the specified implementation.
7.2 The Boundary implementation separates the internal aspects of the Application Layer and the other layers within the component.
7.3 Each Boundary implementation handles a single task, executed using the IInteractor interface.
7.4 Boundary implementations have an internal scope and cannot be instantiated outside the Application layer.

8. The IInteractor requirement

8.1 The IInteractor interface establishes the contract for its derived Interactor implementations.
8.2 The IInteractor has an internal scope and cannot be implemented outside the Application layer.

9. The Interactor Implementation requirement

9.1 An Interactor implementation is derived from the IInteractor interface and follows the specified implementation.
9.2 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.

9.3 Depending on the task, the Interactor implementation orchestrates the mapping from RequestModels to Entities or from Entities to ResponseModels, utilizing the IMapper interface.

9.4 Interactor implementations have an internal scope and cannot be implemented outside the Application layer.

10. The IMapper requirement

10.1 The IMapper interface establishes the contract for its derived Mapper implementations.

10.2 The IMapper interface has a public scope within the system.

11. The RequestModelMapper requirement

11.1 The RequestModelMapper is derived from the IMapper interface and follows the specified implementation.

11.2 The RequestModelMapper is responsible for mapping the values of the necessary data attributes from the RequestModel to an Entity.

11.3 The RequestModelMapper has an internal scope and cannot be implemented outside the Application layer.

12. The ResponseModelMapper requirement

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.

13. The IPresenter requirement

13.1 The IPresenter interface establishes the contract for its derived Presenter implementations, typically implemented as part of the Presentation layer.

13.2 The IPresenter interface has a public scope within the system.

14. The Gateway requirement

14.1 The Domain and Application layers have no dependencies on infrastructure technologies, like web- or database technologies.

14.2 The *[Verb]Gateway* interface establishes the contract for its derived Gateway implementations, typically implemented in the Infrastructure layer.

14.3 The *[Verb]*Gateway interface has a public scope within the system.

14.4 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.

15. The ResponseModel requirement

15.1 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.

15.2 The ResponseModel does not depend on external objects within the architecture.

16. The RequestModel requirement

16.1 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.

16.2 The RequestModel does not depend on external objects within the architecture.

17. The Data Entity requirement

17.1 The Data Entity consists solely of attributes representing the corresponding data fields.

17.2 The Data Entity does not rely on external objects within the architecture.

17.3 The Application layer is the only layer that utilizes the Data Entity.

18. The Gateway Implementation requirement

18.1 The [*Verb*]Gateway Implementation derives from the I[*Verb*]Gateway interface and adheres to the specified implementation.

18.2 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).

18.3 The [*Verb*]Gateway Implementation has an internal scope and cannot be instantiated outside the layer.

19. The Design Principles requirement

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

3) Expander Framework & Clean Architecture Expander Requirements

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

20. The Expander Framework requirement

20.1 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.

20.2 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.

20.3 The Expander Framework loads and executes the configured Expanders. In the case of this research, only the Clean Architecture Expander is applied.
20.4 The Expander Framework supports generic harvesting and injection, which can be used or extended by the Expanders using the OCP principle.
20.5 The Expander Framework supports generic template handling, which can be used or extended by the Expanders using the OCP principle.
20.6 The Expander framework adheres to this chapter's component and software Requirements specified in Sections III-A1 and III-A2.

21. The Clean Architecture Expander requirement

21.1 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.
21.2 The Clean Architecture Expander consists solely of an Application layer and reuses the Domain layer of the Expander Framework.
21.3 The Clean Architecture Expander adheres to this chapter's component and software Requirements specified in Sections III-A1 and III-A2.

4) Generated Artifact Requirements

22. The generated artifact requirement

22.1 The generated artifact adheres to this chapter's component and software requirements specified in Sections III-A1 and III-A2.

IV. 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 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 DvT or 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 I
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 IV-A.

TABLE II
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	−	−	−	++	−

V. 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 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] P. Huysmans and J. Verelst, "Towards an Engineering-Based Research Approach for Enterprise Architecture: Lessons Learned from Normalized Systems Theory," en, in *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, vol. 8827, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2013, pp. 58–72, ISBN: 978-3-319-12567-1 978-3-319-12568-8. DOI: 10.1007/978-3-642-38490-5_5.

[9] G. Haerens, "On the evolvability of the TCP-IP based network firewall rule base," eng, PhD Thesis, Antwerp University, 2021. [Online]. Available: https://hdl.handle.net/10067/1834610151162165141 (visited on 2024-03-24).

[10] D. van Nuffel, *Towards designing modular and evolvable business processes*. University of Antwerp, 2011, ISBN: 90-8994-040-5.

[11] H. Mannaert and J. Verelst, *Normalized systems re-creating information technology based on laws for software evolvability*, English. Kermt: Koppa, 2009, OCLC: 1073467550, ISBN: 978-90-77160-00-8.

[12] H. Mannaert, J. Verelst, and K. Ven, "Towards evolvable software architectures based on systems theoretic stability," en, *Software: Practice and Experience*, vol. 42, no. 1, pp. 89–116, 2012-01, ISSN: 00380644. DOI: 10.1002/spe.1051.

[13] B. Meyer, *Object-oriented software construction*, 1st ed. Upper Saddle River, N.J: Prentice Hall PTR, 1988, ISBN: 978-0-13-629155-8.